Notations for representing three-dimensional geometry and the upgrading of Sasami

Andrew Knight
0305767

Abstract

One of the key principles of Empirical Modelling is observation. The interface to a model is critical to a user's experience. This paper describes the various notations for describing graphics using Empirical Modelling techniques, including notations for specifying complex geometry as the application of functions on primitive objects. The Sasami 3D visualisation notation is discussed and an improvement to Sasami is described. This involves adding support for primitive objects such as cubes, cylinders and spheres, allowing the creation of 3D environments for models using a definitive notation.

1 Introduction

1.1 Background

Empirical Modelling (EM) was developed at the University of Warwick as a new method of modelling using computers based on observation an experimentation. A model is made up of definitions and dependencies. A "definitive notation" (Beynon, 1985) allows a modeller to create definitions to set up dependencies. This method of modelling allows the modeller to concentrate creating and interacting with the model itself, rather than with its supporting stucture.

The tool tkEDEN includes notations such as EDEN, DoNaLD and Sasami. EDEN[1] is a language that combines definitive notation with procedural statements. All other notations within tkEDEN use EDEN as a "back-end", often translating their statements into a series of EDEN definitions. This allows EDEN's "definitive side" to handle keeping dependencies updated and the "procedural side" handles actions based on the state of variables set by dependency (updating the display, for example).

1.2 Existing definitive notations for graphics

There are a number of notations for describing graphics. tkEDEN includes DoNaLD and Sasami and there exist others such as CADNO and ARCA.

The design of a definitive notation for two-dimensional graphics, DoNaLD[2], was originally defined in (Beynon et al., 1986). The notation allows for the display of lines and shapes and can be used to visualise models created in EDEN. Data types in DoNaLD include "int" and "real", as well as "point", "line" and "shape". A line is made up of two points and a shape is made up of a set of points and lines. Another key type is the "openshape" type. This can contain points, lines and other shapes (or openshapes), allowing complex shapes to be made from a combination of simpler ones. An addressing system is used to access child variables within an openshape.

DoNaLD also has some pre-defined shapes, such as rectangles and circles. The notation allows these basic shapes to be drawn by specifying the parameters (such as length or radius) of the shape.

All variables in DoNaLD can be either set to a constant value or set to be dependant on another variable. In the latter case, the display will update automatically whenever the variable is changed. DoNaLD variables are normally made to be dependant on EDEN variables to allow the display to visualise an EDEN model.

1.3 3D visualisation techniques

This section looks at various techniques for visualising models in 3D. Not all of these techniques have been fully implemented.

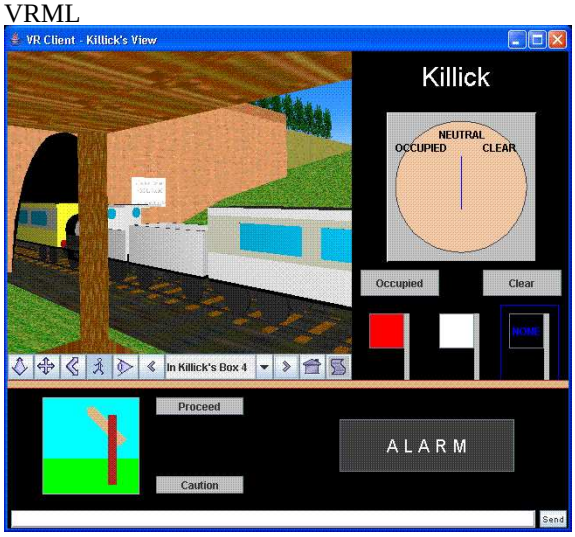
1.3.1 VRML

One of the more advanced models created with tkEDEN is the Clayton Tunnel model[3]. This uses a "dis-

[1] Evaluator of DEfinitive Notations
[2] Definitive Notation for Line Drawings
[3] See claytontunnelChanHarfield2005 in the EM project archive

tributed" variant of tkEDEN, known as dtkEDEN. This allows different agents to have a different view of a model stored centrally on a server. A Java-based tool was created for the model to allow it to be visualised in 3D. The tools uses VRML[4] to create the 3D worlds. Communication between the dtkEDEN server and client is intercepted by the tool (Woodforth, 2000) and used to create a 3D representation of the agent's observation of the model (see Figure 1).

