Chapter 2 Background

2.1 Visualization

According to [5], Visualization has been distributed into two largely parallel stands, scientific visualization concerned with the problem of visualizing data obtained from instruments such as sensors and scanners, or from engineering simulation, and the other with the problem of representing and exploring data in abstract space, sometimes called information visualization.

Application areas of scientific visualization are well known. They include meteorology, fluid dynamics, medical imaging to name but a few. The general characteristic of data sets used in these areas is that the data is located within the physical space in which the measurement or simulation takes place, and each point within the data set can be assigned a position in $\mathbb{R}^n$. Applications for which the underlying data set cannot naturally be embedded in $\mathbb{R}^n$ fall into the general category of information visualization. Examples include:

- business data: sale, revenue, production, etc., indexed by discrete values such as day, location, region, or product;
- graphs and trees: computer networks, file system structure, hyperlinks, organizational models, project charts;
bioinformatics: expressed sequence tags (genome sequences), structural models of molecules, molecular pathways.

A necessary task of any information visualization system is to map the data from the abstract space in which it is defined, either into entities with explicit geometric structure, or into some subset of $\mathbb{R}^n$ which can then be rendered using techniques from scientific visualization.

2.1.1 Software Visualization

Software visualization (SV) is the use of the craft of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software. Software visualization has two broad categories: algorithm visualization and program visualization.

Algorithm visualization is the visualization of high-level abstractions that describe software. Flowcharts, which use animation to communicate how the algorithm works, are a simple example of the static algorithm visualization. Algorithm animation is dynamic algorithm visualization such as the early film Sorting Out Sorting (SOS) of Baecker [1], Brown’s BALSA [3] and Zeus [4], and Stasko’s TANGO system [22].
Program visualization is the visualization of actual program code or data structures in either static or dynamic form. Static code visualization might include some kind of program map such as the SEE Program Visualizer [2], while an example of static data visualization might appear as a “boxes and arrows” diagram of a linked list data structure showing the contents. An example of animated data visualization might show this same diagram with the arrows and contents changing dynamically as the program were running, while simple animated code visualization could highlight lines of code as they are being executed.

Visual programming (VP), VirTool [33] for example, is the use of “visual” techniques to specify a program. Visual programming seeking to make programs easier to specify by using a visual notation is the main difference with software visualization. Programming by demonstration (PbD) is the specification of programs using user-demonstrated examples. The idea is that users may not need advanced programming skills to construct a program, but may be able to demonstrate an example and have the system infer a program.

Figure 2.1 shows the relationship between the various types of software visualization and the related fields.
2.1.2 Modularity in Visualization

Visualization techniques typically are domain specific. A common problem of a visualization method is that it cannot be applied to other program or domain. That is because the structure of the data space and the interactions with the data influence the visualization algorithms.

Reuse is limited to modules within a specific problem area. And even where reuse is possible, finding the right level of component abstraction within a particular domain remains a difficult task. A more integrated approach to visualization, one that accommodates problems and solutions related to both physically-based and abstract data is desirable and indeed necessary. There are three issues underpin this argument.
1. Integration would encourage better reuse of algorithms and tools, and the development of new techniques. For example, Herman et al. [8] have described a method for visualizing the structural complexity of trees and graphs with 2D representation. The general idea of this method could be extended to a 3D representation as follows:

- Visualize the tree structure using the cone-tree representation [18];
- Calculate a metric for each node of the tree, and interpret this metric as a field strength;
- Visualize the field around the tree using an iso-surface extraction algorithm, and superimpose this over the tree representation.

2. It is desirable if the tools for visualizing this data could be built on top of existing and well tested infrastructure, rather than risking ‘re-invention of the wheel’. For example: the infrastructure for storing the data needn’t to build up again.

3. The understanding of how visualization works, in terms of the capabilities and limitations of human information processing, is the other issue. Within cognitive psychology, research has concentrated on building micro-theories of specific phenomena such as visual proceeding and short-term memory. Marr’s model [12] of visual processing, for example, are important in
understanding low-level aspects of visual representation, but to address issues such as gestalt phenomena, antonym, and affect, a more comprehensive approach is required. In psychology, a gestalt means something that has particular qualities when you consider it as the whole which are not obvious when you consider only the separate parts of it [24]. Works on representations for abstract domains are more concerned with higher levels of task support; how does the representation encode information that users will need? What information is missing in the representation? Using a more general model of visualization data may reveal or provide links between the existing bodies of work [5].

2.1.3 A Graph Visualization Pipeline

Since in our thesis, we focus on visualizing graph-like data structure, we describe the related work of graph visualization. Research reports on graph visualization have usually been at two levels. Either they describe new generic algorithms, such as graph layout, or they describe a tool or representation developed for a specific problem. There are two problems with the approach to visualize a graph structure on modular architecture:
1. Graph visualization is not just a graph drawing. What is needed is the ability to utilize this data in visualization operations.

