DEVELOPERS MANUAL

CONTENTS -

---

1) INTRODUCTION

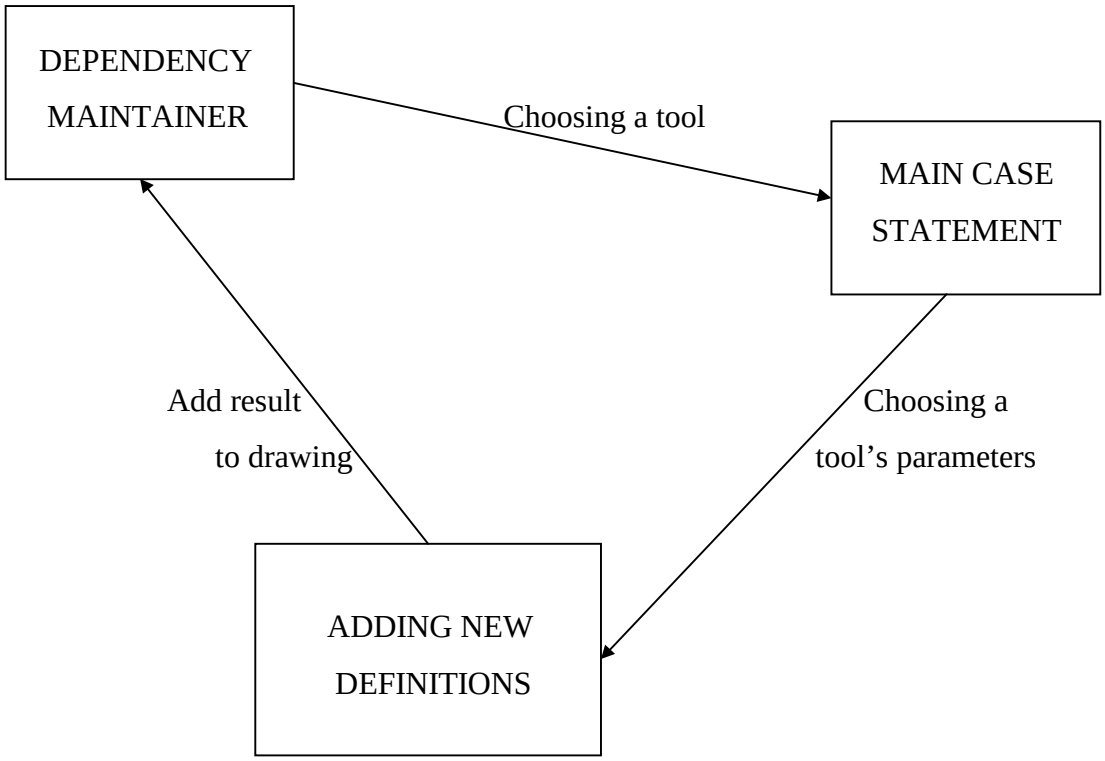
The coordinate geometry model (hereafter referred to as COG) is an environment in which 2-dimensional geometrical concepts can be explored. It is aimed at children from the ages of 10 upwards in its present form but could be easily adapted for younger children with more appropriate language in the help windows. Users can construct their own drawings using the drawing tools provided and perform transformations on the shapes they have created and watch them transform dynamically. There are also facilities for measuring lengths and angles on the drawings and watching how these change as shapes are moved around the editing grid.

2) MODEL STRUCTURE

The model is structured with three major components, each of which links with the others, as shown in the diagram on the next page.

The drawing on the grid is maintained by the definition manager and does not require the modeller to intervene to ensure correctness. The main case statement is called whenever a mouse event is received on the grid, be it motion or mouse button activity. It performs an action depending on the tool selected. Once the parameters for an operation have been chosen (e.g. two mouse clicks to specify a line) then a new

definition is constructed to represent this entity which is then handled by the definition manager. In effect the modellers job is to provide the link between the user and the definition manager.



3) ADDING A NEW TOOL

One of the most likely activities that a developer is going to perform to the model is adding a new tool to the system. This could be for use in a specific situation or a request from teachers or children for a tool they would find useful generally. Here I describe how to achieve this. Whilst reading this section it would be beneficial to have copies of the *geom.s, addtodrawing.e and actionsongrid.e* files available for consultation as they are the files that will need to have definitions and code added to them. The general sequence of steps is as follows :-

i) Create an icon for your tool (of size 40x30 pixels) and save as a .gif file.

ii) Add to the *geom.s* file the definition for the button. This will require a number of window definitions. You will need to decide where to put it on the screen as well. The first window is for the icon image itself. The second is for the label below the icon.

