國立台灣師範大學資訊工程研究所

碩士論文

指導教授 : 鄭永斌 博士

Collision-Based 3D Layout Algorithm for On-line Graph Visualization in DIVA (Debugging Information Visualization Assistant)

研究生 : 曾建智 撰

中華民國九十七年六月

中文摘要

隨著程式的複雜化，一般傳統的除錯方法對於程式設計師的幫助有限，因此除錯器的開發對於程式設計師而言，變的越來越不可或缺。目前的除錯器大部分仍止於文字階段，然而文字所能傳達的僅僅只有變數值的變化，難以表達資料結構間複雜的關係。因此更近一步地，有人設計出可以使用圖形化的方式來表達程式的資料之間相互關係的工具。比起文字，透過圖形化的表現方式，往往能讓程式設計師更加快速且直覺化的掌握程式的脈絡。然而，一但在圖形數量一多，圖形之間可能會互相的重疊與覆蓋(在本論文中用「碰撞」來稱呼重疊與覆蓋)，以至於整個畫面的可讀性隨之下滑，所以視覺化物件的排版就變的非常的重要。因此，我們的研究目標，就是透過適合的演算法讓空間中的物件都有各自的一塊獨立空間，將空間中的除錯資訊做較佳的輸出，提高整體的可讀性。

於本篇論文中，我們針對空間中碰撞的問題在 DIVA(**D**ebugging **I**nformation **V**isualization **A**ssistant)的三維空間環境中設計了一套排版演算法。藉由此套演算法，可以處理空間中碰撞的問題，已達到較佳的視覺效果。

Abstract

Because of the increase in complexity of software programs, debugging without help from tools is no longer adequate. Therefore, debugger has become an important tool in programmer's everyday life. Nowadays, debuggers only show the debugging information in textual form. It is considered inadequate for understanding complicated data structures. There are some debugging tools that can display debugging information in graphic presentation, which is much more informative than textual information.

However, it is useless when too much information is rendered together. In this thesis, a tool called DIVA(**D**ebugging **I**nformation **V**isualization **A**ssistant) is proposed. DIVA has object-oriented framework that enables the separation of VM programming from visualization system and particularly the composability of visualization metaphors. Based on the framework, a layout algorithm called Collision-Based layout algorithm is designed and implemented to arrange visuals in proper positions in a scene.

目錄

圖表目錄

第1章 緒論

DIVA(Debugger Information Visualization Assistant)，是一套用來協助我們程式除錯的視覺化系統。DIVA 與大部分的視覺化工具的理想一樣，希望能透過視覺化的方式將程式由文字的模式轉換成圖片或是一段動畫，讓我們可以更了解這一段程式所想要表達的是什麼意思。而本論文著重在 VM(Visualization Metaphor)方面的設計。我們稱呼一個將資料對應到圖形顯示的元件或程式叫 VM (Visualization Metaphor)，如 Excel 中的圓餅圖。

1.1. Overview

除錯，是程式開發的必然過程。有時所佔據的時間甚至比程式撰寫還要長，通常整合開發環境(IDE)都會提供一個內建的除錯器。除錯器是個複雜的系統程式，通常只具備命令列模式的文字介面。由於命令列模式的介面較不容易上手。許多 IDE 都將除錯器包裝於圖形化介面(GUI)（如圖一）底下來提高可用性。

除錯器主要的功能是經由設定中斷點來暫停程式的執行，然後借此來觀察程式狀態。大部分的 IDE 提供監視視窗(watch window)(如圖一)，讓程式人員來觀察被監視的變數。當變數具有複雜的結構(像 array 或 object)，監視視窗提供展開(unfold)，或收合(fold)指令。在一般的 GUI 設計上，通常是由 + 來代表一個變數還有子結構(如圖 一)。

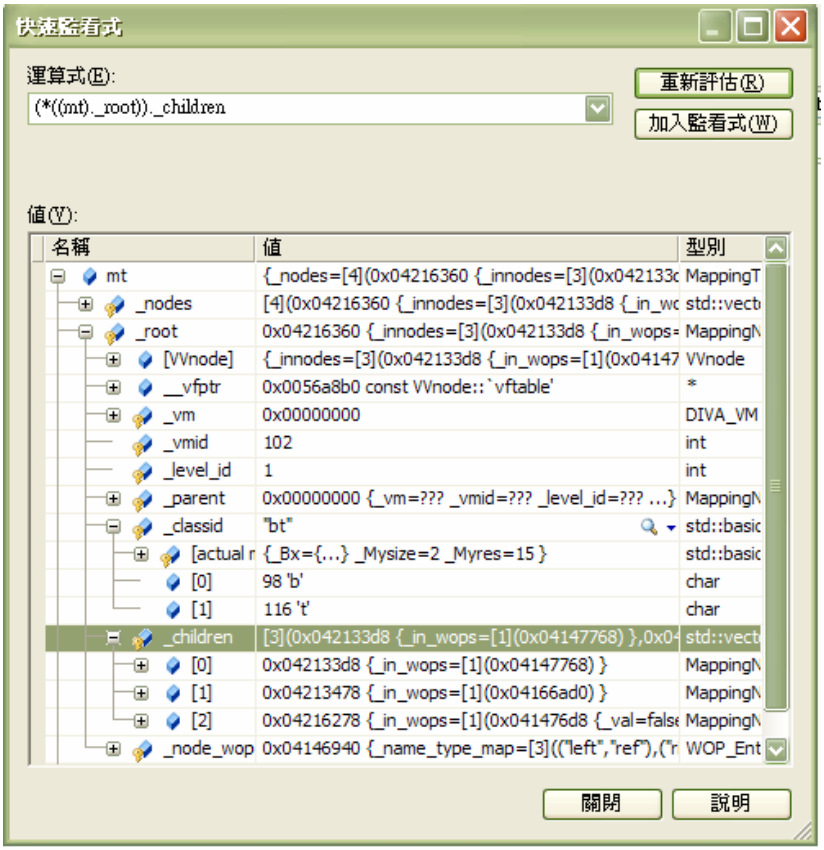

圖 一 debugger 的監視視窗

雖然除錯器是程式設計人員非常重要的工具，但是插入輸出指令（如 cout，printf）來除錯的舊習慣卻依然存在。程式人員這麼做的原因通常是為了將複雜的資料結構整理之後列印到螢幕或檔案，以幫助除錯。有經驗的程式設計師通常需要靠輸出與列印變數來搭配除錯器一起解決問題。傳統的除錯器限制程式設計師只能以文字模式視覺化單一的變數或結構。所以，當一個程式設計師想要了解物件或結構之間的關係時，他只能插入額外的程式碼，以他所偏好的視覺化方式來顯示變數與資料結構。

視覺化的程式碼可以簡單到用 printf，也可以複雜到需要轉換資料

來呼叫函式庫去繪製一個視覺化圖形。舉例來說，在一個有 graph 的應用程式當中，程式設計師可能需要加入一個副程式呼叫，來產生特定的圖檔，像是 AT&T 著名的 graphviz[1]的 DOT 格式，以便讓 graphviz 進行 graph 的視覺化。我們稱呼一個將資料對應到圖形顯示的元件或程式叫 VM (Visualization Metaphor)。當然，視覺化的對映選擇，都必須植基於使用者既有的知識與經驗。例如，一般程式人員在修習過資料結構之後，會熟悉以圓圈來代表樹的節點，以線條則代表節點之間的關連。但是程式設計人員在程式開發的過程中可以產生任意的資料結構。使用固定 1-1 的映射在實際上是不具實用性的。只有程式人員本身才能決定什麼樣的視覺化最符合他的需求。

程式以及資料結構在各領域的多變和複雜性，讓單一的視覺化要涵蓋所有程式設計師的興趣是不可能。這也說明了為何理論上視覺化工具應該可以對軟體開發有很大的助益，但是視覺化工具卻只能以非常侷限的形式存在。以 graphviz 為例，基本上要利用 graphviz 來將 graph 類的資料結構視覺化，使用者必須自行撰寫程式將某個時間點的資料結構轉換成 DOT format。也就是說，程式人員必須寫額外的程式碼，才能得到視覺化的結果。從另一個角度來說，graphviz 並不嘗試去解決困難的 VM 與資料的映射問題，而是將這部分的問題留給人來解決。

因此，我們提出一個除錯視覺化雛形工具 DIVA(Debugging

Information Visualization Assistant)的研究。其主要目的著眼於處理無窮廣泛的資料種類與 VM 的對映。DIVA 導入新穎嚴格的物件導向觀念，使得 VM 可以互相溝通，組合，且和資料完全分離，達到最低的耦合。換言之，一個複雜的 VM 可以由許多基礎 VM 組合而成，每一個都是獨立且可替代的。

而本篇論文主要是著重在DIVA在VM的設計方面和資料是完全分離的論點上，「VM programming」和「Visualization Tool」是切開來的，也就是說 VM programmer 不需要充分的了解 DIVA 的整個架構以及它的運作流程，只需要把精神放在設計 VM 上就可以了。除此之外，論文中我們也會展示一個 layout 的演算法，叫做 Collision-Based Layout VM，在我們的日常生活中一篇文章能不能吸引人們完整的看完，除了文筆精彩之外還必須有好的排版，而在我們的視覺化環境中好的排版也是很重要的，假如除錯的資訊在我們的視覺化環境中呈現出來是糾結在一塊的，通通擠在一起，就好比一篇文章沒有使用標點符號，使用者在觀看時是會非常吃力的，也就失去了視覺化的意義了。

Collision-Based layout，什麼是 Collision-Based？為什麼我們要使用 Collision-Based？在視覺化的工具中不論使用的是哪一種繪圖引擎，2D 的或是 3D 的，都會有空間上面的問題也就是空間中的物件會有重疊或覆蓋的問題，而我們的 Collision-Based layout 想要做到的就是為這些彼

此重疊或是覆蓋的物件在空間中幫它找到一個合適的位置，讓空間中的物件彼此都有間隔一小段距離，消除重疊或覆蓋的現象。

前面有提到在本篇論文我們著重在 VM 的設計與實做，所以我們還會展示另外一種 VM，叫做 Queue Composite VM。Queue 在使用上都是將它當成一個有序的串列，在這邊我們也是一樣將它當成一個有序的容器，當使用者的程式中有 array 等等的資料結構時可以搭配這個 VM 來顯示這一類的資料結構，幫助使用者在程式執行的過程中可以清楚的掌握容器內的資料是否正確。

1.2. 論文架構

本篇論文的架構如下：在第二章，首先我們將先介紹一些背景資料，例如：除錯器(debugger)、軟體視覺化(software visualization)、隱喻(metaphor)以及一些 Layout 的工具。第三章將會簡介 DIVA 整個系統的組成架構。第四章則是介紹 Collision Layout VM 的設計與實做。第五章則會介紹另外一個 VM 的例子，Queue Composite VM。最後在第六章作總結。

第2章 研究背景

2.1. 除錯器 (Debugger)