2. Tools and techniques are typically implemented with the principle: integration would encourage better reuse.

For example, to visualize a large graph with $10^4$ to $10^6$ nodes, one can draw all nodes in a large layout, but sometimes drawing on a small layout with some meaningful pattern is a better choice. The user may need to alternate between identifying small-scale structures, and ‘drilling’ into those structures to investigate internal detail. To support this it may be helpful to have multi-resolution models of the dataset available.

When the information which will be visualized comes, there is a pipeline which defines a way how to visualize the data, especially in a modular visualization system. For example, the Visualization Toolkit (VTK) [20] and AVS [23] are truly modular architecture. On these truly modular architecture systems, it is need a little effort to implement visualization tool and techniques.

AVS is a framework that can be used to develop scientific visualization applications based on a model that integrates interactive visualization into the research and engineer process. It is targeted at scientists and engineers, rather than software
developers. The design goals of this system include: easy to use, low cost, completeness, extensibility, and portability.

VTK does not provide a visual programming environment like that of AVS, but rather provides an extensive class library that can be used to develop compiled applications, or can be used through a scripting language. The architecture of VTK is that of a demand-driven pipeline made up of three kinds of process object: source, filters, and mappers. A source is a component that puts data into the pipeline. Filters take one or more datasets as input, and produce one or more as output. Mappers take a dataset as input and convert it into graphics primitives that can be rendered.

### 2.2 An Overview of Debugger

Debuggers are critical tools for the development of software. They are studied very little as compared, for example, to compiler. But more hours are typically spent on debugging programs than compiling them. Debuggers are hard to build robustly because they depend heavily on operating system and they tend to stress the underlying operating system’s capabilities. Figure 2.2 shows typical debugger architecture.
2.2.1 Classification

- **Source-level (symbolic) versus machine-level**

  Early debuggers were not able to make the reverse mapping back from machine instructions to the original source code, but as applications grew in complexity the need of reverse mapping became more and more important. A developer may have a lot of benefits from reverse mapping because machine instructions are too low level to understand. The goal is to have the debugger with the reverse mapping technique that source code is being directly executed as if the underlying machine is not an Intel, SPARC, PowerPC, or whatever CPU, but is a C/C++, Pascal, COBOL, Basic, or Java execution engine and speeds up debugging process.
But there are times you still need to dip down into the low-level, machine-specific
details of how the program is actually running on the hardware. Therefore, every
source-level debugger also needs to provide low-level information, such as register
values, memory dump, assembly code, and other machine-specific information.

- **Stand-alone versus integrated development environments**

  A stand-alone debugger is a program dedicated solely to debugging and is
  separated from compiling and edition, for example Turbo Debugger [19] which is a
  well-known DOS stand-alone debugger and GDB [27] is the debugger which can run
  on most popular UNIX and Windows variants.

  However, there is a trend in recent years to integrate debugger into IDE
  (integrated development environments). The main reason to move from stand-alone
debuggers to integrated development environments that include debuggers is for
increasing programmer’s productivity. Today, many developers have begun to consider
GUI debugger integrated into an IDE. GUI debuggers offer a more natural user
environment. Furthermore, GUI debuggers integrated into a single environment offer a
much higher level of functionality than possible with stand-alone debuggers.
• **Application-specific versus in-circuit emulation**

Application-specific debuggers are general-purpose, high-level debuggers. They notify OS of their intentions and thereby get notifications from OS when import events occur within one specific application.

There are some specific debuggers such as in-circuit emulators which sit between the operating system and the bare hardware. They can watch and monitor all processes and all interactions between applications and operating system. Typically these kinds of debuggers are low-level and are used for development of add-on hardware or for very special types of heavily system interaction applications (such as debuggers themselves) [4].

**2.2.2 Some Well-Known Debuggers**

• **GDB**

The GDB [27] is the GNU Project debugger, which allows you to see what is going on ‘inside’ another program while it executes, or what another program was doing at the moment it crashed.

There are still several basic debuggers in text mode like GDB, for example: DBX and JDB. Typically, such debuggers only provide a basic set of debugging commands. They serve as a basis for most GUI debuggers, so it is difficult to use these debuggers to
make complex query. A typical debugging process always needs to repeat similar steps such as set breakpoints and print variable values until the cause of defect is found.

- **JDB**

In Java, the typical debugger is JDB [28]. It is a simple command-line debugger for Java classes. It is a demonstration of Java Platform Debugger Architecture (JPDA) [29] that provides inspection and debugging of a local or remote Java Virtual Machine (JVM). Just like GDB in text mode, it provides a basic set of debugging commands. So it is difficult to make complex query.

- **DBX**

The DBX debugger is able to track the execution of your program line-by-line in the source code (C or FORTRAN) and tell you the status of every variable you are computing. It is also possible to tell it to watch a particular variable and report when it changes [25].

However, the initial effort in learning to use DBX is often repaid by the time it saves you finding programming errors. XDE Command [34] provides an AIXwindows interface to the DBX debug program. LDBX [21] is also an interactive front end for the DBX. It provides a terminal window for text debugging, and a graphics window for data structure displays drawn as graphs. The user may select pointer field to be traced as a means of controlling the size of the displayed graph.
Most debuggers can only print values of designated variables, which is of little help in answering complex queries. For example, “which elements of array x[100] are positive?”

