The Virtual Hand Laboratory Architecture

Valerie A. Summers

December 15, 1999

Abstract

The Virtual Hand Lab (VHL) is an augmented reality environment for conducting experiments in human perception and motor performance that involve grasping, manipulation, and other complex 3D tasks that people perform with their hands. Our system supports a wide range of experiments and is used by (non-programmer) experimenters.

Our system co-locates the hand and the manipulated objects (whether physical or virtual) in the same visual space. Spatial and temporal accuracy are maintained by using a high precision tracker and an efficient implementation of the software architecture which carefully synchronizes the timing equipment and software. There are many issues which influence architectural design; two of which are modularization and performance. We balance these concerns by creating a layered set of modules upon which the application is built and an animation control loop which cuts across module boundaries to control the timing of equipment and application.

Augmented objects are composed of both physical and graphical components. The graphical object inheritance hierarchy has several unusual features. First, we provide a mechanism to decouple the movement of the graphical component of an augmented object from its physical object counterpart. This provides flexibility in the types of experiments supported by the testbed. Second, we create subclasses based on properties of the physical component of the augmented objects before creating subclasses based on the virtual components. Specifically, we categorize physical objects as either rigid or flexible based on their level of deformation. This allows us to efficiently implement many of the manipulation techniques. Third, after subclasses based on the physical objects have been created, the implementation of concrete virtual object classes is driven by the goal of creating an easy interface for the experimenters. This was based on our user centered design approach.

1 Introduction

Augmented reality (AR) environments dynamically "augment" the user's view of the real 3D world with computer-generated virtual objects. These virtual objects may overlay physical objects, creating hybrid objects, or the virtual objects may exist independently of the physical objects but still interact with them through behaviors that can differ from those of physical objects. For example, virtual objects can pass through each other or physical objects.

The Virtual Hand Lab (VHL) is an augmented reality environment for conducting experiments in human perception and motor performance that involve grasping, manipulation, and other complex 3D tasks that people perform with their hands. The advantage of an augmented environment is that experiments can explore the limits of human perception and motor performance by simulating situations that would be difficult (even impossible) to create using the purely physical apparatus traditionally employed in studies of this type.

The primary users of our experimental system are the kinesiology researchers who design experiments, recruit subjects, run the subjects through the experiments, and perform post-hoc data analysis. The subjects are the ones who actively interact with the environment.

We have defined the term *direct hand manipulation* to mean grasping, transporting and orienting objects directly with one's hands, where the hand and the manipulated object (whether physical or virtual) lie in the same visual space. This in in contrast to *direct manipulation* which refers to graphical user interfaces (GUIs) and *natural manipulation* which can be confused with non-augmented reality applications, or *tangible manipulation* which is used by some AR groups, but does not include manipulation of purely virtual objects. GUIs separate the controlling objects (mouse and keyboard) from the controlled object (the screen). Tangible manipulation also does not require the controlled and controlling objects to be visually co-located, although the controlling objects are generally more elaborate than the GUI controllers. For example, a doll's head might be used to manipulate a virtual skull. Tangible manipulation by definition also excludes manipulation of non-tangible objects, such as purely virtual objects.

Simultaneously meeting the the requirements for an experimental testbed and augmented reality has proven to be challenging. We support direct hand manipulation with visually and temporally accurate displays. Our system supports a wide range of experiments and is used by (non-programmer) experimenters.

There are many issues which influence architectural design; two of which are modularization and performance. In Section 3 we show how we balance these concerns by creating a layered set of modules upon which the application is built and an animation control loop which cuts across module boundaries to control the timing of equipment and application.

We support direct hand manipulation through co-location of virtual and physical objects. Co-location is accomplished by using a particular configuration of our physical apparatus and using an appropriate head-coupled stereo perspective transformation (Section 4).

Spatial and temporal accuracy are maintained by using a high precision tracker and an efficient implementation of the software architecture which carefully synchronizes the timing equipment and software (Section 5). Trackers are the physical devices which provide data about the location of markers in the system relative to the tracker.

Augmented objects are composed of both physical and graphical components. The graphical components in the VHL all derive from a common abstract base class. Section 7 describes our inheritance hierarchy which has several unusual features. First, we provide a mechanism to decouple the movement of the graphical component of an augmented object from its physical object counterpart. This provides flexibility in the types of experiments supported by the testbed. Second, we create subclasses based on properties of the physical component of the augmented objects before creating subclasses based on the virtual components. Specifically, we categorize physical objects as either rigid or flexible based on their level of deformation. Third, after subclasses based on the physical objects have been created, the implementation of concrete virtual object classes is driven by the goal of creating an easy interface for the experimenters.

We followed a user centered approach when designing the VHL. Part of this design revolved around the controls available to the experimenter to specify an experiment. The experimenters are able to stipulate the flow of control throughout the experiment through a layered interface. Each layer provides a different layer of control, with increasing levels of detail. Details include descriptions of graphical objects such as size, shape and colour; timing controls for when objects should appear in the environment; and location controls which determine where graphical objects appear and how their locations are modified by physical objects in the environment. Section 8 discusses the architectural support for experimental control.

2 Overview

2.1 Data Flow Overview

An overview of the data flow within the experimental system is provided in Figure 1. Data is obtained from various sources such as the tracker and configuration files, and after appropriate manipulation, is used to update the object models and the perspective transformation. These are then used to render the scene.

There are two major groupings of data: real-time data and static data. The static data is provided to the system at initialization time. It includes output from calibration programs, and user modifiable files. Motion data is obtained from the real-time data. The most important of these is the tracker data. Other sources include models, such as one used to simulate eye motion, and animation files. Animation files specify the motion of virtual objects as a function of time.

Static data includes the output of several calibration programs, and files which are used by the experimenters to customize the environment. Users can independently customize the attributes of the graphical objects, configure the environment and specify in advance the behaviour of graphical objects during an experiment. Static data is validated during system initialization.

The real-time data must be validated as it arrives. For each object which requires real-time data, the appropriate segment of data is selected. Validation is performed on this segment. The real-time data obtained from our particular tracking system has an unusual limitation. The positioning data obtained from the tracker is ill-conditioned when the markers are in particular orientations relative to the tracker. In this case, the tracker returns data which looks *a priori* to be valid data. Upon closer examination though, the data returned may be erroneous by several millimeters. Admittedly this is not a large error, but by comparison to the best that the tracker can produce, it is quite significant. The animation files are logs of pre-recorded tracker data, and are treated the same way.

