VERY RAPID SOFTWARE-DESIGN PROTOTYPING WITH

INTERACTIVE 3D CRC CARDS

PANAGIOTIS PAPADAKOS

Computer Science Department

University of Crete

APPROVED:

Constantine Stephanidis, Professor, Supervisor

Anthony Savidis, Associate Professor, Supervisor

Anthony Argyros, Associate Professor, Member

Evangelos Markatos, Professor, Member

Panos Trahanias, Professor, Chairman of the Graduate Studies Committee

---

Panagiotis Papadakos, Author

Στους γονείς μου

Σταύρο και Μαρία

με αγάπη

VERY RAPID SOFTWARE-DESIGN PROTOTYPING WITH INTERACTIVE 3D CRC CARDS

by

PANAGIOTIS PAPADAKOS

THESIS

Presented to Graduate Studies Committee of

the Computer Science Department of

the University of Crete

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE

Computer Science Department

UNIVERSITY OF CRETE

Heraklion, November 2007

Acknowledgements

I would like to thank my supervisors Anthony Savidis and Constantine Stephanidis for their continuous support during the completion of this Thesis. For their support I would like to thank my friends and colleagues. I would also like to express my appreciation to Anna-Maria for her encouragement, understanding and patience, in particular during the most difficult, demanding, and sometimes hard, periods of my work. Finally, I would like to thank my family for always believing in me and for supporting me during all these years.

Abstract

CRC cards (Classes, Responsibilities, and Collaborators) are amongst the most well known techniques to practice quick object-oriented design prototyping. The design process through CRC cards is essentially an exploratory procedure to incrementally, quickly and easily identify the objects and relationships inherent in a target software system. The use of cards offers a familiar metaphor, facilitating participatory design, where developers collaborate in an argumentation-based process to quickly produce a plausible system design.

Existing tools supporting CRC cards emphasize a 2D delivery, with a shift of focus on specification and documentation, rather than on direct manipulation and naturalness of interaction. Overall, such tools hardly outperform the benefits from the physical practicing of CRC cards.

As part of this thesis we have developed the Flying Circus design tool offering interactive, direct manipulation of 3D CRC cards. In this tool, cards may be created, positioned, and manipulated in the 3D world, supporting directional labelled links to illustrate collaborators. Facilities like zooming, panning, 3D rotations, smooth navigation, and stereoscopic display are fully provided, essentially delivering an evolving, exploratory and immersive design space. The benefits range from increased control, focus of attention, visual encoding and better exploitation of the human spatial memory.

Supervisors: Constantine Stephanidis, Professor

Anthony Savidis, Associate Professor

Περίληψη

Οι κάρτες CRC (Classes, Responsibilities, & Collaborators) κατατάσσονται ανάμεσα στις πιο γνωστές τεχνικές για την γρήγορη κατασκευή πρωτοτύπων οντοκεντρικής σχεδίασης. Η διαδικασία σχεδίασης με την χρήση CRC καρτών είναι στην ουσία μια διερευνητική διεργασία για την τμηματική, γρήγορη και εύκολη αναγνώριση των εγγενών αντικειμένων και σχέσεων σε συστήματα λογισμικού. Η χρήση των καρτών προσφέρει μια οικεία μεταφορά, διευκολύνοντας την συνεργατική σχεδίαση, στην οποία η ομάδα ανάπτυξης από κοινού αναπτύσσει την επιχειρηματολογία της για την γρήγορη παραγωγή μιας ρεαλιστικής σχεδίασης του συστήματος.

Τα υπάρχοντα εργαλεία που υποστηρίζουν τις CRC κάρτες δίνουν έμφαση σε μια δισδιάστατη (2D) υλοποίηση, επικεντρώνοντας στις προδιαγραφές και την παραγωγή τεκμηρίωσης, αντί για μία φυσική και άμεση αλληλεπίδραση. Σε γενικές γραμμές, τέτοια εργαλεία ξεπερνούν οριακά τα πλεονεκτήματα της απλής χρήσης των CRC καρτών.

Σκοπός αυτής της εργασίας είναι η ανάπτυξη του εργαλείου σχεδίασης Flying Circus, που υποστηρίζει άμεσο και αλληλεπιδραστικό χειρισμό τρισδιάστατων (3D) CRC καρτών. Σε αυτό το εργαλείο, ο χρήστης μπορεί να δημιουργήσει, τοποθετήσει και χειριστεί τις κάρτες σε έναν τρισδιάστατο (3D) κόσμο, με υποστήριξη περιγραφικών και κατευθυντήριων συνδέσμων για την προβολή των συνεργαζόμενων καρτών (collaborators). Λειτουργίες όπως το zooming, panning, τρισδιάστατες (3D) περιστροφές, η ανεμπόδιστη πλοήγηση και η στερεοσκοπική παρουσίαση υποστηρίζονται πλήρως, δημιουργώντας στην ουσία ένα εξελισσόμενο και διερευνητικό περιβάλλον, στο οποίο ο χρήστης μπορεί να εμβυθιστεί. Τα οφέλη περιλαμβάνουν αυξημένο έλεγχο, προσήλωση της προσοχής, οπτική κωδικοποίηση και καλύτερη αξιοποίηση της ανθρώπινης χωρικής μνήμης.

Επόπτες: Κωνσταντίνος Στεφανίδης, Καθηγητής

Αντώνης Σαββίδης, Αναπληρωτής Καθηγητής

Contents

Page

Acknowledgements vii

Abstract ix

Περίληψη xi

Table of Contents xiii

List of Figures xix

1 Introduction 1

1.1 Thesis Statement 2

1.2 Thesis Outline 2

2 Background 5

2.1 How CRC cards were born 5

2.1.1 The CRC card 6

2.1.2 Example of a CRC design 8

2.1.3 Problem solving and insight 8

2.1.4 CRC cards as a metacognitive process 10

2.1.5 The challenge of finding classes 11

2.2 How CRC cards should be used 12

2.2.1 Project management guidelines 13

Team building 13

Inclusion of application experts 14

Coordination with a formal methodology 14

2.2.2 Filling the cards 15

Classes . . . 15
Responsibilities . . . 16
Collaborations . . . 17
2.2.3 Brainstorming sessions . . . 18
Description of brainstorming principles . . . 18
Using brainstorming to find classes . . . 19
2.2.4 Role-playing scenarios . . . 20
How CRC card role play works . . . 20
Developing the role play scenarios . . . 21
Performing the system simulation . . . 21
2.3 3D User Interfaces . . . 23
2.3.1 Limitations of 2D interfaces . . . 24
2.3.2 Movement from 2D to 3D . . . 24
2.3.3 Advantages . . . 25
Take advantage of the human spatial memory . . . 25
They are attractive . . . 26
2.3.4 Drawbacks . . . 27
Navigation, orientation and way-finding problems . . . 27
No natural 3D interaction . . . 27
No guidelines for the design of 3D worlds and UIs . . . 27
No standard library of 3D UI components . . . 28
2D tasks not always transferable to 3D . . . 28
2.4 Immersive environments . . . 29
2.4.1 Immersion through vision . . . 29
2.4.2 Visual depth cues . . . 30
Monoscopic image depth cues . . . 30
Stereoscopic image depth cues, Stereopsis . . . 30
Motion depth cues . . . 30
Physiological depth cues . . . 31
2.4.3 Properties of visual displays . . . 31

Visual presentation properties . . . 31
Logistic properties . . . 32
2.4.4 Stereopsis in visual displays . . . 33
2.4.5 Visual display types . . . 34
Monitor-based - or Fish-tank . . . 34
Projection based . . . 35
Head based . . . 36
See-through head based displays . . . 37
3 Related work . . . 39
3.1 Existing CRC applications . . . 40
3.1.1 Visual Paradigm for UML . . . 40
3.1.2 QuickCRC . . . 40
3.1.3 EasyCRC . . . 43
3.1.4 ECoDE . . . 45
3.1.5 The CRC Design Assistant . . . 45
3.1.6 Merobase component finder . . . 46
3.2 What is missing . . . 47
3.3 Need for a new tool . . . 48
4 Requirements analysis and design . . . 49
4.1 Requirements analysis . . . 49
4.1.1 Functional specifications . . . 50
Navigation . . . 50
Workspace . . . 51
Project . . . 52
CRC cards . . . 53
Links . . . 54
Workspace file . . . 54
Documentation generation . . . 55
4.2 Architecture . . . 55
4.2.1 Model-View-Controller (MVC) . . . 55

4.2.2 MVC++ . . . 57
4.3 Detailed description of components . . . 58
4.3.1 Model . . . 59
4.3.2 Controller . . . 60
4.3.3 Viewer . . . 60
4.4 User interface design . . . 63
4.4.1 Layout . . . 64
4.4.2 Controls . . . 65
Labels . . . 65
Buttons . . . 65
Checkboxes . . . 66
Textboxes . . . 66
Tabs . . . 66
Manipulators . . . 67
4.4.3 Windows . . . 67
Main menu . . . 67
Card . . . 68
Link . . . 69
Other windows . . . 69
5 Implementation . . . 73
5.1 Tools used . . . 73
5.1.1 Use of a Scene Graph . . . 74
Introduction to Scene Graphs . . . 74
Scene Graphs features . . . 75
How Scene Graphs render . . . 76
OpenSceneGraph . . . 77
5.1.2 libsigc++ . . . 79
5.1.3 Flex, bison and ylmm . . . 80
5.1.4 SCons . . . 80
5.1.5 Bazaar . . . 81

5.1.6 Valgrind . . . 82
5.1.7 GCC . . . 82
5.1.8 Visual Studio 2005 . . . 83
5.1.9 Blender . . . 83
5.1.10 Gimp . . . 84
5.2 Components implementation . . . 84
5.2.1 Viewer implementation . . . 84
Viewer dispatcher . . . 85
3D Widgets . . . 85
Input event handler . . . 87
5.2.2 Controller implementation . . . 87
5.2.3 Model . . . 88
5.3 Stereo support . . . 89
5.4 Portability issues . . . 91
5.5 Performance . . . 91
5.6 Problems . . . 92
5.6.1 Transparency . . . 93
5.6.2 Picking in stereo is problematic . . . 93
5.6.3 Textboxes and big texts . . . 94
5.6.4 No 3D cursor . . . 94
5.6.5 Problems with Localization . . . 94
6 Conclusion and Future work . . . 95

Appendices

A Example of workspace format . . . 97
B Example of documentation generation . . . 109
Bibliography . . . 116

List of Figures

2.1 A typical CRC card . . . 7
2.2 Architecture of OGRE 3D, using CRC cards. . . . 8
2.3 Monitor based display . . . 34
2.4 Projection visual display: CAVE . . . 35
2.5 Head mounted display . . . 36

3.1 QuickCRC environement . . . 41
3.2 QuiCRC Path through sub-scenarios . . . 41
3.3 EasyCRC CRC card diagram . . . 43
3.4 EasyCRC sequence diagrams . . . 44
3.5 The CRC Design Assistant, CRC Card View . . . 46

4.1 Model-View-Controller Architecture . . . 56
4.2 Parts of an MVC++ application . . . 57
4.3 The software design of our tool. Rendered by Flying Circus . . . 59
4.4 Design of model component. Rendered by Flying Circus . . . 60
4.5 Design of controller component. Rendered by Flying Circus . . . 61
4.6 Design of the viewer component. Rendered by Flying Circus . . . 62
4.7 The initial screen of Flying Circus . . . 63
4.8 Top, Front and Side stable views of the world . . . 64
4.9 Manipulators on a CRC card: (a) Move, (b) Rotate (c) and Resize . . . 67
4.10 Main menu with workspace menu opened . . . 68
4.11 A CRC card. The user is editing responsibilities textbox . . . 69

4.12 A bidirectional link in editing mode 70
4.13 The 'Open project/workspace' window 70
4.14 The application 'Options' window 71
4.15 'Edit Settings' window of a CRC card 71

5.1 A typical scene graph 75
5.2 Above-And-Below format 91
5.3 Current transparency rendering. Controls are not blended with the transparent window (Class is the transparent card). 92

Chapter 1

Introduction

Systems analysis is the process of understanding a real world situation, so that we can construct automated systems supporting it. Although many marketers of object-oriented approaches to software development claim that objects are an intuitive, natural way of understanding systems, in real projects it is a challenge to identify the right ones. Furthermore, as the interest in object-oriented technology has grown in recent years, researchers have tried to find ways to aid the process of recognizing objects, their responsibilities and their public interfaces, the way these objects collaborate and communicate.

CRC cards form the basis of one of the most venerable practical techniques for facilitating object orientented design. The CRC Card technique, is included in the Extreme Programming[1] values. CRC stands for Class, Responsibility, and Collaboration. The use of CRC cards facilitates the process of discovering the real world objects which make up a system and its public interfaces. These index cards provide a simple alternative for a Collaborative Design environment, where analysts, designers, and developers are engaged in an interacting process, by using these index cards and a roleplay-driven approach, simulating the behaviour of the system. The CRC process assists analysts and users in mapping the collaborations among classes, defined by the responsibilities each has in the system being modelled.

In its simplest form, CRC cards can be applied to a design project using a stack of index cards and a

[1] Extremme programming or XP, is a deliberate and disciplined approach to software development. XP emphasizes customer involvement and promotes team work and is based on simple rules and practices.

pencil. As the design grows, an automated CRC design tool could facilitate the process of finding classes, responsibilities and collaborations, saving time and reducing design errors. Up to now users of CRC cards can find many commercial computerized 2D CRC systems. Most of these tools, try to simulate using real index cards. This means that they constrain the placement of CRC cards on a 2D plane. This limited environment can strangle the effectiveness of CRC cards usage, because it can restrict the process of coming up with new ideas and discovering new connections/relationships while moving the cards around the workspace and partitioning them in logical units and components. Moreover they don't allow the users to see the whole CRC design, so that they can distinguish patterns and connections between the cards. Finally, when analyzing big projects it becomes increasingly difficult to track all data gathered when collaborations between cards are not displayed.

1.1 Thesis Statement

In this thesis we introduce another automated tool, named *Flying Circus*, aiding the technique of CRC cards design. This tool provides a 3D environment which the user can freely manipulate. CRC cards can be created and placed anywhere in this environment and the user can visualize their collaborations by rendering directional labeled links between collaborating classes. This can increase the effectiveness of the user's spatial memory and facilitate the partitioning of cards. Moreover the concept of the CRC cards is empowered, by allowing the cards to have different sizes and colours, along with the introduction of new meta and comment fields. Documentation for a specific design can automatically be created and files containing a crc desing can also be edited with a simple text editor, allowing easy inclusion of other projects. Finally, by using stereo visual displays, like simple anaglyphic red/cyan glasses, users can have a 'true' 3D experience, which can make the interaction with CRC cards more intuitive.

1.2 Thesis Outline

The present thesis is organized as follows: Chapter 2 will introduce CRC cards, their background and a model for their usage, including brainstorming and role playing sessions. Next 3D User Interfaces will be analyzed, pointing out their advantages and drawbacks followed by an introduction in immersive technologies and more specifically, visual immersive systems. Chapter 3 describes available automated tools

supporting CRC cards, presenting the basic characteristics of each one. Afterwards, we try to analyze what is missing in those tools and how a more effective tool supporting CRC cards could be developed. In chapter 4, specifications of the tool along with its architecture and components design are described. Following, the user interface design of the tool is presented. In chapter 5, implementation details are analyzed. The tools used for the development of the tool are enumerated along with a basic description of a SceneGraph. Details regarding the implementation of components, stereo support, portability and performance of the tool are reported. Finally, Problems of the current implementation are discussed. Finishing, in chapter 6 we discuss our main conclusions and future work plans.

Chapter 2

Background

In this chapter, we introduce CRC cards. After analyzing problem solving concepts, we show how groups can use CRC cards to draw out these skills. A model for the use of CRC cards, which includes project management guidelines, brainstorming and role playing sessions, is also presented. Afterwards we analyze 3D environments and desktops and report their advantages and drawbacks. Following this, we elaborate on immersive environments and what current technology can offer.

2.1 How CRC cards were born

CRC cards were introduced by Kent Beck of Apple Computer, Inc. and Ward Cunningham of Wyatt Software Services, Inc. [9]. According to them the most difficult problem in teaching object oriented programming is getting the learner to give up the global knowledge of control that is possible with procedural programs, and rely on the local knowledge of objects to accomplish their tasks. Novice designs are littered with regressions to global thinking: gratuitous global variables, unnecessary pointers, and inappropriate reliance on the implementation of other objects.

Because learning about objects requires such a shift in overall approach, teaching objects reduces to teaching the design of objects. So we focus on design whether we are teaching basic concepts to novices or the subtleties of a complicated design to experienced object programmers.

Rather than try to make object design as much like procedural design as possible, Beck and Cunningham [9] have found that the most effective way of teaching the idiomatic way of "thinking" with objects, is to immerse the learner in the "objectness" of the material. To do this we must remove as much familiar material as possible, expecting that details such as syntax and programming environment operation will be picked up quickly enough once the fundamentals have been thoroughly understood. It is in this context of their described perspective on object design, that they introduced its concrete manifestation, CRC cards.

Studies have found the usage of the CRC cards technique stimulating the understanding of object oriented concepts [11, 18]. Moreover it was found that one of the biggest contributing factors to their success is the fact that they provide an informal and non threatening environment that is productive to working and learning [18], in contrast with other techniques such as UML diagrams [1] [33]. More recently the technique has become regarded as useful beyond the learning stage because of the subtle way it supports critical characteristics of design [10, 66], especially for responsibility driven design approaches [70, 68]. Surveys such as the *Open Toolbox of Techniques* [35] list CRC cards as a "well tried" technique and acknowledge applicability of CRC cards beyond simply learning.

2.1.1 The CRC card

CRC stands for Class, Responsibility, and Collaboration. CRC cards are index cards (or computerized versions of index cards), which are used to record suggested **classes**, the things they know about itself (knowledge responsibilities) or do (behaviour responsibilities), generally their **responsibilities**, and their relationship to other classes, **collaborations**[2]. According to Biddle et al. [11], responsibilities should outline knowledge responsibility instead of behaviour responsibility. This leads to the separation of the interface from the implementation, which comes in line with precepts of responsibility driven design [69]. Figure 2.1 shows a typical CRC card.

The back of the CRC card may be filled or left blank. It can be used to list attributes of the class or write a description of the class. Some CRC card users have found that using the back of the card for attributes can be distracting and even misleading [10]. If an attribute is significant it will show up on the front of the

[1] UML stands for Unified Modeling Language

[2] Biddle et al. [11] propose the simpler term 'helper' instead of the term 'collaborator', which they found misleading in their experiments. Kent Beck used the same term at the first draft of the initial CRC paper[9], but Cunningham changed it back to collaborator[66]

Figure 2.1: A typical CRC card

CRC card, because it will be the subject of a knowledge responsibility.

By writing down the names of the classes and thinking about what things a class must know or do, we can see how the class that we are defining interlocks with other classes in the system [8]. Moreover we can see how to reinforce the idea of encapsulation or polymorphism by moving responsibilities from one class to another and then introducing collaboration between them [10].

Note that the visual nature of the CRC cards and their physical aesthetic is an important catalyst to problem solving. According to Bellin and Simone [10], by moving the cards around on a table, we can trip new ideas, just as if we were rearranging the letters of an anagram searching for a new word. For this reason, even projects that use automated support for recording CRC cards will benefit from using printed, individual cards for brainstorming and role playing.

Even more important than just writing the cards and looking at the relationships between classes, the CRC card technique is valuable because we are working with other people [57]. Most people, regardless if they like to work in groups or not, will find that brainstorming by many minds and role playing the cards is a fruitful technique for moving from reproductive "thinking", staring their own personal set of CRC cards, to productive "thinking", realizing that they can use a novel class and responsibility allocation [10].

It is important to appreciate the flexibility of CRC cards. They are portable and can be used anywhere, away from any computers, even away from an office [66]. Besides, they can be used to drive down to a very detailed level by a technical team looking forward to design, and even by programmers. Furthermore CRC

Figure 2.2: Architecture of OGRE 3D, using CRC cards.

cards allow users and the people who will design and develop the system, to work together using the same tool [10]. Last but not least, the CRC technique has not been limited only to software engineering. Coplien and Cain [22] have utilized the cards for capturing the essential aspects of organizational modelling.

2.1.2 Example of a CRC design

Figure 2.2 shows the design of the open source, scene flexible 3D engine OGRE[3] 3D[4]. This CRC card design was developed at ICS-FORTH, in the context of an OGRE 3D refactoring project.

2.1.3 Problem solving and insight

In order to understand what to do with CRC cards, when and where to use the technique, it is helpful to establish a common understanding of why we need such tools in the first place. This has a lot to do with the nature of the human mind and the way that we solve problems. CRC cards do not tell the analyst what

[3]Object-Oriented Graphics Rendering Engine
[4]http://www.ogre3d.org/

classes to choose or how they interlock. CRC cards are a catalyst, increasing the likelihood that the analyst will be able to come up with a good solution [10].

Since the time of the ancient Greeks, scientists and philosophers have debated the question of how the human mind is able to solve problems. This search is fuelled by the belief that "thinking" distinguishes us from other forms of life. Aristotle explained in his book *On the soul* [3] the way we think by using a model in which the mind works by association, by connecting up one thing to another until a solution is achieved. J. Locke [46] and T. Hobbes [45] shaped modern scientific thought, by modifying the associative model to include the idea of trails of associations. As a problem solver works through alternatives, he is drawn to certain choices because the associations leading to those choices are stronger. Commenting on this view of problem solving, renowned psychologist E. L. Thorndike, in his *Law of Effect*, described the approach of his scientific peers as trial and error and accidental success [62].

Anyone who has worked on a software project has probably had many personal experiences that support Thorndike's definition. Many developers feel that the time spent on drawn-out analysis is a waste of time. Ultimately, a process of trial and error in coding will drive the process. Analysts on the other hand warn that no one can solve a problem without careful investigation and knowing of what this problem is. The fate of systems that neglect analysis are doomed to fates such as high cost maintenance and endless strings of enhancements [10].

The analysis part of problem solving fulfils a particular set of parameters that can not be duplicated or replaced during design or coding. During analysis, the problem solver is searching for a new way of understanding the problem that will lead to insights into its solution. Insight is the act or result of apprehending the inner nature of things or of seeing intuitively [65]. It is the accidental success that can be the foundation of deliberate successes for the rest of the project. These insights, flashes of understanding, vivid connections to the problem, become the basis for a system model that becomes the roots of the rest of the project [10].

Success often means breaking away from old associations and seeing things in a new way. According to Bellin and Simone this does not mean that association and familiar connections have nothing to do with problem solving in analysis, but they are not the driving force. This is especially true if we want to move from a system concept based on procedural models and languages to one based on object oriented models and languages [10].

Max Wertheimer considered thinking to happen in two ways: productive and reproductive. Productive

thinking is solving a problem with insight and reproductive thinking is solving a problem with previous experiences and what is already known [60]. That's why it is important to know what the problem is. Gestalt psychologists call this metacognition[60]. Metacognition describes a situation in which the people trying to solve the problem understand what it is they are trying to do [65]. They recognize that they are engaged in a process that draws on logical association. They are also aware that the solution is not necessarily self evident [10].

If we apply the idea of metacognition to analysis, one implication is going to be that teams of analysts who are aware of their method are more likely to have new ideas. Tools that support and facilitate this kind of problem solving are therefore very valuable. In fact, almost any analysis tool becomes much more powerful when the people who are using it understand what they are trying to accomplish. If the tool is designed to be a catalyst to new ideas, the team must be both patient and open to unexpected ideas. If the tool is simply used as a form of annotation of old ideas, the likelihood of new insights will be sharply decreased. CRC cards are not an exception [10].

2.1.4 CRC cards as a metacognitive process

Studies on metacognition [38] show that two groups of people will have different degrees of success solving the same problem if one group has first been trained in problem solving and reasoning prior to being presented with the problem. People who are self-conscious about their problem solving strategies do better than people who depend on unconscious response or just getting lucky. Trained problem solvers who know how the technique they are using is supposed to work are much more likely to be successful and will achieve success more quickly. This means that using CRC cards involves more than knowing where to write things down.

For these reasons, most books and research papers on CRC cards [9, 10, 11, 18, 17, 28, 33, 57, 66], propose an approach to understanding the CRC card technique that involves learning about two key problem solving facilitation strategies: brainstorming and role play. Their approach depends on teamwork, and although CRC cards can be used by a single analyst on a small system, more complex problems are much more easily resolved by a group [10]. Brainstorming and role play strategies maximize the advantages of group work, which in turn maximizes the advantages of using the CRC card technique.

Brainstorming is a strategy that has been used in all sorts of situations. The method was first popularized

in the late 1930s by Alex Faickney Osborn [51] although we could say that it has its roots in jazz, where jazz musicians incorporate untested playing into formal performance[10]. For reasons undetermined, the human mind has the capacity to draw in unexpected brilliant connections when "thinking" is the least deliberate. In addition, when this kind of mental exploration goes on in group, individual members may be carried beyond what they can do on their own into unexpected levels of accomplishment. Trading ideas in a fast and uninhibited way yields a better result than the deliberate work of one individual [10].

This same assumption applies to the use of role play in the CRC card technique. After using brainstorming to come up with the elements of the system and some possible arrangements, the team uses a play-acting technique to test different scenarios. The team poses a set of "what if" scripts and then acts them out. Each person in the team literally takes on the role of a class and using the CRC card as a script, acts out the system. The value of this strategy is that the act to "be a class" and figuring out what you have to do, triggers insights. Role play makes team members active participants and instead of just "paying attention" while reading or listening, the team members are engaged directly in the role play [10]. The CRC cards then serve as a catalyst to the fundamental, first step problem that confronts any object oriented project, finding classes.

