



University of Nairobi  
School of Computing & Informatics

**3D Visualization Program  
For  
Executable Code Analysis**

by  
Peter Mulwa  
P58/63309/2011

Supervisor  
Dr. Tonny Omwansa

2013

A project in partial fulfillment of requirement of Master in Computer Science  
University of Nairobi

## **ABSTRACT**

Information Technology is pervasive and new ways to leverage its potential are continuously emerging. A resultant effect is the increase in the code base as new functionality is implemented. And as computers increasingly handle and process more information, the analysis of executables becomes necessary.

Visual Analytics of executable code provides a tool to analyze their structured format, providing an alternative tool comparable to directly analyzing source code, to generate meaningful information. Visualization of the software enhances this process by providing the visual metaphors that represent the code aspects.

Various visual representations have been utilized in visualizing the various aspects of software. This research presents a visual interface for interacting with Binary Code, illustrating the potential of basic geometric shapes and visual interaction in understanding the structure of programs. It proposes that directly manipulating the software structure, with an abstracted visual representation, provides an improved understanding of a program.

The process involved the design and development of a prototype application of a 3D environment within which interactions with visual metaphors enabled visualization and analysis of Binary Code. The key metaphor utilized is based on the lattice structure.

The resultant application provided a 3D visualization environment within which binary code could be analyzed using a lattice-based metaphor. The application provided functionality for visually interacting with disassembled code as well as querying the code and visually viewing the results within the metaphor. The research could provide a basis for research and application of visual reverse engineering in an environment of touch screens and increasing processing capability.

Key words: Binary Code, Visualization, Lattice, Metaphor, Instruction Set Architecture

## **DECLARATION**

I declare that this research project report is my original work and has not been submitted for academic reward in any institution.

Peter Mulwa

Signature:

Date:

This research project report has been submitted with my approval as supervisor.

Dr. Tonny Omwansa

Signature:

Date:

## **ACKNOWLEDGEMENTS**

I like to thank Dr. Omwansa for his guidance and critique from the initial stages of proposing potential projects to finally settling on a project and seeing it to its conclusion. I'm grateful for the opportunity, patience, and encouragement during the endeavour.

I would also like to thank Mr. Moturi for his pace setting and encouragement to begin and complete the project. I appreciate your availability and guidance during the entire degree program.

Thanks also go to my panel of Dr. Wausi, Dr Nganga, and Dr. Orwa, for their input, direction, and asking the questions needed to refine the project. I really appreciate the encouragement you gave.

Thank you Mum.

Thank you Jesus.

## TABLE OF CONTENTS

List of Figures	vi
List of Tables	ix
List of Code Listings	x
Chapters	
1. Introduction	1
I. Background	1
II. Research Basis; Problem Statement and Purpose of Project	2
III. Research Outcome and their Significance to Key Audience	2
IV. Research Objectives	3
V. Research Scope, Limitations, Assumptions, and Complexity	3
VI. Research Justification	4
VII. Definitions of Important Terms	5
VIII. Subsequent Chapters	5
2. Literature Review	6
I. Overview	6
II. Guiding Concepts (Scope)	6
III. Literature Survey (Common Ground)	8
IV. Case For Research (Literature Gaps)	13
V. Conclusion	15
3. System Design	17
I. Introduction	17
II. Development	18
III. Requirements	20
IV. Conceptual	20
V. Data	25
VI. Import Format	26
VII. Content	29
VIII. Metaphor	38

IX.	Construct	45
X.	Visual	62
4.	Results	69
I.	Introduction	69
II.	Test Program	69
III.	Visualization & Analysis	72
5.	Conclusion	82
I.	Introduction	82
II.	Achievements	82
III.	Findings – Review of Research Objectives	85
IV.	Contribution – Addressing of Perceived Gaps	85
V.	Evaluation	87
VI.	Recommendation	89
VII.	Summary	89
	References	90
	Appendices	
A.	Development Platform	93
B.	User Manual	94
C.	Usability Testing	95

## LIST OF FIGURES

<b>Figure</b>	<b>Description</b>	<b>Page</b>
2.1	Code Structuring & Abstraction	7
2.2	Visualization (Code Aspect to Visual Metaphor Mapping)	8
2.3	Literature survey metaphors	13
2.4	Framework for Visual Analytics	15
3.1	Prototype Development Cycle	18
3.2	Interaction Flow	21
3.3	Content Generation	22
3.4	Disassembly of Notepad.exe	28
3.5	Data Files	29
3.6	Data Engine	30
3.7	.ld File Content	32
3.8	.ldl File Content	33
3.9	.ldr File Content	34
3.10	.le File Content	36
3.11	.lel File Content	37
3.12	.ler File Content	37
3.13	Sequential & Control instructions	38
3.14	Runtime dependency	38
3.15	Metaphor representation	39
3.16	Enhanced metaphor representation	39
3.17	Display of branching information	40
3.18	Folding node sequences	41
3.19	Section format	42
3.20	Navigation	43
3.21	Notation	44
3.22	Sequential flow	45
3.23	Control flow	45
3.24	Basic program	47

3.25	Disassembled Basic Program	47
3.26	Visualized Basic Program	47
3.27	‘if’ statement	48
3.28	Disassembled ‘if’ Program	49
3.29	Visualized ‘if’ Program	49
3.30	‘switch’ statement	50
3.31	Disassembled ‘switch’ Program	51
3.32	Visualized ‘switch’ Program	51
3.33	‘do’ statement	52
3.34	Disassembled ‘do’ Program	53
3.35	Visualized ‘do’ Program	53
3.36	‘while’ statement	54
3.37	Disassembled ‘while’ Program	55
3.38	Visualized ‘while’ Program	55
3.39	‘for’ statement	56
3.40	Disassembled ‘for’ Program	57
3.41	Visualized ‘for’ Program	57
3.42	‘continue’ statement	58
3.43	Disassembled ‘continue’ Program	59
3.44	Visualized ‘continue’ Program	59
3.45	‘break’ statement	60
3.46	Disassembled ‘break’ Program	61
3.47	Visualized ‘break’ Program	61
3.48	Graphics Engine	62
3.49	Startup Screen	63
3.50	Visualization Screen	63
3.51	Content Generation	64
3.52	Visualization Screen Layout	64
3.53	Visualization Screen Layout Components	65
3.54	Prototype – Initial Screen Layout	68
3.55	Prototype – Initial Screen Animation	68

4.1	Disassembled Test Program	71
4.2	Test Program Visualization	72
4.3	Navigation & Next Location Highlight	74
4.4	Potential Source Locator	75
4.5	Loops	77
4.6	Potential Endless Loop	79
4.7	Function Call Mapping	80

## LIST OF TABLES

<b>Table</b>	<b>Description</b>	<b>Page</b>
1.1	Word processor sizes	4
2.1	Recurring themes in literature	12
2.2	Literature survey summary	12
2.3	Perceived gaps in literature	15
3.1	External Components	23
3.2	File / Program Entry Points	23
3.3	Key Processes	24
3.4	Content Files	24
3.5	Format for 1 <sup>st</sup> line of an instruction	27
3.6	Format for 2 <sup>nd</sup> & subsequent lines of instructions	28
3.7	.ld File Format	31
3.8	.ldl File Format	33
3.9	.ldr File Format	34
3.10	.le File Format	35
3.11	.lel File Format	36
3.12	.ler File Format	37
4.1	Test Program (1 <sup>st</sup> 8 nodes; 1 <sup>st</sup> section)	72
B.1	Keyboard Commands	94

## LIST OF CODE LISTINGS

<b>Code Listing</b>	<b>Description</b>	<b>Page</b>
3.1	Basic program	47
3.2	'if' statement	48
3.3	'switch' statement	50
3.4	'do' statement	52
3.5	'while' statement	54
3.6	'for' statement	56
3.7	'continue' statement	58
3.8	'break' statement	60
4.1	Test program source code	78

## CHAPTER 1 - INTRODUCTION

### 1.1 Background

Information Technology is pervasive and new ways to leverage its potential are continuously emerging. In this innovation process, software provides a tool for implementing new functionality, with the potential effect of increasing the existing code base.

Review of this code base is difficult for several reasons. Program complexity increases as new functionality is implemented. Large teams are involved in the development process. Different software development tools utilized provide unique syntax and means of expressing semantics. Publishers usually retain the source code; however, executables are released for deployment.

In varied scenarios, executable files need to be reverse engineered in order to understand their functionality. Disassembling an executable provides a human-readable format that resembles the underlying machine code due to the one-to-one mapping of machine & assembly code. Dependent on the size of the executable, the quantity of the information generated can be large. This makes the analysis of information potentially difficult. Besides textually viewing the content, visualization can be utilized to enhance the process of understanding and analyzing the content.

Lattices provide a potentially useful structure that can be adapted to develop a visual metaphor that can be used to visualize & analyze a program's disassembled executable code in order to generate usable information to aid in decision making.

This research project presents a development of a lattice-based metaphor for this purpose. It begins by abstracting a generic platform's Instruction Set Architecture (ISA). Rules are then formulated on how to represent the different combination of instructions in order to enable adaptation to a lattice structure. A notation for displaying information is developed.

Various basic code constructs dealing with branching and looping are then illustrated by a process of abstracting their structural design and then visualizing them using the metaphor on the basis that these constructs are combined in various ways to constitute a program.

## **1.2 Research Basis; Problem Statement and Purpose of Project**

Executables, with their structured format, provide an avenue (at times possibly the only means) of understanding an actual program's design and functionality. And as computers increasingly handle and process more information, the analysis of executables becomes even more necessary.

Visual Analytics of executable code provides a tool for analysis of the structured format of executables. By extracting and visually presenting information, an alternative form of analysis is possible that enables abstraction and interaction of underlying, potentially, complex concepts.

The problem can be summarized as 'extracting and visually presenting information on executable code for analysis and abstraction of underlying concepts'.

This research attempts to design and implement a 3-Dimensional visual metaphor and interaction environment for analyzing disassembled binary code on the basis of the structure of its corresponding Instruction Set Architecture.

## **1.3 Research Outcomes and their Significance to Key Audience**

Various contributing parameters are increasing the need for executable code analysis; they include:

- Executable code is increasing both in quantity and complexity, as it becomes a critical component of modern infrastructure.
- Computing power is increasing, and with the rise of Graphic Processing Units, visual analysis of large quantities of information (programs in this case) is feasible.
- Visual Analytics provides a tool to quickly analyze large code bases in an interactive manner, enabling quicker identification of pertinent areas of interest in less time and with less effort.

The research endeavour aims at providing a 3D visualization environment with associated metaphors that will enable executable binary code analysis in a graphical manner.

Visualization of binary code could enhance the efficiency of the analysis of programs in software domains where the source code is not usually available. This largely occurs in the software security domain with potential fields of use being reverse code engineering, vulnerability research, and malware analysis.

Potential users of the system include:

- Software Engineers maintaining programs without the source code
- Malware Researchers analyzing malware to determine its structure for signature detection
- Security Researchers reverse engineering programs to identify potential flaws

#### **1.4 Research Objectives**

1. To design & develop a prototype 3D Visualization application for Binary Code Analysis.
2. To design & develop a 3D Lattice based visual metaphor to represent Binary Code

#### **1.5 Research Scope, Limitations, Assumptions, and Complexity**

The platform selected for implementation and analysis is the Microsoft-Intel platform with its associated Portable Executable file format and assembly language. This is due to its ubiquity and existing large code base; however, the results are extendable to other platforms.

The research is limited to the Portable Executable File Format, from which further research into the executable code is possible. The file format will provide the ideal entry point.

Code obfuscation complicates the process of analyzing an executable. The research assumes that no attempts are made to complicate the program structure.

The level of complexity and/or size of the programs to be visualized are based on real-world mainstream applications. Examples include the individual applications of the Microsoft Office Suite. For example, the Microsoft Word 97 binary size is approximately 8.5Mb, its assembly code listing approximately 148Mb, and when viewed with single line spacing of font size 10 approximately 32,700 pages of 2.7 million lines of assembly code.

Table 1.1: File size analysis for listed word processors.

<b>Executable</b>	<b>Binary</b>	<b>Assembly</b>	<b>Pages (Lines) (see note below)</b>
Notepad (Windows 2000)	50KB	488KB	150 (~9,000)
Wordpad (Windows 2000)	181KB	2.5MB	830 (~47,000)
WinWord (Office 97)	8.5MB	148MB	32700+ (2.7million+)

Note: Pages format (plain text, font size 10, single line spacing)

## 1.6 Research Justification

Due to the growing complexity of software and its functionality, and the need to analyze the growing quantity of code in shorter timeframes, visual analysis provides a potential intuitive and interactive tool.

Software domains that would benefit from visual analysis include software engineering, reverse engineering, malware research, security research, and vulnerability research.

A common trend in the above specified domains is the availability of only the executable code; the source code is usually retained by the developers and not released publicly. In addition, software is readily available in its executable form. Disassemblers and decompilers exist which attempt to convert the binary code to some high level form. Disassemblers provide a more accurate representation due to the one-to-one mapping between assembly code and machine code. However, analysis of assembly code is not intuitive.

3D visualization provides a graphical alternative to the textual viewing of this code. By generating a concise visual representation of the underlying binary code, an alternative form of interaction with code is possible, and potentially ideal because,

- Software is large, complex, and continuously evolves
- Visual processing is more intuitive than textual
- 3D visualization increases the spatial space for analysis; there are limits to font shrinkage, screen resolution, screen monitor sizes, or use of multiple monitors
- Large quantities of information can be analyzed in shorter time frames

- Various aspects can be analyzed and filtered; presented on demand or using different illustrations, and content can be visually anchored to reduce cognitive load
- Design of a metaphor for binary code enables software visualization to utilize illustrations tailored specifically for code rather than borrowing information visualization metaphors.

### **1.7 Definitions of Important Terms**

Several terms occur within this report as well as in the related literature. These are briefly described below.

- Metaphor – refers to a visual representation of an underlying code concept
- Aspect – refers to an attribute of code
- Anchor – refers to a cognitive reference point used during analysis

### **1.8 Subsequent Chapters**

Content covered in later chapters includes literature review, system design, and results.

## **CHAPTER 2 – LITERATURE REVIEW**

### **2.1 Overview**

This chapter builds a case for the research endeavour outlined in the previous chapter. It consists of the following sections:

- Guiding Concepts, which provide the scope (Section 2.2)
- Literature Survey, which describes the common ground across the literature (Section 2.3)
- Case For Research, which discusses the perceived gaps in the literature (Section 2.4)

### **2.2 Guiding Concepts (Scope)**

The guiding concepts are used to provide the scope for the research endeavour. They are:

- Screen View
- Program Structure, Control Flow
- Cognitive Dimension
- Visualization, Visual Analytics

They are connected in the following manner: utilization of visualization / visual analytics enhanced by cognitive dimensions to analyze program structure / control flow on a computing device's screen via 3D features and functionality.

#### **2.2.1 Screen View**

Information generated from an analysis process on a computing device is usually displayed on its screen. However, several challenges are faced when attempting to obtain an overall view of the data presented. Aspects such as quantity & type of content, nature of analysis tool, and the screen view available can affect the analysis of information.

Limitations are placed on the extent of font shrinkage, screen resolution adjustment, screen size, or use of multiple screens.

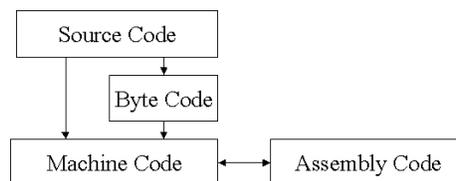
### 2.2.2 Program Structure, Control Flow

Different categories of tools are available for executable program analysis providing varied levels of control and information. Examples include disassemblers & decompilers, debuggers, hex editors, and filters.

However, these tool categories are limited in their capability of providing an initial overall view of the content they analyze due to their textual nature. For example, identifying various control-flows (such as back, critical, abnormal, or impossible) is not easily intuitive. Visualization could provide a global view starting point for visually drilling in, with focused analysis that would cause unreachable code from the current view to be selectively hidden or automatically removed.

Executable code is increasing in both quantity and complexity, providing a potential software domain that requires improved and new ways of handling and processing its content.

Figure 2.1: Code is already structured.



Note:

- Source code is converted directly to machine code for execution via a compiler, or into intermediate byte code that is executed via an interpreter.
- There exists a one-to-one mapping between machine code instructions and assembly code mnemonics.

### 2.2.3 Cognitive Dimension (CD)

CDs are guiding principles used in design, usually of user interfaces and notations, enabling heuristic evaluation of new or existing information artifacts (interactive e.g. applications or non-interactive e.g. graphs). They provide a lightweight approach in the evaluation of a design space, without in-depth analysis due to the existence of tradeoffs, to identify usability issues.

Several CDs are utilized in the research, namely abstraction (synopsis of an annotated structure), diffuseness/terseness (notation space required to provide meaning), secondary notation (extra information carried by notation), and visibility (ease of identifying notation parts).

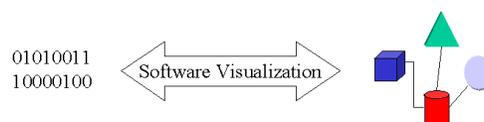
#### 2.2.4 Visualization, Visual Analytics

Visualization provides a graphical tool for analyzing large quantities of data to identify patterns, relations, and structures. Visual Analytics utilizes interactive interfaces to aid reasoning (detect the expected and discover the unexpected).

Software visualization deals with the mapping between visual metaphors and code aspects. Executable code with its structure, though abstract, lends itself to visual analysis to improve program understanding.

In order to achieve the benefits of visual analytics of executable code, the following need to be addressed: the attributes of both metaphors & aspects, the metaphor representation, the information to be required to be derived from the massive content, efficient management of the information derived, and performance enhancements. A methodology also needs to be incorporated to guide the process.

Figure 2.2: Visualization mapping of code aspects and visual metaphors



Note:

- Software visualization maps executable code comprising of 0s and 1s to graphical representations.

### 2.3 Literature Survey (Common Ground)

This section discusses the common goal of visualization on the basis of the guiding concepts. It illustrates how different research work in software visualization, spanning over 10 years, is common and overlapping. The goal of this section is to promote the viability of the research problem's justification.

### **2.3.1 Screen View**

2D view scalability is hindered as content increases (Zeckzer 2010), and even with zoom (Broeksema 2011) and multiple view (Maletic 2011, Reniers 2011) capability they are prone to cognitive overload and lack of intuitiveness (Holy 2012) hence the use of 3D to add a new spatial dimension (Grancanin 2005), enhance memory activity (Marcus 2003), and make information analysis easier (Wiss 1998). Extending visual analysis to 3D increases the spatial space available for interacting with information. Pixel Maps (Marcus 2003), Kiviat (Kerren 2009), and Hull (Lambert 2012) metaphors benefit from 3D. Hierarchical Edge Bundling, which is extended into 3D space, improves visualization (Beck 2011).

The literature indicates the attempts made at maximizing the use of the screen real estate in order to cope with increasing quantities of information.

### **2.3.2 Program Structure, Control Flow**

(Grancanin 2005) outlines how visualization is applicable to the entire software lifecycle, including the support of legacy systems and use in security analysis (Goodall 2009). Both binary as well as source code is utilized. (Quist 2009, Trinius 2009) visualize malware acquired in binary form. (Marcus 2003, Zeckzer 2010, Maletic 2011, Reniers 2011) utilize source code from the perspective of metrics, classes & packages, whole software (for porting), and whole software (for structure) respectively. Binary code would provide a more accurate form for analysis, as it is what is actually executed on the computing device.

The literature shows that various visualization undertakings have been done with software and its attributes for purposes of improving the understanding of programs from both the binary and source code level.

