Chapter 7

Introducing Empirical Worlds

7.1 Introduction

Virtual worlds constructed from three-dimensional geometric objects can be described by the *Virtual Reality Modelling Language* (VRML) [ANM96, SC96]. The language describes shape in a machine-independent way that can then be rendered and explored on a wide variety of computer input and output devices. The language inherits a lot of its design concepts from the *Hypertext Mark-up Language* (HTML) [MK97], which is a machine-independent language for the description of documents containing text, links and images. HTML is viewed on a user's computer screen through a browser or printer where the browser renders the document in a suitable way to present the document to the user on their platform. For VRML, the output devices range from conventional two-dimensional monitors through to immersive virtual reality worlds where a user wears a headset that presents the illusion of walking or flying through a virtual space.

Empirical worlds are virtual reality worlds constructed from scripts of definitions that represent geometric objects and their attributes. Definitions can be used to express observed dependencies between these objects and their defining parameters. *Empirical world* applications can be implemented in Java from a set of *empirical world classes*. They use the JaM Machine API as their underlying dependency maintenance mechanism. An *empirical world* is constructed by attaching an interface to a JaM notation called an *empirical world notation*. This definitive notation is accessed directly by the `JaM.Script.addToQ` method. The right-hand-side of explicit definitions in *empirical world scripts* have similar syntax to VRML nodes.

Empirical world classes can be integrated into applications with support for rendering three-dimensional shapes. The example application in this thesis, called *empirical world builder*, uses a VRML browser as its interface for the interactive inspection of shapes. As definitions in *empirical world* scripts are altered, the corresponding shape in the VRML browser is updated. Many VRML browsers can take advantage of computer-graphics hardware installed on the computer on which they are executing. This hardware is optimised for the display of three-dimensional shape and is to some extent independent of the main processor used for the interpretation and maintenance of dependency for *empirical world* scripts.

VRML scripts are organised with component nodes that can represent primitive shapes, materials, affine transformations, sounds and many other kinds of virtual environment data. The *empirical world* classes demonstrated in this thesis support data types for shapes and shape combinations beyond those that can be described by VRML nodes. Some VRML nodes are used as the basis for the syntax of *empirical world* notations. Not every VRML node has a corresponding *empirical world* class and there are *empirical world* classes that represent geometry that cannot be described in VRML.

Empirical worlds integrate aspects of three different types of modelling, respectively concerned with presentation, description and interaction:

Presentation VRML, including its browser support for the exploration of three-dimensional shapes;

Description the function representation of shape [PASS95];

Interaction empirical modelling using definitive scripts.

In this chapter, *empirical worlds* are introduced by describing the *empirical world* classes and how they achieve this integration. In general, this involves describing how VRML-like syntax is used in explicit definitions for the description of the parametrisations for the mathematical description of point sets using function representation. These include classes that represent *empirical world* notation data types for the primitive shapes (box, sphere, cylinder, cone), affine transformations (scaling, rotation, translation) and binary operations (set-theoretic operations, blending, metamorphosis). In addition to this, the *empirical world* operators for establishing dependency between variables of these data types by implicit definitions are described.

In Chapter 8, applications of *empirical worlds* are discussed. This includes examples of the extension of the classes to include new types and representations for shape primitives and warping transformations. The *empirical world* builder application is described and there is a case-study demonstrating the incremental construction of, and interaction with, a cognitive artefact for geometry.

7.1.1 Motivating Example

As an example of the work presented in this chapter, consider the images of a shape and the associated *empirical world* script shown in Figure 7.1. The *empirical world* script is a description of the shape of a three-dimensional letter **F**. It is parametrised by its `height` and `width`. The shape integrates three shape primitives:

- a cone (`cone1`) that forms the short arm of the letter;
- a cylinder (`cylinder1`) that forms the long arm of the letter;
- a box (`box1`) that forms the spine of the shape.

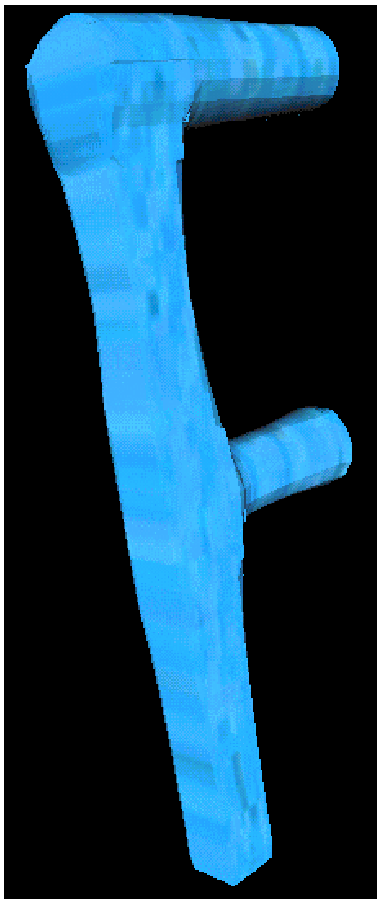
The three primitives are positioned and blended into a union of point sets called `blend1`.

The image of the letter **F** is rendered with a water texture applied in Figures 7.1a and 7.1b. These two images correspond to the script shown beneath the figure. The two images are different views of the same geometric shape, where Figure 7.1b is a *zoomed in* view of the shape to show the detail of the blend between the cylinder and the box. Any definition in the script can be interactively redefined, this will update the representation of the geometry accordingly. Figure 7.1c shows an image of the shape following the redefinition of `height` to a new value of `1.0`[1].

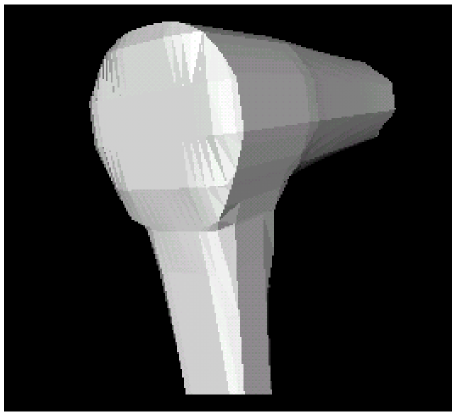
7.2 Empirical World Classes

Empirical worlds can be constructed with the use of *empirical world* scripts that control the instantiation and relationship between objects of *empirical world* classes. These scripts are in a JaM notation that is based on the *empirical world* classes described in this section and Sections 7.3, 7.4, 8.2, 8.3 and Appendix A.4. The majority of the *empirical world* classes are shown in the class diagram of Figure 7.2. The sections of this chapter and Chapter 8 describe an open-ended modelling environment for the creation and exploration of geometric shape on a computer system. The classes are combined in the *empirical world builder*, a server/client definitive environment that supports this open-ended interactivity within a Java applet that

[1]The texture of the image has also been redefined so that a pine texture has been rendered on the surface of the shape.



a



b

c

```
height          = 5.0
width           = 2.0
coneHeight      = multiply(width, 0.6)
coneRadius      = multiply(height, 0.05)
cone1           = cone(coneHeight, coneRadius)
cylinder1       = cylinder(width, coneRadius)
moveCylinder    = multiply(height, 0.5)
moveExtrusions  = multiply(width, -0.4)
transCylinder   = translate(moveCylinder, 0.0, 0.0, cylinder1)
listExtrusions  = list(cone1, transCylinder)
transExtrusions = translate(0.0, moveExtrusions,
                                0.0, listExtrusions)
box1            = box(height, coneRadius, coneRadius)
detail1         = detail 35 25 25
blend1          = blendUnion(detail1, 0.06, 1.0, 1.0,
                                box1, transExtrusions)
water           = ImageTexture { url ".../water.jpg" }
appear1         = appearance( Material { } , water)
attr1           = attribute(appear1, blend1)
```

Figure 7.1: Script and rendering of a letter **F** using *empirical worlds*.

Implicit	`l = 2.0`
	`w = 2.0`
	`h = 2.0`
	`b = box(l, w, h)`

Explicit	`b = Box { size 2.0 2.0 2.0 }`

Table 7.1: Implicit and explicit expressions for a box.

communicates with an embedded VRML browser. This tool is documented in Section 8.4.

In the diagram of classes shown in Figure 7.2, each box corresponds to a class and contains the name of that class. Inheritance between objects is shown from left to right. A class directly connected to and on the right of another class is an extension of the class on the left. Objects shown in boxes with dashed borders are *abstract classes* [CH96]. This means that they cannot be instantiated as new objects during execution as they contain *abstract methods* which have no code in their statement body. These methods must be implemented by all subclasses of the abstract class. All classes in solid boxes can be instantiated as new objects.

The class called `Object` is the `java.lang.Object` that every Java class inherits by default. Inheriting directly from `Object` is the abstract class `JaM.DefnType` that every JaM data type must extend and the abstract class `JaM.DefnFunc` that every JaM operator used for dependency maintenance must extend (see Section 6.2). For most classes that extend `DefnType`, a corresponding class or classes exist that extend `DefnFunc`. These are used to implicitly define variables of the corresponding type in an *empirical world* script. For example, a box centred at the origin with height, width and depth of `2.0` can either be described by a string expression of its explicit value, or by an implicit definition of a box with three floating point values of `2.0`. Table 7.1 shows two expressions for the same geometric shape, where the first is implicitly dependent on the values of `l`, `w` and `h` and the second has an explicit value.

