Handyman - A Multiuser Puppeteering System for Computer Graphics Motion Control

by
Alexander G. M. Smith

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of the requirements for the degree of
Master of Computer Science

School of Computer Science

Carleton University
Ottawa, Ontario

August 1, 1991
© copyright
1991, Alexander G. M. Smith

The undersigned recommend to the Faculty of Graduate Studies
and Research acceptance of the thesis

Handyman - A Multiuser Puppeteering System for Computer Graphics Motion Control

submitted by Alexander G. M. Smith, B.M.
in partial fulfilment of the requirements for
the degree of Master of Computer Science

_____

Thesis Supervisor

_____

Director, School of Computer Science

Carleton University

_____

ABSTRACT

This thesis is concerned with the design and implementation of a system called Handyman for controlling the motion of three dimensional objects in a computer graphics environment. Handyman records, combines, and edits the real-time motions specified by human puppeteers operating controls concurrently at multiple workstations. Handyman provides simple computer graphic renderings during recording and more detailed ones for final off-line production.

The thesis describes three areas of Handyman in detail. One area is the configuration wiring diagram that ties together simple physical modelling, multiple live user inputs and pre-recorded motion into a complete performance. Another is the graphics structure that features some extensions to the conventional directed acyclic graph for increased representational power and speedy caching. The last area discusses the low level networking protocol and the networked Smalltalk proxy object system built upon it.

ACKNOWLEDGEMENTS

I d like to thank the following people and organizations for their input to my thesis. In no particular order they are:

Wilf LaLonde and John Pugh	For the WindowMaker user interface editor that made twiddling with window gadget layout much easier.
Sam Adams (via Wilf)	For the original Pluggable Gauges that served as a model for several subsequent rewrites and enhancements.
William Pugh	For inventing Skip Lists (probabilistically optimal and easy to implement - see WADS 89 proceedings). They are used for the sorted collection of times and values in History objects.
Peter Epstein	For many interesting discussions on Smalltalk and for hints on the use of nil superclasses.
My supervisor Wilf LaLonde	For making me rewrite things when they needed rewriting.
The Campus Convenience Store	Being one of the few (or only) places in town to stock Dr. Pepper.
Brian Fultz	For being patient with all the hardware problems I encountered along the way and for telling a good story.
Duong Nguyen	For making me happy that I didn t have to deal with a segmented architecture while implementing networking :-). Also for being a captive audience and for inspiring the icon UI of Handyman s wiring configuration diagrams.
David Buck	For making a portable ray tracer (Mac / UNIX / Amiga).
David Buck and Joe May	For some Smalltalk matrix manipulation code.
Peter Choynowski	For help with graphics programs and mass data storage on the SparcStations.

TABLE OF CONTENTS

Page

Acceptance Sheet ii
Abstract iii
Acknowledgements iv
Table of Contents v
List of Tables vii
List of Figures viii
1. Introduction 1
    1.1. Animation 1
    1.2. Motion Control 2
    1.3. Recording Support 2
    1.4. Playback Support 4
    1.5. Thesis Overview 4
2. Motion Control Techniques 5
    2.1. Key Frame Systems 5
    2.2. Physical Modelling 5
    2.3. Artificial Intelligence 6
    2.4. Puppeteering 6
    2.5. Previous Work 7
        2.5.1. Felix the Cat 7
        2.5.2. Goopy Muppets 7
        2.5.3. Interactive Parametric Models 8
        2.5.4. Virtual Reality 8
    2.6. Summary 9
3. Handyman 11
    3.1. Motion Control Configuration 11
    3.2. Recording a Scene 16
    3.3. Scene and Take Reuse 18
    3.4. Wiring Diagram User Interface 18
    3.5. Computer Network Considerations 19
    3.6. Summary 20

Table of Contents **Page**

4. Graphical Structures 22
   4.1. Simple Collection 23
   4.2. Tree 23
   4.3. Directed Acyclic Graph 25
   4.4. PHIGS 27
   4.5. Handyman Extensions 29
      4.5.1 Wild Cards 31
      4.5.2 Remote Coordinate System References 32
   4.6. Summary 36
5. Caching for Speed 38
   5.1. Cacheable Things 38
   5.2. Moving Objects and Cache Invalidation 41
   5.3. Graphical Structure to Cache Linkage 43
   5.4. Future Work Time Domain Caching 46
   5.5. Future Work Incremental Rendering 47
   5.6. Future Work Rendering Sound 48
   5.7. Summary 49
6. Networked Smalltalk 50
   6.1. Networked Smalltalk Example 51
   6.2. Proxy Objects 53
   6.3. Export Format 55
   6.4. Simple Objects 57
   6.5. Data Transmission 58
   6.6. Packet Protocol 61
   6.7. Summary 63
7. Conclusion 64
Appendix A: Updated State Diagram 68
References 69

LIST OF TABLES

Table	Description	Page
3.1	Some Configuration Diagram Objects	13
4.1	Structures for Comparison	23
4.2	Directory Modification Operations	33
5.1	Standard 3D Graphics Pipeline	39
5.2	Things that can be Cached	39
5.3	Graphical Structure Changes and Cache Invalidation	42
6.1	Garbage Collection Pseudo-code	55
6.2	Export Formats for all Object Classes	56
6.3	Extra AGMS Block Protocol Features	58
6.4	High Level Description of AGMS Block Protocol	59

LIST OF FIGURES

Figure	Description	Page
3.1	Car Example from User and System Viewpoints	12
3.2	Simplest Configuration Wiring Diagram Example	13
3.3	Overly Simple Motion Recording	14
3.4	Playback Reconfiguration Problem	14
3.5	Inserting a History Object for Better Playback and Recording	15
3.6	Car Example Network	15
3.7	3D Graphics Configuration	16
3.8	Director s User Interface	19
4.1	Tree Hierarchy Street Scene	24
4.2	Graph for Six Triangles and Three Cubes	26
4.3	Image Derived from Previous Graph	27
4.4	PHIGS Structures for Drawing a Robot Arm	28
4.5	Equivalent Directed Graph	29
4.6	Problems with Transformations in Edges	30
4.7	Transformations Moved to Nodes	30
4.8	Car Door Paths	31
4.9	Elbow Joint Coordinates Problem	32
4.10	Remote References Change Shape	34
4.11	Volume of Revolution	36
5.1	Tree Equivalent to a Graph	40
5.2	Spurious Change Propagation	43
5.3	Lock / Unlock Protocol	44
5.4	Version Number Protocol	45
6.1	Proxy Example Code	51
6.2	Proxy Object Pointing to a Real Object	52
6.3	Example of Adding the Elements in a Remote Array	53
6.4	Message Redirector and Packer	54
6.5	AGMS Block Protocol State Diagram	60
7.1	Screen Dump of Wiring Diagram User Interface	66
A.1	Updated State Diagram	68

Chapter One

1. INTRODUCTION

This thesis is concerned with the design and implementation of a computer animation motion control system. Existing systems, such as the one used for controlling computer graphic Muppets in *The Jim Henson Hour,* are examined and various techniques for motion specification are compared. This leads to a description of the real-time puppeteering approach taken by Handyman. The caching features and the enhanced hierarchical structure of the underlying graphics rendering system for fast puppeteering are also described. Finally, the networked proxy object Smalltalk system that makes it possible for multiple users to interact in real-time is explained.

1.1. ANIMATION

Animation (simulated motion) is achieved by rapidly displaying **frames** in succession. Each frame is a picture of a scene at a particular instance in time. Subsequent frames show the same scene at slightly later times. If the frames can be displayed to the human eye fast enough, the human perception system will be tricked into seeing the motion implied by the sequence of static pictures. North American television displays frames at the rate of 30 per second while film is usually displayed at 24 frames per second. Rates as low as 10 frames per second are tolerable for motion, with an increase in perceived flicker at the lower rates. If the frame rate is too slow, the illusion of motion breaks down and the viewer notices that he or she is watching a sequence of static images.

Making good quality images is easy to do with any of several readily available ray tracing and other rendering programs, but doing animation is difficult. Most ray tracers and renderers input a scene description and output an image which can be used as one frame in an animation. Each scene description describes the attributes for all objects in a scene, including such information as colour, size and location in three-dimensional (**3D**) space. Creating an animated cartoon involves a repeated cycle of updating the scene description to account for the motion that occurs between the current frame and the next and then rendering the frame. The rendered frames are saved for later viewing at the appropriate frame rate.

1.2. MOTION CONTROL

The essential goal of a motion control system is to provide the updated object attributes needed for drawing each frame in an animated sequence. There are also several additional features that are desirable in a motion control system. A system which quickly previews the animation so that several attempts can be made to achieve the desired motion is preferable. One that lets the user edit an existing motion is useful for fine tuning of motion and opens the door to reuse of old motions from a library of actions. Finally, users appreciate systems that are easy to use yet powerful.

In two dimensions, finding the locations of objects to be rendered can be done by hand. The animator simply plots out the desired path that the object will take on graph paper and then marks off the positions of the object in each frame, using an intuitive approximation to the object s acceleration and velocity. Such a system is easy to understand but difficult to use. Part of the difficulty comes from the drudgery of reading off object coordinates for all moving objects in the scene and typing them into the scene description. The user also has to make the difficult jump from an imagined motion to a line on paper with timing marks. This technique becomes even harder to use for 3D animation. Any automated motion control system would be an improvement, even just using a computer to find the coordinates from a digitized image of the paper graph would be an improvement.

Various motion control systems have been implemented and are in use today. Many use key frames to specify the positions and shapes of objects at key points in time. Another popular technique is to use physical models to derive the object positions over time. Less mechanical approaches, such as puppeteering, record gestures made by the animator. This technique has been recently used to produce animations for *Felix the Cat: The Movie* and *The Jim Henson Hour.* Virtual reality simulations are an extreme case where the animator becomes the puppet being animated. Some methods actually use a mixture of techniques. For example, a virtual reality may record the animator s motion and then modify the motion with a physical model that simulates inertia and gravity.

1.3. RECORDING SUPPORT

Handyman uses real-time puppeteering as its basic model of motion control. As an optional feature, Handyman also supports simultaneous performances by multiple animators. For example, one animator could be controlling a bouncing ball while the other animator chases

after it with a dog puppet. This leads into the use of parallel processors for speed and multiuser support. A configuration wiring diagram helps the user by making it easy to manage the connections between the controls, processing elements, and animated objects.

On the input side, the actions of the animator(s) have to be sampled at a sufficiently high rate for smooth motion. Handyman uses a decoupled approach where inputs are sampled at a rate different from the rate at which they are used. The results of the sampling are stored in a data structure that uses interpolation to serve as a buffer between the different rates. For example, a joystick control may be sampled 20 times a second while the display of the scene using the joystick data is only updated 8 times a second. After the recording has been done, the scene can be rendered using the extra sampling information for added smoothness in the final production.

A configuration wiring diagram stores the configuration of connections between the various decoupled elements. Besides specifying where the data flows, the diagram can be used to insert processing elements, such as a simulation of inertia, between the controls and the animated object. A graphical display of the wiring diagram is easily modified by the user to connect controls to animated objects and to keep track of different scene setups.

The network of processors provided to support multiple animators fits in quite well with the decoupled processing technique. One benefit of the technique is that one processor can be used for processing at the high sampling and storage rate while another processor can be used for display regeneration. Communication between them is minimal just the passage of information at the slower sampling rate of the consumer processor. The usual benefits of increased CPU power and extra memory over a single processor are also appreciated.

The network interface between processors is hidden from the programmer under extensions to Smalltalk (Handyman s implementation language) that make network operations transparent. More specifically, a class for proxy objects has been added to Smalltalk. When the programmer queries a proxy object for information (such as a control setting at a given time), the proxy transparently sends a message over the network to the real object it represents. The real object evaluates the query and sends back the answer. As far as the programmer is concerned, the proxy object is almost the same as the real object .

A custom network protocol implemented on top of the AppleTalk network supports the proxy object system. The protocol has to be fast for real-time performance during

recording. It also supports some useful operations such as the broadcasting of large blocks of data to multiple receivers with error recovery and minimal overhead.

1.4. PLAYBACK SUPPORT

During recording, the animators have to have feedback on what their puppets are doing. This graphical feedback has to be relatively quick to avoid time lag problems. By taking advantage of stationary objects, which only need to be drawn once, the graphics system can draw complex scenes relatively quickly.

In addition to a hierarchical structure that permits the easy detection and caching of non-moving objects, the graphics system has a few innovative extensions to the conventional acyclic directed graph hierarchy. These extensions and some other features are compared with existing graphical data structures, including PHIGS [ANSI88].

The final result of the graphics rendering subsystem is currently a series of wire frame drawings. For post production, the same graphics structure is used to generate object definitions for a ray tracer that produces the final, almost photographic quality, images.

1.5. THESIS OVERVIEW

The rest of this thesis briefly reviews other motion control systems (chapter two) and then goes on to describe the implementation of Handyman. The implementation description is divided into several parts, starting at the top level and going down the supporting infrastructure. Chapter three describes the motion control aspects of Handyman, chapters four and five cover the graphics system and chapter six provides details about the networked Smalltalk system. The final chapter reviews the main features of Handyman, and what was learned by implementing it.

Chapter Two

2. MOTION CONTROL TECHNIQUES

There are several different ways of specifying motion. They range in complexity from simple key frame systems to dynamic models, artificial intelligence, and complex combinations of techniques. They all share the common goal of motion control: generating a sequence of object attributes for the frames in an animated sequence.

2.1. KEY FRAME SYSTEMS

Key framing is one common way of specifying motion. With this technique, the positions of the moving objects are specified by the animator at important times. For example, the animator may specify the frames for the animated character crouching down to prepare for a jump, then for the character in mid-air and finally for the character absorbing the shock of landing on the ground. In this case the object attributes include the shape of the character as well as its location. The motion control system then computes the object attributes for additional frames between the key frames. There are enough additional frames for smooth motion and shape change from one key frame to the next.

Key frame systems often use splines to calculate the intermediate poses by using the key positions as control points. One difficulty with key frame systems is that inappropriate or insufficient key frames result in motion that isn t desired. For example, if not enough key frames are specified, a swinging leg may end up moving in a straight line rather than an arc. The animator then has to go back and add additional key frames to force the motion into the desired path.

2.2. PHYSICAL MODELLING

Physical modelling techniques simulate the animated objects mass, geometry and applied forces. Gravity, friction and collision effects also can be simulated with quite realistic results [Wilhelms88].

Moving the object is achieved by applying simulated forces to joints and other parts. Often some sort of controller drives the physical model, perhaps generating the forces required for a walking motion [Girard87]. Another method is to use optimization

techniques to find the forces that require the least amount of muscular energy to get from a starting position to a final one [Witkin88].

Physical model controllers are adequate for controlling routine types of motion such as walking but lack generality and are difficult to program. The user still has to direct the overall actions of puppet. Physical modelling also doesn t directly handle emotional expressions input from the animator is still needed. Panne et al [Panne90] suggest a compromise with state-space controllers that each do some simple action (such as jumping towards a target) and that can be chained together for more complex composite actions.

2.3. ARTIFICIAL INTELLIGENCE

Artificial Intelligence is another tool for motion control. One approach is to replace the animator with a controlling rule-based program that moves the character. Another is to create simulated reflex actions (via neural networks) that can be triggered by the animator. However, the goal of providing a character with the ability to do any action, including ones not foreseen by the system developer, is difficult to achieve with AI even more so when done under real-time constraints.

2.4. PUPPETEERING

Puppeteering, the motion control method used in this thesis, is simpler than most of the other techniques and yields good results much more quickly. The motion history needed for rendering is obtained by recording a performance done live by human puppeteers. The recorded motion yields closely spaced key frames that can be interpolated and rendered in a manner similar to the conventional key frame approach. The difference is that the motions recorded in a live performance are intuitively specified rather than being designed by hand.

The higher data rate between the puppeteer and the animation system during a live performance allows easier expression of the subtle motions that give life to a puppet. Multiple performances of the same scene by the puppeteer (called **takes**) allow the choice of a best performance for the final rendering. Editing the recorded motions also can be done to perfect a motion if it is too difficult to do live. The functionality of regular key framing is also available since the puppeteering system essentially stores its motions as very closely spaced key frames.

2.5. PREVIOUS WORK

Recently several animation production houses have turned to puppeteering as a way of quickly specifying motion for computer graphics. They seem to find it a faster and cheaper alternative to key frame and physical modelling approaches.

2.5.1. FELIX THE CAT

deGraf / Wahrman Inc. used real-time puppeteering to animate the head of Felix in *Felix the Cat: The Movie* [S¿renson89]. A polygon based model of the head was created with moveable eyes, mouth and other parts. The motion was too complex to do in one pass so they did the overall head motion, eye motion and mouth in separate passes. Some of their previous work included a live performance with a Human face puppet that required 8 parameters controlled by two puppeteers with joysticks and similar control devices.

S¿renson cites several advantages to puppeteering over conventional computer animation. It gives fluidity of expression and spontaneous improvisation that was previously unheard of. And yet, adjustments can still be made on a frame-by-frame basis. He also mentions an improved production time of under a week for 90 seconds of lip-sync animation. He also points out that puppeteering allows people with body movement skills to make use of their natural motion control abilities.