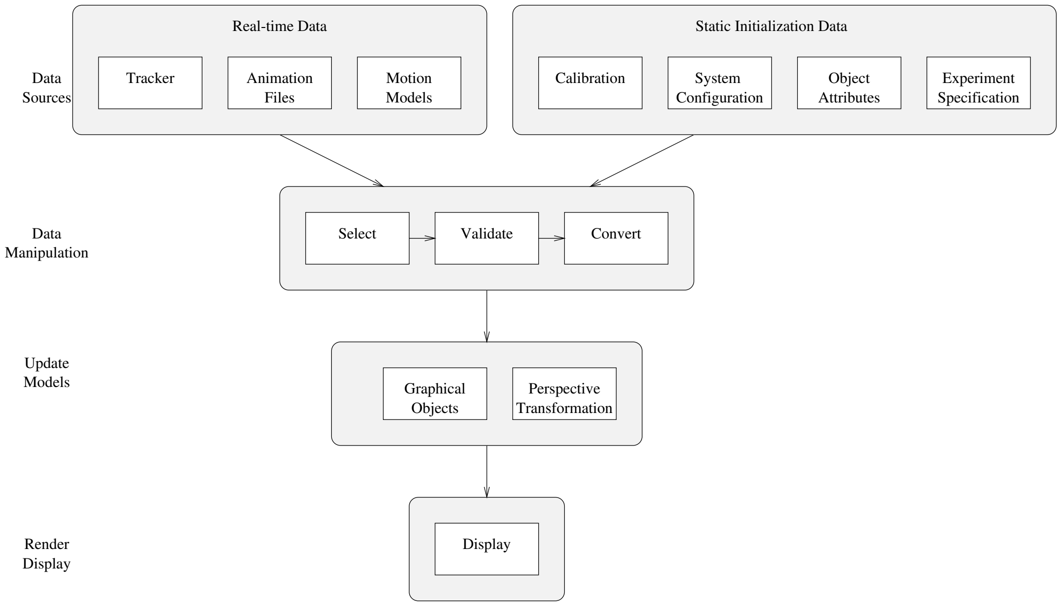

Figure 1: *High level experimental testbed data flow.*

Uncorrected data irregularities cause problems such as "jitter" in graphical displays. 3D displays use data to position objects and also to render the correct point of view. Jitter is especially a problem for 3D displays as people are extremely sensitive to jitter in the point of view. When we identify erroneous data discard it and instead use the last valid data obtained. This increases the lag for some objects, but does reduce the jitter. A recommended system extension is to use predictive models to estimate data when the actual data is invalid.

Once the selected real-time data has been validated, it must be converted to the correct coordinate system.

Motion models are programmed to only provide valid data in the correct coordinate system, and hence do not need to be explicitly validated or converted.

The validated, converted data is used to update the graphical object and point of view models. Once all of the models have been updated, a new display can be rendered.

The animation file input has not been fully implemented, but hooks have been provided in the system so that it can be easily added when required.

2.2 Configuration of Modes

The Virtual Hand Lab (VHL) can be run in four different modes as seen in Table 1. The modes are a product of data (real-time tracker data or simulated data) and control flow (experiment or demo).

Modes	Data Source	
	real-time (tracker)	simulator (no tracker)
experiment	experiment system	experiment simulator
demonstration (no experiment)	demo	demo simulator

Table 1: *There are four modes available in the VHL, depending on the combination of the data source and control flow selected.*

The first major division is dependent upon the availability of the tracking system. When the tracker is not available, the VHL can be executed in *simulator mode*. In this mode, all of the motion data comes from initialization data, mathematical models, or the defaults provided by stub simulation routines. This mode is provided to reduce development time. First, it reduces the dependency on the availability of the tracking system. The tracking system and the room it is housed in are heavily used by kinesiology lab members who run experiments. Second, it reduces the dependency on particular libraries. For licensing reasons, tracker libraries are only available on the subsection of the network which supports the tracker (the experimenter's computer). Compiling on this machine while it is being used by others to collect data for an experiment could potentially affect timing rates, which would reduce the internal validity of that experiment. By convention, no one other than the experimenter even logs onto the experimental machine when an experiment is in progress. An added benefit of the simulator mode is that most development can (and has) taken place not only on a different machine, but at a different location than the tracker. When the tracking subsystem is available, the VHL can be compiled with the tracker libraries and executed in *real-time* mode.

Both simulator and real-time modes can be subdivided based on whether an experiment is selected. In *experimental mode*, The overlying flow of control is determined by the experiment. Experiments are composed of trials and phases which will be discussed in detail in Section 8. Often for demonstration purposes or to test a new object, we do not wish to proceed through an experiment. We refer to this mode as *demo mode*.

The division between simulator and real-time modes is determined at compile time since it is dependent on the availability of libraries. The experimental verses demo mode decision is made during system initialization. The core of the system is a small timing loop which animates the scene. In demo mode, the animation loop is just repeated. In experimental mode, other layers of control are overlaid (Example 1).

```
if (experiment mode)
   for (every trial in that experiment)
      for (every phase in that experiment type)
         call phases
            Animate Scene
else (demo mode)
   while (not done)
      Animate Scene
```

Example 1: *The VHL can be run either in experimental mode or demo mode.*

3 Modularity

3.1 Separation of Concerns

Like most large systems, the VHL has been developed as a layered set of modules. Figure 2 shows a conceptual breakdown. The original design process placed a small percentage of the methods in a global arena to speed up development and to ensure performance by reducing data transfers and function calls. Currently, these methods are being refactored into the appropriate module. This is a straightforward process as modularization was always considered (even if not implemented) during development. Modules which have not yet been fully refactored and currently exist in the global arena are marked with an asterisk (*).

Each module (with the exception of the application) uses only modules in the layer beneath it, making it a pure layered system. The bottom layer is the foundation layer and include external libraries and support utilities. The second layer is the data layer. The third layer deals with data intraction. The fourth layer is the application. The application violates the pure layered system. It cuts across layer boundaries for performance reasons which will be discussed in Section 3.2.6.

If we compare Figures 2 and 1, We can see the similarity. The data sources in Figure1 are managed by the software modules in the Foundation layer in Figure 2. Data manipulation in Figure1 corresponds to the data layer in Figure 2. Updating the models is done by the GraphicalAgents in the Interaction layer. Rendering the display is controlled by the method AnimateScene in the Application layer.

Graphics Library The graphics library used is IRIS GL, although a new version of the system will be ported to OpenGL.

Support Utilities Some basic classes have been grouped together to facilitate the construction of other modules or applications.