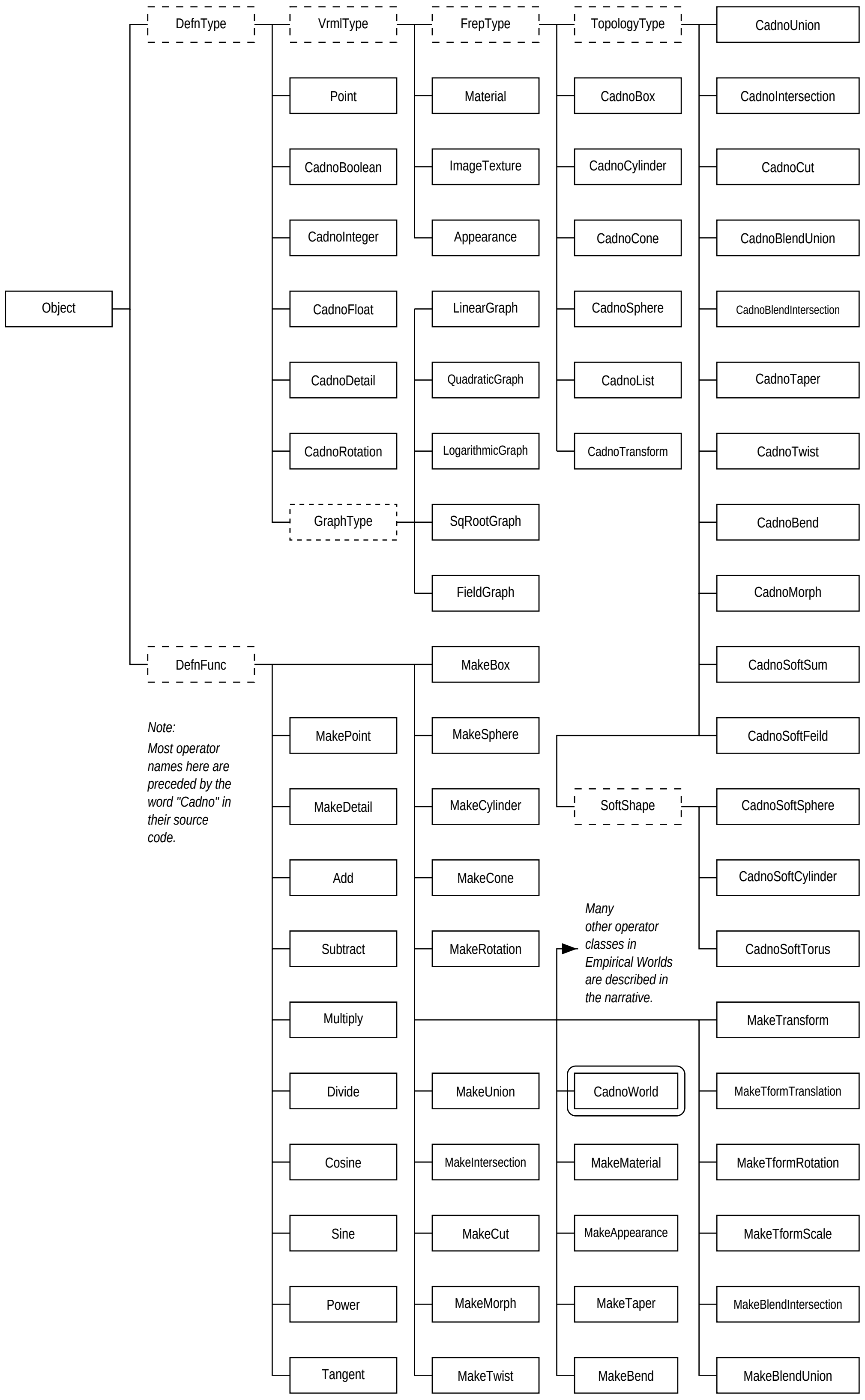


Figure 7.2: Class diagram for the *empirical world* classes.

Classes that directly extend `DefnType` in *empirical worlds* represent data types that have no node to represent them in VRML. These are the basic types of an *empirical world* script, including standard data types for booleans, integers, floating point numbers and three-dimensional points. These are described further in Appendix A.4. Some specialised basic types exists that are specific to *empirical worlds*, including a *rotation type* (`CadnoRotation`) that represents an axis of rotation and an angle of rotation in one data value (see Appendix A.4).

The types `LinearGraph`, `QuadraticGraph`, `LogarithmicGraph`, `SqRootGraph` and `FieldGraph` are classes that extend the abstract class `GraphType`. These types are used to represent single valued continuous functions that map one real number to another. These are required where a definition depends on the shape of a continuous function, such as field functions. They are described further in Appendix A.5. Each function can be sampled at any floating point value x using the built-in operation "`readGraph`" and return a floating point value y that is dependent on x and the graph. The following script will define the value of $y = 2x + 4$:

```
lg = LinearGraph { xCoefficient 2 constant 4 }
y = readGraph(lg, x)
```

Classes that extend `VrmlType`, an abstract class that directly extends `DefnType`, must have a *node* representation in VRML that can be described by a string. For any instance of a class that extends `VrmlType`, the VRML string to describe the instance is generated by a method of the object called `virtualise()`[2]. This method returns a `java.lang.String` object that is a VRML-2 [SC96] node description consistent with the internal data structures of the object instance. Each instance of an *empirical world* class can be described in a VRML-2 file with the string representation generated by this method.

[2]Throughout this chapter, some words in `typewriter` font correspond to identifiers in the *empirical world* classes source code. The name for the `virtualise()` method is chosen to signify that it converts internal object data representations into a representation in the *Virtual* Reality Modelling Language (VRML).

Classes that directly extend `VrmlType` typically represent data types for attributes of worlds, such as materials and textures[3]. For these classes, there is an exact correspondence between the value on the right-hand-side of an explicit definition and the string returned by the `virtualise()` method. For classes that indirectly extend `VrmlType`, if a direct VRML-2 representation is possible then this representation is returned by the `virtualise()` method. If it is necessary to render a more complex shape than one for which a representation exists in VRML-2, a polygonisation or approximation to the represented shape must be provided. This mechanism is encapsulated in classes that extend `FrepType`, as described below.

Any object that extends the class `FrepType`, which itself directly extends `VrmlType`, is an object that describes some solid geometry that can be displayed and explored in a VRML browser. The primitive shape types are defined in *empirical worlds* as `CadnoBox` for a VRML `Box` node, `CadnoCone` for a VRML `Cone` node and so on. In this chapter, new syntax is introduced to describe values on the right-hand side of explicit definitions where no suitable VRML syntax is available. Each object that extends `FrepType` must implement a method `f()` that is the *function representation*[4] for a point set of the geometric shape represented by an instance of the object. This method should return a floating point value for any three float arguments `(x, y, z)` representing a point in space consistent with the condition below.

$$\texttt{f(x, y, z)} \begin{cases} < 0 & \text{if } (x,y,z) \text{ is outside the solid object.} \\ = 0 & \text{if } (x,y,z) \text{ is on the surface of the solid object.} \\ > 0 & \text{if } (x,y,z) \text{ is inside the solid object.} \end{cases}$$

The VRML-2 notation supports the affine transformation of nodes. These are provided by the `Transform` node described in Section 7.3.6. If shapes other

[3]Only a few simple classes are implemented that directly extend `VrmlType` at this time. Future work on *empirical worlds* should extend these to include all VRML-2 nodes.

[4]See Section 3.4.2 and [PASS95].

than the primitive VRML shapes are required by the designer of a world, they must either transform the nodes to appropriate positions and overlap them[5], or calculate a polygonisation of the complex geometric object. Such a polygonisation is a set of edge-connected, filled, three-dimensional triangles oriented in space so that they approximate the surface of the solid object. Sets of triangles can be rendered efficiently by modern computer graphics hardware that is optimised for displaying these triangulations [WND97].

To enrich the range of geometric shapes available in *empirical worlds*, new transformations and operators on point sets are introduced based on the function representation of the objects. This is combined with a generic polygonisation algorithm for all classes that extend `TopologyType`. This abstract class directly extends `FrepType` and implements a `virtualise()` method that calculates a polygonisation from the function representation of its subclasses. Classes that extend `TopologyType` use the function representation binary operators for set-theoretic operations and blends to construct new geometric shapes. The polygonisation algorithm generates a `String` of VRML containing all the triangles that approximate the new geometric shape. All classes that extend `TopologyType` also extend `FrepType` and so they must implement the method `f()` and have their own function representation.

In the following section, controlling the rendering level for the polygonisation algorithm using the `CadnoDetail` data type and the `CadnoMakeDetail` operator is described. As well as being an important introduction to the process of controlling efficient interaction with *empirical world* scripts, this data type and operator is used to illustrate the syntax of the descriptions given for other data types and operators throughout this chapter, Chapter 8 and Appendices A.4 and A.5. Polygonalisation

[5]In most VRML browsers, this produces the effect that the objects appear to be the union of point sets for primitive shapes.

is only used when necessary and if a geometric object is composed of primitives available in VRML then the visualisation output consists of appropriate VRML rather than a list of polygons. Section 7.3 describes classes that represent the primitive geometry available within VRML, including the affine transformations of VRML. In Section 7.4, binary and n-ary operations on point sets for set-theoretic operations, blending and morphing are described.

7.2.1 Rendering Detail

When an object that is a subclass of `TopologyType` is rendered by a polygonisation of its shape, the detail to which that polygonisation is carried out can be controlled by an associated `CadnoDetail` object. The three numbers following `detail` in the string value of the data type correspond to the number of space-dividing voxels that a function representation `f` of a geometric shape is split into for sampling, with edges parallel to the x, y and z axis of the space. The polygonalisation algorithm in the *empirical worlds* softare implements cubic cell polygonalisation in a similar way to that described in [BBCG+97] and further details of the polygonalisation algorithm can be found in Section 8.4.3.

In this thesis, a table with a single line above and below its contents is used to represent *empirical world* classes that extend `JaM.DefnType` (see Section 6.2.2). These tables illustrate the integration of VRML-like syntax into *empirical worlds.* The table for the `CadnoDetail` class is shown below:

Class Name	`CadnoDetail`
Extends	`DefnType`
Value Format	`detail` *integer integer integer*
Default Value	`detail 10 10 10`

In this table, the rows describe the following information:

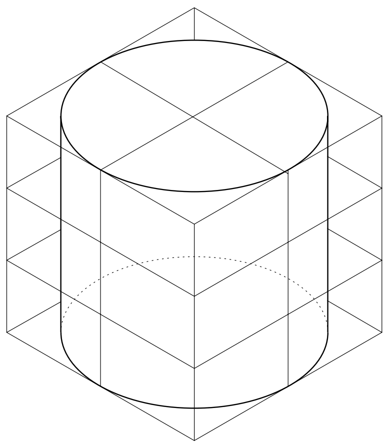


Figure 7.3: Voxels surrounding a cylinder point set.

Class Name The name of the *empirical worlds* class.

Extends The name of the class that the class represented directly extends.

Value Format A representation of the syntax for the description of a value of the data type for the right-hand-side of an explicit definition in an *empirical world* script. Italicised tokens should be replaced by appropriate values in an *empirical world* script.

Default Value If the value of a variable is not explicitly given by an explicit definition, or an abbreviation is available for describing a value, then this is the default value associated with an instance of the data type.

Figure 7.3 shows a cylinder contained in twelve voxels. For this example, the value represented by the instance of the `CadnoDetail` class is "`detail 2 2 3`". The bounding box of the geometry has been split into two in the x direction, two in the y direction and 3 in the z direction. The polygonisation algorithm for this object would sample the function `f` for the cylinder at the intersection and corner points of the voxel describing thin lines in the figure, including those not visible due to the solid material of the cylinder.

An operator called `CadnoMakeDetail` is available in *empirical world* scripts to define the value of a `CadnoDetail` to be dependent on other *integer* values. Operators are represented by *empirical world* classes that extend `JaM.DefnFunc` (see Section 6.2.3 for more details). This operator allows the user to indivisibly link a defining parameter for some geometry to the level of detail at which an object is rendered. The first argument to the operator is the number of voxel divisions for the bounding box parallel to the x-axis, the second is the number of divisions along the y-axis and the third is the number of divisions along the z-axis.

In this thesis, a table with double-lines above and below its contents is used to describe an operator class that extends `JaM.DefnFunc`. These tables represent the way in which dependencies can be expressed between VRML-like values through implicit definitions in *empirical worlds*. The table representing the `CadnoMakeDetail` class is shown below:

Operator Class		`CadnoMakeDetail`
Maps From		*integer* × *integer* × *integer*
Maps To		`CadnoDetail`
Example	Defn.	`d1 = detail(10, 20, 23)`
	Value	`d1 = detail 10 20 23`

In this table, the rows describe the following information:

Operator Class The name of the *empirical world* class that represents an operator in the *empirical world* notation.

