Chapter 8

Applications for Empirical Worlds

8.1 Introduction

In Chapter 7, *empirical worlds* and the associated Java class library are introduced. In this chapter, some applications of *empirical worlds* are described. They include:

- the integration of a broader range of shapes than represented by primitive shape data types. This is achieved by augmenting the *empirical world* class library. An example of this process is given in Section 8.2, which involves integrating operators and data types for the representation of *skeletal implicit surfaces* [WvO97, WvO95, BBCG$^+$97] into *empirical worlds.*

- the integration of a broader range of transformations than the standard affine transformations presented in Section 7.3.6. In Section 8.3, *warping* transformations for bending, twisting and tapering shapes are illustrated.

- a server/client implementation of a tool based on the *empirical world* classes with support for interaction using *empirical world* scripts. The *empirical world* *builder* server and world-wide-web client applications are described in Section 8.4.

- the creation of an artefact for the interactive exploration of a geometric shape using an *empirical world* script. A case-study is presented in Section 8.5 that illustrates the use of some of the data types and operators represented by the *empirical world* classes.

All the support for agency (see Section 6.3.2) and client/server implementation mechanisms (see Section 6.4.2) available in the JaM Machine API can be integrated into applications that are based on the *empirical world* class library. New interfaces to *empirical worlds* can be created by constructing appropriate graphical user-interfaces, or with parsers for a higher level geometric notations than *empirical world* scripts. The potential for real-time interaction with the *empirical world builder* tool is evaluated in Section 8.4.3, where timings for the polygonisation of shapes are presented.

8.2 Skeletal Implicit Shapes in Empirical Worlds

Skeletal implcits[1] are a class of implicit function shape constructed by blending sets of skeletal elements. Any skeletal element S, represented by the implicit mapping r_S, has the property that for a point $\mathbf{p}$, $r_S(\mathbf{p})$ is the minimum distance from $\mathbf{p}$ to the surface of S. The effect of this is that the values for r_S around a skeleton are like a contour map and define an increasing field as you move away from the skeletal geometry. The skeletons described in this section are sphere, cylinder and torus and all extend a class called `SoftType`. The skeletons described here are similar to those implemented by Larcombe in [Lar94] and could easily be extended in the future

[1]Also known as *Blobby* objects.

to include cone, ellipsoid, polygon and plane by implementing more subclasses of `SoftType`.

The approach adopted here is to integrate skeletal-based implicits into a modelling system that supports function representation. To do this the `SoftType` class is used, which directly extends `TopologyType` that in turn extends `FrepType` (see Figure 7.2). The `SoftType` class implements the function representation method `f()`. (see Section 7.2). In this section, it is assumed that if point $\mathbf{p}$ is inside the skeleton, then the value of $r_S(\mathbf{p})$ will be negative, this value is zero if $\mathbf{p}$ is on the surface of the skeleton and positive external to the skeleton. Instead of implementing method `f()`, all classes that extend `SoftType` implement another method called `d()` which returns the value of $r_S(\mathbf{p})$ for a particular skeleton. In the class `SoftType`, the method `f()` is implemented and returns the negative value mapped to by abstract method `d()`. The function representation f of a soft shape is $f(\mathbf{p}) = -r_S(\mathbf{p})$.

Wyvill and Guy suggest a syntax for skeletal implicit objects embedded as part of VRML in [WG97]. This syntax has been loosely observed in the implementation of the *empirical world* classes, except where mentioned in the text. The differences occur for the classes: `SoftShape`, which takes any skeleton S and applies a field function to the minimum distance value $r_S(\mathbf{p})$, for any point $\mathbf{p}$; `SoftSum`, which sums the fields surrounding a list of geometric objects. Both `SoftShape` and `SoftSum` have been extended so that they are effective for all instances of classes that extend `FrepType` in *empirical worlds*. More details are presented in Sections 8.2.4 and 8.2.5.

An important motivation for the inclusion of this section is to demonstrate how easily new JaM data types for a shape can be incorporated into an *empirical world* notation. The skeletal element types and operators represented (including the three skeletal elements described in this chapter, the `SoftShape` type, the `SoftSum` type and all the associated operators) were all implemented during one day. Once the classes themselves were written, it is a simple task to link them into a tool based on *empirical world* classes. As all the skeletal implicit classes extend `FrepType`, all operators available for the combination and manipulation of `FrepType` objects become immediately applicable to the soft objects, including affine transformations, CSG operations, morphing and so on. Conventional CSG-like geometric objects can coexist as components of models with *blobby* objects.

8.2.1 Soft Spheres

A soft sphere can be defined explicitly in *empirical world* scripts with the data type represented by the `CadnoSoftSphere` class. The sphere is parametrised by its `radius` r. Like the `CadnoSphere`, the `CadnoSoftSphere` is centred on the origin and should be translated to another position if a different centre is required by the user. The bounding box for the sphere has minimum point $(-r, -r, -r)$ and maximum point (r, r, r).

Class Name	`CadnoSoftSphere`	
Extends	`SoftType`	

Value Format	`SoftSphere {`
	   `radius` *float*
	`}`	

Default Value	`SoftSphere {`
	   `radius 1`
	`}`	

Parameters	Name	Type
	`radius`	`CadnoFloat`

A soft sphere can be defined implicitly in an *empirical world* script with the operator represented by the `CadnoMakeSoftSphere` class. The argument to the operator in an implicit definition for a soft sphere is the `radius` r.

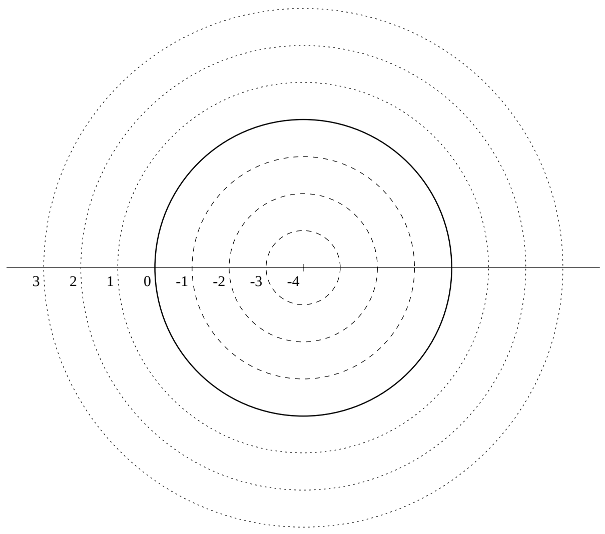


Figure 8.1: Planar slice through a `CadnoSoftSphere`, showing the surrounding field.

Operator Class		`CadnoMakeSoftSphere`
Maps From		`CadnoDetail` × *float* or *float*
Maps To		`CadnoSoftSphere`
Example	Defn.	`ss = softSphere(3.2)`
	Value	`ss = SoftSphere {` `radius 3.2` `}`

The distance d from any point (x, y, z) to the skeleton surface of the soft sphere with radius r is given by

$$d(x, y, z) = \sqrt{x^2 + y^2 + z^2} - r \tag{8.1}$$

This function is C^1 continuous everywhere except at the origin point (0, 0, 0).

Figure 8.1 shows the field around and inside a sphere of radius 4 given by d for a planar slice through the solid sphere geometry in the xy-plane. The thick line represents the surface of the sphere, the dashed lines indicate the field inside the sphere and the dotted lines indicate the field outside the sphere.

8.2.2 Soft Cylinders

Soft cylinders are parametrised in *empirical worlds* by a height h and a radius r and the geometric shape represented has hemispherical ends, simplifying the calculation of the distance field. A `CadnoSoftRCylinder` class can be used to explicitly define the value of a soft cylinder, centred on the origin with its height dimension oriented along the y-axis of the space. The parameters of a cylinder in a script are `height` and `radius`. The bounding box for soft cylinder is defined by minimum point $(-r, \frac{-h}{2} - r, -r)$ and $(r, \frac{h}{2} + r, r)$.

Class Name	`CadnoSoftRCylinder`	
Extends	`SoftType`	

Value Format	`SoftRCylinder {`
	`radius` *float*
	`height` *float*
	`}`	

Default Value	`SoftRCylinder {`
	`radius 1`
	`height 2`
	`}`	

Parameters	Name	Type
	`height`	`CadnoFloat`
	`radius`	`CadnoFloat`

Soft cylinders can be defined implicitly in *empirical world* scripts with the operator represented by the `CadnoMakeSoftRCylinder` class, which extends `DefnFunc`. The arguments to the operator in an implicit definition are a value representing the `radius` r of the soft cylinder, followed by the `height` h of the cylinder.

Operator Class		`CadnoMakeSoftRCylinder`
Maps From		`CadnoDetail` × *float* × *float* or *float* × *float*
Maps To		`CadnoSoftRCylinder`
Example	Defn.	`sc = softRCylinder(3.1, 7.9)`
	Value	`sc = SoftRCylinder {` `radius 3.1` `height 7.9` `}`

The distance d from any point (x, y, z) to the surface of the skeleton of a soft cylinder is given by

$$d(x,y,z) = \begin{cases} \sqrt{x^2 + (y - \frac{h}{2})^2 + z^2} - r & \text{if } y \geq \frac{h}{2} \\ \sqrt{x^2 + z^2} - r & \text{if } -\frac{h}{2} < y < \frac{h}{2} \\ \sqrt{x^2 + (y + \frac{h}{2})^2 + z^2} - r & \text{if } y \leq -\frac{h}{2} \end{cases} \tag{8.2}$$

This mapping is C^1 continuous everywhere except at the planes $y = -\frac{h}{2}$ and $y = \frac{h}{2}$ and along the y-axis between these planes.