Tracker Library Custom software provided by the tracker manufacturer.

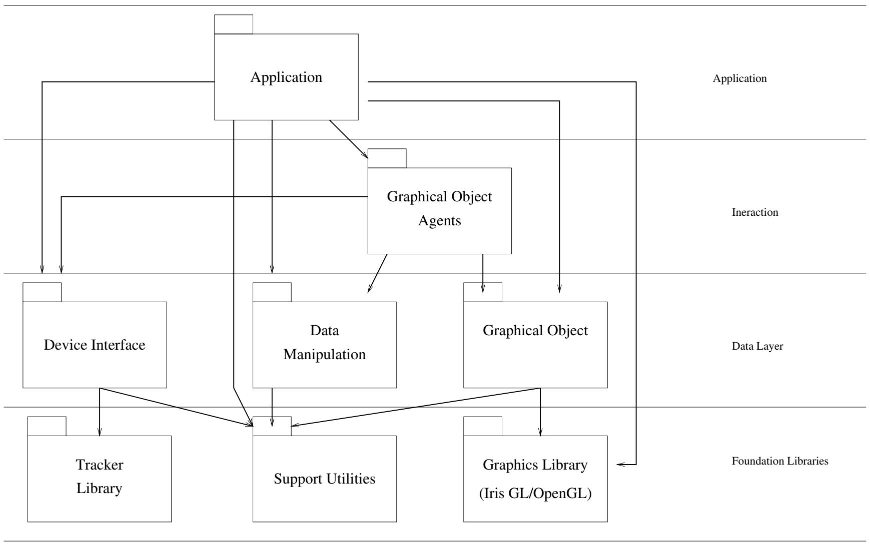

Figure 2: *The VHL is composed of several modules upon which the application is built.*

Device Interface(*) Interactions with specific devices are grouped here, although the subclasses themselves do not interact with each other.

Data Manipulation These methods are given data which they validate against metrics produced by calibration procedures. They then transform the validated data into the appropriate coordinate systems and data formats.

Graphical Objects One change that was anticipated early in the system development was the transition to OpenGL from GL. Consequently, we encapsulate all of the rendering in a common virtual function within a single hierarchy namely the Graphical Object Hierarchy. Data is provided to these objects through an interface, so the objects themselves know nothing of the data source. Specifically, these methods do not depend on the Tracker Interface module.

This module also encapsulates some functionality from the graphics library that is used frequently such as testing for mouse input.

Graphical Object Agents(*) We have defined agents to act as an interface between graphical objects and the data sources. The agents obtain from the graphical objects keywords representing the motion type and motion source. The agents obtain the data from the appropriate source, and pass it back to the objects. The agents are unaware of any specifics of the objects, and thus do not depend on the graphics library. Conversely, since the agents are dealing with the data sources, the objects do not depend on any device libraries.

Application(*) Classes and global methods specific to the application of experimental testbed are placed in this module.

3.2 Class Level Decomposition of Modules

The decomposition of the modules into classes (prefixed with ‘o’) and methods is shown below.

3.2.1 Support Utilities

SbMatrix, SbVec3f Inventor is used strictly as a math library, not for graphics.

oBasis This class represents an orthonormal basis in 3-space plus an origin. We originally tried working with non-orthogonal bases to avoid the performance hit of orthogonalizing them, but found that we lost precision since the vectors were of such different magnitudes. Making all the bases orthonormal solved this problem.

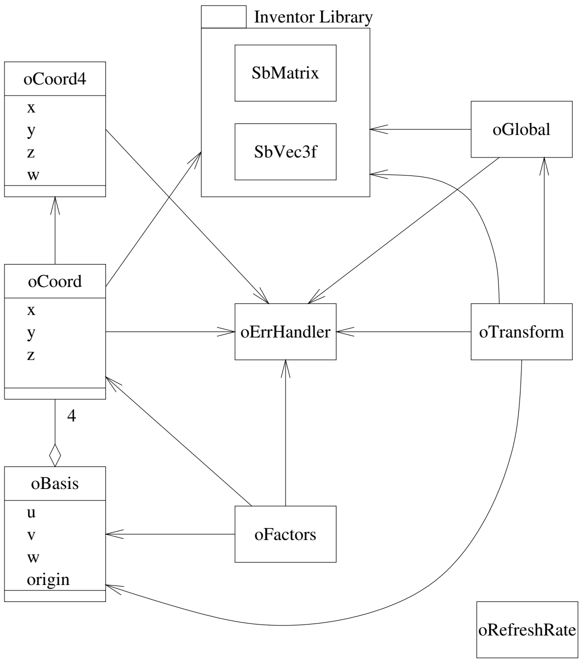

Figure 3: *Support utilities module.*

oFactors One of the main purposes of this class is to estimate the positions of landmarks relative to the positions of other landmarks whose positions are known. For example, it is used to estimate the positions of the left and right eyes relative to markers that track the head of the subject. Mathematically, it is quite simple. Given a basis B and point P in 3-space, it will calculate the vector V representing that point relative to the basis. Conversely, given V and P, it will produce P. This class also converts between a basis and a set of vectors. This class should be merged with oBasis.

oCoord Vector and point manipulation is optimized for three elements (X,Y,Z). Functionality includes: mathematical functions such as arithmetic operators, dot and cross products, conversion betwen radians and degrees, negation, length, normalization; and conversion between oCoord and SbVec3f class representations. oCoord does not subclass from SbVec3f since it adds some features which are incompatible. For example, direct access to the location member data was permitted for performance reasons although it violates the encapsulation of the class.

oCoord4 Vector manipulation is optimized for four elements (X,Y,Z,W) similarly to oCoord.

oErrHandler Static class functions provide consistent error handling across the VHL system.

oRefreshRate Encapsulates timing tests for one or two devices. This is often used by the application to monitor the data obtained from the tracker to ensure we did not miss any frames of data.

oGlobal Global constants such as maximum line size, name lengths, some additional functions to manipulate Inventor classes and some commonly used memory allocation/deallocation functions.

oTransform This class holds a coordinate system defination. It redundantly stores data for ease of use and performance in oBasis and SbMatrix representations. It is separated from oBasis since we intend to replace the Inventor classes with another library.

3.2.2 Device Interface(*)

oTrackerInterface(*) The tracker interface (will) manage calls to the Tracker Library (not shown) and manage the resulting data. The tracker library returns values for all of the markers in a system for a given frame. The interface provides methods to access subsections of that data and to check that the markers requested were actually visible in that frame. When the tracker is not available, the system can be run in simulator mode. Stub methods simply return static defaults instead of real-time data.

oRS232(*) This class will send signals on an RS232 cable. These pulses can be used to synchronize external devices which have independent clocks.