2.1.5 The challenge of finding classes

Let's try to give a definition to the term class. Class is a set of objects described by the same declaration and is the basic element of object oriented modelling. Some have conceptualized a class as an encapsulation of data and procedures, which can be instantiated in a number of objects [26]. Others have defined a class as a set of objects that share a common structure and common behaviour [14]. A class does several things: at runtime it provides a description of how objects behave in response to messages; during development it provides an interface for the programmer to interact with the definition of objects; in a running system it is a source of new objects. Based on these definitions, a class is: a description of the organization and actions shared by one or more similar objects [5].

So classes are important because they drive the analysis at a level of abstraction that inhibits the tendency of people to get wrapped up in particulars. Human beings have a strong tendency to think in particulars, so although we want to think about classes, we get lost in the specifics of one instance or another of this class [10]. According to Bellin and Simone a critical problem for an analysis team setting out on an object

oriented development effort is utilizing information that is often delivered in a very specific form such as system documentation and converting it into a network of collaborating classes. This involves abstract thinking, which singles out the rational, logical qualities [65], translating all the specific information into a model that captures what is going on without prejudice of how it has been done. Also it means trying to find out commonalities and ways to rearrange things in new structures. CRC cards can help in this task by focussing the discussion on classes, using the language of classes and supporting it with a visual aid, the cards themselves.

Another barrier to insight when trying to find classes is that there is a tendency to lock into old ways of seeing the world. Psychologists refer to this as fixation or a mental block [65]. Studies on the problem of fixation looked at the way people solve puzzles, in particular word puzzles, called anagrams [49]. These cases are interesting because we are given the problem in one form, a string of words in a particular order and must solve the problem by reordering the letters in order to form a completely new word. Studies show that problem solvers work much more quickly if the original string of letters does not form a word. Presenting the problem in a form of a solution, like using a meaningful sequence of letters (words), before asking people to rearrange the letters, results in longer solution times and less frequent success. Even something as simple as having the nonsense strings pronounced before the problem solving begins can result in slower work and lower rates of success. The more signs that the problem is self-evident, the more difficult it is to reformulate it and come up with a solution.

Moreover, in order to stimulate productive thinking, the process of using the cards should be a group and iterative process [10]. Many minds will yield many ideas for solutions, and probably the first solution will not be the final solution.

2.2 How CRC cards should be used

In this section a set of guidelines for finding and filling the CRC cards is provided. Choosing and defining classes is one of the most important parts of an object oriented object and CRC cards are an aid. They are easy to use, informal, flexible and adaptable and encourage interaction when used in brainstorming and role playing sessions.

2.2.1 Project management guidelines

As already stated, CRC cards depend on two key working strategies: group work and iteration. Project management guidelines should reflect and respect this. To make CRC cards work, analysis teams and designers should focus on the following key items according to [10]:

Team building

The people participating in the team must feel free to express ideas, contributing not only brilliant insights but also ideas that may prove to be unusable. The process of brainstorming and role playing will only work if everyone feels free and safe about making contributions.

So notions about what is "right" or "wrong" can block progress. They can stall the group or censor good ideas. Moreover sometimes, without realizing it, if we are intimidated by a group, we cut off our own ideas thinking that they are not worthy to mention. This habit of censorship according to Bellin and Simone can be so strong that it becomes subconscious. It is exacerbated by the tendency of people in a group to take on roles: leaders, watchers, talkers and listeners.

That's why Bellin and Simone emphasize the work of the facilitators in CRC card sessions. This person should not only have great experience on the technique, but should also understand group dynamics. The facilitator is the one who sets the tone for the group by using inclusive techniques like round robins, to let everyone know that they are equal as contributors.

Moreover each member should be on the team for a reason, because they contribute a sphere of expertise that is complementary to that of the other members. This applies to personalities as well. Ignoring characteristics that affect behaviour in a group, such as including persons who like to work and figure everything out alone, in a team which is user based and exploratory is a poor choice.

The overall size of the CRC team is important and active participants should be limited to 6 [57], positioned around a conference room table, allowing interaction between all members. He proposes a CRC team that includes one or two object oriented design analysts, who have responsibilities on the project beyond the CRC sessions, one facilitator, one or two scribes, responsible for documenting any business logic and discussion. The rest of the team should be domain users[5].

[5] He also proposes that a number of non active observers participate in the CRC session

Inclusion of application experts

Users are the ultimate measure of acceptance or failure of system. At the same time there is also a sense that users are an obstacle to development [10]. Nothing gets used for its technical beauty. It is appreciated because it meets the users' needs. For this reason, users should be included in the system analysis. Interviewing users is helpful, but bringing users into a CRC card session make them part of the analysis process.

Furthermore, CRC cards offer a framework for a class driven, object oriented discussion of applications, in a format that is both understandable to users and useful to technical people. CRC cards captures user-talk directly in a medium that is valuable input all the way through the development cycles.

Coordination with a formal methodology

CRC cards are a technique that supports the task of defining the elements of a software system: the basic classes, what each class does and how they all collaborate together. However, the CRC card is not a methodology. A methodology deals with the logical principles underlying the organization of a system, usually accompanied by a specific notional scheme and a set of techniques. It models complexity through multiple views that each conveys an aspect of the system under development.

Understood in this context, CRC is a useful technique for capturing application concepts and for modelling the basic framework of a system because it can be used in conjunction with any larger scale methodology. Especially in the case of larger applications, project management entails the selection of a broader strategy for coping with the complexity of aspects involved in the system. We can think of the methodology as an umbrella that encompasses a multiplicity of techniques, one of which applies to the task of finding classes [10].

CRC cards do not work for massive numbers of classes. The limits of the human mind radically reduce the effectiveness of any analysis effort if the domain under discussion is too broad. Use a methodology to support the definition of domains that are reasonably discrete and to map the CRC information for those domains together. This is the best method to scale the CRC technique to very large systems. Bellin and Simone propose the use of no more than 12 principal classes [10].

Moreover CRC cards do not have enough notational power to document all the necessary components of the system. They can not provide a detailed view of the data objects, they do not give implementation

specifics and they can not provide a detailed view of the states of the objects. For these matters it helps to use a full scale methodology.

2.2.2 Filling the cards

But how should we fill in the cards? Let's start with the most important part of a CRC card, the name of the class that it represents.

Classes

There are many ways to identify classes. One of the easiest to start with, is noun extraction. In order to use this technique, each member of the team is required to take an investigation path before the CRC session, in order to flush themselves with knowledge and ideas about the system. Noun extraction identifies all of the nouns and noun phrases from any resource of the system. Such resources can be the document that states the requirements of the system, reports of what the users of the system expectations would be, interviews from other experienced users of the domain and any documentation or files of any available previous system [10]. Other ways to identify classes are to look for items that interact with the system, or things that are part of the system. Ask if there is a customer of the system and identify what the customer interacts with. All these can make excellent candidate classes and can be proposed during the brainstorming session.

Following the earlier suggestion, that the CRC design should not include more than 12 cards, if all of the candidate classes are core classes, the system is too big. However before deciding searching for subsystems, which can be an extensive analysis task in itself, we can analyze the candidate class list. Many of the classes can be duplicates or just irrelevant. Other candidates may actually be attributes and not classes in themselves. Rubin [57] suggests that if a class can not be named with less than three words, then it's probably not a class but a responsibility of another class.

According to Bellin and Simone [10], a good class has the following characteristics:

- Has a clear, unambiguous name easily recognized by domain experts.
- Has a name that fits in with other systems developed by your organization
- Uses a singular noun for its name, not a plural
- Begins with an uppercase letter

- Has responsibilities
- Remembers (has knowledge)
- Is needed by other classes (collaborates)
- Actively participates in the system

After we have a satisfactory list of critical classes, a CRC card should be started for each class.

Class filling problems Biddle et al. [11], have found giving names to cards straightforward and positive. Sometimes though, users give names that are verbs, rather than nouns, something which shows misunderstanding of the approach, or myopic overemphasis on some procedural element. Fortunately naming allows this to be detected and addressed early.

Another more difficult problem is that the distinction between class and object is blurred [11]. This gets worse with systems that include singleton classes, where there is only one object in a class. Moreover, as is common in systems related to the real world, there is a tendency for a class in the system to represent a concept or artefact in the real world. This confusion is more problematic when the class or object had the same name as one of the actors in the system. Börstler [15] suggests that the term 'candidate object' is banned completely. According to him, we should only handle candidate classes and we should be introducing classes (CRC cards) even if we only had single instances. Moreover he proposes the usage of role play diagrams (RPDs) [6] [16], semi formal diagrams quite similar to UML collaboration/communication diagrams, where the objects, called object cards, are based on the UML notation of objects. In RPDs an object card is an instance of a CRC card that shows the instance's name, class name and current knowledge, according to the information noted on its corresponding CRC card.

Responsibilities

After creating the CRC cards with the set of core classes, the team can begin to work out the way in which those classes work together to comply with the requirements of the system. These can be discovered through the use of scenarios and role playing or by just looking at each class and writing down the preliminary set of responsibilities and test their solution with a role play.

[6] From now on RPDs

To compliment the noun extraction technique above, we can use verb extraction. Verb extraction identifies all of the verbs in a problem statement or/and use case scenario. These are usually good indicators of actions that must be performed by the classes of the system. Other techniques include asking what the class knows and what information must be stored about the class to make it unique [57]. Biddle et al. [11] report that many people find it easy to think of occupational or business roles, while trying to find responsibilities.

As already mentioned, responsibilities should outline 'what', behaviour responsibilities, rather than 'how', knowledge responsibilities. As Grady Booch says [14], "*When considering the semantics of classes and objects, there will be a tendency to explain how things work; the proper response is 'I don't care'*". Knowledge responsibilities should be included when a class should provide information about itself to other classes, in order to collaborate with them [10].

After finding responsibilities, we write them down to the appropriate place on the CRC card.

Collaborations

Collaboration occurs when a class needs information that it doesn't have. Classes know specific things about themselves and they do not exist in isolation. They exist as members in a system and very often to perform a task, a class needs information that it doesn't have. Often it's necessary to get this information from another class, in the form of collaboration.

Collaboration can also occur if a class needs to modify information that it doesn't have. One property of information that a class knows about itself is the ability to update information. Often a class will want to update information that it doesn't have. When this happens, the class will often ask another class, in the form of a collaboration, to update the information for it.

Generally for a collaboration to occur, one class is the initiator. In other words, there has to be a starting point for collaboration. Often the initiating class is doing very little work beyond initiating the collaboration itself [57].

The easiest way to establish paths of collaboration between classes with CRC cards is by setting up scenarios and then using CRC role play to test them. Moreover Booch [14] sees hierarchies, inheritance relationships, as a basic and early clue to collaboration. Collaboration can also be discovered by thinking in terms of dependencies. If a class has to assume responsibility for an action, it often depends upon another class for the knowledge needed to fulfil that responsibility [10].

After finding collaborations, we write collaborations down to the appropriate place on the CRC card.

Note that the class doing the using should list the other as a collaborator and not the other way around. Bellin and Simone [10] propose that the collaborations are recorded by writing the name of the server class next to the responsibility on the client class CRC card.

Collaborations filling problems Biddle et al. [11] report that with collaborators some important points did arise. The most common point was uncertainty about the term "collaborator". In particular, whether one class using another should result in each class noting the other as collaborator. They point out that some people prefer the simpler term "helper" and this made the direction of help clear.

2.2.3 Brainstorming sessions

As already mentioned, CRC cards is an excellent technique for taking advantage of the group process in problem solving. Group problem solving works by throwing together a variety of ideas, compare them and synthesize unanticipated solutions. The first step in this process is brainstorming.

Description of brainstorming principles

In a brainstorming session, the group guided by a set of principles [10], combats the censorship and frees team members to propose any alternative from the most logical to the most absurd.

All ideas are potential good ideas The first principle of brainstorming is based on the importance of breaking down any tendency to censor before you speak. Anything a participant of the session comes up in response to the question at hand, should be offered to the group as an idea. All ideas are equal, because all ideas have the potential to lead to an unanticipated solution.

Think fast and furiously first, ponder later The next principle of successful brainstorming relates to the pace of the group. If each person waits quietly for a good idea, the creative juices won't flow. Studies have shown that when people in a group take turns in quick succession, one person triggers the other [10].

Give every voice a turn Some of us are predisposed to talk all the time; others are reluctant to talk at all. To prevent this, Bellin and Simone [10] suggest starting with one person and then go around the table, so that each contributes an idea. This pattern should continue until team members are truly forced to pass

because they are out of ideas. Moreover because brainstorming part is at the beginning of the process, it can set a mood that lasts for the duration of the project. If people who tend to keep quiet experience acceptance when they speak early on, they will be more likely to be regular contributors throughout the project.

A little humour can be a powerful force Humour helps convert a random group of people into a cohesive team. Laughing together creates a bond between people, dispels tensions and signals a shared perspective. As a result, the group automatically establishes a baseline of trust and interdependence that makes it easier to debate the serious issues sure to arise later.

Using brainstorming to find classes

Bellin and Simone [10] propose a step by step approach to creating a set of candidate classes and narrowing that list down to a group of core classes that can be written on the CRC cards. These four steps were chosen because they have proved successful for groups using brainstorming in a wide variety of situations.

Review brainstorming principles Begin the session by reviewing the four brainstorming principles. It is a good idea to write them on a board or post them on a wall in the room. Just seeing these guidelines written on the wall can help people remember to put aside personal style and work with the team.

State session objectives The facilitator should take the time to state the objective for the brainstorming part of the meeting. The objective should be precise and fairly narrow and the facilitator should write it at the top of the board. Writing down things is very important, because it provides an anchor when the discussion slides off track.

Use a round-robin technique One of the most difficult aspects of teamwork is getting everyone in the room to work and talk. The best way to avoid this dynamic is to use a round robin technique to solicit suggestions. As each member of the team contributes an idea, this idea is written down on the board. The goal of the round robin is to allow the group to move ahead at even tempo, giving people enough time to think. Short pauses are fine, but breaks of more that 60 seconds can interrupt the momentum and ideas be lost. If someone is really stumped, they can pass for that round, but they should take their regular turn the next time around. The brainstorming is complete when everyone in the group has to pass.

Discuss and select This is when all of the ideas produced by brainstorming are discussed and winnowed down. It is the time to sift out the best items. But instead of inviting comments for every item in the list one by one, Bellin and Simone propose to ask for suggestions about clear cut winners; those that everyone agrees are critical classes of the system and write them down on a separate list. Next we should address the items that do not fit at all. Finally address the items that fall in between "yes" or "no". The best way to do this is to assign a limited time for discussion and then take a voice vote on whether or not to include the item. If the group can not decide, establish a consistent policy, of either include all undecided items or postpone inclusion until you understand the system better. The last one allows working with items that are definitely at the core of the system.

2.2.4 Role-playing scenarios

Filling in spaces on CRC cards does not guarantee that the selected model will address all of the demands that will be placed on the system in day-to-day operations. An interactive dynamic modelling strategy that mimics the interaction between classes when work is being done can ensure accuracy. CRC role play extends the CRC technique by using the CRC cards as the basis for acting out the system. Scenarios give us the advantages of prototyping without writing any code.

How CRC card role play works

CRC card role play involves six major steps [10]:

- First the team draws up a list of scenarios that will provide them with a line of action. The initial list does not need to be comprehensive. However, before the analysis process is over, a complete list should be developed to confirm that the CRC cards allow the system to meet all of the demands.

- After the scenarios have been listed, each team member is assigned the role of one or more classes. The CRC cards for those classes are given to the actor.

- Next, using the scenario as a script, the team acts out the behaviour and collaborations written on the cards. Performing the part of their classes, team members make requests when there needs to be a collaboration, or announce what they are doing themselves.

- If a behaviour or piece of information is missing from the card, the action is interrupted and the CRC card or cards involved are corrected.
- The two previous steps are repeated until the action flows smoothly.
- When basic scenarios run without interruption, the CRC cards are used to role play scenarios for the less common scenarios and major exceptions.

Developing the role play scenarios

A scenario is like the script of a play. Scenarios are a way to work out the dynamics of the system by scripting what has to be done, and the order of events. Scenarios are an orderly sequence of class interactions which meet a specific need in the problem space. They define the way classes work together to accomplish goals [10].

Once the obvious scenarios are recorded, the team will inevitably come up with borderline cases, scenarios where the routine activity will go on only if certain conditions can be met. Moreover some scenarios may be combinations of situations so complex or unusual that they are not necessarily going to fall within the boundaries of the project. Once the basics of the system are in place, role play of those situations with the existing classes can give the team a sense of the flexibility of the system design and its adaptability to change [10].

If the list of the scenarios is very long, the project may need to be subdivided. CRC can be and is used in for large systems, but in smaller chunks. If there are too many classes, the ability of the group to comprehend and evaluate patterns will diminish and the technique will not be as effective [10].

Performing the system simulation

Each session begins with a warm up, moves into enactment (the actual role play) and concludes with an assessment. The two last parts can be repeated several times. Bellin and Simone [10] propose that the CRC role play session takes no longer than two hours, where each scenario takes about 10 minutes for enactment and 15 minutes for assessment.

The warm up The role of the warm up is to inspire confidence to each of the role play session members and to increase participation. In CRC card role play, the work of the group is distributed to everyone because

everyone has to play the role of some number of classes. The silence of even one person during the role play can leave essential issues unturned. So introductions and interactive exercises, like puzzles can build group identity and focus group energy [10].

Enactment Enactment is the heart of the CRC role play. Once the group has a list of scenarios they are ready to act out the system. The first step is to identify the scenarios the group will work on first, such as scenarios that touches the central framework. The order of the role play can be scheduled in a systematic way, working from "must-do", to "can-do-if" and "might-do" scripts.

To start the enactment part of the session, we distribute the CRC cards to the team members. At the beginning the distribution can be random, but collaborating classes should be given to different actors. Wilkinson [66] suggests that the classes are assigned to the domain experts of those classes. Also if the system is large and there are a number of role play sessions, Bellin and Simone [10] suggest that rotating class assignment might be more productive.

The enactment begins when someone in the team initiates a scenario by holding up a card, stating what they need to accomplish. During the role play, each actor reads the responsibility written on the CRC card and asks for help from a collaborator if necessary. The collaborator then responds, lifting that CRC card to indicate that it collaborates. The actor reads the behaviour of the class and either continues holding up the card, calling for another collaborator or lowers it, having completed its task.

As the enactment progresses, the whole team can visually follow the collaborations and see how allocation of responsibilities for information and behaviour affects the action. Watching the action of the role play should make the impact of allocation decisions clear. If the wrong class holds the wrong information it will show up during the role play [10]. If the client-server relationships do not make sense, the team will see how difficult it is to get the work done. Class actors will realize based on their cards that they are missing information or that they are being asked to carry out responsibilities they don't have. It may be that they need to collaborate with a class not listed yet on the card.

Bellin and Simone [10] suggest that problems are not resolved during the enactment. They propose that a scribe take notes where problems occur and where a scenario gets stuck. If there is an obvious change, the change should be made on the CRC card and the scenario should be run again. If more thorough thinking should take place for a problem, the problem should be recorded and the team should move on to the next scenario.

As the system evolves, the role play enactment should run smoother and once the team can role play the "must-do" scenarios without interruption, the team should move on to the "can-do-if" scenarios. Changes that cause problems for the "must-do" scenarios should be avoided [10].

Assessment The assessment phase is the time in which the group can really delve into any problems encountered in the role play. This may be a straightforward discussion and approval process, or it may involve a repetition of the enactment and assessment cycles. The problematic scenarios should be reviewed and the problem areas should be analyzed. Bellin and Simone [10] propose that complex changes to the classes are discussed first. The team should be focused on the objective of confirming card behaviours and collaborations, and discussion about functional details should be avoided. When the problems of a scenario have been discussed and solved, re-verify cards with another role play enactment, before continuing with the next scenario [10]. When the cards fulfil the goals of the users, as scripted in the defined scenarios, then the CRC cards are correct.

2.3 3D User Interfaces

Nearly all of today's computers are running window based 2D desktop systems using a windows, icons, menus, pointing (WIMP)[7] interface. These systems allow the concurrent execution of multiple applications, providing mechanisms to group related objects and ease the use of available resources. The workspace is presented using a desktop metaphor with movable and overlapping windows. Application functionality can be accessed through different widgets. Input is given using a mouse or keyboard and the actions are serial in nature, so the input form the user and the output for the user are handled one at a time.

As computer Graphical User Interfaces (GUIs) [8] are loaded with increasingly greater number of objects, the next step in constructing User Interfaces UIs[9] is adding a D to 2D and move from predominantly iconigraphic, 2D representations to more realistic, 3D representations [4]. A definition of 3D User Interface is a UI that involves a 3D interaction [19]. The implicit assumption under the increasingly populate attempts to develop compelling 3D environments is that users will find it natural and intuitive to navigate virtual spaces [54].

[7]From now on WIMP

[8]From now on GUI

[9]From now on UI

2.3.1 Limitations of 2D interfaces

Even though WIMP interfaces have been proven to be effective for simple document handling and other common office tasks, it has been argued by many [7, 63] that WIMP interfaces can not cover all user needs today. New applications would benefit from other types of user interfaces and new interaction techniques.

One of the main disadvantages of WIMP interfaces is that the use of a widget is individually easy to learn, but aggregating them creates complexity, as pointed out by van Dam [63]. The more the complexity of the application increases, the harder and more cumbersome the interface becomes to use.

The serialized nature of WIMP interfaces restrict the user and separates the user even more from the feeling of real time working, which is desirable for many applications and tasks. The lack of parallelism on input also restricts or even prevents from using bimanual input, which has been found more efficient for some tasks [50, 6].

2.3.2 Movement from 2D to 3D

3D computer graphics is becoming more and more popular due to the increased availability of 3D hardware and software on all classes of computers. With this new technology, new problems have also been revealed. Many studies have demonstrated that 3D applications are significantly more difficult to design, implement and use than their 2D counterparts [36]. Therefore, great care must be put into the design of user interfaces and interaction techniques for 3D applications.

According to Wurnig [71] there are three foundations on which a 3D desktop can be build:

- Generalization of 2D desktop metaphor to 3D. The field of 2D desktop systems is very well researched and most people do already have knowledge in working with them. This approach has two drawbacks. Inexperienced users will need more time to get familiar with the desktop system and techniques that were designed for 2D can not entirely exhaust the capabilities of 3D work space.

- Creation of a 3D desktop metaphor that is not based on 2D at all. The goal is intuitive interaction with the real world as model. Knowledge of 2D systems is neither advantageous nor disadvantageous.

- Intuitive interaction, supplemented with suitable concepts of 2D desktops. If we design manipulation in the 3D environment as intuitively as possible, the 3D desktop will be easy to handle and predictable even for inexperienced persons. Some concepts of 2D desktop systems might as well be useful in

3D. Instead of reinventing them we could try to extend their functionality and apply them to the new environment. The necessary pre-knowledge for working with the system must be held at a minimum.

2.3.3 Advantages

The specific advantages of realistic 3D GUIs are unclear. Moreover, it is also unclear which attributes about a realistic 3D GUI would make it more useful. Note that the term 3D is not used in the sense of a virtual reality immersive display, but we are talking about a normal, everyday desktop computer screen.

Take advantage of the human spatial memory

Humans operate in 3D space everyday. We use the spatial cognition in order to find our way through the maze of daily life. Spatial memory is the ability of humans to remember the physical relationship (distance and orientation) of objects to each other [21]. Clearly a 3D GUI should not merely be the traditional GUI enhanced with the depth dimension. In the natural 3D world, there are many different factors that "codify" objects. Many of these factors are visually salient attributes that have us to recall objects. If we transfer these factors onto the interface, the interface is more likely to involve a non-regular placement, differing external shapes, enhanced colour, landmarks connectivity, and potential semantic associations than in a 2D GUI. In general 3D GUIs may provide more dimensions to distinguish and identify objects. The more the dimensions available to the user, the greater is the chance of him being able to relate an attribute to a dimension. As a result users might find and recall the objects more quickly in such a 3D GUI than in a 2D GUI [4].

Ark et al. [4] examined whether and which factors of a 3D GUI affect how users can acquire target objects in a laboratory experiment. The results indicate a user will search for and acquire objects more quickly if they are presented with a 3D ecological[10], realistic interface rather than a 2D regular, iconic interface. Specifically, an ecological layout and a 3D realistic representation of objects positively affected experimental task performance. Interestingly, the effects of ecological layout and 3D realistic representation are additive. That is, the subjects' performance was better when either the ecological or the 3D realistic representation was present, and they performed better when both were present. Moreover, the interaction between these factors was not significant; their contributions to the interface are independent.

[10]Layout with non regular placement with landmarks and connectivity

In addition another experimental 3D interface for bookmark-handling, designed to evaluate the impact of spatial cognition in document retrieval by Brunstad et al. [21], showed that spatial memory plays a role in 3D virtual environments. Users using the experimental 3D interface reliably facilitated speedy retrieval of web pages when compared to Microsoft Internet Explorer bookmarks, by allowing users to leverage visual as well as textual cues in finding document locations.

Lastly, Tavanti and Lind [61] described an experiment comparing the effectiveness of spatial memory in computer generated 2D and 3D displays. Their tasks involved recalling the location of letters of the alphabet hidden behind cards depicted in hierarchical 2D and 3D displays. They found that the participants' spatial memory was much better in the 3D condition concluding that a realistic 3D display better supports a specific spatial memory task, namely learning the place of an object.