2.5.2. GOOPY MUPPETS

Pacific Data Images uses puppeteering techniques to animate Waldo C. Graphic for *The Jim Henson Hour* [Schreiber89] and for various other characters. The process starts with an armature glove that is similar to the one used for radio controlled Muppets. The glove mimics the inside of a more conventional hand puppet, with the angle between fingers and thumb controlling the mouth opening of the animated character. A linkage attached to the glove measures the glove s position and orientation. The position information goes to a computer which draws a rough version of the character. A camera positioned at the audience s view-point simultaneously records the live Muppets. The mixed real and computer graphic images are then used by both the real Muppeteers and the computer graphic Muppeteer to perform the scene. After the motion has been recorded, the computer graphic frames are rendered in higher quality with costume changes and goop. Goop is a kind of physical model of inertia that varies in strength for the various body parts of the

animated character. The end result is that the body of the character sloshes around the position defined by the armature glove in a realistic and exaggerated way.

Several advantages to puppeteering are related by Schreiber. As before, puppeteering is a more productive way of doing computer animated characters. He says that they achieved a good level of productivity of over a minute of animation per week or two minutes with two people. Compared to two months for a 30 second TV logo, this is impressive. Another benefit is that the computer generated character rehearses with the live actors so that problems with the scene can be quickly worked out. The live interaction also makes possible the all important muppet eye contact between the live muppets and the computer generated muppet.

2.5.3. INTERACTIVE PARAMETRIC MODELS

Pat Hanrahan and David Sturman describe [Hanrahan85] an earlier puppeteering system called EM. It combines live manipulation of a parametric model with prestored position databases. Objects are described in a block oriented language where each block represents a coordinate system. Statements within a block move the block s contents or draw primitive objects. Nested blocks implement joints in the model. The arguments to all blocks are specified by a database. By changing the database or overriding parts with another database, the entire model s block arguments are modified resulting in movement. Live controls can be attached to objects by specifying the control as one of the arguments to a block.

The authors were happy with the flexibility of their control configuration and block language. However, they did regret not knowing the past or future of a time varying parameter. Knowledge about parameter values over time is useful for doing physical dynamics. Their implementation also required a fixed size display list, making it impossible to dynamically change the topology of the model. Dynamic topological changes are useful for such actions as tearing off a puppet s arms.

2.5.4. VIRTUAL REALITY

Virtual reality is strongly related to puppeteering. In a virtual reality environment, the human is a participant in a simulated world. Typically he wears a helmet or goggles that provide stereoscopic vision into the simulated world. By completely replacing the normal field of view, the participant believes that he or she is really in the simulated world. The

position and motions of the participant are measured by various devices and reflected in the actions of the simulated double of the participant. If the real human turns his head, the simulated double does too and the image shown to the participant changes to reflect the new view as seen from the double s simulated eyes. Other motions such as hand position and gesture can also be measured and used in the simulation, perhaps to grab and manipulate simulated objects. Other flights of fantasy are possible, flying like a bird by pretending that arms are wings is one obvious virtual reality.

Unfortunately, many current virtual reality systems have a slow update rate or low quality graphics. The faster systems use expensive computer hardware to achieve speed and quality. One example of a limited kind of virtual reality is described in [Iwata90]. His system uses a hand controller that lets the user manipulate things in the virtual world. Besides the normal position sensing, force feedback applies pressure to the user s fingers to give a sense of touch. His system operates at an update rate of 4 Hz (4 new drawings of the scene per second) with an eventual goal of 10 Hz.

If virtual reality systems recorded the motions of the participants then the results would be useful for controlling a computer generated character. Even without editing capabilities, virtual reality is useful for live computer generated character performances. The human actors essentially wear computer generated costumes and can do simulated actions that would not otherwise be possible (like flying or changing shape). The problems with slow updates can be overcome in post-production by using interpolation to generate images at a higher rate than was recorded.

2.6. SUMMARY

Several techniques for motion control were described and commented on. Key frame motion control is a popular technique that is relatively common because of its implementation simplicity and ease of use. It suffers from the mental overhead of designing motion by specifying static frames. Physical modelling is an often described technique which is good for simulating real motion but can lack in spontaneity and exaggeration unless they are designed in (the Goop used for computer graphic Muppets is exaggerated physical motion). Physical models only meet half of the requirements for animation a controller of some sort is needed to drive the model. Various approaches such as puppeteering, general state-space controllers, or equation solving to find minimal muscular

energy are required. Real-time puppeteering has the advantages of spontaneity and exaggeration along with higher productivity.

Handyman s puppeteering system draws some inspiration from the earlier EM system s flexible control configuration. The desire of EM s designers for values with a history and puppets that can be dynamically restructured also influenced Handyman s design.

Two existing puppeteering systems were discussed. The animated head of Felix the Cat is an example of real-time puppeteering mixed with pre-recorded motions. The computer generated Waldo C. Graphic illustrates how hand puppet style user interface hardware can be combined with physical modelling to animate a simulated Muppet. Users of both systems claimed improved productivity over conventional computer animation and also claimed to be able to generate motion that is more appropriate for cartoon characters.

Finally, virtual reality systems are seen to be the puppeteering system of the future. They combine live performances by an actor with computer generated costumes and scenes. The result is easily generated high quality motion in a cartoon world. Unfortunately, the supporting hardware and software for doing virtual reality is currently immature.

Chapter Three

3. HANDYMAN

The Handyman motion control system uses the puppeteering approach to motion control. There can be several animators, each one at a separate computer workstation (a Macintosh) controlling one or more objects in the scene. For example, one animator may control the overall forward motion of a running dog s body while another animator has control over the head attached to the body. This division of duty makes it easier to have the dog show alertness by turning its head to look at objects while it is running.

The animators actions are enhanced by an optional physical simulation and other processing elements, recorded and then used to drive the objects being animated. The complete scene is shown in rough form on a central display visible to all animators. If there aren t enough animators to control everything, prerecorded motions can be used for some objects while others are controlled live.

After enough takes have been done to perfect a scene, the final take can be used to produce a higher quality animated sequence. The motions for the final rendering use values interpolated from the live performance to provide smooth motion and to compensate for differences between the final frame display rate and the rate at which the actions were recorded. The final images are drawn using DKB Trace [Buck90], a ray tracer that produces almost photographic quality images for each frame. After being rendered, the images are stored on a large hard disk, mixed with sound, and later played back at the appropriate frame rate direct to video tape.

3.1. MOTION CONTROL CONFIGURATION

Handyman uses a **configuration wiring diagram** to flexibly connect controls to the objects being animated. The user interface for the diagram displays a network of objects, some representing the controls that the animator uses and others representing the objects being animated. The user connects a control to an object to be animated by simply drawing a line between icons representing the desired objects. The wiring diagram can be thought of as a variation of a data flow network, with data flowing from controls, through processing elements, and to animated objects. Note that it isn t a conventional data flow network because there are dissimilar rates of data flow in and out of objects. In addition to mere

connectivity, the wiring diagram can also include motion recording / playback elements between the controls and the animated objects.

Handyman s configuration wiring diagram mediates between the user s view of an animated object (a car for example) and the graphics system s version of the animated object. As far as the graphics system is concerned, a simulated car has a position in space, a shape and a direction that it is pointing in. The animator / performer usually doesn t want to control the car by specifying absolute position and heading. Instead, he would prefer to see controls appropriate to a car: a brake pedal, gas pedal and a steering wheel.

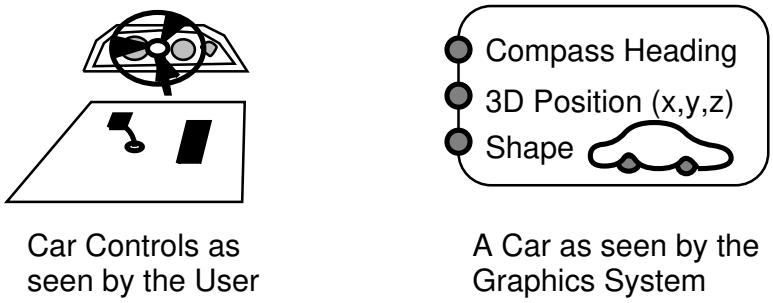


Figure 3.1 Car Example from User and System Viewpoints

Handyman converts the user s control movements to values suitable for the graphics system. These values can also be made more realistic by using simple physical modelling techniques to convert the gas pedal motion into acceleration and velocity along with inertia and gravitational effects. If the user wants to elaborate, acceleration above a certain value could be configured to make the car flip up on its rear wheels. In the future, sound could also be integrated into the wiring diagram, perhaps by keying noise level and pitch to simulated engine speed. Tire skidding noises could also be added when acceleration goes over an appropriate value.

Rather than explicit programming to link the controls to the graphics, the configuration wiring diagram uses motion manipulation objects. These objects can do several simple and not so simple operations including the ones listed in TABLE 3.1.

¥ Copying and modifying values	¥ Reading control devices
¥ Simulating simple physics	¥ Storing and replaying
¥ Manipulating the 3D graphics	

Table 3.1 Some Configuration Diagram Objects

The motion manipulation objects come in three basic styles: active objects, passthru objects and passive ones. During a recording session, the **active objects** each have a running process that repeatedly reads any inputs and writes any outputs, usually at some predefined sampling rate (10 samples per second or better). The **passive objects** return values in response to queries from active objects reading them and store or set values from active blocks writing to them. **Passthru objects** simply alter the values passing through themselves and are placed between active and passive objects. The items read and written are all paired with a time value that helps compensate for various system propagation delays.

The simplest case, shown in FIGURE 3.2, is one with three objects: the control device (passive), a **copier** (active) and a puppet that responds to input (passive). The copier s input is connected to the control device and the output is attached to the puppet. When a take is being performed, the copier s process becomes active and periodically reads a value from its input (the state of the control device) and writes the same value to its output (some attribute of the puppet, such as location). The result is a puppet that moves in response to changes in the control device.

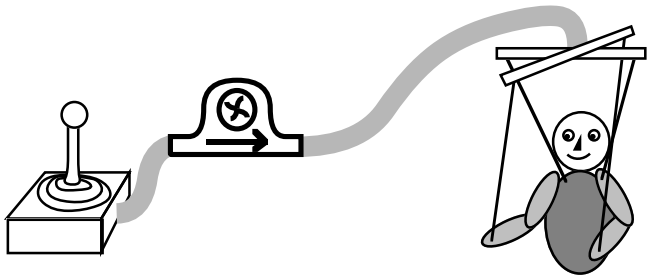


Figure 3.2 Simplest Configuration Wiring Diagram Example

The simple copier setup isn t too useful if you want to replay the take since none of the control values or graphic motions are remembered by the system. To remember the past, a passive **history** object is used. A history object remembers the data written to it and can be

queried to find out what things were like in the past. Internally, the history object stores the values written to it indexed by time. When queried for the value at a particular time, the history object interpolates between known values to get the value for the desired time. The fine turning of motion done in key frame systems can be done by editing the history s stored values.

There are two obvious ways that the history object could be added to the motion processing wiring diagram. FIGURE 3.3 illustrates a simple attempt where the history object records the motions that are also being written to the puppet.

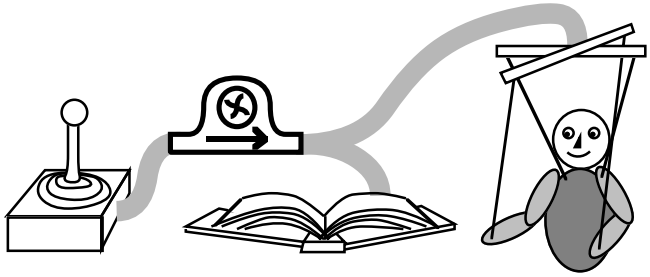


Figure 3.3 Overly Simple Motion Recording

This method has the advantage that the history object has an exact copy of the data points written to the puppet. The disadvantage is that it is hard to use the history object to replay the motion. For any replays, the wiring diagram has to be reconfigured to copy the history to the puppet. FIGURE 3.4 shows the reconfigurations needed.

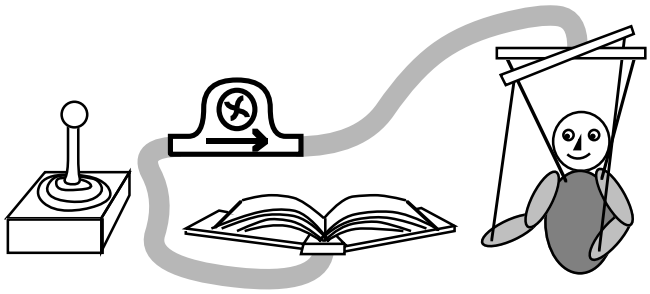


Figure 3.4 Playback Reconfiguration Problem

Rather than reconfiguring, which becomes unmanageable with active objects that modify data, a better technique is to provide multiple data flow stages. FIGURE 3.5 shows how a history object can be inserted into the flow between the control device and the puppet. An additional copier is also needed. During recording both copiers are running and

the history object is being simultaneously written to and read from. During playback only the second copier is running, simply reading from the history object s contents. This configuration also makes use of the history s interpolation feature by moving the puppet using extrapolated data if information from the control is delayed.

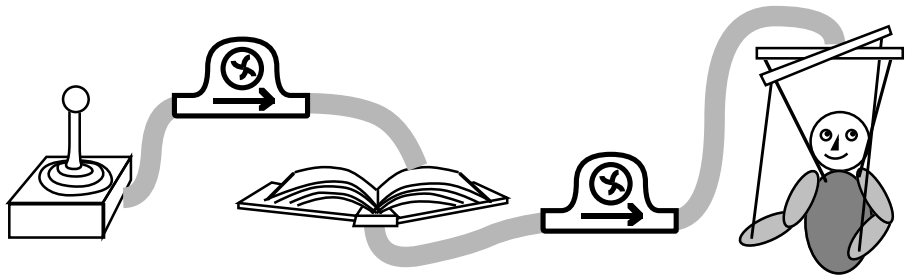


Figure 3.5 Inserting a History Object for Better Playback and Recording

FIGURE 3.6 shows what the wiring diagram for the car example would be with a physical simulator active object that takes acceleration, deceleration and steering inputs and outputs position and orientation. Two history objects are then used to record the position and orientation. If the graphics structure is made into an active object rather than a passive one, then a pair of copiers between the histories and graphics isn t needed.

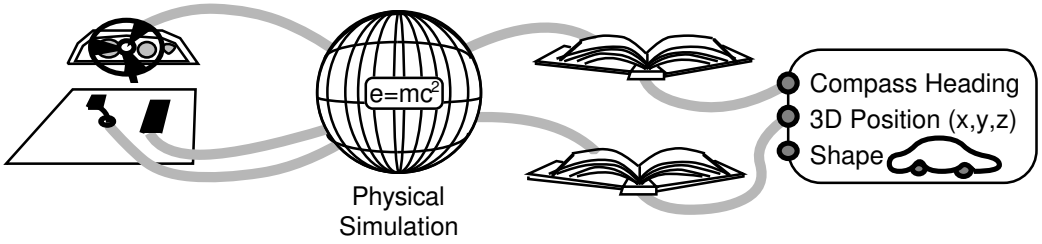


Figure 3.6 Car Example Network

In actual practice, the setup for 3D graphics (FIGURE 3.7) is a bit more complex. An active object represents the rendering engine which draws the scene, passthru objects move the graphics and the usual controls interface between the user and the rest of the system.

The rendering engine uses an hidden reference to an external geometrical graphical data structure (described in chapter 4) along with the location of a camera to define the image being drawn. The engine outputs either bitmap images or ray tracer scene descriptions which can then be fed into a suitable passive object for display or off-line tracing. A history object can be inserted in the output stream to record the bitmap images for fast playback.

The remaining input connections provide timing synchronization to the attached passthru objects.

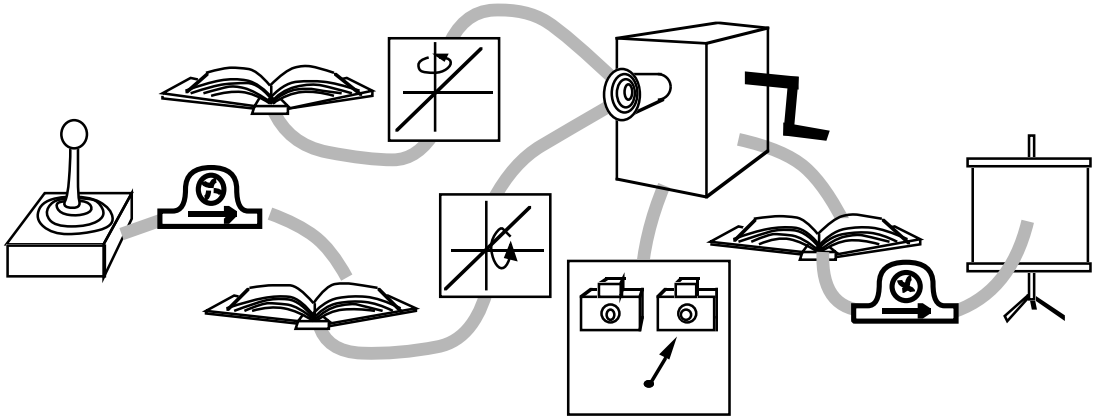


Figure 3.7 3D Graphics Configuration

The attached manipulator passthru objects interface between the motion control system and the geometrical graphical data structure. When a manipulator is read by the rendering engine, it reads its inputs and uses the result to tweak part of the graphics structure. Usually this means modifying a transformation matrix to rotate or move part of the graphics. However, other changes are possible, including changes to a puppet s topology by changing the parents of a graphical object. For example, an arm can be ripped off a puppet by removing the shoulder from the arm s list of parents and adding the arm to the general world as an independent object. Finally, the manipulator signals the rendering engine that it has finished by returning a value.

This system of a rendering engine with separate manipulator objects makes it easy to keep track of what controls affect what part of the graphics. The use of a read operation has the beneficial side effect of synchronizing the modifications to the graphics to the common time stamp in the read requests.

