Chapter 9

Conclusions and Further Work

9.1 Introduction

In this final chapter, the research work presented in Chapters 4 through to 8 of this thesis is reviewed with reference to:

- the research aims of the work as set out in Chapter 1;
- the requirements for tools identified in Chapter 2;
- the technical issues in Chapter 3.

This is followed by some suggestions for further work (Section 9.3) and a proposal for a new application, based on spreadsheet ideas, that can support interactive open development of geometric models as well as other models requiring more complex, general data types than are supported by existing spreadsheet applications.

In Chapter 1, the roles of programmer, user and systems expert are discussed. Empirical modelling supports the conflation of these roles. In the chapters that follow (particularly 3 through to 6) the emphasis is on abstract models, issues of data representation and programming toolkits. Support for high-level definitive scripts (limited in EDEN — see Section 1.2.2) can be better achieved by discarding

EDEN in favour of a new generation of tools that can be best implemented using the DAM Machine or the JaM Machine API. EDEN is reasonably successful in conflating the roles of user and programmer, but only the next generation of tools can enable a programmer to construct new instruments for empirical modelling (see Figure 1.1). These new tools will also provide improved interactivity, and data representations that support a wider range of applications — including solid shape modelling — than is possible at present.

One of the aims for this work (see Section 1.2.2) has been to address issues not tackled by EDEN. The DAM Machine and the JaM Machine API are primarily concerned with the dependency maintenance aspect of EDEN, and the management of data becomes the task for a programmer. The DAM and JaM toolkits address:

- aspects of representing the privileges of interacting agents using definition permissions (JaM Machine API);
- the provision of greater flexibility to represent a broader range of data types (empirical worlds);
- efficiency by using compiled code for operators (DoNaLD to DAM translator) and improving dependency update algorithms (block redefinition algorithm);
- portability through the use of the Java programming language (JaM Machine API);
- the dynamic creation/elimination of definitions (management of the definitive store of the DAM Machine);
- generalised representation of data types (JaM Machine API and patterns of bits for DAM words).

9.2 Review of Definitive Programming

In Section 3.5 the transformation of all representable types of data to serialised data types in a definitive notation is proposed as a solution to many of the technical issues. The JaM Machine API allows a programmer to create special atomic data types for definitive notations by extending the class `JaM.DefnType`. The *empirical worlds* case-study in Chapters 7 and 8 illustrates that this is an effective method for modelling solid geometry using definitive notations. This is a new approach to definitive programming that combines:

- dependency relationships between observables;
- data structure relationships and constructors;
- dependency through expression evaluation;
- dependency between sets (such as point sets)

in unified dependency structures.

Figure 9.1 illustrates such a combined dependency structure. The script shown represents a geometric model of a hammer with a box for a head and a cylinder for a handle[1]. The length, width and height of the head are the defining parameters from which all other parameters of the model are calculated. The figure also shows a description of each parameter and the dependency structure associated with the script.

In this example, the parameters can be classified as follows:

Scalar Values (`a1` ... `a7`) Numerical values that represent defining parameters for other parts of the model. The dependency between `a6` and `a1` is given by a mathematical expression of the value of `a6` (with automatically generated parameter `a6_2`).

[1]There is a diagrammatic representation of the hammer in figure 9.2.

`a1 = 5`	`a1` is hammer head length
`a2 = 3`	`a2` is hammer head width
`a3 = 2`	`a3` is hammer head height
`a4 = divide(a3,2.5)`	`a4` is hammer handle radius
`a5 = multiply(a1,2)`	`a5` is hammer handle length
`a6 = multiply(a1,-1)`	`a6` is handle shift y amount
`a7 = divide(a1,10)`	`a7` is handle shift x amount
`b1 = makePoint(a7,a6,0)`	`b1` is handle translation vector
`c1 = box(a1,a2,a3)`	`c1` is hammer head, no hole
`c2 = cylinder(a4,a5)`	`c2` is hammer handle original
`c3 = translate(b1,c2)`	`c3` is hammer handle translated
`d1 = union(c1,c3)`	`d1` is complete hammer
`d2 = cut(c1,c3)`	`d2` is hammer head with hole