Figure 8.2 shows the field inside and outside a soft cylinder as given by the mapping d for a two-dimensional slice through the solid geometry in the xy-plane.

8.2.3 Soft Tori

One way that a torus shape can be parametrised is by an inside radius ir and an outside radius or. In an *empirical world* script, a soft torus is centred on the origin and inside and outside radius are measured in the xz-plane. The inside radius ir is the parameter `insideRadius` in a script and outside radius or is the parameter `outsideRadius`. The bounding box for a torus is defined by minimum point $(-or, -\frac{or-ir}{2}, -or)$ and maximum point $(or, \frac{or-ir}{2}, or)$.

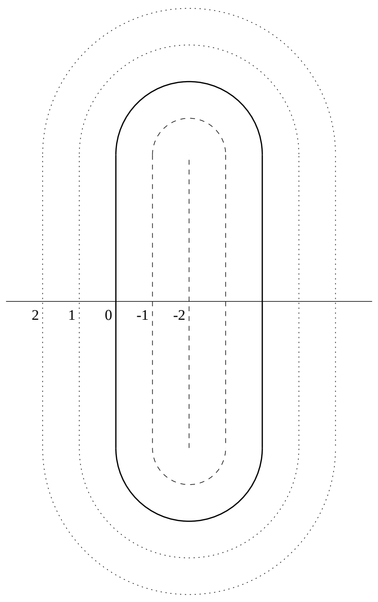


Figure 8.2: Planar slice through a `CadnoSoftRCylinder`, showing the surrounding field.

Class Name	`CadnoSoftTorus`	
Extends	`SoftType`	
Value Format	`SoftTorus` { `insideRadius` *float* `outsideRadius` *float* }	
Default Value	`SoftTorus` { `insideRadius 0.5` `outsideRadius 1.0` }	
Parameters	Name	Type
	`insideRadius`	`CadnoFloat`
	`outsideRadius`	`CadnoFloat`

Soft tori can be implicitly defined in *empirical world* scripts using the operator represented by the `CadnoMakeSoftTorus` class. The arguments to this operator in an implicit definition of a torus shape are the value of the inner radius `insideRadius` *ir*, followed by the value of`outsideRadius` *or*.

Operator Class		`CadnoMakeSoftTorus`
Maps From		`CadnoDetail` $\times$ *float* $\times$ *float* *float* $\times$ *float*
Maps To		`CadnoSoftTorus`
Example	Defn.	`st = softTorus(0.3, 0.8)`
	Value	`st = SoftTorus {` `insideRadius 0.3` `outsideRadius 0.8` `}`

The distance field function d for any point (x, y, z) for a soft torus with inner radius ir and outer radius or is

$$s(x,y,z) = \sqrt{x^2+z^2} - \left(ir + \frac{or-ir}{2}\right) \quad (8.3)$$

$$d(x,y,z) = \sqrt{s(x,y,z)^2+y^2} - \frac{or-ir}{2} \quad (8.4)$$

Every point defines a plane that embeds the point and the y-axis. The distance from the skeletal element is the function representation for a two-dimensional solid circle embedded in this same plane with its centre point located on the xz-plane at a distance of $ir + \frac{or-ir}{2}$ away from the origin. This function is C^1 discontinuous at the origin point $(0, 0, 0)$ and on the circle in the xz-plane defined by $s(x, 0, z) = 0$ and $y = 0$.

Figure 8.3 shows the distance field inside and outside a soft torus for a torus with an inner radius of 3 and an outer radius of 5. The field shown is a planar slice through the solid torus in the xz-plane.

8.2.4 Field Functions and Soft Shapes

A field function can be applied to an object of `SoftType` to modify the linearly increasing fields around soft objects, as illustrated in Figures 8.1, 8.2 and 8.3. The contribution of a component shape should only have a significant effect on the whole

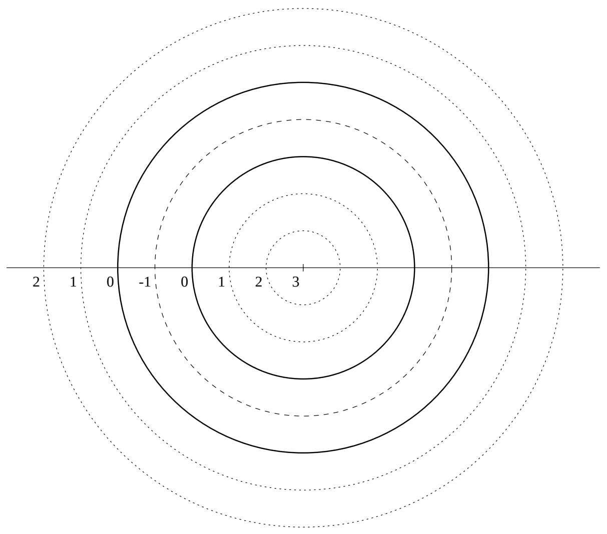


Figure 8.3: Planar slice through a `CadnoSoftTorus`, showing the surrounding field.

shape when components are in close proximity to one another. When skeletal elements are combined by summing the contribution of their function to the field, that contribution cannot be a linear distance fuction as two objects located a significant distance away from one another can interfere with one another. Field functions, typically a cubic function applied to the linear distance fuction for a point, allow a component to have its maximum effect on the whole shape close to its boundary.

In *empirical world* scripts, the field function has been extended so that it also operates for fields around any function representation of geometric shape. Modifying the field around a skeleton effects how geometric objects interact in blends, morphs, sums and so on. For geometric objects that extend `SoftType`, this is a well defined field with values dependent on distance away from the skeleton. For solid geometry represented by classes that do not extend `SoftType`, a field may not be uniform and a user should be aware of this in their selection of a field function. After the application of the field function, any distance properties of the fucntion representation are significantly reduced or lost and the field function representation produces poor results when used in blending and morphing operators, in comparison to the original representation.

To explicitly define a soft shape in an *empirical world* script, a user can use the `CadnoSoftShape` class, which extends `TopologyType`. The parameters required for this explicit definition are:

- the existing geometric shape given by function representation g that is considered a `skeleton` shape;
- a field function parameter is required to define how the field varies for a particular value of g at any point. The parameter is called the `fieldFunction` and must be one of the explicit definitions for a `GraphType` (see Appendix A.5).
- a `weight` value w by which the value of the field function of the skeleton function at a particular point is multiplied;
- the `isoValue` i which determines at what level the value returned by the weighted field function is considered to be the surface of the geometry.

The bounding box for a `CadnoSoftShape` is calculated by starting with the bounding box for the `skeleton` geometry and then executing the `shrinkWrap()` method. The resulting shape for certain weight values w, field functions h or isovalues i may be infinite in which case the bounding box will clip or even miss altogether the bounds of the shape. The user of an *empirical world* script should be aware that this is a possible problem and should choose their parameters carefully.

Class Name	`CadnoSoftShape`	
Extends	`TopologyType`	

Value Format	`SoftShape` {
	    `skeleton` *FrepType*
	    `fieldFunction` *GraphType*
	    `weight` *float*
	    `isoValue` *float*
	}	

Parameters	Name	Type
	`skeleton`	`FrepType`
	`fieldFunction`	`GraphType`
	`weight`	`CadnoFloat`
	`isoValue`	`CadnoFloat`

Soft field functions can be defined implicitly in *empirical world* scripts by using the `CadnoMakeSoftShape` class, which extends `DefnFunc`. The arguments to this operator in an implicit definition of a soft shape are:

1. the `skeleton` geometric object of `FrepType`;
2. the `fieldFunction` instance of a class that extends `GraphType`;
3. the `weight` value w;
4. the `isoValue` i.

Operator Class		`CadnoMakeSoftShape`
Maps From		`CadnoDetail` × *FrepType* × *GraphType* × *float* × *float* or *FrepType* × *GraphType* × *float* × *float*
Maps To		`CadnoSoftShape`
Example	Defn.	`ss = softShape(SoftSphere { }, FieldGraph { }, 1.1, 0.2)`
	Value	`ss = SoftShape { skeleton SoftSphere { radius 1 } fieldFunction FieldGraph { raise 2 constant 1 coefficient 2 } weight 1.1 isoValue 0.2 }`

Field functions for use in modelling with skeletal implicits are intended to work for soft objects where the argument to the function is a positive distance away from the skeleton if the point of sampling is outside the skeletal geometry. To remain consistent with this, the argument to the field function in a `CadnoSoftShape` is a negative value of original function representation g for any point (x, y, z). If g is a function representation for one of the geometric objects of `SoftType`, this achieves consistency because for any instance of `SoftType`, $g(x, y, z) = -d(x, y, z)$.

The function representation f for a soft shape at any point (x, y, z) is

$$f(x, y, z) = i - wh(-g(x, y, z)) \tag{8.5}$$

It is not possible to make any claim about the continuity of this function because the continuity of the field function h is not known. To convert the value calculated for the field function into a function representation for the geometric shape, this value is subtracted from the isovalue i. The `isovalue` parameter allows a user to interactively experiment so as to determine which level of the field produces the most suitable shape and size of geometry for their purposes.

8.2.5 Summing the Fields of Skeletal Implicit Shapes

Once definitions of skeletal geometry exist as function representations, all the set theoretic operations, affine transformations, morphing and warps can be used to modify and combine their shape with other geometric shapes, be these soft or otherwise. One special feature of skeletal geometry is that it is combined with appropriate field functions, such as the `FieldGraph` available in *empirical world* classes (see Appendix A.5), field values at any point can be summed so that when two soft shapes are in close proximity to each other they appear to *automatically* blend. This effect can be achieved in *empirical world* scripts using the explicit data type of the `CadnoSoftSum` class.