3.2. RECORDING A SCENE

A special director object has overall control of the recording session. It coordinates the various active objects that perform a particular scene, finding and hiring them and then doing the shooting of several takes. The director can also be used to save work in progress for later shoots. When a scene has been finished, the director shuts down things and fires

the active objects, making them available for employment by another director in another scene.

The director initially locates the actors (active objects) that are in the scene. This is done by asking all possible actors if they are in the scene (active objects keep track of which scenes they know). If an actor is already employed by some other director for some other scene then the director can t shoot the scene. This metaphor serves as a simple arbitration scheme between directors.

As part of the employment procedure, the director synchronizes the clocks of the active objects with its clock, accounting for propagation delays across the computer network. Once all the active objects are signed up, the director can draw the wiring diagram for the selected scene, filling in the passive objects used by the active objects for that particular scene.

The actual recording of a take is relatively simple. The director merely notifies all active objects that recording will start at a particular point of time in the future. This notification creates a recording process at each active object. A typical recording process starts by doing some initialization and then goes to sleep, using the presynchronized clock to wait for the starting time to arrive. After it wakes up, the recording process does one time step, which involves sampling the active object s inputs and writing to its outputs. After the recording step has been done, the recording process goes back to sleep until it is time for the next step. Finally, the end of the overall recording time period is reached (or the user stops it manually) and the recording process shuts down.

A special non-real-time recording mode is also available for producing the final images. Rather than running in parallel in real-time, each active object is processed in a user defined sequence, taking a series of fixed size steps through a specified time range. This makes use of the otherwise unused high resolution data stored in the history objects and results in smoother motion than the real-time method.

Unlike a conventional data flow network, Handyman s active objects are loosely coupled. One active object could be sampling a control and writing to a history 20 times a second while another one could be reading the same history 8 times a second. Besides using a separate process for each active object, a further level of parallelism is used to read and write data. When reading its inputs, the active object issues read messages in parallel and waits for all of them to complete before proceeding. When writing to outputs, the

active object sends all write messages and doesn t wait for any replies. A time stamp in all messages solves the problems associated with network delays and nonsequential requests.

3.3. SCENE AND TAKE REUSE

After shooting a take, the animator will often decide that it wasn t quite right and that it needs to be reshot. Handyman makes this easier by creating a new wiring configuration diagram for the new take based on the previous take.

Consider the simple example in FIGURE 3.5 (Inserting a History Object for Better Playback and Recording). The animator probably wants to keep the same input controls, copier objects and puppet graphics. The only object that needs to be different for the new take is the history object in the middle. Handyman thus creates the new take s configuration with the same arrangement of objects as the previous one and with new histories in place of the old ones. If the user desires, the automatic creation of new history objects can be turned off for particular histories, usually ones that the animator thinks are perfect.

The animator can also change the wiring diagram in other ways, adding new objects and breaking old connections. The changes will affect only the take that the animator is editing and will be used in new takes based on the edited take.

3.4. WIRING DIAGRAM USER INTERFACE

FIGURE 3.8 shows the main part of the user interface for a director (other parts pop up when the user clicks on the objects or background). Some things to note are that the active objects are square, with inputs on the left and outputs on the right. Passthru objects are shown as small rectangles that have an attachment for active objects on the top and passive clients on the bottom. Passive objects are large rectangles with an in / out connection on the top. The dots along the sides indicate the number of dimensions of the data being handled, in this example everything is 2D. Finally, the names of the objects show object types in this example; normally a history would be called something like Car Location rather than Linear.

Passive objects also have a shadowed right hand side when they are to be replaced by new similar passive objects in a new take, the shadows suggesting a stack of objects rather than just one. A shadow on the bottom edge of a passive object means that it is writeable,

no shadow means that it is read-only. The passive objects also have the scene and take of their creation as part of their name.

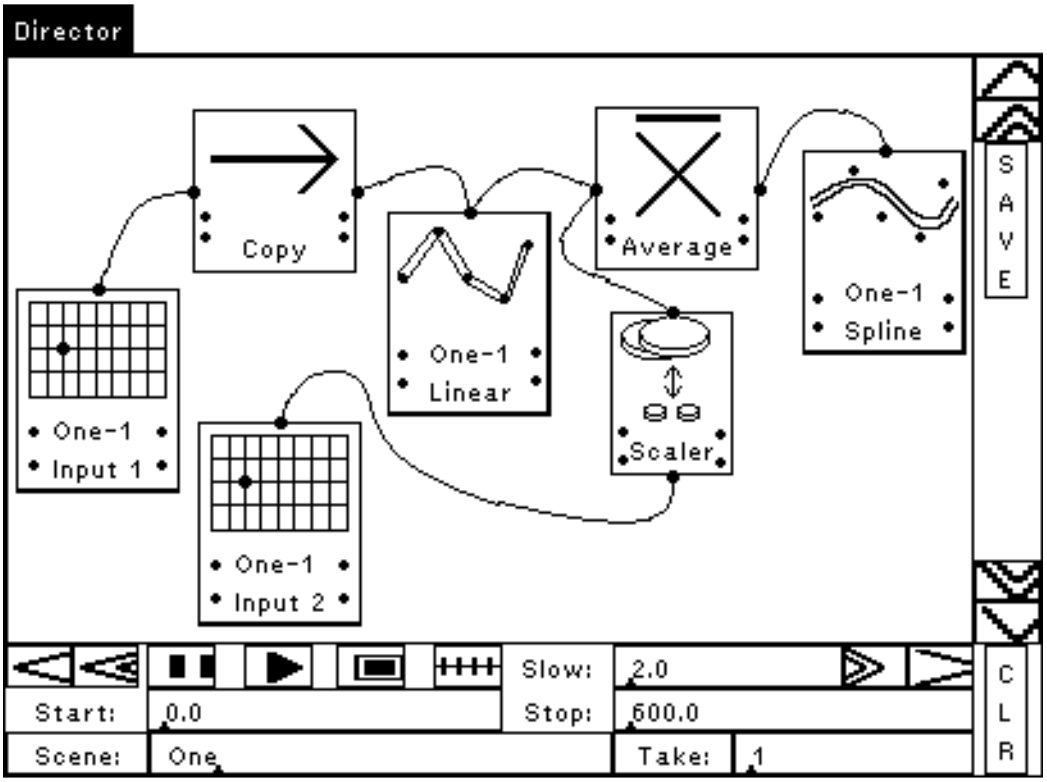


Figure 3.8 Director s User Interface

3.5. COMPUTER NETWORK CONSIDERATIONS

Handyman s use of a distributed approach makes some operations more difficult than usual. Locating and listing objects on multiple processors is more difficult than on a single processor. Object redistribution while loading a saved configuration is another problem created by having multiple processors. A way of finding objects by name goes a long way towards solving these problems.

A directory of all Handyman objects is stored on every computer. When the director is looking for unemployed actors for a scene, it consults the directory. In practice, a dictionary that associates a unique name with every Handyman object is used as the directory. Whenever an object is created, its name is broadcast to all computers. Each receiving computer then adds the new name and object to the local copy of the directory. Similarly, name changes and object deletions are broadcast.

Loading the objects from a previously saved set of scenes is complicated by the need for some objects to reference other objects. To make things easier, loading is done in two phases. The first phase recreates the objects at their target computers by remotely evaluating strings of Smalltalk code. If the desired target computer doesn t exist then the object is created on some other computer. As part of the object recreation phase, the objects names are registered globally. In phase two, references to other objects (connections in the wiring configuration diagram) are resolved. The references are converted from a string naming the object to the object itself (if it is on the same computer) or a proxy (if it is remote).

Storage of objects is done by making the desired objects dump themselves onto a stream. The dumping consists of writing the code needed to recreate the dumped object followed by more code to assign values to the object s internal variables. References to other objects are saved as the name of the referenced object. Later on, the referenced objects are dumped too.

The director s user interface is also complicated by the network. Some operations, such as drawing icons for the objects, have to be done locally for speed reasons. The result is code that isn t written in a pure object oriented style. Where necessary, stand-in objects are used between the user interface and the remote Handyman objects. These stand-in objects do simple things like converting the name (a string) of the real object into a bitmap or other display object that doesn t travel too well over the network. The stand-in objects also solve the problem of opening windows on the wrong screen. This problem occurs because asking an object to open a window will open a window on the object s computer s screen, not the screen where the user requesting the information is located.

3.6. SUMMARY

Handyman uses a configuration wiring diagram to connect user controls, processing elements, storage objects and computer graphics. Multiple takes and variations of a scene are easy to do with a facility that uses previous wiring diagrams as a template for new ones.

The concept of active objects with their own processes makes it easy to take advantage of parallel processors. In particular, parallelism is used to get high resolution data from multiple users operating multiple controls in real-time. History objects, which store time indexed values and return interpolated values based on time, are essential for interfacing between the parallel processes of active objects operating at different rates.

Handyman uses special manipulator objects to interface to a mostly external 3D graphics system. They work by using values read from their inputs to move the graphics on the command of a rendering engine object. When the manipulators have finished adjusting the graphics, the renderer does its work and outputs a picture.

A director is used to control all the objects in the wiring diagram. It tracks the objects in a scene, synchronizes clocks over the communications network and controls recording. The director also plays a part in distributing objects to their home computers when it loads a saved scene. Saving and reloading configuration networks with the associated history data works well though is slow due to some size limits in Smalltalk.

As a whole, I feel that Handyman s control system is a joy to use. Changes to do a new take are effortless and scene setup is relatively easy. Remote objects are as easy to control as local ones, though some care must be taken to avoid making a high data rate connection between different computers.

The major deficiencies at the moment are a lack of good 3D input devices and a slow graphics system. Despite that, Handyman is a lot better than typing in coordinates when making a computer animated cartoon.

Chapter Four

4. GRAPHICAL STRUCTURES

A Handyman puppet is made from several different kinds of graphical objects. The visible flesh is made of geometric surfaces and solids. Under the flesh lies a skeleton or armature made from coordinate system transformations. When the user animates the puppet by manipulating this skeleton, the attached flesh and subsidiary skeletal parts naturally follow.

This chapter explains the Handyman puppet graphical structure by giving an introduction to coordinate systems followed by an examination of several different types of graphical structures, leading up to Handyman s puppet structure. The implementation of the graphical structure used in Handyman is described in the next chapter.

A **graphical structure** is a representation of a complex shape that the user is trying to manipulate. Typically the structure will be composed from various different **graphic primitives** organized in some fashion. Primitive objects such as points, lines, polygons, polyhedra, volumes defined by quadratic surfaces and so on, can be combined to make up the desired shape.

The structure can also contain non-primitive user defined objects such as grouping objects. These make it possible to treat a collection of primitive objects just like another primitive. For example, a collection of cubes and cylinders may make up an arm shape. Putting this collection into a group allows the easy use of an arm object anywhere in the graphical structure.

Other non-graphical objects such as cameras are convenient. A camera could be composed of several primitives: a point for representing the camera s location, another point that the camera is looking towards and perhaps a third point to specify which direction is upwards for that camera.

All primitive objects in a graphics structure need to be drawn relative to some coordinate system. A **coordinate system** in 3D consists of perpendicular X, Y and Z axes going through a common origin point. Objects in the structure are located in space by reading off their coordinates from the three axes of the coordinate system. Conversely, objects can be drawn in a coordinate system by looking up the object s preset coordinate values on the axes and then drawing the object at that location.

If the coordinate system is changed and the objects are redrawn with the same coordinate values as they had before the change, the objects will appear to move with the coordinate system. For example, compressing the scale used on the axes will make all the objects appear to shrink. The axes can also be translated and rotated as well as being scaled. Of course, this assumes that the observer is outside the coordinate system being changed. If the observer was inside then it will be stretched, rotated and moved just like the objects being observed, resulting in no observable relative change.

A graphical structure is concerned with the arrangement of objects and their coordinate systems. TABLE 4.1 enumerates the different types of structure that will be described. They range from the simple flat collection of primitive objects, through various graph structures to the structure used for Handyman. The simpler ones are generally easier to implement and are actually more cumbersome to use than the complex structures.

¥ Simple One Level Collection	¥ PHIGS Hierarchy
¥ Tree Hierarchy	¥ Handyman s Graph Extensions
¥ Directed Acyclic Graph	

Table 4.1 Structures for Comparison

4.1. SIMPLE COLLECTION

Sometimes the structure is just a simple collection of primitive graphic objects where everything is drawn in the same coordinate system. This minimal structure is quite easy to implement and understand. However, it is difficult to group things together. In the previous arm example, all the parts in the group representing the arm would have to be individually adjusted when the whole arm is moved.

4.2. TREE

A popular graphic structure is based on the hierarchical tree. It trades off some extra complexity for easier grouping of objects. By inheriting a coordinate system from their parent, the children automatically follow their parent when the parent is moved.

The inherited coordinate system is the same system used by the parent or is a modified copy based on the parent s coordinate system (for example, one that is like the parent s but shifted down and to the right a bit). All children are positioned using this inherited

coordinate system. When the parent moves, each child s coordinate system moves in the same way. The result is that children move around with their parents without having to adjust the values of their location or orientation. It s all done by moving the coordinate system, not by touching the coordinate values. If the modifications to the parent coordinate system that generate the children s coordinate system are changed, the children move relative to the parent while still following the parent s overall motion.

One tree style structure is described in [Foley82]. The edges of the tree contain an optional transformation between the parent and child coordinate systems. The leaf nodes are the objects to be drawn and interior nodes represent groups. FIGURE 4.1 shows an example of such a tree. The edges with transformations (modified parent coordinate system inherited) are labelled with the transformation name. Transformation A can be modified to make the car move down the street. When the car is moved, both the car body and the door move. Transformation C can be used for rotating the car door about its hinges, simulating opening and closing the door. Similarly B is useful for moving the whole tricycle and D is for turning the front wheel of the tricycle. The objects in thick outline are the primitive ones that actually get drawn.

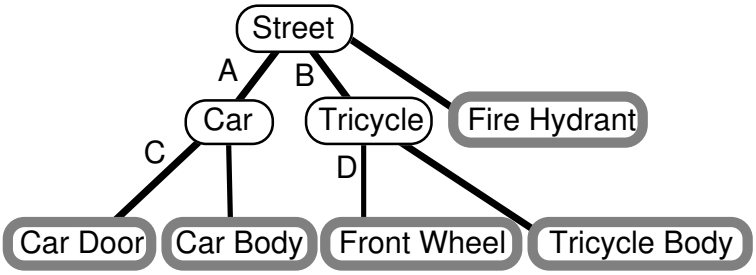


Figure 4.1 Tree Hierarchy Street Scene

File systems with subdirectories are trees too. This brings up the natural analogy of **naming** parts of the tree. Child objects can be referred to in the same way that subdirectories are. For example, Street / Car / CarBody identifies the car body while Street / Tricycle / TricycleBody identifies the tricycle s body.

[Foley82 section 9.5.3] describes the data structure for implementing this kind of tree structure. Each block of memory is either a leaf node or has a transformation matrix and a list of child blocks which are to be rendered under that transformation matrix.

The **transformation matrix** is just a compact way of storing a coordinate system. The transformation represented by the matrix consists of the translations, rotations and scaling operations needed to convert from the desired coordinate system to the parent coordinate system. Multiplying transformation matrices together is equivalent to applying the transformations one after the other.

When a tree is drawn by recursive traversal, a current transformation matrix is used to keep track of the composite transformation needed to get from the local object coordinates to the common top level world coordinates (the ones seen by the observer). Whenever a leaf node is encountered, it is drawn in world coordinates using the current transformation matrix to convert the primitive object s coordinates into world coordinates. When a transformation node is encountered, the current transformation matrix is multiplied by the transformation node s matrix and the result becomes the new current transformation matrix. After the children of the transformation node have been drawn, the current transformation matrix is restored to its value before the multiplication.

4.3. DIRECTED ACYCLIC GRAPH

The tree structure can be extended to a directed acyclic graph (**DAG**) structure by allowing multiple parents for each child. The directed constraint implies that edges go from a parent down to its children. The acyclic restriction is present to avoid infinite loops which would occur while trying to draw an object that has itself as a parent.

The additional complexity of multiple parents yields the benefit of object reuse. Compare the graph structure with simpler structures that define each object in the scene explicitly. If there are several cubes in the scene, there will be corresponding shape definitions for each cube in the simple structure. The DAG structure can define one cube and then make it appear at several places in the scene. The result of drawing an object in a coordinate system found by following a particular path through the graph to the master object is called an **instance** of the master object. FIGURE 4.2 shows what the DAG structure looks like for a scene with six triangles and three cubes (nine instances) yet only one triangle and cube (two master objects) in the directed graph. Note that there are six triangles because there are six paths through the graph (ABEFH, ABEGH, ACEFH, ACEGH, ADEFH, ADEGH) down to the master triangle, object H. FIGURE 4.3 shows a close up view of the same scene.