### **2.3.3 Cognitive Dimension**

Abstraction plays a role in reducing information & cognitive overload. (Grancanin 2005) mentions 2 concepts that support this. Elision property of ‘abstract distant objects, detail closer objects’, and Bruce Shneiderman’s visualization mantra which specifies the default sequence of

‘overview first, zoom & filter, details on demand’. Similar echoes are captured in (Marcus 2003). Complementing components for information extraction are visual and textual representations. (Goodall 2009) proposes visualization for higher levels and textual for lower levels. The complex interactions between software entities are prone to make visualization cluttered with the potential effect of increasing the cognitive load (Caserta 2011, Goodall 2009) and ignoring information (Kuhn 2010).

Concise information display is enhanced by the use of metaphors that have the capability to represent the required information, i.e. expressiveness. Cognitive processing is enhanced by visibility & idealness of the required information encapsulated & presented in metaphors, i.e. effectiveness. Both expressiveness and effectiveness are properties of metaphors (Grancanin 2005) and tools for the design and evaluation of metaphors (Marcus 2003).

Enhancing abstraction is possible by not displaying all information at once. Pertinent information can be displayed dependent on the current context or upon demand, by encoding it in the metaphor (Reniers 2011). Furthermore, the use of mental models to aid in program comprehension, have been proposed. (Kuhn 2010) proposes the conceptual and structural models, in addition to introducing the concept of anchors, which provide a reference point in the analysis.

The literature brings out the concern of information overloading during the analysis of large quantities of information. Various solutions are proposed and guidelines presented to address the concern.

#### **2.3.4 Visualization, Visual Analytics**

In binary code visualization, metaphors represent aspects of code. However, since the code is abstract, these metaphors can take varied forms, for example, geometric shapes (Grancanin 2005, Caserta 2011) or real world objects (Grancanin 2005). In addition to shape, other visual attributes include size, height/depth, colour, texture/bumpmaps (Holten 2005), transparency, elevation, and position. These represent various code attributes such as sequence, control structure, nesting

level, declarations & implementations, classes & inheritance, occlusion, etc. (Marcus 2003, Holten 2005, Zeckzer 2010, Beck 2011, Lambert 2012) mention these attributes.

Various representations have been proposed: Pixel Maps / Cylinder Bars (Marcus 2003), Matrices / Row-Column (Zeckzer 2010), Tree Maps & Edge Bundling (Caserta 2011), Hulls (Lambert 2012). The above papers use graphs as a basis in different ways: replacing, compressing, and enhancing respectively. Graphs are covered in (Reniers 2011). Other representations include Treemaps (Reniers 2011, Kerren 2009), Radial (Reniers 2011), Kiviat (Kerren 2009), and Cartography (Kuhn 2010). Furthermore, various combinations of representations can be utilized concurrently (Broeksema 2011, Maletic 2011).

Visualization generates usable information, for example with refactoring (Broeksema 2011), which involves determining effort estimation and rewrite impact, or maintenance (Maletic 2011), which identifies high code turnover areas for purposes of either rewrite, code defect identification, regression tests, or fan in/out. Integration with other tools, either via input or output files is possible (Maletic 2011, Kuhn 2010).

Ultimately, visualization should increase the level of understanding of the information being processed, possibly by maintaining a consistent mental model (Wiss 1998) for recurrent use (Kuhn 2010). Richard Hamming's statement, 'insight, not number is what computing should evolve to', is a guiding principle. Abstraction of complex aspects to everyday equivalents (Medani 2010) and incorporating animation increases understanding (Medani 2010, Kerren 2009), which is further enhanced by lowering clutter by component aggregation (Holy 2012). Manipulating of the visualization also increases understanding (Wiss 1998). Navigation and location identification can be enhanced by both animation and panning features (Wiss 1998). However, animations are susceptible to large changes (Beck 2011).

From a rendering performance perspective (Wiss 1998 also raises this concern), the capability of being able to utilize the GPU to enhance performance is beneficial. Texture usage, for example, is natively performed by the GPU.

Due to the varied potential uses of visualization and the abstract nature of the information, a methodology is required to determine the ideal visualization for a given scenario. (Wiss 1998) introduces 2 parameters namely the data set (may require prototyping) and task analysis (involves the parameters of overview, zoom, filter, details on demand, relation, history, and extraction). (Beck 2011, Goodall 2009) also indicates these 2 parameters. Furthermore, (Wiss 1998) enhances the notion of the uniqueness of visualizations to the problem domain, indicating that if a methodology doesn't fit, the alternative could be either modify the design, add functionality, or use different concurrent visualizations.

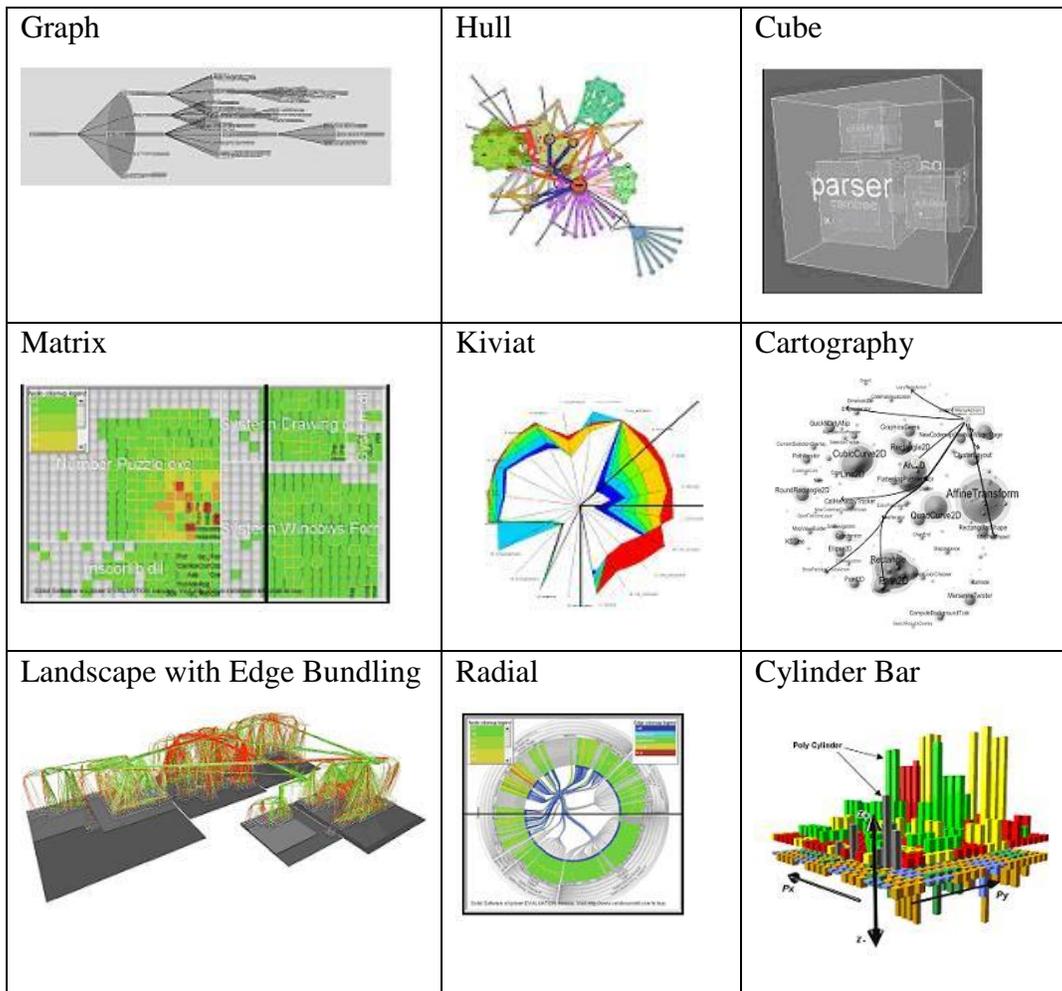
Table 2.1: Recurring Themes in Literature

Category	Recurring Themes
Screen View	Use of 2D & 3D
Program Structure	Large Code Bases, Software Lifecycle, Integration
Cognitive Dimension	Interaction, Navigation
Visualization	Natural Code-Metaphor mapping

Table 2.2: Literature Survey over the period 1998-2012 on Software Visualization

Category	Reference
Overview	Concepts (Grancanin 2005)
Metaphors	Abstraction (Medani 2005) Cube (Wiss 1998), Cylinder Bars (Marcus 2003, Broeksema 2011) Graphs (Holy 2012, Reniers 2011), Hull (Lambert 2012) Kiviat (Kerren 2009), Landscape (Wiss 1998) Matrices (Zeckzer 2010), Radial (Reniers 2011) Trees (Kerren 2009, Reniers 2011), Tree Maps (Holten 2005)
Aspect	Animation (Kerren 2009), Classes (Zeckzer 2010) Clutter (Holy 2012), Maintenance (Maletic 2011) Malware (Quist 2009, Trinius 2009), Metrics (Marcus 2003) Porting (Broeksema 2011), Realism (Holten 2005) Structure (Reniers 2011)

Figure 2.3: Illustration of Metaphors in Literature Survey (see Table 2.2 for sources)



## 2.4 Case For Research (Literature Gaps)

The previous section describes the common landscape based on the guiding concepts. This section (based on reviewed literature sources) identifies and discusses perceived gaps (which have not been addressed) and limitations (which could be enhanced) with the focus of being applied to the research problem.

### 2.4.1 Screen View

The role of 3D is increasing in the analysis of information. However, a direct conversion and/or utilization of 3D representations don't directly imply improved information analysis. The underlying information being analyzed and its interaction requirements should determine the

representation to be utilized. Hence analysis capability can be improved by incorporating the strengths of 2D and 3D within a 3D environment.

It is proposed that the 3D be mainly utilized to provide a work environment (increase the spatial analysis view, while maintaining a natural-based layout of interaction) within which both the 2D and 3D metaphors can be manipulated, rather than resorting to 3D ports when visual limitations are encountered.

#### **2.4.2 Program Structure, Code Analysis**

As software increases in quantity and complexity, visualization is providing a means of comprehending its functionality. Various representations have been used to determine structural aspects, while graphs are frequently utilized to represent program flow. However, providing a complete view of the program structure and code flow at a go, doesn't enhance screen view utilization or cognitive load required.

It is proposed that program structure, at the file level, be utilized as a basis for further drilling in, due to the ease of identifying relevant sections. In addition the compact nature and standardized format of binary code, rather than source code, would enhance both screen view utilization and cognitive load required.

#### **2.4.3 Cognitive Dimension**

In order to minimize cognitive overload, abstraction plays a role by providing the ability to view the entire content. Diffuseness and Visibility are not ideally intuitive, as the metaphors utilized are not designed specifically for code but borrowed from data visualization. However, incorporating abstraction a natural mapping can be utilized to represent the underlying concept. Secondary notation has been mostly textual though it can be enhanced within 3D spatial environments.

It is proposed that initial interaction with an abstracted model with simple geometric metaphors (designed for code), will enable personalized mental models to be derived for analysis potentially reducing the initial and subsequent cognitive load.

#### 2.4.4 Visualization, Visual Analytics

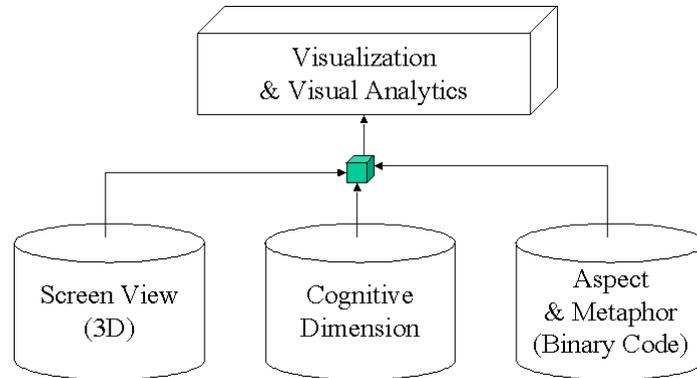
Various representations have been proposed for software visualization based on those utilized in information visualization. Hence they are not native to code analysis.

It is proposed that a code-biased representation would improve program comprehension by providing a code based navigation and interaction mechanism to encourage recurrent use.

Table 2.3: Summary of Perceived Literature Review Gaps

Category	Perceived Gap
Screen View	Use of synopsis / abstraction
Program Structure	Focus on mainly source code
Cognitive Dimension	Minimal cognitive offloading
Visualization	Metaphors borrowed from Information Visualization

Figure 2.4: Framework for Visual Analytics (based on the above table)



Note:

- (see above table for descriptions of figure components)

#### 2.5 Conclusion

As computing devices' capability increases, and new functionality and features required, the quantity and complexity of executable code will increase. Scenarios will arise where the code will need to be analyzed to understand its functionality in decreasing time frames.

Current tools and frameworks, adept at their designed functionality fail to provide a 3D visual analytical interface, with associated metaphors, for executable code. By providing an overall snapshot of an executable's structure and visual interaction functionality, a more intuitive form of analysis is possible, with the potential of increasing the quantity and quality of code analysis within shorter timeframes.

3D visual processing, which enables humans utilize an innate analysis capability is a viable tool for executable code analysis. By abstracting away language features, since similar logic can be viewed using different languages and focusing on the binary code attention can be directed on understanding the logic of the process rather than the logic of the language.

## CHAPTER 3 – SYSTEM DESIGN

### 3.1 Introduction

The design of the application is based on the structured format of code. At the binary level, the code is structured in a form ready for execution by a processing unit. This code can be directly abstracted to assembly language mnemonics while still preserving the inherent structured form. It is at this level of abstraction that the application analyzes an executable.

Programs are usually distributed in their executable format (.exe files). In order to more easily analyze them, they need to be converted into their higher-level language equivalents. Assembly language equivalents are obtained via a disassembler for the target platform. Due to the varied disassemblers available and their correspondingly different outputs, an input format for the application is specified for this application (derived from the dumpbin utility availed with Microsoft's Visual Studio Integrated Development Environment). Once in the appropriate format, the disassembly listing can then be imported into the application for visualization.

This chapter describes the research analysis, design, and implementation process. It consists of the following sections:

- Development, describes the prototyping method (Section 3.2)
- Requirements, describes the program features (Section 3.3)
- Conceptual, describes the interaction flow and content generation process (Section 3.4)
- Data, describes the sources, collection, and introduces analysis & validation (Section 3.5)
- Import Format, describes the structure of the assembly listing of the disassembled executable (Section 3.6)
- Content, describes the data files, interfaces, and formats used (Section 3.7)
- Visual, describes the user interface aspects (Section 3.8)
- Metaphor, describes the visualization design (Section 3.9)
- Construct, describes the analysis and validation of assembly code (Section 3.10)

## 3.2 Development

### 3.2.1 Prototyping

Due to the iterative and incremental nature of the design and development process, the prototyping methodology of software development is utilized. It enables enhancing functionality until a complete system is implemented.

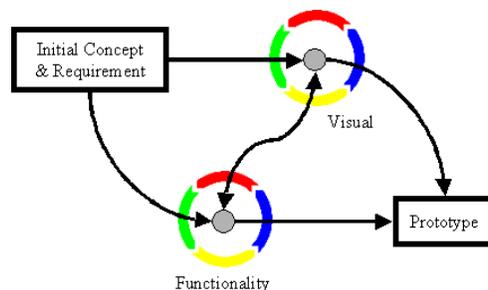
The prototyping approach utilized is evolutionary in that the prototype will be refined and eventually become the final product. This is through an iterative process of reviewing and improving on the current build. It combines both horizontal and vertical prototyping enabling both a broad and component based approach respectively. The horizontal aspect enables designing of the user interface, while the vertical aspect aids in functionality design. These 2 aspects form the key cycles i.e. the visual and functionality features.

The cycles enable incremental prototyping as the various components can be developed and then integrated into the system. This would result in a module-based design that would enable extensibility for additional future features.

### 3.2.2 Cycles

There are 2 key concurrent cycles, the visual and functionality, in order to handle the key structural components of the application. The visual cycle addresses the views presented by the application for interaction; the front end. The functionality cycle addresses the background processing upon which the visual component operates by retrieving content; the backend.

Figure 3.1: Diagrammatic illustration of the utilized prototyping development cycle.



Note:

- From the initial concept to the final prototype, the visual and functionality components of the system will be design and developed concurrently, in an evolving and interconnected manner. (The illustration is a customized derivative based on evolution prototyping.)

### **3.2.3 Benefits**

The prototyping methodology provides benefits to this development endeavour that include:

- Ideal for the design of graphical front ends, which are prone to initial change due to being a contact point for the system's users.
- Enables feedback to be incorporated into the next cycle.

### **3.2.4 Limitations & Handling**

The iterative nature of prototyping can result in the method being liable to ever changing design cycles. To overcome this aspect, once the basic visual and functionality features are formulated, changes are minimized and focus change to information extraction using the visual and functionality components.

## **3.3 Requirements**

### **3.3.1 Functional**

These describe what the system does, in the process describing and driving the design.

Key features:

- External disassembler incorporation to enable visualization & analysis of programs for which there is a disassembler for the target platform.
- Text-based data storage engine to contain the disassembled code in a format that enables visualization & analysis.
- Lattice-based visualization metaphor for visualization, analysis, interaction, and navigation of disassembled code.

### **3.3.2 Non Functional**

These describe quality related features of the system, in the process describing and driving the architecture.

Key features:

- Extensibility to enable building new functionality in order to enhance the program.
- Scalability to enable handling of large disassembled files.
- Modular to enable adaption for other platforms.

## **3.4 Conceptual**

### **3.4.1 Interaction Flow**

The main interacting components of the program comprise:

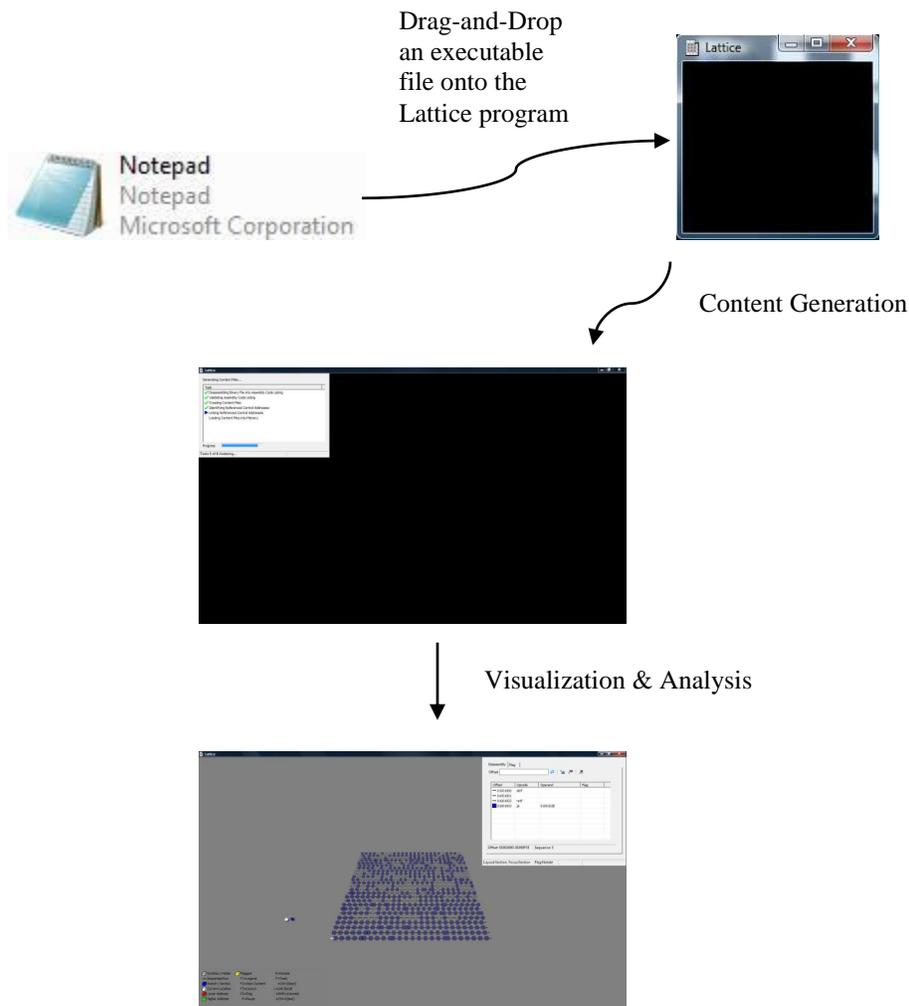
- An executable to be analyzed, either in binary format, or in its equivalent disassembly.
- The Lattice application, which visualizes the executable.