Maps From Description of the data types possible in the sequence of arguments.

Maps To Data type associated with the identifier on the left-hand side of an implicit definition that is based on the operator.

Example An example of the operators use, split into:

Defn. an implicit definition containing the operator;

Value an explicit definition that corresponds to the example implicit definition, as it associates the same value with the identifier on the left-hand side.

The implicit definition "`d1 = detail(10, 20, 30)`" defines the value associated with identifier `d1` to be the same as the explicit definition "`d1 = detail 10 20 30`".

7.3 Primitive Shapes in Empirical Worlds

All primitive shapes in *empirical worlds* are data types represented by subclasses of `FrepType`. They have both a VRML string representation of a VRML node and also a function representation corresponding to the equivalent point set. The primitive solid geometric types presented in this section are `Box`, `Cylinder`, `Cone` and `Sphere`, the same as the primitive shape types in VRML.

All primitive shapes in VRML and *empirical worlds* are defined so that their dimensions are centred at the origin[6]. To move a primitive away from the origin, it is necessary to use the affine transformation data type (an instance of `CadnoTransform`) that represents transformed geometry. The `CadnoTransform` data type allows the combination of any rotations, scaling, reflections, translations and linear shears on any list of geometric objects represented by classes that extend `FrepType`.

As the `CadnoTransform` can be exactly described by a string representing a VRML node, it is treated along with the primitive shapes and the `FrepList` in this section as a subclass of `FrepType` for which the `virtualise()` method returns

[6]The origin is not necessarily the centre of gravity for a solid. Consider a `Cone` primitive with more material below the origin than above the origin.

```
Shape {
  appearance Appearance {
    material Material {
      diffuseColor 0.2 0.8 0.6
    }
  }
  geometry Transform {
    rotation 0 0 1  1.57
    children [
      Box { size 1.0 2.0 3.2 },
      Cylinder { radius 3.1 height 4.5 },
      Cone { bottomRadius 2.1 height 3.3 },
      Sphere { radius 9.3 }
    ]
  }
}
```

Table 7.2: VRML-2 file containing *primitive shape* nodes.

a string of VRML that does not contain a polygonisation of a shape. The way in which classes in this section are rendered is determined by the VRML browser in which their shape is viewed. Table 7.2 shows an example VRML-2 file containing all the primitives documented in this section of the chapter, along with an `Appearance` node that is a special attribute node described further in Appendix A.4.

7.3.1 Box Point Sets

A box point set has six rectangular faces, eight vertices and is parametrised by its length (`a0`) along the x-axis, width (`a1`) along the y-axis and height (`a2`) along the z-axis. Each face is parallel to either the $x = 0$, $y = 0$ or $z = 0$ planes. A diagram of a box with parameter "`size a0 a1 a2`" is shown in Figure 7.4. In *empirical worlds*, the centre of the material contained within the box is the origin. Every instance of a subclass of `FrepType` has a field to represent a bounding box. The minimum coordinate of a `Box` in each dimension is $(\frac{-a0}{2}, \frac{-a1}{2}, \frac{-a2}{2})$ and the maximum coordinate is $(\frac{a0}{2}, \frac{a1}{2}, \frac{a2}{2})$. These coordinates define the bounding box for `Box` shapes.

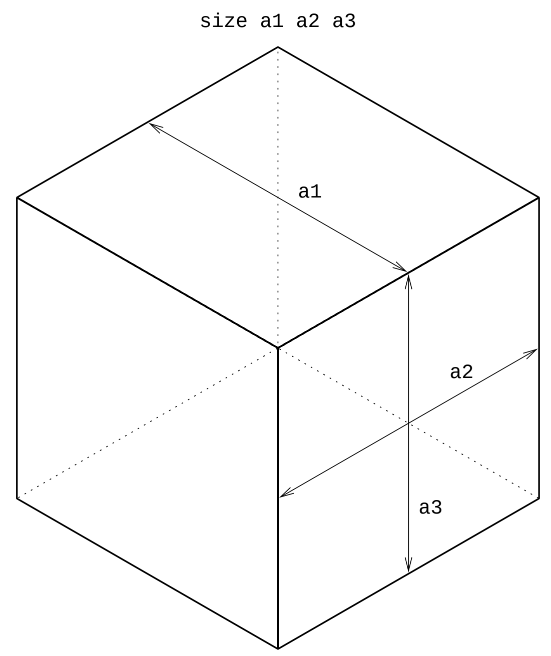


Figure 7.4: Diagrammatic representation of a `Box` point set.

Class Name	`CadnoBox`
Extends	`FrepType`
Value Format	`Box` { `size` *point* }
Default Value	`Box` { `size 2 2 2` }
Parameters	Name Type
	`size` `Point`

In an explicit definition for a box, there can be optional parameters between curly braces "{" and "}". All primitive shape types in *empirical worlds* have a **Default Value** for all defining parameters. The deafult value corresponds to the value of a parameter if it does not explicitly appear between the curly braces in an explicit definition in an *empirical world* script. In the case of the `Box`, the expression of a value "`Box` { }" is equivalent to the expression "`Box` { `size 2 2 2` }"[7]. The data type of the parameters that appear within the curly braces is shown in the **Parameters** section of the table, where the parameter *Name* is next to its *Type* in

[7]This is the same convention as in VRML, where most nodes have a default value.

terms of other *empirical world* classes. The grammar that matches this data type identifies the parameter from its string.

Operator Class		`CadnoMakeBox`
Maps From		*float* $\times$ *float* $\times$ *float* or *integer* $\times$ *integer* $\times$ *integer* etc.
Maps To		`CadnoBox`
Example	Defn.	`b1 = box(3.2, 4, 5.6)`
	Value	`b1 = Box { size 3.2 4 5.6 }`

Class `CadnoMakeBox` describes an operator that takes three arguments that represent the length, width and height of a box and returns an instance of a `CadnoBox` class with these dimensions, centred at the origin. This is the mechanism for establishing dependencies between boxes and other geometry. It is impossible to have a box with negative length sides and so the absolute value is taken for any negative parameters for a box during an update.

The function representation of a box point set is defined as the intersection of six *half spaces*. For any point (x, y, z) a possible function representation for a box f is given in the formulae

$$h(u,v) = -\left(u - \frac{v}{2}\right)\left(u + \frac{v}{2}\right) \tag{7.1}$$

$$i(u,v) = u + v - \sqrt{u^2 + v^2} \tag{7.2}$$

$$f(x,y,z) = i(i(h(x,a0),h(y,a1)),h(z,a2)) \tag{7.3}$$

The mapping $h(u, v)$ is the function representation for the intersection of two half spaces, with normal vectors oriented along the u-axis, separated by distance v and equidistant from the plane $u = 0$. The mapping $i(u, v)$ is a function representation for the intersection of two other function representation values at the same point u and v.

The function implemented in the method `f` for the `CadnoBox` object has C^1 discontinuity only at the edges of the box. Smoother geometry and aesthetically pleasing images are created for operators that combine point sets, such as blends, for function representations that are C^1 continuous everywhere except shape edges.

7.3.2 Sphere Point Sets

Sphere point sets are centred on the radius and parametrised by a radius `r`. The bounding box for a sphere is given by the minimum point of $(-r, -r, -r)$ and the maximum point (r, r, r). Figure 7.5 shows a diagrammatic representation of a sphere point set, where the three small arrows at its centre point are oriented along the three axes of the space. (Each arrow has the end point of its tail located at the origin.)

Class Name	`CadnoSphere`	
Extends	`FrepType`	
Value Format	`Sphere` { `radius` *float* }	
Default Value	`Sphere` { `radius 1.0` }	
Parameters	Name	Type
	`radius`	`CadnoFloat`

Class `CadnoMakeSphere` extends `DefnFunc` and allows for the implicit definition of `CadnoSphere` instances in *empirical worlds*.

Operator Class		`CadnoMakeSphere`
Maps From		*float* or *integer*
Maps To		`CadnoSphere`
Example	Defn.	`s1 = sphere(3.7)`
	Value	`s1 = Sphere` { `radius 3.7` }

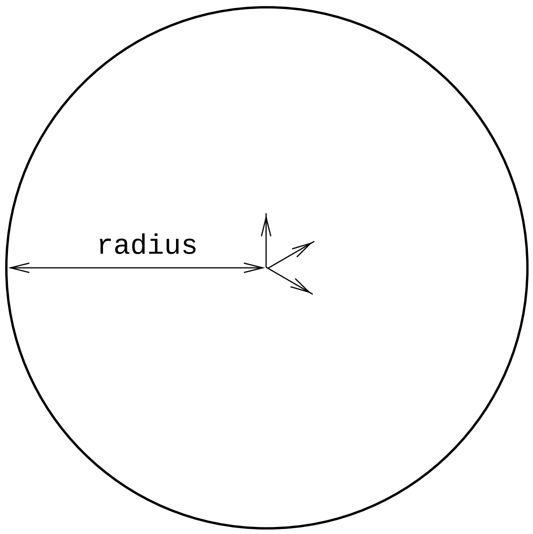


Figure 7.5: Diagrammatic representation of a `Sphere` Point Set

The implemented function representation for a `CadnoSphere` object for any point (x, y, z) in three-dimensional space with radius r is given by the equation.

$$f(x, y, x) = r^2 - (x^2 + y^2 + z^2) \tag{7.4}$$

This function has C^1 continuity everywhere.

7.3.3 Cylinder Point Sets

Cylinder point sets are parametrised by their height (`h`) and radius (`r`). In *empirical worlds* and VRML, the central axis along which the height is measured, is oriented along the y-axis. The cylinder is centred at the origin and has a bounding box with minimum point $(-r, \frac{-h}{2}, -r)$ and maximum point $(r, \frac{h}{2}, r)$. A diagrammatic representation of a cylinder centred at the origin is shown in Figure 7.6.

Class Name	`CadnoCylinder`	
Extends	`FrepType`	

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

Default Value	`Cylinder {`
	`height 2.0`
	`radius 1.0`
	`}`	

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

Two parameters are used in an explicit definition of a cylinder. The order in which they appear is not important when the string representing a value is parsed to create and instance of a `CadnoCylinder`. If one of the parameters is missing then the default value for that one parameter is assumed. For example, "`Cylinder { height 4.5 }`" is equivalent to "`Cylinder { height 4.5 radius 1 }`". If an *integer* value is given for one of the parameters, it is converted from the internal representation of the cylinder to a floating point value.

Class `CadnoCylinder` represents an operator in *empirical world* scripts for the implicit definition of cylinder shapes. There are two arguments to this operator — the first represents the height of the cylinder and the second represents the radius.

