Toward Arbitrary Mapping for Debugging Visualizations

Yung-Pin Cheng, *Member, IEEE,* Wei-Chen Pan, *Member, IEEE,*

Abstract

Despite the progress that has been made in the field of program visualization, programmers nowadays still rely on inserting extra code (e.g. print statements) to visualize complicated program states during debugging. There are many obstacles that have impeded and continue to impede program visualization for practical use. One such major obstacle is that a wide variety of data types and interpretations from data to visualizations are arbitrary. It is unlikely that visualizations will be available a priori to cover everything that might be of interest. In an attempt to address the problem, a debugging visualization tool called *x*DIVA is presented here. The visual effects of *x*DIVA are 3-D shapes, colors and animations from a 3-D rendering engine. *x*DIVA conducts a novel and meticulous object-oriented design so that visualization metaphors are interactive, composable and decoupled from data, i.e. a complicated visualization can be composed and assembled from basic ones, each of which is independently replaceable. A new mapping system that maps data to visualization is proposed to support such composition and assembling.

Index Terms

Debugging Visualizations, Mapping, Reusable Visuals, Visual Mapping Language.

- *Yung-Pin Cheng is with the Department of Computer Science and Information Engineering, National Central University, ZhongLi, Taoyuan, TAIWAN.*
  E-mail: ypcheng@csie.ncu.edu.tw

Toward Arbitrary Mapping for Debugging Visualizations

1 INTRODUCTION

Software systems are notorious for being difficult to understand and maintain. So, approaches that employ visualization to address the problem have been popular for decades. For example, UML has become a popular and successful approach to visualizing the structural aspects of software systems, which are considered more difficult to grasp through source code. Simply put, humans can assimilate complicated information much more efficiently if the information is represented by graphics, images or diagrams.

In many application domains, successful visualizations, no doubt, have been applied to communicate clearly and stimulate viewer engagement and attention. Still, the visualization problem remains a difficult one if the domain of data is not constrained, limited, or fixed. A simple example can explain the problem. Consider a class X with three attributes which have been poorly named.

```
class X {
 int a ;
 int b ;
 int c ;
};
```

The meaning of class X can be interpreted in several ways depending on the application domains. For example, X can be a vector in a 3-dimensional space. A visual with an arrow pointing to the 3D coordinate *(a,b,c)* in a 3D space is most suitable for the interpretation. In the domain of geometry, *a,b,c* may be the coefficients of a 2D line *ax+by=c.* A viewer often wants to draw the line on an XY-plane with a lattice graph. In many other cases, a viewer may expect a visualization that is computed indirectly from class X. One common example is to draw a pie chart based on $(\frac{a}{a+b+c}, \frac{b}{a+b+c}, \frac{c}{a+b+c})$ so that the composition of ratios for $a, b,$ and c can be easily understood.

In fact, such interpretations can go on and on forever. We call such a problem *"Diversity of Visualization Interpretation*s (DVI)." The diversity of interpretations is what makes the problem difficult. In general, the interpretations are often subject to domain knowledge and personal preference. So, given a set of data, it is unlikely that any fixed-mapping visualization will be

available a priori to cover everything that might be of interest. Any visualization tools that claim to provide "*automatic*" visualizations will always fail for other general cases.

Debugging visualization, unfortunately, is a difficult area that has been continuously impeded by the DVI problem. Every programmer learns *printf* debugging as their first debugging technique. However, printing individual variables is inadequate in many cases, particularly when complicated data structures are involved. With the help of visual debuggers (IDEs that provide window-based display for watching variables at break points), programmers can watch interested variables without changing and recompiling the code of a debuggee. Unfortunately, visual debuggers are as limited as simple printf statements. So, programmers often write debugging methods to print the variables and layout the data in a form that can be easily understood by the human eye. Sometimes, these debugging methods can be costly to implement if the size of data is large, layout algorithm is complicated, or aesthetic forms are critical.

In this paper, a debugging visualization tool – *x*DIVA (**D**ebugging **I**nformation **V**isualization **A**ssistant), designed and implemented to address the problem is described. To tackle the DVI problem, xDIVA conducts a novel and meticulous object-oriented design so that visuals are interactive, composable and decoupled from data, i.e. a complicated visualization can be composed and assembled from basic ones, each of which is independently replaceable. Such a design rationale is based on ancient human wisdom. When facing infinite construction choices, humans have learned to find common basic building blocks for all choices, set up the interfaces between the basic building blocks, and then establish engineering methods or frameworks to assemble these building blocks. The mappings between data types and visuals can be customized and reconfigured online by a visual language called *x*DIVA-VML. There is no need to write annotations or rule-like notations to specify the mapping. Some initial work of xDIVA has been published in [1] as a tool paper.

xDIVA is built on an open source 3-D engine – Ogre[2], which is an object-oriented graphics rendering engine. 3-D programming from basic libraries such as OpenGL and DirectX is considered difficult. A 3-D engine like Ogre considerably lowers the programming effort and time. Ogre provides many 3-D effects, which enrich the possibility of visualization and improve the visual quality beyond simple geometric shapes and colors.

This paper is organized as follows: Section 2 describes the background of xDIVA and a short overview of xDIVA is given in Section 3. The design rationale and object-oriented framework behind xDIVA are described in Section 4,5, and 6. Examples are shown in Section 7. Sections 8 and 9 complete the paper with related work and conclusions.

2 BACKGROUND

The work presented in this paper is closely related to a popular research area called software visualization. Software visualization research [3], [4] has been proposed to construct visual representations of information from software systems based on their structure[5], size[6], history[7], [8], or behaviors[9]. These tools visualize software metric data from measurement activities. The software metric data are either presented in static, animated 2-D, or 3-D graphics. They can be used to monitor the quality of source code during development and expose the anomalies earlier.

Another branch of software visualization is run-time visualization. Some focus on visual representations of performance analysis metrics such as memory usage, time spent in methods or classes, etc. Examples include BLOOM [10], Jive [11], and Jinsight[12]. Some focus on visualizing programs with object perspectives with UML-like diagrams, which include JIVE [13] and JaVis [14],[1] Although the purpose of these research tools vary, they are all aimed at helping programmers or project managers to debug and understand their software.

2.1 Visuals, Data, and Bindings/Mappings

Most visualization techniques use a set of (animated) visuals such as images, pictures, colors, or icons to exploit specific knowledge that users already have of other domains. The quality and quantity of visuals often have a direct impact on tool usability. Well-designed visuals and appropriate metaphors can help users understand data more effectively.

Since it is impossible to provide visuals to cover everything that might be of interest, a visualization tool often needs to establish a scope for the data they can visualize so that the quantity of visualization metaphors can be limited. Take the well-known visualization tool *Graphviz*[15] for example, it is designed to visualize graph-like data so it can focus on the graphical representations for node entities and edge entities that link nodes.

Although the quantity of visuals in a visualization tool is limited, it can be costly to implement these visuals. The cost and time devoted to programming visuals can vary considerably depending on the characteristics of an application domain, volume of data, aesthetic requirement, etc. So, maximizing the reuse of visuals as-built is a common goal for visualization tools. There are several common directions to approach the problem. In general, the objective is to seek out methods to "customize" or "reconfigure" the mapping between data and the as-built visuals. Research that addresses the customization of visualization can be found in [16], [17], [18], [19], where [17] made a thorough survey of this topic. In [17], the flexibility of specifying the mapping

1. JaVis is not a 100% visualization tool because its visualization is displayed by UML Case Tool Together.

between data and visuals are classified from "predefinition", "annotation", "association", and finally "declaration". The earliest work that links program variables with parameters of visuals which resembling xDIVA's approach can be found in [20].

Overall, reusing the visuals as-built has been one of the major challenges in visualization research. In general, the more complicated a visual is, the less reusability the visual can offer. So, the ability to share visuals among visualization tools has always been a goal. However, allowing users to reconfigure mapping is a key to the DVI problem. In the next section, more details will be explained.

2.2 Visualization of Program Internal Behaviors

Program behaviors are divided into two parts: One is observable to users (aka program output) and the other is internal behavior that is invisible to users but of concern to programmers. The branch of software visualization research that addresses the visualization of internal program behaviors is often called *program visualization*. However, most program visualization tools [21], particularly algorithm animation, are limited in their use for educational purposes. They are rarely come up in a programmer's daily work because of the DVI problem.