On the other hand, there is at least one research paper [24] suggesting that the effectiveness of spatial memory is unaffected by the presence or absence of three dimensional perspective effects in at least monocular static displays. Their experiment is the same as that of Tavanti and Lind, except the fact that in their 2D interface they incorporated perspective effects. They concluded that there is no real difference to the effectiveness of spatial memory, between the improved 2D interface and the 3D interface on monocular static displays.

They are attractive

With the current generations of GPUs[11][12] and the development in the computer games market it seems inevitable that companies will also like to incorporate 3D graphics in their business applications. Future applications should have a more modern look so that users will have the feeling that they are using something new and attractive. The users of today do not want their interfaces to look the same as they did in the beginning of the eighties, when the WIMP interfaces where first developed. The wanted situation in this context is to develop applications that embrace this new technology, not to miss out on its new opportunities. A modern look and feel of the interface will no doubt attract users. This can be seen in the field of computer games where 3D modelling has had a tremendous impact on the industry and recently on some desktop window managers using 3D effects like Beryl, Compiz and Aero.

[11] GPU: Graphic Processing Unit

[12] There is a trend to merge processors with GPUs making 3D graphics readily available everywhere. Moreover last GPUs generations do not include a 2D core and everything is done using the 3D engine

2.3.4 Drawbacks

Although the potential exists for 3D UIs to become the way to communicate with our computer in the near future, there are still many open research areas and technological limitations.

Navigation, orientation and way-finding problems

Navigation in 3D worlds is a long standing challenge for computer graphics and visualization applications [34]. Moving freely in a 3D environment is not that intuitive for human beings as we commonly are locked to a plane as we walk or drive a car. Few computer users have experience of flying an airplane, a chopper or a spacecraft [12]. Especially large scale 3D environments exacerbate the situation to the point that users become annoyed and the 3D experience is ruined [64]. As a consequence new navigation tools have been proposed, like **Virtual Prints (ViPs)** [32].

No natural 3D interaction

Moreover, because of technological limitations, natural 3D interaction cannot be completely reproduced and such devices are still very expensive [20]. For user monitoring, most position tracking systems report the complete 6 DOF[13] positions of receivers mounted on parts of user's body and mobile devices [58]. Furthermore to support physical user input by body posture and gestures, we need exotic devices like data input glove devices, huge platforms and uniforms full of sensors. However there are occasions when only the three rotational DOF or only the three translational DOF are required. Hanson et al. [34] try to constrain 3D navigation with 2D controllers, like the 2 DOF mouse, moving the user on a constrained subspace.

No guidelines for the design of 3D worlds and UIs

There is some research underway which should lead to useful guidance on 3D worlds design and development, but generally the work of 3D environment's specification and building is characterized by limited guidance [67]. Lately, some tools that let the user develop, deliver and work with guidelines have been implemented both of them especially for virtual environments and the virtual reality community has started gathering knowledge, including existing guidelines for 2D UIs. Such tools include **i-dove**[14] [42] and the

[13] DOF: Degree of freedom, a particular way a body may move in space, including rotation and translation

[14] Available at http://i-dove.ics.forth.gr

Portal for guidance and standards for Virtual Reality[15] [52].

No standard library of 3D UI components

To a large degree, the 3D UI community has ignored the problem of implementation and development tools, although this problem has been recognized [36]. Most techniques are implemented using existing 3D toolkits and there is no standard library of 3D UI components available. There are many factors that make 3D UI implementation problematic, including the following according to Bowman et al. [20]:

- 3D UIs must handle a greater amount and variety of input data
- There are no standard input or display devices for 3D UIs
- Some input to 3D UIs must be processed or recognized before it is useful
- 3D UIs often require multimodal input and produce multimodal output
- Real-time responses are usually required in 3D UIs
- 3D interactions may be continuous, parallel, or overlapping

So a standard library of interaction techniques or technique components, containing generic, reusable implementations of fundamental interaction techniques for the universal tasks would be a step forward for further research in 3D interaction. This library should interoperate with other toolkits used for developing 3D applications.

2D tasks not always transferable to 3D

Although 3D UIs are often considered as an extension of the 2D direct manipulation paradigm in 3D, it is difficult to build a whole new UI that relies solely on direct manipulation[16] [30]. Adding degrees of freedom to tasks that do not need them can be the cause of undesired complexity. Tasks that are 2D in nature are not necessarily transferable to 3D. Direct manipulation is not always obvious [25], especially when using 2D input devices, because correlating 2D hand movement in the real world to object move in the synthetic world can be difficult [36]. Consequently indirect manipulation may be sometimes desired or required.

[15] Available at http://hci-web.ics.forth.gr/Intuition

[16] Direct manipulation is a human-computer interaction style which involves continuous representation of objects of interest, and rapid, reversible, incremental actions and feedback.

2.4 Immersive environments

A virtual world is an imaginary space often manifested through a medium. A key component of a computer generated experience is how the user perceives the environment. If the medium let the user experience the world and interact with its objects in a physical way, feeling that he is inside the environment, then this is called an immersive environment.

It is important to be clear about the meaning of the term immersion. Webster[17] defines it as the state of being absorbed or deeply involved. Clearly, immersion can occur while we are watching a movie or a television, or while playing video games [56]. Experience suggests that proper 3D cues and interactive animations aid immersion and that the act of controlling the animation can draw the user into the 3D world. In other words, immersion should not be equated with the use of head mounted displays: mental and emotional immersion does take place, independent of visual or perceptual immersion [56].

The ultimate goal of an immersive environment is to become an interfaceless medium. The experience would be designed so well that the boundary between the real and the virtual world would be seemingly nonexistent. The user interfaces would mimic precisely how the user experiences the real world. This is considered by many to be the ultimate interface [58].

The human perceptual system has at least five senses providing information to the brain. On such immersive systems, some of the human senses are presented with synthetic, computer generated stimuli. The physical perception of the virtual world is based on what the computer displays, where the term display is used broadly to mean a method of presenting information to any of the senses [58].

2.4.1 Immersion through vision

It is generally easier to implement a system that limits the number of sensory displays. However the inclusion of additional senses almost always improves immersiveness. Vision is the most important individual sense to the quality of an experience and most systems include some type of an immersive visual display [48]. Moreover Slater states that visually immersive environments increase task completion performance [59]. The other two important senses is audition and tactition. At the following subsections we will see how humans perceive depth information, the properties of visual displays and the basic displays types.

[17] http://www.merriam-webster.com

2.4.2 Visual depth cues

Before discussing visual displays and their properties, we should understand how humans perceive information regarding the relative distance of objects. According to Sherman and Craig [58], humans use many indicators of distance, which are called depth cues. The different depth cues are listed below:

Monoscopic image depth cues

These can be seen in a single static view of a scene, such as in photographs and paintings. They include:

- **Interposition**, when one object occludes another
- **Shading**, gives information about the shape of an object
- **Size**, to determine the relative distance between two identical objects
- **Linear perspective**, parallel lines converge at a single vanishing point
- **Surface texture gradient**, less detail of a texture at a distance
- **Height in the visual field**, horizon is higher in the visual field than the ground in our view
- **Atmospheric effects**, such as haze and fog, cause more distant objects to be visually less distinction

Stereoscopic image depth cues, Stereopsis

Stereopsis is derived from the parallax between the different images received by the retina[18] in each eye. The stereoscopic image depth cue depends on parallax, which is the apparent displacement of objects viewed from different locations. Stereopsis is particularly effective for objects within about 5 meters.

Motion depth cues

Motion depth cues come from the parallax created by the changing relative position between the head and the object being observed. Depth information is discerned from the fact that objects that are nearer to the eye will be perceived to move quicker across the retina than most distant objects.

[18] The retina is a thin layer of neural cells that lines at the back of the eyeball

Physiological depth cues

Physiological depth cues are generated by the eyes muscle movements to bring an object into clear view. The amount of muscular movement provides distance information, especially for object which are close.

2.4.3 Properties of visual displays

According to Sherman and Craig [58] there are many properties associated with all visual display devices. These properties vary from one display to another and can be grouped in two categories: visual presentation and logistic properties.

Visual presentation properties

The visual properties of a display device are a significant factor in the overall quality of an immersion. Careful consideration of these optical properties must be made based on the requirements of the intended applications. However, there is typically a trade off in cost that must be measured against the needs for quality. The most important properties are listed below:

- **Colour**, monochromatic or trichromatic colour, available colour channels. Monochromatic are brighter and with more contrast
- **Spatial resolution**, the number of pixels or dots presented in the horizontal and vertical directions. The more the pixels of a display, the more information we can get from an image.
- **Contrast**, a measure of the relative difference between light and dark. High range contrast makes it easier to distinguish various components of displayed information.
- **Brightness**, a measure of overall display light output. Images projected onto a screen become dimmer as the size of the screen increases.
- **Number of display channels**, where a visual display channel is a presentation of visual information displayed for one eye. In order to achieve stereopsis we need two visual display channels. The brain fuses the pair of images into a single stereoscopic visual display.
- **Focal distance**, which is the apparent distance of the images from the viewer's eyes.

- **Opacity**, the display can hide or occlude the physical world from view or it can include it.
- **Masking**, for see-through displays, masking is necessary when virtual objects come between some physical object and the viewer's eyes or when a physical objects should appear in front of a virtual object.
- **Field of view or FOV**, which is a measure of the angular width of the user's vision that is covered by the display at any given time. The normal horizontal field of view for a human is approximately 200 degrees with 120 degrees of binocular overlap.
- **Field of regard or FOR**, is the amount of space surrounding the user that is filled with the virtual world. In displays with less than complete FOR, stereopsis can be lost when a nearby object is only partially in the display.
- **Graphics latency tolerance**, the amount of lag time between user movements and the update of the display.
- **Frame rate**, the rate at which images are displayed. The faster the frame rate the better the user experience. Modern motion picture film captures 24 frames per second and 30 frames per second are the least acceptable limit for an immersive environment.

Logistic properties

In addition to the visual display specifications, there are a number of logistical factors related to displays used in various applications like the following:

- **User mobility**, can affect both the immersiveness and usefulness of a user's experience. Cables and stationary displays limit user movement.
- **Interface with tracking methods**, displays limit the selection of tracking methods, since some of them have an operating range of a few meters or need specific mounts.
- **Environment requirements**, since it can affect the choice of the visual display. Projection systems need low-light environments and big rooms.
- **Associability with other sense displays**, like aural displays.

- **Portability**, depending on who will be using the system and how far the system will have to travel.
- **Throughput**, the number of people able to take part in an immersive experience.
- **Encumbrance**, a very important factor especially for head mounted displays.
- **Safety**, for example a user wearing a head mounted display can trip over real-world objects.
- **Cost**, which varies greatly.

2.4.4 Stereopsis in visual displays

As already mentioned, the number of display channels is a property of a display. There are several ways to achieve a 3D stereoscopic visual display, by using multiplexing, which permits simultaneous transmission of two or more signals through the same channel. Sherman and Craig [58] report the following multiplexing methods:

- **Spatial multiplexing**, consisting of positioning separate images in front of each eye, by using two small separate screens.
- **Temporal multiplexing**, or time interlacing, presents different images for each eye using shutter glasses, which shutter the view for one eye, in a coordinated, timed sequence, to prevent it from seeing the other eyes view. The normally opaque lens shutters become transparent for one of the eyes during the time the proper view for that eye is presented.
- **Polarization multiplexing**, accomplished by overlaying two separate image sources filtered through oppositely polarized filters, for example by displaying one channel through a horizontally polarized filter and the other through a vertically polarized filter. The participant wears a pair of glasses with a horizontal polarized filter over one eye and a vertical polarized filter over the other.
- **Spectral multiplexing**, or anaglyphic stereo, displays the view for each eye in a different colour. Special glasses neutralize the view for the incorrect type, because each lens correlates to one of the coloured views and in effect washes it out.

  Creating good stereoscopic displays is difficult, and doing it improperly can cause distress in the viewer, which often takes the form of headaches or nausea. Moreover, rendering stereoscopic images

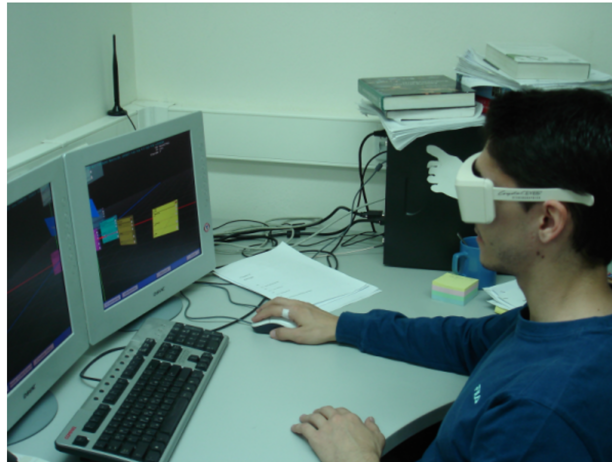


Figure 2.3: Monitor based display

takes about twice the computational resources, requiring twice the graphics hardware, a reduction in frame-rate, or a reduction in image complexity.

2.4.5 Visual display types

Now that we have covered the qualities that differentiate visual display devices from another we can look at the major visual display types.

Monitor-based - or Fish-tank

This is the most common and simpler form of visual displays. The name fishtank comes from the similarity of looking through an aquarium glass to observe the 3D world inside. Instead of just using one display, we can also use a multi screen approach. Stereopsis can be achieved by using shutter glasses or anaglyphic stereo. Figure 2.3 shows a user wearing shutter glasses.

This kind of display has the advantage of being extremely cheap. Most of its technology is mass produced, making it readily available. Even the necessary stereoscopic display hardware is widely available. On the other hand, it has the drawbacks that the user must always face in a particular direction to see the virtual world and the system is generally less immersive than most other displays. The reason for this is that the real world takes up most of the viewer's field of regard and only a small region is filled by the virtual world.

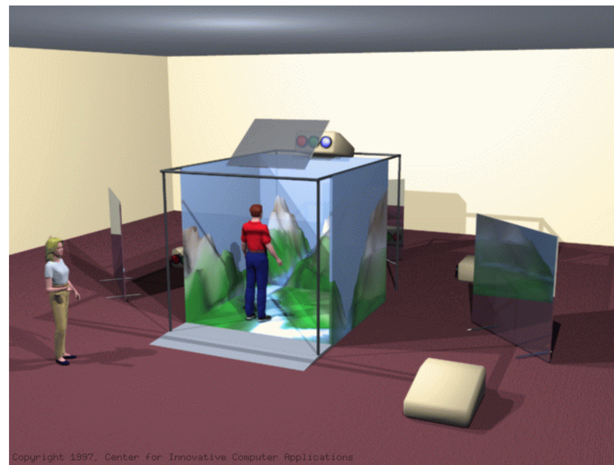


Figure 2.4: Projection visual display: CAVE

Projection based

Projection based visual displays are stationary devices. The screen is much larger that the typical fish-tank display, thereby filling more of the participants' field of view and regard. Although large display systems can be created by setting several CRT monitors side by side, projection systems can create frameless adjoining displays, making the scene more seamless. Most projections systems are rear-projected to avoid the participants casting shadows on the screen. There are many kinds of such displays like single or multi screen projection displays. Single screen projection systems include *Responsive Workbench*, a table top configuration, *ImmersaDesk*, a drafting table style and the *Infinity Wall*, a high-resolution, single screen, multiple projector system. Multi screen displays include CAVEs [19] where projectors are directed to three, four, five or six of the walls of a room-sized cube and Cybersphere [29], the fully immersive spherical projection system. Figure 2.4 shows a CAVE.

A nice feature of projection based displays is that it generally occupies a larger portion of the viewer's field of view than either monitor or head based ones. Moreover because of the larger views, more participants can observe the world alongside with the user of the system standing far enough from the display. So there will be less eye strain in comparison to other displays where the eyes are a few centimetres from the display. Furthermore because the viewer is not isolated from the real world, projection systems are very attractive

[19] Cave: A Cave Automatic Virtual Environment

Figure 2.5: Head mounted display

for applications in which collaboration with co-workers is beneficial. This also means that the users can see themselves and the control devices they are using and also that they have freedom to move around, without hurting themselves.

On the other hand the projection based displays are too expensive, need more time for the calibration of projectors and require more powerful computers and graphics subsystems because of bigger resolutions. Also the fact that the user can see the real world, can lead to the occurrence of occluding errors which can confuse the user and destroy immersion. For example a virtual object is displayed closer than a real world, such as the hand of the user, but the occlusion of the virtual object by the real object, overrides the other depth cues. Lastly because projection systems are big, they require big rooms with appropriate lighting and have portability problems.

Head based

Head based displays are perhaps the equipment most people associate with virtual reality. Figure 2.5 shows a user wearing a HMD. Unlike monitor and projection based displays previously discussed, head based displays are not stationary, they move in conjunction with the user's head. There are many styles of such displays, like head-mounted displays (HMDs), counterweighted displays on mechanical linkages like BOOM and experimental retinal displays, using lasers.

The most positive feature of head based displays is that the field of regard covers the entire sphere

surrounding the viewer. Also many head based displays are more portable than most stationary displays and they take up very little floor space. Another very important advantage of head based displays at least for some applications is the ability to mask the real world from the user. This also means that these displays can be operated in a wider range of venues than projection systems.

Because head based displays mask the real world from the user, it is important that their use is limited to protected areas. In addition the use of head tracking technologies is imperative. Humans are used to see different things when they move their head. Else the human brain thinks that something is wrong and we start feeling nausea. Moreover any lag in the tracking and visual image generation systems can cause noticeable problems to the users, like visual confusion [58]. Also many head based displays are somewhat encumbering and cannot be used for an extensive time period. Finishing, the field of view of a typical head based display is very limited. The same applies for the resolutions. Head based displays with big resolutions and fields of views are too expensive.

See-through head based displays

See through displays are primarily designed for applications in which the user needs to see an augmented copy of the physical world, with computer generated images or to map a virtual world onto another virtual world. Two methods are used to implement such displays: The first one is the optical, using lenses, mirrors and half silvered mirrors and the second one is the video, adding computer images onto a video image of the real world.

The fact that the real world is part of the environment means that the constraints of the real world will affect what can be done in the virtual world. Furthermore, accurate occlusion of objects is very difficult, because determining the proper interposition of objects in the rendered scene is not a trivial problem. Also it is important that very fast tracking technologies are used, because for the view of the real world there is no lag. This can be corrected when using the video method, because the delay of the input can be matched to the delay of the virtual world display.

Chapter 3

Related work

In its simplest form, CRC cards can be applied to a design project using a stack of index cards and a pencil. Each card identifies a class and its properties and relationships with other cards. As a project grows, automated tool support for CRC cards can save time and reduce design errors. As information is entered or modified, references between cards can be instantly updated. Verification checks ensure consistency and completeness. Design scenarios can be identified and simulated.

With an automated tool, CRC card data is entered into a property dialog, much like writing it on a paper index card. One big advantage in using a tool is that new cards can be automatically created for you for undefined superclass, subclass or collaborating class references. Information can easily be inserted, changed or deleted and the card can be resized. When renaming a card, all references on other cards are instantly updated.

In today's world of pressing deadlines and distributed development teams, there are huge advantages to using electronic design documentation. It provides instant network access for peers, reviewers, management or customers. It provides a means to communicate across geographic boundaries, time zones and can streamline the development process.

3.1 Existing CRC applications

Right now there are very few tools that facilitate system analysis and design using CRC cards. Most of them have been developed in universities, to aid students understand object oriented software engineering. On the other hand CRC cards have also been embraced by commercial systems, either specific to CRC cards or suites of tools for object oriented design.

In this section we will describe the most important available CRC applications.

3.1.1 Visual Paradigm for UML

Visual Paradigm for UML[1] is a commercial computer aided software engineering (CASE) tool, developed by Visual Paradigm that uses CRC cards to capture the requirements of an application. The support of CRC cards though is pretty basic. The user is able to define the desired set of cards and each card can hold information about its name, a brief description, super classes, subclasses, attributes, responsibilities and collaborators. Then the user can export the cards to UML class diagrams.

3.1.2 QuickCRC

QuickCRC[2] is a commercial tool for designing object-oriented software with CRC cards on Mac OS X or Windows, developed by Excel Software. With QuickCRC's diagram workspace you can create card and scenario objects. Each card has a class name, description, super classes, subclasses, attributes, responsibilities and collaborating objects. Information can be entered through a dialog or edit on-screen by dragging or renaming cards, attributes, responsibilities and collaborations. Figure 3.1 shows a a screen shot of the QuickCRC environment.

More specifically this tool has the following features:

- **Super class and subclass support** Super class or subclass cards are added by typing the name or by selecting from a pop up list of existing cards.
- **Supports namespaces** Each card has a namespace prefix. The base namespace is used by default, but the designer can define and use up to 1000 namespaces. Namespaces are used to partition cards

[1] http://www.visual-paradigm.com/product/vpuml/

[2] http://www.excelsoftware.com/quickcrcwin.html

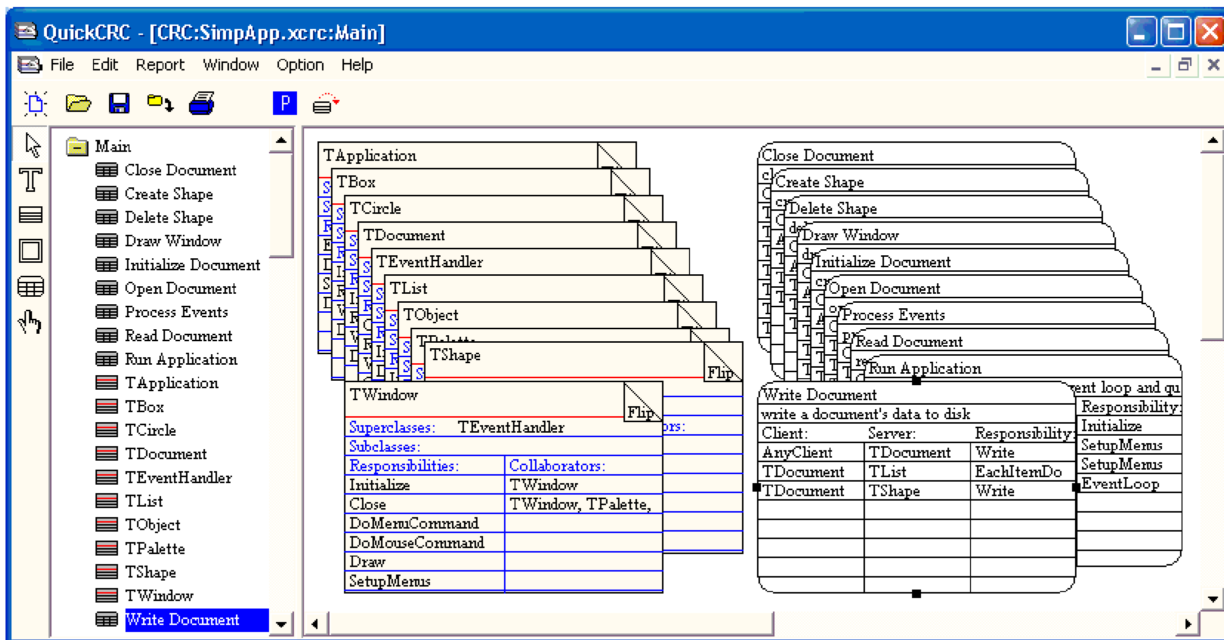


Figure 3.1: QuickCRC environement

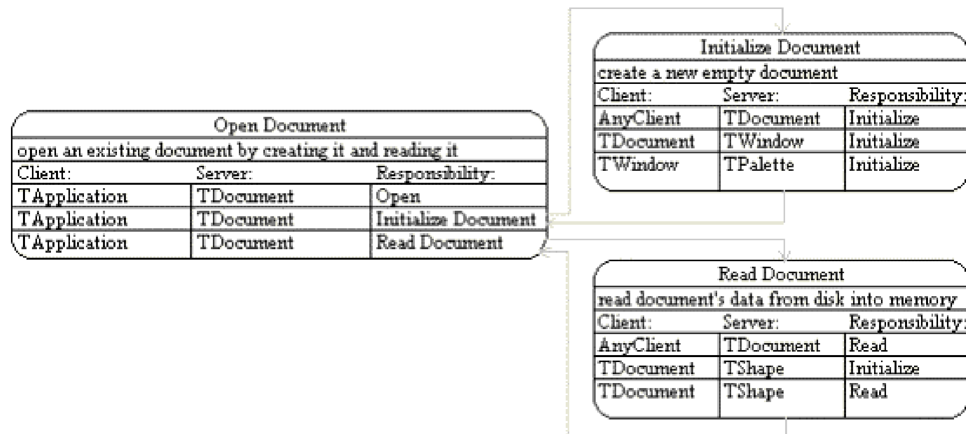


Figure 3.2: QuiCRC Path through sub-scenarios

into different functional areas and provide selectivity when listing specifications, printing cards or exporting information to other tools.

- **Assign attributes** The tool supports the addition of classes' attributes. Attributes can be nouns that are not classes but rather characteristics of classes.

- **On-Screen Editing** Editing changes like add, edit, move or delete information on cards can be made directly on-screen.

- **Supports scenarios** To discover the data each class knows about and the responsibilities it must perform, scenarios are created. A scenario is a list of steps outlining the interaction between groups of classes to implement a mechanism in the design. For each scenario step, a client class uses a responsibility of a server class. Scenarios can also refer to other sub-scenarios.