The process of loading an application for visualization and subsequent analysis is simplified through the use of drag-and-drop functionality. This is intended to minimize intermediary steps that could be required during the loading process as once the application is run, the operating system's graphical file system shell can be utilized to locate the file to be visualized, which is then dragged onto the running visualization application.

Once loaded, the intermediate steps that may be required, based on the loaded file, are executed and the loading progress shown. When all the steps are completed, the visualization is then loaded, and analysis can begin.

Below is an example illustration of the main interacting components (in this case the Notepad application from Microsoft and the Lattice default screen used for loading the application) of the visualization process; beginning with an executable, it is dragged using the mouse, onto the application in a drag-and-drop manner.

Figure 3.2: Interaction Flow



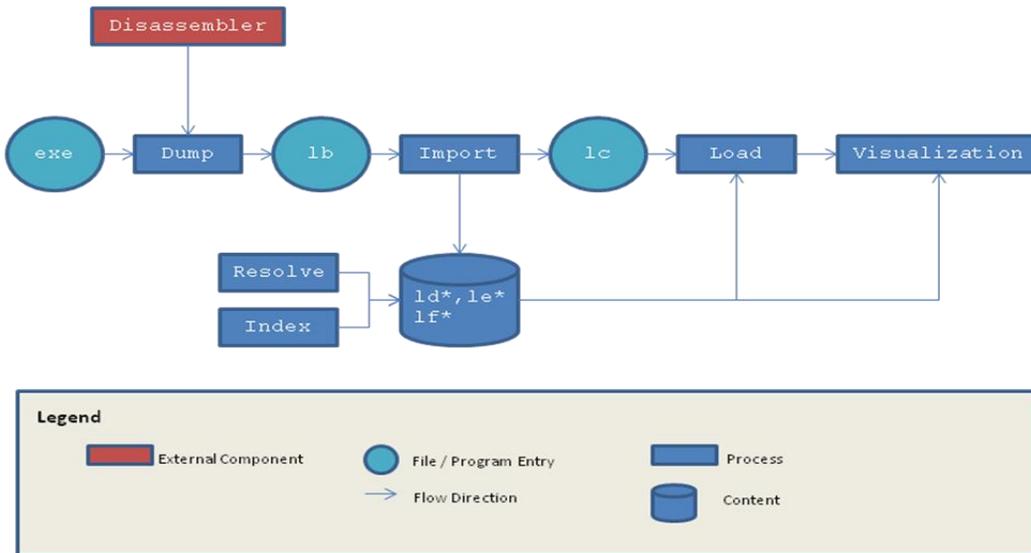
### 3.4.2 Content Generation

Once the executable file has been loaded into the application, various processes are run in order to generate the content that will be used for the visualization. The main processes are:

- Disassembly of a loaded executable file into its corresponding assembly listing (referred to as dumping).
- Importing of the disassembled assembly file listing into the various content files that will be utilized for both visualization and analysis.
- Loading for the generated content files for visualization and analysis.

Below is an illustration of the various processes and outputs involved from the loading of the initial program to its visualization.

Figure 3.3: Content Generation



The components illustrated above are described in the sections below.

- External Components
- File / Program Entry
- Processes
- Content

### External Components

These are not part of the Lattice program but are used to generate content that will be used by the program. For example, various disassemblers are available for the different platform. The appropriate disassembler would be used dependent on the target platform of an executable.

Table 3.1 External Components

Item	Description
Disassembler	An external integratable tool used to generate the assembly listing that will then be imported into the application. Dependent on the executable's platform, the appropriate disassemble can be utilized.

### File / Program Entry Points

These are the high-level files used to either generate content (.exe, .lb) or act as a placeholder for the various content files (.lc; acts as a place holder to represent the various .ld, .le, and .lf files). They also provide entry points to initiate the visualization process, by eliminating the need to perform previously completed tasks, thus saving time.

For example, once an executable file (.exe) has been disassembled into its corresponding assembly file listing (.lb), subsequent visualizations of the same executable file need not disassembly the executable gain. Similarly, once the assembly file listing (.lb) has been imported into the various data files (.ld\*, .le\*, .lf\*; collectively represented by the .lc file), they need not be regenerated again. This saves time when dealing with large files.

Table 3.2 File / Program Entry Points

Item	Description
.exe	The executable file to be disassembled. Initiates the visualization process from the loading of the executable. The content is in binary form.
.lb	The disassembled file (text) that is generated by the disassembler (formats vary per disassembler). Initiates the visualization process from the disassembled file (in order to avoid the need to repeat the disassembly of an already done disassembly).
.lc	A placeholder file (contains no content) used to represent all the other content files (.ld*, .le*, .lf*). It enables initiation of the visualization process from already imported content (content is ready for visualization; there is no need for disassembly nor importing into the appropriate format).

## Processes

These refer to the key processes involved in preparation of the content for visualization and subsequent analysis.

Table 3.3: Key Processes

Item	Description
Dump	Disassembles the loaded program using a disassembler
Import	Parses the assembly listing and generates structured data usable for analysis
Resolve	Identifies and connects control branching offsets found in the assembly listing
Link	Connects control branching offsets for visualization purposes
Load	Retrieves the content from the data files into memory for visualization
Visualization	Provides a visual interaction interface to the disassembled program

## Content Files

These files (with the extensions `.ld*`, `.le*`, `.lf*`), contain the imported content in a structured format that is used for the visualization as well as the data analysis. These are detailed in Content Design (Section 3.7).

Table 3.4: Content Files

Item	Description
<code>.ld*</code>	Set of structured files of the disassembly that can be queried
<code>.ld</code>	Imported assembly code
<code>.ldl</code>	Assembly code lines used to resolve addresses for branching
<code>.ldr</code>	Assembly addresses that need to be resolved
<code>.le*</code>	Set of optimized structured files of the disassembly used for visualization
<code>.le</code>	Extracted & optimized assembly code used for visualization
<code>.lel</code>	Extracted assembly code indices
<code>.ler</code>	Extracted assembly code indices that need to be resolved

.lf*	Flag related data
.lf	Flag database
.lfl	Flag configuration data

Note:

- The filename, obtained from the initial drag-and-drop process, is combined with the various extensions to generate the different files that are used by the program.
- The file extensions, .l\*, were selected in order to enable the application identify files it has generated, and for the user to easily identify files due to their unique file extensions. The files however, comprise of structure raw text.

### 3.5 Data

#### 3.5.1 Source

The data consists of software programs in the form of executable files that will be used as input. Programs are restricted to the 32 bit version for the Intel platform. The approach is based on the availability of the executables in binary format as opposed to source code format (which is usually retained by the developers).

However, since the binary format needs to be translated to at least assembly code for analysis by humans, the input is based on this format. Since there is a 1-to-1 mapping between machine and assembly code (Figure 2.1), and disassemblers available for the various platforms, focus can be directed to the importing and visualization processes.

#### 3.5.2 Collection; Tools & Process

Disassemblers are utilized to generate the equivalent assembly language code for executable programs that are acquired for analysis. The generated assembly code listing is utilized as input into the visualization program.

The basic process (Figure 3.3) involves:

1. Acquire an executable file

2. Generate the assembly language equivalent using a disassembler for the target platform
3. Load the assembly language listing into the visualization program

This process ensures that any program that has been converted into its executable equivalent for deployment can be visualized as long as a disassembler for the platform exists. This provides the benefit of separating the visualization process from the disassembling process, enabling a more modular system. In addition, it enables the use of already existing disassemblers that have been developed for various processors.

### **3.5.3 Analysis & Validation**

In order to validate that the disassembly listing, that will be generated and utilized in the visualization and analysis, is the correct representation of an equivalent program. The following approach is taken.

- Sample programs are created. This starts with a simple program that exits immediately (has no functionality). Other programs representing different constructs and simple functionality are subsequently created based on the simple program.
- The equivalent functionality is described from how it would be implemented using only an Instruction Set Architecture (ISA) for a given platform.
- The programs are then disassembled, and the program flow analyzed to determine a match between the actual platform's disassembly, and the conceptual platform's logic.

The construct's section (Section 3.10) details the analysis & validation.

### **3.6 Import Format**

The input format provides an interface specification for importing assembly listings, generated by different assemblers, into the program for visualization. As long as the output of the different assemblers can be formatted into the specification, any disassembler can be utilized, as subsequently any executable which has a corresponding disassembler can be visualized and analyzed by the program.

The format is based on the dumpbin.exe utility as described in the chapter's introduction (Section 3.1). Lines that don't match the instruction format, specified below, are ignored, as they don't constitute an instruction (e.g. comments generated as part of the disassembly process).

The format is specified for a given line that is to be parsed and imported, and is of the form:

`_XXXXXXXX:XX_XX_XX_XX_XX_XX_<Opcode>_<Operand>`

where `_` is a place holder for a space, and `X` is a place holder for a hexadecimal digit. Details are described in table 3.5 below.

Table 3.5: Format for 1<sup>st</sup> line of instruction

Offset from line start	Length (characters)	Description
0, 1	2	Space
2-9	8	Offset Address (hexadecimal digits)
10	1	Colon
11	1	Space
12-29	18 (3 * 6)	Byte + Space i.e. 2 hexadecimal digits + space
30	1	Space
31-41	11	Opcode
42	1	Space
43	(until end of line)	Operand

If more than 1 line is required to describe the bytes of an instruction, the format is of the form:

`_____ :XX_XX_XX_XX_XX_XX`

where `_` is a place holder for a space, and `X` is a place holder for a hexadecimal digit. Details are described in table 3.6 below.

Table 3.6: Format for 2<sup>nd</sup> & subsequent instruction lines

Offset from line start	Length (characters)	Description
0-11	12	Space
12-29	18 (3 * 6)	Byte + Space i.e. 2 hexadecimal digits + space

Sample Input is illustrated below (taken from an executable's corresponding .lb file):

Figure 3.4: Disassembly Listing of Notepad.exe

```

notepad - Notepad
File Edit Format View Help
Dump of file C:\Users\X\Desktop\notepad.exe
File type: EXECUTABLE IMAGE

01001000: 9F          lahf
01001001: F7          ja
01001002: CB          retf
01001003: 77 29      ja             0100102E
01001005: F4          hlt
01001006: CB          retf
01001007: 77 3D      ja             01001046
01001009: B8 CA 77 89 B8 mov     eax,0E88977CAh
0100100E: CA 77 90   retf     9077h
01001011: BA CA 77 00 00 mov     edx,77CAh
01001016: 00 00      add     byte ptr [eax],a1
01001018: 3B C8      cmp     ecx,edx
0100101A: E1 77     loopne 01001093
0100101C: C8 F7 E3 77 enter  0E3F7h,77h
01001020: 8A 8E E3 77 35 9E mov     cl,byte ptr [esi+9E3577E3h]
01001026: E3 77     jecxz  0100109F
01001028: F9 E3 77 E8 22 reg     dword ptr [esi+22E87E3h]
0100102E: E1 77     loopne 010010A7
01001030: E0 3B     loopne 0100106A
01001032: E3 77     jecxz  010010A8
01001034: E1 34     loopne 0100106A
01001036: E3 77     jecxz  010010AF
01001038: 77 F2     ja     0100102C
0100103A: E0 77     loopne 010010B3
0100103C: 30 7F E3  xor     byte ptr [edi-1Dh],bh
0100103F: 77 AE     ja     010010EF
01001041: 5C        pop     esp
01001042: E3 77     jecxz  010010B8
01001044: DA C8     mov     st,ct(0)
01001046: E3 77     jecxz  010010BF
    
```

a. Disassembly Listing – contains comments

```

notepad - Notepad
File Edit Format View Help
Dump of file C:\Users\X\Desktop\notepad.exe
File type: EXECUTABLE IMAGE

01001150: CF          iretd
01001151: C1 B6 77 49 EC B7 sal     dword ptr [esi+B7EC4977h],77h
01001158: F4          hlt
01001159: 59          pop     ecx
0100115A: B6 77     mov     dh,77h
0100115C: 69 6E 77 F0 5A 86 mov     ebx,dword ptr [esi-4Ah],0B65AF077h
01001163: 77 D6     ja     01001138
01001165: 82 B6     mov     d1,0B6h
01001167: 77 ED     ja     01001156
01001169: 58        pop     ebx
0100116A: 86 77     mov     dh,77h
0100116C: 00 00     add     byte ptr [eax],a1
0100116E: 00 00     add     byte ptr [eax],a1
01001170: F9        stc
01001171: 89 D6     mov     esi,edx
01001173: 77 E3     ja     010011D8
01001175: E5 D5     in     eax,0D5h
01001177: 77 9D     ja     01001116
01001179: 07        pop     es
0100117A: D7 62     xlat   byte ptr [ebx]
0100117B: 77 62     ja     010011DF
0100117D: 95        mov     eax,ebp
0100117E: D6        xchg
0100117F: 77 D5     ja     01001156
01001181: 1F        pop     ds
01001182: D7 62     xlat   byte ptr [ebx]
01001183: 77 F1     ja     01001176
01001185: 5A        pop     edx
01001186: D6        mov     esi,edx
01001187: 77 C6     ja     0100114F
01001189: 9E        sahf
0100118A: D6        ja
0100118B: 77 E3     ja     01001170
    
```

b. Disassembly Listing – highlights show multi line instruction

### 3.7 Content

Various data files are utilized to store the content required for both visualization and analysis (Table 3.4). A text-based database engine, which provides 3 interfaces (read, write, and update) is utilized in modifying the data files.

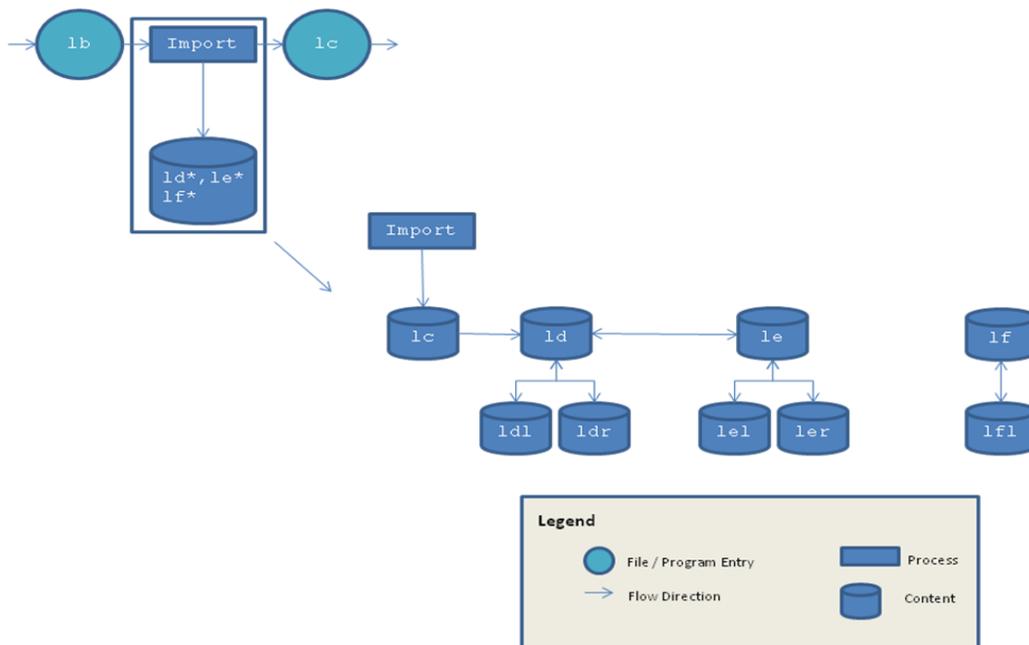
#### 3.7.1 Data Files

This section expounds on the import process, initially mentioned in Content Generation (Section 3.4.2), illustrating the various data files (Figure 3.5) that comprise the content. Details of the file structures are given in Data Format (Section 3.7.3).

In summary:

- .lc files represent a place holder to identify the other associated files
- .ld files contain the imported disassembly listing in a format usable by the program
- .le files contain information, based on the .ld files, intended for visualization
- .lf files contain the parameters used for analysis of the executable code

Figure 3.5: Data Files

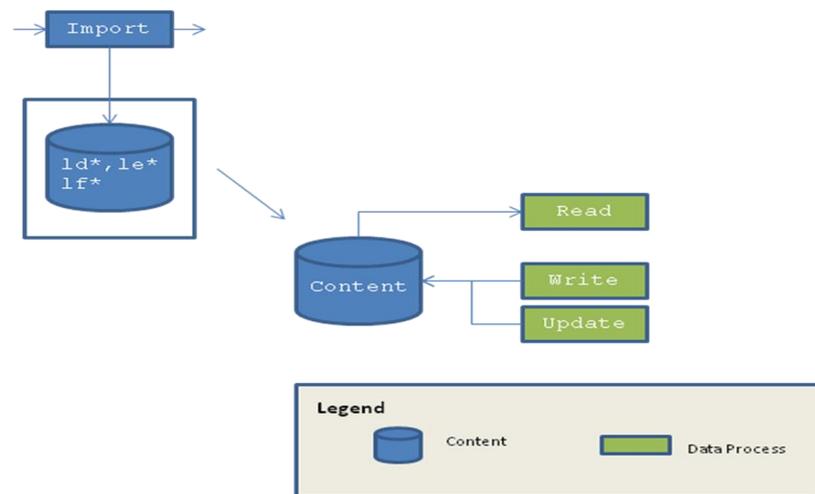


### 3.7.2 Data Interfaces

In order to facilitate reading and writing of the data files, a text-based data engine is utilized to retrieve and store content within the data files. For each data file, 3 interfaces are provided, illustrated below, namely:

- Read – retrieves a single line from the data file
- Write – stores a single line to the data file
- Update – stores the updated version of a line or segment of the line to the data file

Figure 3.6: Data Engine



### 3.7.3 Data Format

This section describes the details of the content files as well as provides sample illustrations. The files consist of structured text and can be viewed with any text editor. Each of the lines of the various files are of a standard size in order to enable efficient read/write and search operations.

#### .lc File

The `.lc` file is a place holder for the program to easily identify the other program files that contain content used for both visualization and analysis. The file is empty and is used to obtain the filename portion of the loaded file. For example, loading the `Notepad.lc` file into the program enables the location of the other related content files such as `Notepad.ld`, `Notepad.le`, `Notepad.lf`, etc.

## .ld File

The .ld file contains the imported assembly listing in a structured format i.e. the assembly lines only, with non-instruction lines not imported.

Table 3.7: .ld File Format

Name	Type	Length	Description
Offset	String	8	Address in hexadecimal format
Byte	String	24	Instruction in hexadecimal format
Opcode	String	12	Instruction opcode
Operand	String	101	Instruction operand
Execute	Number	2	Type of execution 0 = sequence 1 = control
Isa	Number	8	Instruction Type
Offset2	String	8	Branch address in hexadecimal format
Level	Number	2	Direction of execution 0 = sequential 1 = branch to lower address 2 = branch to higher address 3 = branch (determined at runtime)
Line2	String	8	Line within the .ld file from the branched address
Flag	Number	4	Flag Notation

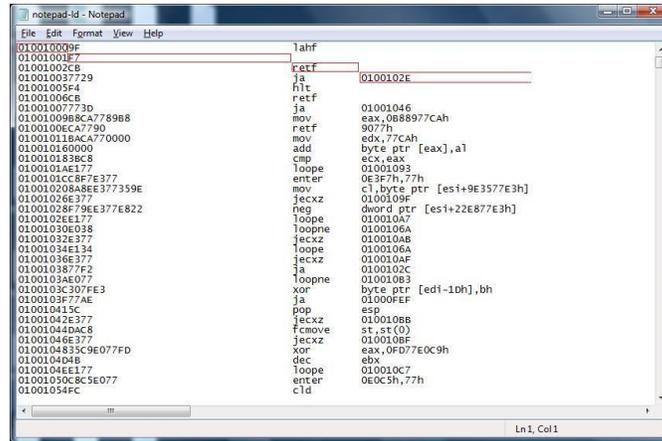
The .ld file builds on the .lb file (the disassembly listing generated by a disassembler), by identifying, extracting, generating, and tag various attributes of each assembly line.

