A 3D Lego Emulator with a Child-Friendly Interface.

Richard Stone.

BSc (Hons) in Mathematics and Computing 2005.

A 3D Lego Emulator with a Child-Friendly Interface.

Submitted by Richard Stone.

COPYRIGHT

DECLARATION

This Dissertation is submitted to the University of Bath in accordance with the requirements of the degree of Batchelor of Science in the Department of Computer Science. No portion of the work in this dissertation has been submitted in support of an application for any other degree or qualification of this or any other university of institution of learning. Except where specifically acknowledged, it is the work of the author.

Signed:

Abstract

In this dissertation, a proposed idea to solve the problem of brick placement is investigated and then subsequently implemented using a combination of Java and the Java 3D API. The program also aims to be user friendly and as accessible as possible and thus takes the form of an applet. Background material is discussed as well as a detailed specification, design and implementation process.

Acknowledgements

I would like to thank the following people, firstly, my supervisor Professor Davenport for agreeing to supervise this project, Secondly, John, Alex, Chris and Barrie without whom lectures would not be as interesting and finally, the debating society and the canoe club for providing me the best possible breaks from work.

Contents Page.

Chapter1: Introduction

1.1 Project Description

Lego™ is a block building game primarily aimed at children and young teenagers. The game lends itself to model building well, but there are very few pieces of software that allows Lego™ models to be built in a virtual environment. This project looks to, using an appropriate graphics package, design and build a user-friendly application to enable models to be built and then displayed in 3D. Such an application will be configured with children as the target user group, and hence feature an intuitive and straightforward interface.

A brief search on the Internet has located a few examples of model building software, not least Lego's™ own version (www.lego.com). But this, and all similar software, suffers from bad brick placement. This application is designed for children and hence the ease with which placing bricks is possible is crucial, part of this project will be looking at ways in which this can be improved.

1.2 Project Aims and Objectives

1.2.1 Aims:

These are the main high level aims, structured to give an overview of the program and the project requirements.

1. To identify the best graphics package for the problem after researching and evaluating the options.
2. To identify the best language for the problem after researching and evaluating the options, keeping in mind the choice of graphics package.
3. To locate suitable graphics libraries of Lego™ blocks or a similar alternative.
4. Briefly investigate Human-Computer Interaction in order to ascertain which interface components younger users find easier to use.
5. To complete a literature review that covers all the research undertaken.
6. To design and program an interface that displays, in 3D, Lego™ bricks.
7. This interface is to be accessible and easy to use by children, yet still have full functionality.
8. Investigate possible methods of brick placement, collision detection and brick stacking; then implement the most appropriate solutions in each case.
9. Complete a dissertation on the program.

1.2.2 Objectives:

Broken down further, these objectives shows the steps necessary to complete the project, they appear in a generic chronological order, but do not have to be precisely carried out in this way (see project plan for more details).

1. Examine the necessary specifications for the program such as machine independence, OS compatibility.
2. Research into TCL/TK, OpenGL, CAD and other graphics systems using above specifications to determine the most suitable graphics package.
3. Research and choose the programming language that which suits objectives (1) and (2).
4. Learn, if necessary, how to use the programming language and the graphics package to the level required for producing the application.
5. Begin building a Lego™ library, either by creating own libraries or locating existing libraries.
6. Research into HCI, focusing on how children interact with user interfaces, and which components they prefer to use.
7. Investigate the best brick placement method; with regard to detecting where bricks can be placed.
8. Investigate methods of collision detection, so user cannot place bricks over other bricks.
9. Investigate methods of stacking bricks, so bricks can be built vertically in a 3D environment.
10. Design an interface based on HCI research undertaken
11. Begin coding application, functions to include are:
    a) A 3D view.
    b) A "Rotate" option.
    c) A method of brick selection.
    d) A colour selector.
    e) An "undo" option.
    f) A "delete block" option.
    g) An Instruction guide.

12. Program the basic interface, and then add the basic brick selector, which will have one brick that can be placed repeatedly.
13. Test this program to see if it works by building basic 3D structures, and if it breaks by attempting collisions.
14. Slowly add functionality to program, possibly in this order.
    a) A "Rotate" option.
    b) A "delete block" option.
    c) An "Undo" option.
    d) The full set of brick libraries.
    e) Colour selection.

15. Briefly test the application after each new section is added in order to ensure the new component works as required.

16. Complete program by tidying up the interface, writing brief instructions and taking care of any other small details.
17. Test application rigorously and fix any problems or bugs that occur.
18. Invite users unfamiliar to the program to use it, and obtain feedback on interface and ease of use.
19. During project, document any problems/issues that arise as well as overall progress.
20. Complete dissertation as final part of the project plan.

1.3 Waypoints and Endpoints

Table 1.1 shows 9 waypoints that are milestones in completing the project, each one represents a logical progression in the program, and the certain points (4, 5, and 8) serve as viable end points if the project encounters unforeseen problems.

Waypoint	Expected Completion Date
Research Complete	End of November.
Literature Review Complete	December 13th. (set deadline)
Basic building systems in place	End of Consolidation Week.
Simple Lego™ builder working	End of S2, Week 2 (28th Feb)
Finished Program, but no testing/feedback.	End of S2, Week 4 (14th Mar)
Program Finished with testing and feedback	End of S2, Week 5 (21st Mar)
Dissertation first draft completed	End of Easter Break (10th April)
Dissertation final draft completed	End of S2 Week 8 (1st May)
Dissertation checked, corrected and ready to hand in.	May 16th (set deadline.

Table 1.1

.

Chapter 2: Requirements Analysis.

2.1 Introduction:

This project is development based as opposed to research based, however, it is critical that the necessary research is undertaken in order to find the required information that will help select the most appropriate tools and methods for the design of the final programme. The purpose of a requirements analysis is to establish the services the system should provide and the constraints under which it must operate (Sommerville, 1992).

The requirements analysis will be looking at exactly what the program should be designed to do based upon the problem statement. The requirements specification will be concentrating on the two key issues for implementing the project, which programming language is used and secondly which graphical package is used. Then it shall layout exactly what the application can do with the chosen language and graphics package.

2.2 Problem Statement.

Lego™ is a block building game primarily aimed at children and young teenagers. The game lends itself to model building well, but there are very few pieces of software that allows Lego™ models to be built in a virtual environment. This project looks to, using an appropriate graphics package, design and build a user-friendly application to enable models to be built and then displayed in 3D. Such an application will be configured with children as the target user group, and hence feature an intuitive and straightforward interface.

A brief search on the Internet has located a few examples of model building software, not least Lego's™ own version (www.lego.com). But this, and all similar software, suffers from bad brick placement. This application is designed for children and hence the ease with which placing bricks is possible is crucial, part of this project will be looking at ways in which this can be improved.

2.3 Requirements Analysis.

Somerville (1992) defines system goals as generic high level characteristics that the program should exhibit but which are not necessarily testable. Extrapolated from the above problems statement are the system goals for this project. They will serve as a starting point for an analysis of user requirements from the software.

- Easily accessible to novice computer users.
- Intuitive to use with easy input methods for building models.
- Sufficient functionality and flexibility to manipulate existing blocks

To further define the requirements, it is essential to choose a programming language and graphics package which will provide a base from which it is possible to achieve the above system goals and the aims and objectives of the project. Then the final part of the requirement analysis of the project can be broken down into the following categories (Sommerville, 1992).

- System Implementations.
- Functional Requirements.
- Non-functional Requirements.

2.3.1 Language Options.

For any piece of software, the choice of language is critical. There are many programming languages available to write any piece of software, and many languages come with or can be combined with graphics packages to provide a solution. Each language has its own advantages and disadvantages; here we look at the most common languages.

Java

Java is a relatively modern object oriented language developed in the 90's by a company called Sun, it was based on C++, however, this is mainly a syntax similarity, Java is a greatly simplified version, and in fact has more in common with languages such as Lisp (Flanagan, 2002). Its main strength is its portability; java is compiled to Java byte code, which can be run on any platform which has a Java runtime environment. Java comes with its own 3D graphics API, this takes an object oriented approach which is different to the procedural approach used in OpenGL. The official Java binding to OpenGL is called JOGL. Obviously it is the only language that supports the Java3D API

C

C is described by Lawlor (1995) as one of the most powerful languages in existence; it is certainly the language in which the developers of OpenGL chose to define their API. C was first developed in 1973 and became standardised as ANSI C in 1983 by the American National Standards Institute. C is mainly used in UNIX applications, and in fact the UNIX system itself is written in C. C is a low level procedural programming language with a very simple core language, the important functionality is handled with the use of library routines.

C++

C++ is a superset of C, an expanded version designed in the early 1980's; its aim was to have the same features of C but to be an object oriented language instead of a procedural language. C++ is a large language (much larger than C) and is used to develop large complex programs which C cannot manage so easily.

Pascal

While Pascal is a relatively popular language, no known OpenGL bindings are known to exist. There exists an extension called Turbo Pascal, but its graphical capabilities would not be sufficient for this project.

FORTRAN

FORTRAN is the oldest high level language; it was developed in the late 1950's. However it is mainly used for scientific applications, and is a very powerful tool for numeric calculations, indeed, research has shown that FORTRAN is still more efficient than Java, C and C++ for numerical computing (Moreira et al, 1998).

2.3.2 Graphics Packages

Computer graphics are an increasingly large part of computer software, especially in commercial software. The final product of this dissertation will use graphics as a way of communicating information, and thus finding the best graphics package is a key element of this problem. The optimum package needs to be selected, which has all the functionality required, but is convenient to use and as flexible as possible. The following standards are the most common currently available. The problem lies in the fact that graphics is a rapidly developing section of the industry, identifying the latest techniques can be an onerous task.
Most programming languages come with their own capabilities for perfunctory graphics (for example the AWT in Java). However, these tend not to be as advanced as specific graphics interfaces designed to run on multiple platforms or with multiple languages. It is at these graphics packages that we turn our attention to, before looking at the most suitable languages for coding our software program.
Whilst covering most of the major graphics standards, the following list is by no means exhaustive; however, the remaining toolkits are limited in scope, specifically designed for certain platforms, no longer supported or need to be licensed (which excludes them from this project).