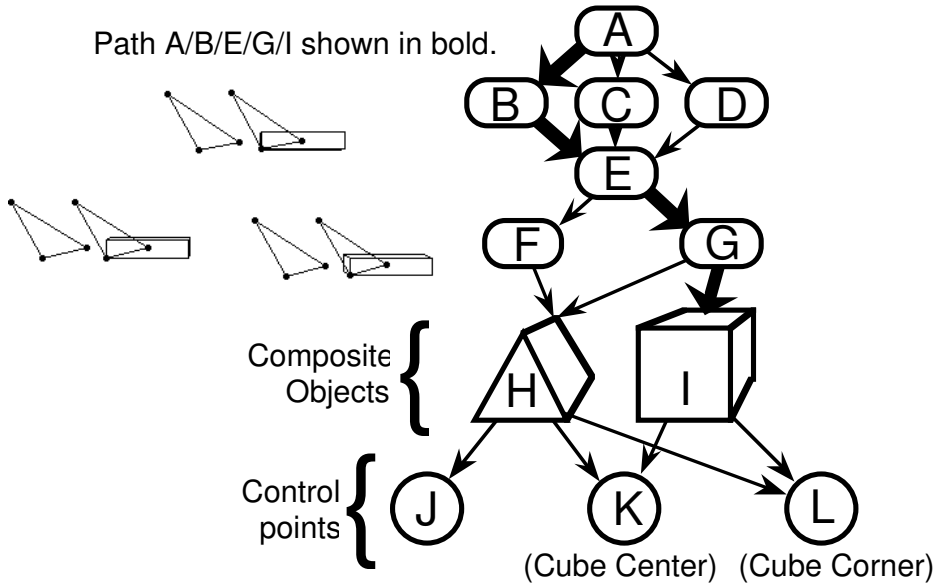


Figure 4.2 Graph for Six Triangles and Three Cubes

The inheritance of shape from a master object makes changes to all instances automatic when the master is modified. The DAG structure also has the advantage of requiring less storage space since all of the multiple instances of an object are represented by pointers to the one master object, not copies of it.

Much like file systems with multiple links to the same file to make it appear in several subdirectories, DAGs can make use of the same **naming** system. Note that the path A / B / E / G / I leading down to a particular instance (one of three) of the cube is highlighted in FIGURE 4.2. FIGURE 4.3 shows which objects come from which paths in more detail. The paths are more useful in DAGs than simpler structures because they identify one instance from many instances of a master object. In the simpler structures each object had only one instance in the scene.

As a final note, DAGs can extend the reuse of objects down to the **control points** used to define larger **composite objects**. The cube (I) and the triangular polygon (H) are examples of composite objects. Each is defined by various control points which are represented in the graph as children of the object. The cube has just two control points, the center and a corner. The polygon s control points define its vertices. Note that the control points reside in the same coordinate system as their parent objects and that they are named by coordinate systems, not strictly by parents. This means that the name for one instance of

control point L in FIGURE 4.3 is A/B/E/G/L, not A/B/E/G/I/L since object I is a cube, not a transformation node.

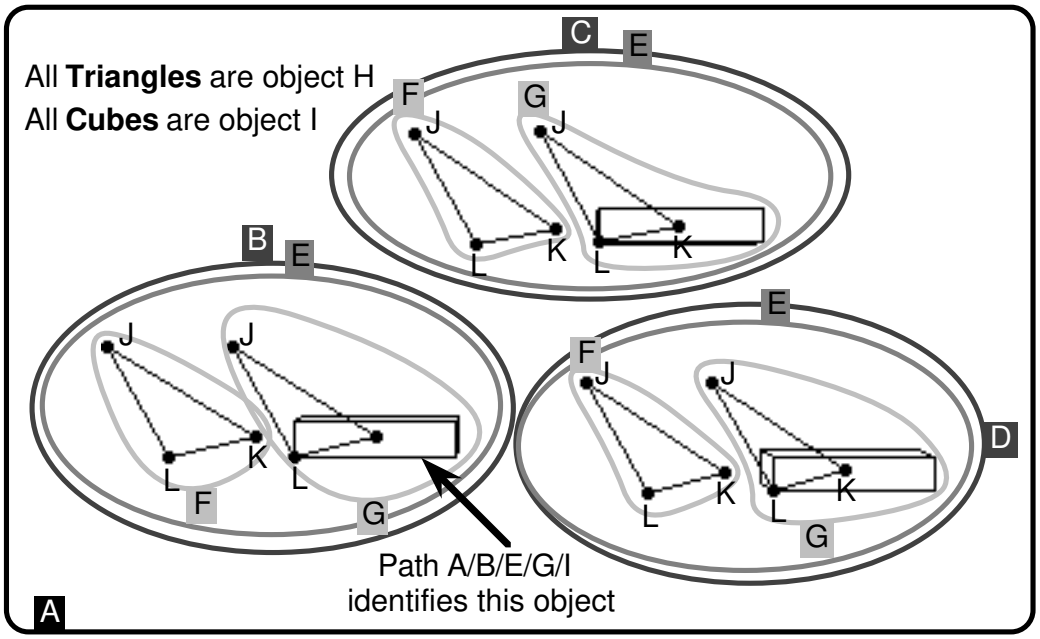


Figure 4.3 Image Derived from Previous Graph

4.4. PHIGS

The PHIGS (Programmer s Hierarchical Interactive Graphics System) standard [ANSI88] is worth looking at because it is a popular and relatively recent design for structuring graphics primitives.

PHIGS graphics are organized as a collection of things called *structures* (in italics to avoid confusion with the graphical structures of this chapter), each assigned a name meaningful to the programmer (such as arm or leg). Each *structure* contains a numbered collection of things called elements. Elements are the usual graphic primitives, attribute (colour and others) setting elements, transformation modifying elements and a special execute *structure* element that is much like a subroutine call to a *substructure*. When drawing a particular *structure*, the elements are processed in a linear fashion, including transformations.

The execute *structure* element identifies a child *structure* to be drawn with inheritance of properties (including the current transformation) from the parent. Several different

execute *structure* elements can refer to the same child and each child *structure* knows about its parents, unlike the earlier tree structure. Various functions are available (inquire paths to ancestors, inquire paths to descendants) for finding out all possible paths through the graph to the top or to leaves from a given element in a given *structure*. The paths are identified by a list of pairs of *structure* names and element position index numbers.

PHIGS s *structure* naming system and linear style seems to be organized more for naming objects of interest to the Human programmer rather than separating things by transforms. For example creating a robot arm with PHIGS lets the programmer list the arm parts in one *structure* and put the hand in another. The arm *structure* would be: draw top half arm, rotate everything from here on to the end of the *structure* by a given amount to represent the elbow joint, draw bottom half arm, do hand joint rotation, and finally execute a *substructure* to draw a hand. Note the appearance of a transformation (the rotation) in the middle of the *structure*, something that is not possible in a DAG.

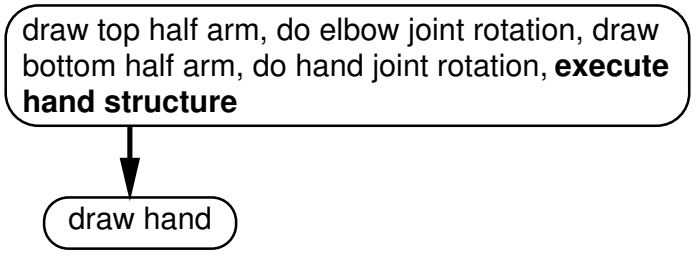


Figure 4.4 PHIGS Structures for Drawing a Robot Arm

PHIGS s linear style is equivalent to a DAG structure. Every time a transformation is specified within a *structure*, a new transformation node appears in the equivalent graph. The execute *structure* element adds the *substructure* as a child of part of the current *structure*. FIGURE 4.5 shows the graph structure that is equivalent to the previous two node PHIGS structure.

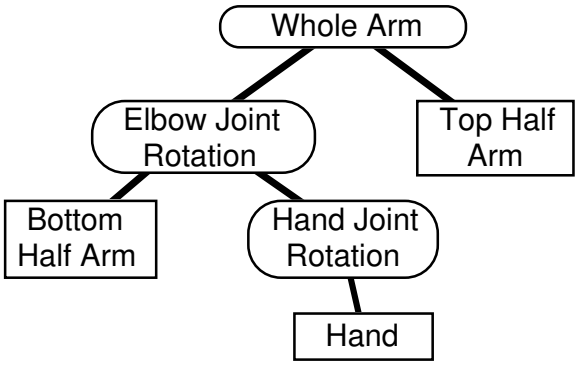


Figure 4.5 Equivalent Directed Graph

Handyman uses the DAG structure rather than the more readable PHIGS structure primarily because PHIGS wasn t available for the Macintosh. As secondary justification, PHIGS isn t needed for its readable code because a visual editor hides it all from the user anyways. The explicit transformations in the graph are also easier for the software to work with than transformations embedded in a *structure*. As a useful extension, caching is best done for each separate coordinate system, which would require extra work to obtain from a PHIGS graphical structure. Finally, some of the Handyman extensions to DAGs would be hard to add to an already standardized PHIGS.

4.5. HANDYMAN EXTENSIONS

Handyman s graphic structure has a couple of extensions to the conventional DAG structure. The use of wild cards and pattern matching of path names helps in selecting particular sets of instances of objects in the scene. The other enhancement is the addition of remote coordinate system references to create objects that are split between different coordinate systems.

Wild cards and remote references need a better definition of path names than previously given. One difficulty with the previously described system of putting coordinate system transformations in the edges of the graph is that ambiguity arises. FIGURE 4.6 shows an example where paths using node names aren t unique. Street / Car / Car Door identifies two car doors, one reached through edge A and the other through edge E. For path names to work, a unique instance needs to be identified instead of two.

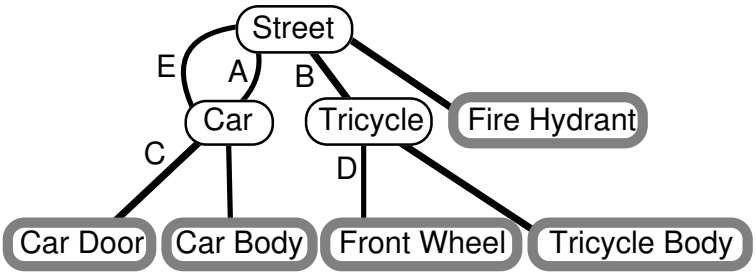


Figure 4.6 Problems with Transformations in Edges

Handyman gets around this problem by moving transformations into nodes rather than edges. FIGURE 4.7 shows that Street / Car A / Generic Car / Hinge Rotation / Car Door identifies a particular car door instance. Note that the Car A node is a transformation node that merely moves an instance of the generic car to car A s location. All the nodes that actually draw something have bold outlines while the rest are transformation nodes that just modify the location and orientation of their children.

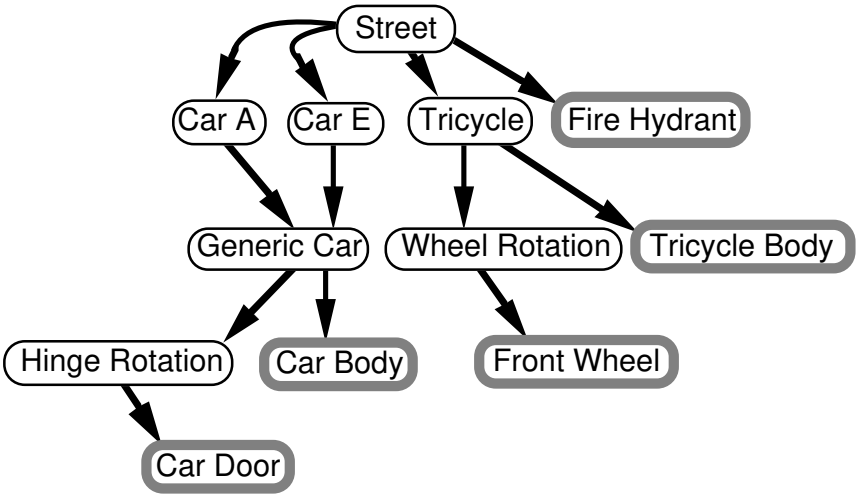


Figure 4.7 Transformations Moved to Nodes

One may think that this unwieldy path could be shortened by adding a transformation to all primitive object nodes. This will push the Hinge Rotation and Wheel Rotation transforms into their respective primitive object nodes. Unfortunately, this impacts on performance by giving all primitive objects different coordinate systems. Having different coordinate systems prevents the sharing of objects (such as control points) in a common coordinate system. Handyman thus keeps the transformations out of the primitive objects.

4.5.1 WILD CARDS

Wild Cards are regular expressions or some other form of pattern definition to represent a set of names. In the file naming convention of many operating systems, the * wild card character matches any number of any characters. Thus a file name of J* will match all file names that begin with the letter J . Similarly wild cards and pattern matching can be used in a named DAG structure.

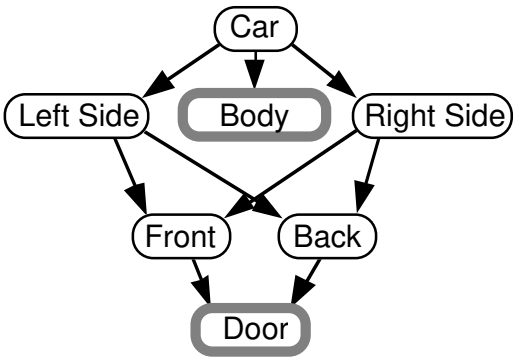


Figure 4.8 Car Door Paths

The graph in FIGURE 4.8 defines a car that has four door instances made from one master. The left and right side transformations move the master door image to the left or right side of the car. The front and back transformations then move it a bit forwards or a bit backwards within a particular side to define either a front door or a rear door.

Wild cards can be used to hide a missing right back door if a damaged car was being defined. A common technique is to specify an include and an exclude pattern for the object. If the path to the object matches the include pattern and doesn t match the exclude pattern then the object is drawn. The default include pattern would match anything while the default exclude pattern would not match anything.

For the car example, the master door s exclude pattern would change to * / RightSide / Back . Remember that there is only one door! That pattern will allow three instances of the master door (LeftSide / Front, LeftSide / Back, RightSide / Front) to be drawn and will cut off the rendering of the remaining instance when that path is encountered in the traversal of the car graph. Note that the pattern should compare the path of transformations down to but not including the master object being drawn.

In practice, four separate door transformations would be used because having shared transformations to move the doors makes the doors all move in unison. For example, changing the Front transformation would make both front doors open somewhat unrealistically. On the other hand, this linked motion could be useful for animating bird wings and chorus lines

4.5.2 REMOTE COORDINATE SYSTEM REFERENCES

Remote coordinate system references are another extension to the DAG structure. They solve the problem of objects that need defining points in two or more different coordinate systems. For example, an elbow joint needs a flexible patch over the joint so that a yawning gap doesn t open up when the arm is rotated about the joint. This patch needs to be attached between the lower arm (point A in FIGURE 4.9) and the upper arm (point B) and thus needs points from both the upper and lower coordinate systems.

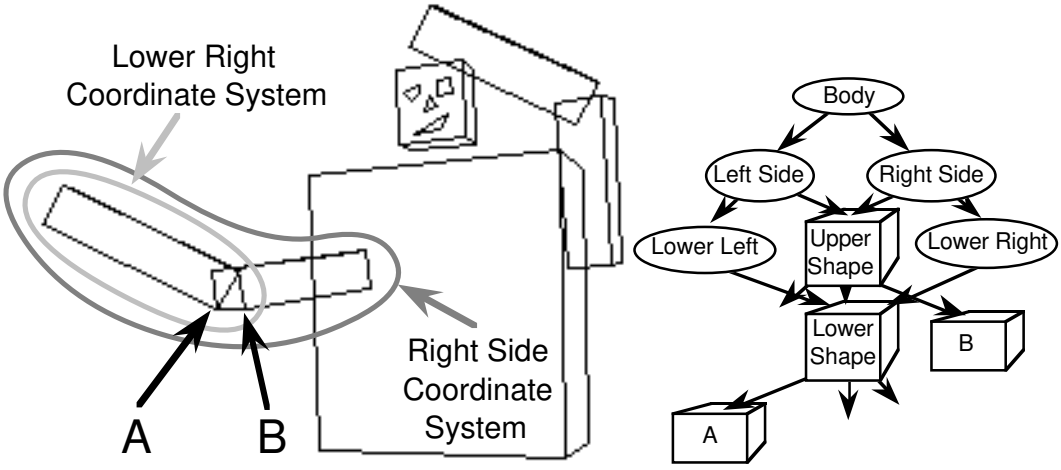


Figure 4.9 Elbow Joint Coordinates Problem

This kind of operation isn t easily supported in the ordinary graph hierarchy since the referenced point isn t a direct child of the object being drawn. The idea is to mix references to points in the local coordinate system with points in an unrelated coordinate system. The first problem is specifying the point (including the coordinate system) and the second is drawing it.

Using an absolute path name (specified fully from the root on down to the target point) would specify a particular coordinate system but is somewhat limited and makes it difficult to rearrange the graph structure without changing absolute path names. Absolute paths also

restrict the remote reference to one particular instance. It may be desirable to have a remote reference to an instance that is related to the current object. In the robot example, it would be nice to define just one patch which gets reused (two instances) for both sides of the robot with references to point B automatically adjusting to the right hand side B or left hand side B instance as appropriate.

The file system analogy suggests the traditional current directory modification syntax which allows references that are relative to the current directory. In this case, the initial current directory is the path from the root down to but not including the remote reference master object.

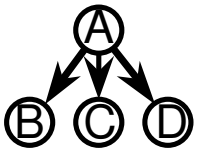
Up	Change to the parent coordinate system. Similar to .. in UNIX and MS-DOS.
X	Descend to subdirectory named X.
Sibling	Switch to the next sibling directory.  Current Directory Before After A/B A/C A/C A/D A/D A/B

Table 4.2 Directory Modification Operations

The patch for the robot s right side elbow is located in the coordinate system Body / Right Side / Lower Right and would use a coordinate system modification string of up / B which is equivalent to Body / RightSide / B . In more detail, the current coordinate system starts out as Body / Right Side / Lower Right when the patch is being drawn. The first element of the coordinate system modification string is up which changes the current coordinate system to be Body / Right Side. The next and last element is B which identifies Body / Right Side / B even though B isn t a direct child of Right Side. This is legal because primitive objects don t include transforms; Upper Shape and its child point B are both in the Body / RightSide coordinate system. Similarly, the same patch when drawn under Body / LeftSide / LowerLeft would reference the point Body / LeftSide / B . The power of path modification lets the patch work under both the left and right sides of the robot.