For example, the opcode is either classified as a sequence or control instruction. If it is a control instruction, the offset of the branching address is identified and the direction of the flow (to either a lower address or higher address) is determined. Certain scenarios prevent identification of the branch address, such as when a CPU register or memory location is involved indicating a

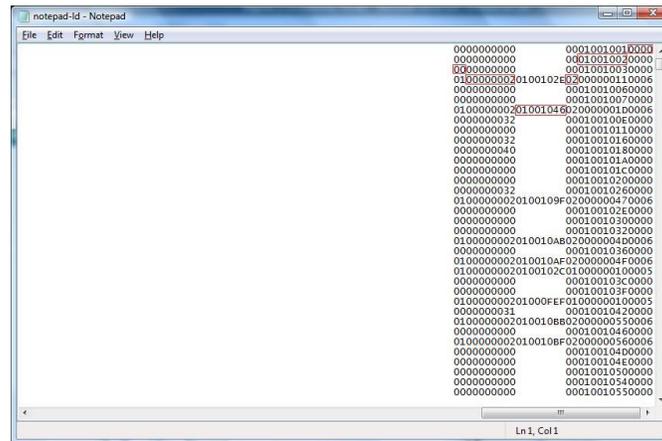
runtime branch. Once a new branching offset is identified, the current line of the instruction is updated to link it to the branched to instruction's line.

Due to its contents, the .ld file forms the main data file that is utilized when any query needs to be done to identify pertinent information. Once generated, the .lb file, is no longer utilized.

Figure 3.7: .ld File Content



a. Highlight shows the initial 4 fields



b. Highlight shows the last 6 fields

### .ldl File

The .ldl file is utilized to aid in searching for address links. It comprises of line information whose content identifies lines in the .ld file.

Control branches that have been identified in the .ld file need to be linked in order to enable the visualization process highlight interconnecting branches. Hence, the ldl file is used to calculate the linkages.

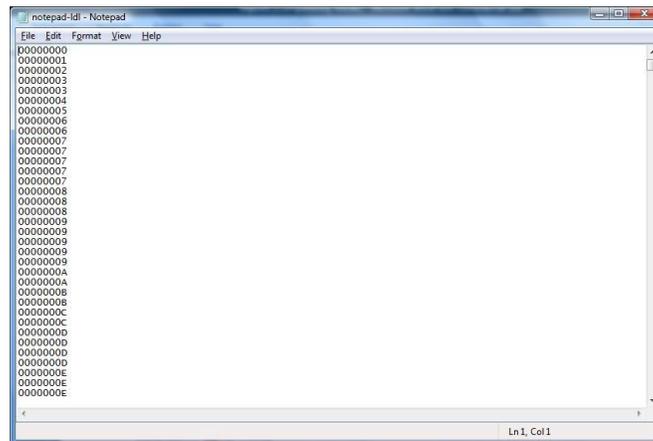
For example, once a branch offset is identified, its offset is subtracted from the initial offset in the assembly listing. The difference is the location of the jumped to branch location. This difference, when reference in the .ldl file corresponds to a line. The contents of the line identify the instruction that will be executed upon branching.

The file is generated concurrently with the .ld file. Instructions in the .ld file that occupy more than 1 byte are reflected in the .ldl file in more than 1 line, i.e. the lines in the .ldl file represent 1 byte each, with multiple byte instructions in the .ld file, occupying an equivalent number of bytes in the .ldl file.

Table 3.8: .ldl File Format

Name	Type	Length	Description
Line	Number	8	Line number (hexadecimal) of instruction located at byte offset from start

Figure 3.8: .ldl File Content



## .ldr File

The .ldr file contains addresses that need to be resolved once the .ld file is generated as they are forward referencing processes.

During the generation of the .ld file branching instructions are identified and noted as either a forward, backward, or runtime branch. It is possible to calculate backward branches as the information has already been generated. However, forward branching is not possible due to the lack of required information, yet to be processed.

Hence, the .ldr file contains addresses that will need to be resolved to enable linking of branching instructions. Once the .ld file generation process is complete, the resolve process utilizes this file to calculate branching connections, which are reflected in both the .ld and .le files.

Table 3.9: .ldr File Format

Name	Type	Length	Description
Offset2	Number	8	Branch address that needs to be resolved
LineLd	Number	8	Location of the line in the .ld file containing the branching instruction
LineLe	Number	8	Location of the line in the .le file containing the branching instruction

Note: The values are in hexadecimal form

Figure 3.9: .ldr File Content (Highlight shows fields)

```

notepad-ldr - Notepad
File Edit Format View Help
0100102E000000300000001
01001046000000000000003
0100109F000000000000005
010010AB000000130000007
010010AF000000150000009
0100102C00000016000000A
01000FEF00000019000000C
0100108B0000001B000000E
010010BF0000001D00000010
010010370000002300000012
0100108000000240000013
01000FF9000000280000015
010010E0000002A0000017
0100108F0000002C00000019
010010E30000002F0000001B
010010E7000000320000001D
01001099000000350000001F
010010660000003900000021
010010FF0000003B00000023
010010AD0000003C00000024
01001107000000400000026
01001130000000430000028
010011300000045000002A
010011800000048000002C
010010690000004B0000002E
010010F90000004D00000030
01001170000004F00000032
010010640000005100000034
0100112F0000005300000036
010010C10000005900000038
0100114B0000005D0000003A
010010800000005F0000003C
010010900000006000003D
01001153000000620000003F
010010A10000006500000041
Ln1, Col1

```

## .le File

The .le file contains the extracted assembly listing derived from the .ld file that is utilized for visualization as well as analysis. The field is generated concurrently with the .ld file, and is the main file that is utilized in the visualization process.

Instructions contained within the .ld file can be broadly classified as either being of a sequential or control nature, with the latter branching to a potentially non-consecutive instruction. Usually more than 1 sequential instruction follows another. Within the .le file, contiguous sequential instructions are reduced to a single line to enable visualization using a lattice structure.

The file compresses the .ld file while storing information that will enable it reference and retrieve the full information from the .ld file if necessary. For example, it stores offset addresses that can be referenced in the .ld file to retrieve opcode and operand information. It also recalculates branching connections within itself as it has compressed the .ld file, and can no longer directly use the branching information in the .ld file.

Table 3.10: .le File Format

Name	Type	Length	Description
Line	String	8	Contains the corresponding line in the .ld file
Execute	Number	2	(Similar to the corresponding .ld field)
Isa	Number	8	(Similar to the corresponding .ld field)
Level	Number	2	(Similar to the corresponding .ld field)
Line2	String	8	Contains the corresponding branch line in the .ld file
Offset	String	8	Contains the address of the line
Offset2	String	8	Contains the address of the branch line
Index2	String	8	Contains the index in the .le file of the branch
Seq	Number	4	Number of constituent instruction sequences
Flag	Number	4	If 'Execute' = 'Sequence', number of flagged sequences If 'Execute' = 'Control', flag notation



Figure 3.11: .lcl File Content



**.ler File**

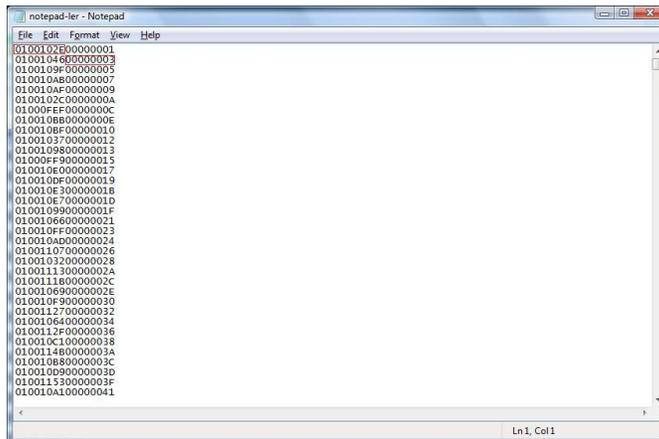
The .ler file provides a similar role to the .le file, that the .ldr file provides to the .ld file.

Table 3.12: .ler File Format

Name	Type	Length	Description
Offset	Number	8	Branch address that needs to be linked
Index	Number	8	Location of line in .le file containing the branch instruction

Note: the values are in hexadecimal form

Figure 3.12: .ler File Content (Highlight shows fields)



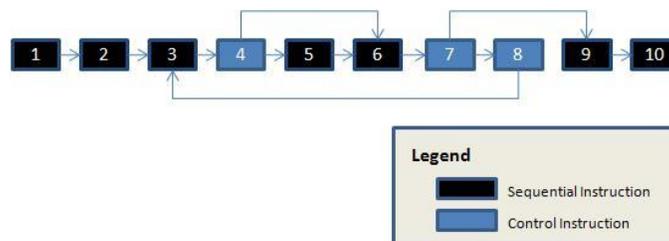
## 3.8 Metaphor

### 3.8.1 Instruction Set Architecture (ISA)

A platform's ISA comprises its assembly language constructs. In general, the instructions can broadly be divided into 2 broad categories:

- Sequential instructions, which will execute the next following instruction.
- Control instructions, which have the potential to alter the flow of the program.

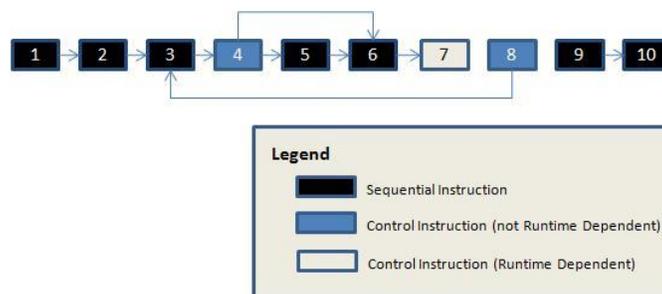
Figure 3.13: Sequential and Control instructions (example program flow of 10 instructions)



Note:

- Program consists of 7 sequential type instructions, and 3 control type instructions.
- Instructions 4 and 7 are of a conditional nature, while instruction 8 is of an unconditional nature.
- The program flow can be determined prior to runtime. For the control type instructions, the branching offset is specified in immediate form.

Figure 3.14: Runtime Dependency



Note:

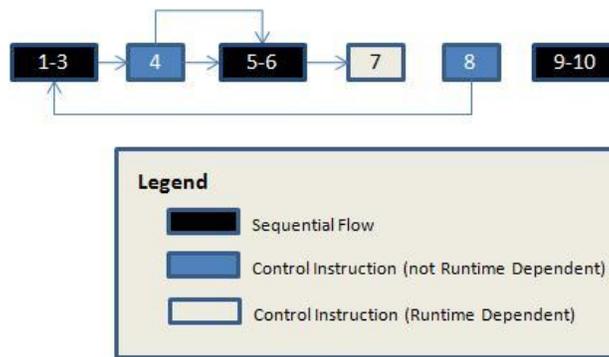
- Instruction 7 is now runtime dependent, possibly due to the operand of the instruction specifying the branching offset in register or memory.

### 3.8.2 'Compression' of the ISA

A potential optimized representation of a program flow can be generated by combining sequential instructions into a single representation and treating them as one unit. Logic flow within the unit begins from the start and flows to the end sequentially; there are no deviations.

Control instructions, however, can't be represented as a single unit due to their potential to deviate from the potentially sequential flow.

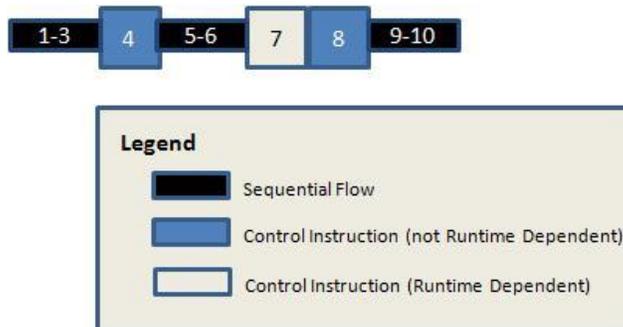
Figure 3.15: Metaphor representation



Note:

- The metaphor representation effectively reduces the potential graphical area required to represent a program's flow.

Figure 3.16: Enhance Metaphor representation

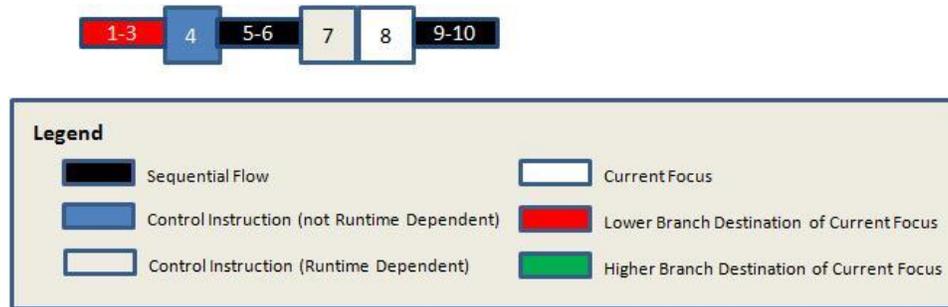


Note:

- The metaphor representation can be extended to both 2D (by adding the Z axis) and 3D (by adding the Y axis).

- Branching information is now stored within the metaphor representation. This increases clarity as the number of branches increases.
- Each combined sequence instruction and each individual control instruction are now referred to as nodes of the metaphor.

Figure 3.17: Display of branching information



a. Branching information for Instruction 8



b. Branching information for Instruction 4

Note:

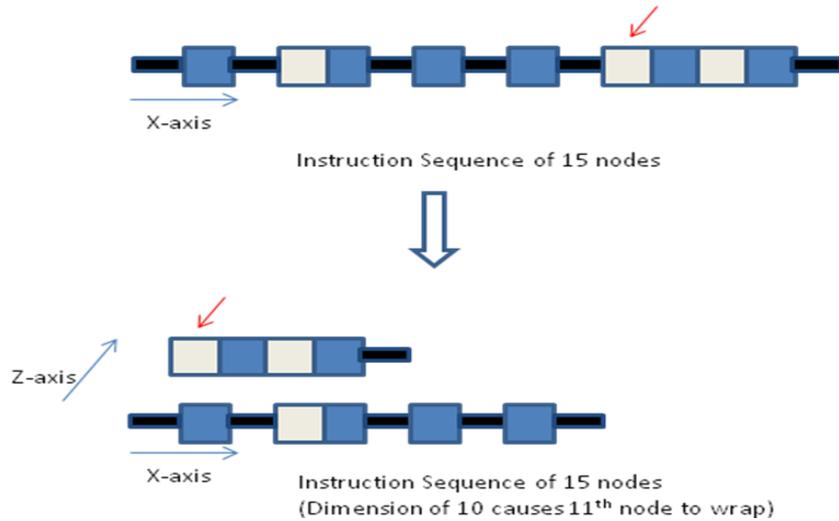
- Information on the exact branching location when branching into a sequential section is displayed by the metaphor textually, when interacting with the metaphor and concurrently viewing the disassembly listing.

### 3.8.3 Folding Instruction Sequences to Sections

As the quantity of nodes representing instructions increases, the linear growth in the X-axis is limited by the available screen space. In order to accommodate this growth, the Z-axis can be utilized. However, this requires the specification of a dimension.

A dimension refers to the number of nodes that will be displayed on the X-axis. For example, if the dimension is set to 10, then the 11<sup>th</sup> node will wrap around and be displayed at the next incremented Z-axis index. The figure below illustrates this folding.

Figure 3.18: Folding Nodes Sequences



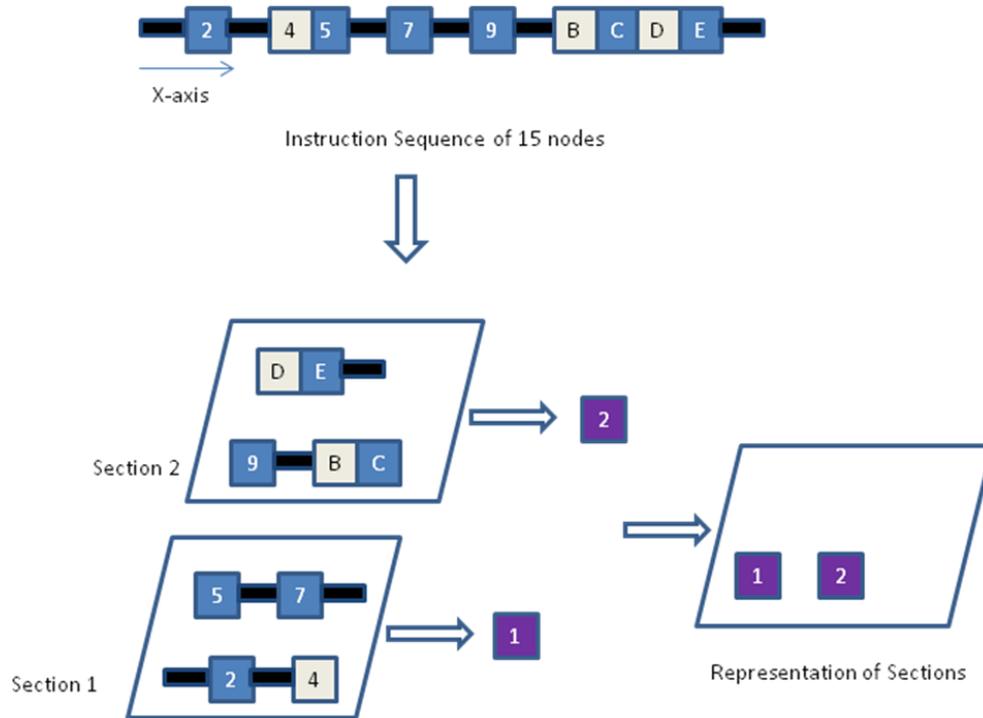
In order to provide clarity as nodes are combined, the follow formats are defined:

- The size of a node in both the X-axis and Z-axis is of equal size.
- When folding a node sequence, the Z-axis leaves spacing equivalent to the size of a node. This results in the Z-axis containing half the number of nodes than the X-axis. For example, using the figure above, if the dimension is set to 10, then there will be a maximum to 10 nodes on the X-axis, and a maximum of 5 rows on the Z-axis.

### 3.8.4 Building Sections

Once the maximum number of rows along the Z-axis is reached (based upon the set dimension), the resultant collection of nodes is referred to as a section. A single section is represented by a different node type differentiated by its colour code. The next node after a section is formed becomes the 1<sup>st</sup> node of the new section. The figure below illustrates this process.

Figure 3.19: Section Formation



Note:

- The initial sequence of 15 nodes forms 2 sections based on a dimension of 4.
- Nodes within a section are located only along the X-axis and Z-axis.

As the number of sections increase, their layout is ordered along the axes in the following order: X-Z-Y. The dimension used when representing sections is defined by the number of sections, with the value being the cube-root of the number of sections.

### 3.8.5 Navigation of Nodes & Sections

Due to the multi-dimensional nature of the lattice structure, navigations in the various domains is possible, and sequentially moving from one node to another is not mandatory.

Movement along the X-axis, with the current input mode set to node navigation, results in either a move to the next (forward) or previous (backward) node. Moving forward from the last node in a given section results in the 1<sup>st</sup> node of the next section being highlighted, while moving backward from the 1<sup>st</sup> node in a given section results in the last node of the previous section being highlighted.