OpenGL

OpenGL is a platform independent environment for the development of 2D and 3D graphics and is strictly defined as "a software interface to graphics hardware" (Wright and Sweet, 2000). It is the most widely accepted industry standard, and can be used in conjunction with most programming languages such as Java, Python or C. OpenGL is an applicable programming interface (API) which allows different operating systems using different hardware to produce consistent graphical images. Originally designed by Silicon Graphics Inc (SGI), OpenGL is now guided by the OpenGL Architectural Review Board, which is an

independent industry consortium, which has helped it remain vendor neutral and widely available. OpenGL is also flexible in that it does not rely on hardware being compatible, although most is, as OpenGL instructions can be either executed on hardware or run as a software routine.

Direct3D

Direct3D is actually one of many applications that form DirectX, the Microsoft graphics standard; direct3D is the part that integrates 3D into applications. Direct3D is completely hardware independent, but can use it if support is given, if not it can run an equivalent implementation in software. The biggest drawback of Direct3D is that is can only be used with Windows systems. It is similar to OpenGL in many ways, such as its library structure and its consistency; however, it has been developed by Microsoft, and thus specifically tailored for certain software and hardware configurations.

Tcl/Tk

Tcl stands for Tool Command Language and was developed as a scripting language which can be used either as a standalone application or embedded in programs written in the C language. Tk is a graphical toolkit (amongst other extensions) and features graphics and object-oriented programming which is used for delivering high performance Graphic Use Interfaces (GUIs) very quickly. Tcl and Tk now nearly always come together in one package and are available as an open source language. Hence all Tk applications are written in Tcl, which is turn is often embedded into other languages. Tcl is platform independent, and was originally designed so that extensions could be written in C; it can also be integrated into languages such as Java. It runs on most UNIX-compatible systems, Windows and Macs.

GTK+

GTK+ is another multi platform toolkit which produces GUIs. GTK+ is based upon three libraries, GLib, Pango and ATK. GLib is the low level library which provides data structure handling for C, portability and much of the interface functionality. Pango mainly handles text in the GUI and the ATK library deals with accessibility through a set of interfaces. GTK+ was a part of the GNU Image Manipulation Programme and thus was written for UNIX, there is a windows port available, but they are not exactly the same. Along with Qt, this is the most widely used of the open software packages.

Qt

Qt is similar to GTK+, except that it uses C++ and not C. Specifically; it is a C++ application development framework, which has multi-platform functionality. One of the main differences from GTK+ is Qt designer, which is a GUI builder. Qt design allows the programmer to design interfaces, which it then automatically displays and provides code for.

FLTK.

FLTK (Fast Light Tool Kit) was originally designed to be compatible with form Libraries written for SGI machines. Now it works with C++ and works on a wide variety of platforms, however it is still in development.

Java3D API

Java 3D is a high level, scene graph based API. It uses either DirectX or the OpenGL low level API to take advantage of 3D hardware acceleration. It aims to strike a balance between providing powerful modification and editing controls and not forcing the programmer to use the lowest levels of graphics applications. It is also one of the first of the 4th generation of graphics which uses scene graphs and behaviours means that programs are easier and less complicated before, but using Java3D means that the programmer surrenders control over rendering. There is also an improvement in efficiency which means that larger more complex programs take up less computing power.

2.3.3 Choice of Language and Graphics Package.

With FLTK being in development still, there are unlikely to be many libraries readily available for use, its limited functionality is also a reason not to select this package. GTK+ and Qt both were written for UNIX and although there is a Windows port available they are not the same which would make it inaccessible to many people. Direct3D has the same problem although it the other way round, as it only works in Windows. As for the languages, FORTRAN and Pascal are not really designed for graphics, based instead in mathematical and sequential programming.

This leaves three reasonable choices, OpenGL with C or C++, Java with Java3D API or Tcl/Tk. each has their own merits, but only one can be chosen, so after careful consideration I chose Java with Java 3D as this offers a greater degree of flexibility than the others and the Java Virtual Machine (JVM) makes it very accessible to computer users, which was one of the original system goals.

2.3.4 System Implementations.

With Java as the choice of language, operating systems are no longer a problem due to java compiling to java byte code, which can be read with any java virtual machine. Unfortunately, if the Java Runtime Environment is bundled with the software, there is no guarantee that the right runtime environment for every user's operating system would be bundled, so to avoid this and to ensure maximum accessibility, the program should be designed as an applet. Then any website it is featured on can have a link to the java homepage to get the necessary downloads.

The available computing resources of the end user are also a factor, as 3D graphics can be very processor and RAM intensive. Whilst the executable program itself is like to small and not take long to download/load, if the user experiences slow refresh rates or a delay of the processing of input, they are likely to become frustrated, an acceptable refresh rate is about 20fps. This can be controlled by limiting the total number of blocks that can be used in a model and/or the quality of the bricks themselves.

2.3.5 Functional Requirements.

The existing problem with Lego™ builders is that it is very difficult to place blocks. This is partly due to the problems that arise from representing a 3D world in a 2D screen. Most 3D graphics systems use a method called perspective projection to produce a stylization of three-dimensional reality. This works very well when viewing objects, and indeed, for viewing 3D objects, this is a more popular choice than planar geometric projection (Watt, 1993). Perspective projection also works well for picking, a method whereby upon a mouse click, a pick ray is shot into the 3D environment until it hits an object (in this case a Lego™ brick) and that brick is then acted upon with a method.

This works fine for selecting objects already on the screen, but when it comes to placing new bricks onto the screen, the pick ray may not coincide with anything and thus where to place to brick can be ambiguous. The current Lego™ programs confront this by providing a base plate from which to build on (similar to the x-z plane in the 3 dimensional Cartesian co-ordinate system. The other constraint that is put into place is that new bricks can only be built onto the base-plate or on top of or near other bricks. This is an unfortunate limitation as it means that models have to built from the ground up and separate parts of a model aren't easy to build (for example, a model of two people throwing a ball couldn't have a ball in mid air). Of course one way round this is to build a line of bricks to the ball, build the ball and then delete the bricks, but this quite laborious and not intuitive.

The other problem with perspective projection is "jumping", which is where the program is trying to select the most appropriate place for a brick to be placed. Most programs have a "shadow" brick which is an outline of a brick, which moves into the place where it go if the user was to click the mouse at the precise point. It does this by taking the VDC (visual display co-ordinate) of the mouse and then applying a formula to work out where that brick should go. However, occasionally this can lead to several problems; occasionally, a VDC will provide two places where the brick can go and jump between them. The main problem however, is that sometimes a user will want to place a brick in a certain location and find that there are no VDCs which place the brick in their location, or if there is, it is sufficiently small and reasonably hard to find. Often it will be a long thin area of the visual display which is hard to click the mouse in with a high accuracy rate. This leads to many misplaced bricks, especially with inexperienced mouse users (i.e. children).

One of the reasons that jumping occurs is that these programs allow you to build at more than one layer at a time, hence there is often confusion when the user is trying to build behind an existing brick on the at the same height or if they are trying to build a second brick directly on top of the first.

So the part of the interface where the bricks are laid must use methods where the placing of bricks is unambiguous and clear, even at the expense of flexibility. There are two commonly implemented systems that can be used for this which will be discussed in more detail in the design process.

As expected, all literature recommends that any product aimed at children should be colourful enough to attract their attention, and simple, clear and easy to use, but also stimulating enough to keep their attention. Furthermore, Hong (2002, citing Borgers, Leeuw & Hox, 2000) suggests that at a young age the main focus of a child's intellectual development is using symbols, such as pictures and words, to represent ideas and convey objects. Hence the final product should try to replace menus and lists with graphical icons (such as a picture of a floppy disk replaces a "save" button).
Preece et al (2002) say, "A main observation is that people find it very hard to learn by following sets of instructions in a manual. Instead they much prefer to learn through doing." Thus the interface should try to allow flexibility and encourage the user to be able to explore each function to determine its purpose, and this underlines the need for an undo button which can be invoked iteratively.
So the program must be a mainly point and click driven interface, with simplicity and intuitiveness built in.

Lego™ bricks' Appearance

The appearance of the bricks in this program is extremely important as a potential use for this application is as a marketing tool for Lego™ or as a showcase for their product. Either way, humans look for realism and clarity in computer graphics. Hence it is required that the bricks take on a similar form to their real world counterparts. This form is that of a small rectangular cuboid and comes in many different colours; an example of bricks can be seen in fig 2.1.

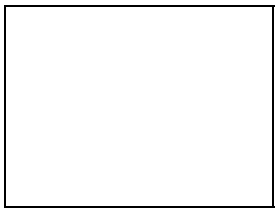
Fig2.1

These bricks have no transparency or any shininess and thus there is no need for texture mapping, colour gradients or other computationally expensive graphical enhancements. A simple colour with the appropriate lighting will suffice. With regard to the colour, either a set of predefined choices can be made or the user can input RGB values to select a colour.

The final choice to be made for the appearance is that of how objects appear on the screen. The two commonly used projections are perspective projection which has already been mentioned in the interface section and orthographic projection. Both projections have their merits with perspective being more realistic and giving a greater sense of depth perception but orthographic being more intuitive and easier to build with.

User Choice

To improve the functionality and usability of the program, users will be able to select options about the Lego™ bricks such as size, shape and colour. In order for this to be possible, the program will need a large library of different style of bricks. However, providing too wide a range of objects would require too much time and effort to locate the libraries (or create them from scratch) and integrate them into the program.

2.3.6 Non Functional Requirements Analysis

System Testing

Testing will be an ongoing thing as when each function is added it will be tested. Chapter 5 will have a formal test plan to test the outcome of the program as a whole in a formal manner. Testing will consist of both positive and negative testing.