A special remote reference object is used in the structure to identify the target coordinate system and the target object. Usually the target object will be a point but theoretically it could be any object in the graph which wouldn t create a cycle. This means any object except ancestors of the remote reference object. In practice, only points and perhaps lines and polyhedra faces are useful candidates for remote references.

Unlike ordinary primitive objects, the remote reference doesn t have any meaningful location in space. Most objects when drawn under the same coordinate system have positions that are the same relative to other objects in the same coordinate system. Remote references (like the point in the elbow patch) appear at different places relative to the non-remote objects, depending on the differences between the coordinate systems. Thus trying to find the center of a polygon with remotely referenced points is doomed to failure since the polygon can change shape when the remotely referenced object appears in a different relative position.

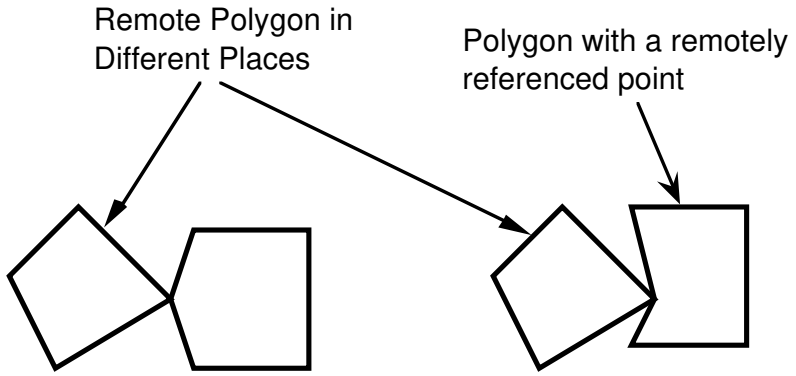


Figure 4.10 Remote References Change Shape

The rendering program takes two extra steps to draw a remote reference. The first calculates the absolute path from the current directory and the directory modification string. Then it can use that path to find the actual coordinate system (requiring a path traversal). Finally, the renderer can draw or otherwise use the target object under that coordinate system.

Another use of remote references (other than elbow patches) is to implement **volumes of revolution**. A volume of revolution is the shape derived from swinging an outline around a central rotation axis, much like the lathe operation in machining. Only the points along the defining edge of the desired volume need to be created if remote references are used. The mirror image points that define the outline of the volume at different angles are

instances of the defining points under different rotation transformations. The circumscribing edges going from one angle of rotation to the next use a remote reference to cross between the different rotation coordinate systems.

In the example below (FIGURE 4.11), the Slice object is a grouping object in case there is more than one polygon making up each segment of the volume. The collection of rotation transformations rotate the slice about the central axis to draw each segment of the whole volume. The Face object is a polygon for one facet of the cylinder. It has four defining vertices, two are ordinary points and two are remote references to the same points under the next sibling rotation transformation. The result is a face polygon that covers a complete segment of the volume while only using two real points. If one of the rotation transformations is modified (perhaps by adding a translation outwards), the face will automatically adjust to connect between the new position and adjacent rotations.

As another example of how path modification works, consider drawing the remote reference to Top Point. The up / sibling / Slice / Top Point path modification string starts working on the initial path Volume of Revolution / 0° / Slice, assuming that the face at the 0° to 36° angle is being drawn. The first up changes it to Volume of Revolution / 0°. The sibling operation then modifies the current path to be Volume of Revolution / 36°. Finally, the rest of the modification string locates the desired point Volume of Revolution / 36° / Slice / Top Point.

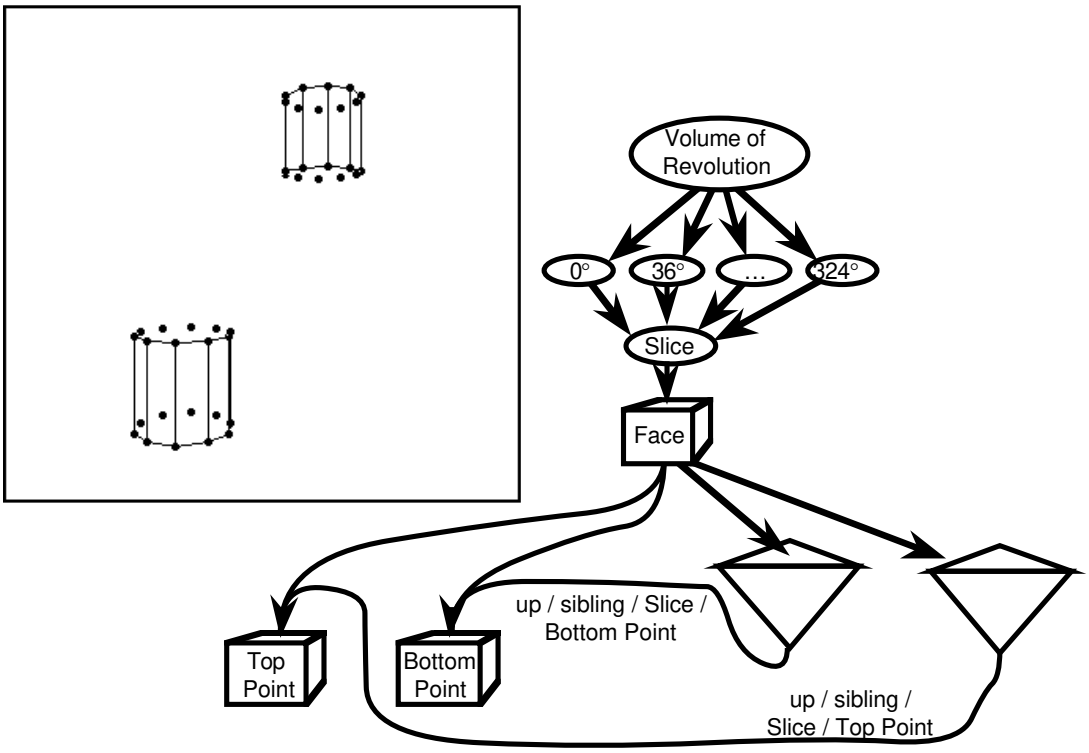


Figure 4.11 Volume of Revolution

The volume of revolution graph structure is too complex to be easily created and should be created by the user interface rather than directly by the user. Once it has been created, the structure makes all of the instances of the defining points follow the master point when it is dragged by the user. Because of the rotation transformations, the instances automatically follow the movement in a natural motion that is radial to the central axis of revolution.

To sum up, Handyman s wild cards, path modification and remote references add new functionality to the directed acyclic graph structure.

4.6. SUMMARY

The various different ways of structuring the graphics primitives trade off complexity for ease of use. The simplest one, a flat collection of graphics objects, is easy to implement yet hard to use for operations such as grouping. Tree structures allow easy grouping of objects and movement of the group as a whole by using inherited coordinate systems. Directed acyclic graphs add reuse of objects, allowing one master object to define multiple instances in the scene. PHIGS is essentially a DAG modified to be in readable chunks. PHIGS also provides some operations for finding path names, though the paths aren t used as part of

the drawing system. Handyman improves on the DAG and path name idea by using path names as part of the drawing process. Wild cards are used to test path names to select or deselect instances to be drawn. Path modification strings are a more useful idea that builds on path names to create remote references. Remote references then can be used to build objects, such as elbow patches, which span several coordinate systems.

Chapter Five

5. CACHING FOR SPEED

Handyman s 3D display of the puppets being animated needs to be drawn quickly to give feedback to the animators. A large part of the necessary speed can be achieved by reusing the drawings of objects that aren t moving and by recycling intermediate data from the drawing process. Besides examining what can be cached, a way of organizing all this data neatly into a tree structure is elaborated upon and some methods for linking graphics structure changes to cache validity are compared. Finally, a brief glimpse into the complexity of time domain caching is given.

There are several reasons why a cache would be useful for improving the speed of a 3D graphics system. Operations in a graphics editor involve mostly small changes to the scene as small parts of an object are manipulated. The same rule applies to animation where the background and stationary objects are the same from one frame to the next. These unchanging parts of the images are worth caching. Caching is also appropriate for a system like Smalltalk which is slow and has lots of available memory.

5.1. CACHEABLE THINGS

Things that take many operations to calculate are worth caching. Looking at the sequence of operations for drawing an object can reveal substantial cacheable items. The conventional three-dimensional graphics pipeline as described in chapter 9 of [Foley82] has these stages:

1	Convert the locations of all primitive graphics objects from the various nested local coordinate systems to a common world coordinate system.
2	Transform the object s world coordinates location into standard view volume coordinates based on the camera position and orientation.
3	Clip against the canonical viewing volume (usually a truncated pyramid shape). Clipping involves intersecting the object with the viewing volume and cutting off the parts outside.
4	Perform image transformations. These allow quick movement of the image of the clipped graphics, without having to go through the rendering of the entire graphical structure.
5	Perform a perspective transformation to convert to screen coordinates.
6	Scan convert the object to a screen image. Can use various techniques to remove hidden surfaces.

Table 5.1 Standard 3D Graphics Pipeline

Handyman uses most of the standard pipeline, excluding world coordinates and image transformations. That leaves view volume coordinates, screen coordinates and images as obvious cacheable objects. The intermediate local coordinates to view coordinates transformation matrices that arise when traversing the graphical directed acyclic graph are also worth caching since they don t need to be recomputed when their transformations aren t changing. TABLE 5.2 shows what is cached in Handyman and what computations are saved by caching.

Cacheable Thing	What Caching Saves
View Coordinates	Calculation involves a multiplication of local coordinates by a local to view matrix. Clipping coordinates to fit the viewing pyramid also takes CPU time.
Screen Coordinates	Multiplication of view coordinates by a perspective matrix. Not quite as expensive as the view coordinate calculation but still costly.
Transformation Matrices	The local to view coordinate transformation matrices are worth saving for every different coordinate system in the graphics structure.
Remote References	Remote references should cache the pointer to the target object s cached data. This saves doing a traversal of the graph to find out where the remote reference path name leads to and which coordinate system transformation matrix is involved.
Bitmaps	The image of an object and its children can be saved and then reused when the object needs to be drawn again.

Table 5.2 Things that can be Cached

Unlike the DAG graphical structure where one node represents several instances of an object, cached data is different for each instance of multiple instances (A / B / object has different cached data than A / C / object). The problem of caching each instance can be solved by storing cached data in a tree structure that is derived from unrolling the DAG. FIGURE 5.1 shows part of the tree equivalent to the graph (FIGURE 4.2) of six triangles and three cubes.

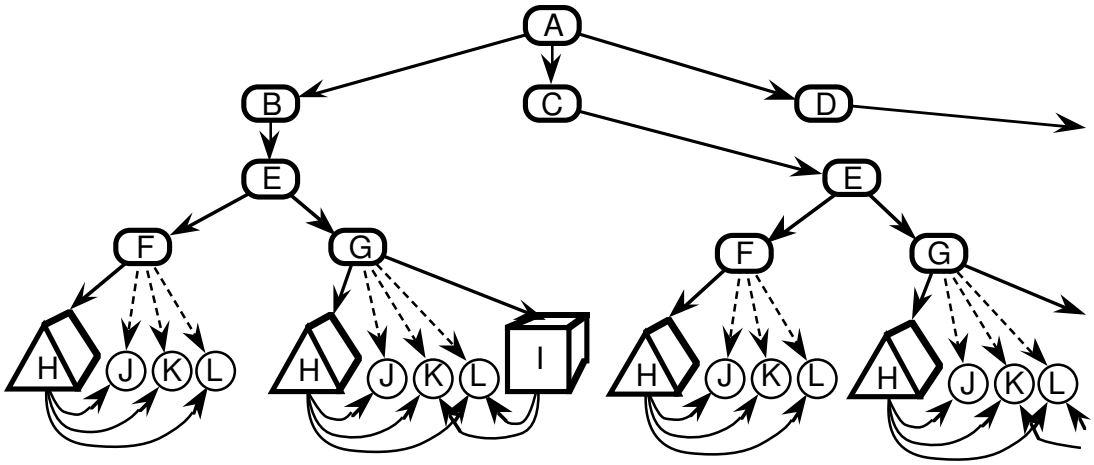


Figure 5.1 Tree Equivalent to a Graph

Note that when a shared object appears more than once in the same coordinate system (such as points K and L which are shared between the cube and triangle), it is in the equivalent tree only once. That way its cached data only needs to be computed once and is then reused when the multiple parent objects draw it. As mentioned previously (section 4.5: Handyman Extensions), this is the reason why primitive objects don t have their own internal transformation matrices and coordinate systems.

To find the cached data when drawing a particular object, the current directory transformation path is used to go down the equivalent tree to the desired coordinate system. Then the object being drawn is used to find the particular leaf of cached data off the last transformation node in the tree. If it doesn t exist then it can be created and calculated. If it is already there then the cached value can be reused.

The tree of cached data is also useful for finding objects under a mouse click. Given a 2D screen coordinate, the cache tree can be recursively searched for all objects that have a bounding screen coordinate rectangle containing the point. Because of the tree structure, whole hierarchies of objects can be quickly rejected as not containing that point when their

topmost object s screen bounding rectangle doesn t contain the point. The result of the search is a path name that identifies a particular instance of a graphical object. If there are several objects then several path names can be found, selecting between them is up to the user interface.

5.2. MOVING OBJECTS AND CACHE INVALIDATION

Whenever an object in the DAG structure is moved or changes shape, the cached data for the moved object has to be invalidated so that its graphics will be updated to show the motion. This is more complex than it seems because more than one object can be invalidated by a single change, for example a move done to a transformation node can affect all its children. The changes to the structure also have to be efficiently reflected in the relevant cache nodes, leading to an investigation of ways of linking the graphical structure to the cache tree.

Handyman tracks cache validity by marking objects in the graph structure as having their drawing information or coordinate system information changed since the last time the scene was drawn. The alternative of marking cache nodes runs into the difficulty of having many cache nodes representing the same object in different coordinate systems. There are different kinds of marks for transform matrices, view coordinates, screen coordinates and image data. They are different because some changes affect only some of the cached data. For example, only bitmap image data has to be marked as invalid when an object s colour changes since the cached geometry data is still valid.

When one object is moved not only is it marked as being changed, but its children and parents may need to be marked too. For example, adding a vertex to a polygon changes the shape of the polygon and thus the shape of everything that uses the polygon. Moving a high level grouping object results in moving all its children, as well as changes to the shape of parent objects using the group.

The following table lists the main types of changes that arise from modifications to the graphical structure and what actions are taken for each change.

Change	Actions
Adding or Removing a Child	Example: adding a vertex to a polygon. Do a shape change since adding or removing a child changes this object s shape
Adding or Removing a Parent	Do almost nothing. Adding a parent results in the automatic creation of a cache node during redrawing, removing one just leaves behind an unused cache node.
Shape	Example: moving a point. The image of this object has to be redrawn. Invalidate this object s view coordinate, screen coordinate and bitmap caches. Send a propagated shape change to all parents.
Propagated Shape	Same effect as a shape change. It is kept separate for stylistic reasons.
Transformation	Example: rotating a transformation node. Invalidate cached local to view transformation matrix. Do a shape change on this object. Send a propagated transformation change to all children.
Propagated Transformation	Example: ancestor transform indirectly rotated this object. Invalidate bitmap, view and screen coordinate data. Also invalidate cached local to view transformation matrix if this is a transform node. Send a propagated transformation change to all children.

Table 5.3 Graphical Structure Changes and Cache Invalidation

There are some complications with adding and removing parents that aren t shown in the table. Adding a parent to an object naturally implies that the object is added as a child of the parent too. Adding or removing a parent also affects remote references that go through the new connection or went through the old connection. The difficulty is that it is hard to determine which remote references use a particular parent link because of wildcards in their path references. The simplest approach is to mark all potential remote references as invalid when a parent is added or removed. A more complex approach would be to find all possible paths that a remote reference could use and put triggers into every object crossed by those paths so that the remote reference is invalidated by any relevant structural changes.

Propagated changes get passed up to all ancestors or down to all children depending on the type of change. Once a particular type of change has passed through an object, a flag is set so that redundant propagations of the same change are detected and stopped. The flags are reset the next time the graph is drawn.

As a side effect of shared objects in multiple coordinate systems, some changes invalidate cached data which shouldn t be invalidated. The advantage of implicitly marking multiple cache node instances by just marking one object in the graphical structure has turned into a disadvantage. In FIGURE 5.2, transformation node B has been rotated and cache invalidation has been propagated around the graph. Theoretically the cache nodes for paths beginning with A/C and A/D should still be valid. However, since we are using the graph to mark many cache nodes as invalid, all instances of an object are invalidated. For example, object I is invalidated which means that cache nodes A/B/E/G/I, A/C/E/G/I and A/D/E/G/I are all marked as invalid even though the latter two (A/C/E/G/I and A/D/E/G/I) still have up-to-date data.