Operator Class		`CadnoMakeCylinder`
Maps From		*float* × *float* or *integer* × *integer* etc.
Maps To		`CadnoCylinder`
Example	Defn.	`c1 = cylinder(3.2, 3.4)`
	Value	`c1 = Cylinder { height 3.2 radius 3.4 }`

A function representation for a cylinder centred at the origin with central

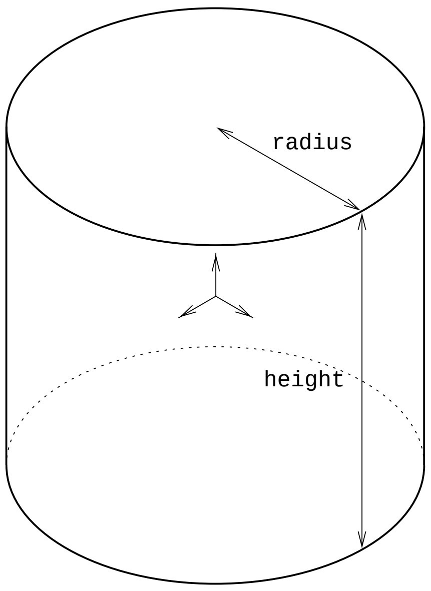


Figure 7.6: Diagrammatic representation of a `Cylinder` point set.

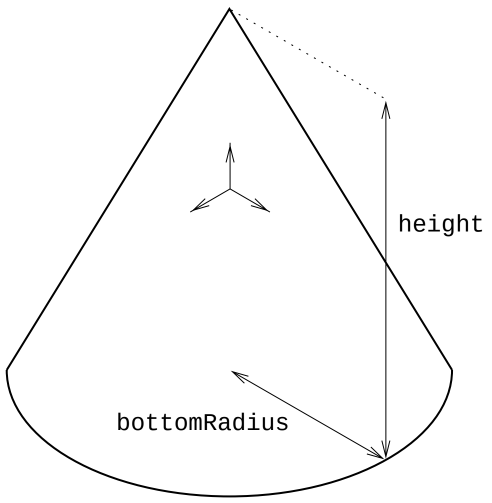
axis set as the y-axis can be defined by the intersection of an infinite cylinder and two half spaces with normal vectors along the y-axis. The function representation f for a cylinder at for any point (x, y, z) is

$$\begin{aligned} f(x,y,z) &= \Big(r^2 - (x^2+z^2)\Big) + \Big(-(y-\frac{h}{2})(y+\frac{h}{2})\Big) \\ &\quad -\sqrt{(r^2-(x^2+z^2))^2 + \Big(-(y-\frac{h}{2})(y+\frac{h}{2})\Big)^2} \end{aligned}$$

This function representation is C^1 continuous except at the edges of its top and bottom faces.

7.3.4 Cone Point Sets

Cone points sets are parametrised by a radius (`r`) for the bottom circle of the cone and a height (`h`) with its central axis oriented along the z-axis. The axis of the cone is centred on the origin and has its top summit point located at $(0, \frac{h}{2}, 0)$. The minimum point of the bounding box for the point set is $(-r, \frac{-h}{2}, -r)$ and the



maximum point is $(r, \frac{h}{2}, r)$. A diagrammatic representation of a cone is shown in Figure 7.3.4, where the `height` and `bottomRadius` parametrisations are indicated.

Class Name	`CadnoCone`	
Extends	`FrepType`	

Value Format	`Cone {`
	`bottomRadius` *float*
	`height` *float*
	`}`	

Default Value	`Cone {`
	`bottomRadius 1.0`
	`height 2.0`
	`}`	

Parameters	Name	Type
	`bottomRadius`	`CadnoFloat`
	`height`	`CadnoFloat`

Class `CadnoMakeCone` extends `DefnFunc` and represents an operator that allows an instance of `CadnoCone` to be implicitly defined in an *empirical world* script. The first argument to the operator is the `bottomRadius` of the cone and the second argument is the `height`.

Operator Class		`CadnoMakeCone`
Maps From		*float* × *float* or *integer* × *integer* etc.
Maps To		`CadnoCone`
Example	Defn.	`c2 = cone(2.3, 7)`
	Value	`c2 = Cone { bottomRadius 2.3 height 7 }`

The function representation f of a cone with height h and bottom radius r can be defined as the intersection between two half spaces and an infinite cylinder whose radius varies in linear proportion to the y-coordinate of the sampling point. In the formulae for the function representation of a cone below: mapping *rad* is used to calculate the varying radius, mapping *half* returns the half spaces used for the intersection and mapping *cyl* is an infinite cylinder along the z-axis with a varying radius. For any point (x, y, z), the cone f is given by the formulae

$$\begin{aligned}
rad(y) &= \frac{-ry}{h} + \frac{r}{2} \\
cyl(x, z, q) &= q^2 - (x^2 + z^2) \\
half(y) &= -(y - \frac{h}{2})(y + \frac{h}{2}) \\
f(x, y, z) &= half(y) + cyl(x, z, rad(y)) - \sqrt{half(y)^2 + cyl(x, z, rad(y))^2}
\end{aligned}$$

The function representation f for the cone has C^1 discontinuity only at the edge of its base and at its summit point.

7.3.5 Lists of `FrepType` Geometric Objects

In VRML, the `Group` node allows nodes of various kinds to be grouped into one node. An example of a group node is shown in Table 7.3 where a `Box`, `Cylinder` and `Cone` have been grouped into one VRML node by placing them in a comma separated list between "`children` [" and "]". Many other nodes in VRML, such as the `Transform` node in the script in Table 7.2, use lists starting with the token

```
Group {
  children [
    Box { size 2 3 4.2 }, Cylinder { },
    Cone { bottomRadius 3 }
  ]
}
```

Table 7.3: A grouping node in VRML-2.

"`children`" as a parameter for grouping the internal structure of the node. Groups of nodes with interfering points sets are typically drawn one on top of the other in a VRML browser. The resulting virtual-world objects look like a set-theoretic union, although if elements of the group have different appearances or textures specified then this can lead to a visually unsatisfactory model.

In *empirical worlds*, it is not possible to place any node in a list in the way that it is in a `Group` node in VRML. A distinction is made between data type classes that extend `FrepType` and other classes that extend `DefnType` without extending `FrepType`. This is so that it is clear which data types correspond to graphically representable solid geometry and which data types are related to texture, appearance, sound, viewpoint and so on. The `FrepList` is a special kind of list that can contain only nodes (*empirical world* notation types) that extend `FrepType`[8]. This also allows operators that map from shapes described by function representations to map from a list[9].

Class Name	`FrepList`
Extends	`FrepType`
Value Format	`children` [(*FrepType*)*]
Parameters	A list of objects that extend `FrepType`.

[8]Lists in *empirical worlds* are not comma-separated as they are in VRML.

[9]This limitation of *empirical worlds*, which is implementation specific, should be a future topic for research and improvement.

This class illustrates an important feature of the JaM Machine API. A list of geometric shapes has an explicit value and is its own data type, for example "`children [ Box { } Cylinder { } ]`". Whether the elements of the list explicitly defined or implicitly defined by the operator represented by class `CadnoMakeList`, the value associated with the left-hand side identifier can be represented by a unique string. Further operators for lists (`CadnoHead`, `CadnoTail` and `CadnoElementAt`), are described in Appendix A.4.

Operator Class		`CadnoMakeList`
Maps From		(`FrepType`)*
Maps To		`FrepList`
Example	Defn.	`l1 = list(Box { size 1 2 3 }, Cylinder { })`
	Value	`l1 = children [ Box { size 1.0 2.0 3.0 } Cylinder { height 2.0 radius 1.0 } ]`

The function representation of an `FrepList` f is the maximum value of the function representation for all the elements of the list $g_1, \ldots, g_n$ at any point (x, y, z), as described by the equation

$$f(x, y, z) = \max(g_1, \ldots, g_n) \tag{7.5}$$

The function representation f represents the set-theoretic union of all the point sets represented by the function representations in the list. Note that the intended use of lists is to group shapes together and not to construct set-theoretic unions. This shape representation of this class can produce similar results to the geometric object overlap that occurs in VRML groups, with the function f having C^1 continuity everywhere except locations where the component function representations have equal values[10]

[10]Detail inside a solid shape is preserved by VRML groups and lost in *empirical world* lists.

7.3.6 Affine Transformations of Point Sets

All the primitive shape points sets in VRML and *empirical worlds* are constructed centred at the origin. A virtual world full of primitive shape point sets centred at the origin would not be a particularly interesting one. In the real world, objects are centred in many different locations. An affine transformation of any geometric solid object allows for its point set to be relocated anywhere in virtual space, rotated around any axis and scaled in a particular orientation to an appropriate size. VRML has a `Transform` node that can be used for the combined purpose of rotation, translation and scaling for any piece of geometry[11]. The analogue to this node in *empirical world* classes is an instance of `CadnoTransform`.

This section explains the process of integrating the VRML `Transform` node syntax with the function representation of shape in a definitive script. This explanation is presented in the form of an annotated transformation of a two-dimensional hexagonal polygon.

Figure 7.7 shows the effect of a set of affine transformations on a hexagon in two dimensions. The transformations shown are carried out in the same order as the transformations in both VRML and instances of the *empirical world* `CadnoTransform` class. Vector $\mathbf{c}$ is the central point for the transformations, vector $\mathbf{t}$ is the translation vector of the geometric objects, $\mathbf{O}$ is a matrix representing a rotation prior to a scaling, $\mathbf{S}$ is a diagonal matrix for the actual scaling with $\mathbf{c}$ at its centre and $\mathbf{R}$ is a rotation matrix representing the required rotation of the object about centre $\mathbf{c}$[12].

In the diagram in Figure 7.7, each stage of the transformation is represented by separate pairs of axes that are labelled "`a`" through to "`g`". On each pair of axes, the hexagonal shape is drawn with a solid line to show it prior to transformation, and

[11]See Table 7.2 for an example of a VRML `Transform` node.

[12]Rotation in two dimensions is about a point, not an axis as in three dimensions. The direction of the axis of rotation in three dimensions is contained in the internal representation of $\mathbf{R}$.

with a dashed line to illustrate the effect of the transformation. Each transformation is described by items in the list below, where each item is labelled to correspond with Figure 7.7:

- a - The point set (hexagon) is translated by vector **-c** so that the other transformations centred at the origin can take place as if they were centred at **c**. The rotated and scaled point set is transformed back by vector **c** in step f.
- b - Rotation of the point set by the inverse scale orientation matrix $\mathbf{O}^{-1}$ prior to scaling. In the example, $\mathbf{O}^{-1}$ corresponds to an anti-clockwise rotation around the origin by $\frac{\pi}{2}$ radians.
- c - Linear scaling of the point set by a factor of 1.5 along the x-axis and 2 along the y-axis. This scale transformation can be represented by a diagonal matrix **S**.
- d - Rotation of the point set by the scale orientation matrix **O**.
- e - Rotation of the point set by the rotation matrix **R**. In this example, **R** corresponds to a clockwise rotation about the origin by $\frac{\pi}{3}$ radians.
- f - Translation of the point set from the origin back to the centre of the transformations at point **c**.
- g - Translation of the point set by the given translation vector **t**.
- h - The point set in its new transformed location after all the combined affine transformations are completed.