Security.

As this program will not hold any sensitive data, there will not be any steps taken to protect the program from damages caused by viruses, hackers or other malicious intent. Lemay and Cadenhead (1999) note that Java applets run under a "better safe than sorry" security model so an applet cannot read or write to files on the user's file system, run any programs on the users system [or] load programs stored on the users system such as shared libraries. However, these restrictions should not present a problem as all the information will be contained within the program.

Software size.

The program itself should be reasonably small as it will only contain executable code and the JVM will not be added to the software package but linked separately if it is required.

Resources

Graphics system textbooks are widely available, Bath's Library and Learning Centre have a sizeable selection, (with titles available on OpenGL, TCL/TK,

Java, C, CAD etc), and specific titles can be ordered on inter-university loan if required. Also, the Internet has extensive literature available on most graphics systems.
Programming textbooks are also available from Bath's Library and Learning Centre, with Internet documentation more thorough than that of graphical systems. This includes tutorials if learning (Objective 4) is necessary.
HCI is covered in at 30 textbooks and journals in Bath's Library and Learning Centre in such books as Schniderman Ben - "Designing the user interface: strategies for effective human-computer interaction, 4th edition". This should be sufficient as research in this area is to be brief. Dr Hilary Johnson is also an expert in this field, and hence might be able to locate the most appropriate literature.

Hardware:

As this program is to run on most home PCs, any BUCS computer should be sufficient, as should my personal desktop PC. As a guide to the minimum specification of a PC, the Lego™ digital designer needs an operating system with a 450MHz Pentium processor, a 16MB graphics card and 128MB ram. However, their programme has many features, such as an online ordering system to buy the blocks used in your creation.

Software:

Nearly all languages are available free of charge, as are most graphics packages, the only foreseeable problem could be in locating existing libraries of Lego™ bricks. All of these languages and packages are compatible with UNIX and Windows and can be implemented on a low spec system. Although BUCS doesn't allow the installations of language implementations or programs, this shouldn't be a problem as I shall be working from home

Other Constraints

The average time to be spent on this project is approximately 300 hours, spread over the course of the academic year from October to May. During this time there is a single programmer with a reasonable knowledge of java, but no knowledge of any 3D graphics toolkits, APIs or specification, including Java3D. These resources will limit the complexity of the program but it should be feasible provided there is good time management by the student and support available from the project supervisor if necessary.

2.4 Requirements Specification

Again we look at the main categories as used in the requirements analysis:

- System Implementations.
- Functional Requirements.
- Non-functional Requirements.

2.4.1 System Implementations.

Language

Our program will use the latest version of Java which is J2SE v1.4.2 and the version of the Java3D API is 1.3.1 both of which are available for download from www.java.sun.com along the latest versions of the Java Runtime Environment for java and java3D.

Operating and Developing Environment

The programmer's preferred development environment is using a text editor and a UNIX or Dos prompt for direct compiling. The operating system used will be Windows 98 and it will be assumed that no dedicated graphics hardware is available so Java3D will be implemented as software.

2.4.2 Functional Requirements.

The choices the user will be able to make in the interface will be as follows:

- A way of choosing a colour. After a colour is chosen all the following bricks will have that colour until another colour is selected.
- A way of choosing the height of a brick. Lego™ bricks come in full size or a "half size" this choice will act similarly to the colour choice.
- A way of choosing the size of a brick, the smallest brick is "1by1" but other common brick sizes are "1by2", "2by2" and "2by4". There should be a whole range of appropriate brick sizes the user can choose.
- There should be a way to rotate the model so a user can see all the sides of their creation.
- The orientation of non-square bricks must be accounted for by allowing the user to either choose a left-pointing or right-pointing brick.
- Brick placement should be implemented in an unambiguous way.
- A collision detection method should be installed to prevent bricks from overlapping.
- There should be a method to delete a block from the model.

Appearance of Lego™ Blocks.

The main appearance of the blocks is a decision which is left up to the user. As detailed above, there will be a certain number of components to choose from, but the appearance choices will all be pre-defined in the program such as “green” or “2by4”. The lighting of the model will be performed by an arbitrary light over which the end user will have no control. Test locations for the light source will happen at some time during implementation and the results will be reported in chapter 5.

Error Handling.

As the user interface will be mainly mouse based, the scope for error is greatly reduced as no bad or missing parameters can be entered. The system will be designed to be as robust as possible and thus there should be minimal errors. Any errors within the code should be caught and corrected during testing to produce a bug free program upon completion of the project.

2.4.3 Non Functional Requirements.

System Testing.

A formal test plan will be created and presented later in the dissertation. It will focus mainly on black-box testing as opposed to white box testing, which will be done informally throughout the build process to ensure the program components have some degree of functionality.

Security.

No provision is required, so none will be made. To secure the project programme, a copy of the program will be kept on a flash drive in case of electronic loss of program.

Other Constraints.

Resources, hardware and software are all available to the programmer and obtaining and the continued use of these will not constitute a problem. The system is to implemented on a stand-alone PC capable of running the Java Runtime Environments and processing the 3D colour graphics.

Chapter 3: System Design

3.1 Human Computer Interaction

Dix et al (1993) state that the aims of the system interface is to achieve learnability, flexibility and robustness for any program and that these categories can be broken down further. The most relevant categories for the Lego™ builder fall mostly under learnability and are as follows:

3.1.1 Predictability.

A user does not take well to the unexpected, instead preferring to be able to predict how the interface will behave in the future based upon past interaction history. One of the main aspects of this project is the placing of blocks; if a user is do this quickly they must be able to hover the mouse in the same place each time (transposed by a brick length). A high level of predictability will mean that after a short while the users will instinctively know where to click the mouse to place a brick.

3.1.2 Familiarity

New users of a program nearly often have knowledge of other programs. Familiarity has do with a user's first impression of the program and how it is perceived. A good design will allow the user to initiate immediate interaction; this program can achieve that by careful use of applet widgets.

3.1.3 Consistency

Consistency relates to the likeness in behaviour arising from similar situations or similar task objectives. There are many forms of consistency, but the interface will attempt to be consistent by demanding the same user action to achieve the similar but different goals.

3.1.4 Responsiveness

Responsiveness measures the rate of communication between the user and the system. Faster system times are more desirable (but necessarily to the point where they are too fast for the user to acknowledge they happened). Individual response times will mainly be dependant on the hardware configuration of the user, but whilst designing the program, response times should either be kept to a minimum or provide the user with a visual clue that there is a delay. Hopefully the response times will be acceptably small and this is the expected outcome, but this will be inspected during testing.

3.2 Design Overview.

The original problem of choosing between the orthographic projection, which puts function over form, as opposed to the perspective projection, which puts form over function, still has not been resolved. The solution proposed in this design is to utilise the best of both worlds. A 2D canvas will be set up on one side for building the Lego™ model in a orthographic projection, and a 3D model on the other side of the applet will be a 3D canvas which fully renders the model in full 3D perspective projection. The 3D canvas will mainly be for viewing and will not allow the user to do much apart from view the rendered and lit model from different angles. The 2D canvas will be for building the model and will consist of a build area and a brick/colour selection area. The model building area and the brick/colour selection area may also be separate canvasses. An outline of the applet is conceptualised in figure 3.1.

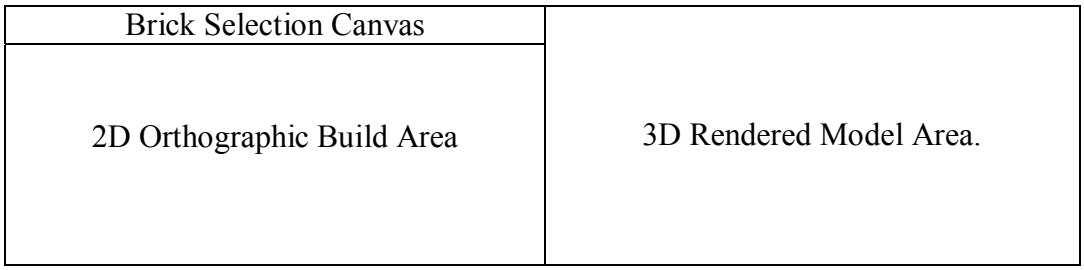

Fig 3.1

3.2.1 Build Area Methods.

Vertical brick placement

The use of a 2D orthographic projection means there are two main methods that can be implemented to provide 3D brick placements. They both use an orthographically projected grid over the build area. This provides users with a visual aid when trying to line up bricks that are in the same horizontal plane. (The methods to make sure these bricks are exactly aligned with each other will be discussed later in the design specification.) The two methods for vertical brick placement then follow thus:

For the first method, the user holds down a selected key on the keyboard, this locks the brick in place in the x-axis and y-axis, the user can then move the mouse up or down to raise or lower the brick vertically. In essence, by pressing and holding the selected key, the user limits the building of bricks to a column. Whilst the key is being held down; any lateral movement of the mouse is ignored by the brick placement methods; only the vertical movement registers. This allows a vertical tower of single bricks to be built very quickly, but is very slow at building lots of horizontally placed bricks at a high level (for example a roof of a building).

For the second method, there is no inherent vertical construction; the user can only build upon one plane at a time. However, the user can move between planes

by pressing select keys to change them (intuitively the up and down arrow keys would be used), buttons can also be used for this purpose. In essence, by pressing the down key, all the current bricks in the build area are shifted down by the height of a single brick. This method is almost as quick at building a vertical tower but sufficiently quicker at building even medium sized models as moving between planes is easier than using a button to build every brick not at the base level. It is for that reason that this method will be the chosen method to avoid brick placement. The ambiguity of where bricks are placed will be eliminated, as there will only be only horizontal plane which bricks can be placed in at any one time.