3.2.3 Data Manipulation

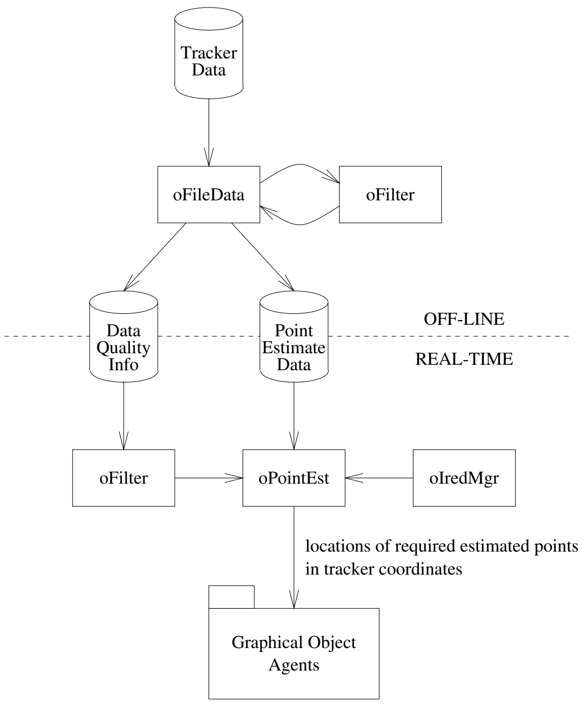

Figure 4: *Data manipulation module.*

oFilter A frame of data consists of a 3-dimensional location for each marker. oFilter provides a set of filters, each of which determines whether a single frame of data is valid according that filter's criteria. Filters are used by both off-line and real-time data manipulation classes. (Hence the addition of some subfilters which can only be used by off-line processing.)

The user of the class can enable the filters desired in any order. We allowed this to make the interface easier to use.

oFileData This class loads a set of data frames (typically from an entire trial) and manipulates it.

An iterator function allows this class to be used as an a animation file, by returning a single frame of data at a time, in the same format as the the real-time data returned by the tracker interface. Data can also be accessed in smaller quantities by specifying a particular marker in a particular frame.

The data set can also be used to produce metrics subsequently used by other programs to validate and manipulate data during real-time execution. The metrics are placed in ASCII files and are loaded by the other programs. A Filter can be applied to the data before it is analyzed and error metrics produced.

The analysis of each data frame results from applying a series of filters to the data. However, the filters are executed in a pre-set, internally defined order. Each filter is applied sequentially to the entire set of frames. First, filters which do not depend on pre-processing by other filters are executed, followed by the dependent ones. Each frame of data is tagged as valid or with an error code indicating which filter the data failed.

The only device dependent function is “load” which depends on a particular data format. To expand this class for multiple devices specialized load functions can easily be added or the data could be preprocessed by another method.

oIRedMgr Manages the usage of ireds which are the markers in our system. Some markers form a rigid body which is used for estimating other points. Other markers are those points that will be estimated in later steps without markers.

A single marker may be part of multiple rigid bodies (such as hinged objects), and a marker may also be estimated from several bases (if we have redundant data). For example, we could have:

RigidBody 0:	marker 0, marker 1, marker 2
RigidBody 1:	marker 1, marker 2, marker 3

and marker 4 could be estimated from either RigidBody 0 or 1, while marker 5 is estimated only from basis 1.

oPointEst Estimates points for missing ireds. For example, calculating the right eye position based on markers located on the side of the subject’s head.

An object interaction diagram is shown in Figure 5. An instance of oFileData loads a set of data frames produced by the tracker. It validates these frames using a Filter. The FileData instance outputs a set of ASCII files. These files are used to initialize a new filter instance. A point estimation instance uses this new filter, some of the other validation files, and an ired manager to create estimates of the left and right eyes based on information from the markers tracking the subject’s head.

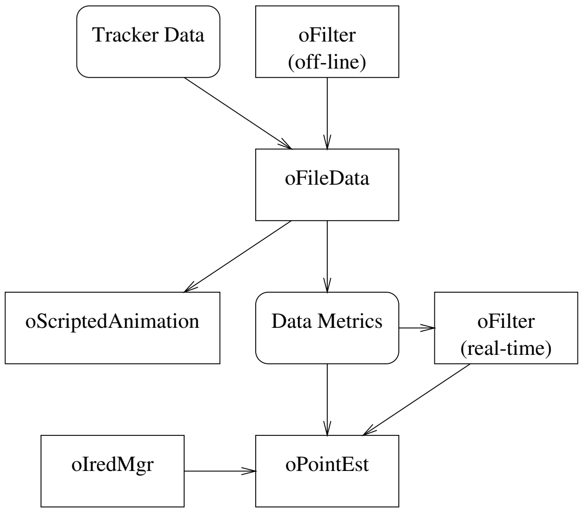

Figure 5: *Filters can be used by FileData classes to process off-line data or by other methods such as oPointEst to validate real-time data.*

3.2.4 Graphical Objects

oGrObject This is the abstract base class for all graphical objects. There are many derived classes. The entire hierarchy is described in Section 7.

oColl A collection of graphical objects.

oPerspective Based on knowledge of the monitor dimensions in pixels, the distance of the mirror to the monitor etc., this class creates the correct perspective transformation, clears buffers, sets viewports and prefpositions.

Misc. C methods(*) Some commonly used methods such as testing and reseting the mouse and reseting the display are encapsulated here.

3.2.5 Graphical Object Agents(*)

These classes act as a buffer between the graphical objects and the other modules (except for the graphics library). Interaction between this module and the graphical object module is minimized. The Update methods each only know about a single class in the graphical object module. This means that as we add new graphical objects, there are no changes required of this module.

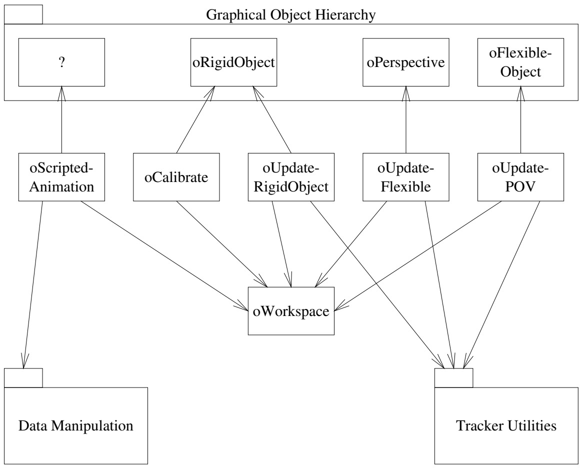

Figure 6: *Graphical object agents module.*

oWorkSpace Takes care of the transformations between the tracker coordinate systems and workspace coordinate systems.

oCalibrate This class calculates the transformation needed to locate the virtual component of augmented rigid objects.

oScriptedAnimation(*) Uses pre-recorded data to simulate movement of virtual objects. This is not fully implemented.