The parameters to a `CadnoSoftSum` class are a list of `children` of function representations for geometry, given by $g_1, \ldots, g_n$. The other parameter to an explicit definition of a value of this type is the field function sampling level i called `isoValue`.

Class Name	`CadnoSoftSum`	
Extends	`TopologyType`	
Value Format	`SoftSum` { `isoValue` *float* `children` [(*FrepType*)*] }	
Parameters	Name	Type
	`isoValue`	`CadnoFloat`
	`children`	List of objects of `FrepType`.

Soft sum geometric shape can be implicitly defined in an *empirical world* script with the use of the `CadnoMakeSoftSum` class, which extends `DefnFunc`. The arguments to the operator in an implicit definition for a soft shape are the isovalue i followed a comma separated list of geometric objects with function representation.

These geometric objects should have an appropriate field function applied to their original skeletal defining geometry, such as the `FieldGraph`, and the soft sum is likely to produce the best results for geometry with skeletons that extend `SoftType`.

Operator Class		`CadnoMakeSoftSum`
Maps From		`CadnoDetail` × *float* × (*FrepType*)* or *float* × (*FrepType*)*
Maps To		`CadnoSoftSum`
Example	Defn.	`ss = softSum(0.1, ssh1, ssh2)` *ssh1* and *ssh2* should be `CadnoSoftShapes`
	Value	`ss = SoftSum { isoValue 0.1 children [ ssh1 ssh2 ] }`

The function representation f for a soft sum at any point (x, y, z), the sum of all the fields of its `children`, is given by the equation

$$f(x, y, z) = i - \sum_{j=1}^{j=n} (-g_j(x, y, z)) \tag{8.6}$$

Figure 8.4 illustrates soft summing of a soft torus centred at the origin and a soft sphere that is translated by the vector given below each image. As the sphere is translated towards the torus, the fields of the two objects merge, creating the effect of the geometric objects blending. The isovalue for the sum shown was 0.5, and the field function h for any value x is given by the equation shown in Figure 8.5 and given by the equation

$$h(x) = 2(1 - 2^{-x})2^{-x} \tag{8.7}$$

8.3 Warping Transformations in *Empirical Worlds.*

The transformations represented by a `CadnoTransform` class are the common affine transformations (see Section 7.3.6). In *empirical worlds*, some interesting effects can

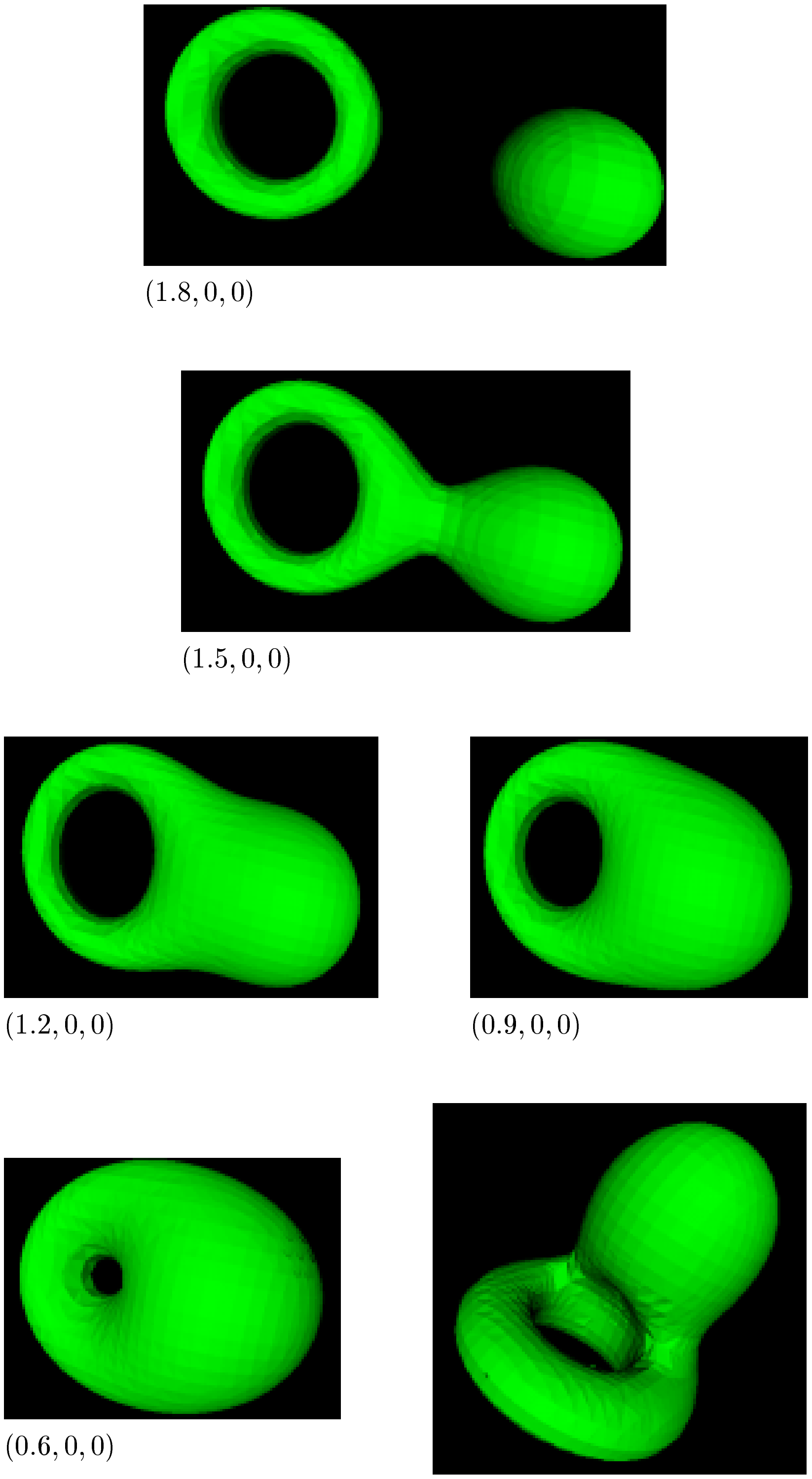


Figure 8.4: Images of a `CadnoSoftSum` of a `CadnoSoftTorus` and a translated `CadnoSoftSphere` (translation vector shown by each image).

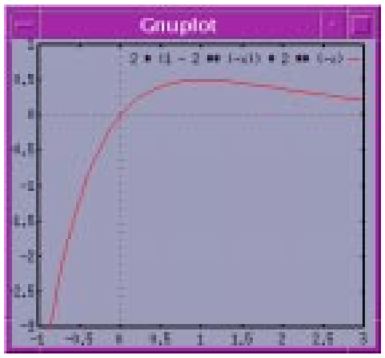


Figure 8.5: Plot of example field function h

be achieved through other transformations (linear and non-linear) that simulate the effect of tapering, bending and twisting objects.

Each of the transformations described in this section is based on the operations described by Wyvill and van Overveld in [WvO97] which are based in turn based on those described by Barr in [Bar84]. For any point $\mathbf{p}$ in space and function representation of a shape g, the warping transformation w of the shape at point $\mathbf{p}$ is a mapping from warped space to Euclidean space such that

$$f(\mathbf{p}) = g(w(\mathbf{p})) \tag{8.8}$$

In this formula, f is the function representation of the warped shape. In *empirical worlds*, each warp has a type that extends `DefnType` and an operator for their implicit definition that extends `DefnFunc`. Each type or operator takes one existing (`original`) piece of geometry and performs the transformation on this geometry. In this section, new VRML-like syntax is introduced for the description of values for data types represented.

8.3.1 Linear Taper

A linear taper operation has a defining axis and a dimension that it affects. In the warped space, the affected dimension has every coordinate point value increased by a factor in linear proportion to the point's projected coordinate value along the taper's defining axis. The example of a linear taper in *empirical world* scripts is a taper of the x dimension along the z defining axis. If the axis and defining dimension are not appropriate, then the geometry can be transformed to suit the taper operation and then transformed back to its original location.

The data type for explicit definition of an instance of a tapered geometric object in an *empirical world* scripts is represented by the `CadnoTaperXZ` class. Explicit definitions requires two parameters:

- a `taperRate` w that represents the rate at which the warp affects the coordinates of the x dimension proportional to the z coordinate;
- an explicit value description of the `original` geometry prior to the warp. This pre-warp geometry can be any geometric object given by a function representation.

The bounding box for the newly tapered shape is calculated by tapering the bounding box for the original geometry and finding a bounding box for this.

Class Name	`CadnoTaperXZ`	
Extends	`TopologyType`	

Value Format	`TaperXZ {`
	    `taperRate` *float*
	    `original` *FrepType*
	`}`	

Parameters	Name	Type
	`taperRate`	`CadnoFloat`
	`original`	`FrepType`

A linearly tapered shape with a defining axis along z and effected dimension x can be implicitly defined in an *empirical world* script by using the `CadnoMakeTaperXZ` class, which extends `DefnFunc`. The arguments in an implicit definition of a taper are the `taperRate` parameter w followed by the pre-warp `original` geometric shape with function representation g.