Brick Alignment

The Lego™ bricks in the program need to line up with each other in order to create smooth surfaces and models. In order to do this, the program has a set number of points on the orthographic grid where the corner of a brick can lie. These points are where lines parallel with the x-axis coincide with lines parallel with the y-axis. In order to line up the bricks, the program takes the current xy-coordinates of the mouse and selects the nearest grid intersection and uses it as the potential location for the next brick, as the mouse is moved across the building canvas, the nearest grid intersection changes with it. These grid intersections will now be referred to as build points, they denote any place where the corner of a brick can lie and thus be placed there.

Brick Orientation

Any bricks that are not square may face one of two ways with respect to the screen, either pointing left or pointing right as shown in Fig 3.2. The program will initialise all bricks to start pointing in one direction, say left, in order to change this, the user can press a key or click a button to change the orientation, the direction will then shift 90 degrees to point the other way and then brick placement can carry on as before with the new orientation. This system works significantly better than the other alternative which is to have two selection options per brick, one for right pointing bricks and one for left pointing bricks as this doubles the number of bricks the user can choose from, but doesn't increase the functionality or usability of the program.

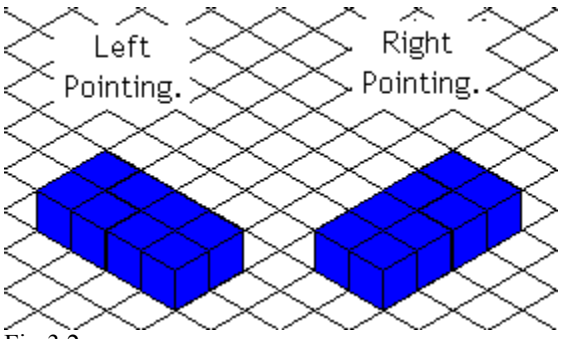

Fig 3.2

Build Area Rotation

In order to rotate the blocks, the build area will have a central area through which the applet will rotate, this will be invisible to the end user, but rotation intuitively takes place around the centre of the model so this will not contravene the predictability requirement. A button on the brick selection canvas will trigger the rotation of the design area. Each rotation will be 90 degrees to allow the user to view each side of their model but also to return easily to the default view.

Keeping Track of Bricks

With many different sizes, shapes and colours of bricks in our build area, there is a need for a mechanism to keep track of which bricks have been placed and where. An easy way of implementing this is to create an array to store the locations and attributes of all the bricks. New bricks can be added to the array as soon as they are created.

Removing Bricks

The remove brick method will be implemented by right clicking on a brick already displayed in the build area, this then remove it and repaint the canvas to show the model without it. The bricks can be removed by deleting them from the array outlined above.

3.2.2Brick Selection Area Methods

There are two user choices available here, brick colour and brick size. For maximum functionality, these two choices will be driven by a point and click interface. There will also be the option to raise or lower the building plane.

Colour

A colour palette will be drawn in this canvas with the multiple colours available. By clicking on a colour, the users sets this as the colour for all new bricks until a new colour is chosen. In order for the colour of every brick to be kept the same, the colour of each brick will be entered into the array that keeps track of bricks. The use of the colour palette as opposed to allowing users to create their own custom RGB colours does limit the choice aspect of the applet, but this is a trade off between choice and complexity. Creating an RGB colour is less intuitive for a user than simply clicking a colour display and most users won't miss the range of RGB colours as long as the main colours are represented.

Brick Size

This will work on a similar system to colour with each available size displayed as a small picture. Clicking on the picture selects that block, which can then be placed at will onto the canvas.

Raising/Lowering Plane

These will take the form of two buttons, clicking the appropriate button will raise or lower the plane to aid the construction of vertical bricks as outlined above.

3.2.3 3D Canvas Methods

The 3D canvas has three main functions; displaying the bricks, allowing the user to manipulate the model view and lighting the model. The methods for each of the following are discussed in turn.

Displaying the Bricks

The array that keeps tracks of all of the bricks can be passed to the 3D canvas, which can then proceed to convert the positioning and attributes into a 3D representation of the model. This can be done by treating the centre of the 2D applet where the rotation occurs as the y-axis and the default building as the plane that coincides with the x-axis and the z-axis. Then it is just a case of determining the appropriate transforms from the array passed from the 2D canvas. The 3D geometry used to create the bricks will be inherently different than that used to create the 2D bricks so that will not form a part of the array. For this reason, the bricks need not be the same size, but can be scaled versions so that smaller models can be enhanced to look larger for example.

Manipulating the Model View

Whilst the model itself cannot be changed in the 3D view, the user is able to view their creation by dragging the mouse around in the 3D canvas area; this will rotate the model through all three axes corresponding to the direction of the mouse drag.

Lighting the model.

To light the model, two light sources will be implemented to shine from opposite angles onto the model. These lights will not be adjustable by the user and exist to provide added realism to the model.

Chapter 4 System Implementation.

This chapter deals with the implementation of the design to see how it translated from theory to practise. The code for the program can be found in appendix B. Due to various problems and constraints the program is far from finished, and although it works; it is not a finished program. The various reasons for this will be looked at in Chapter 6; this part will concentrate on the ideas that were implemented and how ideas would have implemented with more time.

4.1 Vertical Brick Placement & Raising/Lowering Plane

By choosing the method of only allowing building in one plane, this effectively eliminated ambiguous placement of bricks. From this point of view it was a success but it did lead to some uncertainty as to which blocks were on which level at the early stages of creating a design. For example, after the first brick is placed and then moved down, the user has no way of telling whether a brick has been placed and moved down a level or built on the current plane but one space further forward.

A possible solution to this problem is to have the program draw any blocks that are lower than the current building, then to draw the orthographic grid overlay, then to draw any blocks that are on or above the current building plane. As the plane is moved, the bricks and overlay are redrawn accordingly. This could be implemented as the third value in the Lego™ brick display array (`lbdarray`) is the numerical value of the plane the brick was drawn on (`levelBuiltAt`) and comparing it with the current level of the plane to determine if it is drawn before or after the grid overlay.

The actual method to raise or lower the plane is to press the up and down arrow keys; this is to improve the speed of building as otherwise building a tower could take an unnecessarily long time.

4.2 Brick Alignment.

The heights of the Lego™ bricks were chosen to be drawn 20 pixels high as this offered the best compromise between the size of the bricks and the size of the models that could be built on screen. Too few bricks mean that no complicated models could be constructed whilst bricks that were too small become cumbersome for young children to build. This is mainly due to the fact that smaller bricks lead to smaller areas on the screen in which a mouse click means a brick in built in that particular location.

An even number was also chosen so that the vertical distance between intersections on the orthographic grid were a whole number of pixels and not a decimal number ending in .5. The grid itself was then calculated and drawn; with the orthographic grid being set at angle of 30°. The vertical lines were removed from the grid in order for a more intuitive feel, as the lines were not needed to

show depth. This meant that the angles made at the intersection were 60° and 120°. Figure 4.1 shows a normal orthographic gird on the left and the grid used in the program on the right.

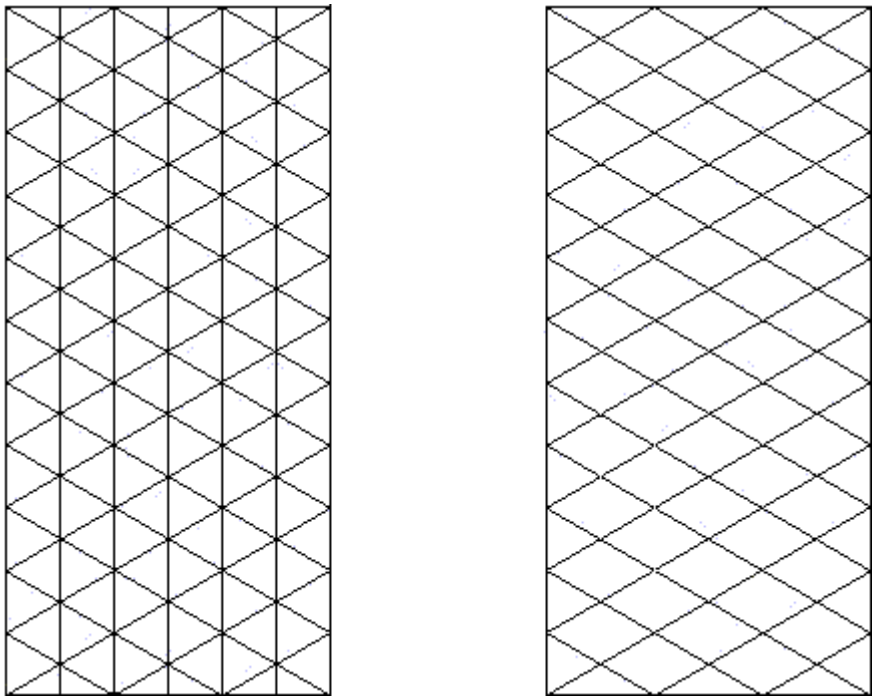

Fig 4.2

The length of 20 pixels for the brick size meant that the intersections were 20 pixels apart; the vertical distance between pixels can be calculated by the formula:

$$\frac{2x20}{\sin(30°)} = 20$$

The horizontal distance between pixels can be calculated by the formula:

$$\frac{2x20}{\cos(30°)} = 20\sqrt{3}$$

Both these distances are summarised in Fig 4.2.