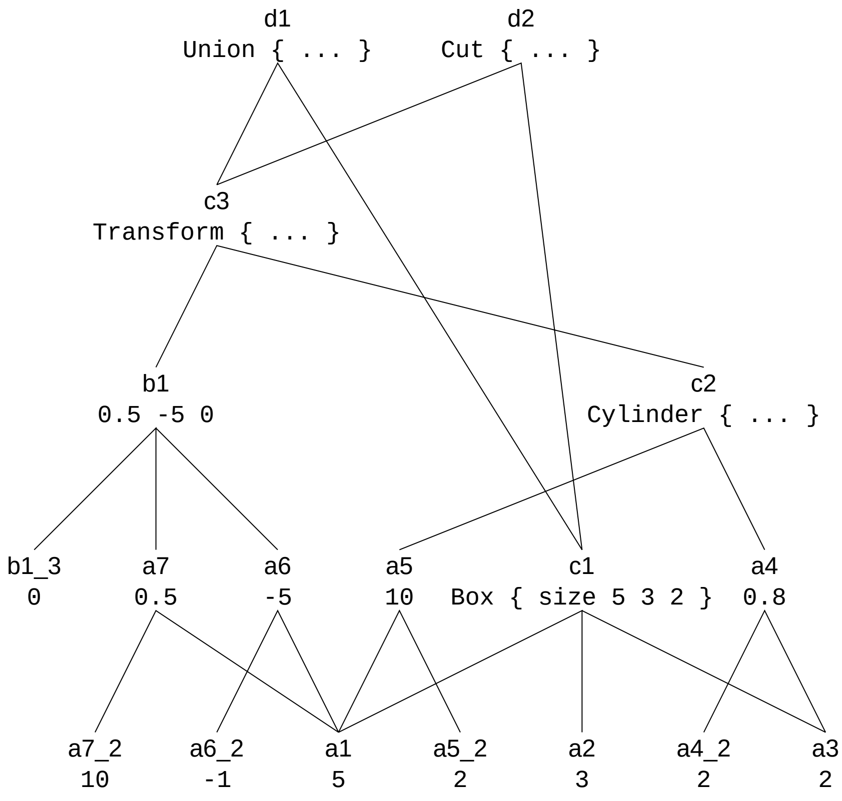


Figure 9.1: Script and dependency structure for a hammer model.

Data Structures (`b1`) A value for a three-dimensional point is dependent on its three sub-component parts. The dependency between `a6`, `a7` and `b1` is for the creation of an instance of a data structure by the "`makePoint`" operator.

Primitive Shapes (`c1`, `c2`) Point sets with their parameters forming values in internal data structures. The dependency between `c2`, `a4` and `a5` defines indicates that the scalar values are parameters in the construction of the point sets for the cylinder primitive shape that represents the hammer's handle.

Point Sets (`d1`, `d2`) Point sets constructed from boolean operations on other point sets. The dependency between `c1`, `c3` and `d1` constructs `d1` as the union of the primitive shape point sets.

These four categories for types of dependency can also be interpreted as different forms of agency: there are agents that are responsible for dependencies that form data structure, other agents responsible for expressions with their evaluation (possibly represented by several definitions to simulate an expression tree), and so on. This categorisation does not include all possible combinations of types of dependency. For example, the translated cylinder shape `c3` is actually a combination of an existing point set with some parameters that represent a translation. This defines a new point set. In Sections 9.2.1 through to 9.2.5, the relevance to empirical modelling of the integration of different types of dependency and data structure is discussed.