Jonathan B. Rosenberg 曾定義除錯器是種用來幫助追蹤程式、隔離且將錯誤從程式中移除的工具。事實上，除錯器可在程式中以動態的方式來幫助程式設計師找尋或修正錯誤及問題[2]。因此，軟體除錯在軟體的生命週期中扮演著很重要的一環，扮演著在軟體生命週期中負責尋找、分析及修復的腳色。Adam Kolawa 說過，在軟體開發的過程中，幾乎都有 60-70%是在作程式的除錯。但事實上，程式除錯不僅僅只佔了 60 - 70%，對於大部分的軟體而言，超過 80%的機率是經常有之的[3]。換句話說，程式在開發的過程中，所花的時間絕大部分都在於除錯。因此，程式設計師便希望能夠有好的除錯器來幫助他們除錯，讓除錯的時間變的更短更有效率。

對一般的除錯器來說，大部分的操作方式皆是採用一步一步執行的方式、或可設定中斷點，執行至中斷點或從中斷點之後繼續執行、亦可以查看某個變數的值。但是有些除錯器，則提供了某些額外的功能像是逆向執行的功能(e.g.[3][4][5])。

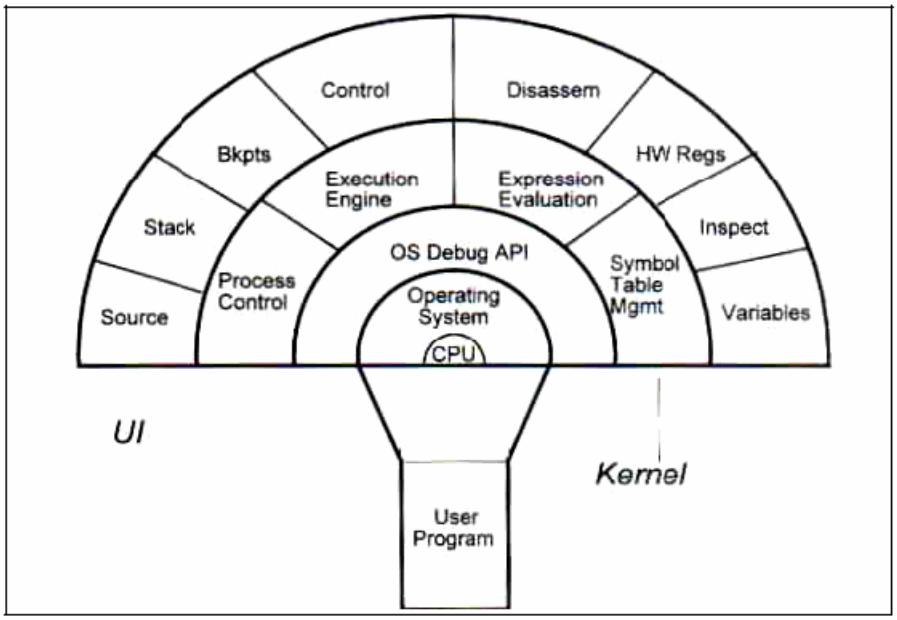

圖 二 debugger 的架構圖

Debugger 的製作相當不容易，因為和底層的作業系統相依性非常高，(圖 二)中顯示出了典型的 debugger 架構。

以下的段落將會簡單介紹一些調查過的 debugger。

2.1.1. GDB

GDB 全名為 GNU Project Debugger，目的是提供一個除錯器的實作[4]。如果程式所使用的語言是 C、C++、Pascal、Perl 等等，GNU Project 出產的編譯器就可以用它來進行除錯。它是一個文字介面的除錯器，且適用於 Microsoft Windows 或 UNIX 的作業系統上。

GDB 最主要提供了下列四種功能來幫助程式設計師找到問題：

1. 執行程式，並說明哪些行為可能會影響此程式。

2. 讓程式停止在某個特定的情況下。

3. 當程式被中斷時，則會去檢查程式發生了什麼問題。

4. 改變一些程式的內容，來測試是否對此程式對於某些錯誤以解決，又或者產生其他問題。

2.1.2. JDB

JDB 全名為 Java Debugger。它也是一般文字界面指令模式的除錯器，可以幫助程式設計師在撰寫 Java 程式時，可以針對任意的變數(variable)、或 Java 類別(class)做錯誤的找尋。JDB 也是文字介面的除錯器，所以程式設計師在每次的除錯都得輸入相同的指令，對於在除錯時相當不方便。

對於一般傳統的除錯器，像是 GDB 或者是 JDB 來說，仍都屬於文字介面，在使用方面必需得了解許多除錯器的指令之後才能進行除錯的動作。對於一個熟練 GDB 的程式設計師來說，下指令是件容易的事，但是對於一個剛接觸到這些傳統除錯器的程式設計師而言，必須要先去學習了解除錯器的指令操作方法，是很耗時耗力的。再者，程式設計師在操作 GDB 時，由於為文字指令模式在使用上仍屬不便。

2.1.3. DDD

在前面介紹了文字模式的除錯器，也提出了文字模式的不方便性，因此在這邊介紹一個圖形介面的除錯器。

DDD[6]是最先提供執行期間的視覺化資訊給除錯人員來了解程式內部行為的工具.。DDD 也是個建構在 gdb 上的圖形化前端除錯器，但具有更多的功能。DDD 以它可互動的有向圖(directed graph)視覺化而聞名，如圖 三。指標資料能被表達成由節點和有向線組成。節點由一個正方形框狀盒組成，節點中的每個變數一個接一個用文字顯示出來。陣列也可以被表示成類似表格的樣子。對某些特殊的情況，例如陣列是個整數的 2 維陣列，DDD 也能將資料傳到 gnuplot 來繪製 3d 曲線圖。

DDD 已經是個完整 open source 開發工具，雖然只能在 unix 平台上使用，用它來處理複雜資料結構的除錯可以提升不少效率。大體上，DIVA 擁有跟 DDD 相同的目標，但在設計上有很大的不同。DDD 原則上只提供固定而單一的資料與視覺化映射。

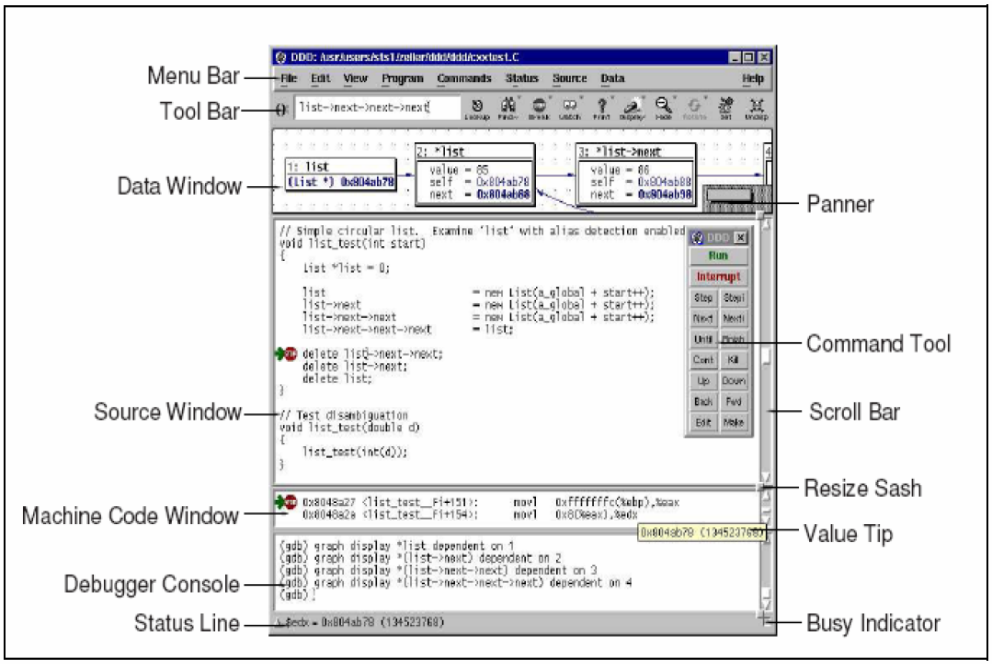

圖 三 Data Display Debugger

2.2.軟體視覺化 (software visualization)

一件複雜的事物可以經由圖形，動畫的表示，使它更容易被理解。利用一個用來增加理解力的方法，就是利用圖形來把抽象的概念更加具體的呈現，也就是利用「Visualization」來幫助理解。

而最一般典型視覺化的工作可以從(圖 四)來表示，把程式經過分析配對之後，以圖形的樣貌表現出來。從圖中我們可以看到三個腳色，第一個是程式設計師，他所做的只是將程式碼產生出來；第二個是動畫設計師，他則需要作配對的動作，將取得的程式碼與程式碼作配對，也就是找到適合的圖；而第三個觀看者即是使用者，顧名思義，即是看到視覺化成果的人。

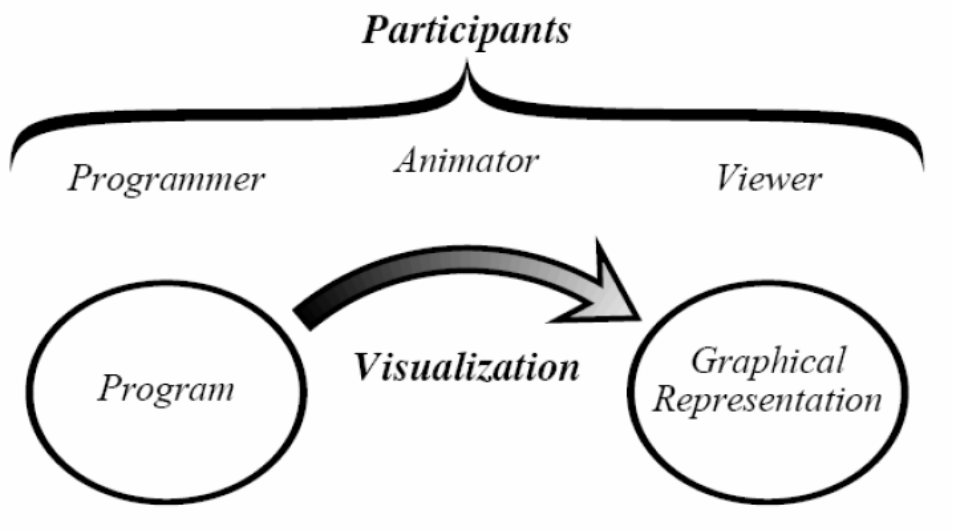

圖 四 把程式以圖形的視覺化

嘗試的利用圖形把這些相當抽象而又複雜的概念顯現出來以及幫助理解，而這個把軟體圖形化表示的過程，我們稱為「軟體視覺化

(Software visualization)」[8]。

而在[7]中提到，軟體視覺化也可以被定義成以現代化的人與電腦之間的互動科技，包含精美的排版、圖形化設計、動畫以及電影藝術，讓使用者了解且更容易去使用這些電腦軟體。軟體視覺化常常會被用來程式設計師在了解程式內容、程式除錯或者像是教學演算法或資料結構時使用。