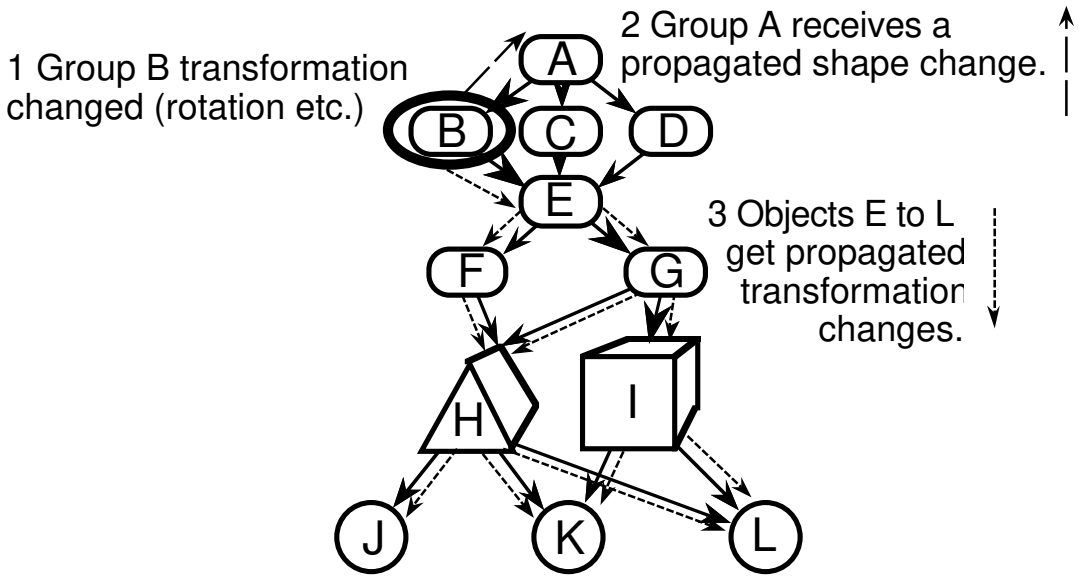


Figure 5.2 Spurious Change Propagation

However, this inappropriate cache invalidation isn t a big problem when caching has taken place upstream of the invalidated nodes. In the example, grouping nodes C and D could have cached their bitmaps from the last time the graphics were drawn. When object A/C or A/D is draw again, it would reuse that bitmap and wouldn t have to traverse down to its children.

5.3. GRAPHICAL STRUCTURE TO CACHE LINKAGE

A mechanism is needed for invalidating all the cache nodes that cache instances of a particular graphical object when that object is changed. There are two contenders: the lock /

unlock protocol and a version number mechanism. Handyman uses the version number method because it has speed advantages in the Handyman environment and allows automatic garbage collection of unused cache nodes.

Smalltalk s View class suggests a **lock / unlock protocol**. A lock operation is used to get the current value from an object. If it is already in a locked state, it just returns the value that it has cached. If it is not locked, it recomputes its value, caches it, switches to a locked state and returns its new value. When an object is locking in a new value, it can lock the other objects that it needs to obtain values from.

Unlocking is equivalent to invalidating cached data. If an object s value becomes invalid and it isn t already unlocked, the object switches to an unlocked state and then tells all things that depend on its value to unlock . Unlocking spreads through the network of objects since objects that are told to unlock then tell all their dependents to unlock.

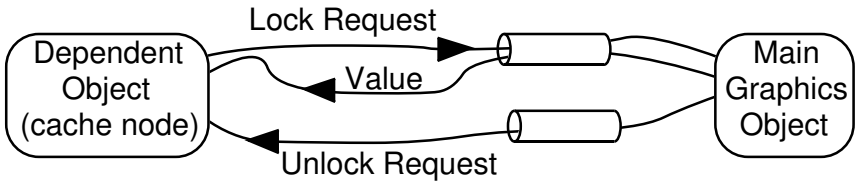


Figure 5.3 Lock / Unlock Protocol

The **version number method** stores a version number in both the main object and its dependents. When a dependent is locked to obtain its value, it uses the cached value if its version number matches the main object s version number. Otherwise it recomputes itself using the current value of the main object, caches the result, updates its version number to match the main object s version number and returns the result.

When the main object changes it simply increments its version number to mark all dependents as invalid.

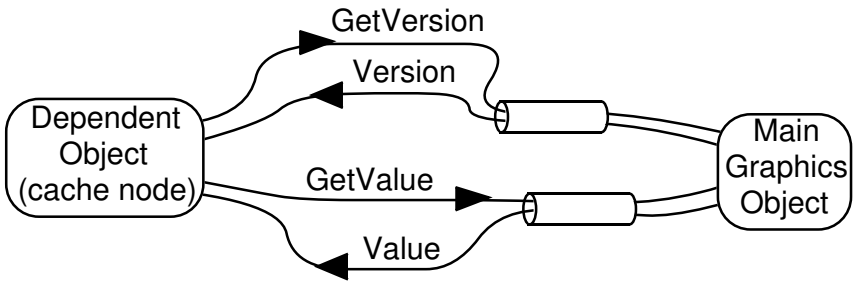


Figure 5.4 Version Number Protocol

The lock / unlock mechanism has the nice but unimportant advantage of automatically handling networks of computation. With the version number system a change in version number doesn t propagate around the network of computation like an Unlock request does. This means that a chain of dependencies (A depends on B s value, B depends on C, etc.) would have to be handled by explicitly testing all version numbers to the end of the chain to make sure that everything is still valid. However, this is a moot point since there are no chains of dependencies in Handyman, just one graphical object with directly connected dependent cache nodes.

A disadvantage of the lock / unlock method is that each main object has to have a list of dependent objects. This list is needed to identify the recipients for unlock messages. The objects that want to stay informed of changes also have to register as a dependent with all objects that they might send a lock request to. The lack of a dependents list in the version number method saves memory space and also eases garbage collection. When a dependent object is no longer needed, it is automatically garbage collected under the version number method as soon as all references to it have gone. With the lock / unlock technique, the unused object would have to be explicitly unlinked from the dependents list before it can be garbage collected.

The version number method can quickly invalidate all the dependent objects by just incrementing the version number in the main object. This becomes more significant when there are many inactive dependents. Changing the main object several times with the lock / unlock technique sends many useless Unlock messages to the dependents which are already unlocked. The version number technique postpones the invalidation to the time when the cached data is actually used. Conversely, the lock / unlock technique is marginally faster at returning the cached data when it is in a locked state since no version number test has to be done.

To sum up, the version number cache invalidation method is more useful for Handyman because it saves on memory space and is faster.

5.4. FUTURE WORK — TIME DOMAIN CACHING

The current caching scheme can be extended to improve caching for replays of animation. Instead of just caching stationary objects, motions that are the same as previous motions are cached. Internally, a sequence of values over time is stored in place of single values. As an example, time domain caching is useful for making a cartoon of a dog chasing a ball where the ball s motion has already been done. The bitmaps and other data for the previously recorded ball motion are cached for all the frames in the take to improve drawing speed on subsequent takes.

Parameters for graphics objects become values that vary over time. A spline or other interpolation technique would have to be used for values that used to be a simple number. Attributes such as an object s list of children should also be allowed to vary over time. For example, tearing off the arm of a cartoon character would require a change in the list of children for the shoulder object. Conceptually, the list of children becomes a collection of lists that are indexed by time.

Cache invalidation becomes more complex with caching over time. Each cache node in the cache tree would contain an indexed collection of cached data, with the time being the index key. If an object is being modified over a given time range (typically the time when a recording session will be done) then the cached data would have to be invalidated over the same range of time without invalidating the data outside the range. This means that a single version number is no longer useful for controlling cache invalidation. A collection of version numbers indexed by time is now needed.

Caches should also recognize stationary objects so that they can reuse the cache from the time when the object first becomes stationary until the next time it moves. A stationary object is one that isn t moving (no shape changes) and isn t being moved by changing parent transformations. Note that some instances of an object can be stationary while others are moving, depending on what path through the graph is taken to create the instance.

Change propagation also becomes tricky when the graph structure changes during the time range because the list of parents and children could have several values. Changes propagating over a time range would have to be split into groups of changes propagating

over contiguous time sub-ranges. For example, if an object s parents are {A,B} for time range [0,2) and {B} for [2,10) then propagating a change upwards for time range [1,5) would be broken into propagation to {A,B} for range [1,2) and another propagation to {B} for time range [2,5).

This somewhat complex detection of moving objects and changing objects needs to be done just once per take if the animator declares in advance what things are going to be changed. A bit more complexity arises if the graph structure is dynamically changed during the take (adding and removing children) since it results in a different change propagation pattern than the one initially calculated.

5.5. FUTURE WORK — INCREMENTAL RENDERING

The idea of incremental rendering [Bergman86] is to draw increasingly better versions of the same image over time. For example, after drawing the wire-frame images, if spare processor time is available, hidden surfaces can be done.

The easiest way to implement cached hidden surfaces is with z-buffers (a bitmap with depth information for each pixel, see [Foley82]) since z-buffers can be combined in much the same way as cached bitmaps are already combined. The wire-frames for moving objects would be quickly drawn into their cached z-buffer, perhaps just setting the depth to be a single shallow value so that the wire-frame shows up in front of all other objects. Objects that are stationary would have their cached z-buffers updated with an improved solid rendering of their image by a low priority process.

Often the cached wire frame data can be used for the higher quality rendering too. For example, the wire frame edge outline of a polygon can be used as a starting point for filling in the inside with colour and for scan converting depth information.

The wire frame drawings of moving objects would be combined with higher quality solid rendering of less dynamic objects as a part of the normal drawing process. The result is a scene with a vivid coloured solid background overlaid by wire frame moving puppets. Whenever something stops moving, it becomes solid and gradually assumes a more realistic colouring.

5.6. FUTURE WORK — RENDERING SOUND

Sound can be added to the graphical structure by defining sound emitting objects and listening objects. The emitters would have an associated sound recording that would be played back at their location in space. The listeners hear all sounds and generate a sound track in the final animation. By using a large sound buffer in the listener s cache node to store time delayed sounds, various audio effects can be simulated.

The sound sources have a sound waveform and a location in space. A sound waveform consists of numerous sound samples, several thousand for each second of sound. Each sample is a signed value that represents the average amplitude of the sound wave over a very short period of time. For comparisons, compact audio laser disks (CD s) use a 16 bit value for each sample and use about 44000 samples per second. Digital telephone systems use 8 bit samples at a rate of 8000 per second.

Each sound listener cache has a buffer that will store the sound waveform it will hear from the current time to a few seconds into the future. When a sound is rendered, the sound emitters store their sound into all sound listener instances. Each sample value in the sound emitter covers a time interval [t, t+∂) where ∂ is the time period for each sample (∂=1/44000=0.000022727 for CD quality). The sample for that time interval is then mapped to the sound listener buffer at time interval [t+d, t+d+∂) where d is the time it takes for the sound to arrive at the sound listener. Time d is based on the speed of sound in the simulated world and the instantaneous distance (in world coordinates, at time (t+t+∂) / 2) between the emitter and listener. The amplitude of each sample is also modified before it is added to the listener to account for the decrease in loudness over distance. The destination time interval usually will cover two sample cells in the listener s buffer so the sample value is split proportionally between them before being added. The end result is sound that has the appropriate volume and phase changes, naturally resulting in correct Doppler effects and other audio sensations. Two sound listeners close together can be used for listening to the world in stereo.

The only restriction is that the sound listeners shouldn t move. If they move then when the sound emitter writes a sample to the listener for some future time, the listener may have moved away and would actually get the sample at some other time. Alternatively, sound emitters could be required to remain stationary.

To save on computer time, the sound object s positions don t have to be updated for each sample. The objects could be moved for each frame of animation and the sound sample rasterization could be batch computed for the time duration of each frame rather than on a sample by sample basis. This would ruin the phase information (no Doppler effects) and reduce the stereo quality.

Discussions in sci.virtual-worlds about virtual sound referenced some research that used the filtering effects of folds in the ear to give cues for the direction of a sound. This use of sound filtering is another area of 3D sound that is worth starting research on.

5.7. SUMMARY

Handyman makes use of a cache tree structure to speed up graphics by recycling information about stationary objects. The tree structure solves the problem of keeping track of multiple instances of objects by making branches for every different coordinate system used. A simple system of version numbers is used to efficiently link modifications of graphics structure objects to all their cached data. Finally, some extensions for incremental rendering and caching motion and sound were considered.

The performance of the caching system is a success. In a test scene with 27 cubes, one of which is moving, the stationary cubes flash onto the screen while the moving one takes noticeable time to appear. Unfortunately, the time it takes to draw even the simplest moving object is extremely long. This is largely attributable to Smalltalk s implementation of floating point numbers as real objects, each primitive operation generating a newly allocated object. This makes matrix and vector operations particularly slow.

Chapter Six

6. NETWORKED SMALLTALK

Handyman uses an underlying computer communications network to collect the motions of many different animators during a real-time performance. As previously mentioned, the information flow starts from the animator s control device (usually but not necessarily a mouse). The device handler is sampled by a sequence of information manipulating processes that interpret control movements as more abstract quantities such as acceleration. After passing through several stages of manipulation, the information feeds into the graphical structure, perhaps changing a joint angle or moving something. Finally, a graphics image is generated from the structure. This chain of information relies on an underlying communications network to transport information from the animator s workstation, through various processing nodes and finally to the computer drawing the graphics.

Why is a network needed? One pragmatic reason is that there is no easy way to attach more than one mouse to a Macintosh. Even if it were possible, seating space is limited at one workstation. Another good reason is that multiple computers have more processing power and storage space than a single computer.

Handyman fits nicely into the distributed model since the motion configuration wiring diagram naturally splits into parts which have lots of local communications (the animator s device and the process recording its actions) and fewer global communications (less frequent, small updates to the graphics). Ideally the programming of communications between the parts should be the same for both local parts talking to each other and remote parts talking over the network. It would also be nice if the Smalltalk object oriented programming style was maintained since the parts are obvious objects and it is easier to send Smalltalk messages to objects than to do explicit message queue handling operations for each part.

Networked Smalltalk is the answer to those wishes. The base for building the whole system is Parc Place Smalltalk-80™ version 2.3 running on an Apple Macintosh™. Several layers of communication protocol were designed and implemented to map from Smalltalk messages all the way down to network operations.

6.1. NETWORKED SMALLTALK EXAMPLE

A proxy object system seems to be the easiest way to keep the Smalltalk message sending paradigm and to make objects on a remote computer look like local ones. The proxy is a local object that pretends to be the remote real object. Every message sent to it gets transmitted over the network (much like a remote procedure call) to be executed by the real object.

The following code example shows how the proxy object appears and is used by the programmer, without worrying about how it is implemented. In the example, a proxy array object is created, initialized and summed up.

```
| remoteArray sum |

remoteArray := NetworkPrimitives evaluateString: 'Array
new: 10' address: Somewhere.

remoteArray atAllPut: 12345.

sum := 0.

remoteArray do: [anElement | sum := sum + anElement].
```

Figure 6.1 Proxy Example Code

The first two lines declare the local variables and initialize the remoteArray variable. RemoteArray contains the result of evaluating the string Array new: 10 on the computer specified by the Somewhere global variable. NetworkPrimitives is the name of a global object that handles network requests and interfaces to the primitive C code that runs the lower level network operations. The result of Array new: 10 is a newly created array of size 10, each array position containing a default value of nil. However, since the array exists on another computer, remoteArray actually ends up containing a proxy to that array.

The resulting object structure looks like FIGURE 6.2. The proxy object is in essence a pointer to the real object, with a network ID and **oop number** (an integer uniquely identifying a particular object within a computer) rather than a memory address.

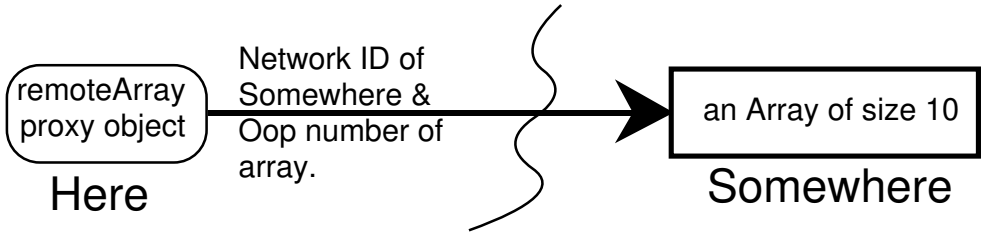


Figure 6.2 Proxy Object Pointing to a Real Object

Next, this statement is evaluated:

```
remoteArray atAllPut: 12345.
```

The message atAllPut: is sent across the network with the argument 12345. Since 12345 is just a number, it gets copied over the network rather than being converted into a proxy. The message is evaluated at Somewhere and sets all the positions of the array to the number 12345. The returned result of the message is the array object itself, which is sent back to Here as yet another proxy. The array proxy result is discarded after it has been received because the result of the statement isn t used.

This useless transmission of an unused result is wasteful but difficult to avoid. Perhaps, as a colleague suggested, using delayed evaluation objects as the results of all remote messages would be effective. Then when someone tried to use the delayed evaluation place-holder object, the actual result would be obtained from the remote system. However, this would add some complexity to garbage collection of unused results.