DDD (Data Display Debugger)[22] is a debugging visualization tool which allows one to see what is going on "inside" a computer program while it executes. From the best of our knowledge, DDD could be one of the few program visualization tools to actually be used by programmers. DDD is also a graphical front-end for command-line debuggers GDB but it also does more. DDD is known for its interactive graphic data display, where a memory block (object or struct) is displayed as a node (box) and pointers are displayed as directed edges. A node is displayed as a box (fixed visualization), in which variables in the memory block are arranged in a top-down layout and displayed in textual and numerical form. Arrays can also be displayed in a table-like visualization. If program data matches some limited types, such as a 2-D array of integers, DDD can send it to *gnuplot* to draw the array as functions in 3-D curve lines.

DDD has been developed into a mature open-source tool. Using it to debug programs in some domains can save considerable time and effort. Overall, *x*DIVA shares similar objectives with DDD but differs in the visualization support.

3 Basic Building Block Principles

The design rationale of *x*DIVA originates from within common human knowledge. When facing infinite construction choices, humans have learned to find common basic building blocks for all choices, set up the interfaces between the basic building blocks, and then establish engineering

methods or frameworks to assemble these building blocks. This knowledge can be explained by a term – *composability*. Composability[23], in general, refers to a system design principle that deals with the inter-relationships of components. A highly composable system provides recombinant components that can be selected and assembled in various combinations to satisfy specific user requirements. A component may cooperate with other components, but dependent components are replaceable.

So, to approach such a human knowledge in visualization, xDIVA needs to address two key issues: Composability and the use of basic build blocks. Composability, however, is never as easy as it may seem to achieve in programming. Decoupling data from visualization code is the first challenge to overcome.

3.1 Is decoupling by model/view paradigm possible?

A visual is a shape drawn inside a rendering system by a piece of code. Typically, the shape is controlled by some parameters or data. The poorest way to implement a visual is by directly referencing the data in the code. Any dependency on data in the code limits a visual to render only that data. Since in this poorest way of implementation, visualization code is completely aware of the data's memory size, structure, and distribution, let it be called "hardwired" if a piece of visualization code is written as such. To decouple the code from data, it seems that there are already a lot of coding tricks to resolve this issue, such as the Inversion of Control (IoC) principle. Unfortunately, creating a piece of visual code decoupled from data is easier said than done. For example, it is very common to write the following code, in which visualization code is one of the behaviors of the class.

```
class binary_tree_node {
   void draw() {
      ...// draw the center node
      ...// draw the left child pointer
      ...// draw the right child pointer
   }
}
```

Although it is object-oriented, it is obvious that the visualization code is coupled with the data type and thus not reusable by other types or problems. So, to further decouple the code, it is common to introduce the popular *model/view* paradigm (which is mainly evolved from the *Model/View/Control* (MVC) paradigm and is actually an observer design pattern). To adopt the paradigm, the code must have an observer (aka *view*) to hold the visualization, as in the following sample code. When registering an observer with a `binary_tree_node` object (aka *model*), and the model is modified, `update()` of `binary_tree_node_observer` is then invoked to redraw the visual.

```
class binary_tree_node_observer {
   binary_tree_node bt ; // ***
   void update() {
       ...// draw the center node based on bt
       ...// draw the left child pointer to bt.left
       ...// draw the right child pointer to bt.right
   }
}
```

In principle, several observers can be registered with an object of `binary_tree_node`. So, the views of `binary_tree_node` are configurable now. Although it is an improvement, the dependency of visualization code to the data type `binary_tree_node` is still not removed. Of course, you can use polymorphism to make `bt` become a pointer (see the line with comment `//***`) to a base class of an inheritance hierarchy, which could extend the types to be used by this observer to the inheritance hierarchy. However, in order to reuse the visualization code, you need to make your data types become the subtypes of such a base class, which is not exactly a good decoupling solution. Recall that MVC's major concept is to separate the GUI code from core computational (business) logic. It actually promises the benefits of easy maintenance and portability but not reusability of views. Conclusively, model view cannot release you from writing hardwired visualization code; configurable views to a specific data type are the major benefits for doing so.

To broaden visualizations which are reusable by different applications, Graphviz's meta format DOT [15] shows a design choice that completely decouples the visualizations from data. However, to draw your data by Graphviz, you need to translate your data into a DOT file on your own. In other words, the visualization code of Graphviz depends on a meta file, not any data types at programming level.

3.2 Maximum reusability with primitive types

One interesting finding from the work of xDIVA is to make a visual reusable, its shape can only be controlled/parametrized by "primitive types" (integer, float, etc.) in its visualization code. This finding can be simply explained. To draw a rectangle, the very basic APIs may be:

```
drawRectangle(int x,int y,int width,int height) { ... }
```

Such a fundamental API typically allows maximum reusability. For example, suppose you have 3D cube data and you want to draw it into a 2D rectangle, you simply slice the data of the third dimension to reuse the visualization code. On the other hand, it is common to see additional APIs are supported to promote OO principle such as:

```
drawRectangle(Rectangle rect) {...}
```

If your goal is to allow this visualization code to be reused, such a move is actually realized in an opposite way. This API is only suitable for data of type `Rectangle`. By introducing more

complicated data types to draw a shape, the reusability of the code is degraded. Furthermore, suppose you want to implement a visualization code which can draw the visuals for a generic binary tree node. The following API may be designed.

```
drawBinaryTreeNode(BinaryTreeNode node) {...}
```

This visualization code opposes the reusability because it depends on a type called `BinaryTreeNode` and it is too complicated a shape to be a reusable unit.

In debugging scenarios, variables are unlikely to be structured to fit pre-defined models. It is also impossible to predict all the types of variables and code them as a priori. So, how should a visual for rendering `BinaryTreeNode` be implemented? The answer is *"Don't implement it but compose it instead."*

3.3 Ultimate Basic Visuals

According to the observation above, xDIVA must implement a set of basic shapes as basic building blocks, whose shapes are solely controlled by primitive-type parameters. Constructing basic shapes in xDIVA requires programming, debugging and testing. Such programming work in general requires creating an adjustable mesh (composed by triangles) for the shape and wrapping the mesh into a class. Although programming a basic shape is not hard (a cone shape has 150 lines of C++ code), it should not need to be done frequently. Recall that in principle it is impossible to implement all the visualizations as a priori. So, if too many basic shapes are required to make xDIVA really usable, it is still a dead end.

Here, similar to searching for basic particles that make up matter, we introduce a type of basic shapes called "Ultimate Basic Visuals."

Definition 1. An *Ultimate Basic Visual* (UBV) is a shape which cannot be transformed and assembled from other basic visuals. The transformation includes scaling and rotating a visual on an XYZ-axis or setting the controlling parameters to extreme values.

For example, should we implement a rectangular cuboid with a shape that can be controlled by three parameters *width, length, height*? The answer is no, because a rectangular cuboid can be transformed by scaling a cube on one XYZ axis. So a rectangular cuboid is not a UBV but a cube is. A frustum in Fig. 1 is another example of an UBV. By making $R_1 = R_2$, you can make a cylinder and by making $R_2 = 0$, you can make a cone. An arrow shape like Fig. 2 can be assembled from a cone shape and a cylindrical shape; so it is definitely not a candidate of UBV. By introducing the concept of UBVs, the number of basic shapes is reduced considerably. However, users can see that some non-UBVs are displayed as basic shapes. Non-UBVs are

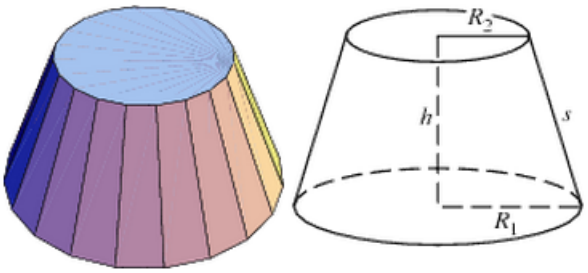

Figure 1. A frustum as an UBV.