Figure 1: The Clayton tunnel model visualised with VRML



The VRML client was developed before Sasami was a mature part of tkEDEN. A combination of Sasami, DoNaLD and SCOUT could now be used to develop a similar visualisation of the agents' perspectives of the model (although the user interface would have to be slightly different). This would have the advantage of using definitive notations to specify the entire model and not having to rely on external applications.

1.3.2 CADNO

CADNO[5] is a definitive notation for geometric modelling, outlined in (Beynon and Cartwright, 1986). It uses similar concepts to DoNaLD for constructing n-dimensional geometric structures. CADNO has three layers for different levels of abstraction, represented by three data types. A "complex" stores a list of labels representing abstract entities from which an object is constructed. A "frame" adds coordinates to represent a complex in cartesian space. A set of frames stores information representing the fundamental components and attributes of an object. An "object" combines information from frames to allow the visualisation of a geometric object.

CADNO does not currently have a working implementation. An interesting project would be the use of the current tkEDEN toolkit to create a working implementation of CADNO. The Agent-Oriented Parser (AOP) could be used to parse the language, and Sasami could be used to render the CADNO "objects".

1.3.3 Empirical HyperFun/Hyperjazz

The Hyperfun Project is "devoted to developing an open system architecture for functionally based shape modelling and its applications" (Adzhiev et al., 1999). These models, known as "F-rep" models, are represented using the high-level HyperFun language. They are mathematical models of geometry, created by applying functions to basic primitive objects. Set-theoretic operations such as "cut" can be applied to two primitives to create more complex objects. These operations are performed using definitions so if, for example, a cylinder was used to create a hole in a box, and the cylinder's size was subsequently changed, they hole's size would update.

Hyperjazz is a language used to represent F-rep models using definitive scripts (Adzhiev et al., 1996). Functions can be declared that perform certain operations on an object (such as 2D polygons or 3D solids). Objects can be defined as a combination of applications of these functions to basic objects (such as polygons).

1.3.4 The Sasami notation

Sasami[6] is the language used for visualising models in three dimensions within tkEDEN. Is was developed by Ben Carter for his third-year project in 1999. It is a layer between an EDEN model and OpenGL, and is "one-way", that is to say that it is purely for visualising models – interactions with the 3D represenatation cannot affect the underlying model. Sasami has types including "vertex", "poly" and "object", similar to DoNaLD's "point", "line" and "shape". A poly is made up of a number of coplanar vertices; an object is made up of any number of polys. Sasami does not, have a type equivalent to DoNaLD's "open-shape". If two objects are required to, for example, move together, they must each have an appropriate

[4] Virtual Reality Markup Language
[5] Computer-Aided Design NOtation
[6] Not an acronym!

dependency set for their positions to keep them stationary relative to each other.

Unlike DoNaLD, Sasami does not have the facility to create basic shapes such as cubes or spheres. It can, however, import 3D meshes that have been created in an external 3D modelling program.

2 An extension to Sasami

An obvious extension of the Sasami notation would be the addition of functional modelling elements such as set-theoretic operations on primitive objects. This would allow complex models to be specified in a purely definitive notation.

While the creation of a complete set of functional operations for Sasami is out of the scope of this project, the Sasami notation has been extended to allow the creation of "primitive" objects such as cubes, spheres and cylinders. This will allow the tkEDEN modeller to rapidly create a basic 3D visualisation of their model and apply existing operations (such as translation, rotation and scaling), without the use or knowledge of external 3D modelling software. This section provides details of the changes made to the Sasami notation to realise this.