To continue the example, the sum of the elements of the array are computed next. FIGURE 6.3 illustrates the messages passing between proxy objects and real objects when the following two lines of code are executed:

```
sum := 0.

remoteArray do: [anElement | sum := sum + anElement].
```

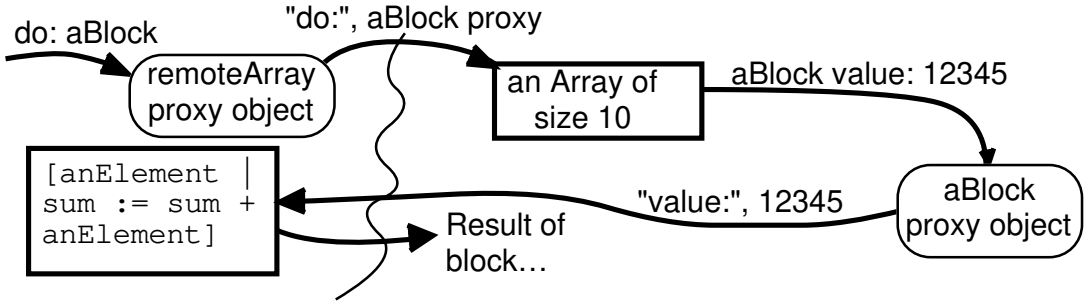


Figure 6.3 Example of Adding the Elements in a Remote Array

Initially, the message do: is sent to the contents of the remoteArray variable with a single argument a block. The proxy for the array bundles up the message selector do: and includes a proxy reference to the argument since it isn t a simple object like a number that could be copied. Note that a **block** in Smalltalk is a context and a piece of code the square box in the lower left with the code inside represents the real block object which is also the argument to the initial do: message.

At Somewhere, the do: method is evaluated by the real array. The array sends the message value: to the block s proxy several times, once with each element in the array. The proxy for the block gets the value: message and sends it to the real block object with the single array element argument given to it. Since the argument is a simple number, it is copied over the network.

The block gets the value: message and executes the sum := sum + anElement statement. The block then returns a result (the partial sum in this case) back to the do: method which discards it. Finally, the do: method finishes all the elements in the array and returns some more irrelevant information to the original invoker. As a side effect, the sum variable on the local computer now contains the sum of the array elements.

6.2. PROXY OBJECTS

Since a proxy object is a local object that represents an object on some other computer, it has to intercept messages sent to it and then somehow send them off over the network to be executed by the real object. These two steps suggest a natural internal division of the proxy into two objects. One object handles the simple task of intercepting messages and the other handles the network accessing. FIGURE 6.4 shows what this looks like.

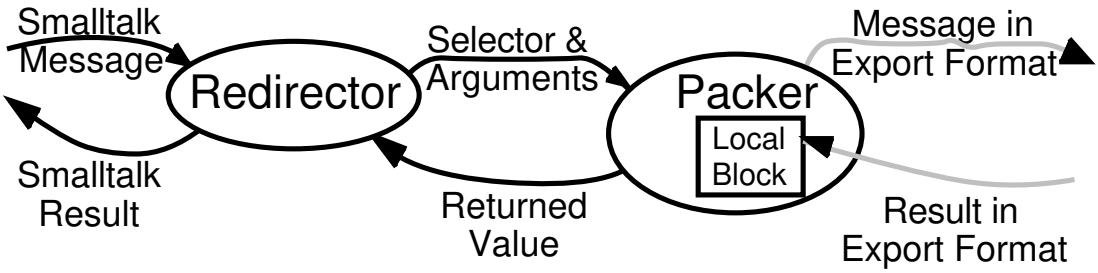


Figure 6.4 Message Redirector and Packer

The redirector is implemented as an instance of a class that has nil as a superclass. The result is an object that doesn t inherit any methods from class Object and thus all messages to it are unknown and handled by the doesNotUnderstand: method. The redirector s doesNotUnderstand: method converts Smalltalk messages into a method selector and argument list which is then passed on to the packer. This technique is far from original, see [Thomas87] for a discussion of its efficiency.

The two part approach makes it easier to manipulate the proxy with the debugger; without it the debugger s probing messages would get sent over the network to the real object. Using two separate objects also allows inheritance from class Object within the packer, where most of the proxy code is. For a minor improvement in efficiency, the packer and redirector can be combined into one object once their code is stable.

The message packer object packs up the message and the arguments into a transmittable format, transmits it, waits for the reply and returns the result of the message. To avoid wasting computer time, the wait for a reply is implemented as a wait on a semaphore. This suspends the context of execution on the local machine while the remote computer evaluates the message. A local background process handles all incoming network data and executes a previously created block when the reply comes in (the reply has the oop number of the block to be evaluated). The block stores away the computation result in a local variable in the sleeping process and then wakes it up.

Besides evaluating incoming messages and distributing returned values, the background receiver process also handles garbage collection. Smalltalk automatically handles garbage collection of unused proxy objects. Objects that have been exported are referenced in a global collection of exported objects, which keeps them from being garbage collected if they are not in use locally. However, exported objects that are not in use locally and not in use remotely still have to be collected.

[Duimovich89] writes about an entry table based multi-processor garbage collection algorithm which is quite comprehensive and more than is needed for a useful system. A simpler technique, described in TABLE 6.1, is used instead. It is similar in that oop numbers are like entry table indices and the movement of referenced objects back into the exported objects collection is a bit like scavenging

1	Move all exported objects from the global exported objects collection into a temporary collection and empty the exported objects collection.
2	Broadcast a garbage collect message, replies from other computers will list the oop numbers of objects on this computer that they are still using.
3	The receiver process handles the replies by adding the listed objects to the global exported objects collection.
4	After a certain long amount of time or when replies have been received from all known computers, the temporary collection of old exported objects is disposed of. Normal garbage collection will now pick up all the previously exported objects that are no longer in the active exported objects collection.

Table 6.1 Garbage Collection Pseudo-code

This technique is flawed in that it doesn t clean up objects in a cycle of references. However, cyclical references are typically not very frequent in objects which are used over the network. A bit of effort on the part of the programmer to break cycles when finishing with an object is all that is required to make this work.

It is also flawed in that it needs to be manually invoked when memory is running low. To make it easier for the user, it has been added as part of the normal manual garbage collection method.

One of the nice things about this method is that Smalltalk can continue to run while the garbage is being collected. Any new exportations of an object merely add it to the fresh global exported objects list.

6.3. EXPORT FORMAT

Export format is the topmost layer of protocol between Smalltalk objects and the network hardware. It converts the Smalltalk objects found in message arguments into simpler Smalltalk objects which can be copied over the network.

In the distributed environment it is desirable to have local objects behave exactly like remote objects. This implies that only objects that are indistinguishable from their copies

should be sent. In Smalltalk, the indistinguishable property is tested with the == operator, which compares object pointers or oop numbers to see if the arguments are the same object. Objects that have indistinguishable copies are nil and instances of classes Boolean, Character, SmallInteger and Symbol.

Performance of a system with just simple objects is very slow. Every time a string is printed, Smalltalk sends a sequence of at: messages to the string proxy object to get each character of the string. It would be much more efficient to transfer strings across the network in one message rather than a message for each character. Thus String and Float class objects are added for efficiency reasons. They are also relatively safe to use as simple objects since most operations on strings and floats return a new string or float rather than modifying the existing object. Proxy techniques would be needed if modifying the existing object was a common operation.

Objects that aren t simple objects have to be handled by sending enough information to create a proxy at the receiving end. This is accomplished by transmitting an array of numbers, which include the oop number and network address. On reception the array s data is used to construct a proxy.

TABLE 6.2 summarizes the export format for all object classes.

Object	Export Format of Object
nil, Character, String, Boolean, SmallInteger, Float	Atomic objects stay as themselves.
Symbol	Symbols become an array with a type code for symbol and the equivalent string.
Proxy objects	Proxies become an array with the type code for proxy followed by the network address and oop number of the real object.
Ordinary objects	Normal objects become an array with the type code for proxy, the network address of the object and the oop number of the object.
Copy holders	These are a special type of object that sends a copy of its contents when sent over the network rather than becoming a proxy. They are used for explicitly transmitting copies of large relatively simple objects (like arrays of numbers) where a proxy would be inappropriate. They become an array with a copy holder code and object being copied (must be simple).

Table 6.2 Export Formats for all Object Classes

On reception, only arrays are processed. Other simple objects (nil, Character etc) are received as themselves. The array s first element is a numeric code that identifies what kind of object is stored in it. If it is a copy holder object, then the second element of the array is returned when the array is converted from export format back to an object. If the code is for a Symbol then the second element of the array (a String) is converted to the equivalent Symbol and returned.

The only complexity arises when converting proxy objects in export format to real objects. If the network address is the same as the address of the computer, then the real object is found from the oop number and returned. If the network address is different, then a new proxy object redirector / packer pair is created and initialized for the received oop number and network address. To save memory, a search for an identical existing proxy redirector / packer pair could be done. However, the search isn t necessary since as far as Smalltalk can tell, the proxies are all identical to the real object and thus identical to other proxies for the same real object.

6.4. SIMPLE OBJECTS

The next protocol layer encodes the previously described simple Smalltalk objects for transmission over the network. In particular, their type as well as their values need to be encoded so that the receiver can recreate them.

Arrays of simple objects are allowed for grouping reasons. It is a lot easier to use languages that have structures or records than languages with no grouping techniques. The same thing applies to sending information between computers. For greater flexibility, arrays of arrays are desirable. This saves having to explicitly encode length counts when a Smalltalk system wants to include a collection of things (like message arguments) to be sent.

C primitives were written to pack and unpack Smalltalk atomic objects from a binary representation. The representation used is a record containing two values:

¥ Field type. This is a numeric code that identifies the type of data being used. It can specify one of nil, Boolean, Character, String, ByteArray, Integer, IntegerArray, Float, FloatArray, generic array of any of these objects.

¥ Field contents. This is the value of the object. For variable size arrays, it includes the array size as part of the value. Generic arrays can recursively include representations of nested objects.

6.5. DATA TRANSMISSION

The next layer between Smalltalk and the network hardware is used for transmitting blocks of data between computers. The result of sending a message to a proxy object is an array of simpler Smalltalk objects which is then mapped into a binary representation. The binary representation is usually short (a few hundred bytes) but can be arbitrarily large when long strings or arrays are being sent. The data block transmission layer handles segmentation and transmission of the binary representation over the AppleTalk network.

Besides fulfilling the requirement for sending large blocks of data, the protocol used has a few extra features, listed in TABLE 6.3.

¥	Broadcast of large data blocks, with handshaking.
¥	Low overhead: messages are received shortly after they are sent.
¥	Receivers identified to sender.
¥	Time division multiplexing for multiple sends in progress.

Table 6.3 Extra AGMS Block Protocol Features

The broadcast feature could be useful for synchronizing the timing of multiple processors. Low overhead helps to reduce time lag between user s motions and the graphic display feedback. Receiver identification can be used to detect network failure or computer failure. Multiplexing makes multitasking more useful by letting short messages get through while a long message is also being sent.

The data block protocol is implemented with C primitives that are invoked by Smalltalk. To keep things alive, a background Smalltalk process periodically calls the C code which reassembles the data packets from the computer s receive buffers and queues up packets for data that is being sent. While Smalltalk is doing other things, operating system interrupt routines handle the lower level DDP (Datagram Delivery Protocol) and hardware interfacing to send and receive data packets.

The actual data block transmission protocol is loosely based on the ZMODEM [Forsberg88] data transmission protocol. In the ZMODEM protocol, the sender transmits

data without waiting for acknowledgement from the receiver. When the receiver detects an error, it asks the sender to go back and re-transmit from the last point where there was good data.

Keeping track of the separation between valid received data and unreceived data with a single position pointer is inefficient. Valid data received after an error is wasted. In a network environment, the order of reception of data packets can be different than the order of transmission due to network delays and changing routing paths. This makes the single separating position method even worse. The ideal solution would be to keep track of the data which has and hasn't been received on a byte by byte basis. With appropriate data structures the ideal solution is practical.

As an implementation note, a set based on range intervals meets the needs of the ideal solution. Instead of storing each element in the set, a list of ranges is used. For example the set {6,7,8,9,20,21,22,23,24,25, ,300} can be represented more compactly as two ranges: [6-9] and [20-300]. In the protocol, the set for keeping track of a 50000 byte data block can contain the integers 0 to 49999, one element for each byte in the data block. The presence of an element in the set is used to mark that byte as being received or transmitted as appropriate.

TABLE 6.4 shows a high level description of the protocol and how non-sequential packet assembly fits in. Essentially the sender sends all its information in one burst and then asks the receivers for a list of bytes to be retransmitted. Note that in practice the data is transmitted in packets containing several hundred bytes, not byte by byte.

Sender	Receiver
¥ Send all unsent data. ¥ Ask the receivers to request re-transmission of parts they didn t get. ¥ Mark the data that the receivers missed as unsent. ¥ Repeat until all data has been successfully sent.	¥ Mark all bytes as unreceived. ¥ Assemble incoming data into the appropriate spot in the received block and mark received bytes as received. ¥ Send a re-transmit request for the missing bytes when appropriate. ¥ Repeat until all data has been received.

Table 6.4 High Level Description of AGMS Block Protocol

At a more detailed level, FIGURE 6.5 shows the state diagram that implements the data block protocol. It was used extensively as an aid in generating the C code for the protocol.

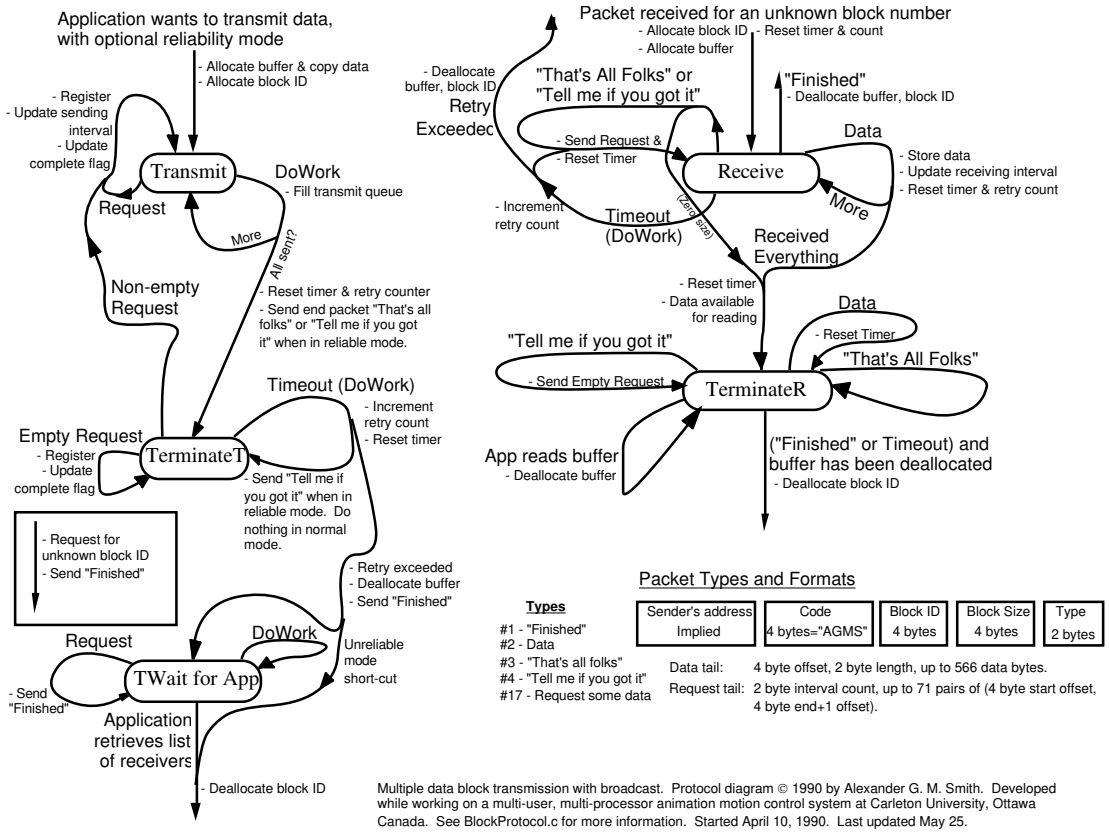


Figure 6.5 AGMS Block Protocol State Diagram

Even though the protocol is efficient under normal circumstances, there are some situations where it breaks down. Most of these situations involve overloads of one kind or another and can lead to even worse network performance due to needless retransmissions.

When the network is busy, the sender will not be able to transmit for long periods of time. If no packets are received for too long a time, the receiver will time-out and deallocate itself. When a long delayed data packet arrives, a new receiver is created that will request re-transmission of the whole data block. This results in multiple useless re-transmissions of the data block, adding to the loading on the network. It is possible that the sender will never finish sending in such a situation.

The same problem of useless re-transmissions occurs when the sender s network software isn t active for long periods of time. This can happen on the Macintosh during long disk accesses.

One implementation limitation is that the search for a buffer to copy an incoming packet into is a sequential search. If there are many different blocks being received, the protocol implementation will be significantly slowed down.

Another minor flaw is that the list of receivers isn t guaranteed to be complete. It is possible to passively receive a data block without ever sending any packets to the sender. On a busy network, the re-transmit request packets that identify receivers to the sender may also be lost. To improve the chances of getting a complete list of receivers, there is a special reliability mode in the protocol. Instead of sending a That s all folks message when all data has been sent, the sender can send a Tell me if you got it message. Upon receiving that message, all the receivers reply with a re-transmit request (possibly empty), rather than the usual behavior of ignoring that message unless they need a re-transmission.

In an optimized test (Smalltalk code busy waiting for received data, directly sending simple Smalltalk objects, not using proxies), the protocol can send or receive between 40 and 50 short messages per second over the local AppleTalk network. Performance for sending large blocks of data is very close to the peak network capacity (30,000 bytes per second) due to the low transmission overhead. This peak performance is so demanding of the Macintosh architecture that all receiving computers slow to a stop, which can annoy all Macintosh users on the network during a broadcast.