Operator Class		`CadnoMakeTaperXZ`
Maps From		`CadnoDetail` $\times$ *float* $\times$ *FrepType* or *float* $\times$ *FrepType*
Maps To		`CadnoTaperXZ`
Example	Defn.	`tp = taperXZ(-0.4, Box { })`
	Value	`tp = TaperXZ {` `taperRate -0.4` `original Box { size 2 2 2 }` `}`

The function representation f for a tapered shape as defined in an *empirical world* script by a definition of a `CadnoTaperXZ` is given by the function f below. It is obvious from the equation that coordinates in the y and z dimension are preserved and only the x dimension is effected. An example of such a taper is shown in Figure 8.14 where the original geometry is shown in Figure 8.12.

$$f(x, y, z) = g(x(1 + zw), y, z)$$

8.3.2 Bend

In a similar way to the taper operation, a bending operation has a defining major axis and a dimension along which its effect is measured. An analogy to real-world bending is that the defining axis is like a fixed post about which the geometry is to be bent by applying a force at each end of the geometry. This is illustrated in Figure 8.6 that shows three stages of bending a long box about the major defining z-axis, where the cross “$\times$” represents the origin point of the space. As a force is applied to both ends of the geometry, each point of the geometry bends by an angle

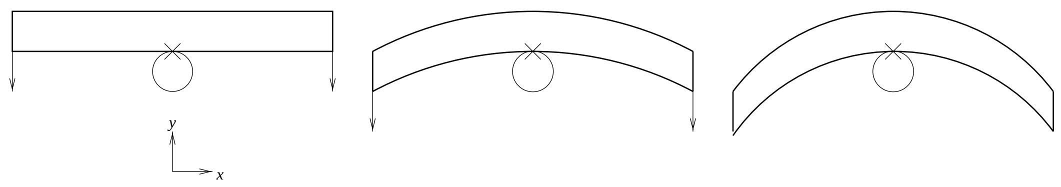


Figure 8.6: Planar slice in the xy-plane showing the bending of a long box about the z-axis with the amount of bend measured along the x-axis.

proportional to the the x component of its original point.

In virtual spaces, like those defined in an *empirical world* script, a bend operation can be achieved by specifying a bend rate k in radians per unit length and an offset value for the centre of the bend along the x-axis *x0*. An explicit definition of some bent geometry about the z-axis with amount of bend measured along the x-axis is represented using the `CadnoBendXZ` class. An explicit definition contains the explicit values for a floating point parameter called `bendRate` which is the value for k, a floating point value `offset` which is a value for *x0* and `original` which is an explicit description of a geometric shape that is to be bent by this operator, with function representation g. The bounding box for the bent geometry is found by taking the bounding box for the original geometry and executing the `shrinkWrap()` method.

Class Name	`CadnoBendXZ`	
Extends	`TopologyType`	

Value Format	`BendXZ` {
	 `offset` *float*
	 `bendRate` *float*
	 `original` *FrepType*
	 }	

Parameters	Name	Type
	`offset`	`CadnoFloat`
	`bendRate`	`CadnoFloat`
	`original`	`FrepType`

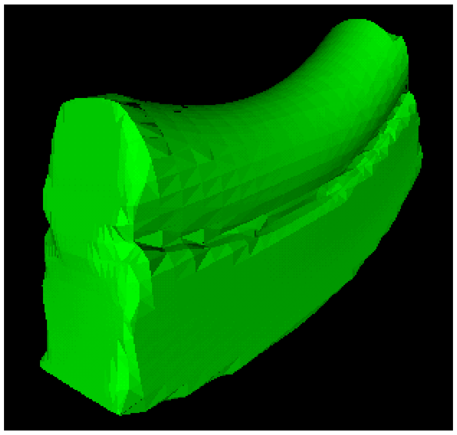


Figure 8.7: Image of a blended long box and cylinder that have been bent around the z-axis, along the x-axis.

Figure 8.7 shows a blended long box and cylinder that have then been bent by the bending operation.

Bent geometric shapes, with major defining axis the z-axis and with the amount of bending measured along the x-axis, can also be defined implicitly in *empirical world* scripts using the `CadnoMakeBendXZ` class, which extends `DefnFunc`. The order of arguments in an implicit definition of a bend are the `bendRate` value k, followed by the `offset` parameter *x0* and the `original` solid geometry with function representation g.

Operator Class		`CadnoMakeBendXZ`
Maps From		`CadnoDetail` × *float* × *float* × *FrepType* or *float* × *float* × *FrepType*
Maps To		`CadnoBendXZ`
Example	Defn.	`bd = bendXZ(1, 0.78, Box { size 5 1 2 })`

	Value	`bd = BendXZ {`
		`        offset 1.0`
		`        bendRate 0.78`
		`        original Box { size 5 1 2 }`
		`        }`

The function representation for the bent geometry f at any point (x, y, z) is

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = g\begin{pmatrix} -\sin(k(x-x0))(y-\frac{1}{k})+x0 \\ \cos(k(x-x0))(y-\frac{1}{k})+\frac{1}{k} \\ z \end{pmatrix} \tag{8.9}$$

The warping has no effect on the z dimension and it can be seen in the equation how the amount of bend is proportional to the value of the offset of the x coordinate of a point.

8.3.3 Twist

A twist operation is can be defined to operate around one major axis. A twisted piece of geometry is created from its original geometry by rotating each point in the original geometry around the defining axis through an angle proportional to the component distance for the point along the defining axis. In *empirical world* scripts, a twist of geometry along the z-axis can be explicitly defined using the `CadnoTwistZ` class. Parameters for twisted geometry are a floating point value for the `twistRate` w in radians per unit length measured along the z-axis and the `original` solid geometry with function representation g. The bounding box for such geometry is found by taking the bounding box for the `original` geometry and executing the `shrinkWrap()` method on the twisted version of the geometry to find the new bounds.

Class Name	`CadnoTwistZ`	
Extends	`TopologyType`	

Value Format	`TwistZ {`
	    `twistRate` *float*
	    `original` *FrepType*
	`}`	

Parameters	Name	Type
	`twistRate`	`CadnoFloat`
	`original`	`FrepType`

Figure 8.8 shows a blended box and cylinder twisted around the z-axis in

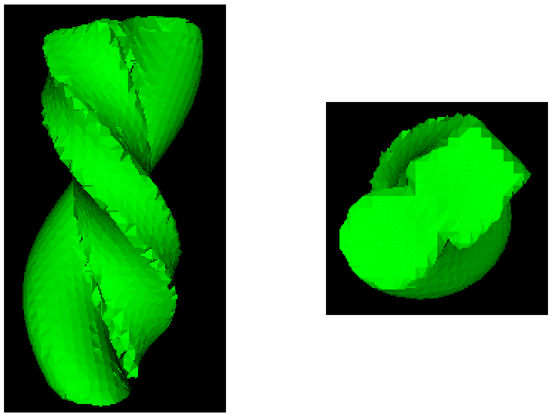


Figure 8.8: Image of a blended box and cylinder twisted around the z-axis

.

side and top views.

Twisted geometric shapes, with defining axis the z-axis, can be defined implicitly in *empirical world* scripts using the `CadnoMakeTwistZ` class, which extends `DefnFunc`. The arguments to an implicit definition of a twist are the `twistRate` w followed by the `original` pre-warp geometry with function representation g.

Operator Class		`CadnoMakeTwistZ`
Maps From		`CadnoDetail` × *float* × *FrepType* or *float* × *FrepType*
Maps To		`CadnoTwistZ`

Example	Defn.	`tw = twistZ(detail 10 10 30, 0.78,`
		`Box { size 1 2 5 })`

	Value	`tw = TwistZ {`
		`twistRate 0.78`
		`original Box { size 1 2 5 }`
		`}`

The function representation f at any point (x, y, z) for a geometric shape twisted around the z-axis is

$$f\begin{pmatrix} x \\ y \\ z \end{pmatrix} = g\begin{pmatrix} x\cos(zw) - y\sin(zw) \\ x\sin(zw) + y\cos(zw) \\ z \end{pmatrix} \tag{8.10}$$

The warping transformation preserves the z-coordinate component of points and rotates the x and y components by an angle proportional to the z component.

8.4 Implementation of Empirical Worlds

The *empirical world builder* tool is an application based on the empirical world classes illustrated this chapter and Chapter 7. The application is a computer-based tool for exploration of shape in an empirical manner. The *empirical world builder* tool is based on a server/client architecture, where a computer acting as a world-wide-web or file server also runs the server-side application. This application is a *JaM server* as described in Section 6.4.2 of Chapter 6. The client application, a *JaM Client*, connects to the server system through network sockets and *empirical world* scripts can be created and edited through the client. The client also includes a VRML browser that allows a user to interactively explore the shape they have described in their *empirical world* script.

8.4.1 Empirical World Builder Server

The server, called `EWServer`, is a command line based Java application that can be run on any Java Virtual Machine. The server is multi-threaded application that listens for connections on a TCP/IP network socket. When the server receives a connection request on the port to which it is listening, a new `java.lang.Thread` class is created to handle the connection. No limit is set for the number of clients that can connect to the server, although there are some system specific limits on numbers of connections that can restrict the numbers of simultaneous clients.

When a new thread is created, its initialization creates an instance of the `JaM.Script` class and adds all the *empirical world* types and functions to this `Script`. There is only one agent for each of these scripts, the `root` super-user agent[2]. The thread then enters a continuous loop which waits at the start of each iteration for user input from the associated client. User input can include new definitions, requests for information on current definitions and instruction to close the network socket and shutdown the thread. All messages sent from the client to the server are streams of characters that are converted to `java.lang.String` objects on the server. Messages from the server to the client are either streams of characters for printing in the client's output window, or streams of data representing some VRML for display in the client's VRML browser.

In response to new definitions or redefinitions, the `Script.addToQ()` method is called to add these definitions to the queue of definitions waiting for the next update of the `Script`. In the strings of characters from a client, the server recognises new definitions or redefinitions as they contain an equals sign (`=`). If the `addToQ()` or `update()` method for the script instance throws an exception due to recently received definitions, an error message is sent to the client along with the `String` of characters that triggered the exception. This allows the user to re-edit the string in case the reason for the error was a simple typing mistake.