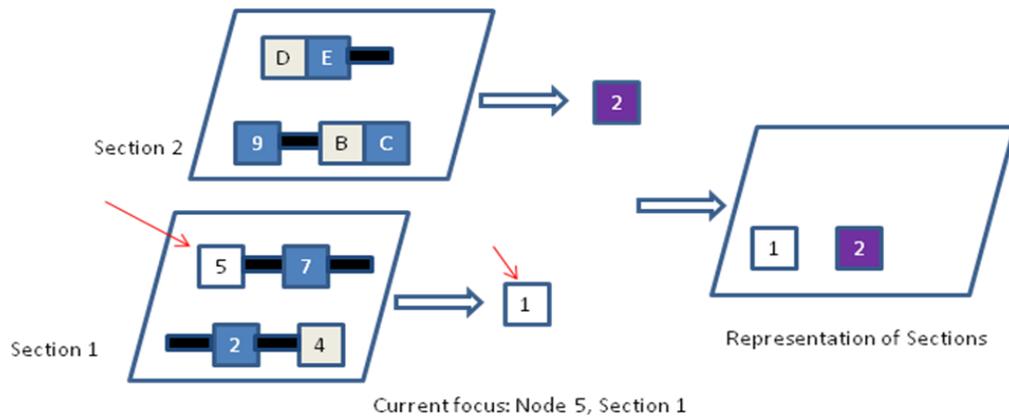
Movement along the Z-axis, with the current input mode set to node navigation, results in either a move to the next (forward) or previous (backward) node using an offset equal to the dimension. This is equivalent to either an upward or downward movement. The movement may either result in the next or previous section being displayed.

Movement along either axis results in the corresponding section of the highlighted node being highlighted in the 'Section' navigation view.

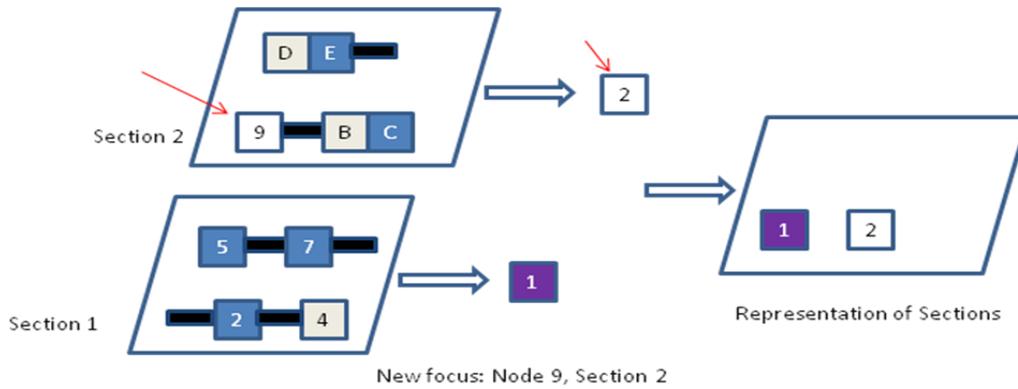
Movement in the 'Section' navigation view resembles the above description of movement in the 'Node' navigation view, with the difference being that when a different 'Section' is selected, the 1<sup>st</sup> node of that section is the one highlighted. This is because a section represents more than 1 node.

The figure below illustrates the focus changing from the 5 node to the 9 node via an upward movement, which is equivalent to adding the dimension of value 4.

Figure 3.20: Navigation



a. Initial Location



b. Location after navigation

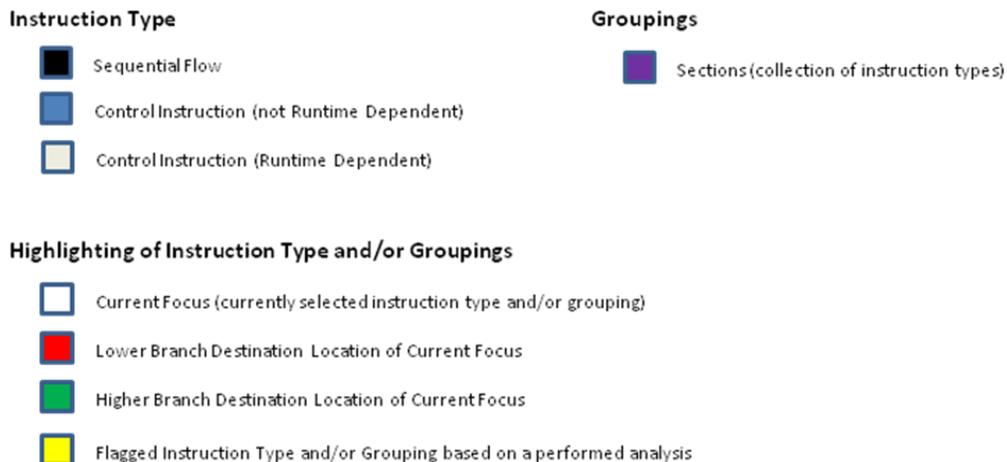
Note:

- Navigation is via either the keyword, which limits movement to either the next, previous, up, and down nodes. To quickly navigation to any node, the mouse can be utilized to select the desired node.
- Once a node is selected, information related to it can be displayed via pressing the ‘Enter’ key.

### 3.8.6 Notation

In order to encode information in the metaphor, colour notations have been utilized. These are used to provide intuitive information from visual analysis of the disassembled program.

Figure 3.21: Notation



## 3.9 Constructs

### 3.9.1 Assembly Language

Given any executable file, its contents can be converted into an equivalent assembly language listing (given a disassembler for the target platform).

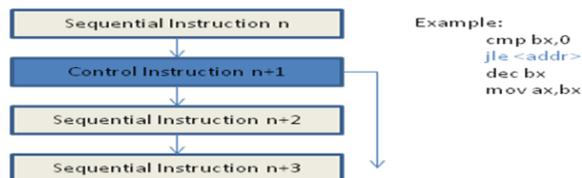
The Instruction Set Architecture (ISA) for a given platform describes the assembly language programming interface for that given. The available instructions are utilized to generate programs either directly or via a higher level programming language. However, at the assembly language level, there is usually a one-to-one mapping with the machine code, if macros aren't utilized.

At this level of analysis, the instructions can broadly be classified as either being sequential or control. Sequential instructions are executed and the following instruction executed next (Figure 3.30). Control instructions on the other hand, have the potential to alter the flow of execution (Figure 3.31).

Figure 3.22: Sequential Flow



Figure 3.23: Control Flow



Hence, executable code comprises of different permutations of sequential & branching instructions using the available ISA.

### 3.9.2 Code

In order to ease the programming effort, higher level languages have been introduced, such as C/C++. These have also resulted in the introduction higher level constructs used to improve the clarity of expression. Examples of these higher level constructs include:

- Branching statements: if-else, switch-case-break-default
- Looping statements: do-while, while, for

These were previous implemented using conditional and unconditional ISA instructions such as jle, jg, jmp, etc. Hence, using the reverse process, these construct can be deduced from the assembly language due to how they function.

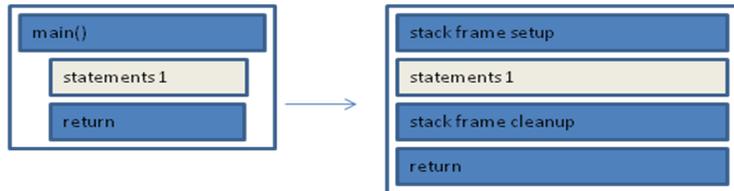
Note:

- In order to be able to identify the code constructs in disassembled code, compiler optimizations need to be turned off, as the optimized compiled code doesn't enable direct identification of the various code constructs described below.
- The C/C++ programming language is utilized to generate the various code constructs.
- The Intel ISA is utilized for the disassembled code.
- The descriptions of the various code constructs will comprise of:
  - A description & illustration of the construct.
  - Code listing in a high level language used to illustrate the construct.
  - Screen shot the disassembled code & its visualization (Section 3.6 discusses the visualization methodology).
- Since additional code is included in the disassembled programs, only the relevant disassembled code will be highlighted.
  - Red highlights will indicate the relevant disassembled code.
  - Blue highlights will indicate the relevant code constructs.
  - Green highlights (& none highlights) indicate the stack frame setup.
- Stack frames are illustrated in the 'basic program' section. They occur with function calls and involve a setup and cleanup process. The setup process save the current stack pointer in addition to allocating memory on the stack for local variables. The cleanup process restores the stack pointer.

## Empty Program

This section begins by illustrating the disassembled code for a basic program. The program calls the entry point function, main, and returns the hexadecimal value 1234. This program will be used as a basis for illustrating the various code constructs subsequently covered in this section.

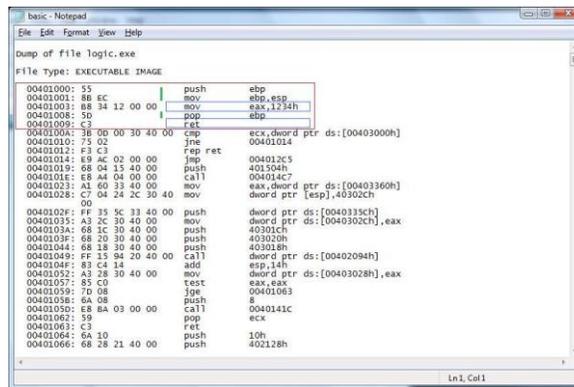
Figure 3.24: Basic Program



Code Listing 3.1: Basic Program

```
int main() {
    return 0x1234;
}
```

Figure 3.25: Disassembly of Basic Program



Note:

- Red highlight shows the disassembled program. Blue highlight show the return value as specified in the 'return 0x1234'. Green highlight shows the stack frame setup and cleanup.

Figure 3.26: Visualization of basic program (Illustration of a single sequential node)

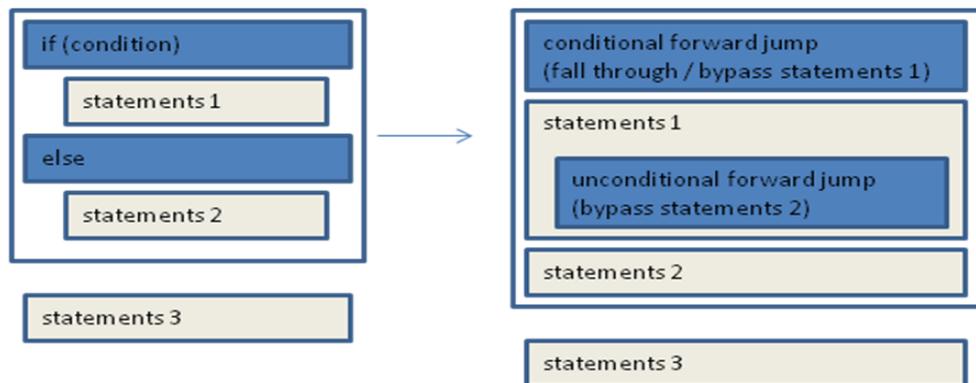


## If-Else

The 'if' section of an 'if-else' statement, provides the capability of skipping a section of code if a condition is not met. The 'else' section provides the capability of executing a sequence of code dependent on a condition being met or not.

Consequently, the presence of an 'if' statement could be identified by the presence of a conditional forward jump to a higher address as well as by an unconditional jump forward to skip the else section.

Figure 3.27: if statement



Code Listing 3.2: if statement

```
int main() {  
    int nCondition = 0;  
    int nValue = 0;  
  
    if (nCondition)  
        nValue = 1;  
    else  
        nValue = 0;  
  
    return 0x1234;  
}
```

Figure 3.28: Disassembly of 'if' statement

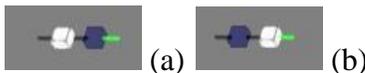
```

if - Notepad
File Edit Format View Help
Dump of file logic.exe
File type: EXECUTABLE IMAGE
00401000: 55          push     ebp
00401001: 8B EC      mov     ebp,esp
00401003: 83 EC 08   sub     esp,8
00401006: C7 45 F8 00 00 00 00 mov     dword ptr [ebp-8],0
00         00
0040100D: C7 45 FC 00 00 00 00 mov     dword ptr [ebp-4],0
00         00
00401014: 83 7D F8 00 cmp     dword ptr [ebp-8],0
00401018: 74 09     je      00401023
0040101A: C7 45 FC 01 00 00 00 mov     dword ptr [ebp-4],1
00         00
00401021: EB 07     jmp     0040102A
00401023: C7 45 FC 00 00 00 00 mov     dword ptr [ebp-4],0
00         00
0040102A: 8B 34 12 00 00 00 00 mov     eax,1234h
0040102F: 8B E5     mov     esp,ebp
00401031: 5D       pop     ebp
00401032: C2       ret
00401033: 3B 00 00 30 40 00 00 cmp     ecx,dword ptr ds:[00403000h]
00401039: 75 02     jne     0040103D
0040103B: F3 C3     rep ret
0040103D: E9 AC 02 00 00 00 jmp     004012EE
00401042: 68 2C 15 40 00 00 push   40152ch
00401047: E8 A3 04 00 00 00 call   004014EF
0040104C: A1 60 33 40 00 00 mov     eax,dword ptr ds:[00403360h]
00401051: C7 04 24 2C 30 40 00 mov     dword ptr [esp],40302ch
00         00
00401058: FF 35 5C 33 40 00 00 push   dword ptr ds:[0040335Ch]
0040105E: A3 2C 30 40 00 00 mov     dword ptr ds:[0040302Ch],eax
00401063: 68 1C 30 40 00 00 push   40301Ch
    
```

Note:

- Red highlight shows the disassembled program.
- Blue highlight shows the 'if' statement.
  - 'je 00401023' shows the conditional jump to the 'else'.
  - 'jmp 0040102A' shows the unconditional jump past the end of the 'if' statement, and is called as the last line of the 'if' portion.

Figure 3.29: Visualization of 'if' statement



Note:

- Image (a)
 

Currently selected node (white highlight) is the 'if' clause, which potentially transfers control to the 'else' code segment (green highlight) thus bypassing the 'if' code segment (black highlight between white & green highlight).
- Image (b)
 

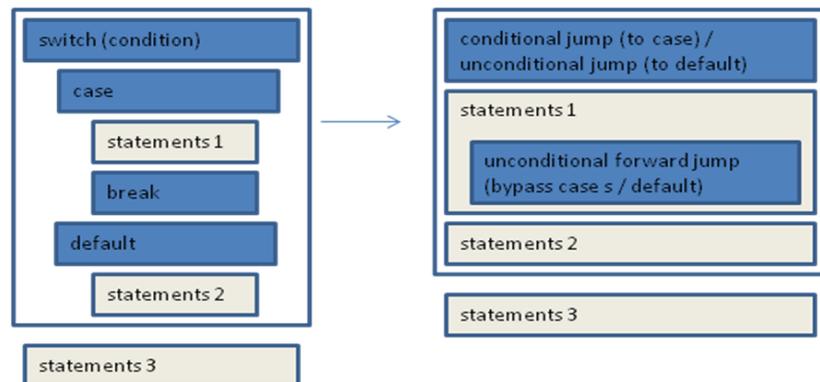
The selected node (white highlight) is an instruction within the 'if' code segment of the 'if' clause that causes the 'else' code segment to be bypassed once the 'if' code segment has completed. It is an unconditional jump to an instruction immediately after the 'if' statement (green highlight).

## Switch-Case-Break-Default

The 'switch' section provides the capability of jumping directly to the 'case' section, with the default being a last option 'case'. The 'break' section enables exiting the switch statement.

Thus unconditional jumps could indicate the presence of a switch statement and associated case/default labels. Unconditional forward jumps could indicate break statements.

Figure 3.30: switch statement



Code Listing 3.3: switch statement

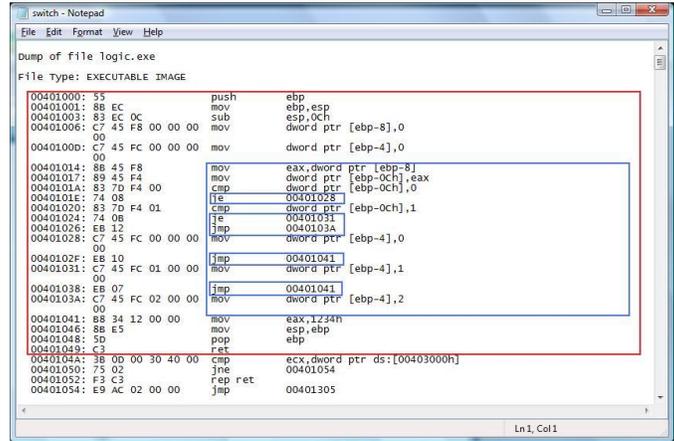
```
int main() {  
    int nCondition = 0;  
    int nValue = 0;  
  
    switch (nCondition) {  
    case 0:  
        nValue = 0;  
        break;  
    case 1:  
        nValue = 1;  
        break;  
    default:  
        nValue = 2;  
    }  
}
```

```

return 0x1234;
}

```

Figure 3.31: Disassembled ‘switch’ statement



Note:

- Red highlight shows the disassembled program.
- Blue highlight shows the ‘switch’ statement.
  - ‘je 00401028, je 00401031’ show the conditional jump to the ‘cases’, while ‘jmp 0040103A’ shows the unconditional jump to the ‘default’ portion.
  - ‘jmp 00401041’ shows the unconditional jump to after the end of the ‘switch’ statement, and are called from the end of each of the ‘cases’. The ‘default’ falls through.

Figure 3.32: Visualized ‘switch’ statement



Note:

- Image (a)

The selected node (white highlight) is the ‘switch’ statement’s check for ‘case 0’, which jumps to the body of the ‘case’ statement (green highlight) or falls through to check the next ‘case’.

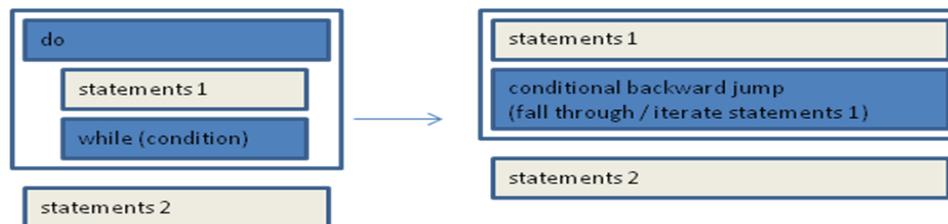
- Image (b)  
The selected node (white highlight) is the 'switch' statement's check for 'case 1', which jumps to the body of the 'case' statement (green highlight) or falls through to the check of the next 'case'. The next 'case' happens to be the 'default'.
- Image (c)  
The selected node (white highlight) is the 'switch' statement's check for the default statement, which unconditionally jumps to the body of the 'default' statement (green highlight).
- The last 2 nodes are the unconditional jumps at the end of each to the 2 'case' statements that are required to bypass the remainder of the 'switch' statement.

### Do-While Statement

The 'do-while' statements enable a loop to be executed at least once. At the end of the loop, the iteration condition is checked and the loop either terminates or continues.

Hence, a conditional backward jump would indicate the possibility of a do statement

Figure 3.33: do statement



Code Listing 3.4: do statement

```
int main() {
    int nCondition = 0;
    int nValue = 0;

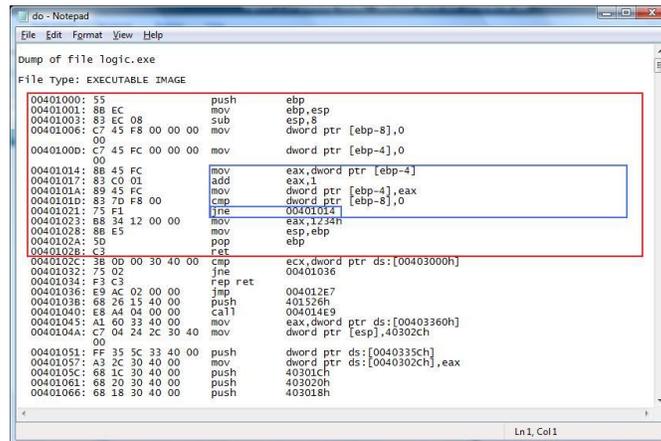
    do {
        nValue++;
    } while (nCondition);
```

```

return 0x1234;
}

```

Figure 3.34: Disassembled ‘do’ statement



Note:

- Red highlight shows the disassembled program.
- Blue highlight shows the ‘do’ statement.
  - ‘jne 00401014’ is the conditional jump back to the beginning of the loop or fall through if the loop is to exit.