傳統的程式設計師在接觸到一隻新的程式時，一個有效的方法都是經由除錯器一步一步的操作來了解這隻程式。可能在這過程中，程式設計師會藉由寫一些文件或者畫流程圖來幫助自己的理解與記憶。但是當程式很大的時候或複雜時，這將會是一個沉重的工作。由於人是感官的動物，對於圖形介面的直覺會比文字來的更容易快速理解，因此，如果使用軟體視覺化工具來幫助理解程式的話，想必是事半功倍。

2.2.1. 視覺化隱喻 (Visualization Metaphor)

大部分的視覺化工具使用一組圖案，圖形和顏色，透過使用者的常識和經驗來進行視覺化。舉例來說，運用資料夾和檔案櫃的圖形來表達檔案系統，就是一種視覺化隱喻。好的 VM 可以幫助使用者了解資料，有時候僅僅是簡單的改變形狀和顏色，或使用另一種圖示法，就可以讓資料更簡易的被了解。舉 pie 形圖，或者長條圖為例，一個 pie 形圖可以簡單的表達出資料在整體中占的比例，是長條圖做不到的。另一個例

子則是像圖二，一堆有位置關係 boolean 變數，使用顏色來區分將比使用文字描述更容易了解分布的情況。這部分是人類視覺系統在認知上的重要利器。

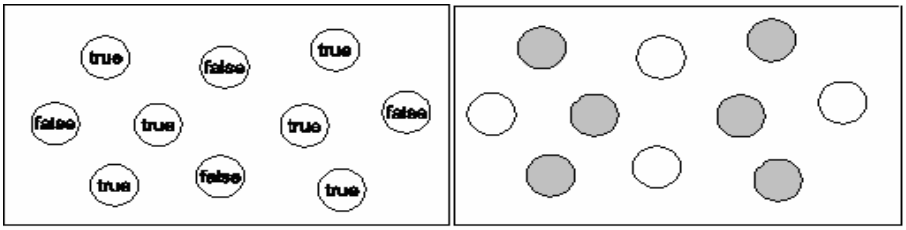

圖 五 顏色與文字視覺化效果的對比

VM 的品質和數量對視覺化工具的實用性有著舉足輕重的影響，卻又是一個很困難的問題。由於資料的廣泛性與多樣化，宣稱要能夠解決廣泛的資料種類與視覺化映射問題，將是一個極勇敢的挑戰，卻很容易淪為不實用的學術界研究。所以，在考量實用性之下，大部分的視覺化工具都會設定視覺化的範疇。以 Graphviz 為例，他只能視覺化有向圖，如 graph, tree。DDD 也選擇有向圖為主要的表達方法。然而，程式的狀態是反覆多變的。舉例來說，{(2,10),(5,7),(7,12)}可以代表任何有可能的意義，可能是一組的 2 維座標，也可能代表矩形的長寬高，除非資料的來龍去脈被告知，否則視覺化工具做錯誤的解釋可能比做安全的解釋還糟糕。所以，像 DDD 這類的工具都選擇以文字模式來表示，因為這是最安全的表示方法。然而，像圖 五所彰顯的，捨棄更有效的視覺化方式是令人婉惜的。

2.2.2. CodeCrawler

CodeCrawler 本身是一個用來做靜態分析的軟體。CodeCrawler 由 smalltalk 設計，用來作為反向工程的工具。CodeCrawler 的特色在於可以將軟體作量化分析，並把結果做成 Polymetric View。所謂 Polymetric View，根據 Lanza 所定義的[10]：

“The principle is to represent source code entities as node and their relationships as edges between the nodes, but to use figure shapes to convey semantics about the source code entities they represent.”

也就是例用矩形圖形(node)和連接線(edge)的幾何特性把資料表達出來，包含：「Node Size」、「Node Color」、「Position」、「Edge Width」、「Edge Color」(圖 六)。

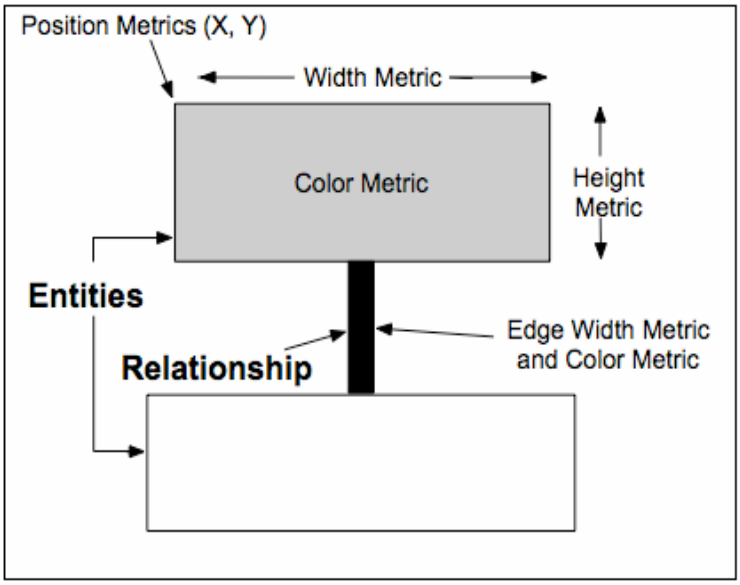

圖 六 Polymetric View

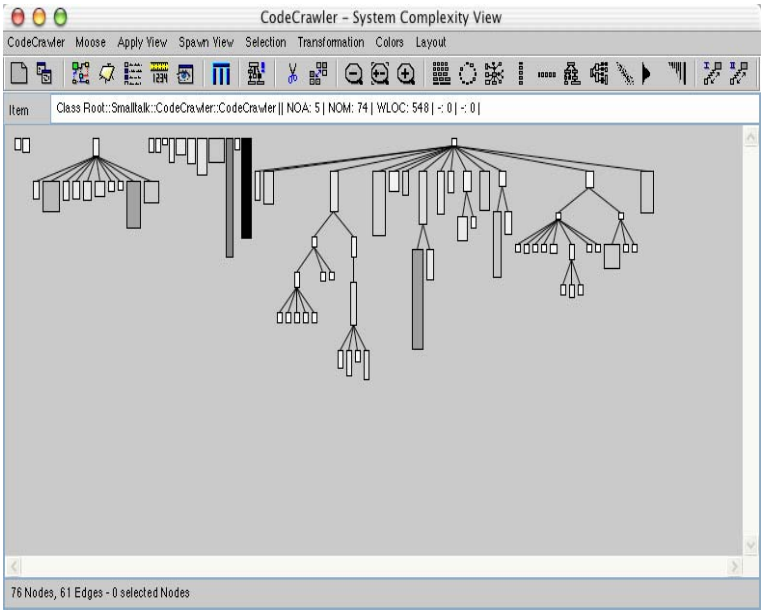

圖 七 System Complexity view

由於考慮到不同的 Polymetric View 可以突顯出不同的分析重點，所以 CodeCrawler 允許使用者動態更換不同類型的 Polymetric Views[11]。(圖 七)的 System Complexity view，用寬度表示變數數量，用高度表示函式數量，用顏色深度表示行數，可以用來觀察哪些函式可能寫了太多程式碼(顏色過深)、或是可能為資料結構(寬而淺)等(如圖八 Method Efficiency Correlation View，的 Method Efficiency Correlation View 則用 X 座標表示 method 行數，用 Y 座標表示總行數，這樣 node 的走向則可以顯示出 method 是否過於龐大，或過長未分行。

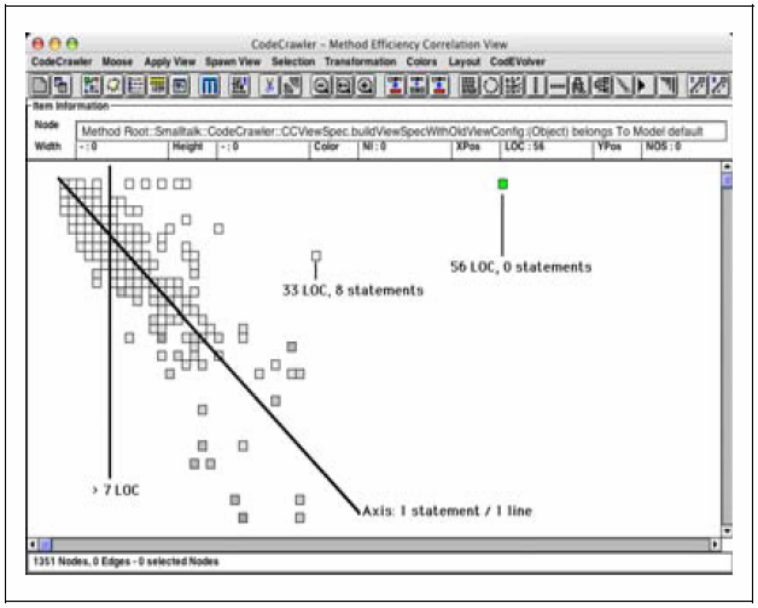

圖 八 Method Efficiency Correlation View

儘管 CodeCrawler 是相當卓越的視覺化分析工具，也提供多種視覺化呈現，但是卻只能做靜態分析。再者，CodeCrawler 的 Polymetric Views 都是預設的，而無法有使用者自行在擴充，也無法將現有的視覺呈現重新排列組合。因此，CodeCrawker 在軟體視覺化上仍然有些限制存在。

2.2.3. GV3D (GraphVisualizer3D)

“GraphVisualizer3D is a system for demonstrating the application of 3D graphics visualizing large software structure in a 3D space.”[12]

GV3D 中視覺化呈現做成許多巢狀的許多 3D 方盒，而不論是 method、data 都會被做成一個個的 3D 方盒，被包在一個代表 class 的 3D 大方盒中。當然，class 本身也可能被其他更高階層的模組的 3D 方盒包覆住[12]。

GV3D 是呈現軟體的靜態結構，並做成巢狀的有向圖形[12]。而他最大的優點是利用 3D 來呈現，可以遠比 2D 表現出更複雜更大規模的結構，而利用巢狀包覆的功能也讓階層關係一目了然。

而缺點則是他僅有這種巢狀的呈現模式，而無法再行擴充。當然，也只能呈現固定形式的靜態架構，而無法像 CodeCrawler 可以把資訊作量化的呈現以供使用者分析。

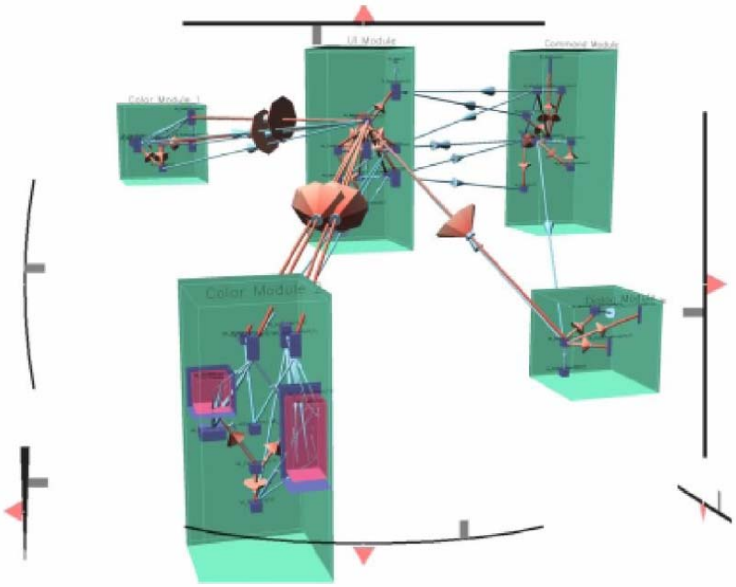

圖 九 GV3D 的 3D 巢狀結構[12]