9.2.1 Empirical Modelling with Geometry

Cognitive Artefacts for Geometric Modelling

The *empirical world* tool demonstrates the construction of geometric cognitive artefacts (e.g. the curtain pole support of Section 8.5). By experimenting with the parameters defining the model, replacing component parts of the model or using the constructed model as part of other models, the script of definitions describing the geometry is available for open development and exploration of shape. Shape modelling is not supported in the *Tkeden* tool. For tasks other than shape modelling, it would also be possible to build new applications that use the *empirical world* classes as a rendering method for providing three-dimensional, solid shape metaphors for cognitive artefacts. With appropriate operators, it is possible to represent the dependency between observables for the artefact model and its graphical representation as shape with definitions.

Animating Geometric Models

With the *empirical world* tool, it would not be possible to animate the timepiece artefacts of Section 2.3.2 with three-dimensional geometry in real time. There may be a delay of more than one second in the time taken to render the model, both in terms of polygonisation and transmission of the model to a VRML browser. This is because the tool is optimised for exploring implicitly defined shapes by flying around them when they appear in a browser. This is an implementation-specific problem and when programming libraries become available for Java to support three-dimensional modelling[2], it will be possible to control computer graphics hardware more directly from JaM classes. In addition, it may be possible to apply techniques similar to those of the DoNaLD to DAM Machine translator of Section 5.4 to animate smoother rotation of the second-hand of a clock than is possible in *Tkeden*.

[2]The Java3D API is under development and currently exists in an alpha-test version. Initial tests by the Manufacturing Group at the University of Warwick have shown this to be a very powerful toolkit for implementing fast rendering of three-dimensional shape.

Support for Collaborative Modelling

By using classes of Java programming APIs that support threading processes and network communications, it is possible to support simultaneous and incremental development of cognitive artefacts by several cooperating agents. The support for this in the JaM Machine API is implemented through setting permissions for each definition and providing security by a username and password system. Each human agent can interact with a model at a workstation through a graphical interface, such as a web browser, that can display geometry that is a visual metaphor for their view of an artefact. It is also possible to take advantage of the threading of processes to introduce automatic agents into models, with guarded protocols to take action. It is the job of a programmer to build interfaces using the JaM Machine API and manage the scripts of definitions — the advantage of this method is that a script that represents an artefact and its current state can be distributed between human agents for interactive collaborative working[3].

9.2.2 Empirical Modelling for Geometry

Shape Modelling

The general data representation technique provided by the JaM Machine API has been used to provide an extendable definitive notation for geometric modelling. The *empirical world* class library supports all the geometric shape primitives available in the VRML notation and builds on these through the use of the function representation for shape with a polygonisation algorithm for their display. The basic standard primitives that are used in CSG modelling are combined in *empirical worlds* with skeletal-based implicit representation for shapes. Other shape representations, in-

[3]It is interesting to note that a lot of tools that support collaborative working are rather static, working on publish/subscribe models where one agent works on a model, publishes it and then another user subscribes and works on the model. The JaM Machine API has the potential to support simultaneous and interactive collaborative working.

cluding surface models, can be added to *empirical worlds* by introducing a class to represent their function representation together with a VRML-like syntax for describing their parameters. These could include, for example, the Bézier volume function representations described by [MPS96]. All the operations on the point sets are closed — they produce other point sets in function representation. For example, it is possible to cut a shape constructed by CSG techniques away from a shape created by using the BCSO representation.

Dependency and Constraints

Relationships between geometric entities can be expressed as definitions in scripts. The parameters and structure of models can be efficiently updated given a change to a value by the block redefinition algorithm. There is no need for a constraint satisfaction algorithm and a script of definitions will never be over- or under-constrained[4]. The classes support the open development of models and enable a user to incrementally construct models from their experience. The models themselves then generate new experience. In the hammer model script shown in Figure 9.1, the box and cylinder could be constructed initially and then subsequently be located and scaled. Alternatively, the order of construction could reflect the order of the script. The proportions in the model having been established, the hammer head shape can be replaced by a more realistic shape.