UpdateRigidObjects(*) Updates rigid object locations based on movement source and movement type of rigid objects (which are described in Section 7.)

UpdateFlexible(*) This method updates object locations for the for the only flexible object we have in the experimental system, apointer.

UpdatePOV(*) This updates the point of view for the perspective transformation.

3.2.6 Application Objects(*)

A complete description of the interaction of these classes is found in Section 8.

In Figure 2 it appears appears as though the application interacts will all layers. In fact, only one method, AnimateScene, cuts across all module layer boundaries. DrawComponents interacts with two layers, the Foundation Layer and the Data Layer. This was done for performance reasons.

oPhase(*) The oPhase class contains a set of independent phases.

oExperiment(*) The type of experiment determines which phases are executed, and in which order. The interaction of phases and experiment type is reminiscent of file data and filters.

oTrial This class loads a trial specification file, and provides it a trial at a time to the experiment. It provides record keeping for which trial controlled graphical targets should be displayed, although it only knows about the targets by type.

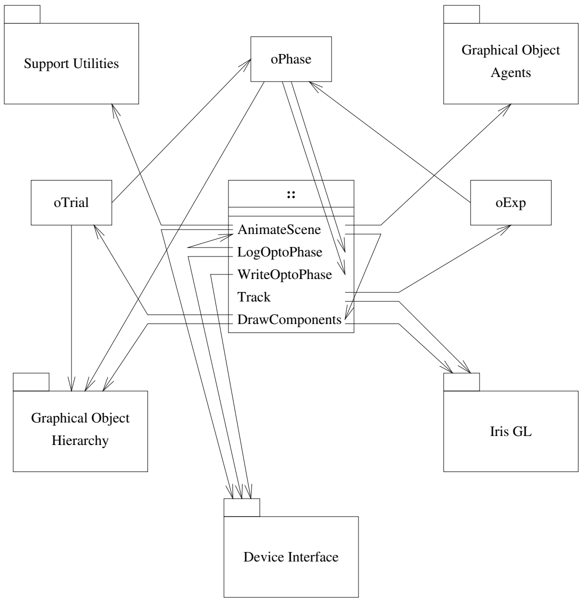

Figure 7: *Application module.*

LogOptoPhase(*) This method is called by the phases in the application object module. LogOptoData calls AnimateScene and logs the tracker specific data until the phase transition criteria is encountered.

WriteOptoPhase(*) This method is called by the phases in the application object module. WriteOptoData writes the data log created by LogOptoPhase to a specified file.

AnimateScene(*) This method is the timing loop used to animate the system. A common loop structure for double-buffered animation is to clear the back buffer, update the model or scene, render the scene to the back buffer, and swap the buffers. This is the structure we follow when our software is running in simulator mode, and we do not have live data from a tracker. In the simulator mode, we can provide a left eye monoscopic view, a right eye monoscopic view or a stereo view. Since rendering a scene is composed of setting the perspective and then drawing the scene, drawing in stereo simply does this twice, once for each eye.

Once we start using real-time tracker data, we must add data acquisition to the model. Acquisition of new data must occur before we update the model. We found we could not sustain the necessary update rate when we executed a blocking request for data due to SCSI delays. Consequently, we split the data acquisition into two stages, a data request and a data get. After trying many combinations, we found we got the best performance by requesting the data prior to clearing the buffers, and getting the data immediately afterwards as shown in Figure 8. (Synchronization details will be discussed later in the Section 5.2

DrawComponents It sets the graphics mode and calls the draw routines for all objects in the system. Setting the mode could be encapsulated as a utility in the Graphic Object Module, but for performance reasons was not. Once the system migrates to a faster machine, encapsulating the mode setting would reduce the layers with which DrawComponents interacts to one, the goal of a modular design. DrawComponents is only called by AnimateScene.

Track This method executes a single trial. Based on the experiment type, it sequentially calls the appropriate phases.

4 Co-location of Virtual and Physical Objects

4.1 Apparatus

The apparatus has been previously described in [Summers et al., 1999], but is revisited here for completeness. Text within parentheses indicate definitions that will be used later in the paper.

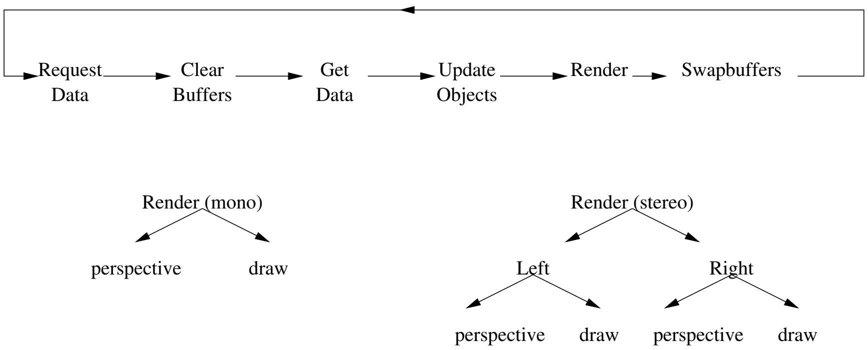

Figure 8: *The core of the system is a small loop to animate the scene. The render step can be broken down depending on whether the view is monocular or stereoscopic.*

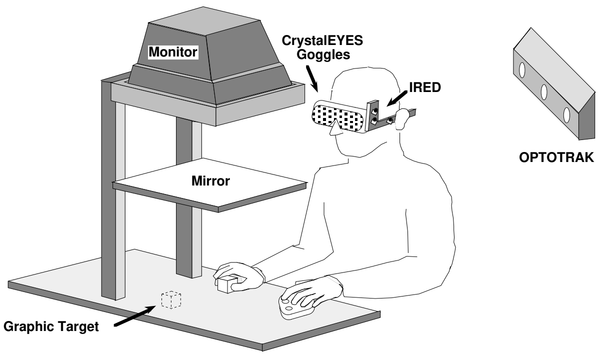

Figure 9: *Physical configuration of apparatus.*

The hardware supporting our augmented environment consists of an SGI Indigo 2 Extreme workstation, a slaved secondary SGI monitor (*monitor*) that is reflected through a mirror, a Northern Digital Optotrak 3D motion analysis system (*tracker*), StereoGraphics CrystalEyes stereo-graphic glasses (*stereo glasses*), and an ATI Force/Torque Sensor System (*force sensor*). The SGI displays stereo images on its regular screen (which is seen by the experimenter) and also on the slaved monitor. The mirror places these images appropriately within the subject's view. The Optotrak senses the 3-D positions of infrared emitting diodes (*markers*) that are strobed under computer control. The markers are placed on all objects (including the subject) whose position or orientation is required during an experiment. A combination of tracking the stereo glasses and calibration procedures previously described in [Summers et al., 1999] enable accurate point of view estimations. These estimations enable the monitor to display head-coupled stereo images for the subject.