2.2.4. BLOOM

BLOOM 是一套專門用來幫助理解軟體的工具，他所著墨的重點在於使用者自訂的 Visual Browser(類似 DIVA 的 VM)。BLOOM 同樣提出沒有完美的 Visual Browser 存在，應該要讓其可以客製化，以符合當時的需求，或是可以用不同的 Visual Browser 從不同角度來視覺化以突顯

出軟體可能的問題。不過 BLOOM 主要是用來視覺化軟體的靜態特質，如 class 的大小、數量、關連等等統計資料。而客製化的部份，也傾向於利用現有的視覺化修改數值去做微調。原則上是偏向屬於 programmer manager 所使用的 software metric，如同前面第一項所提及 CodeCrawler 這和 DIVA 想要強調的 dynamic program behavior 並不相同。

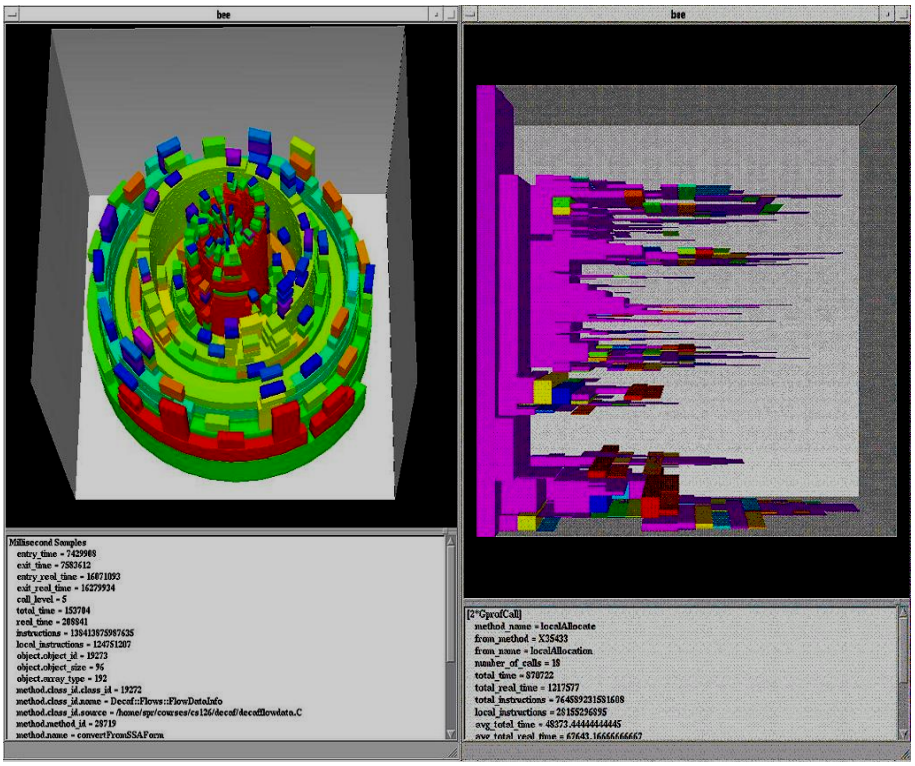

圖 十 BLOOM 的視覺化呈現

2.3. Layout Problem in Visualization

隨著軟體越來越龐大以及越來越複雜，想要對他的內容去做視覺化也變的更複雜了。所以我們需要一個視覺化的工具可以幫助我們去排列視覺化的物件，有好的排版我們可以對這段程式碼有更好了理解。

在排版方面的設計跟我們用來做視覺化工具的環境是息息相關

的，在空間的維度方面，我們可以把它分成兩種：2D 和 3D，然而不論我們選擇哪一種視覺化的工具，當物件在我們選擇的工具裡被視覺化出來後接下來的排版才是真正要面對的難題。

2.3.1. 2D VS. 3D

DIVA 建構在 Ogre 這個開放原始碼的 3D 引擎上。針對有些應用，2D 的視覺化無疑是足夠的。但是當資料變得龐大且複雜時，2D 的視覺化就明顯的不足，因為所有的圖形元素都只能擠在一個二維平面上(如圖 十一)。相對的，3D 繪圖提供額外的一個維度來表達資訊，可以明顯的改善這個問題。3D 繪圖的深度，也可以用來截取離肉眼比較遠的物件，這是人眼的特性，稱為集中焦點。不重要的物件可以暫時從視線中隱藏。在這一點上，3D 比 2D 有效率很多。

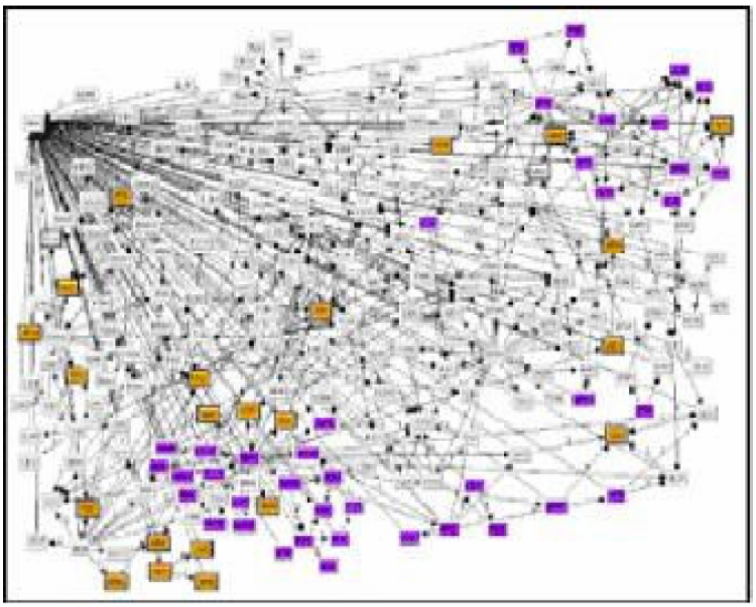

圖 十一 Call graph from a medium size system[13]

另一個 DIVA 使用 3D 的理由是為了使用動畫。3D 引擎皆可用來顯示一段即時的動畫，而不只是單純的靜態物件。如前所述，軟體是具有動態行為的。在我們的想法裡，用靜態視覺化來表達動態的行為在先天上就有不足的地方。

2.3.2. Graphviz

Grapgviz 是一個客製化的圖形編譯器，一種能夠完整呈現有向圖的視覺化工具。它可以在指令模式、網路端服務或者適用圖形化介面下執行。Grapgviz 能以圖(graph)、節點(node)以及邊(edge)三種方式組合呈現出結果[14]。

```
1:  digraph G {
2:      size ="4,4";
3:      main [shape=box];   /* this is a comment */
4:      main -> parse [weight=8];
5:      parse -> execute;
6:      main -> init [style=dotted];
7:      main -> cleanup;
8:      execute -> { make_string; printf}
9:      init -> make_string;
10:     edge [color=red];   // so is this
11:     main -> printf [style=bold,label="100 times"];
12:     make_string [label="make a\nstring"];
13:     node [shape=box,style=filled,color=".7 .3 1.0"];
14:     execute -> compare;
15: }
```

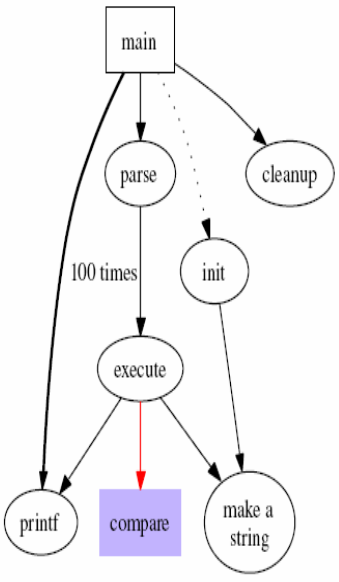

圖 十二圖 十三 DOT 程式與 Grapgviz 經由程式產生的圖[14]

Grapgviz 擁有自己特殊的語言(DOT language)，且存在著十分嚴謹的排列演算法。使用者必須了解 DOT 程式的架構以及語法，Grapgviz

會照著程式碼將根據內容以整齊的樣貌呈現，對於在複雜的圖，每個節點都不會完全重疊到圖(如)。

從上面的描述以及圖來看，Grapgviz 的在視覺化方面的能力還蠻不錯的，也被拿來廣泛的使用，但大多數的時間都被拿來用在畫圖上面，而不是當作視覺化除錯工具，而在使用 Grapgviz 來幫我們做排版的做之前，程式設計師必須要先針對想要排列的圖形利用 DOT 語言來描述圖形的結構，假如圖形非常的複雜或是節點數量非常多的時候，通常會額外寫一段程式來幫我們將圖形的結構轉換成 DOT 檔，而不是透過簡單的點選，那麼使用上就比較不方便。此外 DOT 是屬於二維的視覺化工具，一旦資料變的複雜，畫面中則會佈滿了點跟邊，對於使用者在觀看時，在了解圖的架構或內容上，必須要多花一點時間。

第3章 DIVA 簡介

DIVA 是用來視覺化程式執行中斷時的程式狀態與資料結構。目前所配合的命令列介面除錯器為 Java 的 JDB。DIVA 的第一個子系統是 debugger 的 GUI 的前端程式，稱之為 Minerva。這個程式跟其他除錯器的前端程式沒有太大的不同。程式設計師可以瀏覽程式碼，設定中斷點，執行除錯，以文字模式顯現出變數。最大的差別在，上面有個 visualize 按鈕，當使用者視覺化一個變數時，DIVA 會將此變數繪製到一個 3D 環境裡，

3.1. 架構與系統元件

DIVA 是建構在 windows 系統下，DIVA 和其前端處理器的溝通是透過網路交換訊息，因為前端處理器是由 java 實做完成，這個設計甚至使得程式設計人員可以在 Linux 下 debug 程式，然後將結果呈現在 windows 平台上。

圖 十四是 DIVA 的架構圖。DIVA 有相當多個重要元件，第一個是 Watched Object Pool Module(WOPM)，WOPM 儲存所有等待 DIVA 視覺化的變數，指標，和物件。這些資料由其除錯器前端取得，當中斷點改變時會改變其值。前端負責通知程式狀態的改變，有一些複雜的程序在

這時後啟動來確保 WOPM 和 debugger 的資料同步。

第二部份是用來處理同步的程序，也就是圖中 command agent 元件，舉例來說，當接受到 visualize p 這個指令的時後，command agent 會送出一連串的指令向前端要求 p 的資訊，當資訊取得後，command agent 則在 wopm 建立一個物件。