In addition to definitions, direct instructions can be issued from the client to the server. A user of the client can type the following instructions:

`update` Causes the server to call the `Script.update()` method so that if there are any redefinitions on the queue, the state of the script is updated.

`value` *id* A user instructs the server to return the value of identifier *id* in the current *empirical world* script, by calling the `printVal()` method (see Table 6.5). The server either returns the explicit value of *id*, or an error if there is no identifier called *id* in the script.

`defn` *id* A user instructs the server to return the implicit definition of the identifier

[2]See Section 6.3.2 in Chapter 6 for more details.

id in the current *empirical world* script by calling the `printDef()` method. The server either returns the implicit definition of *id* if it is implicitly defined, otherwise the explicit value is returned. If there is no identifier called *id* in the script, an error is reported to the client.

`printall` A user request from the client that instructs the server to send to the client a list of all definitions in the current state of the *empirical world* script. The server does this by creating a string of characters containing the output of the method `printAll()` that contains every definition in the script.

`?` *id* A user instructs the server to return to the client some additional information about identifier *id* other than its explicit value or implicit definition. The server calls the `inspectDef()` method of the script instance for the identifier *id*. If *id* exists within the script, its owner, permission fields and dependents definitions are sent back in `String` form to the client, otherwise an error is reported. In the current implementation, JaM permissions and owners are not in use so every definition is owned by the super-user `root` and has all definition permissions set to "`rwr-`".

`exit` A user instructs the server to close the network connection to the client and destroy the thread for handling this connection. The destruction of the thread also destroys the *empirical world* script that was associated with the connection to the client.

The server also performs any necessary polygonisation of implicit shapes in classes that extend the `TopologyType` class. This polygonisation is described in Section 8.4.3 of this chapter. The motivation for polygonisation to be carried out on the server is the assumption that a server computer often has more numerical computing power than a client and client computers resources can be dedicated to the exploration of shape through a VRML browser.

8.4.2 Empirical World Builder Client

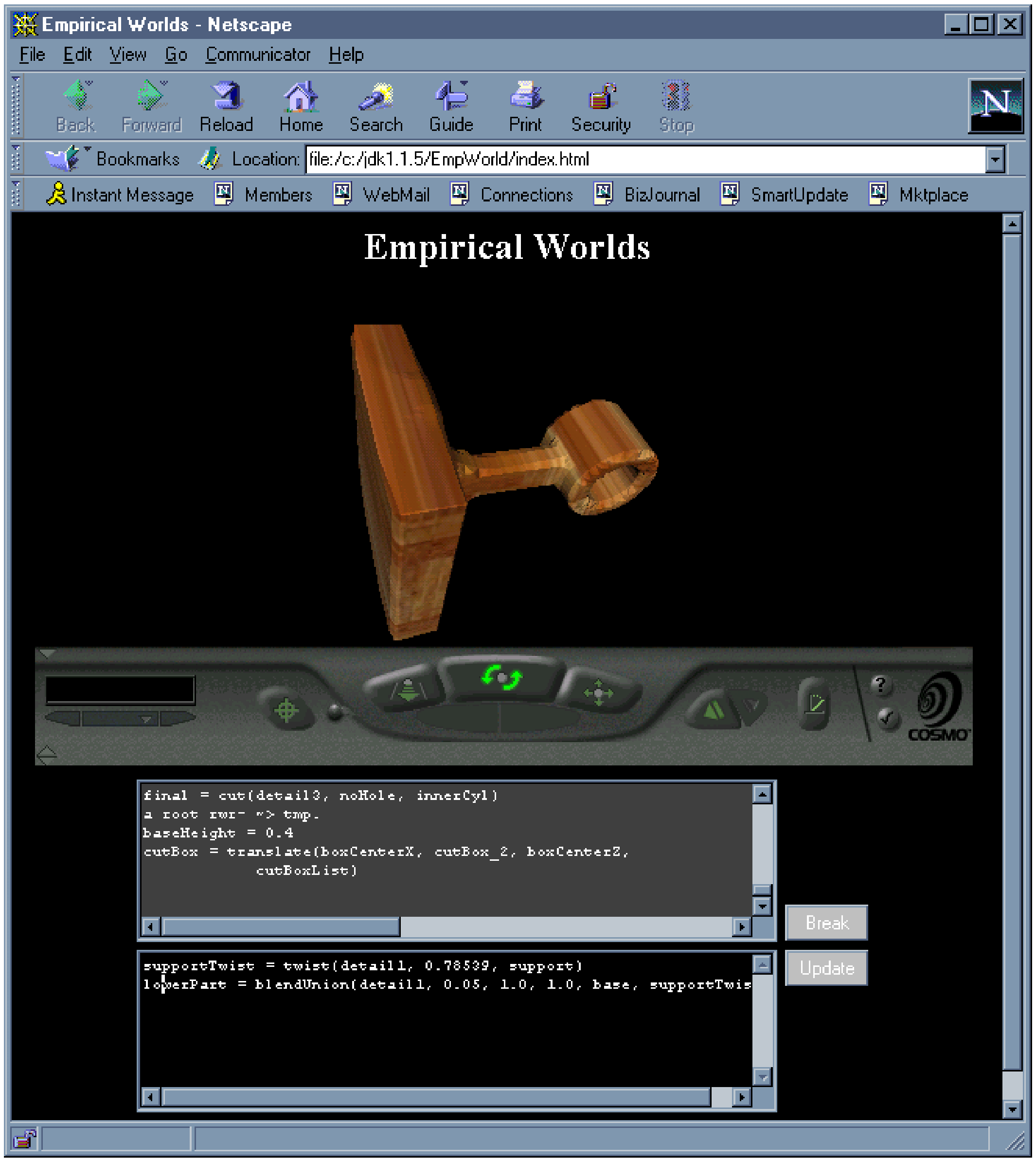
The *empirical world builder* client application is implemented as a web page containing a *CosmoPlayer*[3] VRML browser plugin and a Java applet that communicates with the VRML browser and the *empirical world* server. In the screen snapshot of a Netscape Communicator browser shown in Figure 8.9, the client web page is demonstrated when in use. The top panel in the window with some geometry displayed and some viewing controls is the CosmoPlayer browser. The bottom panel containing two text components and two buttons is the Java applet interface between the VRML browser and the *empirical world* server.

The Java applet in the bottom panel is initially disconnected from the server. The two buttons are labelled `Connect` and `Update`. Clicking on the `Connect` button causes the client to try to connect to an *empirical world builder* server on the server from which the whole web page was downloaded[4]. Output from the server and error messages directly from the client appear in the top text component of the applet panel, which has a grey background. The user cannot edit any text in the component but can copy its contents to the operating system clipboard. When connection to the server is established, the message "`Connected to server.`" appears in this output window and the `Connect` button changes to a "`Break`" button that forces disconnection.

One implementation special operator exists in *empirical world* scripts that defines the interface between the values for definitions in a script and the client's VRML

[3]This web browser plugin, developed by a subsidiary of Silicon Graphics Inc., is freely available and runs under Windows 95, Windows NT, HP-UX and Irix. See `http://www.cosmoplayer.com/` for more information.

[4]One of the security restrictions enforced by Java applet *Security Managers* is that a network client applet can only connect to the server from which the Java classes for that applet were downloaded [Har97].


Empirical Worlds - Netscape
File Edit View Go Communicator Help
Back Forward Reload Home Search Guide Print Security Stop
Bookmarks Location: file:/c:/jdk1.1.5/EmpWorld/index.html
Instant Message Members WebMail Connections BizJournal SmartUpdate Mktplace
Empirical Worlds
COSMO
Break
Update

```
final = cut(detail3, noHole, innerCyl)
a root rwr- ~> tmp.
baseHeight = 0.4
cutBox = translate(boxCenterX, cutBox_2, boxCenterZ,
                cutBoxList)
```

```
supportTwist = twist(detail1, 0.78539, support)
lowerPart = blendUnion(detail1, 0.05, 1.0, 1.0, base, supportTwis
```

Figure 8.9: Screen snapshot of a web browser when viewing the interactive *empirical worlds* page.

browser or a file. This is the `CadnoWorld` class that directly extends `DefnFunc`[5]. In its current form, the operator maps a list of geometric objects (`FrepList`) defined by function representations to the same list, with the side effect that a file containing VRML-2 nodes to describe that geometry is generated. For an implicit definition of the form "*identifier* `= world` (*list1*)", a file called "*identifier*`.wrl`" is generated on the local filing system to the server, in the same directory from when the *empirical world* client was downloaded. A special case exists when the *identifier* is "`w`". Instead of the VRML being sent to a file, it is sent to the VRML browser of the *empirical world* client[6].

Operator Class		`CadnoWorld`
Maps From		`FrepList`
Maps To		`FrepList`
Example	Defn.	`w1 = world(children [ Box { } ])`
	Value	`w1 = children [ Box { } ]`
		Generates VRML file `"w1.wrl"`.

The user types new input into the bottom text component of the applet panel, which has a black background. Each line of the text is considered as a separate definition, redefinition or server instruction. When the `Update` button is pressed, the current content of the text component is sent to the server line by line and the server responds in the way described in Section 8.4.1. The client appends an `update` instruction onto the end of each package of definitions and instructions, so that the script on the server will be updated consistent with any definitions currently on the scripts queue every time the `Update` button is clicked. The input text component is also cleared, awaiting further new user input.

[5]The position of the class in the *empirical world* class hierarchy is shown in a circled box in the class diagram in Figure 7.2.

[6]The variable `w` in *empirical world builder* has the same purpose as the `screen` variable in SCOUT [Dep92].

If there are any errors during the `Script.addToQ()` method on the server then these are reported by being appended to the end of the output text component. The definition which caused the error is returned to the client and placed in the input window for editing and then resubmission to the server by the user.