The icon may require an input box below it as well (e.g. the rotate tool, scale tool etc.), which is defined as a *TEXTBOX* window. Also you will need to add a triggered procedure when the button is clicked on to ensure that it highlights when selected, that the interactive help string appears and the *toolselected* observable is changed to your name for that tool (e.g. *toolselected = "rotate";).* If in doubt look at the many examples in the *geom.s* file, copy the definitions and change the relevant bits.

iii) Add to the *actionsongrid.e* file the entry in the case statement in the *Clickedoncoordinategrid* procedure. Each tool that requires mouse clicks on the grid to define the parameters for the tool will need an entry in the case statement. The easiest way of doing this is to copy the code from a different tool which has the same number of parameters and then change the relevant bits. The add label command requires one parameter, the add line requires two parameters and the view angle command requires three parameters. For commands that require more than three parameters look at the difference between the two parameter tools and the three parameter tool and how to add more parameters should be easily noticeable. If you are adding a drawing tool then to ensure that it "rubber bands" correctly look at the difference between the add line entry and the measure tool entry. The latter simply requires that you click on two points and then the measure is added. The former creates the line and then moves it around with the pointer until you fix it. You will need to add a procedure call to the operation to be performed once all of the parameters have been defined. These procedures can be found in the *addtodrawing.e* file.

iv) Add to the *addtodrawing.e* file the code which generates the new definition to add to the system. This will need to be done using the EDEN "execute" command. This is because the definition will need to be general to any shape and any entity in that shape. By this I mean that if you create a line then that line may be part of any shape and will be the n'th line in that shape. There are three actions that need to be undertaken to add a new definition to the system. The first is to find the existing number of the entity that you are going to add to the drawing (i.e. the number of lines, labels, circles etc.).

The code to do this is :-

```
/* get next label free */
 nextshape =strcat("_usershape_Example",str(currentshapeno),"_lb",str(1));
 z = 1;
 while (type('nextshape')!="@") { z = z + 1;
   nextshape=strcat("_usershape_Example",str(currentshapeno),"_lb",str(z));
 }
```

The second action is to declare the new entity in DoNaLD. This must be done since we are not permitted to define a DoNaLD primitive without declaring it first. We must also ensure that we declare the entity as part of the correct shape and in the correct viewport. An example of this is :-

```
/* create the label declaration in DoNaLD */
                    execute("%donald
                         within usershape {
                         within Example"//str(currentshapeno)//" {
                         label lb"//str(z)//"
?A_usershape_Example"//str(currentshapeno)//"_lb"//str(z)//" is
\"color="//str(measurecolour)//"\";
                         }
                         }
                         ");
```

We can then add the new definition to the system. I have chosen to do this in EDEN since then the definition will not need to be translated from DoNaLD to EDEN, which should reduce the time taken to add the definition to the system. It also means that the definition can span many lines of code since EDEN represents the end of a statement with a ;. DoNaLD however recognises an end of statement with a carriage return and hence for long definitions you must keep all the definition on one line which is a pain when editing and constructing it. This is usually the longest of the execute statements since the definition must contain all the relevant dependencies so that it will

automatically be updated by the definition manager when one of its components changes. It will also incorporate all of the parameters that are used to define it as well, so the more parameters the entity has the longer the statement (in general) will be.

/* define the label in EDEN */

execute("_usershape_Example"//str(currentshapeno)//"_lb"//str(z)//" is label(str(decplaces(dist("//str(firstpoint)//","//str(second)//"),dp)/1),scalar_div((vector_add("//str(firstpoint)//"s,"//str(second)//"s)),2));

");

This sequence of events is best understood by looking at one of the tools already in the system, such as the measure tool (some of the code above) and then using that as a basis for adding the new tool.

4) ACTIVITIES

One of the ideas that I talked about in my dissertation [1] was that any piece of educational software should have with it some pre-defined activities or investigations that can be put to immediate use by the teacher. I have developed three such activities that can be used with COG, there are many more, some of which are listed below. The three activities that I have developed are a curves of pursuit investigation, a rose pattern investigation and a solar system animation.