6.6. PACKET PROTOCOL

The messages transmitted between proxy objects and real objects eventually find their way down to the network packet protocols and then to the network hardware. The choice of a good low level protocol to serve as a building block for the rest of the system isn't obvious.

The AGMS Block Protocol is built on **DDP** (Datagram Delivery Protocol) [Apple86]. This protocol lets you send packets of up to 586 data bytes to a particular network address. The address specifies a network zone (**zones** are clusters of computers connected together on a common bus network, a bridge device copies packets between zones), machine and software entity number, allowing addressing to a particular destination program or function

on a particular machine. Messages can also be broadcast to all machines in one network zone. Transmission errors are handled by ignoring packets with a bad CRC (cyclic redundancy check).

An alternate candidate that was considered as a base for the Networked Smalltalk system is **ATP** (AppleTalk Transaction Protocol). This level builds on the DDP packets by adding handshaking for re-transmissions and delivery exactly once. The amount of data sent in a packet is reduced further to 578 bytes for requests. Responses can have up to 8 ATP packets (each identified by a bit in a byte). Thus responses can contain 4624 bytes of user data.

Several things are missing from the standard AppleTalk protocols. The main difficulty is that none of them support transmission of arbitrarily large chunks of data. Another minor problem is that the protocol layers above DDP can t do broadcasts. This can be worked around by sending several individual transmissions and by synchronizing local clocks separately.

Each Smalltalk message maps nicely into an ATP request (the message sent to the remote computer) and a response (the result of the message evaluation). ATP would be the ideal solution if it supported arbitrary data sizes and broadcasting.

An alternative is to break down large chunks of data into smaller blocks that ATP could send. This isn t too good since it requires a response handshake from the receiver for each block of data.

The chosen solution was to use DDP and a custom handshaking method (the AGMS Block Protocol). DDP was selected because it provides CRC error detection, broadcast capability, routing between networks and has a low overhead (no handshaking).

After being bundled up in a DDP packet, the data is sent as an **ALAP** (AppleTalk Link Access Protocol) frame over the local bus style network hardware. It uses a CSMA/CA protocol (Carrier Sense Multiple Access with Collision Avoidance) to handle collisions between computers trying to simultaneously transmit on the network. There is a slight bit of overhead from request to send and clear to send control frames that are sent before transmitting the data frames (except broadcasts). The standard LocalTalk network hardware is a 230,400 bits per second serial port (with support for carrier sense) connected by a bus network to all other computers in the same zone.

6.7. SUMMARY

Networked Smalltalk is a success even though some Smalltalk code won t work when networked. For example, testing if some remote object s class is in a local class hierarchy will always fail because the hierarchies are duplicated, not shared. However, most code does work. It is quite rewarding to be able to open a window to inspect an object on some other computer.

Networked Smalltalk can be used with very little extra programming effort because proxy objects behave just like ordinary objects even though they are on another computer. More programming design effort is needed when real-time performance is desired to avoid ping-pong chains of proxy messages, particularly when accessing arrays.

Smalltalk shows its strengths here in the easy addition of proxy objects. With most other languages, proxies or remote procedure calls wouldn t be as easy to implement or wouldn t be as transparent.

The AGMS Block Protocol works well as a primitive in Smalltalk. Sending large amounts of data is done quite efficiently. However, it degrades when there are many different data blocks being sent or received due to a slow algorithm for locating the state machine corresponding to each incoming packet. Other problems with network overload rarely crop up in practice.

The C source code for the protocol has been uploaded to various commercial information services where people have found it to be reasonably interesting (downloaded over 100 times so far from CompuServe and BIX).

Most importantly, Networked Smalltalk makes distributed HandyMan possible.

Chapter 7

7. CONCLUSION

This thesis has presented a description of the Handyman motion control system and supporting subsystems, including an innovative graphics subsystem and a unique communications protocol. Handyman s configuration wiring diagram and the Handyman system as a whole also represent steps forward in computer puppeteering. In more detail the important results of Handyman are:

- ¥ Remote references and other novel extensions to standard graphical structures.
- ¥ A caching system with an unusual use of version numbers for change updates.
- ¥ An improvement to data flow diagrams that adds asynchronism: the configuration wiring diagram.
- ¥ A custom networking protocol that allows broadcasts using disordered arbitrarily sized packet delivery.

The graphical puppet being animated is made from a directed acyclic graph of primitive objects that are grouped and moved by parent transformation objects. While path names to identify a particular object from among multiple instances aren t new, wildcards and path modification strings are a novel adaptation of concepts from file naming systems. Remote references are a new extension to the typical graph structure to allow the use of objects from foreign coordinate systems. They solve the problems of flexible surfaces that need to automatically cover a moving joint and are useful for making volumes of revolution that aren t merely a set of polygons generated by some external program. The result is a graphics system suitable for jointed character animation.

The graphics structures are supported by a unique object oriented caching system that takes advantage of stationary objects to reduce the time for drawing a sequence of animated images. The caching system matches the complexity of the advanced directed acyclic graphical structure by unrolling it into a tree that separates cached data by coordinate system. Partial cache invalidation after window resizing and other partially destructive operations is a significant feature that allows speedy redraws by reusing cached intermediate values. A simple yet effective change propagation mechanism is used to invalidate the relevant graphical objects. A novel use of version numbers provides an

alternative yet efficient and unobtrusive invalidation link between the graphical objects and the cache nodes in the tree. Without caching Handyman would be unbearably slow.

The graphical subsystem is presented to the user as one building block in the larger whole of Handyman s configuration wiring diagram. The diagram combines information generating, processing and storage nodes, to connect the animator s controls to the graphics being animated. The wiring diagram offers a previously unseen improvement on constraint networks and data flow diagrams by adding asynchronism: data can be flowing into a node at a rate different from the outgoing rate. This is made possible by history objects which store time stamped values in a sorted collection and can interpolate or extrapolate the value read from the history at a given time. Active objects, each with its own process running at its own rate possibly on its own computer, pump the data around the network. One of many nice features of this design is that communication and other delays are smoothed over by the time stamps and extrapolation.

Variations of the wiring diagram for a scene are stored as takes. Wiring diagrams for new takes are automatically generated by using the previous one as a template. Unlike more traditional data flow systems, no recompiling or restarting is needed to do a new take. When wiring diagram connections are copied, Handyman objects either carry over into the new take or are replaced by fresh clones. The data recorded in the histories of all takes is persistent and immediately available for reuse in the new take or for the reworking of an old take. This makes it easy for the animator to shoot a scene reusing perfected motions for some objects and doing other motions live. Slow motion recording and a library of object types, including active objects that add previous motion to live delta controls, make fine tuning easy.

A typical session with Handyman uses a top down approach to define motion. First overall spatial motion is recorded live and then perfected over several takes. Next, the spatial motion is frozen and new takes are done during a slow motion playback / recording session to synchronously record squash & stretch and other effects, including texture and colour changes. In the last stage, scene descriptions are created for each frame and rendered with a ray tracer to produce the final high quality textured images.

FIGURE 7.1 shows a screen dump of the user interface for a wiring diagram that connects a dial control to a history object and the X axis rotation of a stack of cubes to the same history. Active objects are squares, with inputs on the left and outputs on the right.

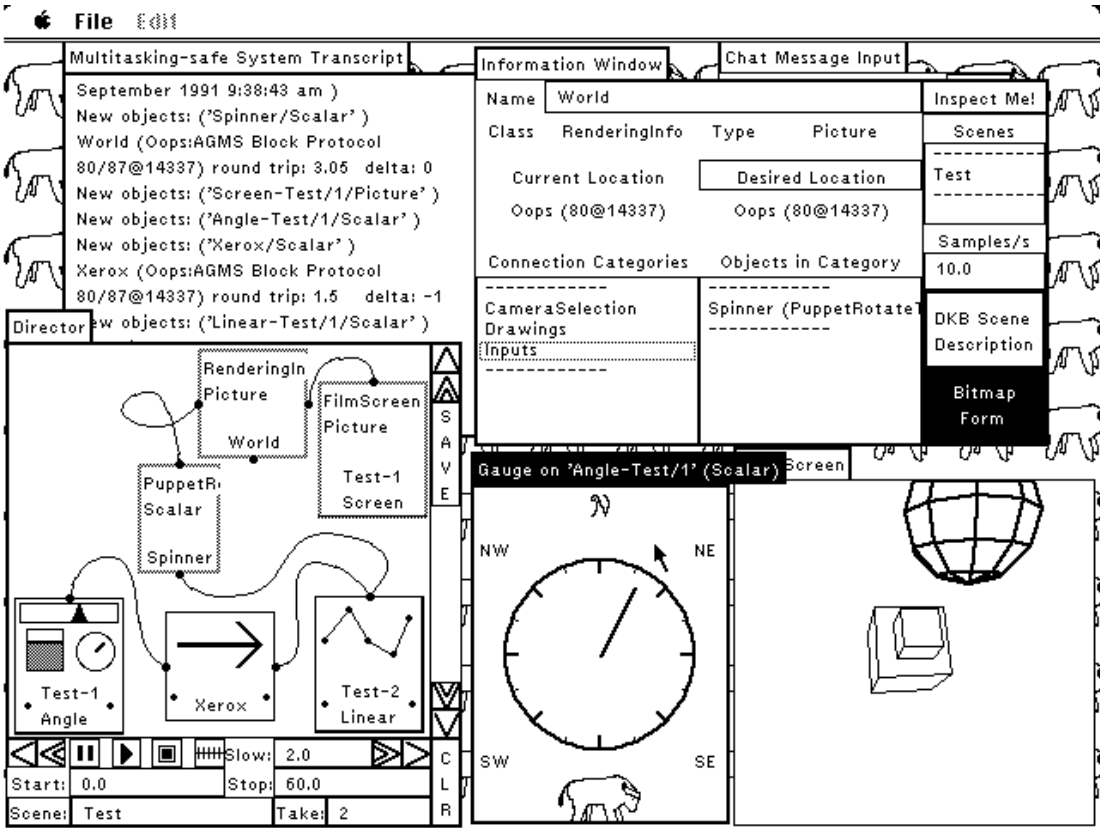


Figure 7.1 Screen Dump of Wiring Diagram User Interface

Under the configuration wiring diagram lies a foundation of proxy objects and network protocols. These make objects on a remote computer seem like local objects, resulting in a much easier implementation of Handyman s distributed configuration wiring diagram. The AGMS block protocol layer is of note for its unusual ability to segment and broadcast large blocks of data in any arbitrary packet order, with arbitrarily varying packet sizes, error correction, and reasonable overhead. Its source code has attracted the interest of over one hundred people who have downloaded it from CompuServe and BIX.

There are many areas for refinement and extensions to Handyman. On the practical side, obtaining a 3D user interface device would be a great improvement over the current 2D mouse input. The possible addition of MIDI compatible controls is also worth

investigating as an economical way of enriching user interaction. The graphics system offers opportunities for better quality through incremental rendering (section 5.5). On the research side, development of caching over a range of times (section 5.4) is a logical extension of the current caching system. Finally, the addition of sound sources and listeners as graphical objects (section 5.6) is worth doing to achieve integration of animation and sound.

The Handyman project leaves behind a legacy of over a megabyte of meticulously commented source code. Various parts (Skip Lists, Proxy Objects, AGMS Block Protocol, Matrices & Vectors, 3D Graphics) are available for reuse by other students.

To sum up, one view of Handyman is as an integrated system of many components that convert raw motion into stunning animation. Handyman is also a system that permits several animators to make believe on a shared stage. Easily done multiple takes combined with good hand generated motion permit the creation of cartoons that come to life.

All in all, Handyman is a nice change from the mechanistic motion of the flying logos that dominate computer graphics today. As Neil Armstrong never said, One giant leap for Handyman, one small step for Puppeteering!

APPENDIX A: UPDATED STATE DIAGRAM

Various enhancements were made to the AGMS Block Protocol for increased speed after experience with the original version pointed up a few slow spots. In particular, reliable mode was removed and acknowledgement of transmission was used to terminate senders before their time-out period is reached. This cuts down on the time for a transmission to be completed and saves transmitting a packet or two. FIGURE A.1 shows the new state diagram.

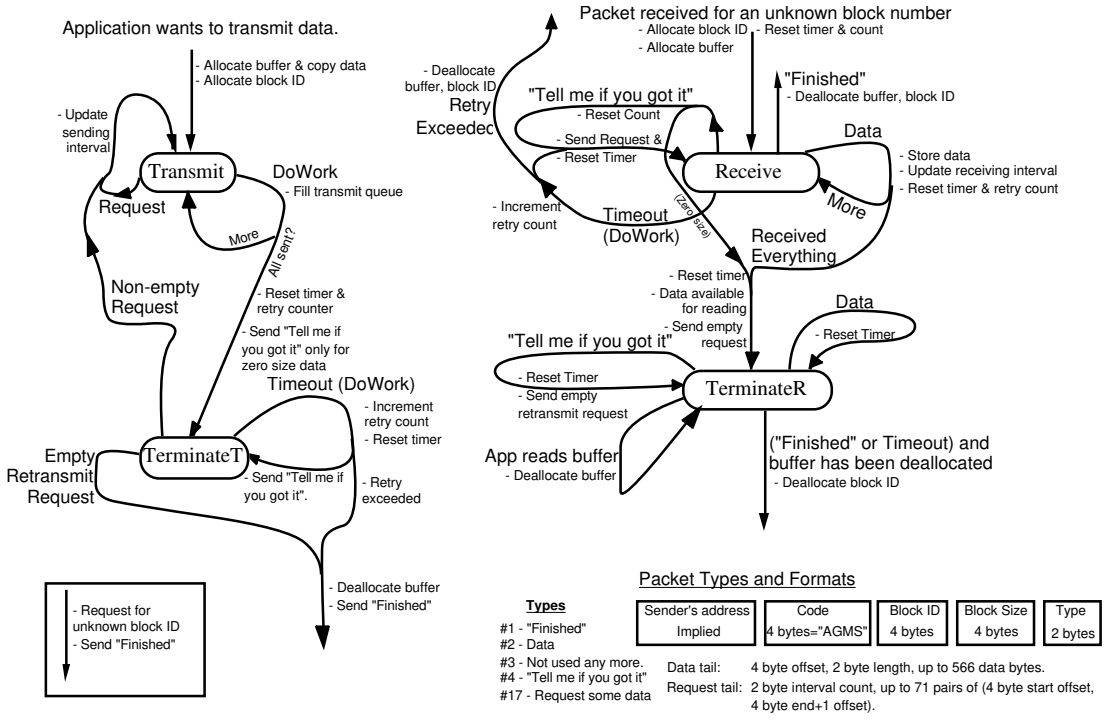


Figure A.1 Updated State Diagram

REFERENCES

[ANSI88] American National Standards Institute, Inc., Programmer s Hierarchical Interactive Graphics System (PHIGS), Document X3.144-1988, September 26, 1988.

[Apple86] Apple Computer Inc., *Inside AppleTalk,* Apple Programmer s and Developer s Association, Renton WA, 1986.

[Bergman86] Larry Bergman, Henry Fuchs, Eric Grant, and Susan Spach, Image Rendering by Adaptive Refinement, SIGGRAPH 86 Proceedings, published as *Computer Graphics,* 20(4), August 1986, pp. 29-37.

[Buck90] David K. Buck, *DKB Trace 2.0 source code,* various places including anonymous FTP from alfred.ccs.carleton.ca (134.117.1.1), 1990.

[Duimovich89] John Duimovich, *Garbage Collection in a Multiprocessor Smalltalk System,* Master s thesis, School of Computer Science, Carleton University, 1989.

[Foley82] James D. Foley, and Andries Van Dam, *Fundamentals of Interactive Computer Graphics,* Addison-Wesley Publishing Company, Reading Massachusetts, 1982.

[Forsberg88] Chuck Forsberg of Omen Technology Inc, *ZMODEM source code version 2.02,* various places including the UNIX file area on BIX, 1988.

[Girard87] Michael Girard, Interactive Design of 3D Computer-Animated Legged Animal Motion, *IEEE Computer Graphics and Applications,* 7(6), June 1987, pp. 39-51.

[Hanrahan85] Pat Hanrahan, and David Sturman, Interactive Animation of Parametric Models, *The Visual Computer,* 1(?), 1985, pp. 260-266.

[Iwata90] Hiroo Iwata, Artificial Reality with Force-feedback: Development of Desktop Virtual Space with Compact Master Manipulator, SIGGRAPH 90 Proceedings, published as *Computer Graphics,* 24(4), August 1990, pp. 165-170.

[Panne90] Michiel van de Panne, Eugene Fiume, and Zvonko Vranesic, Reusable Motion Synthesis Using State-Space Controllers, SIGGRAPH 90 Proceedings, published as *Computer Graphics,* 24(4), August 1990, pp. 225-234.

[Schreiber89] Ghostly Schreiber, Goopy Real Time Muppets, *Animation Magazine,* Winter 1989, pp. 56-57.

[S¿renson89] Peter S¿renson, Felix the Cat - Real Time Computer Animation, *Animation Magazine,* Winter 1989, pp. 54-55.

[Thomas87] Dave Thomas, Wilf Lalonde, and John Duimovich, *Efficient Support for Object Mutation and Transparent Forwarding,* Report SCS-TR-128, School of Computer Science, Carleton University, 1987.

[Wilhelms88] Jane Wilhelms, Matthew Moore, and Robert Skinner, Dynamic animation: interaction and control, *The Visual Computer,* 4(6), 1988, pp. 283-295.

[Witkin88] Andrew Witkin, and Michael Kass, Spacetime Constraints, SIGGRAPH 88 Proceedings, published as *Computer Graphics,* 22(4), August 1988, pp. 159-168.