Figure 3.35: Visualized ‘do’ statement



Note:

- Currently selected node (white highlight) is the ‘while’ code segment of the ‘do’ statement that jumps back to the beginning of the loop (red highlight).

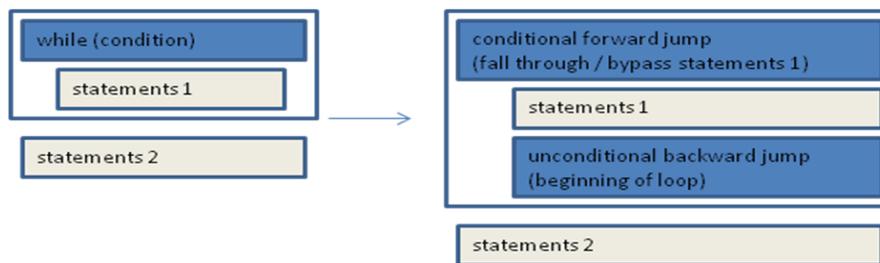
## While Statement

A ‘while’ statement provides the feature of enabling a condition to be checked prior to entering a loop. If the condition is met then the loop’s statements are executed. Prior to iterating through the loop, the condition is checked once again to determine whether the loop can terminate or iterate.

Hence, ‘while’ statements will have 2 conditional jumps; the 1<sup>st</sup> that checks the condition prior to entering the loop, and the 2<sup>nd</sup> that checks whether the loop is to terminate. The 1<sup>st</sup> one jumps beyond the 2<sup>nd</sup> hence bypassing the loop, while the 2<sup>nd</sup> jumps to just after the 1<sup>st</sup>.

Consequently, a conditional forward jump and/or a conditional backward jump indicate the potential presence of a ‘while’ statement.

Figure 3.36: while statement



Code Listing 3.5: while statement

```
int main() {  
    int nCondition = 0;  
    int nValue = 0;  
  
    while (nCondition) {  
        nValue++;  
    }  
  
    return 0x1234;  
}
```

Figure 3.37: Disassembled ‘while’ statement

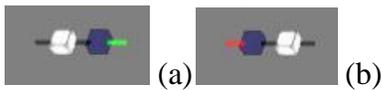
```

while - Notepad
File Edit Format View Help
Dump of file logic.exe
File type: EXECUTABLE IMAGE
00401000: 55      push   ebp
00401001: 8B EC   mov    ebp,esp
00401003: 83 EC 08 sub    esp,8
00401006: C7 45 F8 00 00 00 mov    dword ptr [ebp-8],0
0040100B: 00
0040100D: C7 45 FC 00 00 00 mov    dword ptr [ebp-4],0
00401014: 83 7D F8 00 cmp    dword ptr [ebp-8],0
00401018: 74 08   je     00401025
0040101A: 8B 45 FC mov    eax,dword ptr [ebp-4]
0040101D: 83 C0 01 add    eax,1
00401020: 89 45 FC mov    dword ptr [ebp-4],eax
00401023: EB EF   jmp    00401014
00401025: B8 34 12 00 00 mov    eax,1234h
0040102A: 8B E5   mov    esp,ebp
0040102C: 50     pop    ebp
0040102D: C3     ret
0040102E: 3E 00 00 30 40 00 cmp    ecx,dword ptr ds:[00403000h]
00401034: 75 02   jne   00401038
00401036: F3 C3   rep ret
00401038: E9 AC 02 00 00 jmp    004012E9
0040103D: 68 28 15 40 00 push  401528h
00401042: E8 A4 04 00 00 call  004014EB
00401047: A1 60 33 40 00 mov    eax,dword ptr ds:[00403360h]
0040104C: C7 04 24 30 40 00 mov    dword ptr [esp],40302Ch
00401053: FF 35 5C 33 40 00 push  dword ptr ds:[0040335Ch]
00401059: A3 2C 30 40 00 mov    dword ptr ds:[0040302Ch],eax
0040105E: 68 1C 30 40 00 push  40301Ch
00401063: 68 20 30 40 00 push  403020h
    
```

Note:

- Red highlight shows the disassembled program.
- Blue highlight shows the ‘while’ statement.
  - ‘je 00401025’ is the conditional jump to either bypass the loop or fall through into the loop.
  - ‘jmp 00401014’ is the unconditional jump to the beginning of the loop.

Figure 3.38: Visualized ‘while’ statement



Note:

- Image (a)
 

The selected node (white highlight) represents the ‘while’ statement that checks whether the condition is met or not. In the latter case, the check results in bypassing the ‘while’ loop and continuing execution after the end of the loop (green highlight).
- Image (b)
 

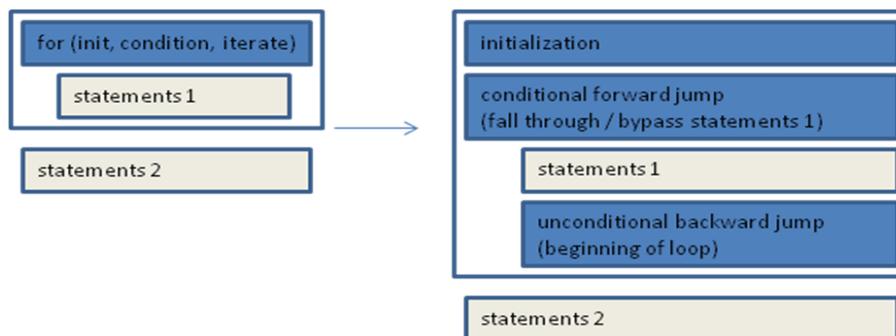
The selected node (white highlight) represents the end of the while statement, which causes the loop to begin again. Control is passed backwards to the beginning of the loop (red highlight).

## For Statement

A 'for' statement provides the feature of being able to initialize variables prior to entering a loop, in addition to specifying how the iteration check changes.

The initialize occurs prior to entering the loop and is performed only once. If the condition is met, the loop is entered, while if the condition is not met, the loop is bypassed. Hence, the condition checking behaves as a 'if' statement, while the loop behaves as a 'do' statement.

Figure 3.39: for statement



Code Listing 3.6: for statement

```
int main() {
    int nCondition = 0;
    int nValue = 0;

    for (nCondition = 0; nCondition < 10; nCondition++) {
        nValue++;
    }

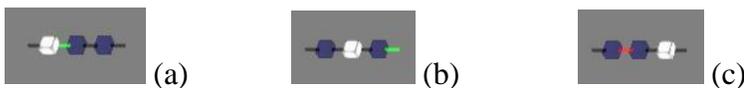
    return 0x1234;
}
```

Figure 3.40: Disassembled ‘for’ statement

Note:

- Red highlight shows the disassembled program.
- Blue highlight shows the ‘for’ statement.
  - ‘jmp 00401026’ is the initialization portion of the ‘for’ statement
  - ‘jge 00401037’ is the conditional jump to either end the loop, or continue from the beginning of the loop. It is the condition portion of the ‘for’ statement.
  - ‘jmp 0040101D’ is the unconditional jump to the beginning of the loop.

Figure 3.41: Visualized ‘for’ statement



Note:

- Image (a)
 

The selected node (white highlight) is an unconditional jump that bypasses the iteration portion of the ‘for’ statement that increments the counter by 1. It jumps to a location just before the ‘for’ condition is checked (green highlight).
- Image (b)
 

The selected node (white highlight) is the checking of the ‘for’ condition, which can potentially exit the loop, to the instruction immediately after the loop (green highlight).

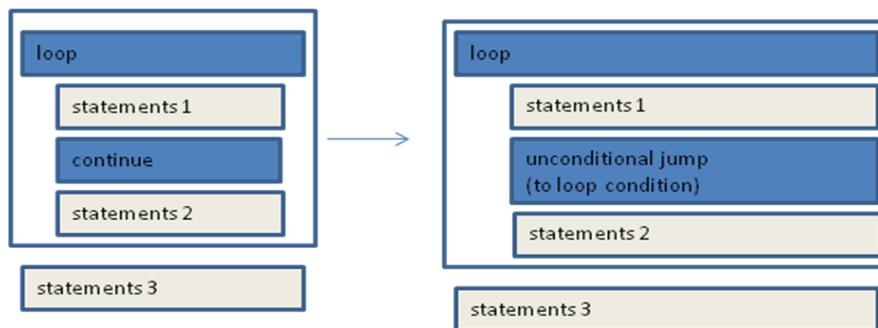
- Image (c)

The selected node (white highlight) is the instruction located at the end of the ‘for’ loop that transfers control to the beginning of the loop (red highlight); first incrementing the counter, and then checking the condition.

### Loop Support Statements: Continue Statement

Loop statements provide for the use of continue statements. A continue statement cause the loop to begin from the start, and thus can be represented with an unconditional jump to the start of the loop.

Figure 3.42: continue statement



Code Listing 3.7: continue statement

```
int main() {  
    int nCondition = 0;  
    int nValue = 0;  
  
    while (nCondition) {  
        nValue++;  
        continue;  
        nValue += 2;  
    }  
  
    return 0x1234;  
}
```

Figure 3.43: Disassembled 'continue' statement

```

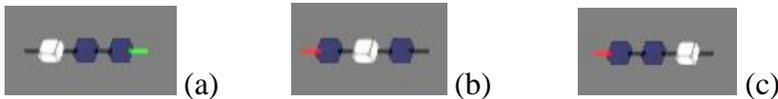
Dump of file logic.exe
File Type: EXECUTABLE IMAGE

00401000: 55          push    ebp
00401001: 8B EC       mov     ebp,esp
00401003: 83 EC 08   sub     esp,8
00401006: C7 45 F8 00 00 00 mov    dword ptr [ebp-8],0
0040100D: C7 45 FC 00 00 00 mov    dword ptr [ebp-4],0
00401014: 83 70 F8 00 cmp    dword ptr [ebp-8],0
00401018: 74 16     je     00401030
0040101A: 8B 45 FC   mov    eax,dword ptr [ebp-4]
0040101D: 83 C0 01   add    eax,1
00401020: 89 45 FC   mov    dword ptr [ebp-4],eax
00401023: EB EF     jmp    00401014
00401025: 8B 40 FC   mov    ecx,dword ptr [ebp-4]
00401028: 83 C1 02   add    ecx,2
0040102B: 89 40 FC   mov    dword ptr [ebp-4],ecx
0040102E: EB F4     jmp    00401014
00401030: B8 34 12 00 00 mov    eax,1234h
00401035: 8B E5     mov    esp,ebp
00401037: 5D       pop    ebp
00401038: C3       ret
00401039: 3B 00 00 30 40 00 cmp    ecx,dword ptr ds:[00403000h]
0040103F: 73 02     jne   00401043
00401041: F3 C3     rep ret
00401043: E9 4C 02 00 00 jmp    004012F4
00401048: 68 32 15 40 00 push  401532h
0040104D: E8 A3 04 00 00 call  004014F5
00401052: A1 60 33 40 00 mov    eax,dword ptr ds:[00403360h]
00401057: C7 04 24 2C 30 40 mov    dword ptr [esp],40302Ch
00
  
```

Note:

- Red highlight shows the disassembled program.
- Blue highlight shows the loop with the 'continue' statement.
  - 'jmp 00401014' is the unconditional jump to the beginning of the loop.

Figure 3.44: Visualized 'continue' statement



Note:

- Image (a)
 

The selected node (white highlight) is the check at the beginning of the loop, which can potentially bypass the entire loop to an instruction immediately after the loop (green highlight).
- Image (b)
 

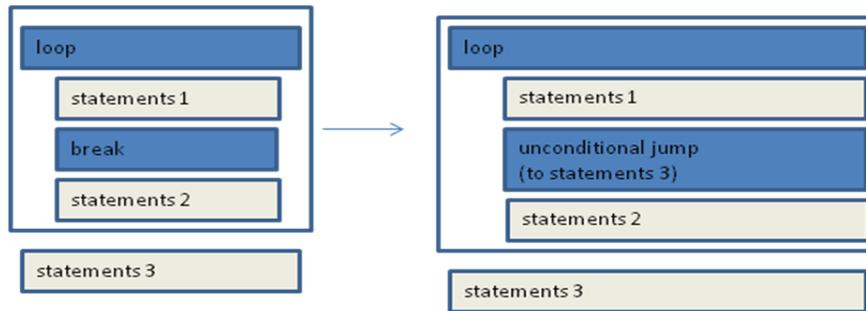
The selected node (white highlight) is the 'continue' statement that unconditionally transfers control back to the beginning of the loop (red highlight).
- Image (c)
 

The selected node (white highlight) is an instruction at the end of the loop that causes the loop to iterate by transferring control back to the beginning of the loop (red highlight).

## Loop Support Statements: Break Statement

Loop statements provide support for the use of break statements, which cause the loop to exit. From the assembly level perspective they can be indicated with an unconditional jump to a location after the end of the loop.

Figure 3.45: break statement



Code Listing 3.8: break statement

```
int main() {  
    int nCondition = 0;  
    int nValue = 0;  
  
    while (nCondition) {  
        nValue++;  
        break;  
        nValue += 2;  
    }  
  
    return 0x1234;  
}
```

Figure 3.46: Disassembled 'break' statement

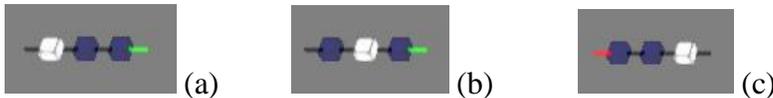
```

break - Notepad
File Edit Format View Help
Dump of file log1c.exe
File type: EXECUTABLE IMAGE
00401000: 55 EC push ebp
00401001: 8B EC mov ebp,esp
00401003: 83 EC 08 sub esp,8
00401006: C7 45 F8 00 00 00 mov dword ptr [ebp-8],0
00
0040100D: C7 45 FC 00 00 00 mov dword ptr [ebp-4],0
00
00401014: 83 7D F8 00 cmp dword ptr [ebp-8],0
00401018: 74 16 je 00401030
0040101A: 8B 45 FC mov eax,dword ptr [ebp-4]
0040101D: 83 C0 01 add eax,1
00401020: 89 45 FC mov dword ptr [ebp-4],eax
00401023: EB 08 jmp 00401030
00401025: 8B 4D FC mov ecx,dword ptr [ebp-4]
00401028: 83 C1 02 add ecx,2
0040102B: 89 4D FC mov dword ptr [ebp-4],ecx
0040102E: EB E4 jmp 00401014
00401030: 8B 34 12 00 00 mov eax,1234h
00401033: 8B E5 mov esp,ebp
00401037: 5D pop ebp
00401038: C3 ret
00401039: 3B 0D 00 30 40 00 cmp ecx,dword ptr ds:[00403000h]
0040103F: 75 02 jne 00401043
00401041: F3 C3 rep ret
00401043: E9 AC 02 00 00 jmp 004012F4
00401048: 68 32 15 40 00 push 401532h
0040104D: E8 A3 04 00 00 call 004014F5
00401052: A1 60 33 40 00 mov eax,dword ptr ds:[00403360h]
00401057: C7 04 24 2C 30 40 mov dword ptr [esp],40302Ch
00
Ln1, Col1
    
```

Note:

- Red highlight shows the disassembled program.
- Blue highlight shows the loop with a 'break' statement.
  - 'jmp 00401030' is the unconditional jump to after the end of the loop.

Figure 3.47: Visualize 'break' statement



Note:

- Image (a)
 

The selected node (white highlight) is the check at the beginning of the loop, which can potentially bypass the loop to an instruction immediately after the loop (green highlight).
- Image (b)
 

The selected node (white highlight) is the 'break' statement that exits the loop. It is an unconditional jump to an instruction immediately after the loop (green highlight).
- Image (c)
 

The selected node (white highlight) is an instruction at the end of the loop that causes the loop to iterate by unconditionally jumping back to the loop condition check (red highlight).

## 3.10 Visual

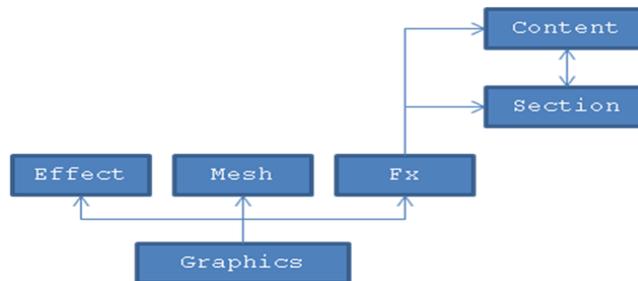
### 3.10.1 Internal Structure

The internal visualization-related components are designed in a modular manner to enable extension of functionality. They include the following modules:

- Graphics – deals with the initialization and management of graphic resources & components of the system.
- Effect – deals with how the GPU renders content.
- Mesh – provides the vertices used to describe objects.
- Fx – provides the base functionality for the visualizations from which additional functionality is implemented by sub classing. The ‘Section’ sub class manages a collection of nodes, while the ‘Content’ sub class manages a collection of sections.

The different modules are implemented in layers.

Figure 3.48: Graphics Engine Internal Structure



### 3.10.2 Interaction Flow

The general flow of the program is illustrated in the Section 3.4.1.

### 3.10.3 User Interface

This section describes the various interfaces that are presented from the start of the application to visualization & analysis of a loaded executable file.

### **Main Screens (User Interface Stages)**

To simplify interaction, drag-and-drop is utilized in the loading of content. The start screen is initially reduced in size to enable files to be dragged and dropped from the desktop. Once files are received, the view is then enlarged.

Figure 3.49: Startup screen (used for drag & drop operations)



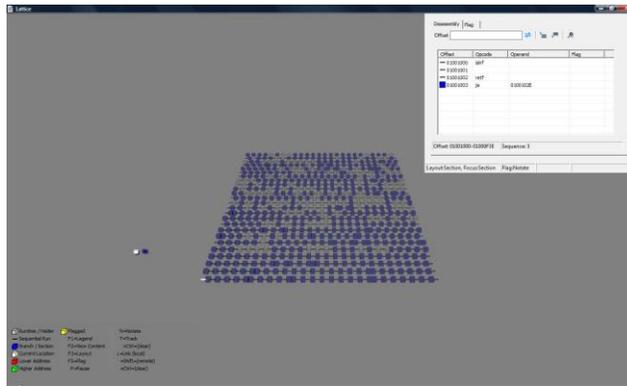
Figure 3.50: Visualization screen (used for visualization & analysis)



### **Content Generation & Loading**

Prior to visualization & analysis, content needs to be generated from the executable file that is loaded, if the content has not already been generated. Once generated the content is loaded into the application for visualization and subsequently analysis purposes. Below is an illustration of the process.





b. With analysis section shown

The various screen areas are described next.

### Visualization Layout

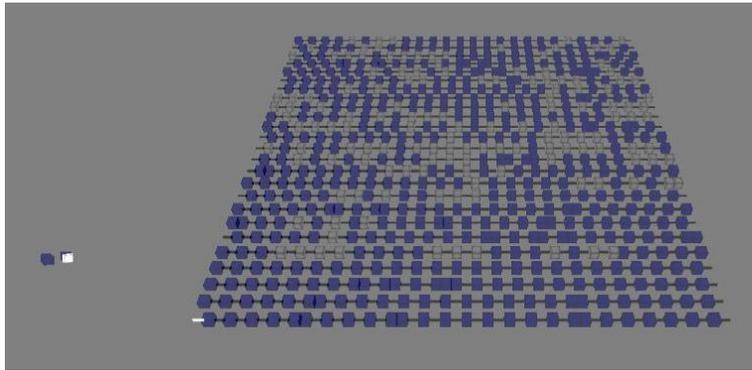
The visualization area is divided into 4 regions, namely

- Legend – located at the lower left region, provides a quick view of the interface notation and commands.
- Node – initially located at the centre of the view, provides the visualization of the disassembled code. This view can be alternated with the ‘Section’ view.
- Section – initially located to the left of the ‘Node’ section, provides a broad overview of the current node location with reference to the entire code being analyzed. This view can be alternated with the ‘Node’ view.
- Analysis – located at the upper right region, provides various tools that can be used to provide the textual view of the node’s disassembled code together is notational information. In addition, it provides an interface for analyzing the visualized code.