CADNO and Empirical Worlds

Empirical world scripts can potentially be used as templates in the realisation section (see page 50) in a new implementation of the CADNO notation. The alternative to this is to implement data types for representing topological structures (complexes) as part of the *empirical world* tool. In practice, it is possible to define points to

[4]See the discussion of two way constraints in Section 9.2.3.

represent abstract observables in *empirical worlds*. For the table model that is used as a case-study for CADNO in Section 2.4.3, it is possible to introduce definitions of the parameters table height, length, width and so on into *empirical world* scripts. In this way, the abstract corner points of the table — that do not physically exist — are incorporated as part of the geometry. One of the features of CADNO is the possibility to restrict the parameters of certain ranges. To do this with *empirical world* classes, a programmer needs to implement methods that prevent definitions for values outside specified ranges, either by intercepting definitions before a call to `JaM.Script.addToQ` or by starting threads to continuously police the ranges of certain values.

9.2.3 Technical Challenges Reviewed

Different Kinds of Copy

In Section 3.2.1 of the thesis, the problem of copying assemblies of definitions is considered. Since no definition represented by the DAM Machine or in a `JaM.Script` defines a value that is an assembly[5] and since no function composition is allowed, a copy of an observable's value is exactly a mirror copy, photographic copy or reconstruction (with reuse) copy. A programmer using the JaM or DAM Machine programming toolkits can choose which kind of copy they wish to make by using various methods, without concerns for how to copy any subcomponents of assemblies of definitions. These methods are described in Table 9.1.

Higher-order Dependency

In the absence of assemblies of definitions, the higher-order dependencies described in Section 3.2.2 do not pose a significant technical challenge.

[5]In a DAM representation, every observable is represented by a pattern of bits. In an instance of a class that extends `JaM.DefnType`, every value has a string representation that can be printed and recognised.

Type of copy	DAM Machine	JaM Machine API
Photographic	Copy the value bit by bit into another memory location in definitive store. No dependency.	Call `JaM.Script.printVal` method for the original definition, change the identifier to the name of the copy. Use this string to call `JaM.Script.addToQ`.
Mirror	Create a subroutine that looks up the value stored at the address passed to it and returns this value. Use this subroutine for the operator associated with the copy. The argument to the operator should be the address of the original value.	Use the `JaM.JaMID` operator (uses `printVal` method to maintain dependency) to create an implicit definition where the the argument is the original and the identifier is the copy.
Reconstruction	Copy the associated operator from the original to that associated with the copy. Also, construct copies of the dependencies list for the original and associate this with the copy.	Call `JaM.Script.printDef` method for the original definition, change the identifier name to the name of the copy. Use this string to call `JaM.Script.addToQ`.

Table 9.1: Methods for copying in DAM and JaM.

Such dependencies can be handled by introducing serialised data types that can represent all observables values at level 0 and operators to represent dependency between these values at level 1. As it is possible to conceive a data type for the representation of a definitive script (a JaM class that extends `JaM.DefnType` and contains a field for an instance of a `JaM.Script` class), dependencies at higher levels representing patterns of dependency between scripts can be considered at level 0 and 1 also. This scripts-within-scripts model is unlikely to provide the detailed support that tools designed to support higher-order dependency will be able to provide[6].

[6]Gehring is currently developing a programming API implemented in Java that supports empirical modelling with higher order dependency, entitled *MODD*.

Although most observed dependency in the world can be described by definitions where one observable is dependent on another, for some dependencies this distinction is unclear. For example, geometric constraints in parametric modelling packages can constrain two lines to be parallel to one another. It is desirable for a tool that supports empirical modelling to be able to handle constraints. One way to deal with such constraints between observables using the JaM Machine API is to choose one direction for the dependency and then introduce an automatic agent (maybe with a thread) who observes all definitions that are being placed onto the queue of redefinitions. If observable a depends on the value of observable b and the value of a is redefined, then the direction of the dependency is switched by the automatic agent. The value of b is redefined to depend on the value of a. There needs to be a mechanism provided by a programmer of an application to allow a user to be able to create and destroy these automatic agents.