2.1 Implementation

Sasami provides two interfaces: one between the Sasami code input and EDEN, and one between EDEN and OpenGL. Sasami notation is converted to EDEN code, which is parsed by EDEN, which in turns calls internal Sasami functions to render the display. It would therefore be possible to implement the code for creating the primitive objects as EDEN code and call existing internal Sasami functions, or to create new internal functions specifically for the task. The former method would require the creation of a large number of EDEN variables, which, for objects with a high polygon-count (such as a smooth sphere), would consume a lot of memory and processing power. The latter approach allows the algorithms for generating the geometry to be written efficiently in C, although the internal geometry would not be accessible to EDEN. This is not a major problem as primitive objects are unit-objects from which more complicated objects can be constructed. The latter approach was taken.

Three types of primitive objects were implemented: cubes, cylinders and spheres. Each has a number of parameters that can be modified to make them more flexible. See Table 1 for details of the parameters provided by each primitive object.

Table 1: Primitive object parameters

cube	length, width, depth
cylinder	length, radius1, radius2, segments
sphere	radius, segments

The parameters were chosen to make each primitive object as versatile as possible, while keeping them easy to use. The cylinder has two radius parameters – if one of these were set to zero, a cone could be created. Also, the "segements" value can be set to change the cross-section of the "cylinder", making a square or triangular cross-section object.

A primitive object in Sasami is a regular Sasami object that has some or all of its polygons automatically generated. Statements such as `object_rot` will work just as with any other object. Due to the individual polygons not being accessible from EDEN, two functions were created to apply materials to the primitive. These are described in Section 2.1.3.

2.1.1 The Sasami parser

A Sasami command is first parsed by the Sasami parser. This creates EDEN statements that are passed to the EDEN parser. The following is an example of how a typical Sasami command is processed.

```
poly_material mypoly mymaterial
```

This Sasami statement is used to apply `mymaterial` to the polygon `mypoly`. Sasami creates two eden statements.

```
_sasami_poly_%d_material is mymaterial;
```

This EDEN statement creates a dependency between an EDEN variable and the specified material (where `%d` is a unique identifier for this dependency).

```
proc _sasami_poly_%d_material_mon :
    _sasami_poly_%d_material {
        sasami_poly_material(
            mypoly,
            _sasami_poly_%d_material
        );
    };
```

The second EDEN statement creates a procedure to monitor the above variable for changes. Whenever a change occurs, the EDEN function `sasami_poly_material` is called with the polygon name and material as parameters. When EDEN

parses this command, it calls the build-in C function of the same name. This changes the internal value of the polygon's material, which determines how it is rendered by OpenGL.

2.1.2 Object deletion

Sasami stores unique identifiers (UIDs) in the EDEN variable for an object. When an object is changed, the UID is looked up, and the internal representation is updated. A problem was encountered with this method: if an object with the same name is re-declared, a new UID is generated. This new UID is then used internally to create and manipulate the object's data. This results in the old UID "dangling", as there are no EDEN references to it. The old object is still in the internal representation, however, so is still rendered by OpenGL. This leads to unsightly duplication of polygons when an object is re-declared with the same name.

This was overcome by storing the name of the EDEN variable as a string within the variable itself. The internal representaion of objects was changed to refer to them by name, rather than UID. Thus when an object name is re-used, Sasami can check whether it is in use, and delete the old object before creating the new object. New internal functions[7] were created to handle deletion of objects, polygons and vertices.

2.1.3 Internal functions

Three main internal Sasami functions[8] were created to create/update each of the three primitive object types. Each one checks whether the object name is in use, and if it is, truncates the polygons of the existing object before creating new ones.