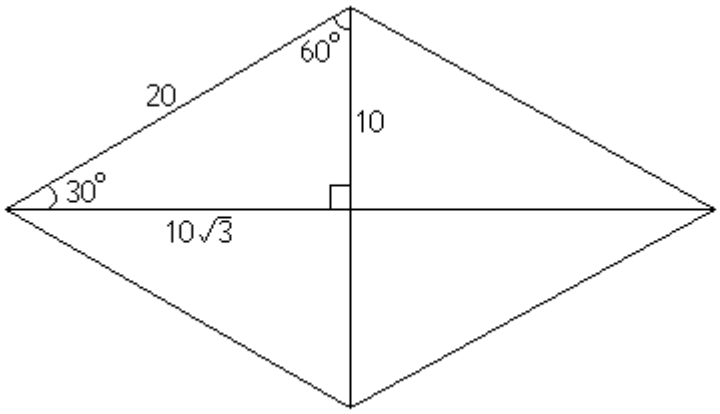

Fig 4.2

By ensuring that the bottom corner of every Lego™ brick is drawn at one of the grid intersections, the problem of block alignment is instantly solved. The program did this automatically choosing the nearest intersection and drawing a "shadow" brick in another colour at that point that became a permanent brick in the event of a mouse click. This was achieved by writing the `getblock` method, which takes the current co-ordinates of the mouse and works out the nearest points. It does this by working out the next highest and next lowest possible x values and y values of the nearest building blocks. This is because of the knowledge that the intersections occur every 10 pixels vertically and every $10\sqrt{3}$ horizontally. Thus it can return two possible nearby points and use Pythagoras' theorem to work out the nearest grid intersection.

This method works very well, however, there a small problem with it, as the horizontal grid intersections are spaced every $10\sqrt{3}$, the values must be rounded to the nearest integer, this means that as bricks are drawn with a horizontal distance of 17 pixels instead of 17.32051 (to 5d.p). Hence over the course of approximately three pixels, the blocks are drawn with a double line between the blocks. This can be seen in figure 4.3.

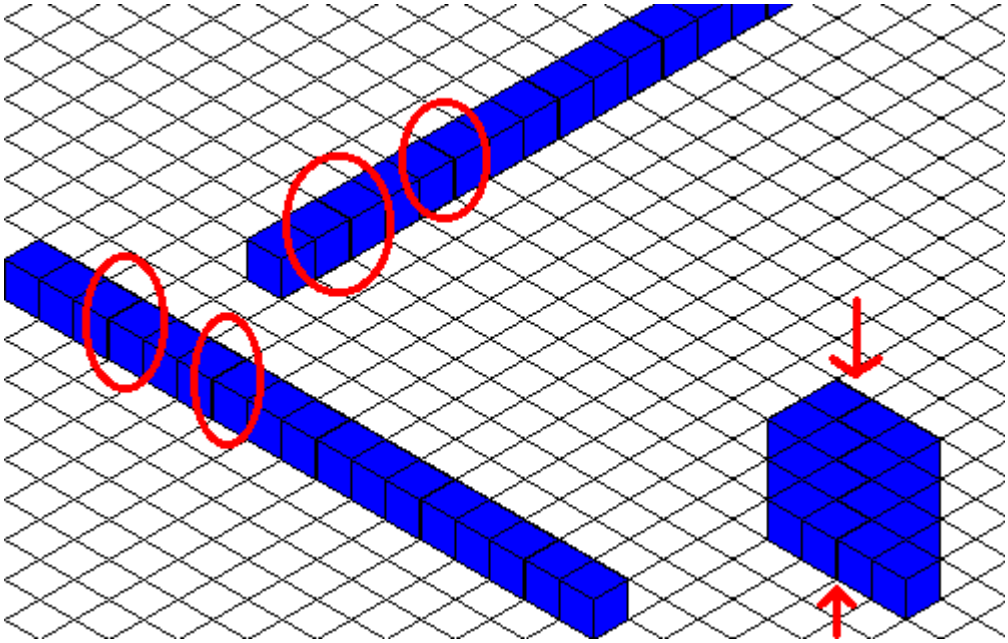

Fig 4.3.

4.3 Keeping Track of Bricks.

Creating an array of arrays solved the problem of keeping track of bricks; in the current version of the program, it has only three elements, the x co-ordinate, the y co-ordinate and the plane that it is on. The final version would have included further information such as colour, shape and size. The matrix still offers a good way of representing the details of all the bricks as the bricks can be sorted into a certain order. This was discovered to be vital as in an early version of the program, the bricks were drawn in the order they were placed. This led to incorrectly formed models, as hidden faces of blocks were visible, often covering the faces of other blocks. This was a problem accentuated with the placement of vertical blocks. Sorting the arrays solved this issue, known as clipping; the methods `AoASorter` and `AoAMixedSorter` are two methods that can sort arrays of arrays by a particular array, keeping the data in all columns together. `AoAMixedSorter` sorts the `lbdarray` firstly by the `levelBuildAt` array in ascending area, and then resolves any ties by sorting the y co-ordinate column in ascending order. The final result is the bricks on the lowest levels are drawn first, starting with those at the back and working forwards to ensure that no brick is drawn over a brick nearer to the user's point of view.

This solution works perfectly well with equally sized blocks, but longer blocks may have a shorter block placed in front of it, but with y co-ordinates smaller than that of the longer brick. The first solution tried was to add a number to the `lbdarray` determining the length and then when sorting the bricks to determining drawing order, the longer blocks would be drawn before shorter blocks with lower y-co-ordinates. However, consider Figure 4.4, the red and yellow 2by1 bricks have the same y co-ordinates, yet the larger blue 7by1 blue block only blocks one of them.

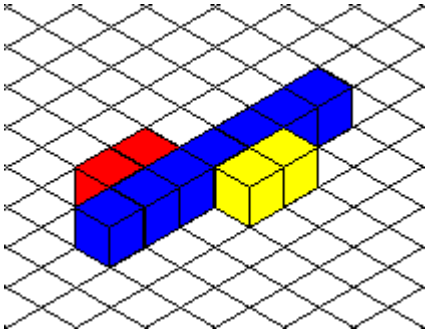
Fig 4.4

Thus this proposed method did not work. A proposed solution would be to split larger bricks up into 1by1 bricks and then draw them as before, sorted first by the plane then by the y co-ordinate.

4.4 Brick Orientation & Brick Size

Although there were problems with selecting brick size in the separate canvas, assuming it was possible, then it would have been a case of drawing larger bricks at the grid. As detailed above, storing larger blocks as one in the array led to complications when it came to drawing, thus a better solution would be to have large blocks split up into many smaller 1by1 blocks, each would be given their own place in the `lbdarray`. The clicking of the mouse on the different sized block button would set a variable that meant that every time a larger brick was placed, it would fill all the array elements with the appropriate graphic (i.e. edge of brick or corner of brick). To do this there would have to be an integer reference to every possible graphic, and an extended for loop in the `paint` class. Filling the array would not be difficult, as when the original build point is known by clicking the mouse, the others can be worked out accordingly as every build point is known. To complete the brick placement, the counter, `currentblock`, that totals the number of bricks would have to be increment as many times for as many new 1by1 blocks are drawn, otherwise some blocks will be missed out when the array is sorted for drawing.

Brick orientation is made easier if the many 1by1 blocks approach is used to solve the problem of brick size. Rather than calculating the co-ordinates of the other 1by1 blocks by assuming the brick is laying to the right, as in Fig 4.4, the program can determine which where to put all the 1by1 bricks by a method used to record the orientation of the bricks, which could be triggered by a keyboard press, possibly the spacebar or perhaps the right and left arrows as the up and down arrows are already being used.

4.5 Brick Removal.

The current program has no way to remove bricks from the model, but this is a very desired feature for the finished product. The proposed method would be to right click on the canvas to remove a brick. This right click would determine which brick was clicked and then remove it. The brick detection method would

consist of working out which brick was being drawn where the mouse click. Every 1by1 brick covers an area in the shape of a regular hexagon with an edge length of twenty pixels. This selection method needs more refinement however as the bricks overlap, hence, an if loop could be implemented to work out if a brick is being overlapped by another one or not. To keep this if loop relatively straightforward, it would only be possible to remove bricks on the current build level. Once the brick that is to be deleted has been selected, this can simply be deleted from the `lbdarray`, `currentblock` decreased accordingly and the remaining elements shuffled down the array to fill the gap left by the removed block. Larger blocks could be removed in the same way, but with the necessary 1by1 blocks being removed from the `lbdarray` simultaneously.

4.6 Build Area Rotation.

By choosing a centre point for the build area (approx (230,242)), the x and y co-ordinates for every block could be swapped for a 90° rotation. The blocks would remained unchanged in colour and shape and relative positioning, but it's possible that a 1by1 block that was the right edge of a large brick is now meant to the left block of the same brick, or that a corner is still a corner but facing in another way. This would necessitate changes to the number corresponding to the graphical depiction of every block. The other potential problem is that there is not a predetermined formula for retrieving the x and y co-ordinates, so this may have to be done by hand and then hard coded into the program, for which there was not enough time (see Chapter 6).

4.7 Colour.

This was where the program hit it's major obstacle, here in a minor fashion, but it was the same that affected the 2D to 3D transformation of the object. The original design called for a small 2D canvas to be positioned above the main 2D building area. From this it would be possible to select the colour of the bricks. However the problem lied in communicating between canvasses. In the program there were two 2D canvasses `can`, the build area and `can2` the colour palette. They were instances of `MyCanvas` and `MyCanvas2`, both declared in the applet's body. However, when writing methods for this class, these methods were not able to apply methods to instances of classes. Yet in the applet, any methods called are only applied once, after which the applet just waits for user input, and then applies the necessary methods upon it.

Writing a method for the colour palette canvas (an instance of `MyCanvas2`) to change the display in the build canvas did not succeed as explained, but no other approach to the problem was successful. It was not possible to declare a global variable in the applet class, as java does not include java variables. Instead, Lemay (1999) recommends using instance and class variables to communicate from one object to another, thus replacing the need for global variables. However, this combination doesn't work when in used in conjunction with applets as described above.

This same problem also manifested itself with the 3D Canvas; in order for it to work, it needed access to the data in `lbdarray`, which is where it encountered a similar situation as above. This time instead of being a separate canvas class, the 3D canvas could be defined in the applets `init` method, but this meant whilst it could get the `lbdarray` from the 2D build canvas, but only once during initialisation of the applet, but the array was always empty at this time. Once the program had started, the 3D canvas was unable to seek the updated values of `lbdarray`.

Chapter 5: Testing.