Moding and Data Structure

The issue of moding variables in a definitive script, introduced in Section 3.3.1, is greatly simplified by considering all data as representable by special atomic types. The level at which a variable is defined implicitly or explicitly can be established by looking through a definitive script. In the JaM Machine API, data structure is expressed via the fields of classes that extend `JaM.DefnType`. These classes are instantiated to represent values for observables in definitive scripts. This underlying data structure is separated from dependency maintenance by only accessing and setting the values of the fields of an instance through method calls and carefully defined operators, which are classes that extends `JaM.DefnFunc`.

9.2.4 The Contribution of the DM Model

The DM Model described in Section 4.2 can be used to reason about dependency structures without any need to consider application-specific information. It is possible to build a DM Model of a script of definitions and to analyse the dependency structure in that script. A dependency structure diagram is a visual formalism for a definitive script. It can be used to examine how redefinitions cause the update of values to propagate through different dependency structures. Consider, for example, counting compare/exchange operations in the sorting algorithm case-study in Section 4.4.3. Through this analysis, it is possible to determine ways to improve the stimulus-response time of interaction with models by reducing the upper bound for the number of updates.

The DM Model can be used as a basis for discussing and optimising update strategies. The block redefinition algorithm is one possible way to keep all values up-to-date given a block of redefinitions. Other update strategies include introducing threads for each vertex that regularly check to see if a value is up to date, with the potential to take advantage of parallel processing hardware. Another method is lazy evaluation, where every time a vertex x is referenced (its current value is requested), a tool checks to see if any recent redefinitions have effected its current value. All the dependencies of the vertex are updated prior the update of s. The DM Model can be used as a basis for comparing and contrasting different update strategies. Consideration of this model lead to the development of the block redefinition algorithm, which out performs existing update algorithms when more than one definition is redefined simultaneously.

The case-studies for finding minimum and maximum values and sorting using dependency maintenance illustrate how a dependency structure can be used as a form of algorithm. Empirical modelling encourages a user to program a computer without them realising that that is what they are doing. Incrementally constructing dependency structures, as in definitive programming, does not require a user to reason in the computation-oriented way that a programmer needs to when implementing an algorithm to perform a task. The process only requires the identification of patterns of dependency that reflect personal insight into the real world. This process supports the conflation of the roles of user and programmer.

9.2.5 Combining Definitive and Object-oriented Methods

In implementing new definitive notations via the JaM Machine API, object-oriented programming techniques are used to represent data types and operators of an underlying algebra for a programmer to implement new definitive notations. In general, this approach to implementing new notations is successful where all classes used by the notation are written specifically for use with JaM. Problems can occur when a programmer tries to integrate classes from other programming APIs with classes that extend `JaM.DefnType`. Classes in standard APIs generally have their own private data fields that can only be accessed by method calls. This means that there are aspects of the current state of an instance of such a class that are hidden from the implementor of the JaM class.

For example, the `java.awt.TextArea` class [CH96] can be instantiated to create a graphical component of a user interface that can be used for displaying text and enables a user to enter and edit text[7]. The current state of the text can be set and read at any time by calling methods of an instance of this class but there is no way to detect the text changing as a user types each character. The object structure hides the key presses from other classes and places the character onto the screen *automatically*. If there is an observed dependency between a certain pattern

[7]The input and output components of the *empirical world* client are text areas, as shown in Figure 8.9.

of characters and another observable, such as a child types a rude word and a teacher tells them off, the use of the standard `TextArea` component is inappropriate[8].

The object-oriented approach to programming requires a programmer to form abstract classifications of real world entities by choosing data fields to represent their internal state. It is then necessary to prescribe the methods for communication with instances of a class. With the JaM Machine API, it is still necessary for a programmer to classify real world entities but there is no need to commit to the methods by which classes communicate. Where the communication between objects serves to maintain indivisible relationships between observables, dependency relationships can be directly introduces using the JaM Machine API. This process requires a library of objects that represents the operators (classes that extend `JaM.DefnFunc`) to be used for this communication. The advantage of this technique is that it is possible for a user or programmer to experiment with the communication between objects on-the-fly through redefinitions. No compilation is required during this process.