- **Simulate scenarios** Using QuickCRC, a designer can simulate a selected scenario. By single stepping forwards, backwards or through each sub-scenario, bugs in the design can be identified and corrected early. An active scenario list shows your position within a hierarchical stack of scenarios. This list helps you keep your bearings in a complex simulation and allows you to change to a different spot in the simulation path. Figure 3.2 shows path through subscenarios.

- **Partition the design** As the number of CRC cards in the design grows, they can be grouped by function. Separate diagrams are used to partition the model into subject areas. The Contents view is used to navigate or move cards between diagrams.

- **Inheritance graph** QuickCRC can generate inheritance graphs from information on CRC cards. These diagrams concisely illustrate the big picture of a large project that might contain thousands of classes and hundreds of diagrams.

- **Work verification** Creating and simulating scenarios will help verify that a design is correct and complete. QuickCRC can perform error checks to locate design problems. For example, responsibilities that are not used in any scenarios may indicate that the design is incomplete or perhaps the responsibility isn't needed. Likewise, a card that is not used by any collaboration may not be needed.

- **Import, Export & Error Checking** CRC models can be verified to locate errors. Information can be imported from or exported to other development tools. This enables generation of CRC cards from

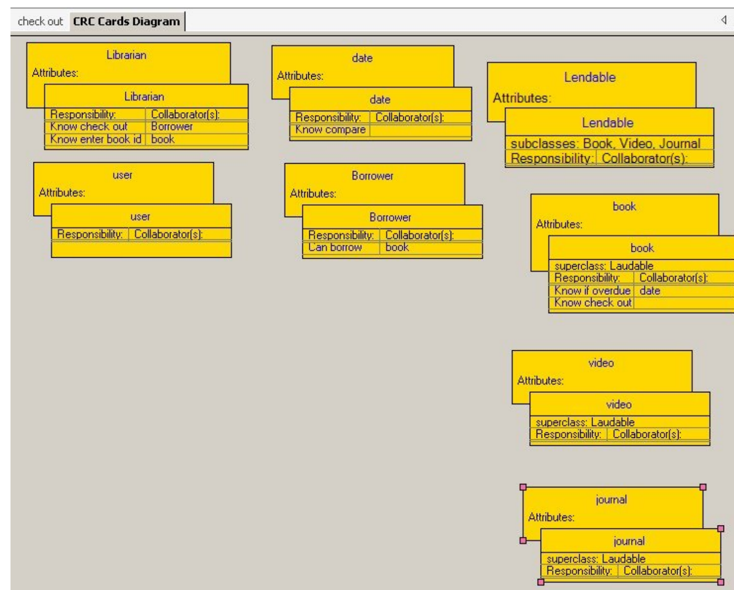


Figure 3.3: EasyCRC CRC card diagram

source code with WinTranslator or MacTranslator. Data from CRC cards can be exported to tools like MacA&D, WinA&D or QuickUML to auto-generate UML class diagrams.

3.1.3 EasyCRC

The EasyCRC tool is a freely available tool[3] and was created to help students in object oriented software engineering (OOSE) classes, by automating the process of defining classes, responsibilities and collaborations. It also supports scenarios modelling, by using UML sequence diagrams. The use of the tool is divided into two major stages: a) identifying the classes (CRC cards) from plain text and b) identifying the collaborators and responsibilities by simulating scenarios through sequence diagrams.

In more detail, the steps are the following:

- **Noun identification** The tool identifies the nouns in a requirements document, by referring to a dictionary either on the Internet or a locally installed.

- **Class identification** The user then can identify which from the discovered nouns are relevant and

[3]http://www.easycrc.com/

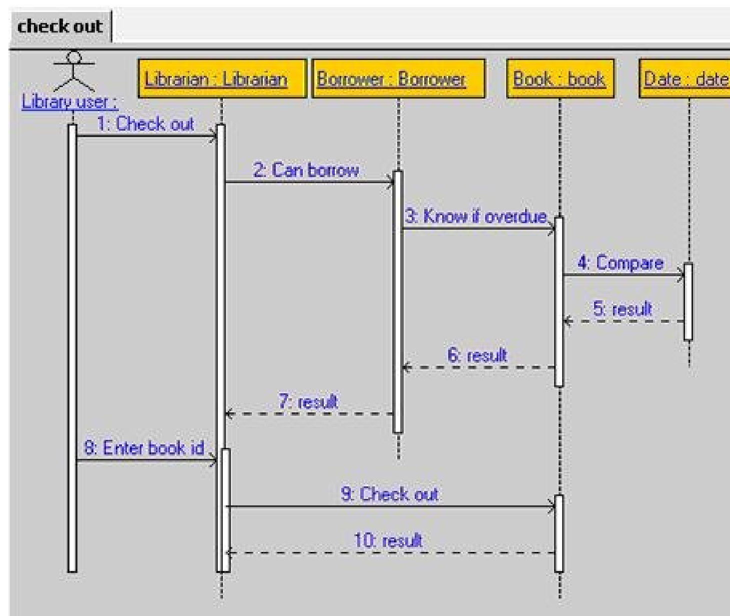


Figure 3.4: EasyCRC sequence diagrams

which are not.

- **Creation of classes** Classes are created using the relevant nouns. Figure 3.3 shows a diagram of CRC cards.

- **Build sequence diagrams** Sequence diagrams are created by adding objects needed for the scenario and then adding messages between the objects. For each message there is an option to add a responsibility to the source card. Figure 3.4 shows a sequence diagram.

EasyCRC can export the CRC design to XMI (XML Metadata Interchange), which is the common base for UML data export and can be imported by the Visual Paradigm. Because UML XMI format does not support the CRC card export, CRC cards are exported as classes. Also the design can be exported to HTML.

3.1.4 ECoDE

ECoDE[4] (Ectropic Collaborative Design Environment), is a development tool designed to capture two key components of Ectropic design[5]: Collaborations (CRC cards) and scenarios. ECoDE includes a graphical user interface targeting novice software designers and attempts to present an environment that couples the flexible and modular structure of well designed object oriented software and the perspicuity of functionally organized software.

ECode divides the object-oriented design process into three distinct phases:

- **Analysis Mode** The user in this mode creates CRC cards and scenarios for identifying the required responsibilities which are then assigned to the CRC cards. Scenarios are build using a pair of CRC card and responsibility.
- **Design Mode** The objective during this mode is to design an implementable and complete design. Each CRC card is reviewed and for each responsibility of the card there should be a corresponding method.
- **Program Mode** In this mode ECoDE checks all CRC cards to insure that all responsibilities have been assigned corresponding methods and converts cards to source code.

3.1.5 The CRC Design Assistant

The CRC Design Assistant was first conceived to assist students in creating real-world software applications and tries to be comparable to using real index cards. As the rest of the tools, '*The CRC Design Assistant*' also helps users to create CRC cards holding name, description, super class, subclass and responsibilities information. It also provides different views based on class, responsibilities and collaborations or CRC cards. Figure 3.5 shows the CRC cards view.

According to [55] some of its unique features are:

- **RTF Documentation** The tool provides the generation of automatic documentation for the application design in rich text format (RTF).

[4]http://www.cc.gatech.edu/ectropic/

[5]Ectropic design is a collaborative design method, by which order and structure are created out of the efforts of multiple, potentially unrelated, software developers

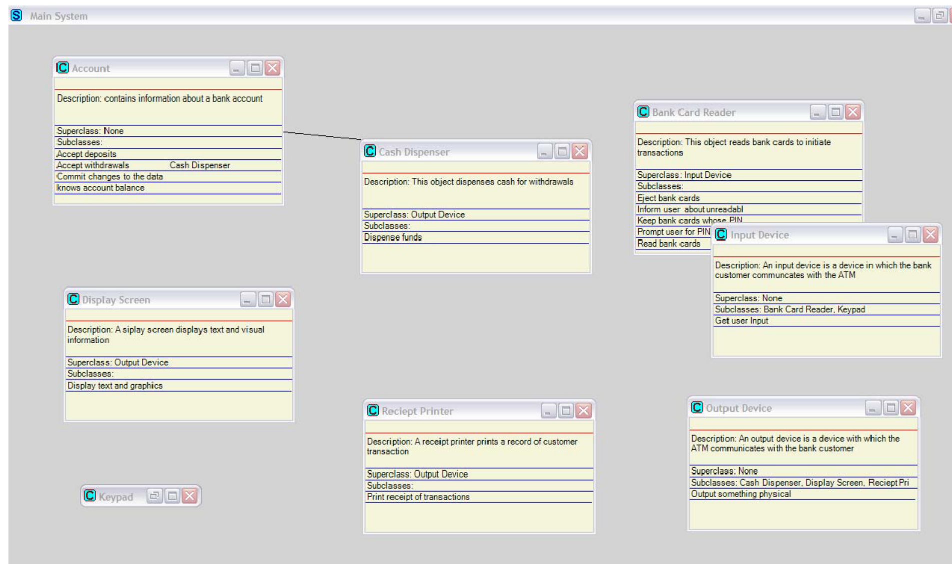


Figure 3.5: The CRC Design Assistant, CRC Card View

- **Visio support** It can also generate CSV files that can be opened using Microsoft Visio to create CRC diagrams and object diagrams.
- **Collaborations using lines** Lines are drawn to represent collaborations between classes. This makes it easier to visualize the extent of coupling in the system.
- **Support for subsystem abstraction** Groups of classes that, together perform a set of related tasks, can be clustered together into a subsystem. This helps the designer see the design at different levels of complexity.
- **Printing** It also allows printing of CRC cards when there is a need for using real index cards.

3.1.6 Merobase component finder

Merobase[6] is a software search engine that allows developers to find, share and reuse software components from the Internet. The engine harvests software components from a large variety of sources, including Apache, SourceForge, and Java.net.

[6]http://merobase.com/

A unique feature of merobase compared to other code search engines is its ability to support interface-driven searches, that is searches based on the abstract interface that a component offers rather than on the text in its source code. This allows merobase to support searches for binary components (e.g. Java bytecode, .NET assemblies) and web services, as well as source code, and significantly enhances its precision. In this context merobase let the user define a set of CRC cards and then searches its' database for components that match the given CRC cards.

3.2 What is missing

Most of the tools discussed earlier, try to simulate using real index cards. This means that they constrain the placement of CRC cards on a 2D plane, which is further restricted by the resolution of the used monitor or projector. Some of them offer ways to expand the desktop, either by using scrolling or pages. However they don't allow the user to see the whole CRC design, so that they can distinguish patterns and connections between the cards. Moreover this limited environment can strangle the effectiveness of the use of CRC cards because it can restrict the process of coming up with new ideas and discovering new connections/relationships while moving the cards around the workspace, just as if we were rearranging the letters of an anagram searching for a new word. Finally, it is difficult for the user to arrange the cards in such a placement that he can remember connections between the cards and how each card interlocks with each other.

Another important point is that all the previous tools use the class name to identify cards, which can be very limiting, especially if we have complex crc designs. In such cases, if the users want to see the whole design, they are forced to zoom out to the point where names of the cards become unreadable. Consequently another option for identifying cards should be introduced. Instead of supporting cards of only one colour, multi-coloured cards can be used, providing the user with an extra cue to identify a class, visible even when the card has a very small size. Moreover, this approach can help us to group cards in visible partitions. This partitioning can be done by using the same colour or small variations of it, for all cards of a group. Another important cue which can be used for identifying cards, is the existence of multi sized cards. For example the user may define the size of the cards, instead of using fixed sized cards. Then, the user can make important cards bigger and thus more easily visible from long distances. Besides this, in the case of 3D cards in a 3D space, cards can also be made thicker.

Furthermore, when a complex system is analyzed, it becomes increasingly difficult to track all data gathered and their context. Existing tools mainly enable drawing the CRC cards, but do not graphically highlight relationships between cards, a feature that could aid the process of finding responsibilities and collaborations. In fact, most of the students in object oriented software engineering university lessons avoid using the aforementioned tools and rather use drawing tools like Visio [53]. Therefore, an important feature would be to draw the relationships between cards, using lines. The direction of the collaboration, which could also be bi-directional, may also be illustrated using arrows. Furthermore, the type of the collaboration, e.g. ISA[7], could also be drawn. Such visualizations can show the extent of coupling in the design. Moreover this rendering of a CRC system enables the user to run any possible scenarios by following cards and links, discovering any missing cards or collaborations.

A common problem is that many novices do not fully comprehend the context of responsibilities and collaborations. It seems that despite the underlying ideas of CRC cards, students continue to think in terms of data responsibilities rather than in terms of service responsibilities of the class [53]. The situation is further aggravated by including in the CRC card attributes of the class, which forces users to think of the data each card holds. In addition, inclusion of the attributes takes up important space in the already limited space of a card and makes cards look cluttered. This information could be inserted at a later stage when a formal methodology will be introduced.

3.3 Need for a new tool

In this context we propose a CRC tool that supports the visualization of CRC cards and their links in 3D, the design of which takes into consideration all the ideas previously discussed. If appropriate stereoscopic display devices and technologies are provided, the tool should also provide an immersive experience to the user, thus increasing task completion performance [59] and effectiveness of spatial memory. Similar systems for 3D visualization of molecular structures and their related properties have been used for many years and found to be important for understanding the dynamics of molecules, chemical reactions and the expression of macroscopic behaviours determined by microscopic properties [13].

[7]ISA or 'is a' refer to a special type relationship which can exist between objects. This relation exists when a class implements a common interface or extends an abstract class, or during inheritance.

Chapter 4

Requirements analysis and design

Software development is a very complicated process for which many models supporting it have been proposed. One of the oldest and best known is the waterfall model. The waterfall model is a sequential software development model in which development is seen as flowing steadily downwards, like a waterfall, through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance [65]. This is the model that was used to develop our CRC tool.

In this chapter we will elaborate for the phases before implementation. That means that we will introduce the requirements of the tool, its architecture and software design, along with any decisions made concerning the user interface of the application.

4.1 Requirements analysis

Requirements analysis encompasses those tasks that go into determining the needs or conditions to meet for a new or altered product. During this process the requirements of the various stakeholders should be taken into account, which sometimes might be contradictory. Requirements analysis is the first step in the development of a software system and critical to its success.

During this phase, firstly we had to communicate with users of other related systems, in order to elicitate their requirements. Afterwards, the gathered requirements had to be analyzed, to determine whether the

stated requirements were unclear, incomplete, ambiguous, or contradictory and then resolve these issues. Finally the requirements where documented in a natural-language document, which was the basis in the development of the tool.

The basic directions for the tool were determined to be the following:

- **3D environment** Take advantage of current 3D technology and create a 3D interface for the visualization of the CRC cards and their links, representing collaborations between cards.
- **Immersive** Since we will have a 3D environment, reinforce the effectiveness of spatial memory and task completion performance by supporting stereoscopic displays, providing an immersive experience to the user.
- **Multiplatform** Make the tool available for most platforms the user might prefer, like Windows and Linux environments.
- **Efficient use of resources** Make the application lightweight enough, so that it can be used in everyday desktop computers and laptops.

4.1.1 Functional specifications

Functional specifications are the blueprint for how an application will look and work. It details what the finished product will do, how a user will interact with it, and what it will look like. By creating a blueprint of the product first, time and productivity are saved during the development stage because the programmers can program instead of also working out the logic of the user-experience.

For our tool functional specifications have been divided in the following categories:

Navigation

As already discussed, navigation is a very important aspect in a 3D environment and can sometimes be cumbersome and problematic. Since we want our tool to be usable in everyday computer systems, navigation is provided through a simple mouse. By pressing mouse buttons and moving the cursor around the user is be able to move the camera in the 3D space changing at the same time the view of the world. By freely manipulating the camera the user should be able to visit any place of the 3D world.

More specifically the following functionality is available:

- **Zoom in / zoom out** This could be done by pressing a mouse button (LMB [1]) and moving the mouse cursor up or down.
- **Rotate freely** This could be done by pressing a mouse button (RMB) and moving the mouse cursor.
- **Pan across the display plane** This could be done by pressing a mouse button (MMB) and moving the mouse cursor.
- **Shortcuts** In order to avoid navigation problems appropriate shortcuts are provided, giving direct access to views of either the whole world or of a selected item only, either a card or a link.
- **Rendering of X, Y, Z planes and axes at the centre of the world** These could work as landmarks, aiding navigation, orientation and way finding. Moreover they provide cues for the placement of the cards in the 3D space.
- **Stable views** Option to enable stable views of the world, by drawing the front, top and side views along with the current view, again for aiding navigation, orientation and way finding.
- **Anti-aliasing** This effect will show the world cleaner and lines prettier, allowing easier navigation and avoiding disorientation

Workspace

A workspace is a file that allows a user to gather various crc designs defined as projects and work with them as a cohesive unit. Workspaces are very helpful in cases of complex projects when maintenance can be challenging.

For our tool, the user is able to:

- **Actions through an always accessible workspace menu** Initially the user can only create or open a workspace.
- **Create workspace** Before finishing this action the user gives a name for the created workspace. If another workspace is already opened, close it. It also activates all other initially deactivated actions in the workspace menu.

[1] LMB: left mouse button, RMB: right mouse button, MMB: middle mouse button

- **Open workspace** User selects which one to open from a list of available workspace files. If another workspace is already opened, close it. It also activates all other initially deactivated actions in the workspace menu.
- **Save workspace** If a filename is not already specified, specify one before saving.
- **'Save As' workspace** The user specifies the filename in which the active workspace will be saved. If a workspace file is overwritten the user is informed.
- **Close workspace** This also closes any open projects, returning to the initial screen of the tool. The workspace menu returns in its initial status.
- **Inform for modifications** Whenever the workspace is changed the user is notified.

Any critical operations are implemented in a safe way, so that they don't get executed by accident.

Project

A project holds a crc card design and is included in a workspace file. It holds information about all the properties of the cards and links.

As long as the user has a workspace opened, he is able to do the following:

- **Actions through an always accessible project menu** Initially the user can only create or open a project.
- **Create project** Before finishing this action the user gives a name for the created project. If another project is already opened, close it. It also activates all other initially deactivated actions in the project menu and the button for creating new cards. The created project is inserted in the active workspace.
- **Open project** User selects which one to open from a list of available projects in the active workspace. If another project is already opened, close it. It also activates all other initially deactivated actions in the project menu.
- **Save project** The active project is saved in the active workspace.
- **Rename project** The active project is renamed to the specified name.

- **Delete project** Deletes active project from active workspace.
- **Close project** Closes active project from active workspace. Clears the CRC world, returns project menu to its initial status and deactivates new crc card button.
- **Inform for modifications** Whenever the project is changed the user is notified.

Again any critical operations are implemented in a safe way, so that they don't get executed by accident.

CRC cards

CRC cards are the basic part of a crc design. A CRC card holds editable information about the name, responsibilities, comment and meta. This information can be edited through textboxes. Buttons are used for any other card specific operation. Collaborators are created automatically when links are created. Moreover the user can select a CRC card by just pointing it with the mouse cursor and pressing a button (RMB). If pointed location is an editable area of the card, the user can edit the corresponding information.

More specifically cards provide the following functionality:

- **Create CRC cards** Creating duplicated classes is avoided. This action is offered through a button in the same place where the workspace and project menus exists.
- **Delete CRC cards** Cards can be safely deleted from project, by pressing appropriate button.
- **Change information of CRC cards** Fast, easy and safe ways to change the fields of a card, including class name, responsibilities, meta and comment. These can be done by selecting appropriate textboxes and inserting desired information.
- **Change properties of CRC cards** The user can change the properties of a CRC card including its position, rotation, size and colour, through appropriate settings window and textboxes.
- **Collaborators update** Changing the name of a class also updates appropriate collaborators.
- **3D Manipulators** Moreover 3D manipulators are available to the user in order to change the position, orientation and size of the card. Appropriate feedback is provided when using the manipulators, about the card position and size values in the X, Y and Z axes along with roll, pitch and yaw values[2].

[2]Roll, pitch and yaw refer to rotations about the X,Y and Z axes

- **Visible editing cards** Editing cards, whose properties are changed by the user, are visible even if the user zooms out from the specific card.

Links

Links are the other important part of a crc design. They are simple lines, connecting two collaborating classes. A pyramid is drawn showing the direction of the collaboration and is coloured with the colour of the card asking for help. If this is a bi-directional collaboration, two pyramids are drawn. If a name for this collaboration exists, it is drawn over the corresponding pyramid.

More specifically links provide the following functionality:

- **Create Links** A user can create a link by first putting the card needing collaboration in editing mode and then pointing the collaborating card and pressing Ctrl and LMB.

- **Delete Links** A user can delete a link by first putting the card from which we want to remove collaboration and then pointing the collaborating card and pressing Shift and LMB.

- **Change information of links** The user can insert or update the type of collaboration by just pressing the corresponding pyramid of the link. Collaborations on the cards then are automatically updated.

- **Visible editing links** Editing links, whose type of collaboration is changed, are visible even if the user zooms out from the specific link.

Workspace file

As already mentioned, this CRC card 3D tool will be able to parse and load a workspace from a file. This file will contain all the appropriate information of a workspace, including it's constituting projects and cards and will have 'crc' as filename extension. Furthermore it is a plain text UTF-8 file, allowing users to edit it not only through the tool but also by simply using their favourite text editor. This way it will be very easy for them to add projects and cards from other workspaces. Also they will be able to continue their CRC card design even if they do not have the tool at hand at a specific moment. An example of a workspace file is provided at Appendix A, page 97. This workspace file describes the CRC design of the developed tool.

Documentation generation

The tool will also be able to automatically generate a documentation file, containing the description of a specific workspace. All these files will be plain UTF-8 text files and will have 'adg' as filename extension. At the beginning of this file the total number of projects contained in the workspace and the specific name of each project will be provided. Afterwards, more detailed information about each project will be presented. These include an overview of all classes and links composing the project and a comprehensive description of each class's properties. These properties will consist of the class name, information about its responsibilities, meta and comment fields along with any links connecting the aforementioned class with any collaborating classes. For the links the two collaborators are shown along with the type of their relation if it exists. An example of generated documentation is provided at Appendix B, page 109. This file documents the CRC design of the developed tool.

4.2 Architecture

Object-oriented design and programming encourage reuse by inheritance and polymorphism, but still the reusability can only be reached by a careful design [41]. Nowadays, as the sizes of software products grow larger and larger, the architecture, design and implementation have to be well organized, clear and maintainable. On the organization level this means that all developers should use similar working processes in different projects. When people switch from one project to another they should already know the basic principles and be able to get productive faster. This is an important factor as the goal is to reduce the time needed for developing a system. These needs are the motivation for creating predefined software architectures, design methods and conventions. Model-View-Controller is one example of such application architecture, on a variation of which, called MVC++, our tool was based on.

4.2.1 Model-View-Controller (MVC)

The idea of the MVC architecture is to divide any application into manageable parts, which have their own responsibilities. MVC architecture was originally designed for Smalltalk-80 applications in Xerox Palo Alto Research Center [43]. Shortly the different layers are the following:

- The *model* layer defines the domain specific representation of the information on which the application

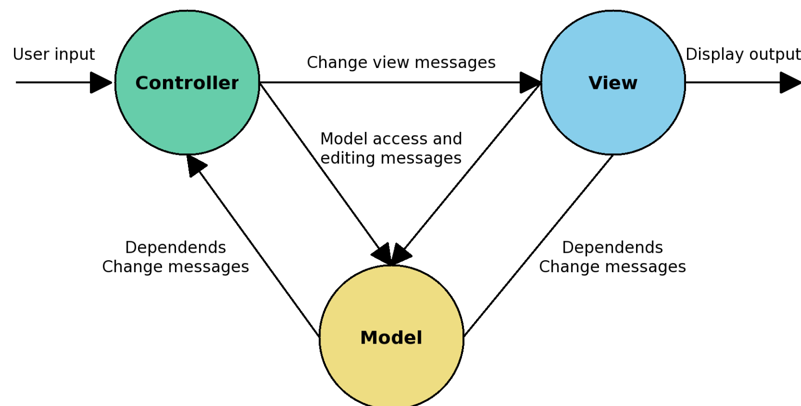


Figure 4.1: Model-View-Controller Architecture

operates. Domain logic adds meaning to raw data. Many applications use a persistent storage mechanism (such as a database) to store data. MVC does not specifically mention the data access layer because it is encapsulated by the model.

- The *view* layer which renders the model into a form suitable for interaction, which is typically a user interface element.

- The *controller* layer processes and responds to events, typically user actions and may invoke changes on both the model and the view.

Views and controllers of a model are registered in a list as dependents of the model, to be informed whenever some aspect of the model is changed. When a model has changed, a message is broadcast to notify all of its dependents about the change. The view or controller responds to the appropriate model changes in the appropriate manner.

The standard interaction cycle in the Model-View-Controller metaphor, is that the user takes some input action and the controller notifies the model to change itself accordingly. The model carries out the prescribed operations, possibly changing its state, and broadcasts to its dependents (view and controller) that it has changed, possibly telling them the nature of the change. The view can then inquire the model about its new state, and update their display if necessary. Controller may change their method of interaction depending on the new state of the model [44]. This message-sending is shown in Figure 4.1.