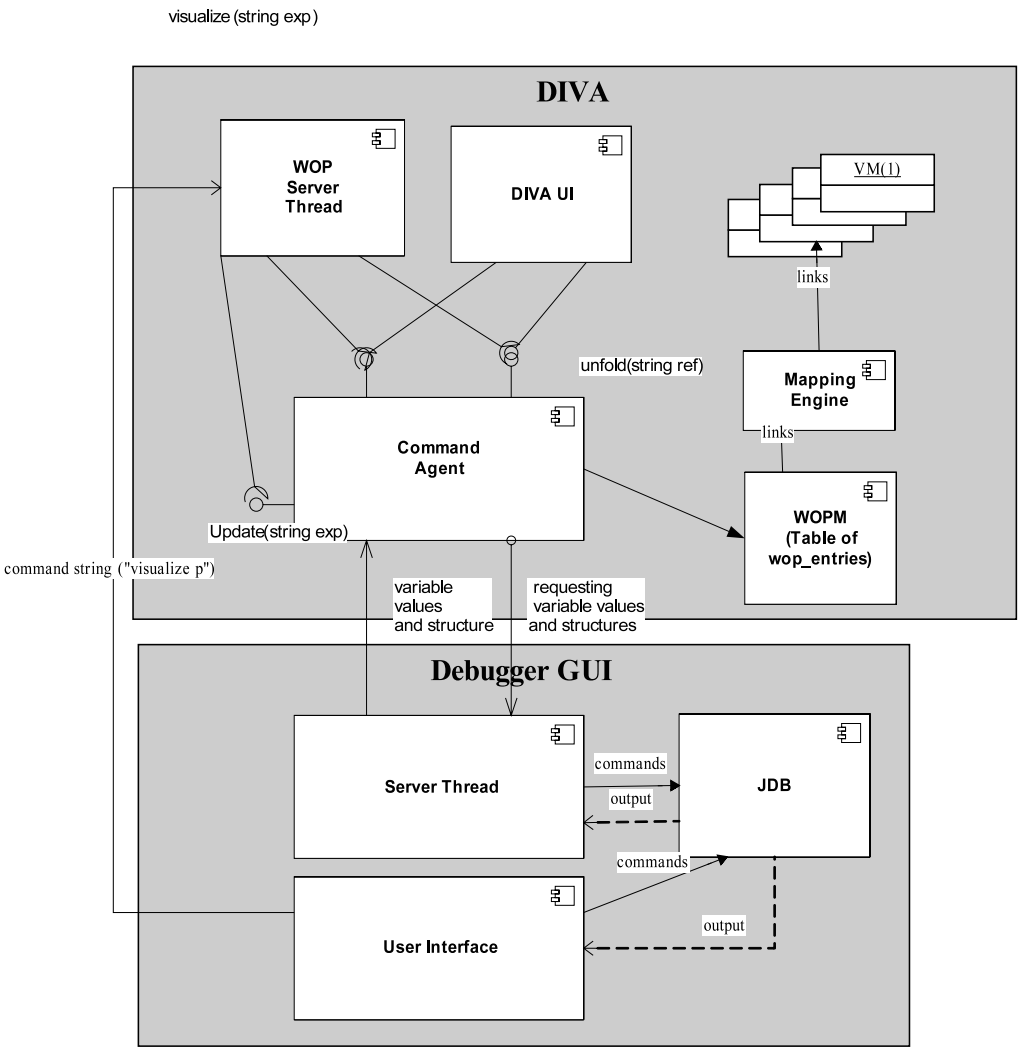

圖 十四 DIVA 架構圖

第三部份是 DIVA UI(DIVA User Interface)，這個部分 DIVA 運用 Open Source 3D 引擎 OGRE [23]來進行 3D 繪圖。DIVA UI 會將 WOPM

中收集到的資訊，經過與視覺化隱喻選擇配對組合之後，以三維空間動態方式呈現在畫面上。DIVA UI 也提供了與使用者互動的功能，使用者可對畫面中的任一物件做許多的動作，例如：移動、選取或者對某些物件作展開或合併等。

第四部份為 WOP Server Thread，在 WOP Server Thread 中會負責接收從 Minerva(Debugger GUI)端或 DIVA UI 傳來的訊息。以上這幾個部份都屬於 DIVA 端的部分。

第五部份為 mapping engine，是我們 DIVA 的核心，主要負責 wopm 裡的物件和 VM 的映射，。

第五部份為 Minerva，是 debugger 的 GUI 的前端程式，在使用上與一般 GUI 介面的 debugger 沒有多大差異。

3.2. VM 設計架構

DIVA 完整的設計原理可以由以下說明。首先，確立基礎的 VM 型別，這有點像是確認磚塊的基本種類。再來，分析如何提供個介面來組合 VM 讓它們彼此溝通。經過長時間的研究，這些 VM 被分類成以下幾種：

- Primitive type VM
- Reference type VM

- Composite type VM
- Layout type VM

每種 VM 都會在下面被介紹

3.2.1. Primitive type VM

Primitive type 的 VM 是用來視覺化 primitive type 的變數，就像 int，float，Boolean 等。嚴格上來說，一個 Primitive type 的 VM 就是就來視覺化單一的 Primitive type 的變數。圖 十五為一個簡單的 VM 例子

在這個例子中，這個 vm 繼承自一個基底 class，定義了所有需要的介面，叫做 DIVA_VM。舉例來說 VManimation()會被 3D 引擎觸發，當 3D 引擎要繪製一頁動畫時。VMvalueUpdated()則是當 Boolean 的值在 WOPM 被改變時，會觸發。在這個例子當中，當 Boolean 值被改變後，我們改變圓球的顏色，從 100 開始到數計時，來啟動 VManimation()。然後在每一個 frame 對圓球慢慢描繪特效。最後，這一群的方法會由其他的 VM 或者 DIVA 來呼叫，這些是用來和他其 vm 溝通的介面。

```
class color_ball_ptvm: DIVA_VM {
    bool * _val ;
    int countdown = 0 ;
    color_ball_ptvm(bool * _value) {
        _val = _value ;
        load a 3-D ball mesh into DIVA.
        if (_val) set color to light green
        else set color to dark red.
    }
    // methods that handle external events
    virtual void VManimation() {
        if (countdown--) make the ball glow
        else turn off the glowing effect.
    }
    virtual void VMselected() { }
    virtual void VMpicked() { }
    virtual void VMdraged() {}
    virtual void VMvalueUpdated() {
        if (_val) set color to light green
        else set color to dark red.
            countdown = 100 ;
    }
    // methods called by other VMs or environment
    virtual Vector3 getPosition();
    virtual void setPosition(Vector3);
    virtual Vector3 getSizes() ;
    virtual void setScale(Vector3);
    virtual void hide();
    virtual void show();
    virtual bool isXaxisResizable();
};
```

圖 十五 Primitive type VM 的例子

3.2.2. Reference type VM (Binary-Relation VM)

在程式設計當中，物件間最常見的關係就是 2 元關係了。其中最常見的就是由指標所建立的 2 元關係，但不僅只於如此。

舉例來說，{(a,b),(c,d)}，物件 a 和物件 b 就是 2 元關系。它可以由一個叫 pair 的類別來完成，這個類別有 2 個指標來指向 a 和 b 這 2 個物件。在 diva 當中，任何和 2 元關係有關的 VM 都必須能套用到這個例子。Binary-Relation VM 就是為了來視覺化一個指標或者一個參考，我們稱他為 Reference VM。圖 十六為 Reference VM 的例子叫 Laser_rtvm 的簡化程式碼。

Laser_rtvm 的主體是一個 3d 的球體。當它被建構以後，他的建構子從 mapping engine 接收一個叫 *p* 的 VM，*p* 指向另外一個被這個指標指到的物件的 VM。當 Laser_rtvm 被運用來視覺化一個指標時，它持續不斷的射出一條雷射光從 3d 圓球到 *p* 所指的物件，其間 *p* 會持續不斷的呼叫他所指的 VM 的 getPosition()，來取得最新的位置，接著當 VMvalueUpdated()事件發生後，首先，讓 3d 圓球體發光，接著呼叫 VManimation()，然後，指向新的 VM 的新的雷射光就此射出。

```
class laser_rtvm : DIVA_VM {
    vm *p;
    laser_ref_vm(vm *v) {
        p = v ;
        load a ball mesh;
    }
    void VMpicked() { make (*p) shining ; }
    void VManimation() {
        clean the previous laser ;
        draw a laser from
        (this->getPosition()) to (p->getPosition());
    }
    void valueUpdated() { make (*this) shining ;}
};
```

圖 十六 Reference type VM 的例子

3.2.3. Composite VM (many-to-one relation VM)

一個 Composite VM 處理許多屬於同一個實體的 VM，像是陣列或者是類別。Composite VM 的建構子從 Mapping Engine 傳入許多 VM(用 vector 儲存)。舉例來說，當一個陣列在 WOPM 裡被建構出來，Mapping Engine 首先為他的每個子元素建立 VM，然後選擇一個 Composite VM 來套用上去(如圖 十七-a)，一個 Composite VM 擁有他自己的 3D 形狀，

用淺而易見的方式套住它所有的 element 的 VM。當一個 Composite VM 被滑鼠按下以後，它必須定義如何將事件傳到它的子元素上。在圖 十七-a 中，當一個 Composite VM 被點下以後，它計算是它的哪個元素被點(row, column)，進而顯示其 index 在 VM 的上方。或者如圖 十七-b，一個直觀的樹節點的 VM，這個 VM 會指出並包住他的子元素，告訴它我們是同棵樹節點。有時後 Composite VM 需約束它的子 VM 的行為，例如子 VM 不能自行從 X，Y 軸改變大小。

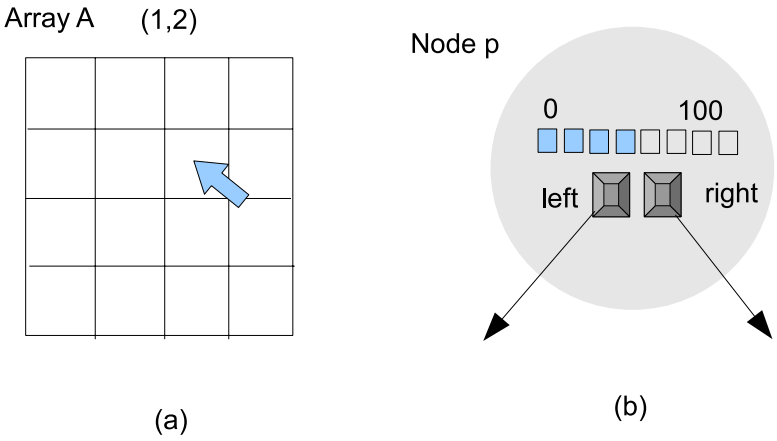

圖 十七 Composite type VM 的例子

3.2.4. Layout type VM

Layout VM 是用來在 3d 空間重新排列 VM。Layout VM 事實上是呼叫所有的 VM 來做排列的演算法。在 DIVa 中，Layout VM 是透過滑鼠右鍵來激發(通常是點擊樹的根節點或者是 graph 的初始節點)，跟前面的 VM 不同的是，Layout VM 不具任何 3D 實體。