A bounding box for a translated shape is found by translating the bounding box for the transformed geometric objects and then creating a new box with faces with normal vectors parallel to the axes. This new bounding box completely

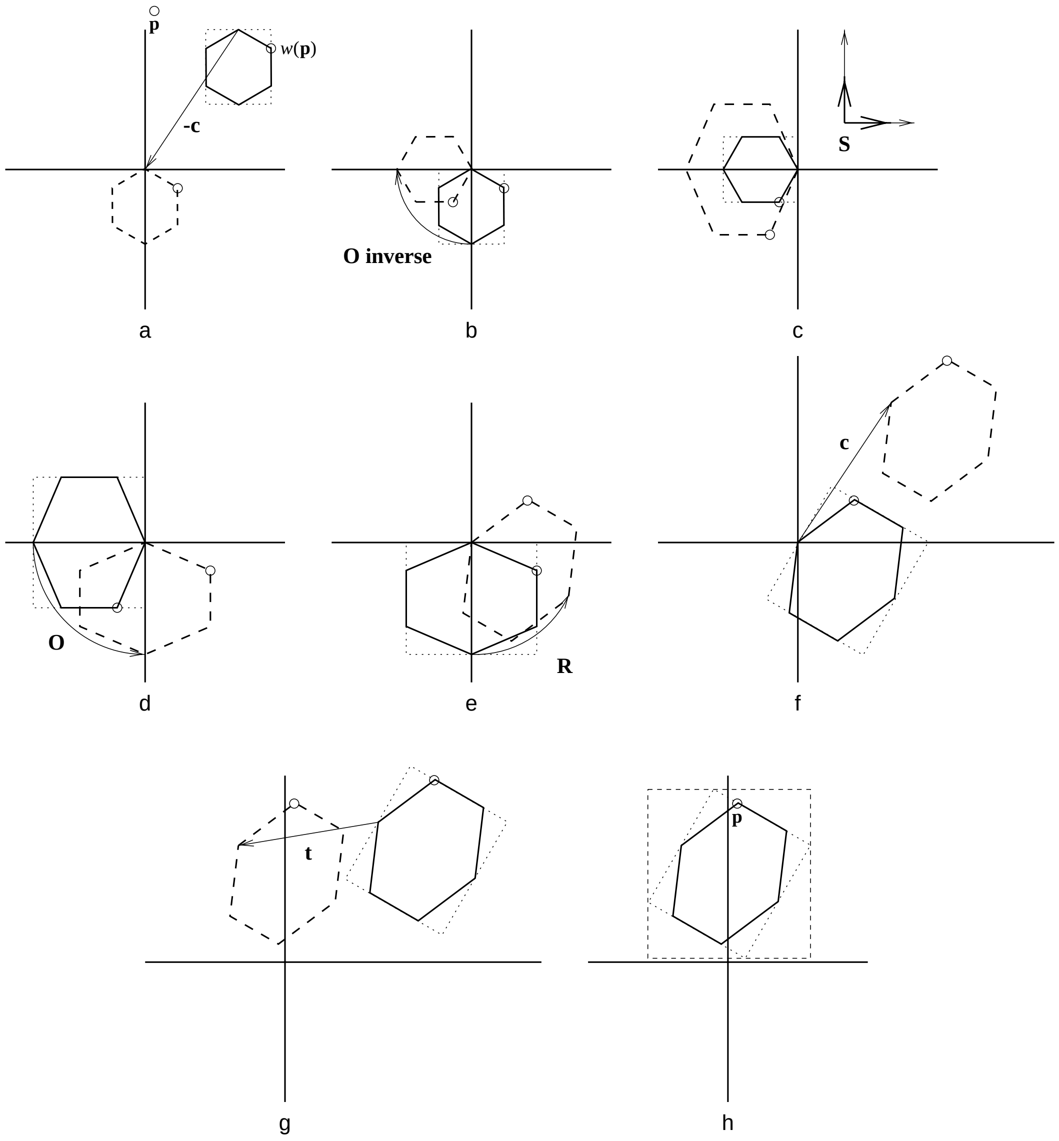


Figure 7.7: Stages for an affine transformation of a hexagonal shape.

contains the translated bounding box. In Figure 7.7, this process is illustrated in two dimensions. In Figure 7.7a, the dotted box represents the original bounding box for the hexagon. The translation of this bounding box is shown in Figures 7.7b through to 7.7g. In Figure 7.7h the new bounding box for the translated geometry is calculated, with sides that have normal vectors along the x and y axis. This new bounding box is represented by a box drawn with a dashed outline.

Bounding boxes constructed by this process are often larger than the minimum box that can tightly enclose the geometry. Further work is required to identify better mechanisms for determining the minimum bounding box for a transformed shape in *empirical worlds*. The bounds are primarily an aid to rendering algorithms, providing a guide to the spacial extent of a solid shape given by a function representation.

Class Name	`CadnoTransform`
Extends	`FrepType`

Value Format	`Transform` {
	 `center` *point*
	 `rotation` *rotation*
	 `scale` *point*
	 `scaleOrientation` *rotation*
	 `translation` *point*
	 `children` [(*FrepType*)*]
	 }

Default Value	`Transform` {
	 `center 0 0 0`
	 `rotation 0 0 1  0`
	 `scale 1 1 1`
	 `scaleOrientation 0 0 1   0`
	 `translation 0 0 0`
	 `children` [ ]
	 }

Parameters	Name	Type
	`center`	`Point`
	`rotation`	`CadnoRotation`
	`scale`	`Point`
	`scaleOrientation`	`CadnoRotation`
	`translation`	`Point`
	`children`	List of objects of `FrepType`.

The `CadnoTransform` class represents three-dimensional versions of affine transformations, with **c** represented in the `center` parameter, **R** in the `rotation` parameter, **O** in this `scaleOrientation` parameter, **S** in the `Scale` parameter and **t** in the `translation` parameter. Once a string represented by a `CadnoTransform` has been parsed, the internal data representation of these parameters are computed as matrices and vectors. This minimises the number of computationally expensive calls to mathematical library functions for sine, cosine and square root during the evaluation of the functional representation of a transformed point set. Note that the default value for a `CadnoTransform` corresponds to the identity mapping for each component transformation, e.g. a `translation` vector of $(0,0,0)$, a rotation through 0 degrees and so on.

The function representation for a transformed point set is found by transforming the original space for the original function representation of the point set. To evaluate the function representation of the transformed shape, any point in the space of the transformed shape is mapped by an affine transformation to the original space, in which the function representation of the original shape is evaluated. This transformation of space is represented by the function w, which corresponds to the inverse of the shape transformation represented, where

$$w(\mathbf{p}) = (\mathbf{OS}^{-1}\mathbf{O}^{-1}\mathbf{R}^{-1}((\mathbf{p} - \mathbf{t}) - \mathbf{c})) + \mathbf{c} \tag{7.6}$$

For any point $\mathbf{p} = (x, y, z)$ and a function representation for the `children` of a `CadnoTransform` instance g, the function representation for the transformed point set f is $f(\mathbf{p}) = g(w(\mathbf{p}))$.

In Figure 7.7, one of the vertices of the hexagon shape is circled in each representation to illustrate this space transformation process. Point $\mathbf{p}$ is shown in transformed space in Figure 7.7h, and is located at a vertex of the transformed hexagon. To evaluate the function representation of the transformed hexagon at $\mathbf{p}$, $\mathbf{p}$ is transformed by w back through Figures 7.7g to 7.7a where the function representation for the original shape is evaluated at $w(\mathbf{p})$.

Four operators can be used in *empirical world* scripts to implicitly define instances of the `CadnoTransform` class. These are: one for translating point sets, one for rotating point sets, one for scaling/linearly shearing point sets and one for a combination of all these transformations.

Implicit Definitions for Translations

Operator Class		`CadnoTformTranslation`
Maps From		*float* × *float* × *float* × *FrepType* or *integer* × *integer* × *integer* × *FrepType* etc.
Maps To		`CadnoTransform`

Example	Defn.	`t1 = translate(2.3, 2.4, 2.5,`
		`        children [ Box { size 2 3 1 } ] )`

	Value	`t1 = Transform {`
		`        center 0 0 0`
		`        rotation 0 0 1  0`
		`        scale 1 1 1`
		`        scaleOrientation 0 0 1  0`
		`        translation 2.3 2.4 2.5`
		`        children [ Box { size 2 3 1 } ]`
		`        }`

The translation of a point set can be implicitly defined in *empirical worlds* using the operator represented by the `CadnoTformTranslation` class. The data type of the value associated with the left-hand side of an implicit definition containing this operator is represented by the `CadnoTransform` class. The first three arguments to the operator represent the translation vector **t** and the fourth argument is the original point set prior to translation.

Implicit Definitions for Rotations

Operator Class		`CadnoTformRotation`
Maps From		`Point` × `CadnoRotation` × *FrepType* or `CadnoRotation` × *FrepType*
Maps To		`CadnoTransform`

Example	Defn.	`t2 = rotate(2 2 3, 0 1 0 3.1415,`
		`      children [ Cone { } ] )`

	Value	`t2 = Transform {`
		`      center 2 2 3`
		`      rotation 0 1 0  3.1415`
		`      scale 1 1 1`
		`      scaleOrientation 0 0 1  0`
		`      translation 0 0 0`
		`      children [`
		`        Cone { bottomRadius 1 height 2} ]`
		`      }`

Example	Defn.	`t3 = rotate(0.5 0.25 0.25  1.5708,`
		`      Sphere { radius 3.2 })`

	Value	`t3 = Transform {`
		`      center 0 0 0`
		`      rotation 0.5 0.25 0.25  1.5708`
		`      ⋮`
		`      children [ Sphere { radius 3.2 } ]`
		`      }`

The class `CadnoTformRotation`, when used as an operator for an implicit definition in an *empirical world* script, creates instances of `CadnoTransform` class representing the given geometric shapes rotated by the specified amount. The first argument to the operator is an optional vector represented as a `Point` that is the centre (**c**) of the transformation and, therefore, a point that the axis of rotation must pass through. The next argument is a `CadnoRotation` representation (**R**) (see Appendix A.4) of the axis for the rotation and the anti-clockwise angle of rotation. The last argument is the function representation for the point set for the geometry that is being transformed.

In the table describing the `CadnoTformRotation` class above, the second example shows the rotation of a `CadnoSphere`. The resulting point set is actually identical to the original one. With all objects in *empirical worlds*, many different descriptions can exist for the same point set. The difference between the representations of the point sets can be established by comparing their string representations. Any user interacting with a script of *empirical world* definitions should take care to ensure they consider the most efficient way to represent their point sets by examining the issues relating to dependency structures discussed in Chapter 4.