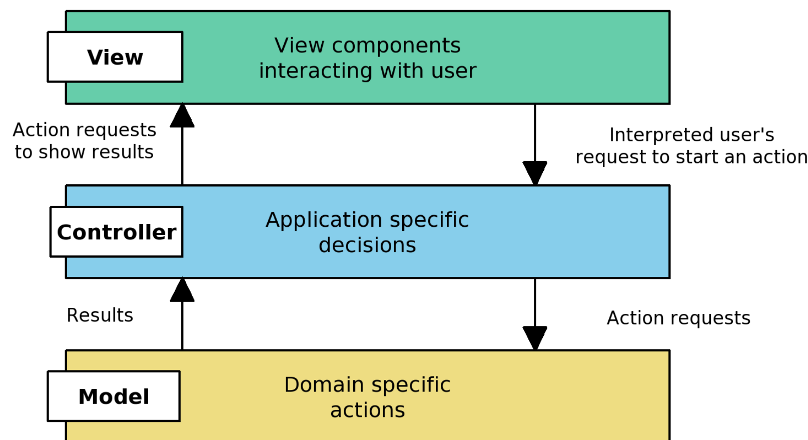


Figure 4.2: Parts of an MVC++ application

4.2.2 MVC++

MVC architecture defines the structure and abstract base of classes for Smalltalk-80. As similar guidelines for designing and implementing X/Motif software using C++ did not exist, at Nokia Telecommunications the MVC architecture was adapted to suit development in C++. This modified MVC has been named MVC++ [39]. The MVC++ has the same three functional layers as the original MVC, but their interaction is a bit different. In MVC++ there is no direct connection between the view and the model. The user interacts with views and they pass the information to their controllers. At this point the controller makes the application specific decisions and operates according to them. In the original MVC architecture the view does not receive user input as the controller makes the decisions and delegates the needed actions to view and model layers [43]. Shortly, the tasks of the different layers in Figure 4.2 are the following [40]:

- The *model* layer defines the classes that represent the concepts of the problem domain.
- The *view* layer forms the user interface.
- The *controller* layer is the glue between model and view layers. It handles the interaction between model and view and therefore contains a lot of the logic of the application.

In more detail, the model part contains all application domain specific data and knows how to manipulate it. The model does not have any kind of user interface and it does not define any application specific logic.

The model should be able to perform the data related tasks independently, without knowing anything about the controller and the view layers. On many occasions the model can be completely separate of the rest of the application as it may have only a static role of doing the operations when requested. As an exception some models may be acting independently. For example, they may perform real-time monitoring or receive events from an external source. In such cases the model layer is able to invoke operations by calling the controller layer without any user interaction [39].

The view shows the state of the model to the user, displays the user interface, and manages all user interactions. The view can also contain reusable view components that may have a simple view-only implementation or they can also define their own internal MVC++ structure [39]. The view component can have methods for manipulation, feedback and querying the state of the view.

The controller manages the interaction between the model and the view parts. Whereas model defined the logic of the real world, it is a job of the controller to define the actual application logic. The controller knows the tasks that the model and view can perform, and delegates the application tasks to them. The controller does not need to have detailed knowledge of how the model and the view actually handle the delegated operations [39].

4.3 Detailed description of components

As previously stated, our tool was based on the MVC++ architecture. From the beginning of the design phase, the CRC card technique was used to facilitate the process. Unfortunately all other available computerized tools were not satisfactory for our needs, as previously analyzed, so we decided to use the CRC card technique manually. After developing the tool, the initial manual design of Flying Circus was transfered to the system itself. The following screenshots of the basic components of the tool are taken from the Flying Circus.

As a consequence of the MVC++ architecture, the basic components of our tool are the model, the controller and the viewer. The viewer in our case knows how to render a crc design and gets the user input. When an action should take place, it communicates with the controller which has all the application logic. If the controller needs to get or update any domain information, it communicates with the model. Else it updates the viewer if needed. As you can see, there is no communication between the model and the viewer.

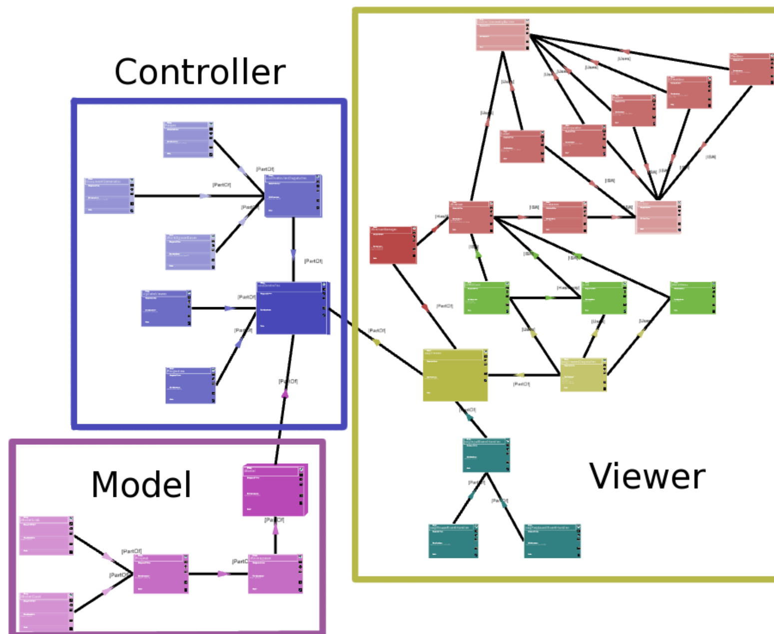


Figure 4.3: The software design of our tool. Rendered by Flying Circus

Figure 4.3 show the components of the tool and how they communicate using CRC cards in Flying Circus[3]. A detailed description of each component is given in the following subsections.

4.3.1 Model

The model layer defines the domain specific representation of the information on which the application operates. In our case the model knows how to handle workspaces. A workspace is part of the model and can hold many projects. Moreover a workspace knows which the current active project is, and provides methods to add or remove projects along with methods for changing its attributes. Each project has many crc cards and links between those cards. It also provides methods to change its attributes and for the addition or removal of cards and links. The project is also responsible for keeping the consistency of the design. If a class is renamed or a label of a link changed, it is responsible for updating appropriate information in cards and links. Finishing, the card and link classes hold all appropriate information for a CRC card or link and provide methods to change their attributes. Figure 4.4 shows the model component using CRC cards.

[3]The original colours of the Flying Circus screenshots, have been inverted, because the background is black

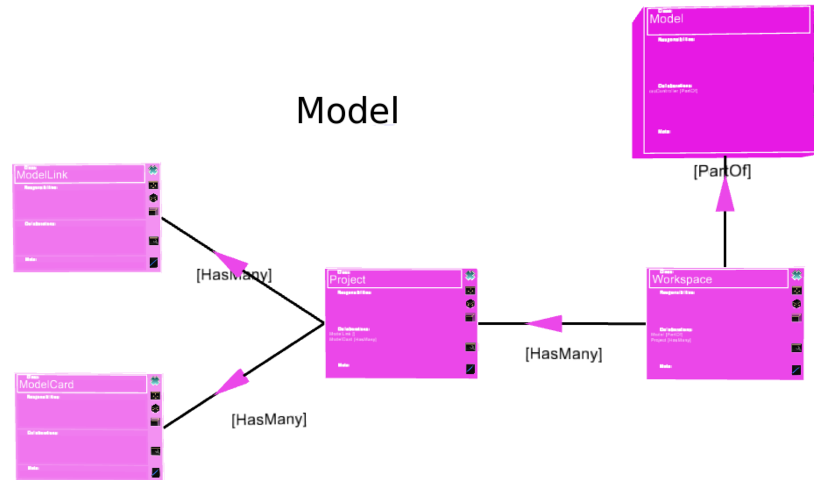


Figure 4.4: Design of model component. Rendered by Flying Circus

4.3.2 Controller

The controller layer is the glue between the model and viewer layers and contains a lot of the logic of the application. This layer is realized by the crcController class. This class receives signals from the viewer and by using a dispatcher knows how to analyze them and take care of any actions. These actions can include updating the model component, which is part of the controller or updating the viewer component, which can be done using the UpdateViewer class, which is also part of the controller. If the dispatcher needs to parse a workspace file, it uses the parser class and the controller updates the model. If it needs to generate documentation for the active workspace it uses the DocumentGenerator. Lastly if it needs to save a workspace, it uses the WorkspaceSaver class. The properties of the application, such as paths for designs, logs and documentation can be found in the Properties class, which is part of the controller. Figure 4.5 shows the controller component using CRC cards.

4.3.3 Viewer

The viewer forms the User Interface of the application. It gets the input from the user and draws the output of the application to him. The viewer is realized by the osgViewer class that renders CRC cards, links and other graphical elements the user can interact with.

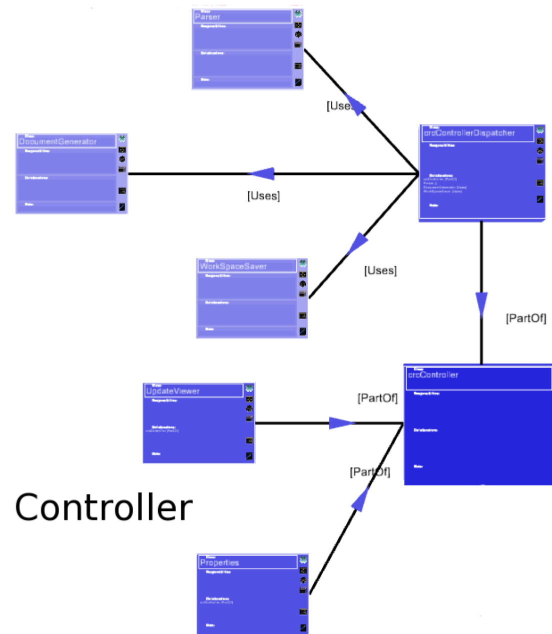


Figure 4.5: Design of controller component. Rendered by Flying Circus

The viewer can be divided into two parts. The first part is the InputEventHandler which is responsible for informing the osgViewer for any input events, either by the keyboard or the mouse. Events from keyboard or mouse are caught by the appropriate mouse or keyboard event handlers which are part of the input event handler.

The second part is responsible for the rendering of the scene. The base class of the scene is the WindowManager, which knows how to manage the rendered windows. Everything in the 3D world is represented using windows. CRC cards, links, the main menu and any other window the user can interact with are subclasses of the base Window class. A Window class is a container of Controls. Controls are the 3D widgets or in other words the interface elements that the computer user interacts with. Moreover each Control knows how to render itself by using the helper class ControlGeometryBuilder. Textbox, Button, Label, Manipulator, Tab and other control classes are sub classes of the basic Control class.

When an action takes place in a Control, a signal is send to the Window. The Window then informs the WindowManager and the WindowManager the osgViewer. Afterwards the signal is analyzed in the osgViewer using a dispatcher, the osgViewerDispatcher. If the signal relates to the state of the viewer, the

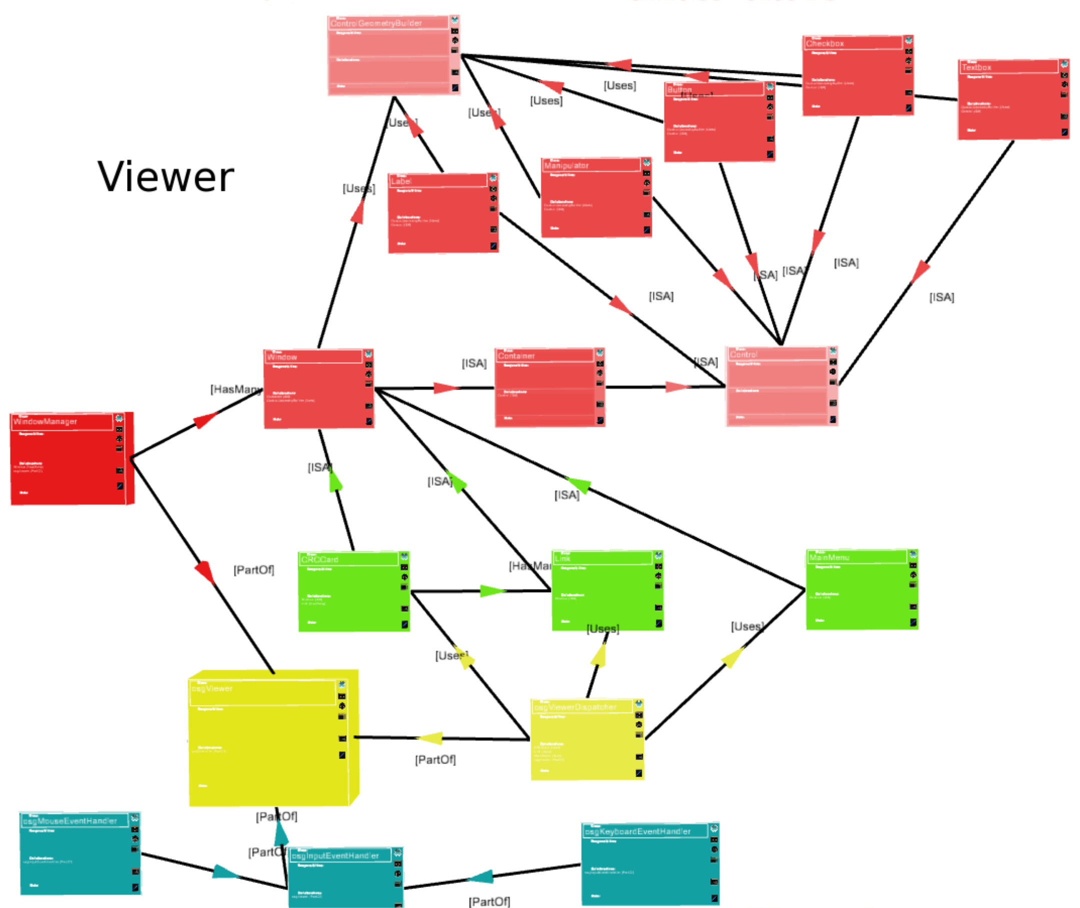


Figure 4.6: Design of the viewer component. Rendered by Flying Circus

dispatcher takes the appropriate actions. Else the osgViewer signals the controller. The dispatcher uses the CRCCard, Link, MainMenu and other Window subclasses in order to make the appropriate actions. Moreover the dispatcher, analyzes signals send by the controller to the viewer and takes care of appropriate actions.

Figure 4.6 shows the viewer component using CRC cards. The yellow cards are the osgViewer and osgViewerDispatcher classes, the turquoise are for handling input, the red are related to the windows framework and the green are the window classes like CRCCard and Link used in our tool.

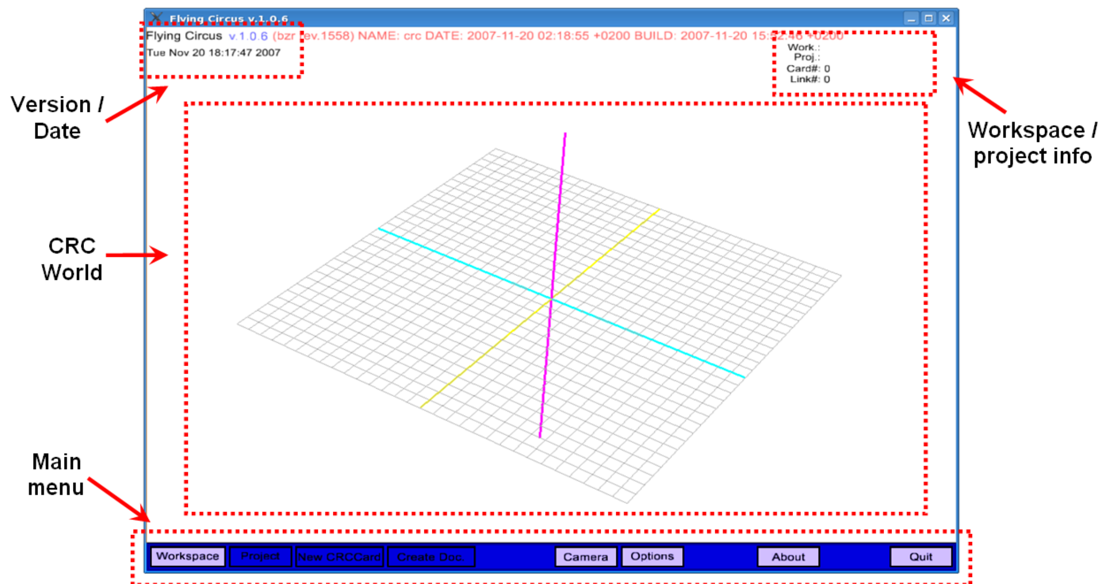


Figure 4.7: The initial screen of Flying Circus

4.4 User interface design

The user interface is the aggregate of means by which people interact with a particular system. The user interface provides means of giving input, allowing the users to manipulate a system, and output allowing the system produce the effects of the user's manipulation. The design of a user interface affects the amount of effort the user must expend to provide input for the system and to interpret the output of the system, and how much effort it takes to learn how to do this. A well designed UI can support the efficiency and effectiveness of a system along with user satisfaction and experience.

Flying Circus was based on a 3D Graphical User Interface, accepting input via a simple computer keyboard and mouse. The design of a 3D user interface is as already explained a challenging process. Our design was based on expanding 2D well known UI metaphors from everyday desktop computers to 3D. So the manipulation of the system is done through 3D controls, extending 2D ones. Therefore these controls can exist everywhere in the 3D space and except from height and width also have thickness. For some specific actions, when 2D controls didn't exist, the previous controls were supplemented with intuitive 3D metaphors. An example of such metaphors is manipulators.

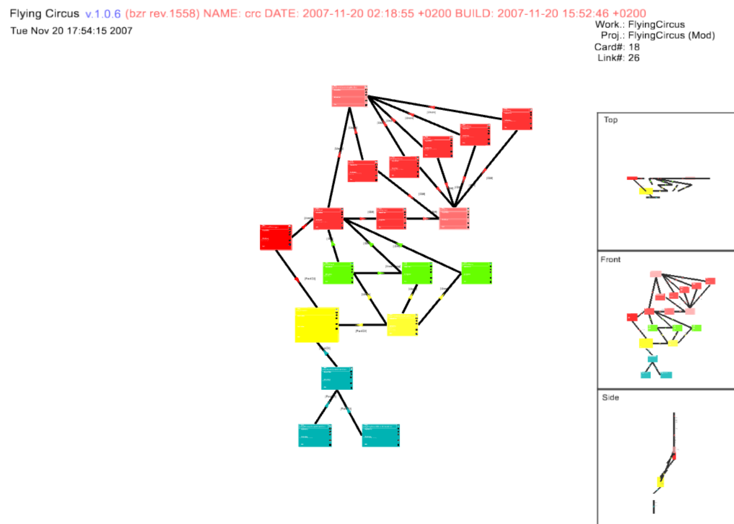


Figure 4.8: Top, Front and Side stable views of the world

4.4.1 Layout

Figure 4.7 shows the initial screen of the tool. There are two parts of the UI, the 3D CRC world, which the user can freely manipulate, by zooming in/out, rotating or panning and the other Head-Up-Display (HUD) elements, which are again composed by 3D widgets but are projected on the monitor orthographically and have a fixed position.

The CRC world is the place where all the CRC cards and their collaborations, links exist. The cards can be placed anywhere in the world the user prefers and links are drawn automatically based on the properties of the two collaborated cards. Optionally grids and axes can also be rendered. By manipulating the camera, using the mouse, the user can visit every single corner of it. Moreover keyboard shortcuts are available so that the user can be transferred quickly to specific areas of the 3D space, like in front of cards or links.

The HUD component contains the main menu bar at the bottom of the screen and the stable camera views, when activated, at the right side. The stable camera views for a CRC design can be seen in figure 4.8. Moreover it contains the labels with the version of the tool at the upper left corner and the labels with information about the active workspace id, active project id and the number of created cards and links at the upper right corner. The information labels also show the position, rotation and scale values of a CRC card when in editing mode. The HUD also holds any windows that are not part of the 3D CRC design. These

can be windows for changing the properties of a card, the window for the options of the tool and any other window displayed, when the system waits for a user's action or wants just to inform him. These windows are displayed at the centre of the screen.

4.4.2 Controls

As previously stated, the manipulation of the system is done through 3D controls. These are controls that look like exactly as their 2D relatives if rendered in orthographic projection and using a camera which is exactly in front of them. Moreover a user or programmer can position this controls anywhere in the 3D space, instead of just positioning them on the monitor plane. Furthermore these controls can have thickness. That means that they can expand in the z axis, making them visible even if the user was looking them from along the y axis. On the other hand we designed the manipulator controls, which do not have 2D counterparts, based on the manipulators used in many 3D modelling packages and have similarities with the 'Virtual sphere' [37]. Controls were designed to use simple primitives and a small number of vertices, so that the system could be run in everyday desktop computers with a current low level graphics card.

Labels

Labels are pretty simple controls. They are used just to inform the user and do not allow any interaction. Labels just hold a string which can be rendered in front of a coloured background. Sometimes it is useful in 3D environments to create labels that always face the user. This can be done using billboards[4]. In our application all the non interactive strings on the windows are created using labels.

Buttons

Buttons are one of the most important controls. The user can interact with them, by pointing them and clicking the LMB. Then an action takes place. Buttons can either have a label or an icon, identifying the action of the button. If the mouse pointer intersects with the button, a simple animation which gradually scales and shrinks its size is drawn and the button is highlighted, showing that the button can be 'clicked' by the user. Buttons can be either activated or deactivated. Deactivated buttons have darker colours and more transparency than activated ones and do not have animation when pointed with mouse cursor. Also buttons

[4]Orienting a polygon based on the view direction is called billboarding [2]

can be pushed or un-pushed, with the second ones have lighter colours. Buttons are the basic controls for interacting with Flying Circus.

Checkboxes

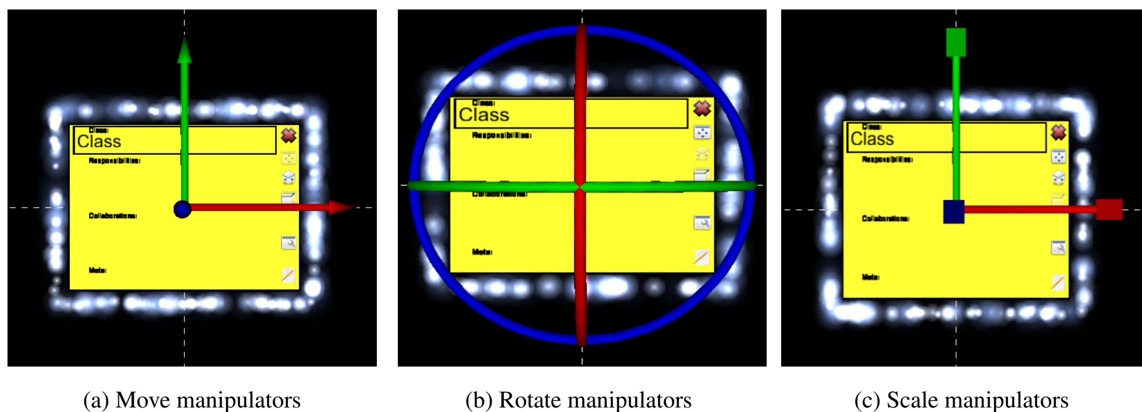
Checkboxes are very simple controls that either activate or deactivate an option. They are renderer by a simple rectangle and a label next to it, describing what option this checkbox enables/disables. By pointing the rectangle with the mouse and clicking LMB, the user can activate or deactivate respectively the option. If the option is enabled, a simple star is drawn in the rectangle. This control is used in the Options window, responsible for the options of the Flying Circus application.

Textboxes

Textboxes are the second most important control with which the users can interact. They allow the input of text in the application. They are rendered by a simple rectangle holding the text. Optionally a label is rendered above the textbox. By pointing the mouse to the rectangle and clicking LMB, we can set the textbox in editing mode and by pressing keyboard keys, insert text. A cursor is also visible in editing mode and cursor keys can move it across the text. When the text inserted by the user is bigger than the size of it, the textbox should divide the text in pages and using appropriate buttons, give access to every page, rendering only one each time. Textboxes can optionally be disabled, disallowing the input of text. Also they can optionally be configured to act as billboards. They are the basic blocks of the crc cards, links and any other windows that need text input from the user.

Tabs

Tabs are used when we want to have multi paged windows. They are containers of controls which also have a button on the top. If this button is activated or clicked by the user, then the controls of the page are rendered. Else they are hidden. Tabs are used in the Settings window, changing the properties of a CRC card. Position, rotation, size and colour all have their own page, which the user can activate by clicking appropriate buttons.



(a) Move manipulators (b) Rotate manipulators (c) Scale manipulators

Figure 4.9: Manipulators on a CRC card: (a) Move, (b) Rotate (c) and Resize

Manipulators

Because a mouse is used for 2D input and there is no 3D input device, free moving, rotating or scaling a window is not very easy. That's why we use 3D manipulators for moving, rotating and scaling a window. These manipulators allow constrained move, rotation or scaling on an axis. When the mouse pointer intersects with a manipulator, the manipulator is animated to show that the user can interact with it. By dragging move, rotation or resize handles, the user can move, rotate or resize accordingly a window, about the selected axis. By moving the mouse right the user can increase the value of position, rotation or resizing on this axis and the opposite can be done by moving the mouse to the left. When manipulators are enabled, appropriate info with the position, rotation or scaling of a window is shown at the upper right corner of Flying Circus. Moreover, when manipulators are enabled, stippled lines extending far away help the user placing the card correctly. Figure 4.9 show these three types of manipulators applied on CRC cards.

4.4.3 Windows

In this subsection we will describe the most basic windows in our application.