當中，我們展示了簡單的 C++類別來說明。圖 十八這個例子是一

個 graph 的資料結構。一個直覺了當的視覺化結果就如圖 十九所示。對於每個圖的節點，一個 primitive vm 來表達其 No.，3 個 reference vm 來當作其 outedges，然後將這 4 個 vm 放入一個 composite vm。對於圖裡面的的每 edge，一個 primitive vm 來達示其 cost，2 個 reference vm 來表達其 start 和 end。

```
class node {
    int no ;
    edge * outedges[3] ;
};
class edge {
    int cost ;
    node *start ;
    node *end ;
};
```

圖 十八 Graph 的範例程式

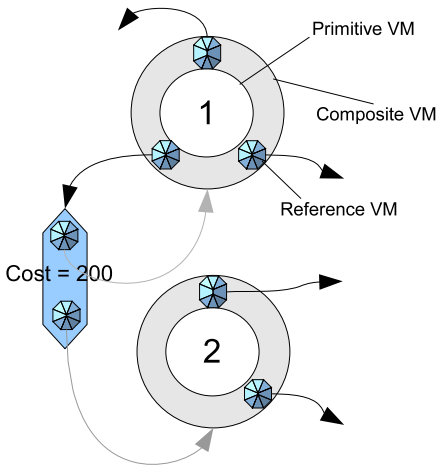

圖 十九 一個直覺的 VM 配置

第4章 Collision-Based Layout VM

排版是為了以最合適的方法放置視覺化物件和呈現各種資訊。即在有限的版面空間裡，將版面構成要素根據特定內容的需要進行組合排列，把構思與計劃以視覺形式表達出來。

在 DIVA 中，所謂的 layout VM 就是一些演算法來排列與配置 DIVA 環境中的 VMs。簡單的說，如果使用者的資料結構是個二元樹，我們自然希望有個二元樹的「off-line layout VM 」能夠配置所有二元樹的樹節點，令這些樹節點就各就各位，所謂的 off-line layout VM，所想要表達的是它並不是即時的，也就是說當使用者有需要時才會去選用，目前我們針對 off-line layout VM 的使用方式為，當使用者想要使用 off-line 時就按下滑鼠的右鍵，接著就會彈跳出一個選擇 layout VM 的視窗供使用者來選擇他要的 layout VM。另外，如果使用者的資料結構是二元樹，當我們點擊 pointer 將二元樹節點一一展開的時候，我們自然希望有所謂的「 on-line layout VM」，自動化地將我們新展開的樹節點一一按照二元樹的型態排好，而 on-line layout VM 所提供的就是即時的服務，不需要再透過按下滑鼠右鍵來選擇 layout VM，除非是要改用其它的 layout VM。

4.1. 動機

利用視覺化的方式來幫助我們除錯，是本篇論文一直強調的重點，也是 DIVA 努力想要達到的目標。然而在視覺化的環境當中不論是 2D 或者是 3D 的都會面臨到相同的問題，就是視覺化物件之間發生重疊的情況，這個問題會嚴重的影響到我們觀察與理解視覺化的結果。再者，重疊應用在別的方面可能是為了加強效果用，像是 flash 動畫，然而我們想提供的是除錯方面的資訊，所以當物件重疊時會影響到使用者操作以及觀察上的不便。如圖 二十為例，我們將一個 graph 的程式利用 DIVA 將它視覺化出來，左邊的圖幾個物件都重疊在一起了根本看不出節點之間的連結的關係，而右邊的圖沒有節點是重疊在一起的才是我們比較希望看到的。

首先重疊一辭在本論文中我們用另外一種方式來稱呼它—碰撞。為了解決碰撞方面的問題，我們實做出一個排版的 VM 來解決 DIVA 中 VM 之間排列以及 VM 之間發生碰撞的處裡。

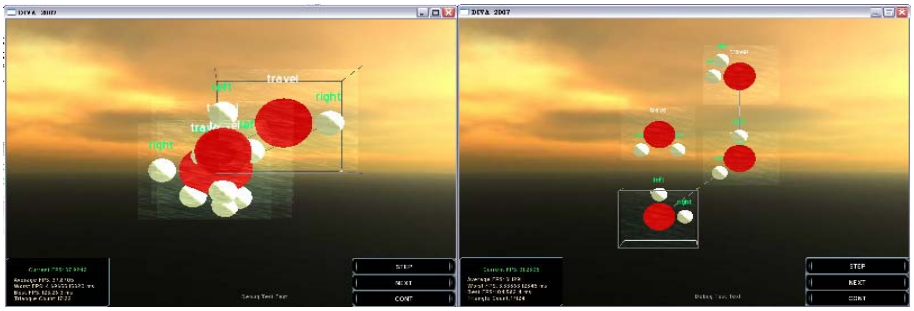

圖 二十 簡單的 graph 例子

4.2. VM Relation Structure (VMRS)

在開始介紹我們的 Collision-Based Layout VM 之前有一個重要的結構必需要先介紹—VM Relation Structure（VRMS）。VMRS 在 DIVA 中主要是提供 layout VM query 之用。舉例來說，如圖 二十一為一個簡單的樹的範例，假設現在我們要對這二元樹從左邊的排列方式，變成右邊的排列方式，那麼我們必須知道的這些節點之間的關係，像是 Root 的話我們可以輕易的透過 VMRS 的得到它有 L1 及 R1 這兩個兒子接著我們就可以容易的將 L1 及 R1 放到我們想要排列的位置。VMRS 主要是讓 VM programmer 可以透過簡單的查詢就可以得到必要的資訊以進行運算，這樣可以減輕 VM programming 的負擔，不用大費周章的去另外寫一個程式來取得這一些資訊。

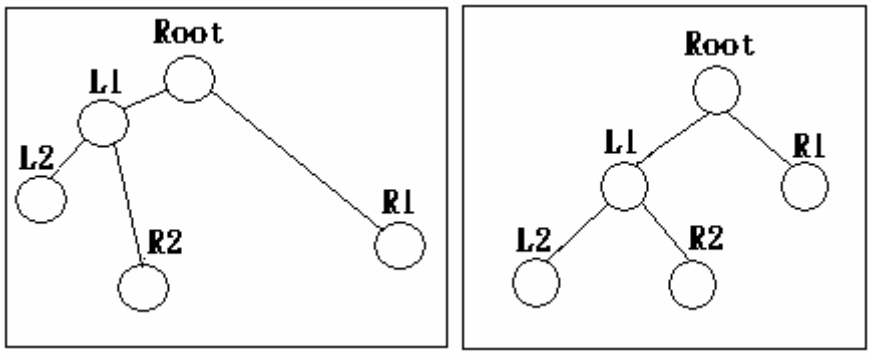

圖 二十一 樹的範例

當然 VMRS 所能做到的除了上述的功能外，還可以查詢一個 VM 它和哪些 VM 有一對一的對應關係，目前哪一個 VM 是最新被建立出來的，或

是我們想收集在 DIVA 中某個型別的 VM 如 composite type VM，以及一些簡單的 sort 像是將 DIVA 中的 VM 根據它的 x 座標或是 y 座標或是 z 座標座由大到小的排列等等，透過 VMRS 都可以輕易的取得這些資訊。

4.3. Collision-Based Layout VM

Collision-Based layout VM，顧名思義就是在調整 VM 位置的過程中考慮碰撞的問題，其最終目的是讓每一個 VM 都配置於適當的位置，而且與其他 VM 間隔一段距離。目前我們的作法著重在 on-line 的部份，也就是即時的處裡 VM 之間碰撞的問題。目前我們的作法著重在 on-line 的部份，也就是即時的處裡 VM 之間碰撞的問題。on-line 的概念在我們日常生活中還蠻常見的，像在通訊方面，我們可以透過電話的方式來做溝通，這一種即時的互動就是 on-line 的一種，msn、skype 等傳遞即時文字、影像和聲音也是屬於 on-line 的應用，而我們的 Collision-Based Layout VM 所想要傳達的也是如此，當物件之間發生了碰撞我們希望它是即時處裡的讓使用者可以在很短的時間看到結果，也就是當 DIVA 偵測到有碰撞發生了就馬上為使用者作處裡，而不是讓使用者用所謂 off-line 的方式，按下右鍵之後彈出一個 layout VM 選擇視窗讓使用者去選取要使用哪一種 Layout VM。

在處裡碰撞的過程中 VM 勢必會被移動，而在移動的過程中我們是利用動畫的方式呈現，這樣的方式可以讓使用者在 VM 調整位置的過程中可以清楚的知道為什麼 VM 要被移動以及被移到哪邊去了，假如 VM 是直接

跳到目標位置上，那麼當碰撞一多的時候使用者只會看到一堆 VM 在畫面上跳來跳去會看的一頭霧水，而即時的利用動畫的方式來顯示移動的過程也是 off-line 比較做不到的地方，因為每次發生碰撞時都要去選擇一次 off-line 的碰撞 VM。

4.3.1. 演算法

在這一個段落中我們將為大家介紹 Collision-Based Layout VM 的演算法以及它在處裡過程中的步驟，圖 二十二為 Collision-Based layout VM 的演算法，主要分成三個步驟，第一個步驟為先收集跟我們要做碰撞處裡的 VM 相同型態的 VM，像是 composite VM；第二個步驟為收集被碰撞的 VM，第三個步驟為決定被碰撞的 VM 要往哪個方向做移動以及幫這些 VM 在我們的 DIVA UI 中找到一個位置來放置它們。

```
1. procedure doLayout()
2. begin
3.       collectVMs();
4.       collidedVMs();
5.       position();
6. end
```

圖 二十二 collision 的主要演算法

以下為針對圖 二十二所做的說明：

3：VM v ：我們想要處裡碰撞的 VM

函式 collectVMs 主要是透過 VM Relation Structure 去取得在 DIVA 中和 v 相同型別的 VM。

4：函式 collidedVMs 從 collectVMs 收集而來的 VM 中去找尋被碰撞的 VM。

5：函式 position 用來決定被碰撞的 VM 該往哪個方向做移動以及尋找一個位置來放置來放置這些 VM。

```
1. procedure collectVMs
2.     container c = get all VMs from VMRS
3.     VM vm is the input
4.     for VM v in c
5.         if(v.type == vm.type)
6.             put v in container ret
7.     end
8. end
```

圖 二十三 函式 collectVMs

collectVMs 要作的事就是去收集在 DIVA UI 中的 VM，以下是針對圖 二十三所作的說明：

2：容器 c 是用來裝所有在從 VMRS 中回傳的 VM，裡面包含了各種的 VM，primitive type VM、composite type VM 等等的。

3：vm 就是我們要處裡碰撞的 VM，這邊當作輸入用來判斷容器 c

中哪些 VM 的型別跟它相同

4-7：假設 v 的型別和 vm 一樣的話，就將 v 放到要回傳的容器 ret 中。

```
1. procedure collidedVMs
2.     container c = collectVMs
3.     VM vm is input
4.     for VM v in c
5.         if(v collided with vm)
6.             put v in container ret
7.     end
8. end
```