Implicit Definitions for Scaling

Operator Class		`CadnoTformScale`
Maps From		`Point` × *FrepType* or *float* × *float* × *float* × *FrepType* or `Point` × `CadnoRotation` × `Point` × *FrepType* or `CadnoRotation` × `Point` × *FrepType*
Maps To		`CadnoTransform`

Example	Defn.	`s4 = scale(1, 2, 1,`
		`        children [ Box { size 2 2 2 } ] )`

	Value	`s4 = Transform {`
		`        ⋮`
		`        scale 1.0 2.0 1.0`
		`        ⋮`
		`        children [ Box { size 2 2 2 } ]`
		`        }`

A scaling transformation of a point set can be described by an implicit definition with the operator represented by class `CadnoTformScale`. There are four possible sequences of types in arguments to this operator and, in each case, the data types associated with the identifier on the left-hand side of the implicit definition is represented by a `CadnoTransform` class. One of these sequences of arguments consists of a `Point` representing the scale transformation **S**, followed by a

`CadnoRotation` representation of scale orientation matrix **O**, followed by a `Point` representing the centre vector **c**. The final argument in the sequence corresponds to the function representation for the shape being transformed.

Implicit Definitions for Affine Transformation

Operator Class		`CadnoMakeTransform`
Maps From		`Point` × `CadnoRotation` × `Point` ×
		`CadnoRotation` × `Point` × *FrepType*
Maps To		`CadnoTransform`
Example	Defn.	`t5 = transform(2 2 3, 0 1 0  3.1415,`
		`1 2 1, 1 0 0  1.5708, 2 3 4,`
		`children [ Box { size 2 4 2 } ] )`
	Value	`t5 = Transform {`
		`center 2 2 3`
		`rotation 0 1 0  3.1415`
		`scale 1 2 1`
		`scaleOrientation 1 0 0  1.5708`
		`translation 2 3 4`
		`children [ Box { size 2 4 2 } ]`
		`}`

Every parameter for an instance of a `CadnoTransform` class can be implicitly defined in an *empirical world* script by using the operator represented by the `CadnoMakeTransform` class. The ordering of the sequence of arguments to the operator is: `center` **c**, `rotation` **R**, `scale` **S**, `scaleOrientation` **O** and `translation` **t**. The last argument of this sequence corresponds to the function representation for the original point set that is being transformed.

7.4 Shape Combination in Empirical Worlds

In this section, binary and n-ary operations for point sets are described. These form new data types and operators in *empirical world* scripts. They include types for the description of, and operators for the implicit creation of, new shapes that are a combinations of other existing point sets. The new point sets can also be used as parameters to further combination operations. The combinations described here are the three standard Constructive Solid Geometry[13] operators [Bow94] of union, intersection and difference (also known as cut) of point sets, along with two variations of these, blend-union and blend-intersection. Finally, an operation representing the metamorphosis between two existing point sets is described.

None of the operators described in this section are part of the existing VRML notation and new syntax has been adopted to describe them. The shapes described by the combination operators must be rendered through the polygonisation algorithm described in Section 8.4.3[14]. To distinguish these shapes from those that may have a VRML representation, all subclasses of `DefnType` described here also extend another abstract class called `TopologyType`. The relationship between these classes is shown in the class diagram in Figure 7.2. All objects that extend `TopologyType` have the same `virtualise()` method to render their geometry and must implement a method `f` corresponding to their function representation for any three-dimensional point (x, y, z).

The classes that represent operators described in Section 7.4.1 through to the end of this chapter can take an additional `CadnoDetail` parameter as an argument in *empirical world* scripts (see Section 7.2.1). This parameter allows for the interactive control of the quality of the polygonisation and hence the level of detail of the image displayed in the VRML browser. For all implicit definitions of shape in these sections, the sequence of arguments to the operators can be preceded by an optional argument that is a variable represented by the `CadnoDetail` class.

[13]Commonly abbreviated to CSG, also known as set-theoretic operations.

[14]At the current time, polygonisation is the method adopted for rendering function representation shapes in *empirical worlds*. The same function representation can be used to create a ray-traced image [FvF+93].

7.4.1 Union of Point Sets

In this section, the integration of the set-theoretic union of point sets into *empirical worlds* is described. The only parameters that describe a union in *empirical worlds* are a list of `children` classes that extend `FrepType`. The bounding box of a union is a box that bounds the union of all the bounding boxes of the defining shapes. If the bounding boxes of the defining shapes tightly encloses their geometry then the bounding box of the union point set will also tightly enclose its geometry. Figure 7.8 shows a diagrammatic representation of the union of a `Box` and a `Cylinder` point set.

Class Name	`CadnoUnion`	
Extends	`TopologyType`	
Value Format	`Union` { `children` [(*FrepType*)*] }	
Parameters	Name	Type
	`children`	List of objects of `FrepType`.

An explicit definition for a union point set in an *empirical world* script is of the form

```
u1 = Union { children [ Box { } Cylinder { height 3 radius 0.5 } ] }.
```

A definition for a union of point sets has an explicit right-hand side value that directly represents a point set for an instance of a `CadnoUnion` class. Implicit definition of a union point set in an *empirical world* script is possible by using an operator based on the `CadnoMakeUnion` class. This operator creates and maintains dependencies based on its arguments for an instance of a `CadnoUnion` class. The two possible sequences of arguments to this operator are either a sequence of variables represented by the `FrepType` class or a variable associated with an existing instance of the `FrepList` class.

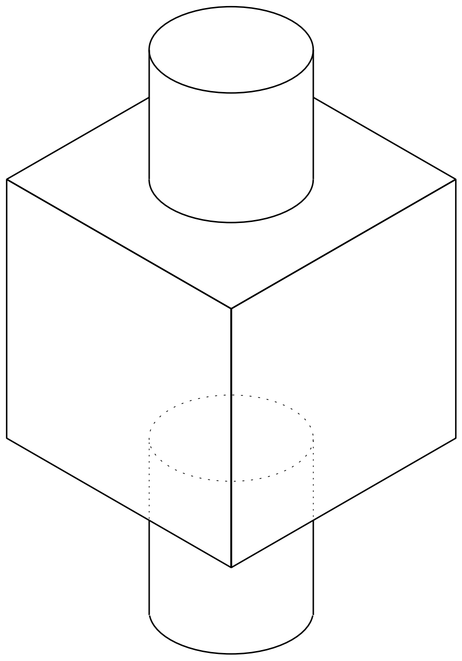


Figure 7.8: Diagrammatic representation of the union of a `Box` and a `Cylinder`.

Operator Class		`CadnoMakeUnion`
Maps From		`FrepList` or `CadnoDetail` × `FrepList` or (*FrepType*)* or `CadnoDetail` × (*FrepType*)*
Maps To		`CadnoUnion`

Example	Defn.	`u1 = union(detail 14 14 14,`
		`children [`
		`Box { size 2 2 2 }`
		`Cylinder { radius 0.5 height 3 }`
		`] )`

	Value	`u1 = Union {`
		`children [`
		`Box { size 2 2 2 }`
		`Cylinder { radius 0.5 height 3 }`
		`] }`

In the table above describing the `CadnoMakeUnion` class, the example shows a union of a `Box` and a `Cylinder` equivalent to that shown in Figure 7.8. If rendered in a VRML browser, the polygonisation used to generate the image of this example shape would split the shape into $14 \times 14 \times 14$ equal voxels that are contained within the bounding box of the union point set.

One possible function representation for a union of point sets with function representations $g_1, \ldots, g_n$ at any point $\mathbf{p} = (x, y, z)$ in space, is to find the maximum value of $g_1(\mathbf{p}), \ldots, g_n(\mathbf{p})$. This representation can have C^1 discontinuity wherever any pair of the arguments are equal. Better continuity can be achieved with an instance of the `CadnoBlendUnion` described in Section 7.4.4 of this chapter, with a value for `displacement` set to zero. The blend union is a binary operation whereas the standard union described in this section is an n-ary operation.

7.4.2 Intersection of Point Sets

In this section, the integration of set-theoretic intersection into *empirical world* scripts is described. The parameters for an intersection of point sets in *empirical worlds* are a list of `children` containing the defining point sets of the intersection. This operation can result in an empty point set if there is no common solid material represented by any of the defining points sets.

The bounding box of an intersection point set is a box that tightly bounds the intersection of all the bounding boxes of the defining point sets. If the bounding boxes for the children tightly enclose their geometry then the bounding box for the intersection geometry will also tightly enclose the geometry. Figure 7.9 shows the intersection of a `Sphere` with a `Cone` rendered using the *empirical world* builder tool (see Section 8.4). The resulting geometry from this operation can be described as a truncated cone with a rounded base and top.

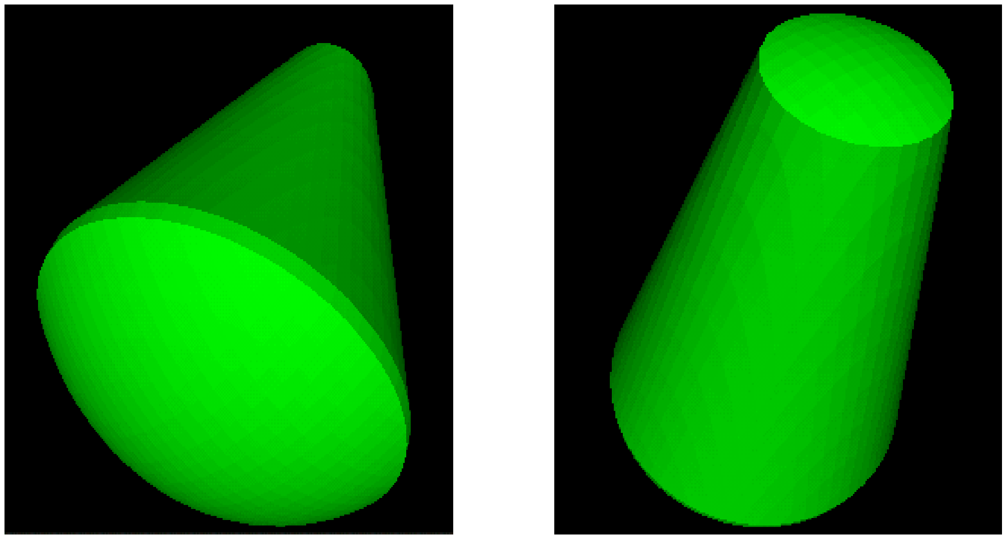


Figure 7.9: Images of the intersection of `Cone` and `Sphere` point sets.

Class Name	`CadnoIntersection`	
Extends	`TopologyType`	
Value Format	`Intersection` { [`children` (*FrepType*)*] }	
Parameters	Name	Type
	`children`	List of objects of `FrepType`.

In an *empirical world* script, an intersection can be defined implicitly by using an operator represented by the `CadnoMakeIntersection` class. The two possible sequences of arguments to this operator are either a sequence of variables represented by the `FrepType` class or a variable associated with an existing instance of the `FrepList` class.