The curves of pursuit investigation *(curvesofpursuit.d)* allows the user to specify the start points of both the pursuer and the pursued and also the point to which the pursued is trying to get to. The speeds of both the pursuer and the pursued can also be specified. These quantities are the observables for this situation. It is by changing these quantities that the creation procedure will be triggered and the pursuit redrawn. This investigation could be the basis of work involving finding the minimum speed ratio between the pursuer and the pursued has to be in order for them to be caught. It should be the ratio of the length of the curve the pursuer takes to the length of the line of the pursued. To find an exact figure the equation of the line needs to be known and

then its length found using integration. To converge on an approximate value we can use trial and error. If the starting points are varied then the situation becomes different. Can a general formula be developed for finding the necessary ratio of speeds if the start points are known?

The rose pattern investigation *(rosepatterns.d)* also has observables that can be changed in the same way as the previous investigation. They are listed at the top of the file should you want to look at it.

The solar system animation *(planets.d)* has the nine planets of the solar system rotating around the sun. All of the relationships between the radii of the planets are correct and the planets are also orbiting the sun at the correct relative rates. There are many possible extensions to this model to make it a more accurate representation of the real solar system and these are detailed in the COG user manual.

What other investigation ideas are there? (You can probably add many more to this list yourself)

i) Rotational symmetry - providing a routine to animate a shape noting how often it fits into itself could form the basis of work on rotational symmetry and lines of symmetry.

ii) Incircles and circumcircles of triangles - providing a routine to define a circle by three points, all of which are on the circumference of the circle, and then specifying the centre of the circle. What is the distance between the centre point of the circle and the three corners? They are equal. How do you construct an incircle of a triangle?

iii) Ideas of angles in shapes - using the angle tool to investigate the interior angles of various shapes and deciding on a general rule.

There are clearly many more than this (indeed I have a page full of them in my notepad (too long to put in here)). Some would require new tools added to the system, some would simply use those that already exist. There is plenty of scope for investigations. Some may be non-geometrical in nature, some may even be non-mathematical. This is the power of the Empirical Modelling environment, it lets us perceive the model in any way we desire.

5) PROBLEMS

A major aim of this model was to be the framework for further work on it. I wanted to provide a system with as much functionality as possible, but with the time constraints imposed I had to settle for providing a subset of the necessary tools. However I was hoping to leave the model in a state where new tools could be added fairly easily, be they geometrical, other mathematical or other completely different tasks. There are however a few problems with the COG model which need to be explained and corrected before much more development work takes place. I shall outline these problems, with possible hints of solution where known :-

i) The major problem seems to be with SCOUT. At the beginning of a drawing session the help window and command history windows both work as they should. However when the screen list is altered (which happens when a dialogue box appears on the screen (such as the fix point, are you sure or current coordinates boxes) these windows, and the highlighting of the tool selected simply freezes. I do not have any idea why this does not work unless there is an underlying tcl problem with displaying the windows. It is actually a modelling error, the dependency is being removed so the screen is not being updated.

ii) Clicking on a transformation tool with no elements in the current shape causes a parse error due to a cyclic definition being generated. This is to do with the searching for the next shape code. It can be solved with a flag that registers whether any elements have been created and only allowing the transformation to occur when it is set.

iii) The creating a new worksheet tool is a big kludge. At the moment it simply sets all the points to very large negative numbers. However when this has been performed a number of times there will be performance degradation since the definitions are still present in the system. A definitive system is not very well adapted to removing definitions from the system since doing so may cause other dependencies and definitions to change or become invalid. A way of removing all entities or simply a single entity needs to be found for the model to be useful to children who will make mistakes and want to delete some of their shapes. Each definition generates three parts. These are its attribute definition, its actual definition and its drawing procedure.

Finding a way of removing these three could prove successful as they seem to possess no side effects.

iv) The TEXTBOX type in SCOUT allows the capture of input data parameters that are required by some of the tools such as translate and rotate. However if the user types in a value, hits return, types in a new value and then hits return and then selects the tool and performs the operation a DoNaLD error will be returned since all of the user's text is saved, including both carriage returns and both numbers. When this is added to an EDEN execute statement which is in DoNaLD (e.g. execute("%donald .....");) the first carriage return is taken to be the end of the statement, which will in general generate a parse error. To solve this problem some way of removing the carriage returns from the TEXTBOX string and taking only one of the values needs to be found.