5.1 System Testing

As the program was unfinished, a testing plan cannot be implemented fully, but there were several aspects of the program that could be tested. Fig 5.1 shows a typical screen from the program. The coloured shape on the right with the black background is the 3D canvas, with a rotating `ColorCube` being drawn to showcase the 3D abilities of java, this spins, although it is not possible to show this in this documentation. The left portion of the screen shows the 2D build area, (the main part to be tested) and the 2D colour palette. This remains unfinished, and clicking on the colours changes the colour once, but after this it fails to change the colour. Furthermore it doesn't change the colour of the blocks but of the writing alongside it.

The 2D canvas itself displays the message "HELLO!" with letters seven blocks high, it also shows the shadow block, which is orange, which shows the potential position of the next block. The numbers on the screen were printed out to help construction of the applet and would not appear in the final version.

chosenColor=0

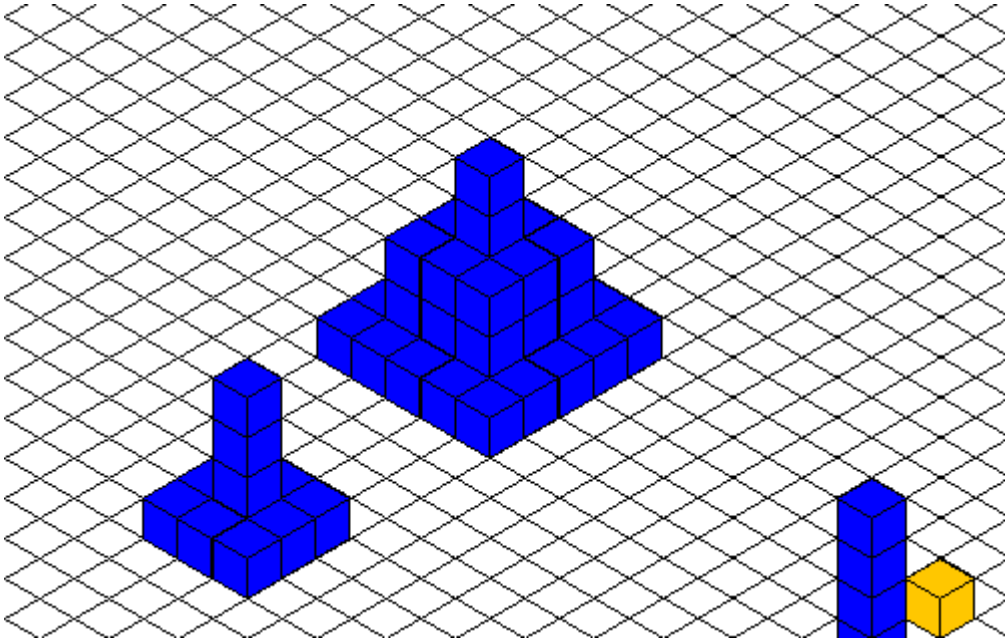

Fig5.2

Fig 5.2 shows the successful building of miniature towers of blocks. This test also tested to see if bricks were retrieved if the movement of the building plane meant they were pushed off the bottom of the screen. Fig 5.3 shows this is successful.

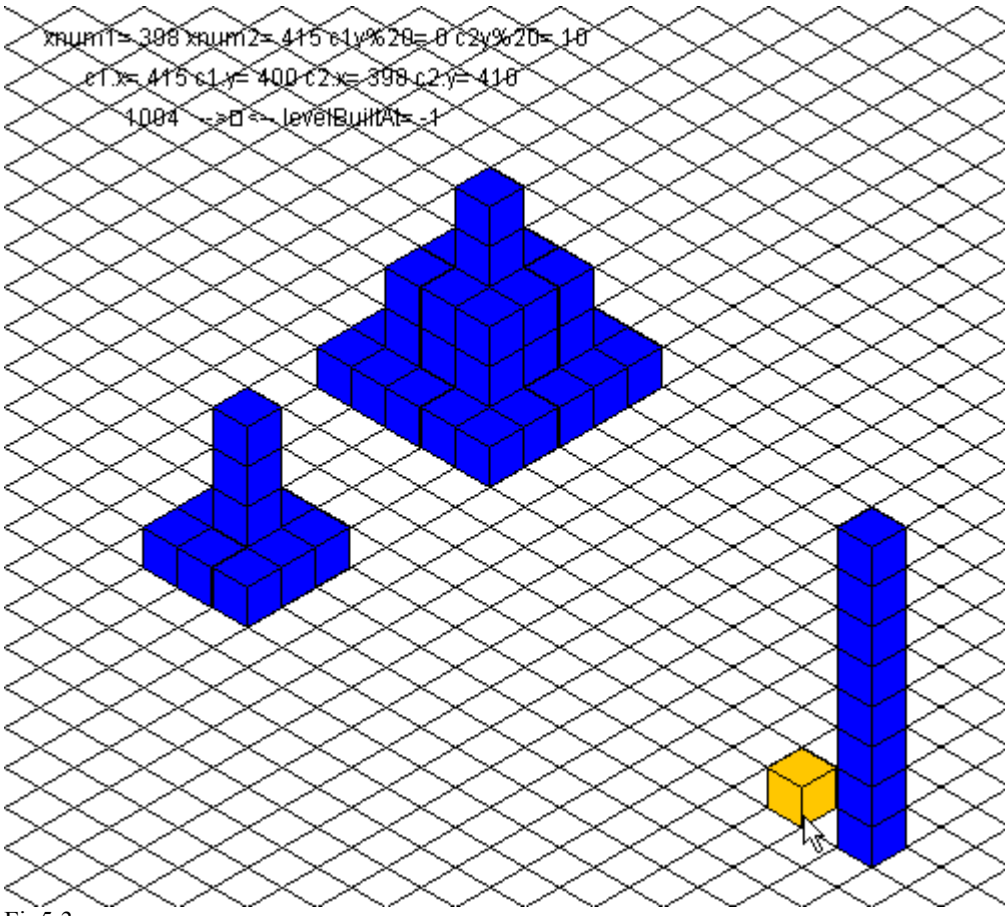

Fig5.3

One problem that was picked up by testing was that of animation flickering. This is one of the side effects of the way images are updated on a Java program, and it is caused by the way Java repaints every frame in an applet. This is because the `repaint` method calls another method called `update`. The easiest way to avoid this is to override both these methods by using double buffering. Double buffering works by creating a second surface off screen to which all the drawing is done, before being copied across to the correct location after it has finished. As all the work is not done in front of the user, intermediate parts of the drawing process do not appear and thus cannot disrupt the smoothness of the algorithm.

Chapter 6: Conclusion

6.1 Dissertation Review

In the early stages of the project, background reading took priority and the searching for previous work and material on the subject. The learning of Java 3D also took up a large amount of the time available for this project. The subsequent lack of being able to demonstrate the programmer's skill in Java 3D is therefore especially regrettable.

The requirements analysis and specification were detailed and covered a wide area of considerations, yet it appears that Java may not have been the best choice of language for implementation. Also, more research could have been undertaken to garner a wider knowledge of the requirements needed from HCI.

The system design and implementation was thorough enough to give the programmer an idea of what the final idea should look like and the functionality it should be provided. Although the design was a flawed design, the documentation behind it was solid.

The system testing was notably small, this has mainly hampered by the fact the program was unfinished, but there was no system plan in place anyway. A further developed plan designed to develop any tangible outputs of the project would have been a better tool to assess the quality of the finished program.

6.2 Dissertation Critique

The Gantt chart in Appendix B shows the original time allocation for each task, not only was this inadequately thought through from the start, but as the project developed over the first few months, the chart was not altered to accommodate the changes. This was coupled with the fact that Java 3D took longer than anticipated to learn and achieve useful results, and the learning of this started later due to concentrating on the documentation early on. This led to rushed programming, which led down many dead ends, some avoidable, whilst others were just another step up the learning curve. Thus towards the end of the project's timeline the decision was made to leave the program and concentrate on the dissertation.

6.3 Implementation Critique

Unfortunately, a bad choice of language was chosen which would render the envisaged project too complex to complete within the allocated timeframe. The program as is does manage to run well with the functionality it does possess. However, this is not enough to be a usable project. Whilst the colour palette canvas and the 2D build area canvas could be merged into one canvas without too much difficulty, it would still create a overly complicated program and could have other problems such as Lego™ bricks being drawn over the colour palette.

Methods to stop this could be implemented, but would have to be hard-coded but this is extra work both for the programmer and for the program at run-time.

More of a problem is getting the 2D build area and the 3D display area to communicate, whilst the 3D canvas is unable to read the updated `lbdarray`, the Lego™ model cannot be displayed in 3D form, which was an important aspect of the project, as there are already many 2D Lego™ modellers available.

Future work on this project, beyond completing the program, would be an exploration into the possibility of an implementation of the program which has a toggling interface switching between 2D and 3D. This would enable the user to take advantage of both the views, but the use of one canvas would mean that the major problem encountered in this project would be eliminated. Another possibility is an infinitely adjustable view, whereby the user has complete control of the view, and can move it into a position to make laying a block in any position an easy task. Whilst this project is completely in 3D and thus has an advantage over the other two, this could mean that the view needed to be adjusted every time a brick was placed, this would then render that program as very unusable.

6.4 Summary

The program developed was a new take on an existing problem, unfortunately the design was flawed and therefore did not translate well from theory into practise, if repeating the project, more time would have been spent on choosing the language and checking that the basic ideas were plausible. Nevertheless, the existing program could be built upon to provide a perfectly passable orthographic Lego™ modeller and the documentation accompanying the project has been detailed and reasonably well written.

Bibliography.

Flanagan, D. 2002. Java in a Nutshell. Fourth Edition. O'Reilly.

Hong, J. K. 2002. cyberEarth. Parsons School of Design.

Lawlor, S.C. 1995. ANSI C Programming. West.