圖 二十四 函式 collidedVMs

collidedVMs 主要功能是去收集那些被碰撞的 VM，以下是針對圖二十四所作的說明：

2：容器 c 的內容物是先前提到的函式 collectVMs 所收集到的 VM。

3：vm 就是我們要處裡碰撞的 VM，這邊當作輸入用來判斷容器 c 中哪些 VM 跟它發生了碰撞。

4-7：這邊是利用 VM 之間的位置以及 VM 的大小(VM 的長、寬、高)來判斷 v 與 vm 是否有發生碰撞，有的話就將 v 放到要回傳的容器 ret 中。

```
1. procedure position
2.     d = densityFunction(vm);
3.     b = ballFieldCount(vm);
4.     dir = direction(d,b);
5.     set vm on position p
6. end
```

圖 二十五 函式 position

Position 則是用來決定我們的 VM 要放到哪一個位置上，以下是針對圖 二十五所作的說明：

2：densityFunction 主要是去收集並統計在每一個區塊中有幾個 VM，如圖 二十六所示，首先從圖中我們可以看到，中間的方塊(R2)是我們要處裡碰撞的 VM，而其他的方塊則是已經存在於 DIVA UI 中的 VM，剛剛有提到的 densityFunction 是用來收集並統計各個區塊中的 VM 個數，而這些區塊是怎麼被區分出來的呢? 在這邊我們利用 R2 當成我們新的座標原點，定出新的座標軸，而新的座標軸會將我們的空間切成八個小空間，而這八個小空間就是我們所謂的區塊，我們將空間切成八個區塊。所以 densityFunction 所作的就是去統計這八個區塊中各有多少個 VM。

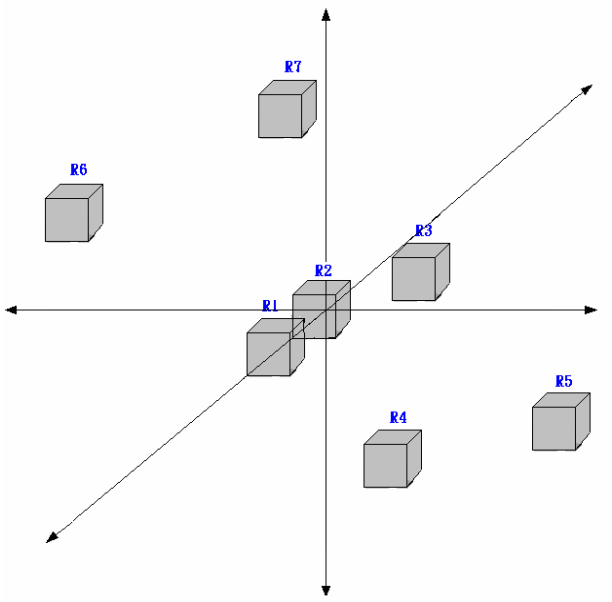

圖 二十六 DIVA UI 中 VM 的分布

3：ballFieldCount 所作的事跟 densityFunction 很類似，也是統計八個區塊中各個方向的 VM 個有多少個，但 ballFieldCount 跟 densityFunction 的不同點在於，ballFieldCount 他只針對在一段特定的距離內八個區塊中客有多少個 VM，如圖 二十七所示，在圖中所看到的圓球所涵蓋的範圍，而圓球的半徑 r 就是我們所提到的一段特定距離，在 DIVA UI 中的 VM 若是和 v1 的距離小於半徑 r 的話，那麼這個 VM 就會坐落在圓球的範圍內，而 ballFieldCount 就是要去統計這一些 VM。而半徑 r 的值是我們每次一個 VM 做移動時的位移量，所以統計到的數據代表我們往那個區塊移動的話會撞到的 VM 數量。

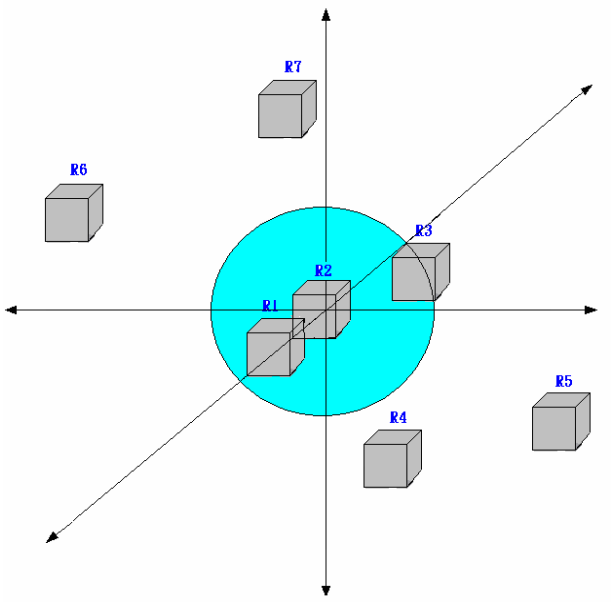

圖 二十七 ballFieldCount 的示意圖

4：direction 這個函式主要是利用先前提到的 densityFunction 以及 ballFieldCount 所回傳的結果，利用這兩個的結果去決定要往哪一個區塊做移動。而 direction 的判定我們是利用參數 d(densityFunction 的回傳值)和參數 b(ballFieldCount 的回傳值)這兩個參數來做判定，利用這兩個參數我們可以在我們所定義找到的八個區塊中 b 跟 d 的值是較小的，這邊我們強調是“較小”的而不是最小，因為參數 b 的數據是代表往那個區塊移動的話會撞到的 VM 數量，所以這個值越小的區塊才是我們考慮的，而參數 d 的數據則是每個區塊的 VM 數量，所以我們選 b 跟 d 的值是較小主要的原因是因為要避免極端的例子，像是某

個區塊它的 b 值很小但是 d 值卻很大，往這個方向移動的話很有可能會造成很多的碰撞，因為我們希望盡可能的做到最少的移動。

5：幫 vm 在 dir 方向的的區塊找一個位置 p，並將 vm 放置到那個位置上。

最後，我們的演算法在處裡碰撞的問題是採用遞迴的方式，如圖 二十八是碰撞的示意圖，vm(中間藍色的方塊)是我們要處理碰撞的 VM 另外 cvm1、cvm2、cvm3(紅色方塊的部份) 則是與被 vm 所碰撞的 VM，在我們的演算法中要被移動的 VM 是 cvm1、cvm2 以及 cvm3 這三個，所以首先我們先移動 cvm1，移動完後當然繼續要對 cvm1 做碰撞的處理，假如沒碰撞那麼就繼續移動 cvm2 和 cvm3，但如果又發生碰撞的話那麼 cvm1 在我們的演算法中就會變成圖中的 vm，必須去移動被他碰撞到的 VM，這樣一直迭代下去直到沒有碰撞發生為止，所以我們的演算法採用遞迴的方式來處理。

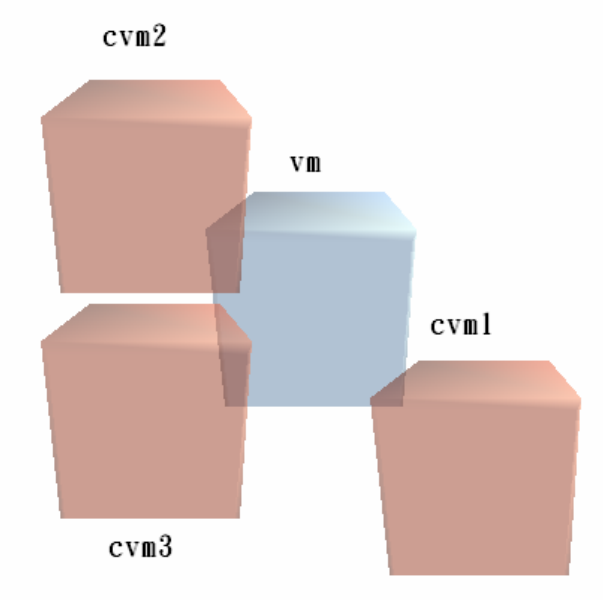

圖 二十八 VM 碰撞的示意圖

而使用遞迴最擔心的是會不會收斂的問題，所以我們處裡碰撞的過程中為了避免會有無法收斂的狀況我們採用了一種方式來解決，把已經處理完碰撞的 VM 標記起來並當成下一次遞迴的 input，這邊的意思是說我們的遞迴式子會有兩個參數，一個是我們要處裡的 VM，另外一個則是一條 vector 裡面紀錄我們目前這一次遞迴運算式中被標記的 VM 也就是處理完碰撞的 VM，會這麼做的原因是要避免掉 VM 回撞的問題，所謂的回撞如圖 二十九所示，在 Time 1 的時候 vm1 和 vm2 發生碰撞，到 Time 2 時 vm2 被移動了且和 vm3 發生碰撞，一直到 Time 4 的時候 vm4 與 vm1 發生了碰撞此時就是我們所謂的回撞，發生回撞時我們應該繼續移動 vm1 嗎？當然不是的，因為這樣會造成兩個問題：

- VM 之間可能會像圖 二十九所示有一個碰撞 cycle，造成遞迴

無法收斂。

- vm1 是目前使用者正在觀察或操作的 VM，在碰撞的過程中假如被移走會讓使用者覺得莫名奇妙，怎麼突然被移走了。

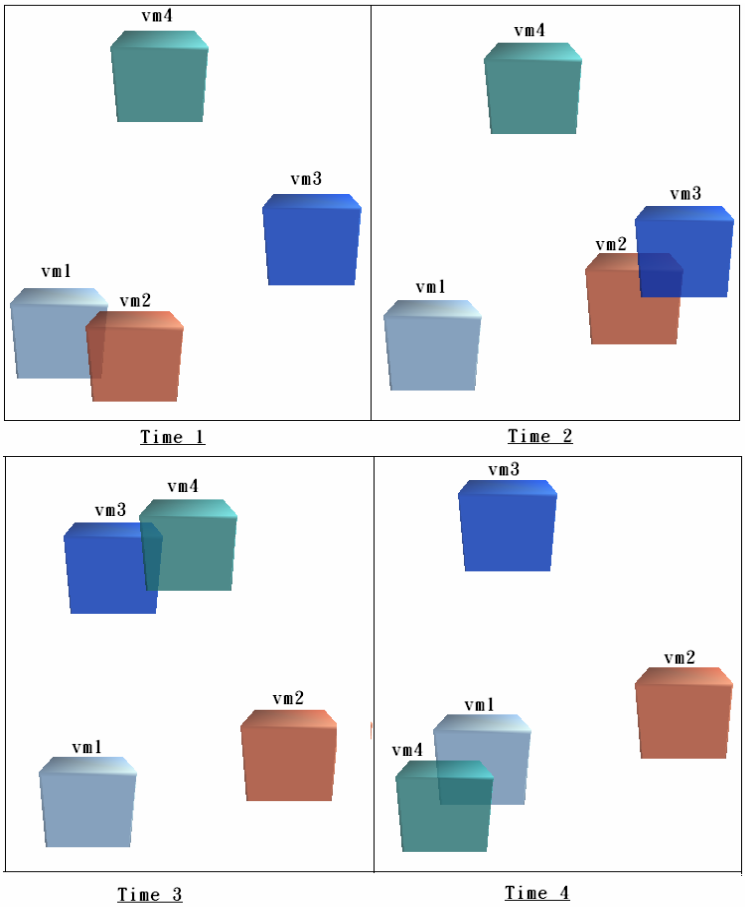

圖 二十九 回撞的示意圖

所以我們運用標記已經處理完的 VM 將這些 VM 視為不可以在被移動的 VM 來解決上述的問題，讓我們的演算法可以在有限的時間內處理完碰撞的問題。

第5章 Queue Composite VM

在前面一個章節介紹了 DIVA VM 其中之一的 Layout VM，而本章節將介紹 DIVA VM 中的 Composite VM，名為 Queue Composite VM。在這個章節中我們將以 programmer 的身份來實做此一 VM。