Main menu

All the user actions concerning the state of the active workspace and the options of the application can be done through the main menu. Figure 4.10 shows the main menu and the workspace, project and camera

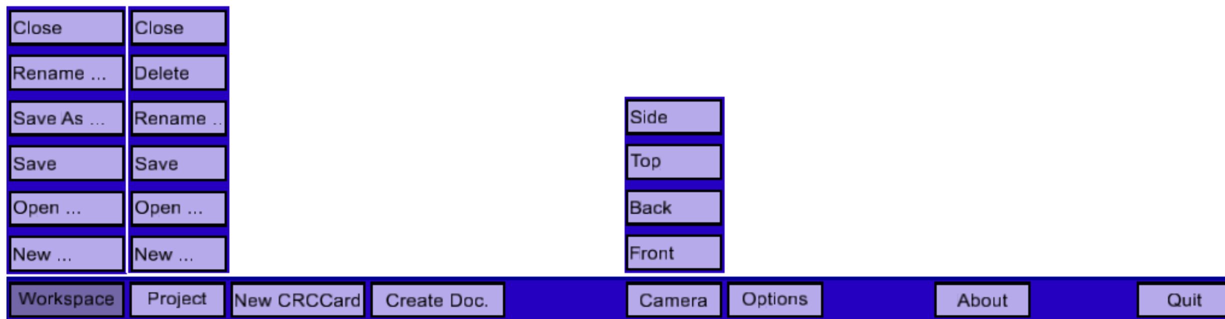


Figure 4.10: Main menu with workspace menu opened

menus open. As you can see the main menu is consisted of a 'Workspace' button, giving access to the workspace menu, a 'Project' button, giving access to the project menu, a 'New CRC Card' button, creating a new crc card, a 'Create Doc.' button, creating a documentation file, a 'Camera' button, giving access to the camera menu, a 'Options' button, opening the options window and the 'About' and 'Quit' buttons. Initially only the 'Workspace', 'Camera', 'Options', 'About' and 'Quit' buttons are active. The workspace menu is consisted of buttons for 'New', 'Open', 'Save', 'Save As', 'Rename' and 'Close' actions. Before opening or creating a workspace, only the new and open buttons are active. The project menu is consisted of buttons for 'New', 'Open', 'Save', 'Rename', 'Delete' and 'Close' actions. Before opening or creating a project, again only the new and open buttons are active. Finally the camera button provide access to predefined camera positions, which show the whole design from the front, back, top or side.

Card

The basic block of our application and of a CRC design is the CRC card. This card consists of textboxes, holding information about the class name, the responsibilities, the collaborations and the meta of the card. These exist at the body of the card. At the right of the card, there are six buttons. The first one closes the card when pressed. The second, third and fourth ones, activate the 3D manipulators. The fifth opens a window which allows changing the properties of the card, such as the position, rotation, size and colour. Finally the last button gives access to a window where a user can insert any comment for the specific card. At the back of the card, the name of the class is written with big fonts, so that it can be noticeable from far away. The collaborations textbox is inactive, since the application can automatically fill appropriate

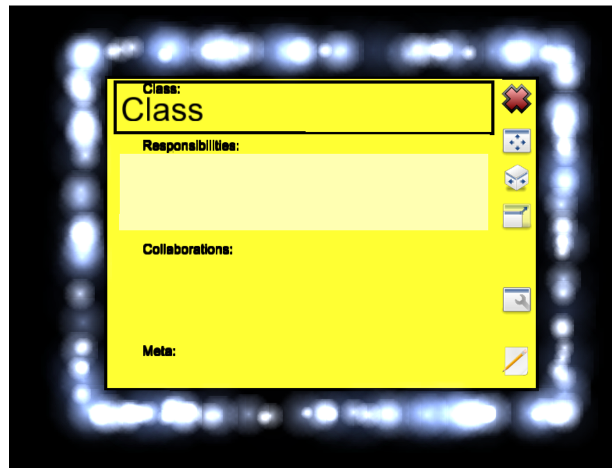


Figure 4.11: A CRC card. The user is editing responsibilities textbox

collaborators. Moreover links can begin or point to only four points of the card, which are located at the right, left, bottom and up side of the card. When the card is in editing mode, that is when editing a textbox or a button of the card is pressed, particles are drawn around the card, so that the user can distinguish editing cards. Particles are also drawn when the user simply clicks in any other, non interactive part of the card, meaning that the card is selected. Figure 4.11 shows a Card where the user is editing responsibilities.

Link

Links are the other basic part of a CRC design. They connect two collaborating CRC cards. They are drawn by a simple line, which starts from the card that needs help and ends to the other card. The length of the link is the least possible from the four points where links can connect as described in the previous section. A pyramid drawn almost at the centre of the link, shows which card helps the other and a label above it show the collaboration. The label is a textbox which can be edited by the user. The colour of the pyramid is the same as the colour of the card who needs help. If the collaboration is bidirectional two opposite pyramids are drawn. Figure 4.12 shows a bidirectional editing link.

Other windows

Other important windows, are the 'Open' window, the 'Options' window and the 'Settings' window. The 'Open Project/Workspace' window is responsible for opening projects or workspaces and uses buttons to

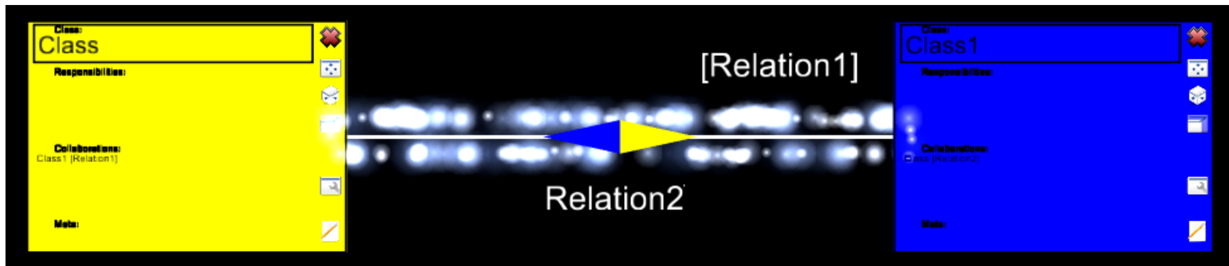


Figure 4.12: A bidirectional link in editing mode

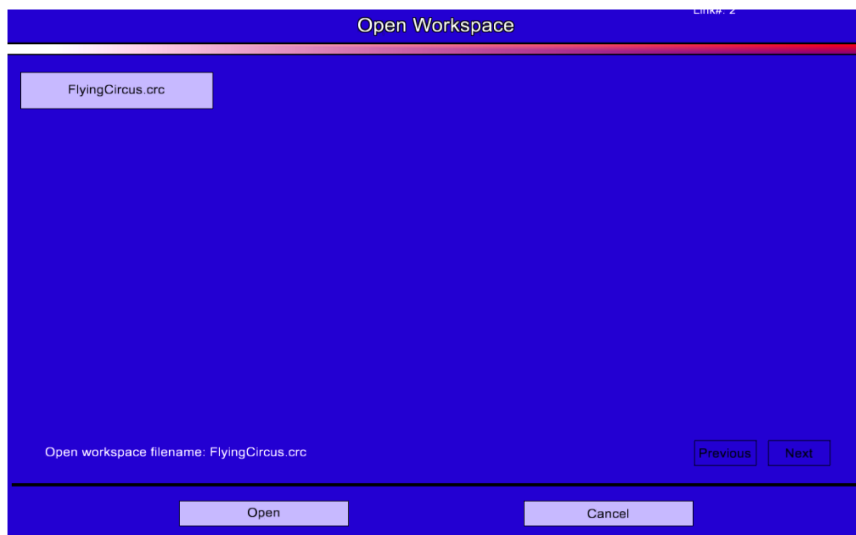


Figure 4.13: The 'Open project/workspace' window

list available options. Next and previous buttons allow paging, if the window can't include all available options. Figure 4.13 shows such a window. The 'Options' window uses check boxes to enable features of the tool, like the axes and grid drawing, the rendering of the stable cameras rendering and the usage of effects like anti-aliasing. This window can be seen in figure 4.14. Finishing the 'Edit Settings' window can be seen in figure 4.15 and uses tabs, labels, textboxes and buttons to change the properties of a crc card, which include position, rotation, size and colour. Pressing 'Preview' shows the card with the new settings in the CRC world, pressing 'Ok' updates the card and pressing 'Cancel' reverts the card to its initial values.

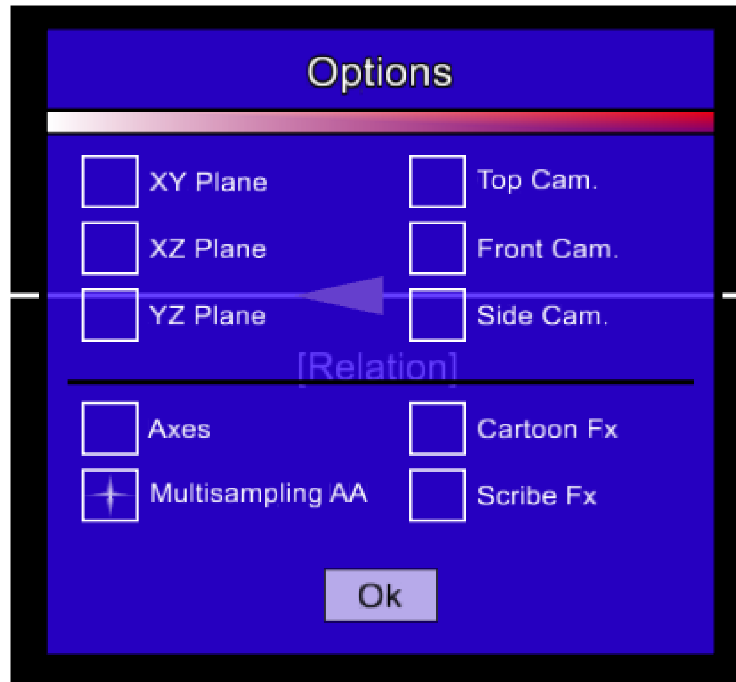


Figure 4.14: The application 'Options' window

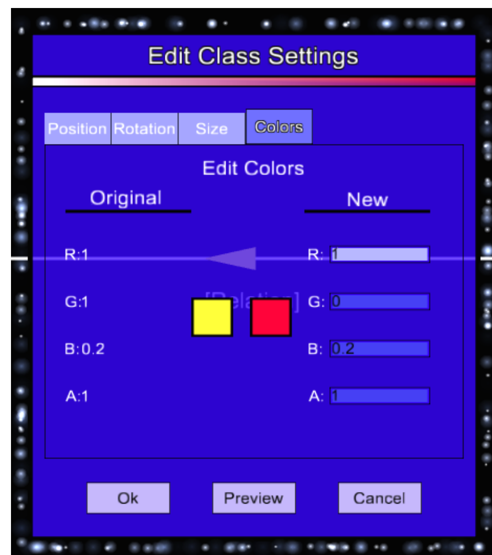


Figure 4.15: 'Edit Settings' window of a CRC card

Chapter 5

Implementation

In this chapter we will talk about how the tool was implemented. Firstly we will talk about the tools we used and especially about OpenSceneGraph, which is the basis of the developed 3D environment. Next we will talk more specifically about the viewer, controller and model components. Afterwards we will refer to the visual displays used for stereo support, analyze the performace of the tool and refer to any portability issues and problems.

Our tool was given the name 'Flying **C**i**RC**u**S**', from CRC Studio. This name is inspired by the homonymous BBC sketch comedy programme from the Monty Python comedy team.

5.1 Tools used

Our tool was based on many open source products. We preferred to use open source software because it is free, readily available, multiplatform, matured and respected by many programmers. Moreover the basic characteristic of these tools is the availability of code, so we were able to make desired changes and contributions, something which can't be done with common proprietary software.

5.1.1 Use of a Scene Graph

We decided early enough to use a scene graph instead of writing low-level OpenGL/Direct3D APIs calls to implement our tool. The point was to try to focus on the design of the CRC tool instead of reinventing the wheel. The scene graph is a very flexible tool and it is the kind of system, one would implement through much trial and error to achieve the same functionality. But what is a scene graph?

Introduction to Scene Graphs

A scene graph is a hierarchical tree data structure that organizes spatial data for efficient rendering. The scene graph tree is headed by a top-level root node. Beneath the root node, group nodes organize geometry and the rendering state that controls their appearance. Root nodes and group nodes can have zero or more children. However group nodes with zero children are essentially no-ops. At the bottom of the scene graph, leaf nodes contain the actual geometry that make up the objects in the scene [47].

Applications use group nodes to organize and arrange geometry in a scene. Imagine a 3D database containing a room with a table and two identical chairs. You can organize a scene graph for this database in many ways. One example is the following. The root node has four children group nodes, one for the room geometry, one transform node for the table, and one node that transforms its children for each chair. There is only one chair leaf geometry node though, because the two chairs are identical. So their parent group node transforms the chair to two different locations to produce the appearance of two chairs. The table group node has a single child, the table leaf node with the geometry of the table. The room leaf node contains the geometry for the floor, walls and ceiling. Figure 5.1 illustrates this example.

Scene graphs usually offer a variety of different node types that offer a wide range of functionality, such as switch nodes, that enable or disable their children, level of detail (LOD) nodes that select children based on distance from the viewer, and transform nodes that modify transformation state of child geometry. Object-oriented scene graphs provide this variety using inheritance; all nodes share a common base class with specialized functionality defined in the derived classes.

The large variety of node types and their implicit spatial organization ability provide data storage features that are unavailable in traditional low-level rendering APIs. OpenGL and Direct3D focus primarily on abstracting features found in graphics hardware. Although graphics hardware allows storage of geometric and state data for later execution, low-level API features for spatial organization of that data are generally

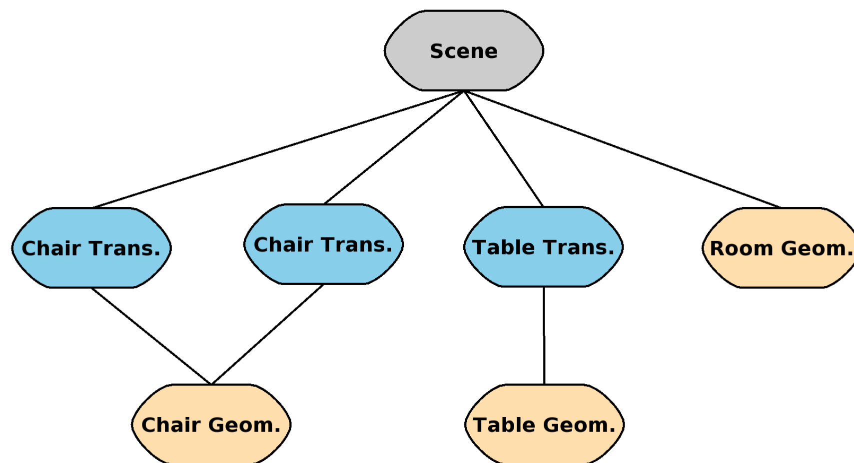


Figure 5.1: A typical scene graph

minimal and primitive in nature, and inadequate for the vast majority of 3D applications. Scene graphs are middle ware, which are built on top of low-level APIs to provide spatial organization capabilities and other features typically required by high-performance 3D applications. Even VRML files, the 3D analogue to HTML, describe 3D objects and worlds using a scene graph [23].

Scene Graphs features

Scene graphs expose the geometry and state management functionality found in low-level rendering APIs, and provide additional features and capabilities, such as the following:

- Spatial organization - The scene graph tree structure lends itself naturally to intuitive spatial organization
- Culling - View frustum and occlusion culling on the host CPU typically reduces overall system load by not processing geometry that does not appear in the final rendered image.
- LOD - Viewer-object distance computation on bounding geometry allows objects to efficiently render at varying levels of detail. Furthermore, portions of a scene can load from disk when they are within a specified viewer distance range, and page out when they are beyond that distance.
- Translucency - Correct and efficient rendering of translucent (non-opaque) geometry, requires all

translucent geometry to render after all opaque geometry. Furthermore, translucent geometry should be sorted by depth and rendered in back-to-front order. These operations are commonly supported by scene graphs.

- State change minimization - To maximize application performance, redundant and unnecessary state changes should be avoided. Scene graphs commonly sort geometry to minimize state changes.

- File I/O - Scene graphs are an effective tool for reading and writing 3D data from/to disk. Once loaded into memory, the internal scene graph data structure allows the application to easily manipulate dynamic 3D data. Scene graphs can be an effective intermediary for converting from one file format to another.

- Additional high level functionality - Scene graph libraries commonly provide high-level functionality beyond what is typically found in low-level APIs, such as full-featured text support, support for rendering effects (such as particle effects and shadows), rendering optimizations, 3D model I/O support, and cross-platform access to input devices and render surfaces.

How Scene Graphs render

A trivial scene graph implementation allows applications to store geometry and execute a draw traversal, during which all geometry stored in the scene graph is sent to the hardware as OpenGL/Direct3D commands. However, such an implementation lacks many of the features described previously. To allow for dynamic geometry updates, culling, sorting and efficient rendering, scene graphs typically provide more than a simple draw traversal. In general, there are four types of traversals, although the last one is not available in every scene graph:

- Update - The update traversal (sometimes referred to as the application traversal) allows the application to modify the scene graph, which enables dynamic scenes. Updates are accomplished either directly by the application or with callback functions assigned to nodes within the scene graph. The update traversal modifies geometry, rendering state, or node parameters, to ensure the scene is up-to-date for the current frame. Applications use the update traversal for example to modify the position of a flying aircraft in a flight simulation, or to allow user interaction using input devices.

- Cull - During the cull traversal, the scene graph library tests the bounding volumes of all nodes for inclusion in the scene graph. If a leaf node is within the view, the scene graph library adds leaf node geometry references to a final rendering list. This list is sorted by opaque versus translucent, and translucent geometry is further sorted by depth.

- Draw - In the draw traversal (sometimes referred to as the render traversal), the scene graph traverses the list of geometry created during the cull traversal and issues low-level graphics API calls to render that geometry.

- Event - Some scene graphs also have a fourth traversal, the event traversal, which processes input and other events each frame, just before the update traversal.

Typically, these four traversals are executed once for each rendered frame. However, some rendering situations require multiple simultaneous views of the same scene. Stereo rendering and multiple display systems are two examples. In these situations, the update traversal is executed once per frame, but the cull and draw traversals execute once per view per frame. That's twice per frame for simple stereo rendering, and once per graphics card per frame on multiple display systems. This allows systems with multiple processors and graphics cards to process the scene graph in parallel. The cull traversal must be a read-only operation to allow for multithreaded access [47].

OpenSceneGraph

One of the most used scene graphs nowadays and the one *Flying Circus* was based on is OpenSceneGraph[1] (OSG)[2]. OSG is a set of open source libraries that primarily provide scene management and graphics rendering optimization functionality to applications. OSG plays a key role in the 3D application software stack. It's the middleware above the lower-level OpenGL hardware abstraction layer (HAL), providing extensive higher-level rendering, I/O, and spatial organization functionality to the 3D application. It is written in portable ANSI C++ and uses the industry standard OpenGL low-level graphics API. As a result, OSG is cross platform and runs on Windows, Mac OS X and most UNIX and Linux operating systems. Most of the OSG operates independently of the native windowing system. However, OSG includes code

[1]http://www.openscenegraph.org/projects/osg
[2]From now on OSG

to support some windowing system specific functionality, such as input devices, window creation, and pbuffers[3].

OSG is open source and is available under a modified GNU Lesser General Public Licence, or Library GPL (LGPL) software licence. OSG's open source nature has many benefits:

- Improved quality - OSG is reviewed, tested, and improved by many members of the OSG community.
- Improved application quality - To produce quality applications, application developers need intimate knowledge of the underlying middleware. If the middleware is closed source, this information is effectively blocked and limited to vendor documentation and customer support. Open source allows application developers to review and debug middleware source code, which allows free access to code internals.
- Reduced cost - Open source is free, eliminating the up-front purchase price.
- No intellectual property issues - There is no way to hide software patent violations in code that is open source and easily read by all.

Furthermore, one of the key features of OSG is that currently it offers excellent and free documentation. Despite most open source projects that lack up to date and consistent documentation, OSG is an exception to this rule. Apart from the extensive number of examples, covering every aspect of the scene graph, there is also a free book available online, *OpenSceneGraph, Quick Start Guide* [47], which is an excellent introduction to OSG.

OSG is designed up front for portability and scalability. Moreover it is designed to be both flexible and extensible to allow adaptive development over time. To enable these design criteria, OSG is built with the following concepts and tools:

- ANSI standard C++
- C++ Standard Template Library (STL)
- Design patterns [31]

The OSG runtime exists as a set of dynamically loaded libraries (or shared objects) and executables. These libraries fall into five conceptual categories:

[3]Pbuffer, also called back buffer or pixel buffer, is an offscreen buffer [2]

- The Core OSG libraries provide essential scene graph and rendering functionality, as well as additional functionality that 3D graphics applications typically require.
- NodeKits extend the functionality of core OSG scene graph node classes to provide higher-level node types and special effects.
- OSG plugins are libraries that read and write 2D image and 3D model files
- The interoperability libraries allow OSG to easily integrate into other environments, including scripting languages such as Python and Lua.
- An extensive collection of applications and examples provide useful functionality and demonstrate correct OSG usage.

The application was initially developed using the 1.x OSG version library and the osgProducer viewer but was later ported to OSG 2.x version, using OSG's osgviewer library. The developed tool requires version 2.2 or greater.

5.1.2 libsigc++

Because of the use of OSG, we used the C++ language to program the tool. C++ is a language that promotes type safety and libsigc++[4] is a typesafe callback system for standard C++. Compile time typesafe callbacks are faster than run time checks, which we would be using if we were using the C language and report typesafety violations line numbers correctly.

Libsigc++ provides the concept of a slot, which holds a reference to one of the things that can be used as a callback, all of which can take different numbers and types of arguments:

- A free standing function
- A functor object that defines operator()
- A pointer to a member function and an instance of an object on which to invoke it

To make it easier to construct these, libsigc++ provides functions for creating slots from static functions and member functions.

[4]http://libsigc.sourceforge.net

For the other side of the fence, libsigc++ provides signals, to which the client can attach slots. When the signal is emitted, all the connected slots are called.

The tool was developed using the 2.0.17 version of the library.

5.1.3 Flex, bison and ylmm

As already mentioned *Flying Circus* should be able to parse appropriate workspace files. So we need the appropriate tools to read these files and discover its structure, by splitting the source files into tokens and finding the hierarchical structure of the program. For the first task we used flex[5], a fast lexical analyzer generator and for the second one, bison[6], a general purpose parser generator that converts an annotated context-free grammar int an LALR(1)[7] or GLR[8] parser for that grammar. Both tools are extensively used in the Unix/Linux world, but are also available for Windows environments.

Because we used C++ for the development of the tool, we also used Yacc/Lex--[9] (ylmm), which is a set of headers and macros, written by Christian Holm Christiansen to wrap Bison generated parsers and Flex generated scanners in C++ classes. Although there are projects like bison++, which has the advantage of direct C++ output, Christian sees the advantages of his method in being independent from the internals of the employed Bison/Lex implementation. Therefore it is more stable with respect to changes in the Bison/Lex projects and not immediately affected by their internal modifications.

5.1.4 SCons

As *Flying Circus* should be a multiplatform application, we should also use a cross platform build system. We found that we could accomplish this target, by using SCons[10], an Open Source software next generation construction tool which has the following features:

- Designed from the ground up for cross-platform builds, and known to work on Linux, other POSIX systems (including AIX, *BSD systems, HP/UX, IRIX and Solaris), Windows NT, Mac OS X, and

[5] http://flex.sourceforge.net/

[6] http://www.gnu.org/software/bison/

[7] **L**ook **A**head one symbol from **L**eft to right producing a **R**ightmost derivation parser[1]

[8] **G**eneralized **L**eft-to-right **R**ightmost derivation parser[1]

[9] http://cholm.home.cern.ch/cholm/misc/ylmm/index.html

[10] http://www.scons.org/

OS/2.

- Configuration files are Python scripts. Use the power of a real programming language to solve build problems.
- Reliable, automatic dependency analysis built-in for C, C++ and Fortran. Dependency analysis is easily extensible through user-defined dependency Scanners for other languages or file types.
- Built-in support for C, C++, D, Java, Fortran, Yacc, Lex, Qt and SWIG, and building TeX and LaTeX documents. User defined builders make it easily extensible for other languages or file types.
- Built-in support for Microsoft Visual Studio .NET and past Visual Studio versions, including generation of .dsp, .dsw, .sln and .vcproj files.
- Reliable detection of build changes using MD5 signatures; optional, configurable support for traditional timestamps.
- Integrated Autoconf-like support for finding includes files, libraries, functions and typedefs.
- Global view of all dependencies. No more multiple build passes or reordering targets to build everything.

5.1.5 Bazaar

Bazaar was the version control system we used during the development of *Flying Circus*. Bazaar[11] is a distributed version control system (DVCS)[12]. DVCS systems primarily aid in allowing independent developers to work asynchronously, and then synchronize and pull in changes from each others. A centralized repository is not necessary to pull/push changes. The user can simply have his working repository in his flash memory stick.

Bazaar has the following features:

- Works everywhere. It is based on Python
- Has good performance

[11]http://bazaar-vcs.org/
[12]From now on DVCS

- Is safe with the data as it uses a huge test suite to ensure data integrity
- Just works. It is friendly and has a natural feel
- Free. Based on GPL
- Easy to integrate, because of Python API

5.1.6 Valgrind

Valgrind[13] is an open source award-winning suite of tools for debugging and profiling Linux programs. With the tools that come with Valgrind, many memory management and threading bugs can be automatically detected, avoiding hours of frustrating bug-hunting and making programs more stable. This tool can also perform detailed profiling, speeding up and reducing memory of programs.

The valgrind distribution currently includes four tools: a memory error detector, a cache (time) profiler, a call-graph profiler, and a heap (space) profiler. It runs on the following platforms: X86/Linux, AMD64/Linux, PPC32/Linux, PPC64/Linux.