9.3 Further Work

This thesis introduces new tools to support the development of new definitive notations and other utilities for empirical modelling. The purpose of the case-studies presented is to demonstrate "proof-of-concept" for these tools. There is the potential to:

- develop new case-studies and applications with these tools;
- improve existing case-studies such as *empirical worlds*, leading to more comprehensive JaM Notations and associated applications;

[8]Many graphical user interfaces use constraint solving methods to locate graphical components inside windows. Definitive programming would seem to be a good way to represent the dependencies between components, as demonstrated by SCOUT [Dep92]. To achieve this successfully using the Java API packages would require rewriting a lot of the classes of the API so that they extend `Java.DefnType`.

- improve on the underlying dependency maintenance mechanisms and data representations in existing tools.

In Sections 9.3.1 through to Section 9.3.4 some suggestions are made for further work based on the research presented in the thesis.

9.3.1 The Future of the DAM Machine

The DAM Machine implementation is intended to be a proof-of-concept tool. The DoNaLD to DAM compiler demonstrates how it can be used as the dependency maintainer that supports a definitive notation. It would be interesting to see if it is possible to implement new versions of EDEN, SCOUT or ARCA on the machine with the potential to support more efficient animation. There may be other new definitive notations that are well suited to implementation supported by DAM. Further investigation is needed into using the machine for maintaining dependency between groups of low-level words in definitive store that represent high-level values.

There is also the potential to investigate new software and possibly hardware implementations of the DAM Machine software. The latest ARM processor (the "*StrongARM*") has a clock speed that is more than five times faster than the one used for the work in this thesis and an on-board processor cache that is large enough to accommodate the DAM Machine assembly code[9]. There is also the possibility of investigating a new version of the DAM Machine that is implemented over the Java Virtual Machine [MD97] and is platform independent, although such a tool may animate models significantly slower than direct implementations that use native assembly code instructions.

[9]Some minor alterations may be required to use the StrongARM processor. It uses the same instruction set, but due to pipelining optimisations and on-board chip cache memory the effect of branch instructions can be different.

9.3.2 Support for Collaborative Working

Support for concurrent development and use of models is integrated into the JaM Machine API. This has been demonstrated so far in an elementary case-study in Section 6.5. Further case-studies are required to investigate the potential for the distributed use of this feature by both human and automated agents. For example, it should be possible to distribute the *empirical world* server to support concurrent engineering.

Some improvements should be considered for collaborative working by the JaM Machine API that should be considered. The current implementation of JaM only allows one agent owner to be associated with a definition. This can be extended to support both agent owner and group ownership properties for definitions in the same way that files are protected on a filing system. It should also be possible to include a time and date stamp for each definition to support better management of scripts of definitions. There is also an existing problem with the JaM class libraries in that the code in them breaks Java applet security rules [Har97]. It is not yet possible to run a stand-alone version of a JaM-based application in a web browser client without a server. This will require the implementation of a new cut-down and secure version of JaM.

9.3.3 New Shape Representations in Empirical Worlds

Two shape representation techniques have been demonstrated in the *empirical world* class library: CSG-style modelling and BCSO modelling. It is possible to extend the representation of shapes to include surface modelling with Bézier surfaces, B-spline surfaces, NURBS [WND97] and so on. *Empirical worlds* would benefit from support for the representation of two-dimensional shapes, that can be extruded to form shapes in three dimensions. For example, consider a shape formed by embedding a solid (filled) circle and a solid square, each in two separated and parallel planes in three dimensions. A three-dimensional shape between the two planes can be formed from the morph between the two profiles in two dimensions. Further investigation is required to compare variational and parametric modelling techniques with empirical modelling for geometry.

CAD packages typically address far more than just geometric shape modelling. There needs to be support in the *empirical world* classes for data exchange from and into other applications, through the use of standards such as STEP [Org98]. It would also be interesting to investigate the generation of CAM machine tool code to make modelled shapes through the expression of dependencies between the geometric shape and the possible motions of the automatic tool. Another common use of CAD models is to generate *finite element analysis* (FEM) models to simulate and analyse the effect of stresses applied to different pieces of geometry.