Lemay, L., Cadenhead, R. 1999. Teach Yourself Java 2 in 21 Days. Sams Publishing.

Moreira, J. E., Midkiff, S. P., Gupta, M. 1998. A comparison of Java, C/C++, and FORTRAN for numerical computing. IEEE Antennas and Propagation Magazine. 40 (5), pg102-105.

Preece, J., Rogers, Y., Sharp, H. 2002. Interaction Design: Beyond Human-Computer Interaction. John Wiley & Sons, Inc.

Sommerville, I. 1992. Software Engineering. Fourth Edition. Addison-Wesley.

Watt, A.1993. 3D Computer Graphics. Second Edition. Addison Wesley.

Wright, R., Sweet, M. 2000. OpenGL SuperBible. Second Edition. Waite Group Press.

Online Resources:

www.opengl.org
www.sgi.com
www.tcl.tk
www.gamedev.net
www.mesa3d.org

```
/*************************************
 * Lego1.java created by Richard Stone *
 * On 10/03/05. Last Edited 15/05/05   *
 *************************************/

import java.applet.Applet;
import java.awt.*;
import javax.media.j3d.*;
import javax.vecmath.*;
import javax.swing.JFrame;

import com.sun.j3d.utils.universe.*;
import com.sun.j3d.utils.geometry.*;
import com.sun.j3d.utils.*;


public class Lego1 extends Applet {

  BorderLayout bl = new BorderLayout();
  MyCanvas can = new MyCanvas();   //2D Canvas (custom)
  MyCanvas2 can2 = new MyCanvas2();              //Brick Selection Canvas (custom)

  GraphicsConfiguration config= SimpleUniverse.getPreferredConfiguration();
  Canvas3D RSCanvas3D = new Canvas3D(config);                                                    //3D      Canvas      (not
custom)

  public void buildConstraints(GridBagConstraints gbc, int gx, int gy, int gw, int gh, int wx, int wy) {

    gbc.gridx = gx;
    gbc.gridy = gy;
    gbc.gridwidth = gw;                    //method to set up gridbag layout quickly.
    gbc.gridheight = gh;
    gbc.weightx = wx;
    gbc.weighty = wy;
    }

  public void init() {                     //Initiation of all applets

//*****************************   3DCanvas initialisation

  BranchGroup scene = createSceneGraph();          //new branch group (a content graph)
  scene.compile();                                          //compiles graph before adding it to simple universe

  SimpleUniverse simpleU = new SimpleUniverse(RSCanvas3D);              //new simple Universe contains the 3D canvas

  simpleU.getViewingPlatform().setNominalViewingTransform();            //Sets viewing platform

  simpleU.addBranchGraph(scene);                                        //adds branch group to universe

//***************************** end of 3D canvas initialisation


    can.requestFocus();                          //sets keyboard focus to build area

    setBackground(Color.white);

    GridBagLayout gbl = new GridBagLayout();
    GridBagConstraints constraints = new GridBagConstraints();
    setLayout(gbl);

    buildConstraints(constraints, 0, 0, 1, 1, 50, 10);
    constraints.fill = GridBagConstraints.BOTH;
    gbl.setConstraints(can2, constraints);
    add(can2);

    buildConstraints(constraints, 0, 1, 1, 1, 0, 90);
    constraints.fill = GridBagConstraints.BOTH;
    gbl.setConstraints(can, constraints);
    add(can);

    buildConstraints(constraints, 1, 0, 1, 2, 50, 0);
    constraints.fill = GridBagConstraints.BOTH;
    gbl.setConstraints(RSCanvas3D, constraints);
    add(RSCanvas3D);
    }

  public BranchGroup createSceneGraph() {                  // 3D content branch graph creation

    BranchGroup objRoot = new BranchGroup();

    Transform3D rotate = new Transform3D();                  //Transforms that rotate object
    Transform3D tempRotate = new Transform3D();
```

```
    rotate.rotX(Math.PI/4.0d);
    tempRotate.rotY(Math.PI/5.0d);
    rotate.mul(tempRotate);                                          //Two transforms combine to create double rotation

    TransformGroup objRotate = new TransformGroup(rotate);   //rotation transform added to transformgroup node

    TransformGroup objSpin = new TransformGroup();
    objSpin.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);          //Allows it to change, needed for animation

    objRoot.addChild(objRotate);
    objRotate.addChild(objSpin);

    objSpin.addChild(new ColorCube(0.4));                    //Calls a color-cube, not related to lego, but used to
                                                                                    //demonstrate Java 3D
    Transform3D yAxis = new Transform3D();
    Alpha rotationAlpha = new Alpha(-1, 4000);               //time function, cube spins every 4seconds infinitely.

    RotationInterpolator rotator = new RotationInterpolator(rotationAlpha, objSpin); // yAxis, 0.0f, (float) Math.PI*2.0f);

// this creates the spin function

    BoundingSphere bounds = new BoundingSphere();            //Bounds covers cube so it can spin
    rotator.setSchedulingBounds(bounds);
    objSpin.addChild(rotator);

    return objRoot;
    }

  }


class MyCanvas2 extends Canvas {

  int chosenColor=0;

  public boolean mouseDown(Event evt, int x, int y) {  // method to select colour, not working!

    if (x<250) {chosenColor=4;}
    else if (x<200) {chosenColor=3;}
    else if (x<150) {chosenColor=2;}
    else if (x<100) {chosenColor=1;}
    else if (x<50) {chosenColor=0;}

    repaint();
    return true;
    }

  public int getColorChoice() {

    return chosenColor;
    }

  public void paint(Graphics g) { //Creates colour palette

    g.setColor(Color.red);
    g.fillRect(0,0,50,50);
    g.setColor(Color.blue);
    g.fillRect(50,0,50,50);
    g.setColor(Color.green);
    g.fillRect(100,0,50,50);
    g.setColor(Color.yellow);
    g.fillRect(150,0,50,50);
    g.setColor(Color.orange);
    g.fillRect(200,0,50,50);
    String tempString4 = new String ("chosenColor="+chosenColor);          //For feedback purposes, not an intended
    g.drawString(tempString4, 275,25);                                                  //part of final program
    }
  }

class MyCanvas extends Canvas {      //2D campus

  final int MAXBLOCKS = 100;         //sets the total number of bricks allowed.
  final int ARRAYINFO = 3;           //The amount of details the brick array carrries,
                                                  // 0 = xPoint, 1 = yPoint, 2 = Z-Level.

  int[][] lbd = new int[ARRAYINFO][MAXBLOCKS]; //The array where every lego brick is stored, (lbd = lego brick display)

  Point chosenPoint;                              //These are various variables that are used by more than one
  Point choice1 = new Point(0,0);      //method and are thus declared initially
  Point choice2 = new Point(0,0);
  int xnum1, xnum2, xscale;
  float c1d, c2d;
  int colorChoice = 0;
```

```
 char typedChar = (char)0;

 int currentBlock = 0;                  //This int keeps track of how many blocks there are

 int levelBuiltAt = 0;      //This int keeps track of which plane the user is building on.

 int intKey;                //Temporary not intended to be in the final program


 public boolean keyDown(Event evt, int key) { //Method that detects up and down buttons

  intKey = key; // Used for feedback purposes, not intended for final program

  if (1004==(char)key) {  // 1004 is ASCII for the up arrow

   levelBuiltAt++;                  //raises the z-plane level up one
   }

  else if (1005==(char)key) { //105 is the ASCII for the down arrow

   levelBuiltAt--;                  //decreases the z-plane by one
   }

  else if (32==(char)key) { // 32 is ASCII for the space bar

   ;   //method for orientation, currently a work in progress
   }

  repaint();              //updates the applet as the up/down buttons moves the lego blocks
  return true;            //so they need to be redrawn
  }

 public boolean mouseDown(Event evt, int x, int y) { //Method places brick

  if (currentBlock < MAXBLOCKS) {           //checks max number of blocks are not exceeded

   addblock(x,y);                   //calls addblock method to add block to the tracker array
   repaint();
   return true;
   }

  else {return false;}

  }

 public boolean mouseMove(Event evt, int x, int y) { //Method detects mouse movement

  getblock(x,y);                                                    //and then calls getblock which finds the nearest build point
  repaint();                                                        //to it's arguments int x and int y, which are the mouse co-
ordinates
  return true;
  }

 void addblock(int x, int y) {              //Method responsible for drawing lego blocks, this calls the getblock method
                                                    //and then adds the details of the block to tracker array, then finally calls
repaint
  getblock(x,y);

  lbd[0][currentBlock] = chosenPoint.x;     //adds x-axis location of nearest build point which becomes draw point
  lbd[1][currentBlock] = chosenPoint.y;     //for the block. Next method does the y axis.
  lbd[2][currentBlock] = levelBuiltAt;      //This method records which plane the brick was built in.
  currentBlock++;                                   //Increases the brick count as extra brick was added.
  repaint();
  }

 void getblock(int x, int y) {                              //this method works out where the nearest block building point is

/**********************************************************************************************************
**
 * Note, the build points form a regular pattern. They are the corners of rhombuses with a edge length of 20 and *
 * 60 and 120 angles. This is normal for a orthographic grid. This means that points on the same vertical are    *
 * separated by 20 pixels and on the same horizontal by 20*root(3). This method chooses the nearest two points   *
 * to compare. It does this by selecting the y value of the horizontal row of points directly above it and below *
 * it. Then it finds the x value of the vertical row of points directly to the left and right of it. These four  *
 * lines provide four intersections, two of which are points and two of which are the centre of the tesselating  *
 * rhombuses and not build points. The final part of the method works out which ones are which and once it's     *
 * found the two points, it works out which is nearest and returns it.                                                                                *

**********************************************************************************************************
*/