In addition to these, three other utility functions were also written:

- `sasami_object_delete` (called through Sasami's `object_delete` command) makes use of the new deletion functionality and allows an object to be removed from the scene. It deletes the object, its polygons, and their vertices.
- `sasami_primitive_material` (called through Sasami's `primitive_material` command) sets the primitive polygons of an object to use the specified material. This function is necessary as the primitive polygons are not accessible from EDEN for applying materials.
- `sasami_face_material` (called through Sasami's `face_material` command) sets the material of a specific polygon in the primitive object to the specified material. Again, this function is necessary as the primitive polygons are not accessible from EDEN.

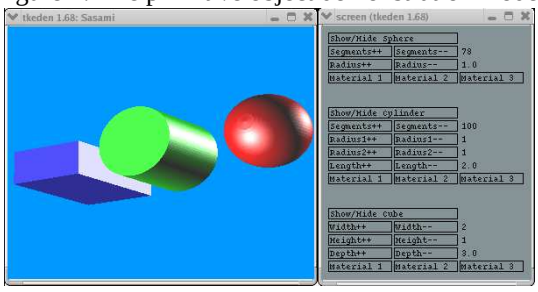
[7] Refer to the "structures.c" in the Sasami source code for further details of the implementation

[8] Refer to "functions.c"

2.2 Demonstration model

A model was created[9] to demonstrate the use of primitive objects in Sasami (see Figure 2). The model consists of definitions representing properties of three primitive objects – a cube, a cylinder and a sphere. This model is represented in 3D using Sasami primitive objects. The model can be manipulated by clicking buttons in the accompanying SCOUT window, which changes the values of the EDEN definitions. These changes can be observed in the Sasami window.

Figure 2: The primitive object demonstration model



2.3 Case Studies

Two existing models were studied, with a view to representing them in 3D using the new Sasami primitive objects.

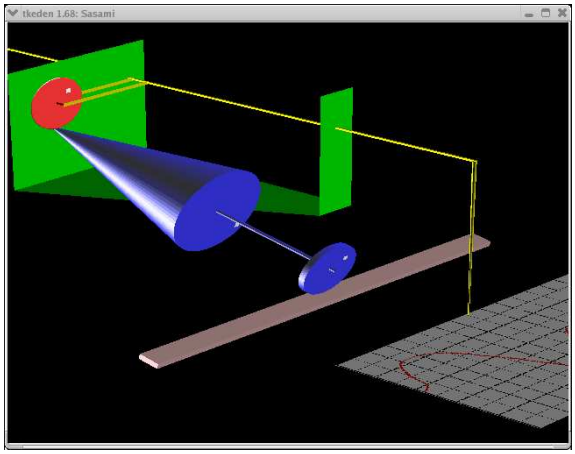
2.3.1 Planimeter

Charles Care's Planimeter model is one of the "showcase" models for the Sasami notation. It models a planimeter device – a mechanical device for measuring the area under a graph. The model consists of a series of dependencies linking various parts of the device – wheels, linkages and so on. The model is represented in 3D using the Sasami notation. Each component of the model is created in 3D either manually, by specifying vertices and polygons, or by loading from a Lightwave .obj file, which is created with 3D modelling software.

[9] See sasamiprimitivesdemoKnight2007

The aim of this study was to simplify the model by using primitive objects where possible. All objects in the model, with the exception of the carrage and the trace board, were implemented using primitive objects. The cone and wheels naturally lend themselves to the cylinder object, while the linkages used either cylinder or cube objects, depending on which was most convenient with respect to the coordinate system[10]. The carrage was kept as a collection of manually specified vertices and polygons due to its arbritrary shape not lending itself to representation by 3D primitive objects. The trace board was also kept using the existing system, as the current implementation of primitive objects does not support texture coordinates.