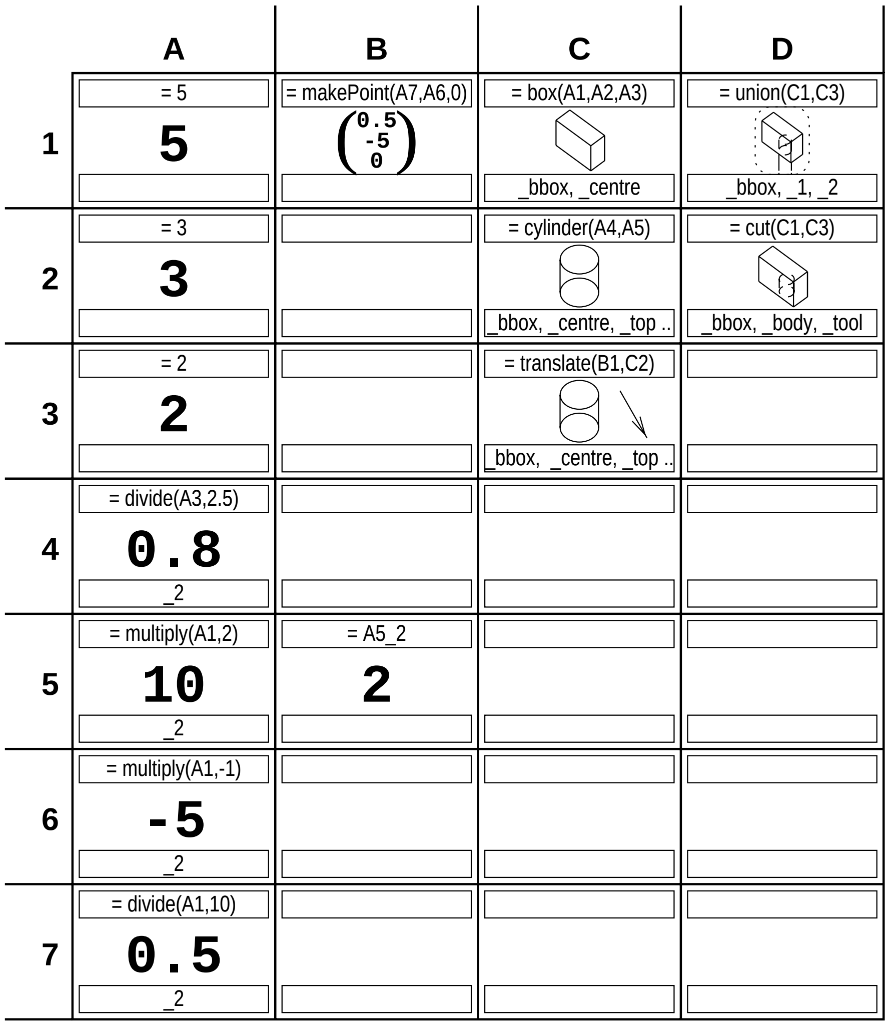
The rendering algorithm currently used in the *empirical world* tools produces rough images at the edges of a geometric model. Future implementations should integrate new algorithms for rendering implicit shapes, both through polygonisation and ray tracing. In this way, it should be possible to support the presentation of models that include fine features in geometry, such as the hair growing on implicit shape representations by Sourin et al [SPS96]. Although it is possible to increase the detail level for rendering a shape using *empirical worlds*, this is currently limited by the available memory in the *empirical world* client. This is due to the fact that the current implementation creates a string that is parsed by the VRML browser to render a shape. It is possible to communicate more directly with the VRML browser through instances of Java classes specifically designed to support the VRML external authoring interface, and future implementations should take advantage of these features.