Figure 9 illustrates the physical set up. A metal rack on rollers supports the slaved monitor face down over the half-silvered mirror. The rack holding the monitor is built with supports on only one side, which cannot face the tracker, but the other side provides an unobstructed view of the workspace, allowing the markers to be sensed by the tracker. The mirror is mounted in a frame which is attached on one side to the rack. The relative positions of the desk, rack with monitor, mirror, tracking equipment and subject vary according to the experiment.

The monitor itself is awkward and heavy to move, so rearranging it in the rack requires more than one person. Consequently, we modify the orientation of the coordinate system in software rather than by physically moving the monitor. This conversion is performed by the class oWorkSpace in the module Graphical Object Modules. As seen in Figure 10, the orientation of the workspace coordinate system is chosen relative to the side of the desk on which the subject will be seated rather than being associated with the monitor. Defining the coordinate system in this way allows experimenters to place subjects on either side of the desktop, optimizing the use of the limited workspace thus allowing for individual differences such as left and right handedness.

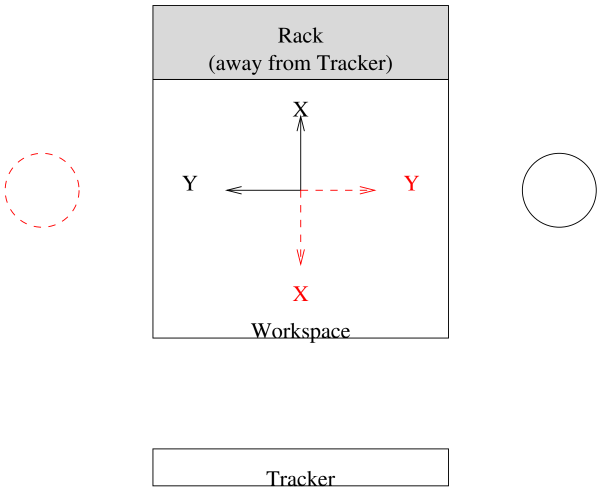

Figure 10: *The X-axis is to the subject's right, while the Y-axis is ahead, independent of which side of the workspace the subject sits. The line type (solid or dashed) shows the coordinate system associated with each subject location.*

The rack is placed so that the monitor and mirror are above a desktop, all parallel to each other, with approximately equal spacing between the face of the monitor on top, the mirror in the middle, and the desktop on the bottom. With this geometry, an image on the monitor is reflected through the mirror and a 2D virtual image appears just above and parallel to the desktop (the relative spacing of the monitor, mirror and desktop determine the exact distance). Viewed without the stereo glasses, this provides a 2D virtual image that floats stably in space just above and parallel to the desktop.

The frame holding the mirror can be moved up or down (Z-axis) with respect to the monitor. This allows us to maintain the equal spacing of monitor, mirror and desktop when the height of the desk changes.

4.2 Perspective Transformation

The workspace is the volume between the mirror and the desktop into which a subject can physically reach. To achieve the illusion of 3D images within this workspace, stereo glasses are used. Different left and right images are sequentially displayed on the monitor providing stereo separation. The class oPerspective in module Graphical Objects create the left and right images.

Figures 11 through 13 show conceptually how this illusion works. In each diagram we consider set of three objects, each drawn at the same 2D position on the desktop, but at increasing heights. Position A on the desktop is the lowest, while C is the highest. Figure 11 shows a side view of the illusion as perceived by the viewer. Notice how the lines joining the eye to the object intersect the mirror and desktop at increasing distances from the viewer.

Figure 12 shows how stereo separation is used to determine where the 2D representation for each eye is drawn. Each 2D image is drawn where the line joining the object and the eye intersect with the projection plane. This is indicated by a small hash mark. For objects drawn on the desktop (the circle at position A), the left and right eye views coincide. As the object moves further from the projection plane (up from the subject's point of view), the left and right eye views diverge as seen by positions B and C.

Figure 13 shows a top view of the same positions. Not only does eye separation increase as the height increases, but the 2D representation also moves away from the eyes and increases in size. This maintains the illusion that the object size is constant while the position changed.

Using a mirror as a "beam-splitter" between the subject and the workspace allows computer augmentation of objects to supersede the visible attributes of physical objects. Physical objects include the physical part of an augmented object as well as the subject's hand. Although not shown in Figure 9, a fluorescent light is mounted on the rack beside the workspace below the mirror. Turning this light on or off determines whether the display is transparent or purely virtual.

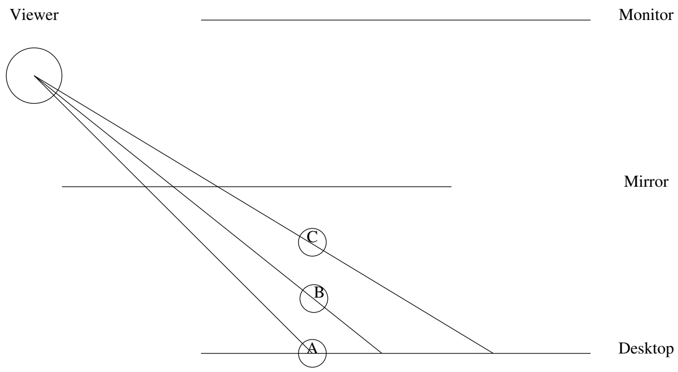

Figure 11: *We simulate 3D by projecting the correct 2D images onto the desktop.*