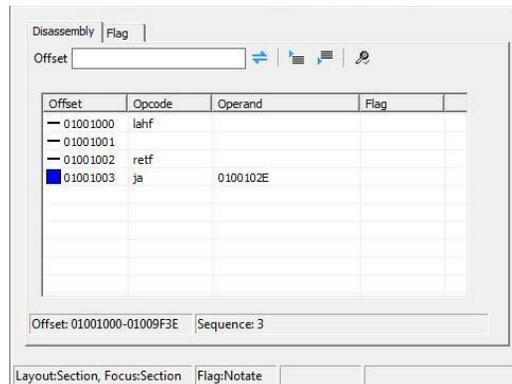
Figure 3.53: Visualization Layout Components



a. Legend View



b. Node (right) & Section (left) View; focus on 1<sup>st</sup> node and 1<sup>st</sup> section



c. Disassembly View

Note:

- The offset provides a means of specified an exact offset to navigation to.
- The displayed buttons provide the following functionality respectively:
  - Synchronize the specified offset with the visualization view when toggled on. If the toggle state is off (as currently displayed) the disassembly information specified by the offset is only displayed in the list view below the offset edit location.
  - Display the instruction just before the top-most displayed instruction in the disassembly view listing.
  - Display the instruction just after the bottom-most displayed instruction in the disassembly view listing.
  - Clear any notation flags currently assigned to an instruction.

- Disassembly listing that displays disassembly information beginning from either the currently selected node in the ‘Node’ view or the specified offset.
- Additional information is provided in the status bars such as the range of potential offset addresses.

The screenshot shows a window titled 'Disassembly Flag' with a 'Category' dropdown set to 'Code'. Below the dropdown is a table with two columns: 'Description' and 'Count'. The table contains the following data:

Description	Count
Loop (Jump Backward)	151
Else/Switch Statement (Jump Forward)	116
Immediate Loop (Jump Backward)	8
Do/While	170
If	1137
Nop	0
Hotpatch	0
Frame Pointer	0

At the bottom of the window, there are status bars with labels: 'Layout:Section, Focus:Section' and 'Flag:Notate'.

#### d. Analysis View

Note:

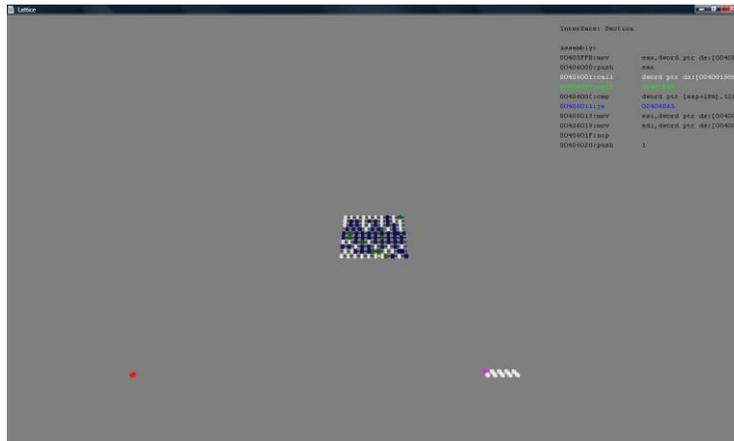
- The category provides a means of filtering the type of analysis to perform. Currently, the 2 categories are either ‘General’ for a non-specific based analysis type, and ‘Code’ for code based specific analysis.
- The displayed button provides a means of clearing the states of selected analyses.
- The display view below the category provides a means of interacting with the visualization by selecting a listed analysis (via double clicking to select or toggle on/off), which is then visualized in the visualization area of the ‘Node’ and ‘Section’.

### **Evolving Design Influencing Final Design**

The design and implementation of the program underwent several iterations. During the reviews between the iterations, features were modified due to feedback. Both the visual as well as the functionality features were affected.

Below are some screen features that were altered during the development cycle.

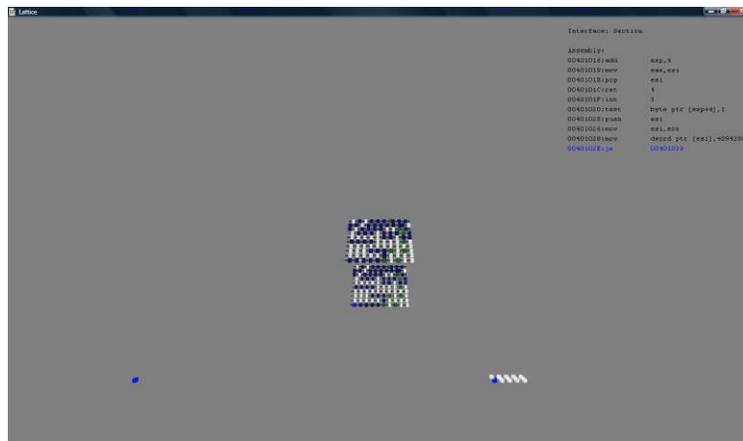
Figure 3.54: Initial screen layout



Note:

- Bottom left & right portions eventually dropped from final program, as they did not contribute any assistance. However, an enhancement based on sections was incorporated by:
  - Aggregating the 2 aspects. It enabled the addition of a legend on the screen.
  - Incorporating a switching feature to toggle the aggregated aspects and the main content.
- Text section enhanced to enable searching.

Figure 3.55: Initial content animation (when traversing different sections of the code)



Note:

- Animation dropped from final program, as it did not enable intuitive navigation.

## CHAPTER 4 – RESULTS

### 4.1 Introduction

Once an assembly listing has been imported into the application, visualization and data analysis can then be performed to gather information about the disassembled executable code.

This chapter is structure as follows:

- Description of program that will be compiled and then disassembled (Section 4.2).
- Visualization & Analysis of the disassembled program (Section 4.3).

### 4.2 Test Program

Code Listing 4.1: Program used for testing

```
int Add(int n1, int n2);
int main() {
    int nSum = 0;
    int nCount = 10;
    int nCondition = 1;
    int nValue = 0;

    //sums the numbers 1 to 10
    for (int i = 1; i < 11; i++)
        nSum += i;

    //adds 2 numbers
    Add(nSum, 2);

    //loops decrementing count
    do {
        nCount--;
    } while (nCount);
```

```
//condition check
if (nCondition) {
    nValue++;
} else {
    //continuous loop
    while (1) {
        }
    }

    return 0x1234;
}
int Add(int n1, int n2) {
    return n1 + n2;
}
```

Note:

- The above program was compiled with optimizations disabled. This was to prevent the compiler from modifying the code, which would have resulted in the disassembly code not being directly reference with the original source code. Illustration of the mapping between the visualization and source code would not be clear.
- The program was also compiled without debug symbols present. Executable programs that are release for public use rarely contain debug symbols as these assist in the process of reverse engineering by providing function name hints.
- Once compiled the program was disassembled. The output is shown in Figure 4.1 below.

Figure 4.1: Disassembled test code

```

Dump of file logic.exe
File Type: EXECUTABLE IMAGE

00401000: 55          push    ebp
00401001: 8B EC      mov     ebp,esp
00401003: 83 EC 14   sub     esp,14h
00401006: C7 45 F0 00 00 00 mov     dword ptr [ebp-10h],0
0040100D: C7 45 FC 0A 00 00 mov     dword ptr [ebp-4],0Ah
00401014: C7 45 F4 01 00 00 mov     dword ptr [ebp-0Ch],1
0040101B: C7 45 F8 00 00 00 mov     dword ptr [ebp-8],0
00401022: C7 45 EC 01 00 00 mov     dword ptr [ebp-14h],1
00401029: EB 09      jmp     00401034
0040102B: 8B 45 EC   mov     eax,dword ptr [ebp-14h]
0040102E: 83 C0 01   add     eax,1
00401031: 89 45 EC   mov     dword ptr [ebp-14h],eax
00401034: 83 7D EC 0B cmp     dword ptr [ebp-14h],0Bh
00401038: 7D 0B      jge     00401045
0040103A: 8B 4D F0   mov     ecx,dword ptr [ebp-10h]
0040103D: 03 4D EC   add     ecx,dword ptr [ebp-14h]
00401040: 89 4D F0   mov     dword ptr [ebp-10h],ecx
00401043: EB E6      jmp     0040102B
00401045: 6A 02      push   2
00401047: 8B 55 F0   mov     edx,dword ptr [ebp-10h]
0040104A: 52        push   edx
0040104B: E8 40 00 00 00 call   00401090
00401050: 83 C4 08   add     esp,8
00401053: 8B 45 FC   mov     eax,dword ptr [ebp-4]
00401056: 83 E8 01   sub     eax,1
00401059: 89 45 FC   mov     dword ptr [ebp-4],eax
0040105C: 75 F5      jne     00401053
0040105E: 83 7D F4 00 cmp     dword ptr [ebp-0Ch],0
00401062: 74 0B      je      0040106F
00401064: 8B 4D F8   mov     ecx,dword ptr [ebp-8]
00401067: 83 C1 01   add     ecx,1
0040106A: 89 4D F8   mov     dword ptr [ebp-8],ecx
0040106D: EB 0B      jmp     0040107A
0040106F: BA 01 00 00 00 mov     edx,1
00401074: 85 D2      test   edx,edx
00401076: 74 02      je      0040107A
00401078: EB F5      jmp     0040106F
0040107A: B8 34 12 00 00 mov     eax,1234h
0040107F: 8B E5      mov     esp,ebp
00401081: 5D        pop    ebp
00401082: C3        ret
00401083: CC        int    3
00401084: CC        int    3
00401085: CC        int    3
00401086: CC        int    3
00401087: CC        int    3
00401088: CC        int    3
00401089: CC        int    3
0040108A: CC        int    3
0040108B: CC        int    3
0040108C: CC        int    3
0040108D: CC        int    3
0040108E: CC        int    3
0040108F: CC        int    3
00401090: 55        push   ebp
00401091: 8B EC      mov     ebp,esp
00401093: 8B 45 08   mov     eax,dword ptr [ebp+8]
00401096: 03 45 0C   add     eax,dword ptr [ebp+0Ch]
00401099: 5D        pop    ebp
0040109A: C3        ret

```

## 4.3 Visualization & Analysis

### 4.3.1 Visualization

Once loaded into the application, the disassembly results in the following visualization.

Figure 4.2: Test program's visualization



Note:

- The dimension is set to 4 (hence 4 nodes along the X-axis and 2 along the Z-axis).
- There are 3 sections.
- By default upon startup, the 1<sup>st</sup> node and section is highlighted (white highlight).

### Description

The visualization of the program illustrated in Figure 4.2 above, shows 4 sequential portions as well as 4 control nodes within the 1<sup>st</sup> section. Below is a summary of the nodes in the order they appear (incremented along the X-axis then the next row in the Z-axis),

Table 4.1: 1<sup>st</sup> 8 nodes; 1<sup>st</sup> section

Node	Type	Description
1	Sequential	All the instructions prior to reaching the 'for' loop
2	Control	Beginning of the 'for' loop. The control bypasses the 'for' iteration code (Node 3), and transfers control to the 'for' condition, which determines whether the 'for' body will be entered. It is an unconditional jump to the 'for' condition check at the end of the body of the 'for' loop (Node 4)
3	Sequential	The 'for' iteration code

4	Control	The 'for' condition check that transfers control by either, <ul style="list-style-type: none"> <li>• Fall through if the condition is met and the 'for' body is to be entered Node 5 is the body of the 'for' loop</li> <li>• Bypasses the 'for' body if the condition is not met, and transfers control to the beginning of the next sequential code (Node 7) that is not part of the 'for' loop. This is illustrated in Figure 4.3a</li> </ul>
5	Sequential	Body of the 'for' loop
6	Control	Refers to the unconditional jump to the 'for' iteration code located at Node 3. This is illustrated in Figure 4.3b
7	Sequential	Refers to the instructions after the 'for' loop, in this case the function call setup for the 'Add' function that passes function arguments and prepares the stack frame
8	Control	Calling of the 'Add' function

Note:

- Node 1 are instructions prior to the 'for' loop
- Nodes 2-6 comprise the 'for' loop
- Nodes 7-8 comprise the 'add' function

### 4.3.2 Analysis

This section discusses the following visualization analyses:

- Navigation – next location highlighting
- Potential source locator
- Loop identification
  - Loops
  - Loops without intermediate branching
- Mapping
  - Function calls

## Navigation – Next location highlight

The visualization interface provides the feature of being able to navigate through a program's code in various forms while providing current location highlight, potential branching location highlight, and overall location highlighting. Refer to Table 4.1 for a description of the nodes.

Figure 4.3: Navigation & next location highlighting



a. Next location highlight

Note:

- The dimension is set to 4.
- The selected node (white highlight) is the disassembled 'jge 00401045' instruction that represents the source code 'for (int i = 1; i < 11; i++)' instruction's condition check, which either enters the loop or bypasses the loop if the check fails (either at the 1<sup>st</sup> check or upon loop completion).
- The potential branch location (green highlight) is the 1<sup>st</sup> instruction after the 'for' loop. It is represented by the disassembled 'push 2' instruction that represents the parameter passing of the source code 'Add(nSum, 2)'.
- In the 'Section' view, the current section is the 1<sup>st</sup> (white highlight), implying that the current analysis location is found near the start of the program relative to the whole program.



b. Validating intermediary visualization

Note:

- The selected node (white highlight) is the disassembled ‘jmp 0040102B’ instruction located at the end of the ‘for’ that enables the loop to iterate. It returns to a location just before the ‘jge 00401045’ instruction in order to increment the counter.

### Potential source locator

Navigation through a program’s flow is usually in a forward direction, i.e. from the current location to potential next locations either sequentially or by control branching. However, the capability of being able to identify potential areas that could have resulted in a branch to the current location is beneficial.

Figure 4.4: Potential source locator



a. Instruction location accessible from 2 different locations

Note:

- The dimension is set to 4.
- The back track feature of the program has been run on the selected node (white highlight) located in the 2<sup>nd</sup> section (white highlight in the ‘Section’ view).
- The selected node (white highlight), which is the ‘mov edx,1’ instruction at offset ‘0040106F’ is the beginning of the ‘while’ loop. The instruction moves a ‘1’ to the register ‘edx’, which is used to check the condition ‘while(1)’.
- This instruction location can be reached via two ways as indicated by the 2 yellow highlights.
  - The 1<sup>st</sup> is from within the current section as indicated by the ‘yellow’ node in the ‘Node’ view. This instruction is the ‘je 0040106F’ at offset ‘00401062’. This is the jump of the ‘if’ statement that jumps the ‘if’ portion and goes to the ‘else’ portion of the ‘if’ statement.

- The focus of the selected ‘Section’ indicated by the white highlight, supersedes its other notation of being yellow highlighted, it contains a potential source location.



b. 1<sup>st</sup> location from ‘if’ statement

- The 2<sup>nd</sup> is from a location in the next ‘Section’ as indicated by the highlighted yellow ‘Section’ in the ‘Section’ view, see figure (a). This particular ‘Section’ is then selected, indicated by the new ‘white’ highlight and a change in the ‘Node’ view (the 1<sup>st</sup> node in that ‘Section’ is highlighted, which in this case happens to be the desired node).
- The selected node in the ‘Node’ view is a jump to a lower address in a different ‘Section’. This is illustrated by the red highlight in the ‘Section’ view. This instruction is the ‘`jmp 0040106F`’ instruction that is located at the end of the ‘while’ loop that transfers control to the beginning of the loop.

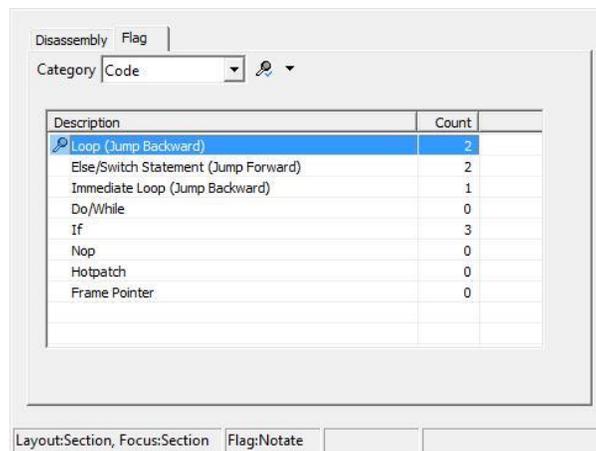


c. 2<sup>nd</sup> location from ‘while’ statement

## Loops – Unconditional Backward Jump

Loops can be identified by a backward transfer of control. In the analysis of code, it may be useful to identify potential loops in the program, for example to determine a program's complexity or runtime implications for performance analysis.

Figure 4.5: Loops



The screenshot shows a software interface with a 'Disassembly' tab and a 'Flag' dropdown. Below this is a 'Category' dropdown set to 'Code'. A table displays the following data:

Description	Count
Loop (Jump Backward)	2
Else/Switch Statement (Jump Forward)	2
Immediate Loop (Jump Backward)	1
Do/While	0
If	3
Nop	0
Hotpatch	0
Frame Pointer	0

a. Setting the analysis parameter

Note:

- In the 'Analysis' view the 'Code' category is selected, and the 'Loop (Jump Backward)' item selected by double-clicking.
- The 'Run' command is executed to run through the disassembled code to generate information.



b. 2 loops have been identified

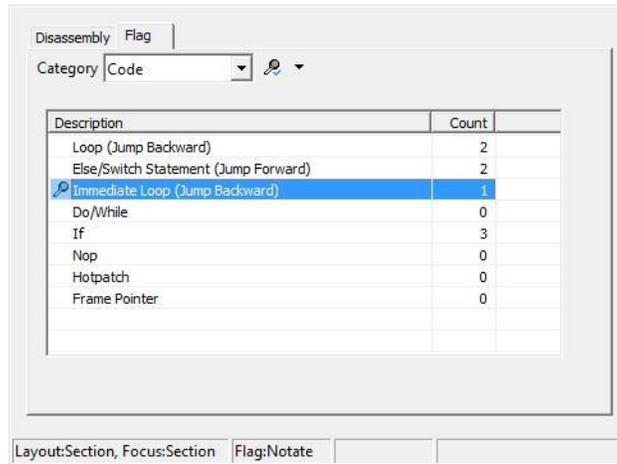
Note:

- The dimension is set to 10; consequently the number of sections based on this dimension is 1, as illustrated by the single white-highlighted ‘section’ at the top left.
- 2 loops have been identified (yellow highlights), which represent the ‘do’ and ‘while’ statements.
- The white-highlighted ‘sequential node’ is the default selected node at the start of the nodes; upon loading a visualization or selecting a ‘section’ the default selected node is always the 1<sup>st</sup> in the program (also if 1<sup>st</sup> ‘section’ is selected) or the 1<sup>st</sup> node in a given ‘section’ if the subsequent ‘sections’ are selected.
- The 1<sup>st</sup> yellow-highlighted node, the 6<sup>th</sup> node from the start of the program, is the unconditional backward jump to the ‘for’ iteration code. It is the jump located at ‘0x00401043’ jumping backward to the location ‘0x0040102B’, which is the code that increments the loop condition by a value of 1 (location 0x0040102E).
- The 2<sup>nd</sup> yellow-highlighted node, the 17<sup>th</sup> node from the start of the program (2<sup>nd</sup> last node), is the unconditional jump at the end of the ‘while’ loop that transfers control back to the start of the loop to check the loop condition in order to determine whether to enter the loop again or not. It is the instruction at location ‘0x00401078’, which transfers control back to the loop condition at location ‘0x0040106F’. The instruction at location ‘0x00401076’ that provides the option to exit the loop is present since optimization was disabled; the compiler includes the instruction since the while condition check results in a value that needs to be checked regardless of whether it changes or not. An optimizing compiler would notice that the loop condition is an immediate value that is known at compile time.