If the update propagate to the special *world* identifier "`w`" then the VRML displayed in the CosmoPlayer VRML browser is updated consistent with the definition of "`w`". The user can explore the selected redefined shapes which are dependencies for `w`, after its VRML description has been transmitted from the server to the client and then rendered by the client in the browser. This process normally takes no more than a few seconds. The user can then rotate, zoom, pan, seek sections, fly around and walk around the geometry displayed in the VRML browser.

In the current implementation, there is no facility for file save and load for an *empirical world* script. It is possible to cut and paste between the text components of the client's applet and other applications such as text editors. To save the current state of a script, a user can issue the `printall` command and retrieve a list of all definitions and their current values in the output text component. This can be selected and pasted into a text editor, edited and saved to a file. Files of definitions can be loaded into the client by selecting and copying to the clipboard the required definitions and then pasting them into the input window.

8.4.3 Polygonisation of Implicit Solid Geometry

The polygonisation of implicit shape used for the *empirical world builder* tool is simplistic and does not produce the best visual results. It was written to be fast and simple, to demonstrate that the tool has potential as an interactive modelling tool rather than to produce high quality visual images suitable for brochures or publications. The polygonisation method presented here is not part of the core work of this thesis, and is presented to demonstrate future potential for tools based on these classes. The algorithm used is similar to the *Marching Cubes* algorithm [LC87]. The future incorporation of a better algorithm is a trivial task as all the method calls and data structures for the polygonisation are already in place.

All objects of `FrepType` contain information about a bounding box for their solid material and a level of detail to which they should be rendered. (The level-of-detail data type is described in Section 7.2.1 of this chapter.) The space within the bounding box for the geometry is split into the number of voxels specified by the level of detail, each voxel having the same dimensions. The voxels form a three-dimensional grid contained within and exactly bounded by the bounding box. Any voxel corner vertex that does not lie on the face of the bounding box is a corner point for three other voxels. For the instance of `TopologyType` being considered for polygonisation, the initial phase of the algorithm is to sample the function representation of the geometry at each corner vertex of all the voxels calling the method `f()` with the coordinates of the vertices. The results of these samples are stored in a three-dimensional array data structure.

Each voxel is then considered in turn. Each voxel has eight vertices and each vertex can either be inside, outside or on the surface of the geometry represented by the instance of `FrepType`. The two cases for "inside" or "on the surface of" are combined into one making a total of 256 possible cases to consider for the way in which a geometric shape can be polygonised within a single voxel. Each of these 256 cases is considered separately in the second phase of the algorithm. Two special cases exists, where all the vertices lie within or on the surface of the shape, or all the vertices lie outside the shape, when the algorithm takes no action and produces no polygons. In all other 254 cases, at least one polygon is created.

Each edge of a voxel with end vertices $\mathbf{p}$ and $\mathbf{q}$ is split at point $\mathbf{s}$ if the edge crosses the surface boundary. This point $\mathbf{s}$ is a linear estimation of the actual position of the surface of the geometry, found by the solution of

$$tf(\mathbf{p}) + (1-t)f(\mathbf{q}) = 0 \tag{8.11}$$

with respect to t to define

$$\mathbf{s} = t\mathbf{p} + (1-t)\mathbf{q} \tag{8.12}$$

Each point $\mathbf{s}$ that can be found for an edge is used as the vertex of a triangle that contributes to the polygonisation of the whole geometry. If there are more than three connected edges of the voxel that can be split, there will be more than one triangle contributed to the overall polygonisation by this voxel. In the current implementation, an arbitrary choice is made without reference back to the geometry about how it is best to do this, which leads to a strange edge effect in the geometry such as warped shapes, as illustrated in Figures 8.8 and 8.7. An improved algorithm would make this decision with further sampling of the function representation[7].

Figure 8.10 shows a voxel with the eight vertices at which the function representation for the solid geometry is sampled. In this example, only one vertex is inside the geometry and this is depicted by a filled circle. Three edges that are cross a surface boundary for the solid are split to define the edge points of a triangle, shown in the figure by a dotted line. This triangle is used as part of the polygonisation of the solid geometric shape.

8.4.4 Rendering Issues

The *empirical world* server application calculates the polygonisation of geometric shapes described by function representations. In this section, statistics on the rendering times for some example geometric shapes are presented. These statistics:

[7]An appropriate way to improve the method would be to sample the function representation at a point in the centre of the voxel and split the problem of polygonising the geometry inside the voxel into polygonising inside six equal sized, square based pyramids.

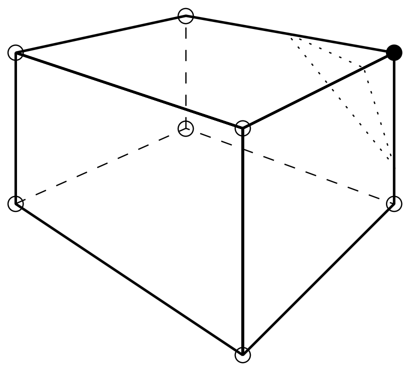


Figure 8.10: One example case for a voxel used in polygonising some solid geometry.

- demonstrate the genuine improvement over the scope for rendering in EDEN (cf CADNORT in Section 3.4.2);
- illustrate the potential for realistic real-time interaction;
- illustrate the network costs involved in a client/server application;
- demonstrate the potential use for Java as a programming platform for shape modelling.

The *empirical world* server produces statistics of the length of time each phase of the algorithm took to compute[8]. Some examples of these timings, averaged over five samples, are shown in units of seconds in Table 8.1. In this table, "*phase 1*" columns represent average timings for the sampling of the function representation at the corner points of the three-dimensional voxel grid. The "*phase 2*" columns are overall timings for the computation of polygons inside each voxel. The "*transmit*" columns show the time that it takes for the polygon image to be transmitted from the server to the client[9]. The polygonisation of three different solid shapes are timed, each at three detail levels of $10 \times 10 \times 10$ voxels, $20 \times 20 \times 20$ voxels and $30 \times 30 \times 30$

[8]Timings were recorded by executing the *empirical world* server on a Pentium 166Mhz processor with 32Mb memory, using Microsoft's *Windows 95* operating system and Sun Microsystems' *Java Virtual Machine*.

[9]This is dependent on the speed of the local network and included only for a guide to the wait between definitions being typed and displayed.

Shape	`detail 10 10 10`			`detail 20 20 20`			`detail 30 30 30`		
	phase 1	*phase 2*	*tran-smit*	*phase 1*	*phase 2*	*tran-smit*	*phase 1*	*phase 2*	*tran-smit*
Sphere	<0.05	0.07	0.26	0.08	0.66	1.29	0.28	3.09	3.01
Blend of a box and a cylinder	<0.05	0.06	0.27	0.14	0.50	1.10	0.50	2.25	2.43
Twist of previous blend	<0.05	0.06	0.34	0.19	0.48	1.51	0.64	2.00	3.62

Table 8.1: Average timings in seconds for polygonisation by the *empirical world* server.

voxels. The first row is for a sphere, the second for the blend of a box and a cylinder and the third for a twist of the blended shape. Timings marked "`<0.05`" were below accurate measurement thresholds (large standard deviation) but were consistently less that 0.05 seconds.

The table shows that phase 1 in these examples typically takes less time than phase 2 and that all timings increase with the level of rendering detail. As the complexity of the shape increases by the number of operators that define the shape (number of levels below the shape in its dependency structure), the time to compute the function representation increases for the same number of points, as shown in the "*phase 1*" columns. The sampling of a sphere takes fewer nested computations with only one level in its dependency structure than the sampling of a twisted blend of a box and a cylinder with a least three levels in its dependency structure. All timings along the rows are proportional, within a margin of timing error, to the number of points at which the function representation are calculated. For a detail of $10 \times 10 \times 10$ this is $11^3 = 1331$ points, $20 \times 20 \times 20$ this is $21^3 = 9261$ points, $m + 1 \times n + 1 \times r + 1$ this is mnr points.

The use of *empirical world* scripts in such a way that they can be interactively explored requires that the user in control understands ways in which they can fine tune their model to optimise the quality and usefulness of modelling interaction.

The methods available in the open-ended client user environment for the user to enhance the shape modelling experience include:

- balancing quality of image with speed of rendering;
- selecting to view an entire geometric part or its subcomponents;
- restructuring the script to reduce the number of levels in its dependency structure.

The client tool is considered as an instrument for shape modelling.

8.5 Example of an Empirical World Script

In this section a case-study of the creation and interaction with some geometry in *empirical worlds* is presented. The geometry represents a curtain pole support. Figure 8.11 shows a sketch of the support in front and side views. This sketch can be considered as specifying the real-world object for the model. The figure is labelled with parametrisations that are considered to be important in the description of the geometry of the shape.

This case-study illustrates a shape that could easily be represented with computer software for CSG modelling[10]. Section 8.5.2 illustrates some redefinitions of the geometry of the support that would not be possible in many CSG modelling packages. The *CSG tree* common to many solid modelling programs is replaced in a script with a dependency structure. It is also interesting to note that parametric modellers such as Parametric Technology's *Pro/Engineer*[11], a constraint based system, requires the recalculation of a whole piece of geometry when one defining

[10]Matra Datavision's *Prelude Solids* package is an example of such a tool. See `http://www.matra-datavision.fr/Products/Family/Prelude/index.html`.

[11]See `http://www.ptc.com/products/mech/proe/index.htm`.