5.1. 動機

當我們在寫程式時常常都會使用到 STL(Standard Library)裡面的一些”容器”，像是 queue、vector 等等的，來幫助我們存放資料，使用這一類的容器在除錯上我們可以很容易的透過 printf 或是使用整合開發環境(IDE)來觀察容器裡面的存放的資料正不正確。

不過除錯通常是蠻無趣的，而這種除錯方式也顯的枯燥乏味了點，所以我們想改用有趣一點除錯方式，將容器裡面的資料利用視覺化的方式輸出到螢幕上，除了一般我們可以了解資料的正確性與否外，還有另外一點，當程式在執行的過程中我們可以透過容器內的變化來觀察程式動態的行為，也就是利用動態的方式來描述動態的行為。

此外，在第一章緒論有提到 DIVA 在 VM 的設計方面和資料是完全分離的論點上，「VM programming」和「Visualization Tool」是切開來的，也就是說 VM programmer 不需要充分的了解 DIVA 的整個架構以

及它的運作流程，就可以設計與實作 VM，在本章節將以此概念來完成這個 VM，再搭配上 mapping gui 來說明這個概念。

5.2. Queue Composite VM

Queue composite VM 所想要呈現的是”容器”的概念，在 DIVA 中用一個容器把這些的 VM 給包起來讓使用者觀察目前的容器中有幾個 VM，舉例來說，當我們在做 BFS (breadth-first search)的時候，利用這個 VM 來顯示目前 queue 中有幾個節點，所以隨著節點被依序拜訪時 queue 的內容歲隨時的在做增加與刪除，而 Queue Composite VM 也會跟著變化，讓使用者更了解 BFS 的原理。

首先我們要先幫容器製作一個外殼，構想如圖 三十用一個像是拱橋形狀的模型當作我們的外殼來裝那些存放在 queue 中的 VM(圖中的黃色圓球)，順著外殼的形狀排列這些 VM，然而，VM 的個數有可能非常的多，所以要將全部的 VM 都排列出來的話會有困難，所以目前我們將可以顯示的 VM 最多設為 6 個，也就是說使用可最多可以看到容器內有 6 個，分別以 queue 中 head 開始計算的前面 3 個，以及包含 tail 的後面 3 個。而對於 Queue Composite VM 而言，裡面所裝的是什麼 VM，有什麼樣型態的資料，完全不重要。對於 Queue Composite VM 來說，唯一在意的是 VM 的個數，以及要顯示的是哪幾個 VM，VM 個數的話 DIVA 會提供而要顯示幾個的部份，即 queue 的 head 和 tail 這兩個

參數可以由使用者自行輸入(pop 出一個讓使用者輸入的視窗)或者是用我們設定的預設值。

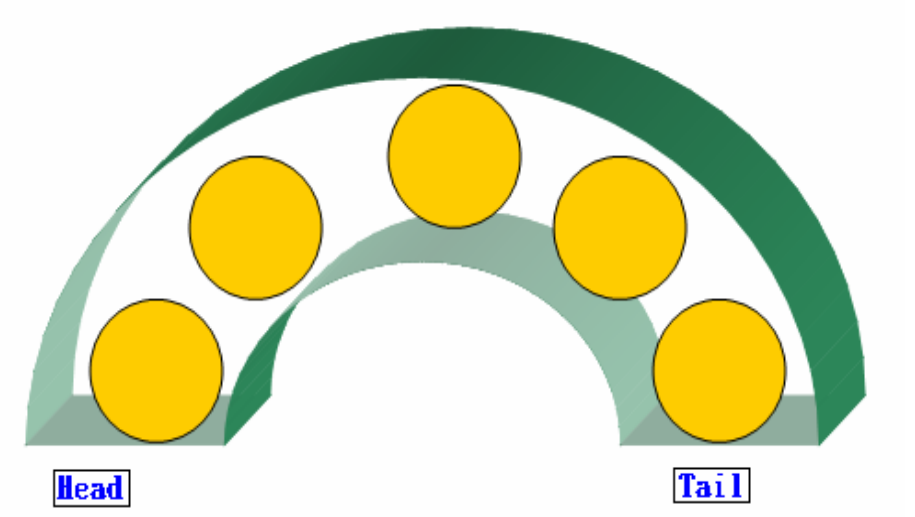

圖 三十 queue 示意圖

而針對本章 queue composite VM mapping gui 的概念圖如圖 三十一所示，圖的左邊是一個 C++的 class 而右邊則是 DIVA 的 mapping gui 示意圖，首先 mapping engine 會將 array 中的每個數值都 new 出一個 VM 然後放到圖中 vm collector B 中，接著 mapping engine 會替圖中的 composite VM A 找到一個適合的 composite VM，選擇適合的 VM 這部份可以由 mapping 提供合適的 VM 來讓使用者選擇，而這一部份 VM programmer 是不用知道，就如同前面所說的 VM programmer 不需要充分的了解 DIVA 的整個架構以及它的運作流程，就可以設計與實作 VM。

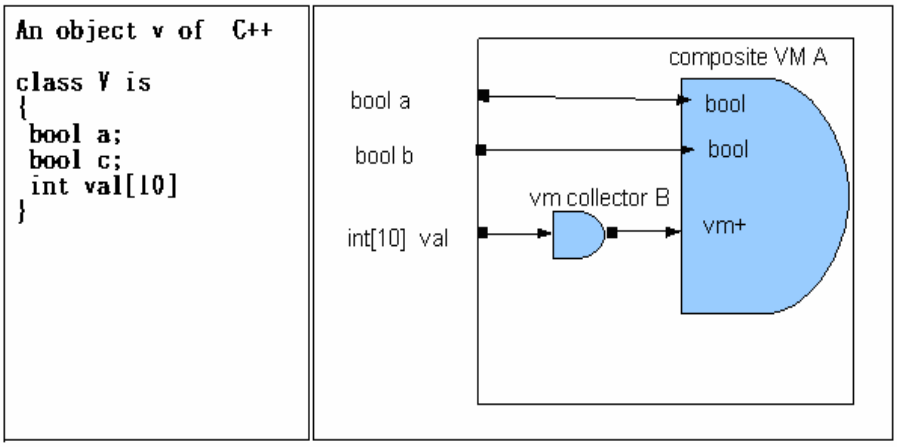

圖 三十一 mapping 的概念圖

結論

過去數十年來大部分的視覺化工具不論是用在教育用途上面的或是已經被拿來廣泛使用的，大都只能針對特定的演算法、資料結構，而應用在任意程式上面的是少之又少，原因在於 VM 的設計上屬於一對一的對應關係。

而本實驗室所開發的 DIVA 雖然目前還是個發展中的系統，但針對 VM 在設計上有 2 個獨特的特性：可組合性以及將 VM 的設計抽離出來，可組合性方面，我們可以在產生一個新的 VM 時候，可以利用已經存在的 VM 來組合成一個新的 VM，而將 VM 的設計抽離出來方面，想表達的是，VM programmer 不需要很充分的了解 DIVA 的整個架構，只需要專心在 VM 的設計就可以了，VM 之間複雜的對應關係在 Mapping Engine 會幫我們把處裡這部份的工作，所以 VM 的設計就可以交由第三方輕易的去製作。

所以在這篇論文中所介紹的兩個 VM 的設計與實做都是根據上述的兩個特性完成，所以假以時日，VM 的數量達到了一定的量後，透過各種組合與替換，應該可以應付大部分的視覺化。

Reference

[1]. Emden R. Gansner ,Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong, A technique for drawing directed graphs. IEEE transactions on Software Engineering, vol 19,NO.3, pp. 214-230, 1993

[2]. J. B. Rosenberg, How Debuggers Work: Algorithms, Data Structure, and Architecture, John Wiley & Sons. , September 27, 1996

[3]. Dr. Adam Kolawa, Parasoft, The Evolution of Software Debugging, http://www.parasoft.com/jsp/products/artical.jsp?articalId=490

[4]. R. M. Stallman, R. Pesch and S. Shebs, Debugging with GDB: The GNU Source-Level Debugger, Copyright (C) 1988-2006, Free Software Foundation, Inc. http://www.gnu.org/software/gdb/documentation/

[5]. Omniscient Debugging ”Because the Debugger Knows Everything”, February 18 2007. http://www.lambdacs.com/debugger/debugger.html

[6]. A. Zeller, “Debugging with DDD”, Copyright (C) 2000 Free Software Foundation, Inc. http://www.gnu.org/manual/ddd/pdf/ddd.pdf

[7]. B. A. Price, R. M. Baecker and I. S. Small, A Principled Taxonomy of Software Visualization, Journal of Visual Languages and Computing 4(3), September 1993, p211-266

[8]. J. T. Stasko, J. B. Domingue, M. H. Brown and B. A. Price, “Software Visualization”, ISBN-13:978-0-262-19395-5, 1998

[9]. Knight, Munro, “Visualising Software - A Key Research Area”, Proceedings of the IEEE International Conference on Software Maintenance ,page: 437, 1999

[10]. M. Lanza, “CodeCrawler – An Extensible and Language Independent 2D and 3D Software Visualization Tool”, In ”Tools for Software Maintenance and Reengineering”, page:74 - 94, RCOST / Software Technology Series. Franco Angeli, 2005

[11]. M. Lanza and S. Ducasse “Polymetric Views—A Lightweight Visual Approach to Reverse Engineering”, IEEE Transactions on Software Engineering archive, volume 29, Issue 9, pages:78 -795, 2003

[12]. C. Ware, G. Franck, M. Parkhi and T. Dudley, “Layout for Visualizing Large Software Structures in 3D”, Proceedings of 2nd internal conference on Visual Information System, 2000

[13]. A.Nganou, R. Al-Amad, S. Shi and X. Xian, Software Visualization 2D versus 3D, Department of Computer Science Concordia University, 2001

[14]. Emden Gansner and Eleftherios Koutsofios and Stephen North, Drawing graphs with dot, January 26, 2006