5.1.7 GCC

For the Linux/Unix environment we built the *Flying Circus* using the GNU Compiler Collection[14] GCC[15]. GCC is a set of compilers produced for various programming languages by the GNU Project. GCC is a key component of the GNU toolchain and as well as being the official compiler of the GNU system, GCC has been adopted as the standard compiler most other modern Unix-like computer operating systems, including Linux, the BSD family and Mac OS X. GCC has been ported to a wide variety of computer architectures, and is widely deployed as a tool in commercial and closed development environments. GCC is also used in popular embedded platforms like Symbian, Playstation and Sega Dreamcast [65].

Specifically we used the 4.2.x version of the compiler. This version of the compiler has a stricter compliance with the C++ standard and appropriate flags to create either a generic build, which is supposed to work for every x86 platform or a native one, where the compiler automatically detects the CPU[16] of the

[13]http://valgrind.org
[14]http://gcc.gnu.org/
[15]From now on GCC
[16]Central Processing Unit

building system and uses appropriate optimizations. It was found that optimized executables were a little smaller in size around 1-2% and a little faster in speed again around 1%. Note that the speed comparison is based on the numbers of frames of a static scene. There might be more speed-ups for operations which dynamically change the scene, which unfortunately can not be measured.

5.1.8 Visual Studio 2005

Visual Studio 2005[17] is Microsoft's flagship software development product for computer programmers. It centers on an integrated development environment which lets programmers create standalone applications, web sites, web applications, and web services that run on any platforms supported by Microsoft's .NET Framework (for all versions after Visual Studio 6). Supported platforms include Microsoft Windows servers and workstations, PocketPC, Smartphones, and World Wide Web browsers [65].

Visual Studio includes the following:

- Visual Basic (.NET)
- Visual C++
- Visual C#
- Visual J#
- ASP.NET

Express editions of Visual Studio have been released by Microsoft for lightweight streamlined development and novice developers which are free.

5.1.9 Blender

All 3D models were made using Blender. Blender[18] is a free software 3D animation program. It can be used for modelling, UV unwrapping, texturing, rigging, skinning, animating, rendering, particle and other simulating, non-linear editing, compositing, and creating interactive 3D applications. Blender is available for several operating systems, including Microsoft Windows, Mac OS X, Linux, IRIX, Solaris, FreeBSD,

[17] http://msdn2.microsoft.com/en-us/vstudio/default.aspx
[18] http://www.blender.org/

and OpenBSD. Blender has a robust feature set similar in scope and depth to other high-end 3D software such as Softimage|XSI, Cinema 4D, 3ds Max and Maya. These features include advanced simulation tools such as rigid body, fluid, and softbody dynamics, modifier based modelling tools, powerful character animation tools and a node based material and compositing system, and Python for embedded scripting [65].

5.1.10 Gimp

All icons and images were processed using The GNU Image Manipulation Program[19], or GIMP[20]. GIMP is a raster graphics editor application with some support for vector graphics. GIMP is used to process digital graphics and photographs. Typical uses include creating graphics and logos, resizing and cropping photos, altering colours, combining multiple images, removing unwanted image features, and converting between different image formats. GIMP can also be used to create basic animated images in GIF format. It is often used as a free software replacement for Adobe Photoshop, the most widely used bitmap editor in the printing and graphics industries [65].

5.2 Components implementation

The basic components of our application are the viewer, the controller and the model. In the following subsections we will talk about their implementation.

5.2.1 Viewer implementation

The viewer component is the part of the application that knows how to render a crc design. It is part of the controller with which it communicates using sigc++ signals. The basic class of the viewer is osgViewer. This class implements a viewer who knows how to draw. This class also holds the root of the scenegraph whose childs are the nodes of the two cameras used in our tool, the 3D world camera and the HUD camera. Moreover it holds the two window managers used to manage opened windows containing controls. The first window manager is for the crc cards and links and the second is for the windows opened in the HUD

[19]http://www.gimp.org/

[20]From now on GIMP

camera. It knows how to provide a front, top, side or back view and also how to enable any available effects, like the scribe effect or multisampling. When the viewer is done a signal is send to the controller to quit the application.

Viewer dispatcher

The osgViewerDispatcher is part of the osgViewer and has the responsibility of dispatching sigc++ messages sent by the controller for updating the viewer state, like moving cards for example. Moreover he is responsible for any internal messages send by the controls to change the internal state of the viewer. Therefore, in order to provide this functionality, the osgViewerDispatcher has to communicate with appropriate windows.

3D Widgets

The implementation of the 3D widgets was a very time consuming work. But the result of this work is a library based on OpenSceneGraph for the creation of 3D windows using 3D controls. Up to now there is no available library with such controls and every organism or research institute working with 3D environments has built its own library. Unfortunately these libraries are not available to the public. Therefore we had to develop our own, which was based on an initial version of a combat simulator windows framework[21]. Hopefully in the future things will change, as one of the basic aims of the OpenSceneGraph community is to provide such a library in the osg core.

Controls The basic class of all controls is Control. This class is the super class of all other controls and provides the basic functionality for a control. Each control has an id and a name. The id is unique in the container holding the control and the name describes what control this is. The control knows in which window manager it exists. Moreover it knows how to build itself by using the ControlGeometryBuilder class and a buildGeometry method. Methods to get the osg node with the geometry of the control along with methods to get and set the position, rotation and size of the control are also provided. Containers are implemented holding controls, so a control must also know the root control along with the parent control. A style class used by the control also knows any properties of the control like colour for example.

Some of the basic controls, are the following:

[21] CSP, Combat Simulator Project, http://csp.sourceforge.net

- **Labels** Labels are a simple controls. They provide methods to set or get the text of the label. The user can also set the size of text, the way the text is rendered, for example if we want it to be outlined or aligned in a specific way, or if we want it to be a billboard.

- **Buttons** Buttons are single control containers holding a label. When a mouse button is clicked, a registered action is executed through sigc++. Buttons also provide methods to set an icon on the button and if we want this button to be activated, deactivated or pushed. Animation callbacks can be set when the mouse points the buttons, to activate simple animations.

- **Tabs** Tabs are containers of tab pages, which are single control containers. Tabs render on top of each tab page a button which when clicked activates the specific tab page and deactivates any other. This is done by using switch nodes provided by OSG.

- **Manipulators** Manipulators are controls that help the manipulation of windows. There are 3 different kind of manipulators, move, rotation and resize ones, and specific manipulators for each axis in each manipulator case, providing constrained manipulation. When the mouse points the manipulator an animation callback is a set and the specific manipulator for the specific axis is animated. When clicked appropriate action on appropriate axis is executed for the parent window.

- **Textboxes** Textboxes are controls that provide methods for setting and getting the text and editing mode, methods to get if the textbox is modified along with methods for adding a char, deleting a char, or moving the internal cursor. Moreover they can be set to be billboards and also specify if you want the text to be outlined in a specific way.

- **Checkboxes** Checkboxes are simple controls which when checked, execute a specific method using sigc++ signals. They also provide a label for explaining the function of the checkbox.

Windows Windows are single control containers. They are the root containers of the previous controls. Moreover they emit signals from controls which are manipulated by the user, like buttons, textboxes, manipulators and checkboxes, to the window manager that holds them. Also they can be rendered using a specific theme. Methods for setting or unsetting them to edit mode are also provided. When in editing mode particles are drawn around the borders of the window. All the opened windows in our application are

subclasses of the basic Window class, extending its functionality. These windows include the CRCCard, Link, MainMenu, Open, Setting, Create, Options and many other windows.

WindowManager The WindowsManager is a class that manages any open windows. It provides methods to show, find or close a specific window. Furthermore it provides methods to see which specific node of a control the mouse pointer picks or intersects. Furthermore the windowManager is responsible for rendering axes and grids. This class is part of the osgViewer class.

Input event handler

The InputEventHandler is responsible for registering event handlers. Moreover it provides methods to get the position of the mouse pointer. The keyboard event handler is a class which is part of the InputEventHandler, and holds registered actions for specific key strokes. It supports all keyboards' keys, even modifiers. The mouse event handler is a class which is also part of the input event handler and holds registered actions for mouse button presses, releases, drags or moves. Intersection functionality provided by the previous handler allows the end user to select a portion of the displayed image. The application performs an operation to map the 2D xy mouse location to the corresponding scene graph node, by drawing a ray from the xy mouse location and finding which objects in the 3D world it intersects. The nearest node is then selected and stored for future operations.

5.2.2 Controller implementation

Controller is the component of the tool that holds all the logic of the application. Its basic class is the crcController. The crcController holds viewers with which it can communicate using sigc++ signals. The crcController can send signals to the viewer and has a slot for getting the corresponding signals emitted by the viewer. These signals are dispatched in the crcControllerDispatcher who also makes appropriate actions. The crcController also holds the model component which it updates when needed. This component also includes the following classes:

- **Properties** Properties is a class that is part of the crcController class and provides access methods to get file paths for the design files, documentation files, and logs.

- **crcControllerDispatcher** The dispatcher is implemented in the crcControllerDispatcher class and is also part of the crcController. When a signal from the viewer arrives in the slot of the crcController, the controller uses the dispatcher to dispatch it and make the appropriate actions.

- **UpdateViewer** The UpdateViewer is responsible for updating registered viewers. It is part of the crcController class and is used by the crcControllerDispatcher. It emits appropriate sigc++ signals which are sent to the viewer.

- **Parser** Parser is a class that knows how to load and parse a crc workspace file. It is also a part of the crcControllerDispatcher class and is implemented using the ylmm headers. These headers provide wrappers for the creation of a C++ parser based on lex and bison tools. The format of the parsed file can be seen in Appendix A, page 97

- **WorkspaceSaver** WorkspaceSaver is a class that knows how to save a workspace. It is a part of crcControllerDispatcher and uses the available workspace in the Model class to write the appropriate crc workspace file, at the path defined in the Properties class. The syntax of the saved file follows a specific grammar which can be shown in Appendix A, page 97.

- **DocumentationGenerator** DocumentationGenerator is class that knows how to create the documentation of a workspace. It is part of the crcControllerDispatcher and uses the available workspace in the Model class, described later, to write the documentation file of the specific workspace, at the appropriate path, defined in the Properties class. An example of the created file can be seen in Appendix B, page 109.

5.2.3 Model

The model is the component of our application that holds domain specific information that means it knows about workspaces, projects, cards and links. It has been implemented with a singleton class which is part of the controller. The Model class holds a workspace and provides methods for getting and setting it. Model is updated only by the controller.

The model component also includes the following classes:

- **Workspace** The Workspace is part of the Model class. Each workspace has an Id and methods to get or change it. Moreover the Workspace class, holds a set of all projects contained in the specific

workspace and provides methods for making a specific project active. Methods for adding, removing and renaming a workspace are also provided.

- **Project** A workspace has a set of Project classes. Project classes hold sets of cards and links. In addition it holds an id and methods to get or change it. Moreover it provides methods for adding, removing or updating CRC cards and links, keeping consistency, by not allowing duplicate card names and appropriately updating collaborators when needed.

- **CRCCard** CRCCard is a class that holds the properties of a CRC card. These include the class name, the responsibilities, the collaborations, the meta tag, any comment along with the position, rotation, size and colour values. All these properties have been implemented as different classes. The CRCCard provide methods to access or change the above properties.

- **Link** Link is a class that holds a link. A Link needs to know two strings which are the unique card names that use the link along with another string, which holds the relation. Each Link has its own id which is unique and it is generated using the unique names of the collaborating cards.

5.3 Stereo support

One of the basic goals during the development of our tool was the support of stereoscopic displays. This would allow users to get immersed in the world of the CRC cards, increasing the effectiveness of spatial memory. The OpenSceneGraph native stereo support takes into account the eye separation, the screen distance and a fusion point which is automatically calculated. It assumes then that the user watches at the screen keeping his head horizontal. Therefore the points of view for each eye are slightly shifted on the left and on the right.

We managed to provide stereo support through shell scripts adjusting appropriate OpenSceneGraph parameters, for both windows and Linux platforms and for every stereoscopic visual display available at the HCI Lab of ICS FORTH. These displays include:

- **HMD** A Virtual Research V8 HMD display, supporting spatial multiplexing. It uses two small lcd monitors in front of the user's eyes, but unfortunately the resolution is pretty small, 640x480 pixels and the field of view is only 60°. The achieving stereo effect though is at a satisfactory level.

The drawback is that HMDs are occlusive displays and since we support input through mouse and keyboard, which do not have a 3D representation in our tool, the user has difficulties to give correct input.

- **EPC-2** The second system is EPC-2, supporting temporal multiplexing. This system includes an infrared emitter to send out a pulse identifying a field as left or right. The stereoscopic eyewear (shutter glasses), receives and decodes the pulse and creates the drive signals necessary to activate the shutter. The EPC-2 includes sync doubling logic necessary on non-stereo ready computers. The images are formatted in an above and below format: The left subfield is approximately the top half of the display area. The right subfield is approximately the bottom half of the display area. In the middle there is a thin strip, the subfield blanking interval, which is sync doubling function, generating an additional vertical sync pulse. The net effect is that the left image is stretched to fill the screen in one field and the right image is stretched to fill the screen in the next. Therefore the vertical resolution is divided by two and the refresh rated doubled. The subfield blanking internal is required to give the monitor time to get the beam back to the top of the screen. Figure 5.2 shows the image format. With EPC-2 we get the best stereo performance in comparison to the other supported immersive displays, but unfortunately monitor characteristics can limit the resolution, since we need a combination of very big resolutions and refresh rates.

- **Anaglyphic stereo** Simple red-cyan or red-green glasses supporting spectral multiplexing. It is the simplest immersive visual display which is readily available and can be used in all everyday computers. Moreover there are no limitations in the supported resolutions, even with projectors. On the other hand simple paper, uncorrected gel glasses cannot compensate for the 250 nanometer difference in the wave lengths of the red-cyan filters so the red filtered image is somewhat blurry, when viewing a close computer screen or printed image.

The HMD and EPC-2 displays have problems with picking using version 2.2 of OpenSceneGraph, which is described later, in the problems section.

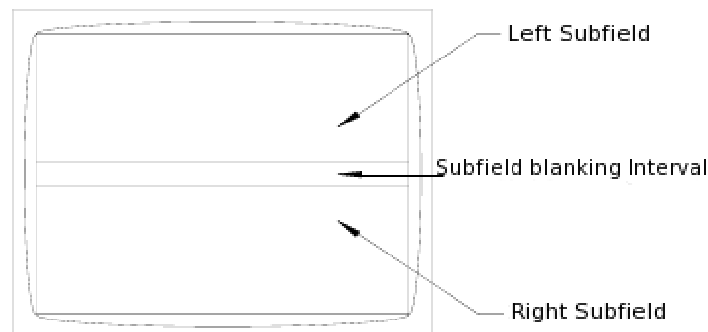


Figure 5.2: Above-And-Below format

5.4 Portability issues

A basic aim of *Flying Circus* was portability. And we managed to at least make *Flying Circus* available to Windows and Linux platforms. Probably many other Unix platforms are also supported but unfortunately are not tested. This goal was realized by firstly using multiplatform tools such as OpenSceneGraph, libsigc++, scons for building the code and bazaar as a version control system. Except from using multiplatform tools it is also important to write portable code. That means avoiding hard-coded system constants, or platform-specific ones when POSIX[22] or commonly-accepted ones exist. Moreover it required the avoidance of hard coded paths. Since windows use a different naming for file paths in comparison with Unix, we used an abstraction mechanism, which provided correct paths for the specific platform for which the executable was built. Finally the code should be tested with different compiler versions. That improved the code quality, since each compiler provides a different compliance with the C++ standards. Making *Flying Circus* portable, also improved the stability of the application, since different problems arose in different platforms.

5.5 Performance

Another basic target of our tool was to have acceptable performance in everyday desktops. In order to achieve this we designed the windows and the controls of the tool using very basic shapes and meshes. We tried to avoid complicated geometries, which would be represented by a large number of vertices, and

[22]POSIX or Portable Operating System Interface is the collective name of a family of related standards specified by the IEEE to define the application programming interface (API) for software compatible with variants of the Unix operating system

Figure 5.3: Current transparency rendering. Controls are not blended with the transparent window (Class is the transparent card).

therefore would make the scene more power demanding. Moreover we did not use lights and avoided unecessary state changes, which are costly. Our tool can render the design of it self, which consists of 30 crc cards and 38 links at about 35 fps[23] on a Linux system, using a 1024x768 display, on an old Pentium4 at 2,4 GHz with a previous generation graphics card (GeForce 7600GS). This framerate is acceptable and it is difficult for the user to notice any lag, which could probably destroy the real-time experience. When using stereo the performance drops to around 20fps, which is rather limiting. The bottleneck for this system is the CPU performance. We came to this conclusion by finding that for bigger resolutions, the fps value is the same. In case the bottleneck was the graphics card, we should see a frame drop for bigger resolutions.

5.6 Problems

Here we will describe the current problems of the *Flying Circus* application. Some of them are related to the underlying layer used, OpenSceneGraph, and the rest of them exist because of non implemented features.

5.6.1 Transparency

Flying Circus supports the rendering of transparent or semi-transparent cards, if this is desired by the user. Rendering of transparent objects is different from the rendition of simple opaque objects. To render transparent objects correctly, the rendering is done in two passes. The first step renders opaque objects and the second one renders transparent objects, which are then blended with the opaque objects. For the second pass it is important to render things from back to front, rendering furtherest objects first and nearest last, so that the blending of objects is correct. Unfortunately, because the 3D world and the camera frustum[24] is rather big in dimensions and controls are rendered very close to cards, there are some precision errors which lead to unpredicted rendering of transparent objects. Sometimes for example the text in the textboxes was not drawn. We partially solved that by rendering things in many passes, where windows for example are rendered first, and controls are rendered afterwards. This renders windows correctly but since controls are rendered afterwards, they are not blended with transparent windows. Figure 5.3 show the current transparency rendering. The best solution for a correct rendering of transparent objects can be done by using shaders and depth peeling [27].

5.6.2 Picking in stereo is problematic

As already mentioned in the stereo section, picking in some stereo modes using a mouse cursor is problematic. This problem is related to OpenSceneGraph and appears in modes that divide the initial screen into two sub screens, each one displaying the right or left view. The cursor tries to pick objects in the 3D world assuming that we have only one big view instead of two. So the objects picked do not correspond to what the user thinks. In our system, this problem will occur when using the HMD or the EPC-2 stereo visual displays. According to the lead programmer of the OpenSceneGraph this problem will probably be fixed in the next version (2.4) of the library.

[23]Frames per second

[24]Frustum is the view volume, a truncated pyramid with rectangular base. Near and far clipping planes can reduce the volume. Only primitives inside the volume are rendered

5.6.3 Textboxes and big texts

A drawback of the current textbox implementation, is that if the text input of the user is bigger from the size of the rendered textbox, then the extra text is not displayed. Moreover the textbox does not inform the user that there is more text and does not provide controls to render it. So if the user wants to see the extra text, the only solution at least for the CRC cards, is to resize them, so that textboxes get expanded and the extra text gets rendered. A more delicate solution is to support either scrollbars and let the user scroll the text or split the text into pages and provide controls for the selection of the desired page.

5.6.4 No 3D cursor

The stereo problem could also be avoided by using a 3D cursor instead of the default 2D one, available from X11 or Windows systems. Such a cursor would exist in the 3D space giving always intersections consistent with what the user sees. Moreover such a cursor would not destroy the stereo effect when using immersive visual displays, something which can happen with the default 2D cursor. The drawback of a 3D cursor is than it needs a 3D input device instead of a simple mouse to be effective.

5.6.5 Problems with Localization

Finally there are problems with localization. Although *Flying Circus* supports UTF-8 workspace files and can render cards with non Latin characters (if such cards are defined in the workspace files), unfortunately OpenSceneGraph can't catch such characters from the keyboard. Moreover it seems that there are no plans for supporting such a feature. So the only way to support localization is to provide our own mappings for *Flying Circus*.

Chapter 6

Conclusion and Future work

The aim of this thesis was to present the design and implementation of a 3D CRC automated tool, called *Flying Circus*. *Flying Circus* is a design tool offering interactive, direct manipulation of 3D CRC cards. In this tool, cards may be created, positioned, and manipulated in the 3D world, supporting directional labelled links to illustrate collaborators. Facilities like zooming, panning, 3D rotations, smooth navigation, and stereoscopic display are fully provided, essentially delivering an evolving, exploratory and immersive design space. The benefits range from increased control, focus of attention, visual encoding and better exploitation of the human spatial memory. Although *Flying Circus* is a 3D application, we tried to efficiently use available resources, therefore making this tool available for use on everyday computers. Moreover we managed to make *Flying Circus* multiplatform, supporting both Windows and Linux platforms.

We plan to extend *Flying Circus*, with more features, since this work focused basically on the development of the basic framework for the visualization of 3D crc designs. The system still needs some interface enhancements, like support for scrolling or paging in textboxes, the implementation of a 3D cursor, support of a pivot point which users will be able to position freely in the 3D world and from which new cards will be created , along with shortcuts for constrained moving, rotating or scaling a card across an axis. A very important feature missing right now is support of a global undo/redo stack. The tool is not forgiving regarding users errors. Moreover support of copy/cut/paste operations for text and cards, will increase design speed. Furthermore, implementing a feature that could load a design and automatically play it

back it, would be really interesting. This could help an external watcher understand the steps and phases a particular design passed through. The play back could be done either using animation, with the original timing of the design actions or using a stepwise animation without timing. Finishing, *Flying Circus*, could be expanded by using new, more sophisticated 3D input devices, which would make interaction with the cards more intuitive along with support for initial code generation for the most frequently used Integrated Development Environments (IDEs), based on the created CRC cards.