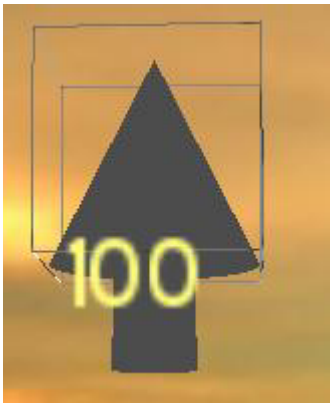

Figure 2. The arrow VM composed from a cone and a cylinder UBVM.

provided to users for the sake of convenience, since users may not possess the concept behind UBVs.

Each UBV, in general, is controlled by several 3D properties. Take a cube for example. A cube shape can be determined by properties shown in Table 1. These properties are called *default properties.* In addition to default properties, most shapes typically require additional parameters to determine their shapes. For example, a sphere has a property called *radius.* A cone shape has a *radius* and *height.* A pie UBV is shown in Fig. 3(a), which is controlled by additional parameters θ_1,θ_2 and h. To visualize data into a pie chart, θ_1,θ_2 for each section UBV can be computed from the data of the object and then a pie chart like Fig. 3(b) can be constructed. Also, by considering different values of h, the fancy visualization in Fig. 3(c) can be done in such a compositional way.

property group	parameters	notes
position (coordinates)	rx,ry,rz	relative position at xyz-axis
size	xize yszie szie	size of bounding box
scale	xscale yscale zscale	scale factor of bounding box
color	r,g,b	primary color
rotation	pitch, yaw, roll	rotation degress
rotation	quaternion	rotation by quaternion
transparency	trans	transparency of visuals (0..1.0)

Table 1

The default parameters of an UBV.

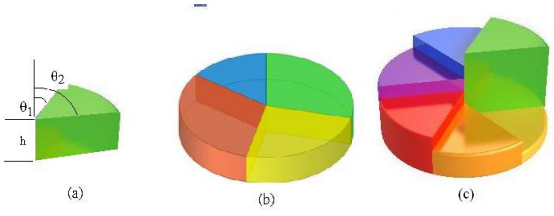

Figure 3. (a) The pie UBV. (b) and (c) The pie-chart visualizations that can be assembled from (a).

4 MAPPING ENGINE

Upon making visuals decoupled from data types (other than primitive types), the next challenge is how to let data control visuals and let users compose visuals to make up a visualization they want. Any data, eventually will be read, processed, and stored in the memory of a computer program. When a program (debuggee) is stopped at a break point, a programmer can probe/view/explore variables and variable structures interactively via a debug interface or framework. xDIVA relies on plugins to communicate with the debug frameworks of Visual Studio (C/C++) and Eclipse (Java, C/C++ via CDT). Fig. 4 shows the high-level architecture of *x*DIVA. Releases and tutorials of *x*DIVA can be accessed at *x*DIVA's website:

`http://oolab.csie.ncu.edu.tw/wiki`

In Visual Studio or Eclipse, xDIVA's plugin provides a button called "visualize". When a break point is hit, a programmer can use the button to visualize a variable when visualization is needed. In this section, an overview along with an example is given to describe the basic

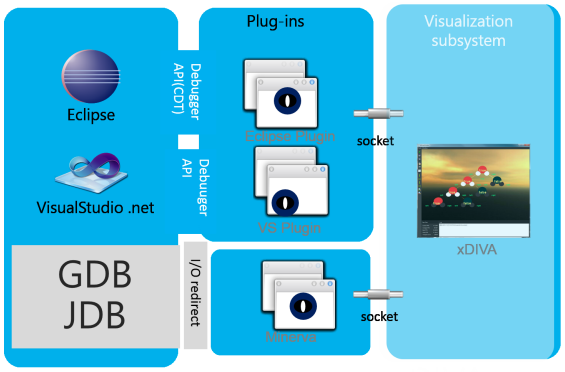

Figure 4. The architecture of xDIVA and debuggers

features of *x*DIVA.

4.1 A mapping example

When a variable is visualized the first time, a mapping dialog (see Fig. 5) pops up displaying the variable names and values in the panel A. The functions of panel A are similar to a debugging watch window in conventional IDEs. If a variable is a compound variable (such as an object, an array, a pointer or reference), you can unfold a compound variable by clicking on the plus icon (⊞). When an unfolding action is invoked, xDIVA talks to the plugins to retrieve more memory contents from the debugger.

Suppose an object of a binary tree node *bt* (in Java) in the following

```
class bt {
  boolean travel ;
  bt left;
  bt right;
};
```

is visualized. Suppose a programmer wants to visualize the *bt* object to satisfy the following visualization requirement:

1) Use a ball with radius 50 to represent the variable *travel.* If travel is true, its color is red, otherwise, the color is black.
2) Reference *left* and *right* are drawn as smaller balls (radius 25) attached to southwestern (-50,-50,0)/southeastern (50,-50,0) corners of *travel*. When clicked, the ball shall emit a laser at the object it points to.

In the beginning, panel B is empty. Panel B is called the editing area of a visual programming language called xDIVA-VML. xDIVA-VML is a data-flow type visual programming language.

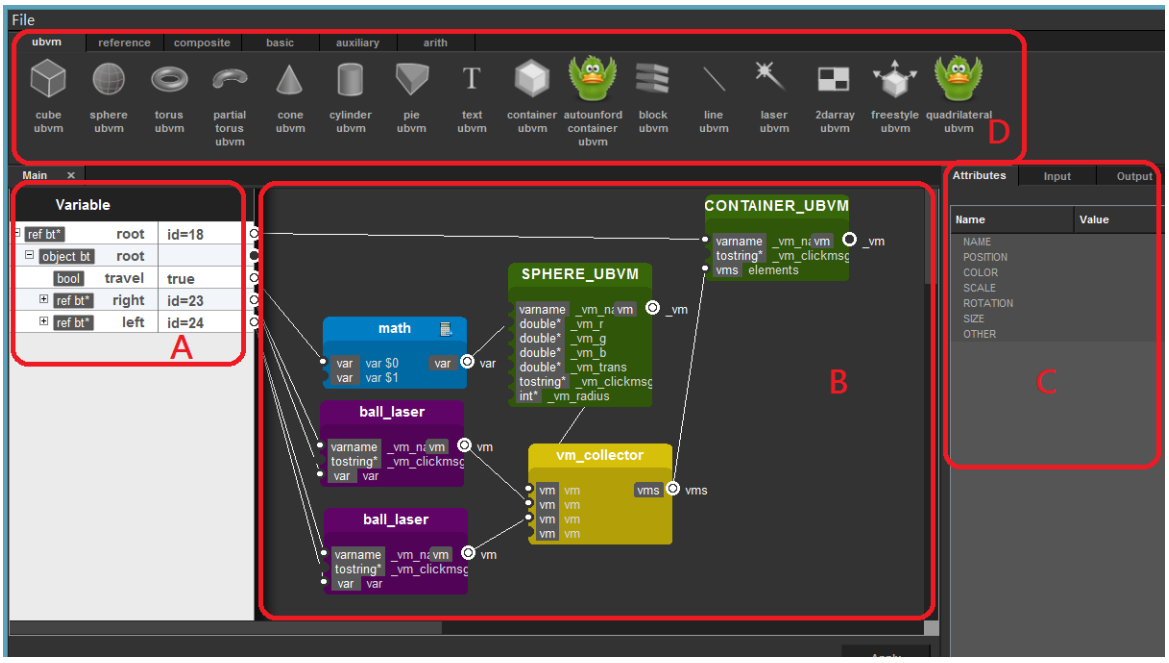

Figure 5. The mapping dialog a binary tree node example.

The computation units of xDIVA-VML are called *mapping nodes*. A mapping node has input ports (left-hand-side) and output ports (right-hand-side). As seen in Fig. 5, the data flow begins from the variables in panel A. Linking a variable to a port of a mapping node can be understood intuitively. For example, if you link an integer variable to the radius of a sphere, the variable controls the size of the sphere.

In panel D, there are different types of mapping nodes categorized by their purposes. In Fig. 5, category UBVM (Ultimate Basic Visualization Metaphor) is shown, which contains the mapping nodes that render UBVs. To complete the visualization, mappings can be constructed in panel B by:

1) Drag and drop a `math` mapping node from panel D.
2) Drag a link from variable `travel` to the math mapping node's input port $0.
3) Set the math formula as "$0 * 1.0", so that the output of the math mapping node is 1.0 when `travel` is true and 0.0 when `travel` is false. xDIVA transfers boolean values into 0 and 1 and this math mapping node transfer integer values into float.
4) Drag a sphere UBV T and by default its initial relative location is $(0, 0, 0)$.
5) Drag a link from `travel` to $T's$ input port `varname` to tell xDIVA that visualization of T represents variable `travel`.
6) Link the result of math node to sphere T's input port - red color (ranging fro m 0..1.0) of

RGB.

7) Drag a ball laser UBV L. A ball laser UBV renders the source of a pointer/reference by a sphere and shoots a laser at its appointed object.
8) Open the location attributes of the ball laser UBV L (see panel C) and set the relative location as $(-50, -50, 0)$. All input ports of a mapping node have default values if no link is connected to them. If you want to set the values directly, it can be done by the input port property window in panel C.
9) Drag a link from `left` to L's *var* input port to let the `left` control the $L's$ behaviors and drag a link from *left* to $L's$ `varname` to tell *x*DIVA that L represents the variable `left`.
10) Drag a ball laser UBV R. Open the location attributes of the ball laser UBV R and set the relative location as $(50, -50, 0)$. Drag a link from *right* to R's *var* input port to let the *right* control $R's$ behaviors and drag a link from *right* to $R's$ `varname` to tell *x*DIVA that R represents the variable *right*.
11) Drag a *vmcollector* mapping node V and drag links from the vm output ports from T, L, R to V. A `vmcollector` can collect an arbitrary number of vm objects and output a type of vms, whose concept is similar to a vector type in OO programming.
12) Drag a *container_ubvm* C and drag links from V to C. A *container_ubvm* can group all the vm in vms under a scene graph node. The scene graph is a structure that arranges the logical and often (but not necessarily) spatial representation of a graphical scene. In addition, C will mask and dispatch mouse events for T, L,and R. A click event to T, L, and R will always be handled first by C and then$C$ will dispatch the event downward to its child vm.
13) Drag a link from object *bt* to the `varname` of C to tell xDIVA that C represents the compound object *bt*. So, when there is a pointer shooting a laser to this *bt* object, the laser should target C instead of T, L, R.

Once the mappings are done, a visualization of the *bt* object is shown in 6. In 6, only a *bt* node will be visualized first. The two laser balls (for references *left* and *right*) can be clicked. Once they are clicked, the variable pointed by *left* and *right* will be unfolded. A new *bt* object will pop up new mapping dialog for you to specify the mappings again.

4.2 Mapping Visual Language (xDIVA-VML)

What xDIVA proposes in this example is a visual programming language (VPL) for constructing a visualization. Typically, visual programming languages are designed and adopted for the following reasons:

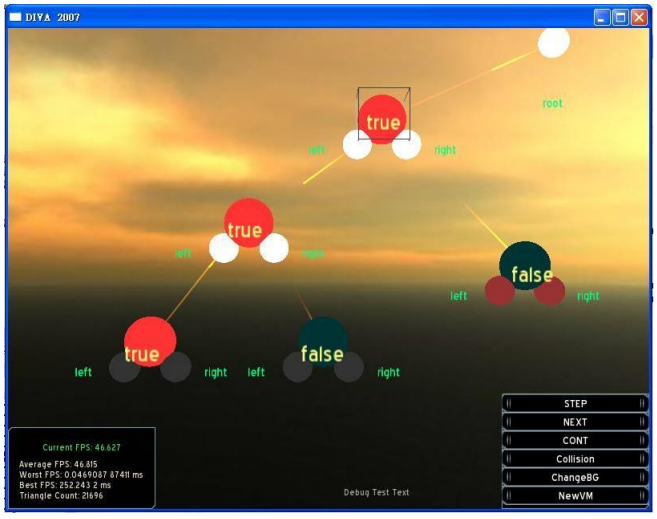

Figure 6. The visualization of bt.

- easy to learn and use
- no need to memorize syntax or fix compilation errors as in textual-based programming languages
- easy to debug

Some of the popular visual programming languages are lego's NXT-G[24] and scratch[25]. It is comparatively much easier to start programming from these VPLs because these VPLs typically let users create programs by manipulating program elements graphically rather than by specifying them textually. However, VPLs are rarely used as general-purpose programming languages due to their limitations, particularly when problem complexity and scale are increased.

*x*DIVA introduces xDIVA-VML for a very practical reason: Programmers are already impeded by compilation errors and runtime errors as a daily basis. If using a debugging visualization tool forces programmers to learn and write a new programming language, fix syntax errors, and debug the runtime errors, this tool is simply too inconvenient and troublesome to use. With xDIVA-VML, a mapping should be easily completed the first time within seconds or minutes.

4.2.1 Data-flow Semantics

Unlike most VPLs which are control-flow based, xDIVA-VML works as a data-flow system. Data values and data properties flow from the lefthand side, starting from visualized variables, to the righthand side. A *mapping node* (see Fig. 7) is responsible for accepting values and properties from its input ports and working as a computation unit to produce results in its output ports. There is only one link allowed to connect to an input port but there are many links allowed to

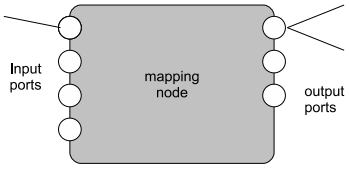

Figure 7. An abstract mapping node.

start from an output port. There are no cyclic links permitted. These mapping nodes and links constitute a *mapping net*.

The computation unit of a mapping node can be responsible for rendering a 3D shape (rendering-based) or not (non-rendering based). A rendering-based mapping node is called a *vm*(Visualization Metaphor) node, which in general has all the default properties in Table 1 as input ports and output ports. Once a *vm* mapping node is executed, a 3D shape is constructed in the scene and a *vm* object is produced in an output port for other mapping nodes to utilize.

4.2.2 Rendering based mapping nodes

Reference VM

Besides the rendering based mapping nodes that construct UBVs, another kind of basic visual is reference *vm*. Reference *vm*, as the name implies, is used to render a pointer/reference. Currently, there are four kinds of reference *vm* which are implemented (shown in Fig. 8). A pointer/reference is a variable as well. So, in visualization, a programmer may choose any shape to represent a pointer. The first three reference *vm* allow you to install any *vm* to represent the pointer variable. In the previous *bt* example, a ball laser is an exact composite *vm* in which laser reference *vm* has a sphere installed as the pointer-side *vm*. An orbit reference *vm* utilizes the animation power of a game engine, in which, when pointer-side *vm* is clicked, an ogre head appears to orbit the target object.

The first three reference *vms* are all designed to unfold a pointer when pointer-side *vm* is clicked. When a pointer is unfolded, the target object is retrieved from the debugger and the visualization of the target object is added to the scene. The last auto-unfold reference *vm* is especially important for a language like Java. An auto-unfold reference *vm*, as the name implies, would unfold its pointer automatically when it is rendered in the scene. An auto-unfold *vm* has no pointer-side *vm* but has a location p. When its target object *vm* is added to the scene, the target object *vm* is moved to the location p. Auto-unfold reference *vms* are frequently used for Java array. Unlike C/C++, Java array elements are all references. It is natural to use auto-unfold reference *vm* to render Java array elements.

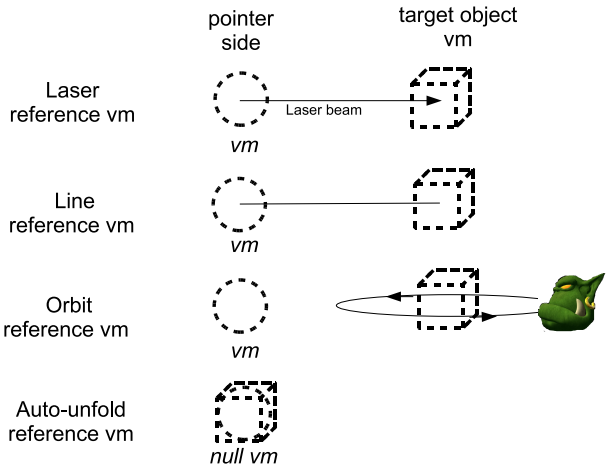

Figure 8. Basic reference *vms*.

4.2.3 Non-rendering based mapping nodes

In many visualizations, it is common that the relationship between data values and visuals is not straightforward. For example, drawing a pie-chart from three data values a, b, c involves computation of $(\frac{a}{a+b+c}, \frac{b}{a+b+c}, \frac{c}{a+b+c})$. So, xDIVA-VML must provide auxiliary mapping nodes to support the various needs of visualizations. Due to the limited space, only mapping nodes worthy of mention are described here.

Math mapping node

A math mapping node is a frequently used mapping node in xDIVA-VML. Many data values can be linked to a math mapping node and they are indexed as $0,$1,$2... consecutively. Clicking the math node allows you to enter a formula using $0,$1,$2... to compute a math formula.

Container mapping node

A container mapping node is another frequently used mapping node in xDIVA-VML, which is a key to the composability mentioned in the previous section. Just like many graphical editing applications which can group objects into a composite one, it is essential to group a set of *vms* into a composite *vm*. What a container mapping node does is to recompute the bounding boxes of child *vms* and glue them in a scene graph. A scene graph is a general data structure commonly used by vector-based graphics and 3D game engines that arranges the logical and spatial representation of a graphical scene. A container mapping node is also responsible for dispatching mouse/keyboard events to its child *vms* recursively.

String composition mapping node

When a program or an algorithm is wrong, appropriate visualizations can manifest the errors easily. However, when an error is found, a programmer needs to trace the problem back to its source code. If clicking a visual can display important information in the code, such as object id, debugging efficiency can be greatly increased.

xDIVA does not predefine any clicking message/behavior for each vm. Instead, an input port called "`_vm_clickmsg`" can be used to specify the clicking message with the help of a string composition mapping node. This mapping node is similar to `math` mapping node, which allows arbitrary links to connect to its input ports and these inputs are indexed from \$0,\$1,\$2...etc. A user can use these inputs to compose a string. For example, in the pie-chart visualization, when a pie UBV is clicked, we want a message like "Percentage ratio: x%" to be displayed. First, we use a math node to compute the value x and then link it to a string node at \$0 input port. Right click the string node to enter the string "Percentage ratio \$0%" and it is done.

Clock mapping node

xDIVA is built on an open-source 3D engine. Animation is the major strength of such technology. An animated visualization can be more informative than a static one in many cases. For instance, to visualize a variable named `speed`, people would find that connecting the variable to any static 3D shape is awkward. The concept of speed, on the other hand, can be better visualized by animated visualizations, such as spinning a cube faster if `speed's` value is larger. Such a need can be satisfied by a clock mapping node. A clock mapping node has four input ports (`low`, `high`, `increment`, `time_interval`) to set up. It works as a counter to generate values repeatedly from `low` to `high` by the `increment` in every `time_interval` seconds. Use a math node to compute `1/speed` and then set the clock mapping node as $(0, 360, 10, 1/speed)$. The clock mapping node generates 0,10,20,....,360,0,10... repeatedly every `1/speed` second. To make a cube spin, simply link clock node's output to one of cube's rotation input ports.

5 Type Mapping to Memory Block Structure

When a break point is hit, visual debuggers typically provide a "watch" feature to display the contents of interested variables in a watch window. Since it is often not necessary to display the whole memory of a debuggee, a debugger only retrieves and updates variables of interest which could be related to a bug. In summary, a programmer can add variables to the watch list or unfold an array, an object, or a pointer to retrieve the memory block. There is, however, some inconvenience to this approach. Take DDD for example. Suppose you have a linked list

which contains 100 nodes. To visualize the linked list, you need to display the head node of the linked list first and then click the pointer variables one by one. Basically, the features of xDIVA described in previous sections mimic this approach. However, in practice, if allowed, most programmers would expect much more. For example, every time a debuggee is rerun or a new break point is hit, the whole linked list should be updated and visualized again automatically. This simple goal, however, introduces a lot of problems that must be overcome.

5.1 Dependency of Visualizations

In order to explain the contents in this section, a representative example is given here to show how complicated visualizations can be and how these problems are overcome in xDIVA. The representative example is graph data structure. Following is a common implementation of two basic elements `node` and `edge` in a graph.

```
class node {
  int id ;
  edge edges[3] ;
}
class edge {
  int cost ;
  node start ;
  node end ;
}
```

Suppose there are three node objects n_1, n_2, n_3 and three edge objects e_{12}, e_{13}, e_{23} instantiated, where $e_{ij}$is an edge between n_i and n_j. A common visualization is to draw n_i as a ball and e_{ij} as a line between n_i and n_j. Now, suppose we want to visualize them one by one manually. Since drawing e_{ij} requires the location of n_i and n_j, an attempt to visualize e_{ij} before $n_i$and n_j is doomed to fail. This problem introduces the concept of visualization dependency. In hardwired visualization, the problems of visualization dependency are typically solved in the code. In xDIVA, in order to repaint the visualizations automatically, we must deal with visualization dependency.

Definition 2. A visualization mapping v is a mapping net which transforms a set of *data values* into a set of (animated) 3D shapes.

Let's continue the example. Suppose $n_1$and n_2 are successfully visualized as balls . Let the *vm* of n_i be B_i. Next, we attempt to visualize e_{12} from the debugger plugin. `start` and `end`

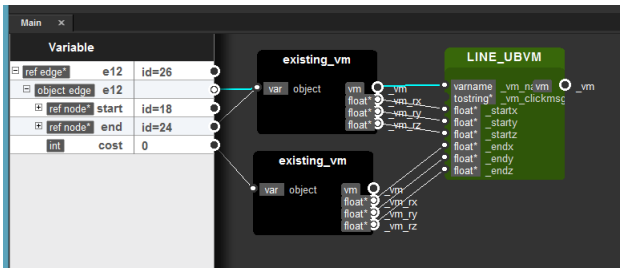

Figure 9. Existing *vm* mapping nodes to construct *e12*.

of e_{12} are n_1 and n_2 respectively and already have *vms* $B_1$and B_2 as their representative visuals. So, there are two tasks which must be completed by xDIVA:

1) When e_{12} is visualized, its attributes `start` and `end` are retrieved.
2) xDIVA detects that `start` and `node` already have representative *vms*. They should not be unfolded further to construct new visualizations.
3) How is a line to be drawn between $B_1$and B_2?

xDIVA's solution is to display two black mapping nodes both labeled "existing vm mapping node" in the edit area (see Fig. 9). Users can treat these two mapping nodes as vm of B_1 and B_2. As in the figure, a line UBV can use the location of B_1 and B_2 to draw a 3D line to finish the visualization of e_{12}.

Definition 3. A visualization mapping v is said to be dependent on u (denoted $v \prec u$) if v must use $u's$ *vm* properties to determine the shape/location/color/rotation (aka basic properties) for rendering.

Tools such as DDD which provide only fixed visualizations and do not allow you to construct a visualization seem to be able to avoid the visualization dependency problem. However, for tools that claim to be capable of constructing arbitrary visualizations, ignoring the visualization dependency causes failure as per the above simple example. In addition, visualization dependency is crucial when we want to automatically visualize a set of objects. Given a set of mapping nets, the order of visualizations can be computed by simply parsing the mapping net to determine the dependency.

5.2 Type Mapping

The visualization of the graph example, of course, can be manually constructed piece by piece as described, but the whole process is exhausting. In most visualizations, the visualizations of B_1, B_2, B_3 are likely to be the same. Once a mapping of B_i is specified, typically it would be

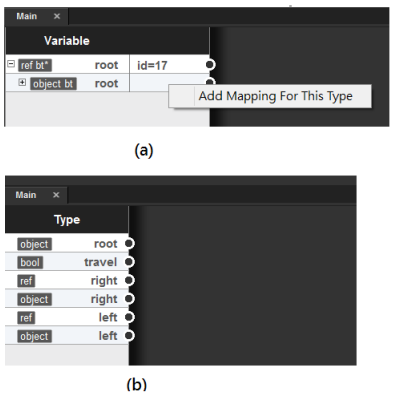

Figure 10. Type mapping GUIs.

applied to all i in the memory. To address the problem, xDIVA introduces a concept called *type mapping*. With type mapping, visualization mapping can be automatically applied to homogeneous memory blocks. This is an important key to automatic visualization.

Definition 4. A *type visualization mapping* t is a mapping net which transforms a set of *data types* to a set of (animated) 3D shapes.

Note that, Definition 4 is different from Definition 2, where data values are replaced by data types. To specify a type mapping in xDIVA of a type t, you just need to visualize an object of t first. For example, in Fig. 10(a), when the first *bt* object is visualized, a user can right click the *bt* object to bring up a type mapping dialog. In type mapping dialog, variable values are no longer displayed. If attributes of a class is a pointer/reference or array, one level of unfolding is automatically done (see Fig. 10(b)) for the users. If an attribute is a pointer/reference to a type s and s already has a type mapping, a black mapping node called "existing type mapping node" will automatically be displayed in the dialog. That is, visualization dependency between two type visualization mapping also exists.

Definition 5. A type visualization mapping s is said to be dependent on a type mapping t (denoted $s \prec t$) if s must use $t's$ *vm* properties to determine the shape/location/color/rotation (aka basic properties) for rendering.

In practice, type mapping is more commonly used for most visualization tasks because homogeneous objects are too common once data is processed and stored in the memory. Fig. 11 shows the difference between a type mapping and a normal mapping. The type mapping features suggests that

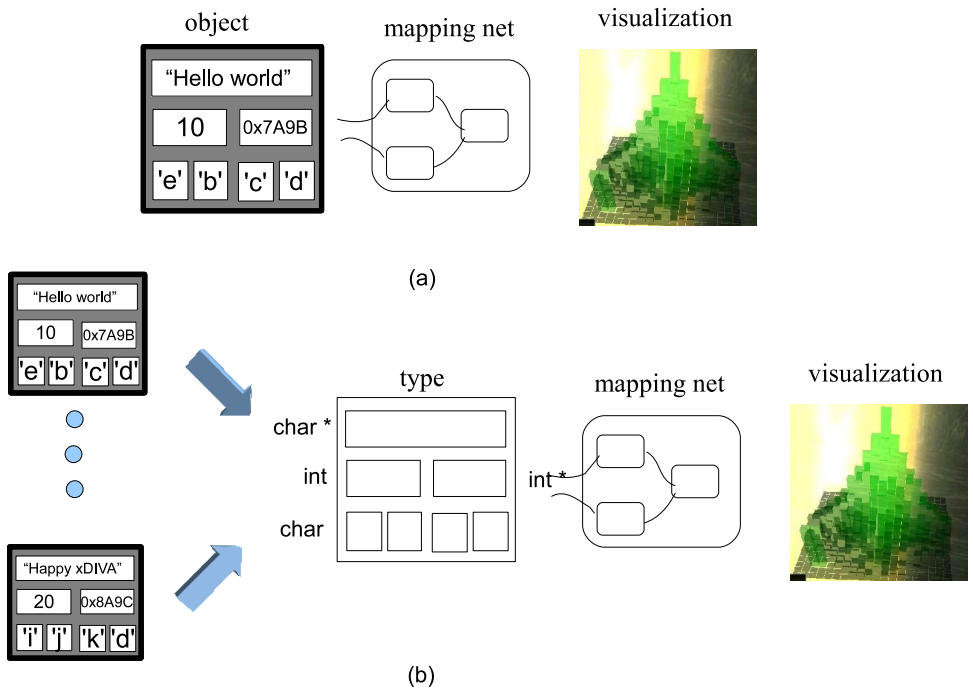

Figure 11. (a) Mapping between data values and visualization. (b) Type mapping between data types and visualization.

If homogeneous memory blocks exist, specifying the mappings from data types to visuals instead, not data values,

5.3 Memory retrieval bot

So, as shown in Fig. 11(b), xDIVA must allow programmers to retrieve memories of interest so that all the homogeneous memory blocks can be fit to the type mappings to create visualizations automatically. This can be an easy task if you are writing hardwired visualization code. Unfortunately, a tool like xDIVA must deal with arbitrary memory distribution from different programs.

Once data are stored in memory, the data is seldom uniformly distributed in contiguous memory addresses. Take a linked list for example, a linked list node can only be accessed from the node that precedes it. To address the problem, xDIVA allows you to specify a memory retrieval bot. A memory retrieval bot can be initiated by visualizing a variable as a start (see Fig. 12(a)). Then, users can specify the paths (by giving the names of pointer/reference variables) for the bot to traverse,retrieve the memories and stop.

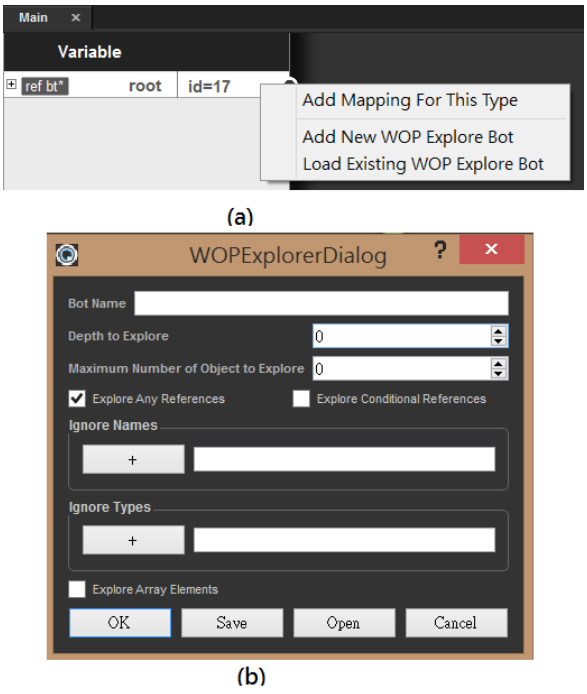

Figure 12. Memory retrieval bot.

6 LAYOUT

Layout algorithms are algorithms that deals in the arrangement of visual elements in a scene. Any visualization tools cannot ignore layout. Popular tools such as *Graphviz, iGraph, Zest* [26], [15] have built a lot of layout algorithms for specific domains. One major problem to reuse these algorithms is the lack of unified interfaces.

In xDIVA, layout is needed at three places:

1) The arrangement of logical and spatial representation in a composite *vm:* When a set of child *vm*s is grouped into a composite *vm*, the positions of child *vms* can be set manually by users. The positions of child *vms* can also be automatically determined by a container which comes with a layout algorithm.
2) A new *vm* created by clicking a reference *vm.*
3) The arrangement of logical and spatial representation in a scene: When a set of *vm*s is visualized in the scene, if their positions are not bound to data values, they can be arranged automatically in an aesthetically pleasing way by a layout algorithm.

Primary layout concern should focus on item 3. In an earlier version of xDIVA, a set of APIs for manipulating the *vms* in the scene was constructed. Then, some simple layout algorithms (e.g., binary tree) were implemented for the purpose of showcasing xDIVA. Unfortunately,

following such a design, the layout algorithms must be compiled with xDIVA, which prevents layout algorithms from being easily added to xDIVA. xDIVA's upcoming solution is to allow other tools or libraries to easily hook on, particularly iGraph and Graphviz. For example, by default, Graphviz's dot operates in filter mode, reading a graph from stdin, and writing the graph on stdout in the `DOT` format with layout attributes appended. If you run `dot -Tdot`, it will generate an attributed `DOT` which adds the layout coordinates of nodes and edges. These coordinates (albeit of 2D) can be parsed to place the *vms* in xDIVA's 3D scene.

7 Examples and Evaluation

7.1 Examples

In Fig. 13, screen shots of several examples are shown. Fig. 13(a) shows xDIVA is still capable of visualizing data in the conventional way. Using these *vms* makes the visualization power of *x*DIVA equal to DDD. In Fig. 13(b), a cube is used to render an image pixel. By linking the RGB values of a color pixel to the cube's color properties, the array of image pixels are visualized. Using the string composition mapping node, when a cube is clicked, the RGB values are displayed as a string "(R, G, B)". Fig. 13(c) continues the example with a larger image. In Fig. 13(d), a 2D array of integers are visualized. Each integer in the array element is linked to a cube's `zsize` property and color properties so that the larger the integer, the longer the cube and brighter the color.

Fig. 13(e) demonstrates the cases that locations of *vms* are controlled by the visualized variables. The underlying program of this example to solve the traveling salesman problem (TSP). In many application domains, such as computational geometry, wireless sensor networking problems, etc., linking coordinates of entities to a visual's location properties is a common visualization mapping. To make the visualization more exquisite, a green flat board is added. Such a green flat board is called a background *vm*, the purpose of which is to enhance a visualization. Background *vms* can be added to the mapping net but typically there are no links from data to these *vms.* The properties of this green flat board can be set manually from the property window C in Fig. 5.

Fig. 13(f) shows a practical application of xDIVA in an area called Electronic Design Automation (EDA)[27]. EDA companies produce software tools for designing and producing electronic systems ranging from printed circuit boards (PCBs) to integrated circuits (IC), which are also known as CAD tools in the IC industry. The real data structures of a VLSI design describes layers of 2D polygons, which is difficult to debug without visualizations. The visualization in Fig. 13(f) is a VLSI polygon layout which renders similar data structures from an EDA tool.

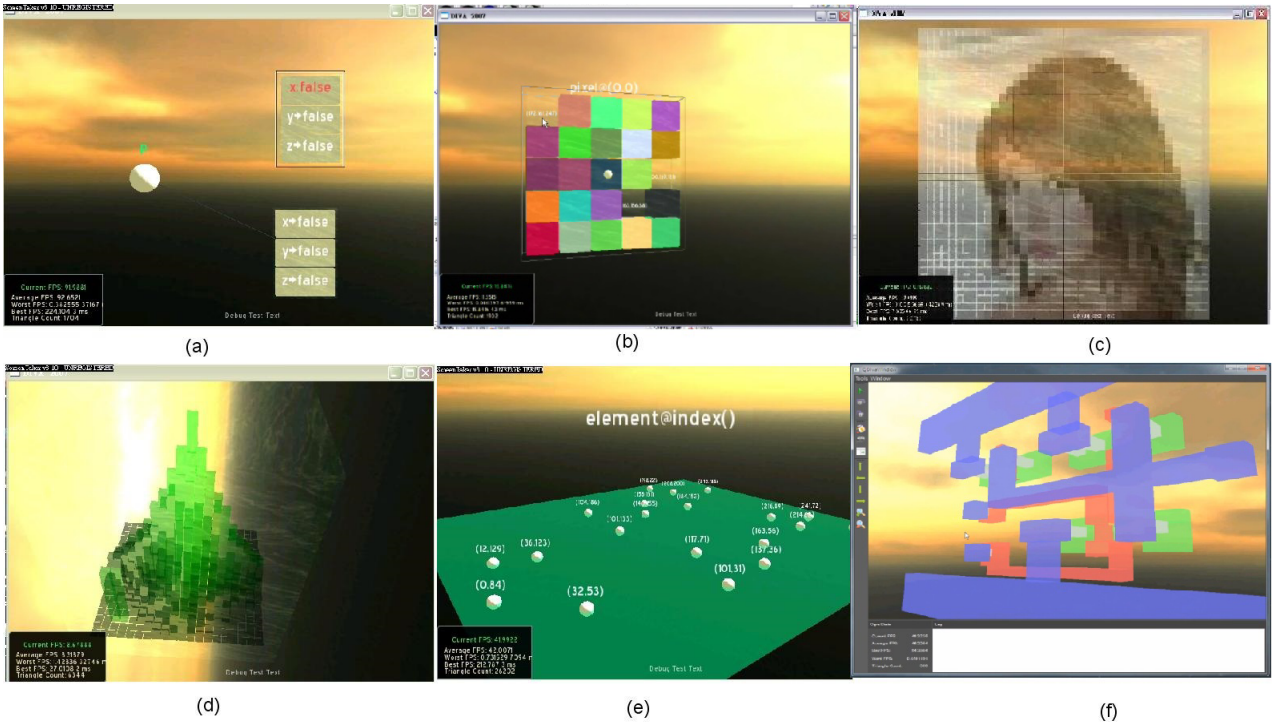

Figure 13. Six visualization examples.

So far, it seems that no complicated shapes are composed from the simple ones for debugging visualizations. In our opinion, debugging visualization is fundamentally different from general visualization. The main purpose of debugging visualizations is debugging. So, as long as the purpose is fulfilled, it doesn't matter if the visualizations are exquisite or not. However, xDIVA, in principle, can be used as a general visualization tool. More works will be done in this direction.

7.2 Evaluation on basic building block principle

The results of this research can be mostly concluded by the key xDIVA features. They are:

- Ultimate basic visual principle
- The design and implementation of xDIVA-VML
- The mapping configuration GUI for xDIVA-VML
- Type mapping concept and its support such as type existing mapping nodes.
- Visualization dependency between visualization mappings

From the beginning of xDIVA's development, many examples from different domains have been chosen to test xDIVA (see Fig. 2). For example, the graph problem is such a representative

Application Domains	Algorithms or Data structures
Basic data structures	trees (binary trees), graph, array, ...
Computational geometry	convex hull problems, ...
General algorithms	travel salesman problems, ...
Common charts	histogram, bar chart, pie chart, ...
Networking algorithms	radio network problems, ...
Image processing	image pixel array, region growing,...

Table 2

Application domains and examples from the domains.

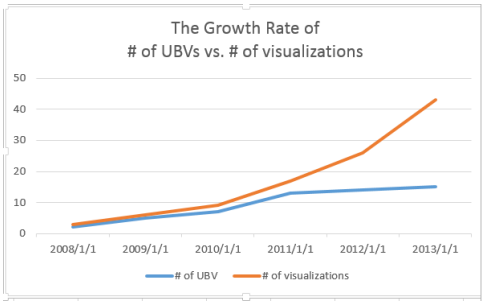

Figure 14. The growth rate of # of UBVs and # of visualization.

example that xDIVA must introduce an existing mapping node and type existing mapping node to xDIVA-VML. Sometimes, existing UBVs cannot compose a visualization suitable for the example; this is when a new UBV is implemented. For example, when trying to visualize a radio network problem, a ring shape UBV is implemented so that visualization can show the communication radius of a radio node. We browse the xDIVA's version control database and extract the date when a UBV was implemented. In Fig. 14, the number of UBVs and the number of visualizations through these years are compared. Apparently, the number of UBVs increases very slowly, which support the assertion that the basic building block principle is the correct direction in which to continue.

8 Related Work

Among the research tools mentioned in Section 2, *Lens* [28] is a debugger-based visualization system that is worth mentioning. It is an early tool that attempts to bridge the gap between debugger and algorithm animation systems, such as [29], so that the work does not limit their use in pedagogy. The focus of *Lens* is to produce animation by associating animation actions to source code where interesting event animation commands are attached to debugger

breakpoints. The problems and difficulties that could be encountered when using debugger-based systems for visualization have been well and thoroughly studied. Simple mapping can be configured in a pop-up dialog, such as associating variable values with the positions of graphic objects. Although Lens focuses on algorithm animation (where graphic objects are represented by very simple shapes) and does not address the arbitrary visualization interpretations, it is an influential work on later debugger-based visualization systems.

In general, debugger-based visualization systems do not necessarily include algorithm animation capability, though algorithm animation is one of the debugging-aid approaches. More recent research for algorithm animation includes JIVE[11], SWAN[30], JAWAA [31], LIVE[32], SKA [33], etc. These tools, however, are mostly limited to educational purposes.

Some systems, such as jGRASP[33], JIVE [13] and JaVis [14], provide limited views for data structure visualization and place more emphasis on UML-style visualization

The large number of possible visualization variations is a problem faced by all visualization tools. As described in previous sections, it is expensive and impractical to build pre-defined visualizations for each task. So, suitable visualizations ought to be *developed* on demand. Online mapping configuration or customization of visualization is a goal pursued by all visualization tools. Much of the research in the field of software visualization [3], [4], [6], [34], [35] which supports visualizations for software metrics can be configurable in different degrees, within the limits of model-view paradigm. For example, Vizz3D [34] proposes a *model-view-scene-controller* paradigm, where the bindings are configurable and specified in an XML script. Different paradigm variations, such as *model-scene-controller* and *model-view-controller,* are also studied in this research area. In general, the *model-view* approach is inapplicable in debugging visualization because the data produced in debugging has no models to which it should conform. In other words, software visualization research has developed a clear knowledge and understanding of what analysis data (e.g. software metric) should be collected and organized. The mapping flexibility is considered constrained within the paradigm.

9 Conclusions and Future Works

The initial idea of xDIVA has been published as a tool paper in [1] and its support for program animation was also published as a tool paper in [36]. In this paper, major design principles that make xDIVA different from other visualization tools and its major technical advantages are described.

So far, the lines of code of xDIVA already reach 80K. To increase the tool usability, xDIVA began with a debugger front-end called Minerva and now has two plugins for Visual Studio

and Eclipse. Maintaining a visual program language like xDIVA-VML is not an easy task but we believe it is the right direction to commit ourselves to.

xDIVA-VML is still a growing language. In principle, a visual programming language lowers the learning curve and reduces the programming efforts, but its expressive power is compromised. So, it is possible that some visualizations may not be achieved by current versions of xDIVA-VML and UBVs. Currently, some part of xDIVA is under refactoring so that new mapping nodes or new UBVs can be added more effortlessly and can be contributed outside the xDIVA team.

REFERENCES

[1] Y. Cheng, J. Chen, M. Chiu, N. Lai, and C. Tseng, "xdiva: a debugging visualization system with composable visualization metaphors," in *Companion to the 23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2008, October 19-13, 2007, Nashville, TN, USA*, G. E. Harris, Ed. ACM, 2008, pp. 807–810. [Online]. Available: http://doi.acm.org/10.1145/1449814.1449869

[2] S. Streeting, "The object-oriented graphics rendering engine (ogre)," *available on http://www.ogre3d.org*. [Online]. Available: http://www.ogre3d.org

[3] S. Diehl, "Software visualization," *Lecture Notes in Computer Science*, vol. 2269, 2002. [Online]. Available: citeseer.ist.psu.edu/zeller95ddd.html

[4] C. Knight, *System and Software Visualization. In Handbook of software engineering and knowledge engineering. Vol 2, Emerging technologies (Vol.2)*.

[5] M. L. Staples and J. Bieman, "3-d visualization of software structure," *Advances in Computers*, vol. 49, pp. 96–143, 1999.

[6] M. Lanza, "Codecrawler - polymetric views in action," in *Proceedings. 19th International Conference on Automated Software Engineering*. Los Alamitos, California: IEEE Computer Society, 2004, pp. 394–401.

[7] T. Girba, A. Kuhn, M. Seeberger, and S. Ducasse, "How developers drive software evolution," in *Proceedings of International Workshop on Principles of Software Evolution*. IEEE Computer Society, 2005, pp. 113–122.

[8] F. L. Lopez, G. Robles, and B. J. M. Gonzalez, "Applying social network analysis to the information in cvs repositories," in *International Workshop on Mining Software Repositories (MSR 2004)*. IEEE Computer Society, 2004.

[9] A. Kuhn and O. Greevy, "Exploiting the analogy between traces and signal processing," in *Proceedings IEEE International Conference on Software Maintainance (ICSM 2006)*. IEEE Computer Society, 2006, pp. 101–106.

[10] S. P. Reiss, "An overview of bloom," in *PASTE*, 2001, pp. 2–5.

[11] S. P. Reiss and M. Renieris, "Demonstration of jive and jove: Java as it happens," in *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA: ACM Press, 2005, pp. 662–663.

[12] W. D. Pauw, E. Jensen, N. Mitchell, G. Sevitsky, J. M. Vlissides, and J. Yang, "Visualizing the execution of java programs," in *Software Visualization*, 2001, pp. 151–162.

[13] P. V. Gestwicki and B. Jayaraman, "Methodology and architecture of jive," in *SOFTVIS*, 2005, pp. 95–104.

[14] K. Mehner, "Javis: A uml-based visualization and debugging environment for concurrent java programs," in *Software Visualization*, 2001, pp. 163–175.

[15] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo, "A technique for drawing directed graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–230, 1993.

[16] M. E. Tudoreanu, D. Hart, and G.-C. Roman, "Reshapeable visualizations," in *ISMSE*, 2000, pp. 245–252.

[17] G.-C. Roman and K. C. Cox, "Program visualization: The art of mapping programs to pictures," in *ICSE*, 1992, pp. 412–420.

[18] P. L. 0002, C. C. Gomberg, and S. F. Roth, "Visage: Dynamic information exploration," in *CHI Conference Companion*, 1996, pp. 19–20.

[19] G.-C. Roman and K. C. Cox, "Declarative visualization in the shared dataspace paradigm," in *ICSE*, 1989, pp. 34–43.

[20] T. G. Moher, "Provide: A process visualization and debugging environment," *IEEE Trans. Software Eng.*, vol. 14, no. 6, pp. 849–857, 1988.

[21] "The complete collection of algorithm animation," *available on http://www.cs.hope.edu/alganim/ccaa/index.html*.

[22] A. Zeller and D. Lutkehaus, "DDD - a free graphical front-end for UNIX debuggers," *SIGPLAN Notices*, vol. 31, no. 1, pp. 22–27, 1996. [Online]. Available: citeseer.ist.psu.edu/zeller95ddd.html

[23] "Composability," http://en.wikipedia.org/wiki/Composability, accessed: 2014-09-18.

[24] "Lego nxt visual programming languages," http://en.wikipedia.org/wiki/Lego_Mindstorms_NXT#NXT-G, accessed: 2014-09-21.

[25] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond, "The scratch programming language and environment," *TOCE*, vol. 10, no. 4, p. 16, 2010. [Online]. Available: http://dx.doi.org/10.1145/1868358.1868363

[26] G. Csardi and T. Nepusz, "The igraph software package for complex network research," *InterJournal, Complex Systems*, vol. 1695, no. 5, 2006.

[27] "Electronic design automation," http://en.wikipedia.org/wiki/Electronic_design_automation, accessed: 2014-09-18.

[28] S. Mukherjea and J. T. Stasko, "Toward visual debugging: integrating algorithm animation capabilities within a source-level debugger," *ACM Trans. Comput.-Hum. Interact.*, vol. 1, no. 3, pp. 215–244, 1994.

[29] J. T. Stasko, "Tango: A framework and system for algorithm animation," *IEEE Computer*, vol. 23, no. 9, pp. 27–39, 1990.

[30] C. A. Shaffer, L. S. Heath, and J. Yang, "Using the swan data structure visualization system for computer science education," in *SIGCSE*, 1996, pp. 140–144.

[31] S. H. Rodger, "Introducing computer science through animation and virtual worlds," in *SIGCSE*, 2002, pp. 186–190.

[32] A. E. R. Campbell, G. L. Gatto, and E. E. Hansen, "Language independent interactive data visualization," in *Proceedings of ACM SIGCSE 2003*. ACM, 2003, pp. 215–219.

[33] I. James H. Cross, T. D. Hendrix, J. Jain, and L. A. Barowski, "Dynamic object viewers for data structures," in *SIGCSE '07: Proceedings of the 38th SIGCSE technical symposium on Computer science education*. New York, NY, USA: ACM, 2007, pp. 4–8.

[34] T. Panas, R. Lincke, and W. Löwe, "Online-configuration of software visualizations with vizz3d," in *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*. New York, NY, USA: ACM, 2005, pp. 173–182.

[35] A. Marcus, L. Feng, and J. I. Maletic, "3d representations for software visualization," in *SoftVis '05: Proceedings of the 2005 ACM symposium on Software visualization*, 2003.

[36] Y. Cheng, H. Tsai, C. Wang, and C. Hsueh, "xdiva: automatic animation between debugging break points," in *Proceedings of the ACM 2010 Symposium on Software Visualization, Salt Lake City, UT, USA, October 25-26, 2010*, A. Telea, C. Görg, and S. P. Reiss, Eds. ACM, 2010, pp. 221–222. [Online]. Available: http://doi.acm.org/10.1145/1879211.1879251

PLACE
PHOTO
HERE

Yung-Pin Cheng Yung-Pin Cheng received the M.S. degree in computer science from National Chiao-Tung University, HsinChu, Taiwan, in 1991, and the Ph.D. degree in computer science from Purdue University, West Lafayette, IN, in 2000. He is currently an Associate Professor with the Department of Computer Science and Information Engineering, National Central University, Zhongli, Taiwan. His research interests include software visualization, software design and analysis, software verification, software testing, and computer science education.

Wei-Chen Pan Wei-Chen Pan received the M.S. degree in computer science from National Central University, Zhongli, Taiwan, in 2013.