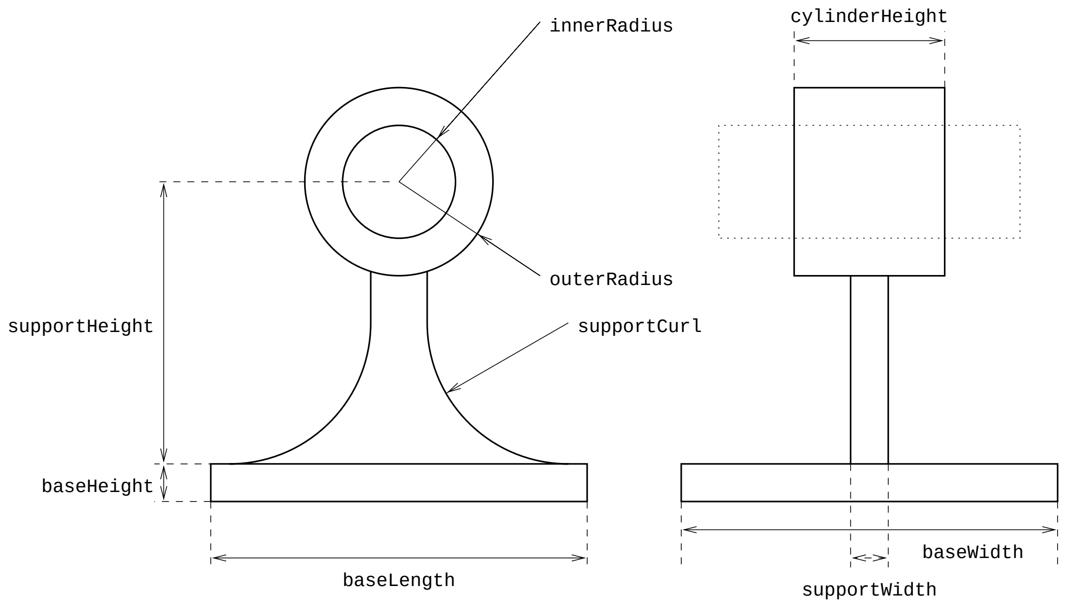


Figure 8.11: Two-dimensional representation of a curtain pole support with parametrisations.

parameter is altered (see Section 2.2.2). With the *empirical worlds* tool, only components of the overall geometry of the support that need updating are updated, not all the components.

8.5.1 Constructing the Curtain Pole Support Script

The parametrisations of Figure 8.11 are transferred into an *empirical world* script for the curtain pole support as a demonstration of a suitable process for the creation of a model. Firstly, a box is constructed to represent the base of the support. All primitive shapes are initially created centred at the origin and then translated to their final positions. (The implicit references `halfBaseLength` and `halfBaseHeight` are introduced here for use in definitions that appear further through the script.)

```
baseHeight     = 0.2
baseLength     = 2.0
baseWidth      = 2.0
halfBaseLength = divide(baseLength, 2)
halfBaseHeight = divide(baseHeight, -2)
baseOriginal   = box(baseLength, baseWidth, baseHeight)
```

The second stage of the modelling process is to create the cylinders that will form the head part of the curtain pole support. This head will be constructed from a solid cylinder with another cylinder used as a tool to remove material from its interior. The inner tool cylinder is set to be taller (longer) than the body cylinder, so that problems associated with rendering coincident faces when their are slight inaccuracies in floating point arithmetic are not introduced.

```
cylinderHeight   = 0.8
holeHeight       = multiply(cylinderHeight, 2)
outerRadius      = 0.5
innerRadius      = 0.3
outerCylOriginal = cylinder(cylinderHeight, outerRadius)
innerCylOriginal = cylinder(holeHeight, innerRadius)
```

The support that connects the base and the head of the curtain pole support is the most complex piece of geometry. It is constructed from a thin box with two cylinders and two boxes cut away from it. The original geometry for these components is described by the segment of script shown below. The original body of material for the support has identifier `supportBodyOriginal`, the cylinder that will be the tool that cuts away material to make the curved shape towards the base of the support is `cutCylOriginal` and the material cut away to make the neck of the support is `cutBoxOriginal`.

```
supportHeight       = 1.5
supportCurl         = 0.75
supportWidth        = 0.2
supportLength       = 0.3
doubleSupportCurl   = multiply(supportCurl, 2)
halfSupportLength   = divide(supportLength, 2)
halfSupportHeight   = divide(supportHeight, 2)
supportBodyOriginal = box(baseLength, supportWidth,
                                  supportHeight)
cutCylOriginal      = cylinder(baseWidth, supportCurl)
cutBoxOriginal      = box(doubleSupportCurl, baseWidth,
                                  supportHeight)
```

The third stage is to translate the `cutCylOriginal` and `cutBoxOriginal` into their correct locations to become tools to cut the support. Their respective translated geometries are identified as `cutCyl` and `cutBox`.

```
cylCenterX = add(halfSupportLength, supportCurl)
cutCyl     = translate(cylCenterX, 0.0, supportCurl,
                                      cutCylOriginal)

boxCenterX = add(supportCurl, halfSupportLength)
boxCenterZ = add(halfSupportHeight, supportCurl)
cutBox     = translate(boxCenterX, 0.0, boxCenterZ,
                                      cutBoxOriginal)
```

At this point, the tool for cutting the support geometry can be created as a whole without creating separate cylinder and box tools for the left and right hand sides of the cut. The union of `cutBox` and `cutCyl` is created and then rotated around the z-axis by π radians. The union of the original tool and the rotated copy form the definitions of the complete tool `supportTool`.

```
detail1 = detail 4 4 4
detail3 = detail 30 30 30
supportTool1    = blendUnion(detail1, 0.0, 1.0, 1.0,
                                  cutBox, cutCyl)
supportTool2    = rotate(rotation 0 0 1  1pi, supportTool1)
supportTool     = blendUnion(detail1, 0.0, 1.0, 1.0,
                             supportTool1, supportTool2)
```

It is also necessary to translate each piece of component geometry of the final shape to its final position ready for the application of point set combinations. The base will be called `base`, body material of the central neck support is called `supportBody`, the top cylindrical shapes are called `outerCyl` for the body material and `innerCyl` for the tool to cut the hole.

```
base         = translate(0.0, 0.0, halfBaseHeight, baseOriginal)
supportBody  = translate(0.0, 0.0, halfSupportHeight,
                                  supportBodyOriginal)
outerCyl     = translate(0.0, 0.0, supportHeight,
                                  outerCylOriginal)
innerCyl     = translate(0.0, 0.0, supportHeight,
                                  innerCylOriginal)
```

The final stage of the modelling process is to combine the point sets that are now in their correct locations to form the final geometry of the curtain pole support. The central neck `support` is created by cutting the `supportTool` away from the `supportBody`. Then the support and the base are blended together to make the lower part (`lowerPart`) of the final shape. This is then blended with the outer cylinder of the head of the shape (`noHole`) before the material from the inner cylinder is removed to make the hole in the top of the object. The final geometry of the support is called `final`.

```
lowerPart = blendUnion(detail1, 0.05, 1.0, 1.0,
                                  base, supportBody)
noHole = blendUnion(detail1, 0.03, 1.0, 1.0,
                                  lowerPart, outerCyl)
final = cut(detail3, noHole, innerCyl)
```

In addition to the geometry of the shapes, it may be necessary to attach attributes of colour and texture to the geometry and place this geometry in a VRML world where it can be interactively explored. Figure 8.12 shows the curtain pole support script rendered in a VRML browser with a wooden texture. Relevant definitions to apply this image texture to the geometry are given below.

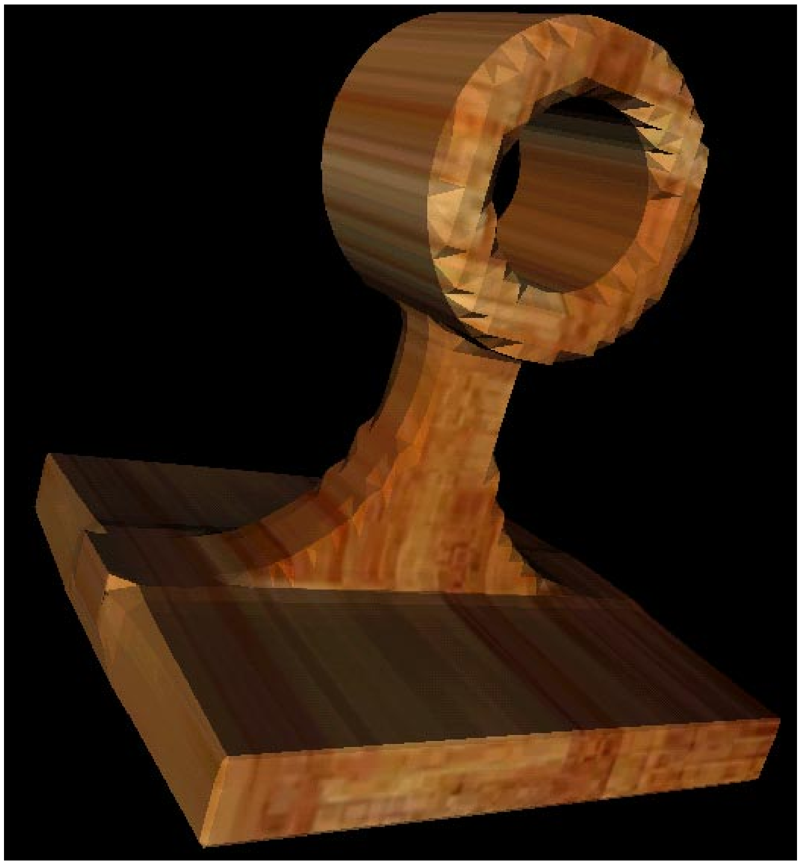


Figure 8.12: Image of the example curtain pole support (with no modifications).

```
wood      = ImageTexture { url ".../wood_floor.jpg" }
appear    = appearance( Material { } , wood)
tmp       = attribute(appear, final)
finalList = list(final)
w         = world(finalList)
```

8.5.2 Redefinitions of the Curtain Pole Support Script

The curtain pole script can be considered as a template script for a whole family of geometry with similarities to the original support. The possible versions of the geometry can be interactively explored in *empirical worlds* by making redefinitions of any identifier of a script. Three examples of redefinitions of the curtain pole support illustrated in Figure 8.12 are presented in this section.