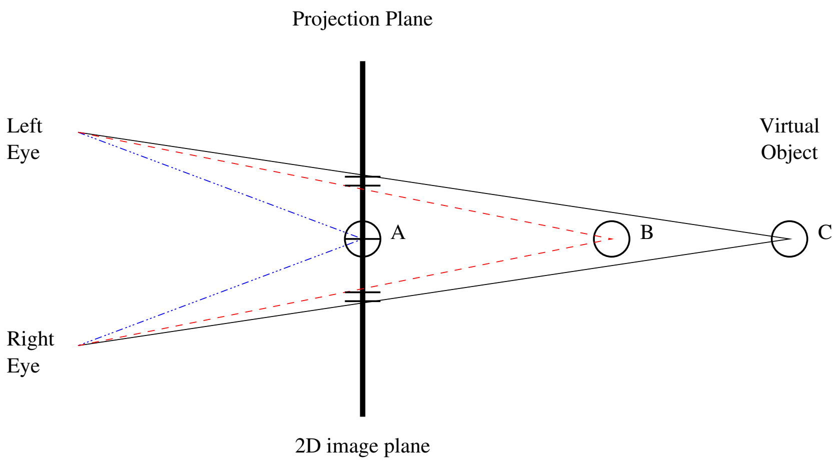

Figure 12: *By separating left and right eye images, we can simulate depth on a 2D display. The 2D image is reflected onto the desktop, so object A appears "lowest" and object C appears "highest".*

If the light is turned off, none of the physical objects are visible, but the virtual ones are unaffected; with the light on, both physical and virtual objects are seen.

5 Equipment

5.1 Equipment Data Flow

A simplified data flow model will first be presented, followed by a detailed equipment level data flow.

At the simplest level, the user views the 3D augmented reality environment and reacts to it by manipulating equipment such as tracked objects, the keyboard or the mouse. Data arising from the manipulation is provided to the application software which uses it to update the visual display. The user views this new display, and the process is repeated. (Figure 14).

In Figure 15) we make explicit the data transmitted though the system. Data can be transmitted in one of three ways: via permanent physical connections, through non-physical means, and through intermittent physical contact. Permanent physical connections include the obvious computer cable connections, but also equipment (such as stereo goggles or markers) that are secured to the user. Non-physical connections include objects viewed by the user and the use of infrared light to transmit data. Intermittent physical contact occurs when a subject touches or manipulates physical objects in the environment, by grasping and releasing them. The physical connections (both permanent and intermitent) can be further subdivided into those that transmit data in the traditional computer science sense of the

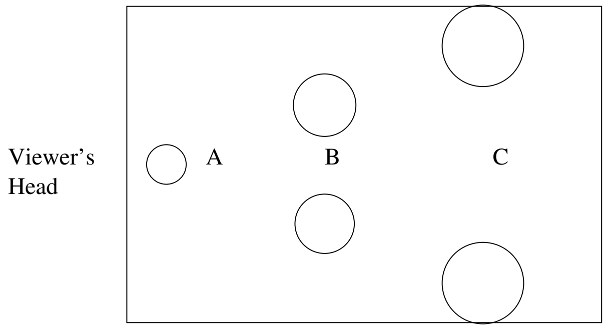

Figure 13: *The top view of the display shows how the 2D representation moves away from the view and increases in size as the height of the object increases.*

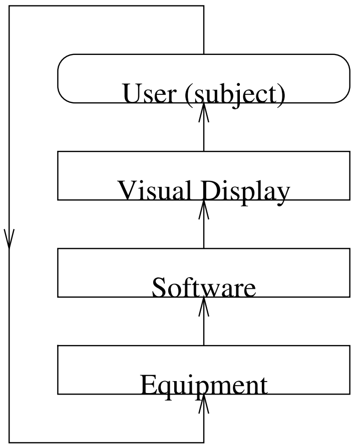

Figure 14: *Based on the visual display, the user manipulates physical apparatus in the scene. The data from these manipulations is fed back to the application, which then updates the visual display.*

word, and those that induce data. Objects which are physically connected to the IREDs induce data. The motion of the objects in a sense create the data recorded by the IREDs. This includes the stereo goggles, the force recorder and other augmented objects in the environment.

It is important to note that the subject is physically connected to the system, which makes it a "tethered" system. Unlike untethered systems, this limits the mobility of the subject. The experimenter is untethered and thus able to roam about the experiment room, or even leave it while while an experiment is in progress.

For purposes of discussion, we can decompose the entire system data flow into four smaller flows. The first involves the tracker, the second the SGI, the third the display and the fourth the physical objects.

The first sub-flow is the loop that involves the tracker (top left quadrant). The Optotrak System Unit strobes the infrared emitting diodes (IREDs) though the strober collection box(es). Each box controls up to six IREDs, and several boxes may be chained together if more IREDs need to be tracked. The IREDs send (non-physical) infrared beams which are intercepted by the tracker (a Northern Digital Optotrak). The tracker uses three independent cameras to estimate the location (but not orientation) of each IRED. This location information is passed back to the Optotrak System Unit.

The second major sub-flow involves the SGI (middle of diagram). The System Unit passes the location information to an SGI Indigo2 Extreme workstation via an Optotrak Data Acquisition Unit (ODAU) and a SCSI. The SGI also looks for data from the keyboard and mouse. The SGI is running the VHL software, which interprets this data, and uses it to update the experimenter's monitor. The subject's monitor is daisy-chained to the experimenter's monitor, and shows the same image except that the experimenter's monitor is not interlaced. The SGI controls the timing of the CrystalEyes emitter and can also emit pulses on an RS232 cable.

The third sub-flow involves the display seen by the subject (lower left of diagram). The emitter triggers the Crys-