Operator Class		`CadnoMakeIntersection`
Maps From		`FrepList` or
		`CadnoDetail` × `FrepList` or
		(*FrepType*)* or
		`CadnoDetail` × (*FrepType*)*
Maps To		`CadnoIntersection`
Example	Defn.	`i1 = intersection(children [`
		`Cone { bottomRadius 1 height 4 }`
		`Sphere { radius 1 }`
		`] )`
	Value	`i1 = Intersection {`
		`children [`
		`Cone { bottomRadius 1 height 4 }`
		`Sphere { radius 1 }`
		`] }`

The *empirical world* script for the images shown in Figure 7.9 is the same as the one in the table above.

If the defining point sets have a functional representations $g_1, \ldots, g_n$ then one possible function to represent the intersection of these point sets for any point $\mathbf{p}$ is to find the minimum value of $g_1(\mathbf{p}), \ldots, g_n(\mathbf{p})$. This function is C^1 continuous except at any point where any pair of the component function representations have equal values. For better continuity, an intersection of point sets represented by the `CadnoBlendIntersection` class described in Section 7.4.5 can by used, with the `displacement` parameter set to zero.

7.4.3 Cutting Point Sets by Other Point Sets

In this section, the integration of set-difference operation into *empirical worlds* is described. A point set representing some solid material can appear to have the material from another point set cut away by the set-difference of the two point sets. The set-difference operation described here is a binary operation, where the original point set is considered the `body` and the cutting point set as the `tool`. This operation can lead to an empty point set in the case where all the material in the body is also contained in the tool. For a body point set B and a tool T, the set-difference operation $B \setminus T$ is the same as $B \cap \neg T$, the intersection of B with every point not inside T.

A bounding box that is guaranteed to enclose a cut point set is the same as the bounding box for the body shape. A tightly enclosing bounding box for the shape may be smaller than this, as material can be removed from the body by the cut operation. The method `shrinkWrap()` implemented in the class `TopologyType` samples the function representation at the faces of the bounding box for the current level of detail. A good guess to the bounding box is initially found before it is grown or shrunk in the x, y and z directions until it approximates a tight bounding box that *wraps* the geometry given by the function representation. This process can go wrong, especially for infinite solid shapes. To prevent continuous looping and searching for shape, the process is limited to a maximum number of iterations. In most cases, the maximum number of iterations is not reached. When the maximum number of iterations is exceeded, this process can lead to the rendered geometric shapes having holes in their surfaces.

Figure 7.10 is a diagrammatic representation of a `Box` point set body cut by a `Cylinder` point set tool. The parameters `body` and `tool` in an instance of a `CadnoCut` class are the geometric solid shapes used in the set difference.

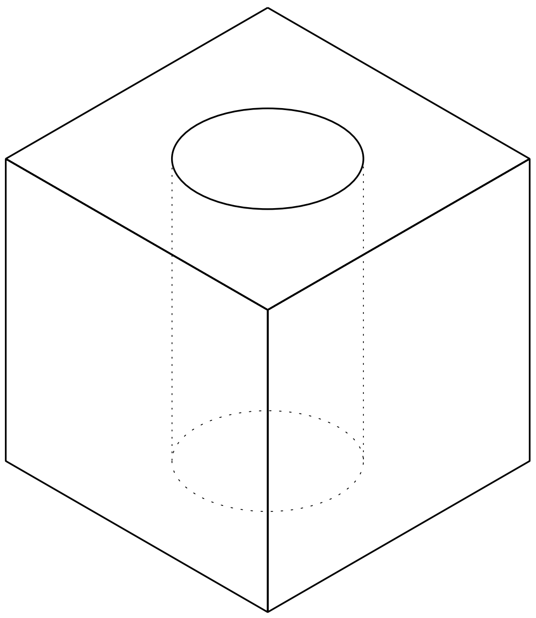


Figure 7.10: Body point set `Box` with a tool point set `Cylinder` cut away from it.

Class Name	`CadnoCut`	
Extends	`TopologyType`	
Value Format	`Cut` { `body` *FrepType* `tool` *FrepType* }	
Parameters	Name	Type
	`body`	`FrepType`
	`tool`	`FrepType`

An instance of a `CadnoCut` class can be implicitly defined in an *empirical world* script by the use of an operator represented by the `CadnoMakeCut` class. The order of arguments to this operator are an optional `CadnoDetail` followed by a solid geometric object representing the `body` for the cut operation and then another solid geometric object that is the `tool`.

Operator Class		`CadnoMakeCut`
Maps From		*FrepType* $\times$ *FrepType* or
		`CadnoDetail` $\times$ *FrepType* $\times$ *FrepType*
Maps To		`CadnoCut`
Example	Defn.	`c1 = cut(Box { size 2 2 2 },`
		`        Cylinder { radius 0.5 height 4 } )`
	Value	`c1 = Cut {`
		`        body Box { size 2 2 2 }`
		`        tool Cylinder {`
		`          radius 0.5 height 4 }`
		`        }`

The *empirical world* script example in the table above is consistent with the diagrammatic representation of a `CadnoCut` in Figure 7.10.

The function representation f of one point set cut by another can be found by taking the intersection of the body function representation g_1 with the negative version of the tool function representation g_2 at the same point, where

$$f(x,y,z) = g_1(x,y,z) + (-g_2(x,y,z)) - \sqrt{g_1(x,y,z)^2 + g_2(x,y,z)^2} \qquad (7.7)$$

This function representation has C^1 discontinuity only at a point where the value of g_1 and g_2 are both equal to zero.

7.4.4 Union Blending of Point Sets

Using the function representation of shape, it is possible to describe variants of the set-theoretic operations that create shapes that appear as if they are blended together. Blending with functional representations is described by Savchenko and Pasko in [SP94]. The union of points sets with blending is described in this section and the blending intersection of point sets in Section 7.4.5.

Point set blending involves the addition or subtraction of some material around the outside of the union of the two point sets. To do this with function representations of two geometric shapes g_1 and g_2, and their set-theoretic union u_1, it is necessary to add some blend material in the proximity of both of the two shapes. This can be achieved by adding a little extra of the values of $g_1(\mathbf{p})$ and $g_2(\mathbf{p})$ to the function representation of the value of the union. To do this, the proportion that the blend is affected by each of the defining shapes, a_1 for shape g_1 and a_2 for shape g_2, must be determined. It is also necessary to define the displacement value d that represents how much the blend should affect the union operation as a whole. Then function representation f for a point set resulting from a blend-union operation is given by the formulae

$$u(x,y,z) = g_1(x,y,z) + g_2(x,y,z) + \sqrt{g_1(x,y,z)^2 + g_2(x,y,z)^2} \tag{7.8}$$

$$f(x,y,z) = \frac{d}{1 + \left(\frac{g_1(x,y,z)}{a_1}\right)^2 + \left(\frac{g_2(x,y,z)}{a_2}\right)^2} + u(x,y,z) \tag{7.9}$$

The blending process produces much better results when the functions g_1 and g_2 are C^1 continuous almost everywhere. Figure 7.11 shows three snapshots of the blend union of a `Box` point set and a `Cylinder` point set. The displacement value d is set to -0.5 in the left-hand image, 0.0 in the central image and 0.5 in the right-hand image. In each case, the values for a_1 and a_2 are set to 1.0. With $d = 0$, the point set is exactly the same as for the set-theoretic union operation.

A blend union is parametrised by a float representing d, the `displacement`, and two floats representing the proportion of blending a_0 and a_1 for each defining point set, called `firstDistance` and `secondDistance` respectively. The bounding box for a blend union is found by starting with the bounding box for the set theoretic union of the defining shapes and executing the `shrinkWrap()` method, which approximates bounds that are just outside the actual geometry.

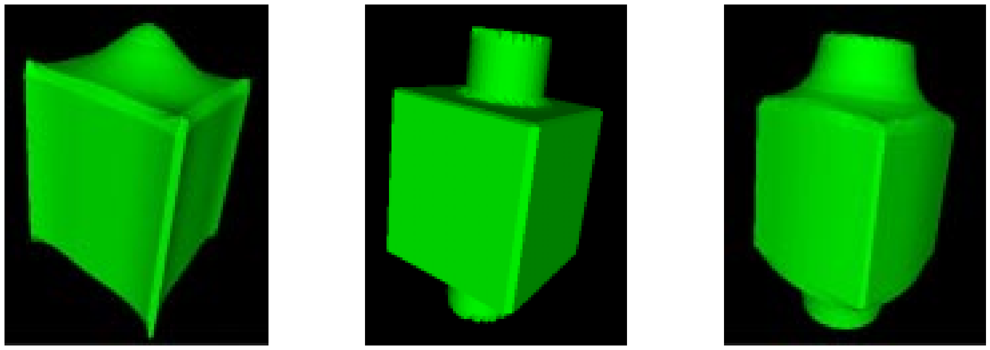


Figure 7.11: Blend union of a `Box` and a `Cylinder` for `displacement` values of -0.5, 0.0 and 0.5.

Class Name	`CadnoBlendUnion`	
Extends	`TopologyType`	

Value Format	`BlendUnion` {
	    `firstSolid` *FrepType*
	    `secondSolid` *FrepType*
	    `displacement` *float*
	    `firstDistance` *float*
	    `secondDistance` *float*
	}	

Parameters	Name	Type
	`firstSolid`	`FrepType`
	`secondSolid`	`FrepType`
	`displacement`	`CadnoFloat`
	`firstDistance`	`CadnoFloat`
	`secondDistance`	`CadnoFloat`

Implicit definitions of blend unions in *empirical world* scripts can be made by the use of the class `CadnoMakeBlendU`, which extends `DefnFunc`. The order of arguments to the operator in a script are a floating point value for the `displacement` d, followed by the `firstDistance` floating point value a_1, then the `secondDistance` floating point value a_2, then the `firstSolid` with function representation g_1 and finally the `secondSolid` with function representation g_2.

Operator Class		`CadnoMakeBlendU`
Maps From		`CadnoDetail` $\times$ *float* $\times$ *float* $\times$ *float* $\times$ *FrepType* $\times$ *FrepType* or *float* $\times$ *float* $\times$ *float* $\times$ *FrepType* $\times$ *FrepType*
Maps To		`CadnoBlendUnion`
Example	Defn.	`bu = blendUnion(detail 10 10 10, 0.5, 1.0, 1.0, Box { size 2 2 2 }, Cylinder { radius 0.5 height 4 } )`
	Value	`bu = BlendUnion { firstSolid Box { size 2 2 2 } secondSolid Cylinder { radius 0.5 height 4 } displacement 0.5 firstDistance 1.0 secondDistance 1.0 }`

7.4.5 Intersection Blending of Point Sets

The intersection blending of point sets creates a point set similar to a set-theoretic intersection of the same two point sets, except that some material is added or removed to achieve the effect of a blend between the intersecting shapes. The amount of material added is controlled by a displacement parameter d and the proportion of material from each geometric shape defining the blend by parameters a_1 and a_2. For any point $(x,\ y,\ z)$ and function representations for two solid geometric shapes g_1 and g_2, the function f for a blend intersection is

$$u(x,y,z) = g_1(x,y,z) + g_2(x,y,z) - \sqrt{g_1(x,y,z)^2 + g_2(x,y,z)^2} \qquad (7.10)$$

$$f(x,y,x) = \frac{d}{1 + \left(\frac{g_1(x,y,z)}{a_1}\right)^2 + \left(\frac{g_2(x,y,z)}{a_2}\right)^2} + u(x,y,z) \qquad (7.11)$$

In em empirical world scripts, the explicit value of a blend intersection is given by explicitly specifying two shapes `firstSolid` and `secondSolid`, along with the `displacement` parameter d. The proportion of the effect of each shape a_1 and a_2 on the operation can be expressed in `firstDistance` and `secondDistance` parameters respectively. The bounding box for the shape is found by taking the bounding box for the set-theoretic intersection of the two geometric shapes and then executing the `shrinkWrap()` method to find a more accurate set of bounds to suit the blended shape.