v) The inclusion of continual mouse definitions has caused some problems. When the mouse is over the grid the first entry in the mouse list varies between 0,1,2,3,4 and also sometimes very large numbers. This is irrespective of which mouse button is actually held down. The best way to understand this is to run the model and have the history window visible. Play with the mouse over the grid and select various tools and watch the first list entry change to sometimes incorrect values. The upshot of this is that to ensure smooth dragging the first entry has to be ignored which means that when the right mouse button is pressed down the coordinate box appears but if the mouse is moved then the button released the point will move with it. This problem preferably needs to be sorted out at the mouse level otherwise many problems with mouse inconsistencies may occur.

vi) Rotation at the moment can only be performed anti-clockwise. This is mainly because this is the way DoNaLD is implemented. It is simply a matter of giving the user a choice and then if they choose clockwise to subtract the value from 360 degrees and perform the animation in the opposite direction.

6) EXTENSIONS

There are many possible extensions to this model. Some of them are described in my MSc project dissertation [1] and are the most important for the model to fully meet its aim of enabling children to explore geometrical ideas for themselves. Ideas for future work include (i,ii,iii taken from [1]) :-

i) Construction tools - midpoint of a line, intersection point, parallel lines, perpendicular lines and bisectors, angle bisectors, fixing points to lines and circles. Shapes such as ellipses, parabolas, conics, arcs and regular polygons.

ii) File handling tools - saving and loading of drawings and parts of drawings as DoNaLD script. A useful extension would be to save drawings without the Cartesian representation of the points so that the scripts could be incorporated into users models with no modification. The environment would then be also a DoNaLD resource editor.

iii) Operation tools - area of a shape, perimeter of a shape (both appear on the interface but due to difficulties in deciding how to calculate the areas and perimeters of self-intersecting shapes were shelved from my project), tessellation tool, equation tool and question tool (e.g. Is this line parallel to that line? Yes/No)

iv) Better and customisable help - At the present time the help window text is not suitable for young children. It would be better to allow the teacher to select a help level from a box so that the language was suitable for the children they were teaching. This would require writing a number of different help levels with suitable language which would need input and suggestions from teachers.

v) Records of work - One of the wishes of the teachers that I consulted during my project had was to be able to have a record of the work that the children had undertook. There were two strands to this idea. The first was to improve the history list window to incorporate all of the commands that the child had performed. This would be not too difficult, it would simply require not deleting the first entry of the list. The trickier part would be to display the last 10 entries (I am not sure whether the definition (**command_history = strcat(His[His#-10],His[His#-9],... , ... ,His[His#]);**) would generate an error or not). The second wish was to be able to print out the grid to

the printer. I am unaware of any model that prints anything to any printer but it may be possible.

vi) Multiple worksheets - Instead of having one grid to edit on, a user could create multiple drawings on multiple grids and cut and paste between them as much as they desired. This would require being clever with creating definitions for the correct viewport. This should not present too many difficulties. The more obvious problem is that having multiple grids will increase the number of definitions that the system has to contend with considerably.

vii) Floating window based interface - It would be more familiar to teachers if the environment was a windows based display where they could manipulate the windows themselves, by moving them, sending them to the back of a stack or closing them altogether. It would be necessary with a large number of tools to have floating toolbars because otherwise we will run into screen space problems. Defining a generic window type which allows the user to move, resize and close it would bring the Empirical Modelling environment more in to line with what teachers expect from modern software. This may or may not be a good thing, it is just more likely that they will use something that has surface familiarity to them (as defined by Pratt in [2]).

viii) Colours and attributes - DoNaLD allows us to modify the presentations of any element of an openshape and currently with the COG model this can only be done by an experienced user through the input window. There are observables that represent the current colour for many of the operations and the colours of translated, scaled, enlarged and reflected shapes can all be changed. However there is no way for the user to currently use the interface and mouse to change the presentation aspects of their drawing. This is very important if the model will be used by very young children who will want to see a pretty looking drawing. With no way of filling shapes and choosing nice colours this will not happen.

CONCLUSIONS

The COG model has a lot of possibilities for future work with it. When changing it ensure that you keep in mind that your additions should not comprise the model as a whole in being easily extendible. This flexibility is a quality of the model that I would like it to retain through any future improvements that may occur with it.

References

[1] C.Roe - MSc project dissertation - Dept. of Computer Science, University of Warwick -1999.

[2] D.Pratt - Phenomenalising the Stochastic - The design of a virtual domain of abstraction - http://www.fcis1.wie.warwick.ac.uk/~dave_pratt - (1999).

.