Appendix A

Example of workspace format

```
workspace FlyingCircus

project FlyingCircus begins

card [
        class                   Button
        responsibilities        "Fire action when pressed"
        collaborations          "ControlGeometryBuilder [Uses], Control [ISA]"
        meta                    "viewer"
        comment                 ""
        attributes [
                position        821.422 841.858 -600
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0.8 1
        ]
]

card [
        class                   CRCCard
        responsibilities        "Knows to build a CRC card\nChange properties of CRC card"
        collaborations          "Window [ISA], Link [HasMany]"
        meta                    "Viewer"
        comment                 ""
```

```
        attributes [
                position        350 240 -500
                size            145 105 1
                angle           -25 0 0
                color           0.6 0 1 1
        ]
]

card [
        class                   Checkbox
        responsibilities        "Allows user to enable or disable option"
        collaborations          "ControlGeometryBuilder [Uses], Control [ISA]"
        meta                    "viewer"
        comment                 ""
        attributes [
                position        1000 900 -600
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0.8 1
        ]
]

card [
        class                   Container
        responsibilities        "Contains controls"
        collaborations          "Control [ISA]"
        meta                    "Viewer"
        comment                 ""
        attributes [
                position        600 500 -600
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0.8 0.8
        ]
]

card [
        class                   Control
        responsibilities        "Knows id\nKnows properties"
        collaborations          ""
        meta                    "Viewer"
        comment                 ""
```

```
        attributes [
                position        900 500 -600
                size            145 105 50
                angle           0 0 0
                color           0 0.6 0.6 0.6
        ]
]

card [
        class                   ControlGeometryBuilder
        responsibilities        "Builds geometry of controls"
        collaborations          ""
        meta                    "Viewer"
        comment                 ""
        attributes [
                position        401.195 1084.04 -600
                size            173.495 105 1
                angle           0 0 0
                color           0 0.6 0.6 0.6
        ]
]

card [
        class                   DocumentGenerator
        responsibilities        "Creates documentation"
        collaborations          ""
        meta                    "controller"
        comment                 ""
        attributes [
                position        -725.594 520 0
                size            145 105 1
                angle           0 0 0
                color           0.6 0.6 0 0.7
        ]
]

card [
        class                   Label
        responsibilities        "Holds label"
        collaborations          "ControlGeometryBuilder [Uses], Control [ISA]"
        meta                    "viewer"
        comment                 ""
```

```
        attributes [
                position        463.537 726.535 -600
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0.8 1
        ]
]

card [
        class                   Link
        responsibilities        "Knows how to build a link\nUpdates collaborators"
        collaborations          "Window [ISA]"
        meta                    "Viewer"
        comment                 ""
        attributes [
                position        720 240 -500
                size            145 105 1
                angle           -25 0 0
                color           0.6 0 1 1
        ]
]

card [
        class                   MainMenu
        responsibilities        "Provides access to basic menus and options"
        collaborations          "Window [ISA]"
        meta                    "Viewer"
        comment                 ""
        attributes [
                position        1000 240 -500
                size            145 105 1
                angle           -25 0 0
                color           0.6 0 1 1
        ]
]

card [
        class                   Manipulator
        responsibilities        "Contrained manipulation on specific axes"
        collaborations          "ControlGeometryBuilder [Uses], Control [ISA]"
        meta                    "viewer"
        comment                 ""
```

```
        attributes [
                position        661.865 745.39 -600
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0.8 1
        ]
]

card [
        class                   Model
        responsibilities        "Holds model"
        collaborations          "Workspace [HasA]"
        meta                    "model"
        comment                 "Singleton"
        attributes [
                position        -200 -280 400
                size            160 130 100
                angle           0 0 0
                color           0 1 0 1
        ]
]

card [
        class                   ModelCard
        responsibilities        "Knows attributes of cards"
        collaborations          ""
        meta                    "model"
        comment                 ""
        attributes [
                position        -800 -600 400
                size            145 105 1
                angle           0 0 0
                color           0 0.6 0 0.8
        ]
]

card [
        class                   ModelLink
        responsibilities        "Collaborators of a link\nlabel of a link"
        collaborations          ""
        meta                    "model"
        comment                 ""
```

```
        attributes [
                position        -800 -400 400
                size            145 105 1
                angle           0 0 0
                color           0 0.6 0 0.8
        ]
]

card [
        class                   Parser
        responsibilities        " Parses crc files"
        collaborations          ""
        meta                    "controller"
        comment                 ""
        attributes [
                position        -500 680 0
                size            145 105 1
                angle           0 0 0
                color           0.6 0.6 0 0.7
        ]
]

card [
        class                   Project
        responsibilities        "Holds project\nAdds/Deletes a card\nAdds/Deletes a link"
        collaborations          "ModelLink [HasMany], ModelCard [HasMany]"
        meta                    "model"
        comment                 ""
        attributes [
                position        -500 -500 400
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0 1
        ]
]

card [
        class                   Properties
        responsibilities        "Knows properties of the tool"
        collaborations          ""
        meta                    "controller"
        comment                 ""
```

```
        attributes [
                position        -493.391 -20 0
                size            145 105 1
                angle           0 0 0
                color           0.8 0.8 0 1
        ]
]

card [
        class                   Textbox
        responsibilities        "Allows user to change text"
        collaborations          "ControlGeometryBuilder [Uses], Control [ISA]"
        meta                    "viewer"
        comment                 ""
        attributes [
                position        1200 973.963 -600
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0.8 1
        ]
]

card [
        class                   UpdateViewer
        responsibilities        "Updates viewer"
        collaborations          ""
        meta                    "controller"
        comment                 ""
        attributes [
                position        -562.607 200 0
                size            145 105 1
                angle           0 0 0
                color           0.8 0.8 0 1
        ]
]

card [
        class                   Window
        responsibilities        "Holds many controls"
        collaborations          "Container [ISA], ControlGeometryBuilder [Uses]"
        meta                    "Viewer"
        comment                 ""
```

```
        attributes [
                position        300 500 -600
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0.8 1
        ]
]

card [
        class                   WindowManager
        responsibilities        "Manages open windows"
        collaborations          "Window [HasMany]"
        meta                    "Viewer"
        comment                 ""
        attributes [
                position        50 410.946 -600
                size            150 120 50
                angle           0 0 0
                color           0 1 1 1
        ]
]

card [
        class                   WorkSpaceSaver
        responsibilities        "Saves workspace"
        collaborations          ""
        meta                    "controller"
        comment                 ""
        attributes [
                position        -494.191 360 0
                size            145 105 1
                angle           0 0 0
                color           0.6 0.6 0 0.7
        ]
]

card [
        class                   Workspace
        responsibilities        "Holds workspace\nAdd projects\nDelete projects"
        collaborations          "Project [HasMany]"
        meta                    "model"
        comment                 ""
```

```
        attributes [
                position        -200 -500 400
                size            145 105 1
                angle           0 0 0
                color           0 0.8 0 1
        ]
]

card [
        class                   crcController
        responsibilities        "Holds the controller\nApplication logic\nAdds viewer"
        collaborations          "crcControllerDispatcher [HasA], UpdateViewer [HasA], Properties [HasA], Model [H
        meta                    "controller"
        comment                 ""
        attributes [
                position        -200 200 0
                size            200 150 100
                angle           0 0 0
                color           1 1 0 1
        ]
]

card [
        class                   crcControllerDispatcher
        responsibilities        "Dispatches actions from viewer"
        collaborations          "Parser [Uses], DocumentGenerator [Uses], WorkSpaceSaver [Uses]"
        meta                    "controller"
        comment                 "Singleton"
        attributes [
                position        -200 520 0
                size            160 120 50
                angle           0 25 0
                color           0.8 0.8 0 1
        ]
]

card [
        class                   osgInputEventHandler
        responsibilities        "Informs for input events"
        collaborations          "osgMouseEventHandler [HasA], osgKeyboardEventHandler [HasA]"
        meta                    "Viewer"
        comment                 ""
```

```
        attributes [
                position        350 -240 -300
                size            145 105 1
                angle           0 0 0
                color           1 0.3 0.3 1
        ]
]

card [
        class                   osgKeyboardEventHandler
        responsibilities        "Informs for keyboard evetns"
        collaborations          ""
        meta                    "Viewer"
        comment                 ""
        attributes [
                position        550 -500 -300
                size            173.396 105 1
                angle           0 0 0
                color           1 0.3 0.3 1
        ]
]

card [
        class                   osgMouseEventHandler
        responsibilities        "Informs for mouse events"
        collaborations          ""
        meta                    "Viewer"
        comment                 ""
        attributes [
                position        250 -500 -300
                size            152.378 105 1
                angle           0 0 0
                color           1 0.3 0.3 1
        ]
]

card [
        class                   osgViewer
        responsibilities        "Knows how to render"
        collaborations          "osgInputEventHandler [HasA], osgViewerDispatcher [HasA], WindowManager [HasA]"
        meta                    "Viewer"
        comment                 ""
```

```
        attributes [
                position        254.958 0 -400
                size            200 160 100
                angle           0 0 0
                color           0 0 1 1
        ]
]

card [
        class                   osgViewerDispatcher
        responsibilities        "Handles internal viewer actions\nHandles controller actions"
        collaborations          "CRCCard [Uses], Link [Uses], MainMenu [Uses]"
        meta                    "viewer"
        comment                 "Singleton"
        attributes [
                position        650 0 -400
                size            160 120 50
                angle           0 25 0
                color           0 0 0.8 1
        ]
]

link Button Control "[ISA]"
link Button ControlGeometryBuilder "[Uses]"
link CRCCard Link "[HasMany]"
link CRCCard Window "[ISA]"
link Checkbox Control "[ISA]"
link Checkbox ControlGeometryBuilder "[Uses]"
link Container Control "[ISA]"
link Label Control "[ISA]"
link Label ControlGeometryBuilder "[Uses]"
link Link Window "[ISA]"
link MainMenu Window "[ISA]"
link Manipulator Control "[ISA]"
link Manipulator ControlGeometryBuilder "[Uses]"
link Model Workspace "[HasA]"
link Project ModelCard "[HasMany]"
link Project ModelLink "[HasMany]"
link Textbox Control "[ISA]"
link Textbox ControlGeometryBuilder "[Uses]"
link Window Container "[ISA]"
link Window ControlGeometryBuilder "[Uses]"
```

```
link WindowManager Window "[HasMany]"
link Workspace Project "[HasMany]"
link crcController Model "[HasA]"
link crcController Properties "[HasA]"
link crcController UpdateViewer "[HasA]"
link crcController crcControllerDispatcher "[HasA]"
link crcController osgViewer "[HasA]"
link crcControllerDispatcher DocumentGenerator "[Uses]"
link crcControllerDispatcher Parser "[Uses]"
link crcControllerDispatcher WorkSpaceSaver "[Uses]"
link osgInputEventHandler osgKeyboardEventHandler "[HasA]"
link osgInputEventHandler osgMouseEventHandler "[HasA]"
link osgViewer WindowManager "[HasA]"
link osgViewer osgInputEventHandler "[HasA]"
link osgViewer osgViewerDispatcher "[HasA]"
link osgViewerDispatcher CRCCard "[Uses]"
link osgViewerDispatcher Link "[Uses]"
link osgViewerDispatcher MainMenu "[Uses]"

project FlyingCircus ends
```

Appendix B

Example of documentation generation

```
==> Flying Circus automatically generated documentation <==
-----------------------------------------------------------

Workspace FlyingCircus

Total projects (1):
        Project FlyingCircus

Project "FlyingCircus"

Classes Overview (30):
        Button
        CRCCard
        Checkbox
        Container
        Control
        ControlGeometryBuilder
        DocumentGenerator
        Label
        Link
        MainMenu
        Manipulator
        Model
        ModelCard
        ModelLink
```

```
        Parser
        Project
        Properties
        Textbox
        UpdateViewer
        Window
        WindowManager
        WorkSpaceSaver
        Workspace
        crcController
        crcControllerDispatcher
        osgInputEventHandler
        osgKeyboardEventHandler
        osgMouseEventHandler
        osgViewer
        osgViewerDispatcher

Links (38):
        Button              [ISA]               Control
        Button              [Uses]              ControlGeometryBuilder
        CRCCard             [HasMany]           Link
        CRCCard             [ISA]               Window
        Checkbox            [ISA]               Control
        Checkbox            [Uses]              ControlGeometryBuilder
        Container           [ISA]               Control
        Label               [ISA]               Control
        Label               [Uses]              ControlGeometryBuilder
        Link                [ISA]               Window
        MainMenu            [ISA]               Window
        Manipulator         [ISA]               Control
        Manipulator         [Uses]              ControlGeometryBuilder
        Model               [HasA]              Workspace
        Project             [HasMany]           ModelCard
        Project             [HasMany]           ModelLink
        Textbox             [ISA]               Control
        Textbox             [Uses]              ControlGeometryBuilder
        Window              [ISA]               Container
        Window              [Uses]              ControlGeometryBuilder
        WindowManager       [HasMany]           Window
        Workspace           [HasMany]           Project
        crcController       [HasA]              Model
        crcController       [HasA]              Properties
```

```
        crcController       [HasA]              UpdateViewer
        crcController       [HasA]              crcControllerDispatcher
        crcController       [HasA]              osgViewer
        crcControllerDispatcher[Uses]              DocumentGenerator
        crcControllerDispatcher[Uses]              Parser
        crcControllerDispatcher[Uses]              WorkSpaceSaver
        osgInputEventHandler[HasA]              osgKeyboardEventHandler
        osgInputEventHandler[HasA]              osgMouseEventHandler
        osgViewer           [HasA]              WindowManager
        osgViewer           [HasA]              osgInputEventHandler
        osgViewer           [HasA]              osgViewerDispatcher
        osgViewerDispatcher [Uses]              CRCCard
        osgViewerDispatcher [Uses]              Link
        osgViewerDispatcher [Uses]              MainMenu

Classes Detailed (30):
        Button
                Responsibilities: Fire action when pressed
                Comment:
                Meta: viewer
                [ISA] Control
                [Uses] ControlGeometryBuilder

        CRCCard
                Responsibilities: Knows to build a CRC card\nChange properties of CRC card
                Comment:
                Meta: Viewer
                [HasMany] Link
                [ISA] Window

        Checkbox
                Responsibilities: Allows user to enable or disable option
                Comment:
                Meta: viewer
                [ISA] Control
                [Uses] ControlGeometryBuilder

        Container
                Responsibilities: Contains controls
                Comment:
                Meta: Viewer
                [ISA] Control
```

```
Control
        Responsibilities: Knows id\nKnows properties
        Comment:
        Meta: Viewer

ControlGeometryBuilder
        Responsibilities: Builds geometry of controls
        Comment:
        Meta: Viewer

DocumentGenerator
        Responsibilities: Creates documentation
        Comment:
        Meta: controller

Label
        Responsibilities: Holds label
        Comment:
        Meta: viewer
        [ISA] Control
        [Uses] ControlGeometryBuilder

Link
        Responsibilities: Knows how to build a link\nUpdates collaborators
        Comment:
        Meta: Viewer
        [ISA] Window

MainMenu
        Responsibilities: Provides access to basic menus and options
        Comment:
        Meta: Viewer
        [ISA] Window

Manipulator
        Responsibilities: Contrained manipulation on specific axes
        Comment:
        Meta: viewer
        [ISA] Control
        [Uses] ControlGeometryBuilder
```

```
Model
        Responsibilities: Holds model
        Comment: Singleton
        Meta: model
        [HasA] Workspace

ModelCard
        Responsibilities: Knows attributes of cards
        Comment:
        Meta: model

ModelLink
        Responsibilities: Collaborators of a link\nlabel of a link
        Comment:
        Meta: model

Parser
        Responsibilities:  Parses crc files
        Comment:
        Meta: controller

Project
        Responsibilities: Holds project\nAdds/Deletes a card\nAdds/Deletes a link
        Comment:
        Meta: model
        [HasMany] ModelCard
        [HasMany] ModelLink

Properties
        Responsibilities: Knows properties of the tool
        Comment:
        Meta: controller

Textbox
        Responsibilities: Allows user to change text
        Comment:
        Meta: viewer
        [ISA] Control
        [Uses] ControlGeometryBuilder

UpdateViewer
        Responsibilities: Updates viewer
```

```
        Comment:
        Meta: controller

Window
        Responsibilities: Holds many controls
        Comment:
        Meta: Viewer
        [ISA] Container
        [Uses] ControlGeometryBuilder

WindowManager
        Responsibilities: Manages open windows
        Comment:
        Meta: Viewer
        [HasMany] Window

WorkSpaceSaver
        Responsibilities: Saves workspace
        Comment:
        Meta: controller

Workspace
        Responsibilities: Holds workspace\nAdd projects\nDelete projects
        Comment:
        Meta: model
        [HasMany] Project

crcController
        Responsibilities: Holds the controller\nApplication logic\nAdds viewer
        Comment:
        Meta: controller
        [HasA] Model
        [HasA] Properties
        [HasA] UpdateViewer
        [HasA] crcControllerDispatcher
        [HasA] osgViewer

crcControllerDispatcher
        Responsibilities: Dispatches actions from viewer
        Comment: Singleton
        Meta: controller
        [Uses] DocumentGenerator
```

```
        [Uses] Parser
        [Uses] WorkSpaceSaver

osgInputEventHandler
        Responsibilities: Informs for input events
        Comment:
        Meta: Viewer
        [HasA] osgKeyboardEventHandler
        [HasA] osgMouseEventHandler

osgKeyboardEventHandler
        Responsibilities: Informs for keyboard evetns
        Comment:
        Meta: Viewer

osgMouseEventHandler
        Responsibilities: Informs for mouse events
        Comment:
        Meta: Viewer

osgViewer
        Responsibilities: Knows how to render
        Comment:
        Meta: Viewer
        [HasA] WindowManager
        [HasA] osgInputEventHandler
        [HasA] osgViewerDispatcher

osgViewerDispatcher
        Responsibilities: Handles internal viewer actions\nHandles controller actions
        Comment: Singleton
        Meta: viewer
        [Uses] CRCCard
        [Uses] Link
        [Uses] MainMenu
```

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Boston, 1986.

[2] T. Akenine-Möller and E. Haines. *Real-Time Rendering*. A K Peters, Natick, Massachusetts, second edition, 2002.

[3] Aristotle. On the soul. *Aristotle*, 8:1–203, 1936.

[4] W. Ark, D. C. Dryer, T. Selker, and S. Zhai. Representation matters: The effect of 3d objects and a spatial metaphor in a graphical user interface. In *People and Computers XIII, Proc of HCI'98*, pages 209–219, 1998.

[5] D. J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[6] R. Balakrishnan and G. Kurtenbach. Exploring bimanual camera control and object manipulation in 3d graphics interfaces. In *CHI'99: Proceedings of the SIGCHI conference of Human factors in computing systems*, pages 56–62, New York, NY, USA, 1999. ACM Press.

[7] M. Beaudouin-Lafon. Designing interactions, not interfaces. In *Proceedings of the working conference on Advanced visual interfaces*, pages 15–22, New York, NY, USA, 2004. ACM Press.

[8] K. Beck. Crc: Finding objects the easy way. *Object magazine*, 3(4):42–44, 1993.

[9] K. Beck and W. Cunningham. A laboratory for teaching object-orient thinking. In *OOPSLA '89 Conference Proceedings*, New Orleans, Louisiana, 1989.

[10] D. Bellin and S. S. Simone. *The CRC card book*. Addison-Wesley, Reading, Massachusetts, 1997.

[11] R. Biddle, J. Noble, and E. Tempero. *Reflections on CRC Cards and OO Design*. Cambridge University Press, 2002.

[12] J. Biström, A. Cogliati, and K. Rouhiainen. Post-wimp user interface model for 3d web applications. Seminar: Research Seminar on Digital Media, December 2005.

[13] A. Bohne-Lang and E. Lang. Java applets for displaying 3d molecule structures ? an overview, 2004.

[14] G. Booch. *Object Oriented Analysis and Design with Applications*. Benjamin/Cummings, Redwood City, 1994.

[15] J. Börstler. Classes or objects? crc-cards considered harmful. In *1st Scandinavian Pedagogy of Programming Workshop*, 2004.

[16] J. Börstler. Object oriented analysis and design through scenario role play. Technical report, Department of Computing Science, Umeå University, Sweden, April 2004.

[17] J. Börstler. Improving crc-card role-play with role-play diagrams. In *ompanion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 356–364, New York, 2005. ACM Press.

[18] J. Börstler, T. Johansson, and M. Nordström. Teaching oo concepts - a case study using crc cards and bluej. In *32nd ASEE/IEEE Frontiers in Education Conference*, 2002.

[19] D. Bowman, E. Kruijff, and I. Poypyrev. *3D User Interfaces: Theory and Practice*. Addison Wesley, Boston, 2004.

[20] D. A. Bowman, J. Chen, C. A. Wingrave, J. Lucas, A. Ray, N. F. Polys, Q. Li, J. S. Kim Y. Haciahmetoglu, R. Boehringer, and T. Ni. New directions in 3d user interfaces. *The international Journal of Virtual Reality*, 5(2):3–14, 2006.

[21] S. Brunstad and T. F. Eie. Adding a d to 2d - a better interface?, 2002.

[22] B. G. Cain and J. O. Coplien. A role-based empirical process modeling environment. In *Proceedings of the Second International Conference on the Software Process*, pages 125–133. IEEE Computer Society Press, 1993.

[23] R. Carey and G. Bell. *The Annotated VRML 97 Reference Manual*. Addison-Wesley, Boston, 1997.

[24] A. Cockburn. Revisiting 2d vs 3d implications on spatial memory. In *Proceedings of the fifth conference on Australasian user interface*, volume 53, pages 25–31, Dunedin, New Zealand, 2004.

[25] E. Cuppens, C. Raymaekers, and K. Coninx. Vrixml: A user interface description language for virtual environments. In *Proceedings of the ACM AVI'2004 Workshop, Developing User Interfaces with XML: Advances on User Interface Description Languages*, pages 111–118, Gallipoli, 2004.

[26] H. L. Dershem and M. J. Jipping. *Programming Languages: Structures and Models*. PWS Publishing Company, Boston, 1995.

[27] C. Everitt. Interactive order-independent transparency. Technical report, NVIDIA, April 2001.

[28] M. Fayad, H. Sánchez, and H. Hamza. A pattern language for crc cards. In *The 11th Conference On Pattern Languages Of Programs*, 2004.

[29] K. J. Fernandes, V. Raja, and J. Eyre. Cybersphere: the fully immersive spherical projection system. *Communications of the ACM*, 46(9):141–146, 2003.

[30] J. D. Folley, A. van Dam, and S. K. Feiner. *Computer Graphics: Principles and Practice in C*. Addison-Wesley, New York, NY, 1995.

[31] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patters: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Boston, 1995.

[32] D. Grammenos, M. Filou, P. Papadakos, and C. Stephanidis. Virtual prints: Leaving trails in virtual environents. In *Proceedings of the Eight Eurographics Workshop on Virtual Environments (VE2002)*, pages 131–138, 2002.

[33] K. A. Gray, M. Guzdial, and S. Rugaber. Extending crc cards into a complete design process. In *Proceedings of the 8th annual conference on Innovation and technology in computer science education*, 2003.

[34] A. J. Hanson and E. A. Wernert. Constrained 3d navigation with 2d controllers. In *Proceedings of the 8th conference on Visualization 97*, pages 175–ff, Phoenix, Arizona, United States, 1997.

[35] B. Henderson-Sellers, A. Simons, H. Younessi, and I. S. Graham. *The Open Toolbox of Techniques*. Addison-Wesley, 1998.

[36] K. P. Herndon, A. van Andries, and M. Gleicher. The challenges of 3d interaction. In *CHI'94 workshop*, volume 26, pages 36–43, 1994.

[37] K. Hinckley, J. Tullio, R. Pausch, D. Proffitt, and N. Kassell. Usability analysis of 3d rotation techniques. In *Proceedings of UIST 97*, Alberia, Canada, 1997.

[38] R. W. Hollingworth and C. McLoughline. Developing science students' metacognitive problem solving skills online. *Australian Journal of Educational Technology*, 17(1):50–63, 2001.

[39] A. Jaaksi. Implementing interactive applications in c++. *Software Practice & Experience*, 25(13):271–289, 1995.

[40] A. Jaaksi, J. M. Aalto, A. Alto, and K. Vättö. *Tried and True Object Development, Industry Proven Approaches with UML*. Cambridge University Press, 1999.

[41] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object-Oriented Programming (JOOP)*, 1988.

[42] P. Karampelas, D. Grammenos, A. Mourouzis, and C. Stephanidis. Towards i-dove, an interactive support tool for building and using virtual environments with guidelines. In *Proceedings of the 10th International Conference on Human-Computer Interaction (HCI International 2003)*, pages 1411–1415, Mahwah, New Jersey, 2003. Lawrence Erlbaum Associates.

[43] G. E. Krasner and S. E. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *Journal of Object-Oriented Programming (JOOP)*, 1988.

[44] G. E. Krasner and S. E. Pope. A description of the model-view-controller user interface paradigm in the smalltalk-80 system. *Journal of Object Oriented Programming*, 1(3):26–49, 1988.

[45] J. Laird. *Hobbes*. Russell & Russell, New York, 1968.

[46] J. Locke. *An Essay Concerning Human Understanding*. Hackett Publishing Company, Indianapolis, 1996.

[47] P. Martz. *OpenSceneGraph Quick Start Guide, A quick introduction to the Cross-platform open source scene graph API*. Skew Matrix Software, Mountain View, California, 2007.

[48] K. McMenemy and S. Ferguson. *A Hitchhiker's Guide to Virtual Reality*. A.K. Peters Ltd., 2007.

[49] L. R. Novick and S. J. Sherman. On the nature of insight solutions: Evidence from skill differences in anagram solution. *Quarterly Journal of Experimental Psychology*, 56A:351–382, 2003.

[50] D. L. Odell, D. Richard, S. Andrew, and P. K. Wright. Toolglasses marking menus, and hotkeys: a comparison of one and two-handed command selection techniques. In *GI'04: Proceedings of the 2004 conference on Graphics interface*, pages 17–24, Waterloo, Ontario, Canada, 2004. School of Computer Science, University of Waterloo.

[51] A. F. Osborn. *Applied imagination: Principles and procedures of creative problem solving*. Charles Scribner's Sons, New York, NY, third edition, 1963.

[52] N. Partarakis, A. Mourouzis, C. Doulgeraki, and C. Stephanidis. A portal-based tool for developing, delivering and working with guidelines. In *Proceedings of the 12th International Conference on Human-Computer Interaction (HCI International 2005)*, pages 507–516, 2007.

[53] A. Raman and S. Tyszberowicz. The easycrc tool. In *Proceedings of the International Conference on Software Engineering Advances(ICSEA 2007)*, 2007.

[54] B. Reeves and C. Nass. *The Media Equation*. Cambridge University Press, Cambridge, 1996.

[55] S. Roach and J. C. Vásquez. A tool to support the crc design method. In *Proceedings of the International Conference on Engineering Education*, 2004.

[56] G. Robertson, M. Czerwinski, and M. van Dantzich. Immersion in desktop virtual reality. In *Proceedings of 10th annual ACM symposium on User Interface software and technology*, 1997.

[57] D. M. Rubin. Introduction to crc cards. Technical report, SoftStar Research, Inc., 1998.

[58] W. R. Sherman and A. B. Craig. *Understanding Virtual Reality, Interface, Application and Design*. Morgan Kaufmann Publishers, San Francisco, 2003.

[59] M. Slater, V. Linakis, M. Usoh, and R. Kooper. Immersion, presence, and performance in virtual environments: An experiment using tri-dimensional chess.

[60] R. J. Sternberg. *Cognitive Psychology*. Wadsworth Publishing, fourth edition, 2005.

[61] M. Tavanti and R. Williges. 2d vs 3d, implications on spatial memory. In *Proceedings of IEEE InfoVis 2001 Symposium on Information Visualization, San Diego*, San Diego, 2001.

[62] E. L. Thorndike. Animal intelligence: An experimental study of the associative processes in animals. *Psychological Review Monograph Supplement*, 2:1–109, 1901.

[63] A. van Dam. Post-wimp user interfaces. *Commun. ACM*, 40(2):63–67, 1997.

[64] N. G. Vinson. Design guidelines for landmarks to support navigation in virtual environments. In *CHI'99 Proc.*, pages 278–285, 1999.

[65] Wikipedia. Wikipedia, the free encyclopedia, 2007. [Online; accessed October 2007].

[66] N. M. Wilkilson. *Using CRC cards - An informal Approach to Object-Oriented Development*. Cambridge University Press, 1996.

[67] J. R. Wilson. From potential to practice: Virtual and interactive environments in workplaces of the future(view). In *In Proc. of Virtual Reality International Conference (VRIC 2002)*, 2002.

[68] R. Wirfs-Brock and A. McKean. *Object Design-Roles, Responsibilities, and Collaborations*. Addison-Wesley, Boston, 2003.

[69] R. Wirfs-Brock and B. Wilkerson. Object oriented design: A responsibility-driven approach. In *OOPSLA '89 Conference Proceedings*, pages 71–75, New Orleans, Louisiana, 1989.

[70] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing object-oriented software*. Prentice-Hall, Englewood Cliffs, 1990.

[71] H. Wurnig. Design of a collaborative multi user desktop system for augmented reality. In *Proceedings of the CESCG '98*, 1998.