Class Name	`CadnoBlendIntersection`	
Extends	`TopologyType`	
Value Format	`BlendIntersection` { `firstSolid` *FrepType* `secondSolid` *FrepType* `displacement` *float* `firstDistance` *float* `secondDistance` *float* }	
Parameters	Name	Type
	`firstSolid`	`FrepType`
	`secondSolid`	`FrepType`
	`displacement`	`CadnoFloat`
	`firstDistance`	`CadnoFloat`
	`secondDistance`	`CadnoFloat`

Figure 7.12 shows the blend intersection of a sphere and a translated box for different values of the displacement parameter d and with the values of a_1 and a_2 set to 1.0. With a value for $d = 0$, the image is the same as the set-theoretic intersection of the same two shapes. Notice how blend material can be subtracted from the set-theoretic intersection with a negative value for d, as well as added to the intersection with a positive value for d.

Blend intersections in an *empirical world* script can be implicitly defined by the use of the class `CadnoMakeBlendI`, which extends `DefnFunc`. The order of arguments to the operator in a script is the same as the order for the blend union

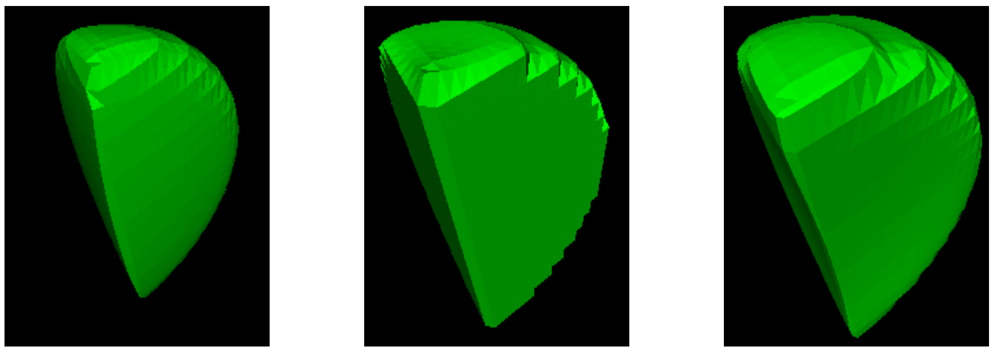


Figure 7.12: Blend intersection of a `Cone` and a `Sphere` for `displacement` values of `-0.5`, `0.0` and `0.5`.

operator.

Operator Class		`CadnoMakeBlendI`

Maps From		`CadnoDetail` × *float* × *float* × *float* ×
		   `FrepType` × `FrepType` or
		float × *float* × *float* ×
		   `FrepType` × `FrepType`

Maps To		`CadnoBlendIntersection`

Example	Defn.	`ci = blendIntersection(0.5, 1.3, 1.8,`
		`        Cone {`
		`          bottomRadius 1 height 4 },`
		`        Sphere { radius 1 } )`

	Value	`ci = BlendIntersection {`
		`        firstSolid Cone {`
		`          bottomRadius 1 height 4 }`
		`        secondSolid Sphere { radius 1 }`
		`        displacement 0.5`
		`        firstDistance 1.3`
		`        secondDistance 1.8`
		`        }`

7.4.6 Metamorphosis of Point Sets

A metamorphosis between point sets, known as a *morph*, is defined as a continuous transition between two pieces of solid geometry parametrised by a value t. When $t =$

0, the morph shape is exactly equal the same as first piece of geometry, and when $t =$ 1 the morph shape is the same as the second piece of geometry. When $t = 0.5$, there is an equal contribution from both pieces of geometry. Such a morphing transition can be achieved by with the function representation for solid objects by the simple equation shown below (Equation 7.12), where g_1 is the function representation for the first solid and g_2 is the function representation for the second. An example of function representation morphing between a box and a spiral by Pasko et al is found in [PSA93]. The morph f is C^1 continuous everywhere that the defining function representations are C^1 continuous, where

$$f(x, y, z) = (1 - t)g_1(x, y, z) + tg_2(x, y, z) \tag{7.12}$$

The morph between two solid objects can be explicitly defined in an *empirical world* script through the data type represented by the `CadnoMorph` class. The first solid with function representation g_1 is represented as the parameter `zeroSolid`, indicating that this is the solid represented by the morph for a value of $t = 0$. Similarly, the second solid with function representation g_2 is parameter `oneSolid`. The floating point value of t is the `proportion` parameter. The bounding box for a morph is calculated by taking the bounding box for the set-theoretic union of the two defining point sets and then executing the `shrinkWrap()` method to approximate bounds just outside the morph solid shape.

Class Name	`CadnoMorph`	
Extends	`TopologyType`	

Value Format	`Morph` {
	 `zeroSolid` *FrepType*
	 `oneSolid` *FrepType*
	 `proportion` *float*
	 }	

Parameters	Name	Type
	`zeroSolid`	`FrepType`
	`oneSolid`	`FrepType`
	`proportion`	`CadnoFloat`

Figure 7.13 shows a morph between a union of a box and a cylinder ("box/cylinder"), and a sphere with a cylindrical hole ("cut-sphere"). The figure is split into eight images with the corresponding value for t shown below each image. When $t = 0$, only shape for the box/cylinder is visible and when $t = 1$ then only the cut-sphere shape is visible. For values between 0 and 1, the shape can be observed to transform between the first and second defining solids. Two additional images are shown to demonstrate the effects that it is possible to achieve by using values for t that are outside the range $[0, 1]$.

A morph between two shapes can be defined implicitly in an *empirical world* script with the use of the `CadnoMakeMorph` class. The order for the arguments to the operator in a script is an optional `CadnoDetail` rendering level, followed by a floating point value for the `proportion` t, then the first shape (`zeroSolid`) with function representation g_1 and finally the second shape (`oneSolid`) with function representation g_2.

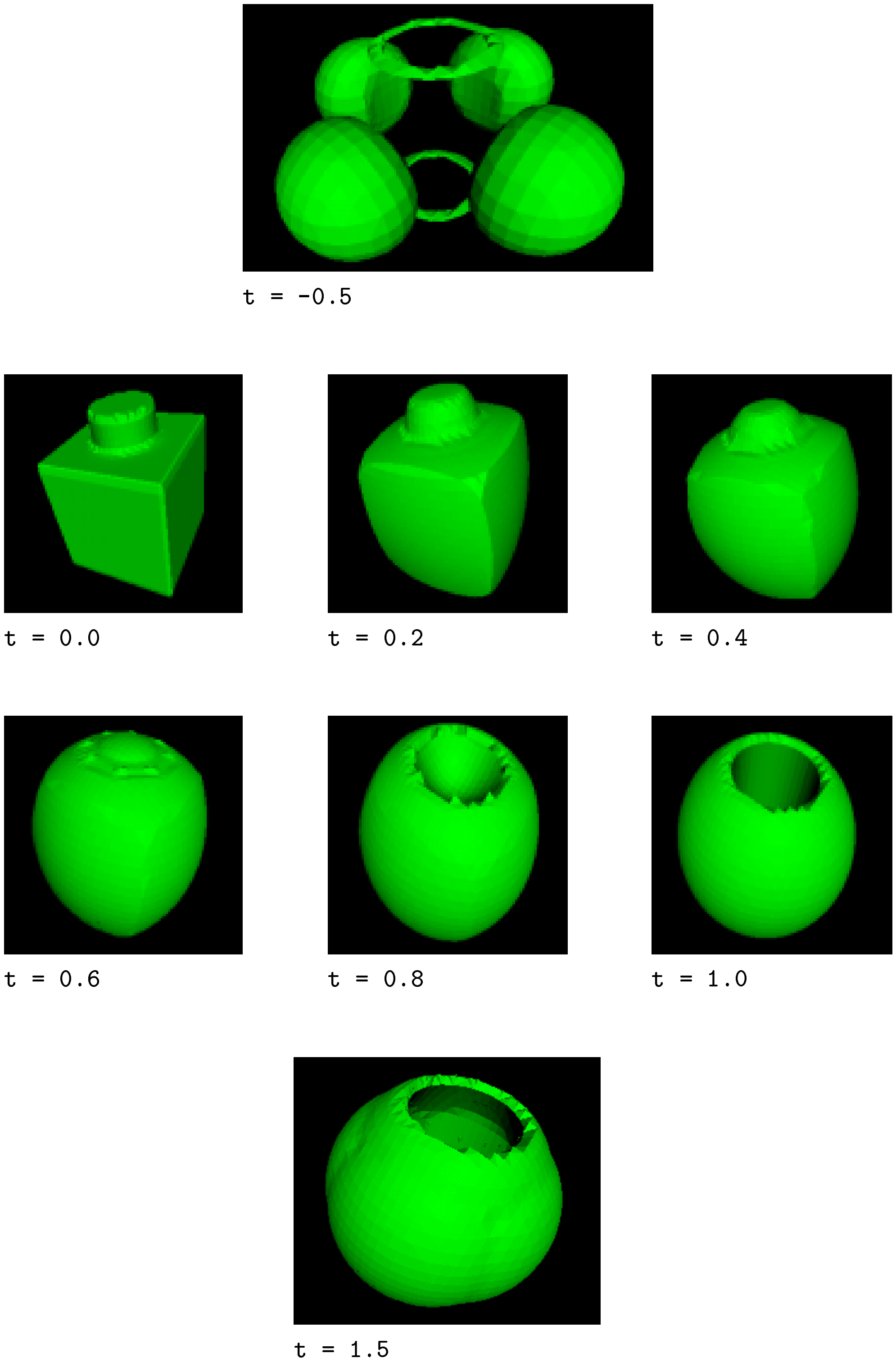


Figure 7.13: *Morph* between a box/cylinder shape and a cut-sphere shape, for varying values of `t`.

Operator Class		`CadnoMakeMorph`
Maps From		`CadnoDetail` × *float* × *FrepType* × *FrepType* or *float* × *FrepType* × *FrepType*
Maps To		`CadnoMorph`

Example	Defn.	`m1 = morph(0.5,`
		`Sphere { radius 1 },`
		`Cylinder { radius 1 height 2 } )`

	Value	`m1 = Morph {`
		`zeroSolid Sphere { radius 1 }`
		`oneSolid Cylinder {`
		`radius 1 height 2 }`
		`proportion 0.5`
		`}`