Figure 3: The planimeter model

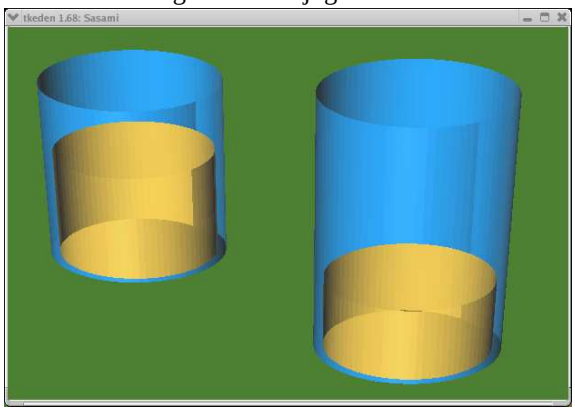


The resulting model[11] can be seen in Figure 3. The original Planimeter model used 1119 lines of code (7627 including model data). The new model allows a similar result to be produced with 902 lines of code, with no external model data required. It is hoped that this more efficient method of visualising models will make model-building a easier and faster process in the future.

2.3.2 Jugs

The Jugs model[12] is a model that represents two jugs of know size, each of which can be filled or emptied. The contents of one jug can also be poured into the other jug. The model is represented using a text-based display in a SCOUT window. Various versions of this model have been made over the years. A recent version, written by Anthony Harfield for a CS405 lab, was chosen for this study, as it makes use of EDEN clocks to efficiently animate the filling and emptying, rather than the resource-intensive while-loops of the older versions. This model is visualised as a line drawing in 2D using DoNaLD and SCOUT.

Figure 4: The jugs model



A 3D visualisation of this model was created using Sasami[13] to demonstrate how the primitive objects can allow for the rapid creation of 3D representations of models. The Sasami code is dependant on the EDEN variables and dependencies that make up the underlying model. The display updates its view of the model whenever the model is changed by user interaction with the SCOUT buttons. Figure 4 shows the 3D representation of Jugs in Sasami.

3 Conclusion

Section 2 details an extension to the Sasami notation allowing the simple creation of primitive objects. This extension improves the model-building experience by allowing the modeller to write smaller more efficient model code.

Further additions to Sasami could include the ability to create more complex solid objects through techniques such as extruding an arbitrary polygon, transformations such as shearing, and set-theoretic operations. This would allow complex models to be represented using a purely definitive notation (Adzhiev et al., 1996).

Sasami can currently only be used to visualise a model. A possible addition would be the ability to interact with the model directly using the 3D interface - for example, the pen of the planimeter could be dragged across the trace, or the contents of the jugs

[10] cylinder objects have their origin in the middle of one end, while cube objects have their origin in the centre of the cube.

[11] See planimeterKnight2007

[12] See jugsBeynon1988

[13] See jugs3dKnight2007

could be poured by moving the 3D representation of the jug. Such interactions could provide a richer experience for user of the model.

Acknowledgements

The author would like to acknowledge the work of Anthony Harfield and Charles Care, which provided a base for the models created and studied in this project, and to thank Meurig Beynon for his guidance and advice regarding the project topic and direction.

References

V. Adzhiev, R. Cartwright, E. Fausett, A. Ossipov, A. Pasko, and V. Savchenko. HyperFun project: A framework for collaborative multidimensional F-rep modeling. *Proc. of the Implicit Surfaces '99 EUROGRAPHICS/ACM SIGGRAPH Workshop, Bordeaux, France, 1999*, pages 59–69, 1999.

V. Adzhiev, A. Pasko, and A. Sarkisov. Hyperjazz project: development of geometric modelling systems with inherent symbolic interactivity. *In Proceedings CSG 1996. Information Geometers*, 1996.

W.M. Beynon. Definitive notations for interaction. *Proc. HCI'85*, pages 23–24, 1985.

W.M. Beynon, D. Angier, T. Bissell, and S. Hunt. DoNaLD: A line-drawing system based on definitive principles. *University of Warwick Computer Science Research Report #86*, 1986.

W.M. Beynon and A. Cartwright. A definitive programming approach to the implementation of CAD software. *Intell. CAD Systems II: Implementation Issues*, pages 126–145, 1986.

J. Woodforth. Interactive VRML front-end to dtkeden. *3rd Year Project Report, Dept. Computer Science, University of Warwick*, 2000.