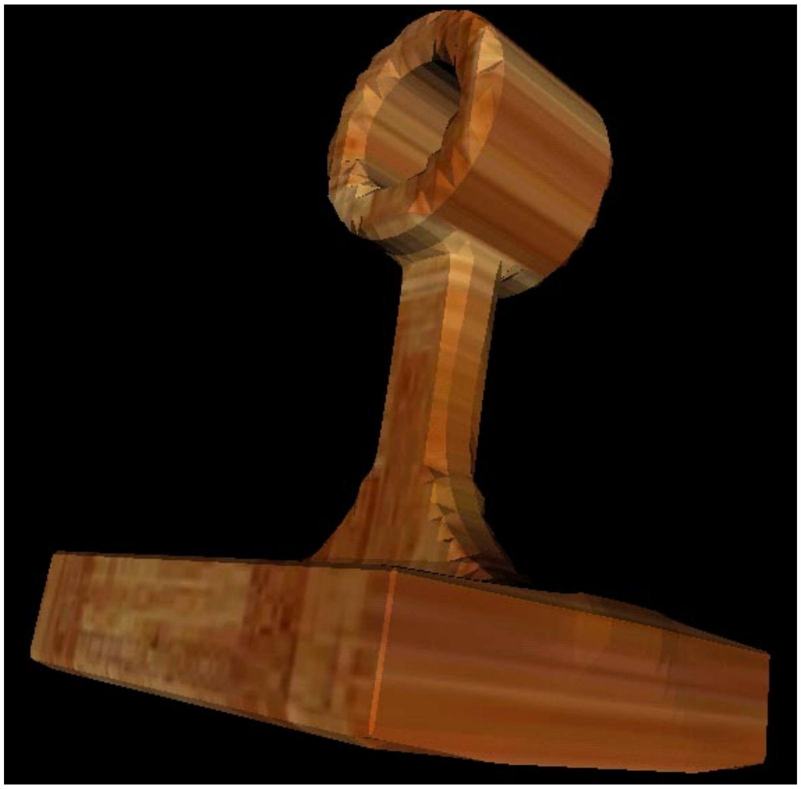


Figure 8.13: Image of the curtain pole support with an extended neck.

The conversion of the diagram of the support in Figure 8.11 into a script of definitions required the identification of some defining parameters of the three-dimensional geometry. One of these is the `supportHeight` that defines the height of the *neck* of the support that connects the cylindrical head to the box base. In a real-world location, there may be a scenario where the shape of a window required the curtains need to hang further away from the wall than with the standard part. Suitable geometry can be achieved by one simple redefinition of the *empirical world* script for the pole.

```
supportHeight = 2.0
```

The height of the pole is increased from `1.5` in the original script to `2.0` by this redefinition. The new state of the geometry is rendered in Figure 8.13.

Another way to make a new version of the support is to use it with an operation such as tapering or bending. The redefinitions shown below cause the whole geometry of the support (`final`) to be linearly tapered along the z-axis (`finalTapered`) in such a way that the cylindrical head is stretched out of shape. The redefinitions to achieve this are shown below and an image of the redefined geometry is illustrated in Figure 8.14 with a marble-like image texture applied.

```
final = cut(detail1, noHole, innerCyl)
finalTapered = taperXZ(detail3, -0.35, final)
finalList = list(finalTapered)
```

In the first of these three redefinitions, the level of detail of `final` is reduced. The reason for this is that `final` is no longer the focus of the modelling process, therefore reducing the detail level will speed up the overall response time for the system. The `finalTapered` redefinition is an implicit definition representing the tapered geometry. In the final redefinition, the tapered geometry is selected to be viewed in the VRML browser by setting it to be the only element of `finalList`. Notice how both the original geometry and the tapered geometry now exist as their own separate entities within the same script simultaneously, available for future modification or as arguments to other implicit redefinitions.

The final example of redefinition in this case-study examines the modification and replacement of a component of the geometry. In the redefinitions below, a new version of the support part is created (called `supportTwist`) that is a twisted version of the original `support` geometry along the z-axis. This new version of the support geometry replaces the original in the definition of the lower part of the geometry `lowerPart`. A close up image of the support to cylindrical head join is shown in Figure 8.15. The redefinitions associated with the twist are shown below.

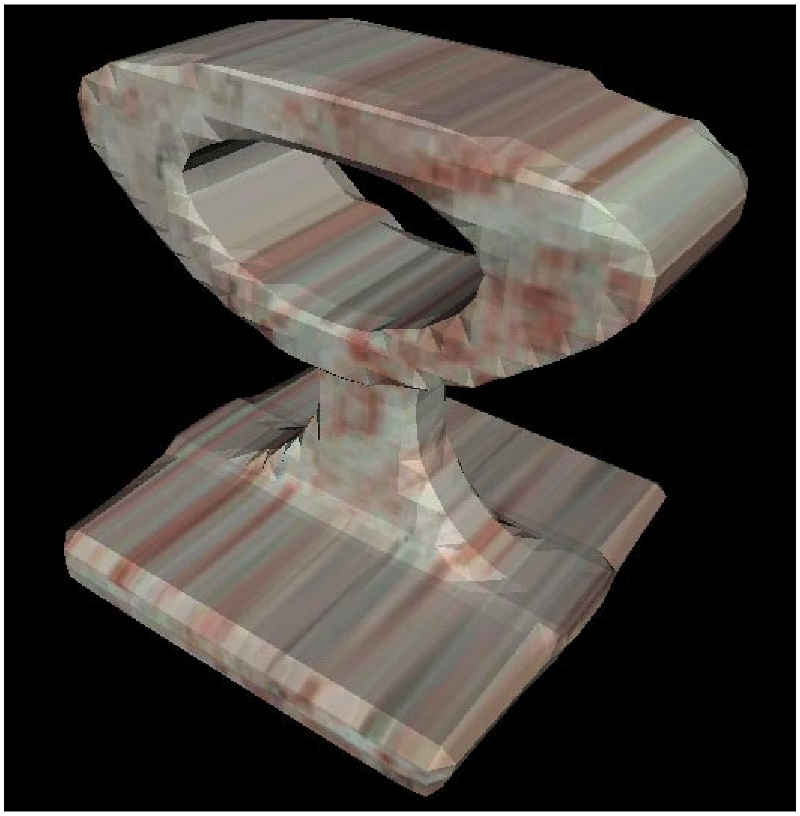


Figure 8.14: Image of the curtain pole support tapered along the z-axis.

```
final = cut(detail3, noHole, innerCyl)
supportTwist = twistZ(detail1, 0.78539, supportBody)
lowerPart = blendUnion(detail1, 0.05, 1.0, 1.0,
                              base, supportTwist)
finalList = list(final)
```

Having created the definition of `supportTwist`, it is easy to subsequently return to the original support geometry by substituting `support` for `supportTwist` in the definition of `lowerPart`.

These case-studies have shown how one piece of geometry is represented is represented in an *empirical world* script, a user can interactively modify and explore that geometry in an open-ended way. What has not been demonstrated is the ability to then use this geometry or its component parts to create other geometry. The support geometry and any of its component parts can be used in any operation (CSG, blending, morphing, bending) to build or model other geometry. In one snapshot of state, the script of definitions is a specification for shape that can be stored in a plain text file and communicated to other people by any means of textual transmission, such as e-mail, letter or computer chat program[12]. The script of definitions for the curtain pole support represents a computer-based artefact for the exploration of the geometry for a particular shape.

[12]The *Arithmetic Chat* application, presented in Section 6.5, for the interactive discussion of scripts can be used as a template for the implementation of a geometric chat application.

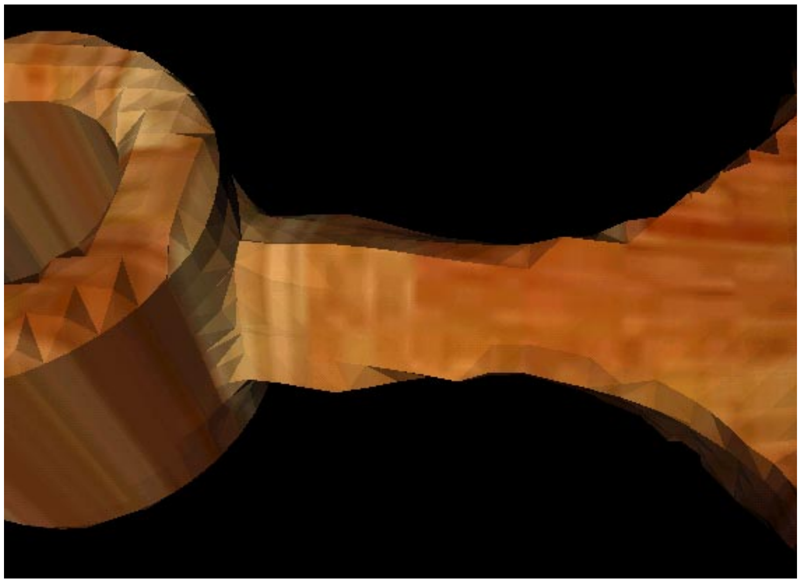


Figure 8.15: Image of the curtain pole support with a twisted neck.