9.3.4 Spreadsheets for Geometric Modelling

There are many similarities between definitive programming and the use of a spreadsheet. Rather than identifiers in a script of definitions, a spreadsheet provides an identity to a value by its location in a grid of cells (*A1*, *C3* etc.). Every cell has a value and, if it is dependent on the values in other cells, a formula defining its value. The research in this thesis relating to the representation of data structures and dependency is also relevant to spreadsheet applications. Spreadsheet applications are contrived to support financial and mathematical models and the data types represented in cells are numbers, strings of characters or dates and times. It is possible to link spreadsheet values to defining parameters in other applications, such as AutoCAD, using *object link embedded* (OLE) mechanisms that are provided in some operating systems. The proposal in this section is that it is possible to use the JaM Machine API to expand the range of data types supported by a spreadsheet application.

Consider the *empirical world* script for a hammer model shown in Figure 9.1. The identifiers have been chosen to correspond to references for cells in a geometric spreadsheet and an example of such a spreadsheet is shown in Figure 9.2. The cells of this spreadsheet are larger than those of a standard spreadsheet and each cell contains three sections. The top of the cell is its defining formula (definition), the centre is a representation of the value currently stored in the cell and the bottom section contains a list of references provided to the value in the cell. The box shape in cell **C1** depends on the values in cells **A1**, **A2** and **A3**. The list of references shown correspond to parameters that might be useful for a user to make reference to in the definition of other cells. For the box, this includes a bounding box that contains all the corner points for the box and a centre point.

The complete hammer shape is represented in cell **D1**. The image of the



	A	B	C	D
1	= 5 5	= makePoint(A7,A6,0) $\begin{pmatrix}0.5\\-5\\0\end{pmatrix}$	= box(A1,A2,A3) _bbox, _centre	= union(C1,C3) _bbox, _1, _2
2	= 3 3		= cylinder(A4,A5) _bbox, _centre, _top ..	= cut(C1,C3) _bbox, _body, _tool
3	= 2 2		= translate(B1,C2) _bbox, _centre, _top ..	
4	= divide(A3,2.5) 0.8 _2			
5	= multiply(A1,2) 10 _2	= A5_2 2		
6	= multiply(A1,-1) -5 _2			
7	= divide(A1,10) 0.5 _2			

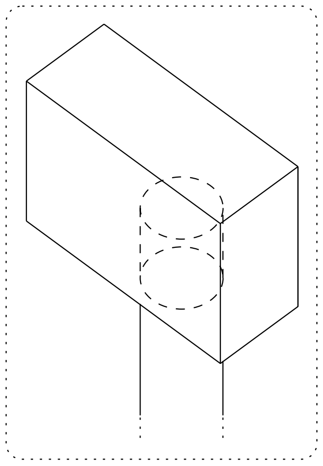


Figure 9.2: Geometric spreadsheet concept.

point set is only a *thumbnail* image. Ideally there should be a way to click on a cell with the result that a new window opens with a large rendered version of the geometric shape is displayed, as illustrated in the box at the bottom of the Figure 9.2. It should be possible to explore this shape in the same way that it is possible to explore a shape in a VRML browser[10]. This larger window can include graphical representations of the references to the geometry that can be selected by using a pointer.

A spreadsheet for geometric design, or any other application for which it is possible to write some JaM classes, allows a designer/modeller to integrate their work with other relevant data, such as financial data. It is possible to use a spreadsheet as a cognitive artefact for the explanation of a new design. A manager can experiment with parameters of a design indicated by a designer. They can describe a whole range of designs for assessment in one spreadsheet, rather than one design that will almost certainly require further modifications. The designer/modeller benefits from a tool that allows them to try out several *what-if?* experiments in the same way that a spreadsheet is often used as a way to test the financial feasibility for a project before its commencement.

In this thesis, the benefits of empirical modelling with geometry and empirical modelling for geometry have been discussed and explored. New tools to support general empirical modelling have been introduced and it shown that these tools can support three-dimensional shape modelling. The geometric spreadsheet is an obvious next step. It has the potential to support a large range of shape representations in one integrated application that can support complex data structures and interactive collaborative working on geometric models.

[10]A programming library like the *Java3D* API will be able to support this kind of interaction.