  choice1.y = y-(y%10);                     //works out the two y coordinate values of the nearest two points by
  choice2.y = y+(10-(y%10));                        //working out the two numbers divisible by 10 that are nearest to the
```

```
                                                                    //actual y value It then assignes them to the two points, it does
this
                                                                    //randomly as the X values will be added later. Both y values
will
                                                                    //be a final point co-ordinate so this will always work.


   xscale = (int)(x/Math.sqrt(3));
   xnum1 = (int) ( (xscale- (xscale%10) ) *Math.sqrt(3) );                    //works out the two x values
   xnum2 = (int) ( (xscale+10-(xscale%10) ) *Math.sqrt(3) );

   if ( 5> (int)(xnum1/Math.sqrt(3))%20 || 15<(int)(xnum1/Math.sqrt(3))%20) {

            //this if loop lines up the mods of the x and y values and matches them. This works because the lines
            //of the orthographic grid with build points on are as follows. X value lines occur every 10*root(3)
            //pixels and y value lines occur every 10 lines. The if loop works because the odd y lines (10,30,50 etc)
            //coincide with the odd x lines (10*root(3), 30*root(3), 50*root(3)) and the even lines coincide in a
            //similar fashion, this allows mod 20 to be taken and the x values and the y values to be matched correctly
            //to each other and to actual build points and not rhombus centres.


    if (choice1.y%20==0) {

     choice1.x = xnum1;          //this attaches the x values to the corresponding y values
     choice2.x = xnum2;          //that were put into the choice1, choice2 points earlier.
     }

    else {

     choice1.x = xnum2;          //Does the same as above but with a different configuration.
     choice2.x = xnum1;
     }
    }

   else {

    if (choice1.y%20==10) {

     choice1.x = xnum1;          //Does the same as above but with a different configuration.
     choice2.x = xnum2;
     }

    else {

     choice1.x = xnum2;          //Does the same as above but with a different configuration.
     choice2.x = xnum1;
     }

    }


   c1d = (float) Math.sqrt((x-choice1.x)^2 + (y-choice1.x)^2);    //distance from first possible point to mouse location
   c2d = (float) Math.sqrt((choice2.x-x)^2 + (choice2.y-y)^2); //distance from second possible point to mouse location

   if (c1d <c2d) {                          //this selects the nearest build point

    chosenPoint = choice1;
    }

   else {

    chosenPoint = choice2;
    }
   }


  public void paint(Graphics g) {                          //paints the applet

   g.setColor(Color.black);

   for(int j=-340; j<600; j=j+20) {                        //draws the grid

    g.drawLine(0,j,600,(int)(j+(600*Math.tan(Math.PI/6))));
    }

   for(int k=0; k<940; k=k+20) {

    g.drawLine(0,k,600,(int)(k-(600*Math.tan(Math.PI/6))));
    }


   int[][] notNull = new int[ARRAYINFO][currentBlock];
```

```
   for (int i=0; i < ARRAYINFO; i++) {                    //this for loop creates a new array of arrays that
                                                                  //takes only the actual entries from the main array and
     for (int j=0; j < currentBlock; j++) {          //and not the null values (as with a 1000 block limit
                                                                  //there are often 950+ null entries to cycle through
       notNull[i][j] = lbd[i][j];                    //also this creates problems when trying to sort in
      }                                                          //decreasing order as is required later
    }

   AoAMixedSorter(notNull, 2, 1);

/*************************************************************************************************************
***********
 * AoAMixedSorter is a specialist array sorter developed specifically for this application. It takes three arguments,    *
 * an array of arrays and two ints. The two ints are arrays whose data is to be sorted, but it is not just that row          *
 * that is sorted, but whole columns that are swapped around, so the attributes and locations of each lego block is kept  *
 * the same, but the order in which they are drawn is changed. The first row is sorted in decreasing order                      *
 * and then then any ties are sorted by the second row, this time though they are sorted in increasing order. The         *
 * reason for this is that to avoid lego bricks being built in the wrong place, lego blocks which start of the lowest       *
 * (and thus have a high z number) are drawn first, then as we build up, the lower bricks are covered. The same principle *
 * also means that smallest y values are drawn first so blocks and the back are drawn first                                    *

*************************************************************************************************************
**********/


   for (int i=0; i < currentBlock; i++) {

     int xP[] = {(notNull[0][i]),(notNull[0][i]-17),(notNull[0][i]-17),(notNull[0][i]),(notNull[0][i]+17),(notNull[0][i]+17)};
     int yP[] = {(notNull[1][i])-(levelBuiltAt-notNull[2][i])*20,
              (notNull[1][i]-10)-(levelBuiltAt-notNull[2][i])*20,
              (notNull[1][i]-30)-(levelBuiltAt-notNull[2][i])*20,
              (notNull[1][i]-40)-(levelBuiltAt-notNull[2][i])*20,
              (notNull[1][i]-30)-(levelBuiltAt-notNull[2][i])*20,
              (notNull[1][i]-10)-(levelBuiltAt-notNull[2][i])*20};  //defines a polygon that is the cube shape

     int points = xP.length;

     Polygon poly = new Polygon(xP, yP, points);

     g.setColor(Color.blue);    //currently arbitrary, eventually colour will be set by the array

     g.fillPolygon(poly);   //draws previously defined polygon

     g.setColor(Color.black);
     g.drawPolygon(poly);                 //draws outline of polygon in black

     g.drawLine(notNull[0][i],          notNull[1][i]-20-(levelBuiltAt-notNull[2][i])*20,   notNull[0][i],          notNull[1][i]-(levelBuiltAt-
notNull[2][i])*20);
     g.drawLine(notNull[0][i],          notNull[1][i]-20-(levelBuiltAt-notNull[2][i])*20,   notNull[0][i]-17,   notNull[1][i]-30-(levelBuiltAt-
notNull[2][i])*20);
     g.drawLine(notNull[0][i],          notNull[1][i]-20-(levelBuiltAt-notNull[2][i])*20,   notNull[0][i]+17,   notNull[1][i]-30-(levelBuiltAt-
notNull[2][i])*20);
     }

//the above lines draw the three lines inside the cube to complete the 3D look.



   int xP2[] = {(chosenPoint.x),(chosenPoint.x-17),(chosenPoint.x-17),(chosenPoint.x),(chosenPoint.x+17),(chosenPoint.x+17)};
   int yP2[] = {(chosenPoint.y),(chosenPoint.y-10),(chosenPoint.y-30),(chosenPoint.y-40),(chosenPoint.y-30),(chosenPoint.y-10)};
   int points2 = xP2.length;

   Polygon poly2 = new Polygon(xP2, yP2, points2);

   g.setColor(Color.orange);
   g.fillPolygon(poly2);
   g.setColor(Color.black);
   g.drawPolygon(poly2);

   g.drawLine(chosenPoint.x,    chosenPoint.y-20, chosenPoint.x,    chosenPoint.y);
   g.drawLine(chosenPoint.x,    chosenPoint.y-20, chosenPoint.x-17, chosenPoint.y-30);
   g.drawLine(chosenPoint.x,    chosenPoint.y-20, chosenPoint.x+17, chosenPoint.y-30);

//the above code does exactly the same cube drawing, but the values are not taken from a stored array but from the current
//values of the mouse location, this provides a "shadow" brick, which is a different colour and shows where the new brick
//would technically go.


   String tempString1 = new String("c1.x= "+choice1.x+" c1.y= "+choice1.y+" c2.x= "+choice2.x+" c2.y= "+choice2.y);
   String tempString2 = new String("xnum1= "+xnum1+" xnum2= "+xnum2 +" c1y%20= "+ (choice1.y)%20 +" c2y%20= " +
(choice2.y%20));
   String tempString3 = new String(intKey+"   -->"+typedChar+"<-- levelBuiltAt= "+levelBuiltAt);
   g.drawString(tempString1,40,40);
```

```
    g.drawString(tempString2, 20,20);
    g.drawString(tempString3, 60,60);
    }

//A set of string literals, used to fine tune the brick placement system not for use in the completed program.



  public void AoASorter(int[][] data, int sortLine) {   //A sorter that sorts arrays of arrays by the row argument given
                                                                                                                    //needed for the more complex AoAMixedSorter
method.
    int numUnsorted = data[sortLine].length;
    int max, temp;

    while (numUnsorted > 0) {

      max=0;

      for (int i=1; i<numUnsorted; i++) {

        if (data[sortLine][max]<data[sortLine][i]) {

          max=i;
          }
        }

      for (int x=0; x<data.length; x++) {

        temp = data[x][max];
        data[x][max] = data[x][numUnsorted-1];
        data[x][numUnsorted-1] = temp;
        }

      numUnsorted--;
      }
    }

  void AoAMixedSorter(int[][] data, int firstLineSort, int secondLineSort) {

                //the previously described AoAMixedSorter.

    AoASorter(data, firstLineSort);

    int[][] bigTemp = new int[data.length][data[secondLineSort].length];

    for(int a=0; a<data.length; a++) {

      for(int b=0; b<data[secondLineSort].length; b++) {

        bigTemp[a][b] = data[a][b];
        }
      }

    for(int c=0; c<data.length; c++) {

      for(int d=0; d<data[secondLineSort].length; d++) {

                data[c][d] = bigTemp[c][data[secondLineSort].length-1-d];
                }
      }



    for (int i=0; i< data[firstLineSort].length-1; i++) {

      if (data[firstLineSort][i] > data[firstLineSort][i+1]) {i++;}

      else {

        int j = i;

                for (int k = j+1; k<data[firstLineSort].length ; k++) {

          if (data[firstLineSort][i] == data[firstLineSort][k]) {j++;}
          }

                int[][] temp = new int[data.length][j-i+1];

                for (int x=0; x<data.length; x++) {

                  for (int y=0; y<j-i+1; y++) {
```

```
                    temp[x][y] = data[x][i+y];
                }
            }

            AoASorter(temp, secondLineSort);

            for (int m=0; m<data.length; m++) {

                    for (int n=0; n<j-i+1; n++) {

                        data[m][i+n] = temp[m][n];
                }
            }

                    i=j;
        }
    }
  }
}
```