In order to improve the query expressiveness, a high-level query language “DUEL” is designed by Michael Golan and David R. Hanson [7] specifically for source-level debugging of C programs. Expressions of DUEL are a superset of C language and it is implemented on top of GDB and a new command interpreter is built to evaluate DUEL expressions and display their result.
- **Dalek**

  Dalek [15] [16] is based on GDB. It is similar to many other modern debuggers. Dalek and the code being debugged execute as separate processes, with Dalek controlling, observing, and altering the code’s execution according to code written in Dalek languages.

  You can specify Dalek code dynamically during a debugging session, so you can specify what is of interest as you inspect information during execution. Dalek code can be executed from the command level or automatically at a breakpoint or when an event triggered. However, the behavior of setting breakpoints still manual.

- **MDL**

  It uses a form of dynamic code generation called dynamic instruction to collect data about the execution of an application. Specification of what data to collect are written in a specialized language called the Metric Description Language (MDL) [9], which is part of the Paradyn Parallel Performance Tools [10] [13].

  MDL allows platform-independent description of how to collect performance data. When a request is made to instrument the application program during execution, the specified description will be translated into machine code and inserted into the running program.
2.3 3D Graphical Programming

“3D Computer Graphics” are actually two-dimensional images on a flat computer screen that provide an illusion of depth, or the third “dimension”. There are many uses of 3D graphics in current computer application. Real-time 3D is applied to interaction game, simulation, and scientific, medical, business data visualization. More computing time, higher quality of images can be generated. Generally, we have to define models and scene, and then process these definitions to generate 3D images with high quality. The definitions of the scene and models is called scene graph, which include all objects and the relationship between them.

2.3.1 A Basic 3D Scene

An example of 3D scene is shown as Figure 2.4.

Figure 2.4 – A 3D Scene
A 3D scene can contain objects as follows: 3D models, lights, cameras, backdrops, billboards, terrain, force fields and audio. These objects have some common properties:

- They may have 3D shapes (even a light);
- They are movable;
- Some of these objects can be grouped;
- Some of them may look alike;
- They may have animation or they can deform;
- These objects may be affected by light
- Sometimes their attribute may be modified.

Because there are various objects in a 3D scene, in a large 3D scene, we need some data structures called scene trees or scene graphs to keep relationships between these objects.

2.3.2 Scene Graph

Scene graph is a directed acyclic graph. A scene graph contains all objects in the scene and relationships between them. A simple hierarchical scene graph is shown as Figure 2.5, and a node in the tree is shown as Figure 2.6. The main benefit to use the hierarchical scene graph is no need to concern the absolute position of each object in
the scene, just care about the position of the object related to its parent. The reason is hierarchical scene graph contains the information of some objects groups.

Figure 2.5 – A Simple Hierarchical Scene Graph

Figure 2.6 – A Scene Object Node

The root of the scene graph is usually at the origin of the scene. When a node do a transform to move its position and change direction, its children won’t change the relate position between them. For example, a sub-graph of scene graph composed of
bicycle is shown as Figure 2.7. Once the bicycle needs to move forward, the body should be move forward, and its wheels should also move forward and rotate. With the hierarchical scene graph, we just need to move the body forward and let two wheels rotate. There is no need to compute where the wheels should be shifted to. A wheel can concentrate on its rotation task. Scene graph is particularly useful when we need to construct a complex object composed by many parts.

![Figure 2.7 – An Example of Scene Graph: Bicycle](image)

There are another advanced scene graphs: Bounding Volume Hierarchies (BVHs), Binary Space Partition Trees (BSP Trees), Octree, Quadtree, and etc. These graphs are often used for visibility culling or collision detection.

### 2.3.3 An Overview of 3D Engines

3D engine is a program module that can be re-used to develop 3D programs. There are some types of 3D engines:
3D engines may be thin interface to hardware, and they are graphics assembly and with good performance. But it needs a lot of coding work to use this type engines. DirectX [26] is an example of this type.

They may only support 3D graphics only, and they are cross-platform, for example: RenderWare [30] 1.0. These engines provide a set of 3D graphics API which is composed by 3D assembly codes.

Some engines support full functions for 3D environments such as: collision detection, scene management, audios, and etc. The engines usually used for general purpose.

Other engines usually used for limited and specific purpose, such as: game, virtual reality. For example: Unreal [32] is an engine for first person shooting (FPS) games.

Although there are so many types of 3D engines, all 3D engines have the key elements:

- They support versatile functions and can be scalable for various machines;
- The semantics of engines’ API need to be consolidated and easy to learn;
- High-performance and stability are needed by good 3D engines;

Between 3D engines, they have a common feature: they always have a scene management, and users can maintain the scene graph with a set of APIs.
In this paper, metaphors are implemented with a 3D engine named “WTK” [31]. WTK is an API built on OpenGL which runs on PC, SGI, and more. It is easy to understand and use this API. Besides, WTK supports many kinds of input device, such as: digital glove, 3D space tracer, and etc. This feature is helpful to add other input devices in DIVINE and interactive with users.