### **Loops without intermediate control instruction**

In some cases, a perpetual loop may be created. For example, a program’s main loop continuously runs until explicitly it is either terminated by the program or by the user. Identification of such loops may be required in order to identify loops that could run indefinitely, either by intentional or accidental design.

Figure 4.6: Potential endless loop



a. Setting the analysis parameter

Note:

- In the ‘Category’ drop down, the ‘Code’ option is selected, and the ‘Immediate Loop (Jump Backward)’ item selected by double clicking it.
- The ‘Run’ command is executed to analyze the program.



b. 1 potential endless loop identified

Note:

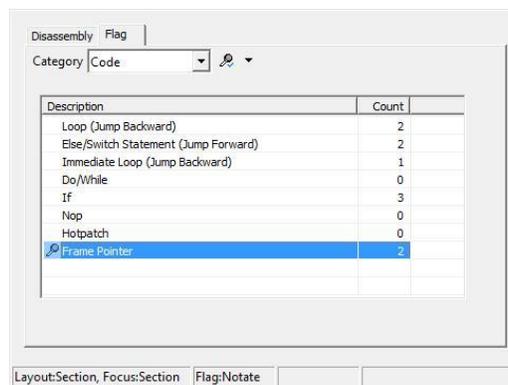
- The dimension is set to 10 resulting in the number of ‘section’ being 1 (indicated by the single white-highlighted ‘section’ at the top left).
- 1 potential item (yellow highlight) has been identified. It is the ‘do’ statement. This statement has been implemented without a potential exit in the form of a branching out of the loop.
- The yellow-highlighted node (10<sup>th</sup> node) refers to the instruction located at ‘0x0040105C’ that transfers control back to the beginning of the loop at location ‘0x00401053’; involves decrementing the loop condition variable.

- Between the beginning of the loop and the check at the end of the loop there are no intermediate ways in which the loop can exit. The loop condition is the only way of potentially exiting the loop.
- Though this is a functionality of the program, it indicates a potential loop that may loop more than intended dependent on the value passed to the loop condition variable. Hence by highlight such loops extra attention could be place to determine whether the loop exit.

## Mapping – Function Calls

Program functionality is usually implemented in functions for modular design. Identification of these functions helps to narrow down areas of analysis. They also indicate a location where local variables will be defined.

Figure 4.7: Function call mapping



Description	Count
Loop (Jump Backward)	2
Else/Switch Statement (Jump Forward)	2
Immediate Loop (Jump Backward)	1
Do/While	0
If	3
Nop	0
Hotpatch	0
Frame Pointer	2

a. Setting the analysis parameter

Note:

- In the ‘Analysis’ view, select the ‘Flag’ tab. In the ‘Category’ drop down, select the ‘Code’ item. Select the ‘Frame Pointer’ item by double clicking on it.
- Generate the program information by executing the ‘Run’ command.



b. Identified functions

Note:

- 2 stack frame pointer instructions are identified indicated by the yellow highlights. These refer to the 2 functions within the program, i.e. the 'main' function, which provides the program's entry point, and the 'Add' function.
- The 1<sup>st</sup> yellow highlight (1<sup>st</sup> node), refers to the program's initial stack frame generated by the 'main' function. It refers to the instructions beginning at the location '0x00401000'.
- The 2<sup>nd</sup> yellow highlight (last node), refers to the 'Add' function's stack frame. The function is placed at the end of the program, and begins at the location '0x00401090'.
- Identification of stack frames provides an insight into how a program's functionality is implemented as well as how the program flows. This is because functionality is usually implemented in modules.

## **CHAPTER 5 - CONCLUSION**

### **5.1 Introduction**

As visual processing capability increases with advances in processing speeds of both the Central Processing Unit (CPU) and Graphics Processing Unit (GPU), the visual analysis of information is increasingly possible to greater extents. Software visualization benefits from these advances.

This research explored a lattice-based metaphor for software visualization and analysis of an executable's disassembled code. The feasibility of the lattice structure for both tasks was implemented and illustrated. The metaphor design was based on the abstraction of a processor's Instruction Set Architecture (ISA). Various basic code constructs that form building blocks for programs were derived from basic program flow, and then visualized and analyzed. A test program was then implemented in a high level language (C/C++), disassembled, visualized, and analyzed, with the goal of extracting potentially useful information.

An additional outcome of the research effort resulted in a potential process/methodology that could be used in the design of new software visualization metaphors. This involved the sequence of abstraction of the underlying software aspects to be visualized, generation of the basic building blocks that are extendable to software built using the basic constructs, notation development, and finally design of an interaction mechanism.

The test results illustrated that the proposed metaphor provided a means of visually interacting with disassembled code to both obtain a better insight as well as extract potentially useful information for further analysis.

### **5.2 Achievements**

#### **Problem Identification**

The quantity of executable code (without access to the original source code) that needs to be analyzed is increasing as new functionality is required and implemented. Furthermore, the time frames needed for analyzing the software is decreasing. This results in a need for new methods

of analyzing and extracting potentially useful information. Visualization provides one alternative. However, the metaphors that are usually utilized are borrowed from information visualization and hence may not be ideally suited for software visualization in which code aspects rather than numerical data needs to be analyzed.

With the increasing capability of both CPUs and GPUs, previously resource intensive graphic visualization is now possible. This enables more use of a system's graphical capability to enable use of 3D graphics for visualization, resulting in an increased spatial space for analysis, in addition to the use of 3D metaphors to enhance analysis of information.

Access to a program's source code is not always available as publishers usually retain it, but release the executable code. However, with disassemblers for a target's platform, the executable code can be converted to its equivalent assembly code with a one-to-one mapping. Hence by obtaining both an executable and its platform's disassembler, any program can be potentially analyzed.

The research project focused on the design and implementation of a 3D metaphor for visualization of binary code to enhance this capability.

### **Literature Review & Scope Identification**

Visualization of information involves mapping of information aspects to visualization metaphors. In scientific computing, the metaphors are already predefined due to the structured nature of the information they represent. However, in information visualization, the information being represented is abstract in nature with no predefined structure. Software visualization usually utilizes information visualization metaphors, which are not always suitable. Software though abstract in nature, inherently has a structure.

The literature reviewed indicated a high reliance on information visualization metaphors. The research project proposed that software visualization would involve a hybrid of both scientific as well as information visualization. Hence it attempted to map the inbuilt structure of binary code to a predetermined structure based on a lattice.

## **Methodology**

Software comprises of a collection of instructions based on a processor's ISA, regardless of the implementation language. These instructions can be grouped into constructs that form building blocks for programs. Examples include branching constructs, loop constructs, function calling constructs.

The ISA instructions were broadly categorized as either sequential or control type. The former resulted in the next instruction being executed. With the latter, the subsequent instruction to be executed was an option of 2 possible instructions based on the result of the control check. These 2 categories resulted in the generation of 2 models that mapped directly to the lattice structure, making it feasible for software visualization. This was in the form of nodes (for control type) of the lattices as well as the spaces/links (for sequential type) within.

Based on this, a whole program could be concisely represented using a lattice structure, with additional functionality being built on top of the lattice metaphor.

Furthermore, the process used in the research project can be generalized, and is not only applicable to the current study or to a specific technology. The concepts are abstracted to enable use in related fields of research.

## **Results**

Once implemented, the lattice metaphor was utilized to view a test program in order to determine whether the visualized representation mapped the known program's source code. This involved implementing specific constructs and functions, compiling the source, disassembling the generated program, and importing the disassembly listing into the visualization program.

The visualization was then analyzed to determine whether the implemented functionality could be identified as well as whether queries could be run to generate information that could be confirmed by the availability of the source code. Success of this process would then be extendable to any program regardless of the availability of its source code.

The visualization was tested with varied programs of different sizes in order to determine the scalability of the visualization. This was essential as program functionality increases as new functionality is implemented.

### **5.3 Findings – Review of Research Objective**

The objectives of the research project were to design and develop a 3D visualization application for binary code analysis that utilizes a 3D lattice-based metaphor to represent the binary code.

The research project implemented the 3D metaphor within a 3D environment that was capable of visualizing an application for which there was the platform's corresponding disassembler. The disassembler however needed to generate the disassembly listing in a specified format required for importing of the assembly code into the visualization program. However, with a customize parse program, any disassembler's output can be converted into the appropriate format.

The 1<sup>st</sup> objective was achieved by implementing a 3D environment that enabled the X, Y, and Z dimensions to be visually perceived. The 2<sup>nd</sup> objective was achieved by mapping the extracted sequential and control instructions onto a lattice structure that utilized the X, Y, and Z dimensions. The 3D metaphor was then placed into the 3D environment within with interaction was possible.

### **5.4 Contributions – Addressing of Perceived Gaps**

#### **Theoretical**

The research project presented the feasibility of a lattice-based metaphor built from an ISA. The metaphor provided a concise and scalable means of visualizing and analyzing programs in assembly code. The results indicate that software visualization is a hybrid of both scientific and information visualization as software though abstract has an inherent structure. Consequently, metaphors that provide structure could be adapted for software visualization rather than relying on information visualization metaphors exclusively.

## **Methodology**

A potential process for designing novel or adapting existing metaphors specifically for software visualization is presented. The steps presented could be utilized in the design of other visualization metaphors.

## **Practical**

The research project resulted in an application that can be utilized to visualize and analyze assembly code, and hence indirectly any program. The scalability feature enables programs of any size to be visualized and analyzed. Furthermore, due to the structuring of the imported disassembled content, various analyses not currently included can be incorporated into the application enhancing its usefulness.

The metaphor provides a means of reducing cognitive overload by providing a means of abstracting and providing a synopsis of a program in a visual manner. Drill down capability enables obtaining greater detail in a specific section of the visualization. Cognitive Dimensions are utilized to enhance the metaphor. For example, diffuseness (notation space required to provide meaning) is utilized to provide an overall view of a program, while secondary notation is embedded within the metaphor.

3D environments provide a feasible interaction mechanism for dealing with large quantities of information. Limits exist on how much fonts can be reduced to increase the amount of textual information displayed. Screen resolutions/sizes and use of multiple monitors is limited.

With the minimal use of the keyboard in interacting with the metaphor, relying more on the mouse, 3D visualizations and analyses can be extended to touch screen computing devices whose processing capability is increasing, both the CPU and GPU. This creates new uses for touch devices.

## 5.5 Evaluation

### **Does the study make significant value add contribution to current thinking?**

Building on the surveyed literature, common concepts are extracted which indicate common practices in the visualization field and its associated research. Potential gaps which could be addresses and/or improved are identified. The literature covered various aspects of software visualization focusing on different levels of software, different stages of software development, and different uses of software metrics.

Based on the survey, a perceived gap was identified with the type of visualization metaphors used (mostly from information visualization), minimal focus on fully exploiting 3D natively (mostly focused on 3D metaphors in a 2D environment), and focus on source code level metrics (rather than binary code level).

### **Will the study change the practice if implemented in the area?**

The need to be able to understand large quantities of information in shorter time frames is becoming essential especially in software analysis. This is due to the critical role technology plays in modern processes. With the increasing capability of computing devices, and their ubiquity, research is being undertaken in the field of software visualization to find viable ways of using the concepts in the field to achieve this goal. The need for software visualization and associated analyses is bound to increase especially as software is increasingly being used for malevolent purposes.

### **Are the underlying logical answers & supporting evidence compelling?**

The results of the application of software metaphors in visualization and analysis of binary code provide an intuitive way of interacting with software. The movement from textual representation that is read to visual representation that engages during interaction increases the use of additional cognitive faculties. For example, when drilling in and out of content, a mental picture is formulated as compared to scrolling through text in a sequential manner. Hence by demonstrating the feasibility of a software metaphor, this potential is illustrated, and with refinement would provide a new means of interacting with software.

### **How well does the research convey completeness and thoroughness?**

Metaphors represent underlying aspects. In the research, the lattice metaphor represented the underlying binary code. Using an ISA, the basic building blocks are identified. These are combined together to constitute larger programs. By identifying these basic building blocks, comprising of sequential blocks, branching & looping constructs, these are exhaustively and comprehensively described, visualized, and analyzed prior to using them to visualize and analyze larger and more complex programs.

### **Is the thesis well written and flow logically?**

The research project begins by outlining the current state of information growth, both from the data and code perspectives. It outlines the need to be able to analyze the growing quantity of information in shorter time frames. Visualization provides a tool to achieve this. By incorporating the use of visualization in code analysis, the need for metaphors arises. A 3D environment enhances a metaphor's capability and consequently the visualization and analysis capability.

On this basis, a research problem area is identified. Literature review indicated the viability of software visualization as supported by the various metaphors that have been used in the field. Potential gaps are identified and form the viability of the research. A system is designed taking into account various aspects of design such as conceptual, requirements, data, content, import format, visualization, and constructs. These are used to guide the system development.

### **Why now? Is the topic of interest to other practitioners in the area?**

Software now plays a critical role in modern infrastructure by running processes that manipulate information. As computing devices become more capable, new features and functionality is being implemented. This results in an increase in the code base. The need to analyze these processes (code) in shorter time frames is becoming essential. Consequently, research is being carried out in the field of visualization, both scientific and information, in order to use the benefits and capabilities to address growing information analysis needs.

### **Who else including academic readers are interested in the topic?**

Various software domains require the ability to analyze binary code directly due to the unavailability of a program's source code. These include software engineers who are maintaining or integrating software, antivirus engineers who need to understand malware in order to update detection signatures, or security engineers would use visualization to identify vulnerabilities. Hence, beyond the theoretical aspects there are practical areas of use.

### **5.6 Recommendations**

Various areas of potential enhancements exist.

Currently, the metaphor relies on only 2 node models – for sequential and control type instructions. However, control instructions can further be categorized, for example into jumps and calls. Different models could be utilized to visually encode these sub categories, which would enhance visual perception and intuitiveness.

GPU features such as lighting could be improved to enhance the visualization environment and metaphor. This would increase the visual fidelity encouraging adoption and use. Since the features are native to the GPU, analysis performance would not be affected.

Drag and drop functionality for interacting with code segments could be implemented. This would provide the capability of manipulating the lattice structure enabling new levels of visual interaction and program manipulation.

### **5.7 Summary**

This research project began with the concept of visualizing binary code due to its role in processing of information and the ready availability of programs in executable format. The developed lattice-based metaphor provided a concise visual metaphor for interacting and analyzing the equivalent disassembled code demonstrating the feasibility in 3D.

As computing devices increase in processing capability, 3D visualization provides an alternative to textual analysis of large quantities of information, which includes binary code.

## REFERENCES

Beck, Fabian, et al (2011), Visually Exploring Multi-Dimensional Code Couplings, VisSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

Broeksema, Bertjan, et al (2011), PortAssist: Visual Analysis for Porting Large Code Bases, VizSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

Caserta, Pierre, et al (2011), 3D Hierarchical Edge Bundles to Visualize Relations in a Software City Metaphor, VisSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

Goodall, John (2009), Visualization is Better! A Comparative Evaluation, VizSec 2009, International Workshop on Visualization for Cyber Security, October 2009

Grancanin, Denis, et al (2005), Software Visualization, Innovations in Systems and Software Engineering, September 2005, Volume 1, Issue 2, Pages 221-230

Holten, Danny, et al (2005), Visual Realism for the Visualization of Software Metrics, IEEE Workshop on Visualizing Software for Understanding and Analysis, 2005

Holten, Danny, et al (2006), Visualization of Software Metrics using Computer Graphics Techniques, Conference of the Advanced School of Computing and Imaging, 2006

Holy, Lukas, et al (2012), Lowering Visual Clutter in Large Component Diagram, 2012 International Conference on Information Visualization, July 2012

Kerren, Andreas, et al (2009), Novel Visual Representation for Software Metrics using 3D and Animation, Software Engineering Workshop, 2009

Kuhn, Adrian, et al (2010), Embedding Spatial Software Visualization in the IDE: An Exploratory Study, SoftViz 2010 International Symposium on Software Visualization, 2010

Lambert, A, et al (2012), Visualizing Patterns in Node-Link Diagrams, 2012 International Conference on Information Visualization, July 2012

Maletic, Jonathan I., et al (2011), MosaiCode: Visualizing Large Scale Software (A Tool Demonstration), VizSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

Marcus, Andrian, et al (2003), 3D Representations for Software Visualizations, SoftViz 2003 ACM Symposium on Software Visualization, 2003

Medani, Dan, et al (2010), Graph Works – Pilot Graph Theory Visualization Tool, SoftViz 2010 ACM Symposium on Software Visualization, 2010

Quist, Danny, et al (2009), Visualizing Compiled Executables for Malware Analysis, VizSec 2009, International Workshop on Visualization for Cyber Security, October 2009, Pages 27-32

Reniers, Dennie, et al (2011), Visual Exploration of Program Structure, Dependencies, and Metrics with Solid SX, VizSoft 2011 IEEE International Workshop on Visualizing Software for Understanding and Analysis, September 2011

Trinius, Philipp, et al (2009), Visual Analysis of Malware Behaviour Using Treemaps and Threaded Graphs, VizSec 2009, International Workshop on Visualization for Cyber Security, October 2009, Pages 33-38

Wiss, Ulrika, et al (1998), Evaluating Three-Dimensional Information Visualization Designs: A Case Study of Three Designs, IEEE Conference on Information Visualization, July 1998

Zeckzer, Dirk (2010), Visualizing Software Entities Using a Matrix Layout, SoftViz 2010 ACM Symposium on Software Visualization, 2010

**Notes**

(Holten 2005) and (Holten 2006) papers comprise the same content.

## **APPENDIX A – DEVELOPMENT PLATFORM**

### System Build Features:

- Intel 32/64 bit processor
- Windows Operating System supporting DirectX 10.1 and above
- Visual Studio 2008 Professional Edition
- Graphics Processing Unit supporting DirectX 10.1 and above

### Support Tools

- Dumpbin – provided with Visual Studio development tool
- Link – provided with Visual Studio development tool

## APPENDIX B – USER MANUAL

### Visualization Keyboard Commands

Table B.1: Keyboard Commands

Key	Description
Views	
F1	Toggle ‘legend’ view
F2	Display/Hide ‘analysis’ view
F3	Switch ‘node’ and ‘section’ views
Q	Quit visualization (stops the current visualization)
S	Sets focus to the ‘node’ view for navigation
C	Sets focus to the ‘section’ view for navigation
(arrow keys)	Navigates the ‘node’ / ‘section’ views dependent on current mode
(enter key)	Displays a node’s related information
(left click)	Selects node
Visual	
X	Rotates the node/section view
R	Resets the node/section view to default rotation
I / O	Zooms in / out on section view
Information	
F5	Run through disassembled code to generate information
P	Pauses/Resumes run through disassembled code
L	Displays branching information with the current section +Shift – Displays branching information outside the current section +Ctrl – Clears branching information
B	Displays potential source locations for the current node
T	Simulates a run through the code +Ctrl – Clears the run through simulation
N	Displays selected flagged notation

## APPENDIX C – USABILITY TESTING

### Questionnaire

Question (Tick in one of the boxes)	Poor	Okay	Good
1. Initial impression of application			
2. Ease of navigation through the code			
3. Ease of use of the application			
4. Ease of generating beneficial information			
5. Value of analysis features			
6. Insight into the visualized program			

Question (Tick in one of the boxes)	No	Yes
7. Would you use it		
8. Have you used a similar program before		
9. Would you recommend it to someone		

10. Comments on the program: