Chapter 2 Background

2.1. Visualization

Visualization at large is defined as “a cognitive process performed by humans in forming a mental image of a domain space [11].” It is often said “a picture is worth a thousands words.” Using a map as an example, we can easily have an idea of location, distance, and etc. This is the basic idea behind visualization. The main goals of visualization are [12]:

- Maximize human understanding, communication, and collaboration without huge effort.
- Make evident the meaning of abstract entity.
- Reduce the complexity of phenomena.
- Enhance understanding of concepts and processes.
- Gain new insight.

This is why visualization has been widely used in the scientific area in general and computer science in particular.

Visualization can be classified into two areas, scientific visualization in which the data sets are derived from measurements or simulations grounded in physical space, and information visualization where the data sets are defined over abstract spaces [1].

With the more actual data sets around our life, the goal for scientific visualization is more intuitive than that of information visualization with the abstract data sets.

A necessary task of 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 n-dimension (n is usually 2, 3 or 4) space which can then be rendered using techniques from scientific visualization. Some researches [6] [7] have discussed these issues.

Data sets are always one of the research topics in the visualization community. How to store, access, and manipulate numerous data sets and then producing, using, and perceiving the presentation of visualization have been described in many relative researches [8] [9] [10].

2.1.1. Software Visualization

Information visualization appeared in the 90's with the objective of visualizing any kind of data in order to find patterns in a short time. Before that, visualization in computer science had a long history starting with the development of "Flow Charts" in the 60's with the work of Donald Knuth. In the 70's, "Pretty-Printing" was another major step followed by "Algorithm Animation" in the 80's. The 90's symbolize the revolution area with the introduction of CASE (Computer-aided software engineering) tools (e.g., Visual Studio, Rational Rose). In the last several years, the focus is on

"Dynamic Data and Software Processes [11]."

With the huge progress in computer graphics over the last decade, lots of visualization tools have been introduced in the computer world. In nowadays, high-level GPU (Graphic Processing Unit) has become as complicated as CPU. The price of 3D graphics video card has dramatically reduced. This hardware progress has made PC more powerful than graphic workstations several years ago. This trend of computer graphics combined with the pressing needs in software comprehension and maintenance led to the birth of software visualization as an independent field [11].

Software visualization can be defined as a "discipline that makes use of various forms of imagery to provide insight and understanding and reduce complexity of the phenomena under consideration [14]." Since software is an abstract element without spatial representation like a car or a computer, it's somehow difficult to project software into some forms our mind can easily understood. That is why software visualization is often referred to as the process of "making the invisible visible [15]."

In recent years, software visualization has two broad categories: algorithm visualization and program visualization. Algorithm visualization emphasizes the visualization of high-level abstractions that describes the behavior of the software. Flow charts and UML diagrams can be thought of typical examples of the static algorithm visualization. The most interesting part of algorithm visualization is to show

how the algorithm operates. Program visualization, on the other hand, often focuses on low-level program code. Figure 2-1 is the classification of software visualization introduced by J. Stasko et al. [13] and shows the related fields of software visualization.

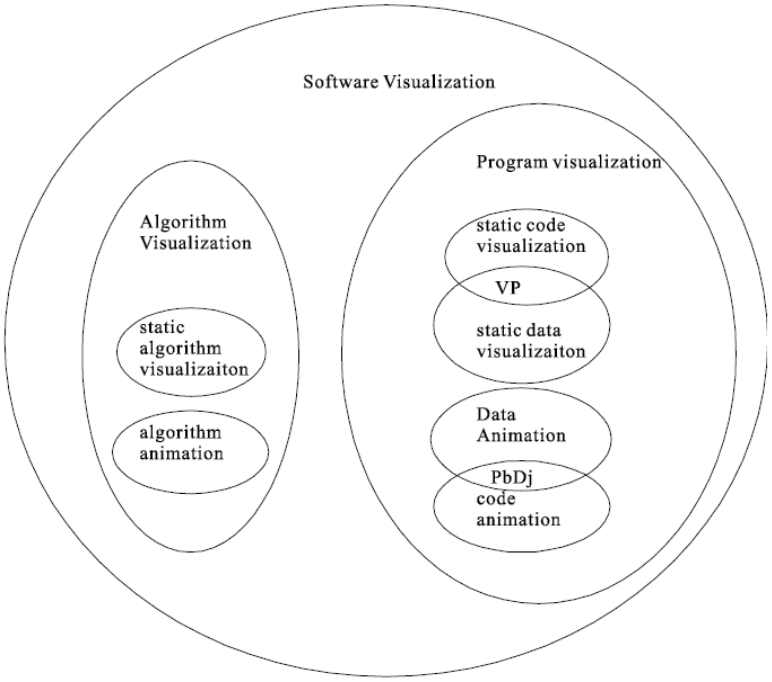

Figure 2-1: The domain of software visualization

Within the classification, VP (Visual Programming) and PbD (Programming by Demonstration/examples) in program visualization have different objectives in contrast to that of software visualization. The goal of VP and PbD is making the programmer to develop the software more easily and efficiently with visualization techniques, whereas software visualization makes an algorithm or a program be understood more clearly.

The main challenges of software visualization listed by Young [16] are:

- *Metaphor (representation)*: how to represent entities of software.
- *Abstraction*: what to present and in how much detail.
- *Navigation*: how to move through the sets of visual objects that represent the system.
- *Correlation*: how to link visual abstractions to source code and documentation.

As Koike [17] pointed out, "information has no shape and color." This disadvantage makes people hard to comprehend information. So, people use suitable virtual shapes to resolve this difficulty. By using the most popular and simple 2D visualization techniques, programmers can trace source code in a tool environment, where it is pretty-printed with indented lines, colored key words, and etc. But when programs become larger and more complex, the assistance from these visualization techniques is no longer sufficient [12].

In order to overcome this problem, some approaches have been developed and applied to analyze software artifacts and visualization tools are then used to display the relationship between visual objects. These approaches [12] include impact analysis, program slicing, dependency analysis, and etc. Visualization techniques combined with these approaches have been adopted by algorithm animations, debuggers, and testing tools.

2.1.2. 2D versus 3D Visualization

With software becoming more complicated and larger, the contents needed to be visualized are more complex. When showing the contents in a narrow 2D displayer, users are forced to watch information in a view port. Related information can be divided by the view port, which makes the visualization difficult to use. Even though we can navigate to the other part of the visualization by manipulating the view port, some information and the relationship among visualized objects are inconvenient to comprehend. In other words, 2D visualization forces human to adapt to the restriction of hardware, which causes low usability when the amount of visualization information scales up. This is a problem of field of view (FOV). As shown in Figure 2-2, a call graph of a medium sized system, is useless because of its poor visibility [12].

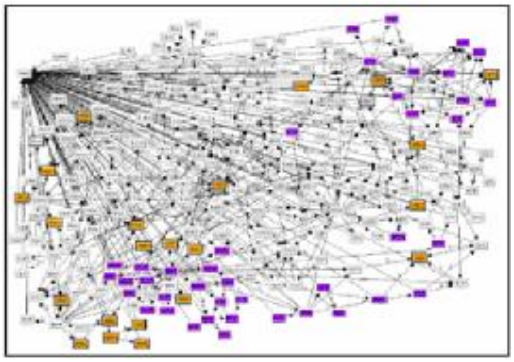

Figure 2-2: A call graph from a medium size system [12]

Lately, some researches have shifted towards 3D because of the progress of graphic techniques. With the following analysis, we can see how the third dimension employed in 3D visualization removes the restriction in 2D. *First*, 3D makes more

effective use of available screen space. It can use the additional dimension to encode more information. *Second*, from some principles of cognitive science and human perceptual system, 3D software visualization can provide the intuitive exploration and interaction of the system [19]. In many 3D applications, human can take advantage of wide range of a user's perceptual senses and their ability to navigate intuitively without any transformation [20], i.e., the learning curve is smooth.

Young [16] has made a raw classification of 3D techniques listed as follows:

- *Mapping form the data domain to the visualization space*: cityscapes that extended from 3D bar charts.
- *Information representation techniques*: perspective walls, cone trees and cam trees, 3D-Rooms, and information cubes.
- *Dynamic information visualization techniques*: fish-eye views, emotional icons, and self-organizing graphs.

Besides the techniques mentioned above, there are still some other techniques that applied to software visualization tools like human centered approaches, clustering, automate information filtering, pan + zoom, shading and etc [12].

2.1.3. Visualized Debugger

Debuggers are critical tools for finding bugs in software and are often combined into integrated development environment (IDE). These debuggers are so-called

"general purpose debugger (GPD)." Most IDE nowadays also integrates other tools, such as editor, compiler, linker, file dependence checker, version controller, and etc. It goes without saying that the integration of these tools increases the efficiency in software development. Although debuggers have been used frequently, their functions have made no obvious progress compared with other tools, like editors.

There have been some researches (see [28], [29], [30], and [37]) using 2D visualization to show debugging information. Within these debuggers, for example, they can display linked-list or graph-style data structures as shown in Figure 2-3.

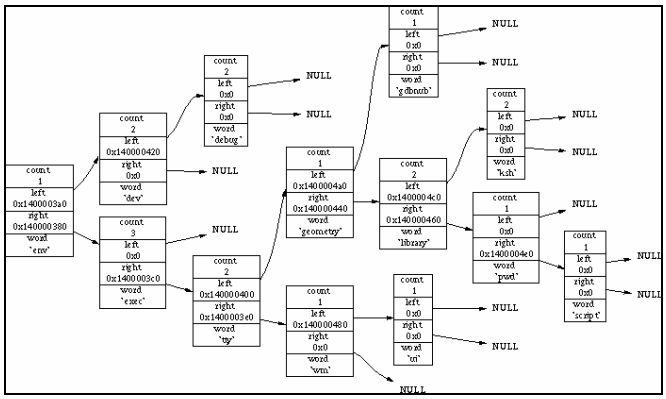

Figure 2-3: An output example of 2D visualized debugger [29]

Theoretically, visualizing the state of programs should help debugging programs. But, such researches in visualized debugger haven't been broadly accepted for some reasons. These reasons can be explained in the following:

- The cores of debuggers are somehow platform dependent. In visualized debuggers, rendering systems are typically platform dependent. These reasons make visualized debuggers have low portability as well as wide acceptance.

- As we mentioned in the previous section, the display in 2D visualization has the problem of low usability when visualizing massive data set. Take Figure 2-3 for example, this kind of visualization is well-known by computer-science experts, however, graph-model problems, such as determining node location and planning layout, often does not scale well.
- The rendering functionalities provided in these debuggers lack flexibility and variation. Most visualized debuggers start with familiar data structures like graph model or linked list as shown in Figure 2-3. Once users use data structures of other kinds, the visualization can not make any sense in debugging. In other words, the visualized objects can be suitable for particular data structures but can be poor to render others.

For the reasons mentioned above, there is room for further improvement in visualized debuggers.

2.2. An Overview of 3D Engines

The term “game engine” arose in the mid-1990s, especially referred to 3D games such as first person shooters (FPS). In early days, games are built from scratch, such as *id Software's* [25] *Doom* and *Quake*. Afterward, third party developers licensed the core portions of the game engine and designed their own graphics, characters, weapons and levels - the "game content" or "game assets." Later games, such as *Quake 3* and

Epic's [26] *Unreal* were both designed by this approach, where game engine development and design of contents are separated [21].

Reusable engines make developing game sequels much easier and faster, an advantage in the competitive computer game industry. The continued refinement of game engines has allowed a strong separation between rendering, scripting, artwork, and level design. Modern game engines are written by some complex applications, and frequently featuring dozens of finely tuned systems interacting to ensure a finely controlled user experience [21].

2.2.1. Game Engine versus 3D Engine

Wikipedia, the on-line free-content encyclopedia, defines "game engine" as "The core software component of a computer game. It typically handles rendering and other necessary technology, but might also handle additional tasks such as game AI, collision detection between game objects, and etc [21]." With the definition, a game engine can be broken down into the following parts: rendering engine, audio engine, networking engine, scripting system, tools programming, AI engine, and physics engine.

A 3D engine, or more precisely a 3D graphics engine, is usually considered as "an engine that only provide real-time 3D rendering." Such engines include *Genesis3D* [22], *Irrlicht* [23], and *Ogre* [24].

Either game engines or 3D engines have the common key elements:

- Support a variety of functions and have scalability for various machines.
- The semantics of engines' APIs need to be identical and easy to learn.
- High-performance, stability and platform-independency.
- The scene graph management with a set of APIs for user managing the scene graph.

Thus, an engine is large enough that the principles of object-oriented design must be practiced with great care.

3D engines usually provide lots of visual effects, such as the simulation of weather, explosion, particle effect (see Figure 2-4), water ripple, and etc., to create various fantastic visualization metaphor and human-computer interaction (HCI). In our research, we will concentrate on visualization and HCI but not the core techniques of computer graphics. By using game or 3D engines, we can get excellent 3D rendering result with less effort.

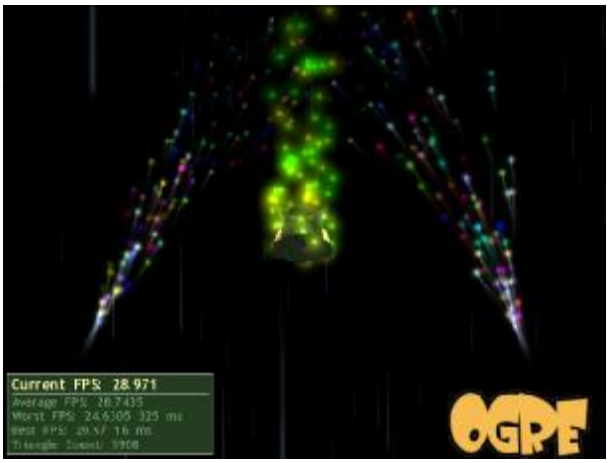

Figure 2-4: A snap shot of particle effect demo in OGRE

Most of commercial engines have various professional effect systems and full supports in various functionalities but expensive in price, whereas free or open source engines have the problem of maintenance and development. We have surveyed an open source 3D engine - OGRE that fits most of our needs in visualization. We will discuss it in next section.

2.2.2. OGRE – Object-oriented Graphics Rendering Engine

As the name stating, OGRE is simply a rendering engine. Described in the introduction on the official website, OGRE is a scene-oriented, flexible 3D engine written in C++ designed to make it easier and more intuitive for developers to produce applications utilizing hardware-accelerated 3D graphics. The class library abstracts all the details of using the underlying system libraries like Direct3D and OpenGL and provides an interface based on world objects and other intuitive classes [24].

With the following features, we choose OGRE as our 3D rendering system:

- Simple and easy to use object-oriented interface designed to minimize the effort required to render 3D scenes
- Some common requirements like render state management, hierarchical culling, and dealing with transparency are done automatically to save the development time.
- Support cross-platform input and both Direct3D and OpenGL APIs.

- Support skeletal animation, progressive meshes (Level of detail, LOD), particle systems, bill-boarding, and transparent objects.
- Available under the GNU Lesser General Public License (LGPL), thorough tutorials, API documents, and complete discussion forum.

For more features designed in OGRE, refer to the introduction about OGRE [24].

Although OGRE is nothing but a rendering engine which is deliberately designed to provide a general graphics solution, it has been used in some commercial games as described in [27] with its high flexibility to be incorporated into other libraries, such as sound, networking, collision, physics, and etc. With these features given above, we believe that OGRE is an adequate engine for our visualization.

2.3. 3D User Interface[1]

"Easy to use," "intuitive," or "designed with your needs in mind" are usually touted by most computer-related products, though most of them force users to adapt to the various interaction styles of these products. Modern users are familiar with typing words on keyboards, opening files with mouse clicks, selecting commands on pull-down menus, and watching "windows" in monitors, and etc. But, these designs are inappropriate for the nontraditional computing environments and applications under development today. For example, an application may allow a user to rotate a 3D

[1] Some content of in this section are extracted from [31].

object in a virtual environment, making the design of a 2D mouse inadequate.

Thus, these nontraditional applications need a new set of interface components: new devices, new techniques, and new metaphors. Most of these nontraditional environments work in real or virtual 3D space, so these new interfaces are termed *3D user interfaces* [31].

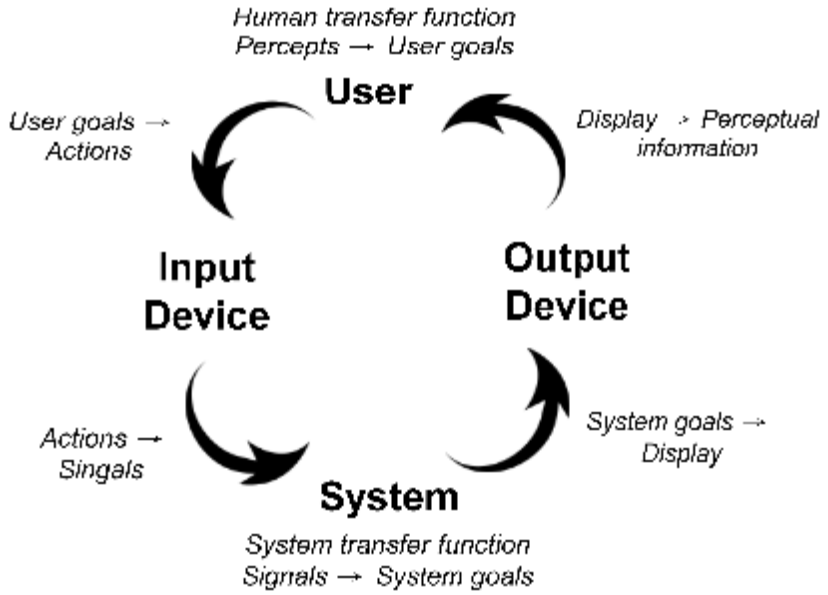

Figure 2-5: Human-computer communication through devices and translations.

Human-computer interfaces (HCI) or user interfaces (UI) serve as translators between a user and a system because they do not speak the same language. In fact, as shown in Figure 2-5 above, there are multiple translating steps involved: the user first translates her or his goals into actions; next, the input device translates those physical actions into an electronic form for the system; finally, the system deciphers those signals based on the current system state. All of the communication in 3D UIs must use

physical devices – hardware that serves as the medium of communication between the parties [31].

D. A. Bowman et al. [31] summarized some reasons for using 3D UI as follows:

- *3D interaction is relevant to real-word tasks.* For example, virtual environments provide users with a sense of presence, i.e., the feeling of "being there," which makes sense for applications such as gaming, training, and simulation.
- *The technology behind 3D UIs is becoming mature*. The standard parts of traditional WIMP (Windows, Icons, Menus, and Pointers) interfaces are still popular, but nontraditional devices and interface components are proliferating rapidly. These include *spatial input devices* such as trackers, and whole-hand devices that allow gesture-based input; *multi-sensory 3D output technologies*, such as stereoscopic projection displays, and head-mounted displays (HMD).
- *3D interaction is difficult.* Physical world contains many more cues for understanding and constraints for action that cannot currently be represented accurately in a computer simulation environment. Traditional WIMP interactions cannot provide a complete solution to 3D applications. So, 3D user interfaces based on real-world interaction or some other metaphor must be developed.

2.3.1. Input Hardware

Input devices, by which the users give information to the system, can be used in

3D UIs of both immersive and desktop applications. An important part of 3D UI design is choosing the appropriate set of input devices that allow the user to communicate with the application. For example, we may use a light-pen in path drawing without using a mouse to mark points along a path.

There are many characteristics can be used to describe input devices. The most important is the degrees of freedom (DOF), which is a particular and independent way that a body moves in space. Taking a tracker for example, it generally captures three position values and three orientation values for a total of six DOF. Thus, we often use DOF to describe how complex an input device is. D. A. Bowman et al. [31] also summarized other ways to characterize input devices listed below:

- *Input type and frequency of the data generated by input devices.* Data can be classified into discrete components (e.g., boolean value), continuous components (e.g., real numbers, coordinates, etc.), or a combination of both.
- *Physical interaction required to use the device.* With *purely active* devices, they will not provide any information to the system unless they are manipulated in some way, while *purely passive* devices continue to generate data even if they are untouched.
- *The purpose of the device.* Some devices are designed as locators; some are valuators which produce values, while some are used to indicate some choices.

Other characteristics include relative or absolute measurement, direct or indirect input, allows position or rate control, and etc.

There have been many different kind of input devices developed and used in 3D UIs. In [31] , these devices are categorized as follows:

- *Desktop input devices.* These devices, such as keyboards, 2D mice, joysticks, and pen-based tablets, have been traditionally used in 2D desktop applications and most of them work well in 3D UIs. They are generally purely active to provide information.
- *Tracking devices.* These devices provide information about users or physical objects' location in 3D space and most of them are purely passive. The most common devices are motion trackers, eye trackers, and data gloves. Figure 2-6 shows an example of data glove.

Figure 2-6: A data glove of Fifth Dimension Technology. (http://www.5dt.com/)

- *3D mice.* These devices are defined broadly as handheld or worn input devices that combine motion tracking with a set of buttons and other discrete components,

like sliders, knobs and dials. Contrary to regular 2D mice just moving along a flat surface, users can physically move these devices in 3D space to obtain position and orientation information.

- *Direct human input.* These techniques, such as speech, bioelectric, and brain input, obtain data directly from signals generated by the human body.

Besides the devices mentioned above, there are also many special-purpose input devices designed for specific applications, such as shape tape [38], slippers [39], transparent palettes [40], and etc.

2.3.2. Output Hardware

Output device (or display device) presents information to one or more of the user's senses through the human perceptual system, which are mainly focused on stimulating the visual, auditory, or haptic (i.e., force and touch) senses. These output devices require a computer to generate the information through techniques such as rendering, modeling, and sampling. The information is then translated into a form that human can percept by these devices. Therefore, displays actually consist of the physical devices and the computer systems used in generating the content presented by physical devices. What we focus here is on display devices, especially in visual displays.

Obviously, visual displays are the most common display devices used in 3D UIs.

For these devices, real-time computer graphics rendering techniques are used to produce the images that display device presents to the user. When describing visual display devices, a number of important characteristics must be considered [31]:

- *Field of view (FOV) and field of regard (FOR).* The term, field of view, refers to the maximum number of degrees of visual angle that can be seen instantaneously on a display. For example, with a large, flat projection screen, the horizontal FOV might be 80 or 120 degrees depending on the user's position relative to the screen. A device's FOV is often less than or equal to the maximum FOV of the human visual system (approximately 200 degrees) and will be lower if additional optics such as stereo glasses are used. The problem of FOV in 2D display has been mentioned in section 2.1.2. The field of regard of a visual display device refers to the amount of the physical space surrounding the user in which visual images are displayed. For example, if the user stands in a cylindrical display, the display has a 360-degree FOR horizontally.
- *Spatial resolution.* The spatial resolution is related to pixel size and is considered a measure of visual quality which is often given in dots per inch (dpi). The more pixels displayed on the screen, the higher the resolution. But, the resolution depends on both the number of pixels and the size of the screen. For example, two visual display devices with the same number of pixels, the smaller screen will

have higher resolution than the larger one because of the smaller size of pixel.

- *Screen geometry.* Visual display screens come in a variety of different shapes, including rectangular, hemispherical, and hybrids. Nonstandard projection algorithm, which can affect visual quality, is required in nonrectangular screen.
- *Light transfer mechanism.* There are various ways to transfer the light, such as front projection, rear projection, laser light directly onto the retina, and the use of special optics.
- *Refresh rate.* This refers to the speed with which a visual display device refreshes the displayed image from the frame buffer and is often reported in hertz (Hz). Low refresh rates (e.g., below 50-60 Hz) may cause flickering image depending on the sensitivity of a particular user's visual system.
- *Ergonomics.* When interacting with 3D applications, we want the user to be as comfortable as possible, especially when the display needs to be worn on a user's head.

Other characteristics include brightness, color contrast, and gamma correction.

There have been many different types of visual displays used in 3D UIs, which includes monitors, surround-screen displays [41], workbenches [42], hemispherical displays, head-mounted displays, arm-mounted displays [43], virtual retinal displays, and auto-stereoscopic displays. Some of these devices are stationary and do not move

with the user, while some are worn on the user's body [31].

One of the most common head-coupled display devices used in virtual environment application is the head-mounted display (HMD). As shown in Figure 2-7, an HMD is a sophisticated piece of equipment because of its complex integration of electronic, optical, mechanical, and audio components. Regardless of internal design, the main goal of an HMD is to display images directly in front of the user's eyes using one screen for monoscopic viewing or using two screens for stereoscopic viewing [31]. By wearing an HMD that cooperates with a 3-DOF tracker to capture the orientation of our view, we can have a good sense of presence in the virtual environment.

Figure 2-7: I-glasses head-mounted display (http://www.i-glassesstore.com/)

2.3.3. Manipulation in 3D User Interface

Manipulation is one the most popular operation with objects in 3D interaction. "The goal of manipulation interface design is the development of new interaction techniques or the reuse of existing techniques that facilitate high levels of user-manipulation performance and comfort while diminishing the impact from inherited human and hardware limitations [31]." In general, manipulation usually

refers to handling physical objects with one or two hands. But, we narrowly define our manipulation task to spatial rigid object manipulation that preserves the shape of objects. In [31], three canonical manipulation tasks are listed as follows:

- *Selection*, or target acquisition, is the task to acquiring or identifying a particular object from the entire set of objects available. In real world, selection task is picking an object with a hand.
- *Positioning* is the task of changing the 3D position of an object. In real world, position task is moving an object from a staring location to a target location.
- *Rotation* is the task of changing the orientation of an object. In real world, rotation task is rotating an object form a staring orientation to a target orientation.

Besides the canonical tasks, there are some application-specific manipulation tasks.

The choice of devices often restricts which manipulation techniques can be used. Two key characteristics of input devices in manipulation tasks are the *number* of control dimensions (how many DOF the device can control) and the *integration* of control dimensions (how many DOF can be controlled simultaneously with a single movement). For example, trackers allow simultaneously control of both 3D position and orientation. Multiple DOF devices with integrated control of all input dimensions are usually best for 3D manipulation [31].

Pointing techniques is one of the most fundamental classes of 3D manipulation

techniques. The motivation behind the technique is allow the user to select and manipulate objects easily by simply pointing at them. The pointing-based techniques include ray casting [44], two-handed pointing [35], flashlight technique, aperture technique [45], image-plane technique [46], and fishing-reel techniques [34]. Pointing is a powerful selection technique, but is generally a poor positioning technique: object manipulation can be efficiently accomplished only in radial movements around the user (perpendicular to the pointing direction), and when the task does not require changing the distance between the user and objects [31].

Another fundamental class of 3D manipulation technique is *virtual hand technique* (see Figure 2-8) by which the user can select and directly manipulate virtual objects with her or his hand. These techniques include go-go techniques [32] and simple virtual hand technique. Typically, a cursor is used to visualize the current locus of user input. We often use a 3D human hand model in the virtual environment just as an arrow sign in 2D application [31]. The position and orientation of the input device are mapped onto the position and orientation of the virtual hand. These symbols can be some feedback by which users are aware whether the interactions are finished successfully or not.

The techniques mentioned above have the common restriction about the length of user's arm and can not efficiently select objects located outside of the user's reach. An

alternative to extend the length of the user's arm is *Word-in-Miniature technique* [33], which scales the entire world and brings it within the user's reach [31]. Figure 2-9 shows an example of a room model's WIM.

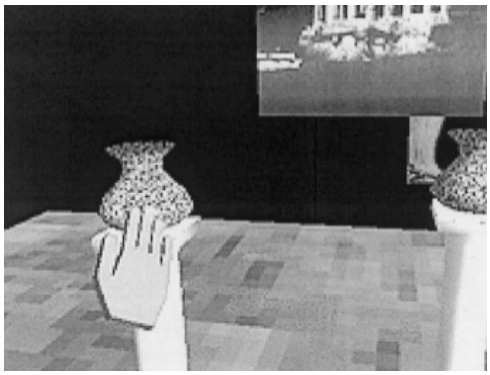

Figure 2-8: Virtual hand [32].

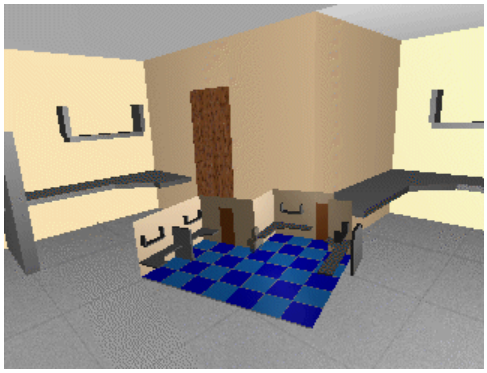

Figure 2-9: WIM technique [33].

It is difficult to design a single best 3D manipulation technique that fits all interaction scenarios. Thus, combining some manipulation techniques has been an active research direction that attempts to bring the good properties of different interaction techniques together. Two approaches can be applied to combine techniques, *aggregation of techniques* which provide user an explicit mechanism, i.e., 3D menu system, for choosing the desired manipulation technique from a limited set of possible options, and *integration of techniques*, in which the interface switches transparently between interaction techniques depending on the current task context. Some of these combining techniques have been developed, such as HOMER [34], scaled-world grab [35], voodoo dolls [36], and etc.