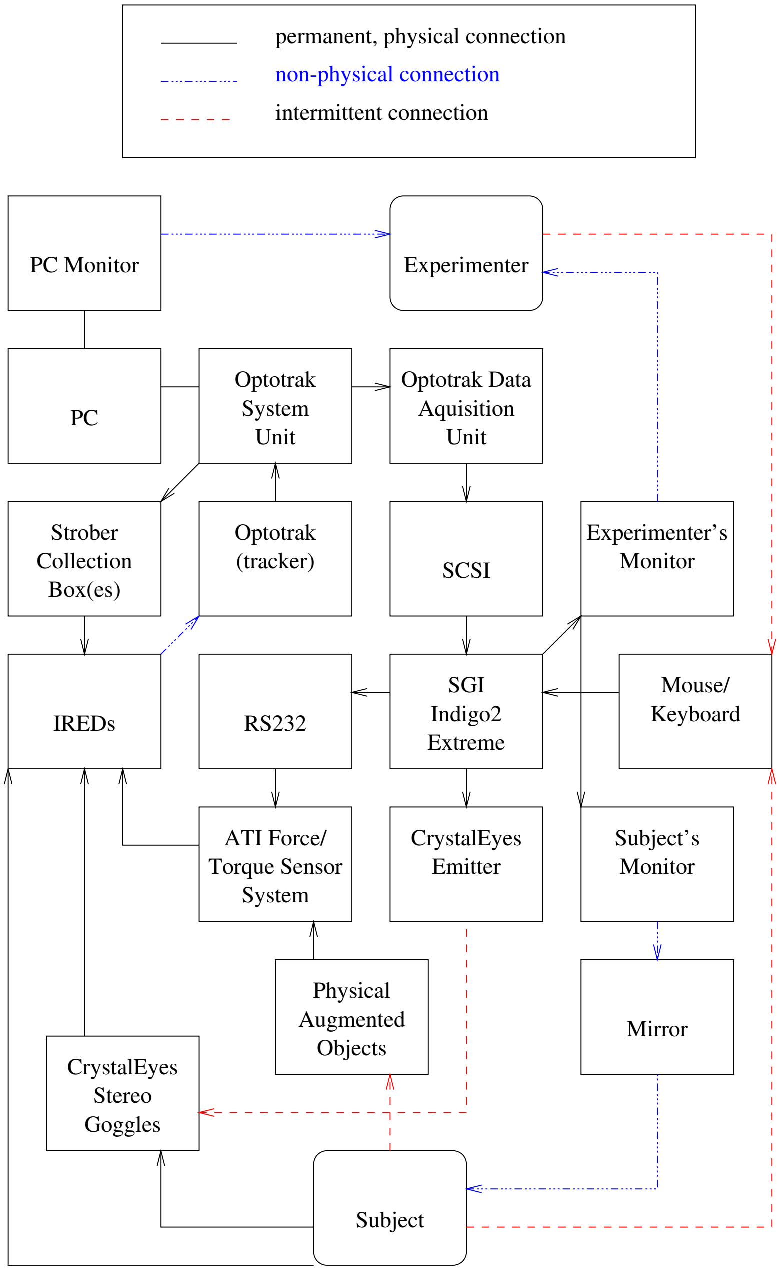

Figure 15: *Data flows through equipment which not just physically connected (solid lines) but though indirect or intermittent connections (dashed lines). Indirect flows include infrared beams, while intermittent connections include human interaction with physical items in the environment. The arrows indicate the direction of the data flow during system operation, which may be different than during initialization. (The PC is only used during initialization).*

talEyes goggles to alternate left and right views. The goggles are physically attached to the subject and to a set of IREDs. The IREDs on the goggles are used to track its location, and (using some calibration techniques) to estimate the positions of the subject's left and right eyes.

The last sub-flow deals with the subject's tactile interaction with physical objects (bottom of diagram). Through intermittent physical contact, the subject can manipulate physical and augmented objects in the environment as well as use more traditional input techniques such as the mouse and keyboard. The keyboard is not currently used in any experiments, but has been used in the system for demonstration purposes.

5.2 Synchronization

There are three major components of the system which need to be synchronized: the tracker, the display, and the software which lies between them (Figure 16). During the software *RequestData* routine, an asynchronous signal is sent to the tracker to begin collecting data. The software thread continues with other operations such as clearing the buffers while the tracker collects data. Once the tracker has collected the data, an internal flag is set. The software routine *GetData* blocks on the tracker's *SendData* routine. The data is returned as soon as it becomes available, or immediately if the tracker's *CollectData* thread has finished.

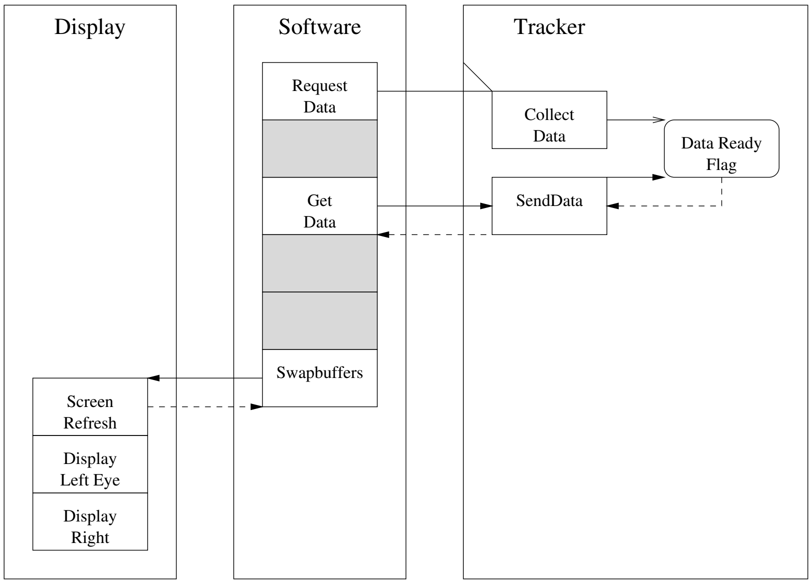

Figure 16: *The software synchronizes with the tracker to collect and receive data, and with the monitor to display the stereo scene.*

Single buffering is suitable for still scenes, but not for animation where smooth movement is required, such as augmented reality. Consequently, we are using double buffering. When the software is finished rendering both left and right views (which may be done in either order), it swaps buffers. When buffers are swapped, they block, waiting for the next screen refresh of the display device (a monitor). The StereoGraphics emitter is also synchronized with the display by the SGI hardware. The left shutter of the stereo goggles is opaque when the image for the right scene is displayed and vice versa.

5.3 Timing

The SGI is programmed to update the display at 60 Hz in stereo mode. The glasses are shuttered at 120 Hz, giving each eye 60 images per second for a total of 120 images each second. Tracking is performed at 60 Hz by the Optotrak. Force sensing is conducted at 200 Hz or higher, but at the present time it is not used to control the system, but only for data acquisition.

The Optotrak provides a number of parameters to control the tracking system which are preset by the experimenter [Northern Digital Inc., 1992]. Figure 17 shows how these inter-relate. The *NumberOfMarkers* is the number of markers tracked in the experiment. These are be used to track the head, augmented objects, or parts of the body such as fingers or wrists. The tracker tracks each marker at a particular *Frame Frequency*. Markers are activated at intervals of *MarkerFrequency*. Each marker is activated for *DutyCycle* percentage of the marker period and emits infrared light during that time period. The markers are strobed in order (whether attached to one or more collection boxes), so the location of each marker is uniquely identified. This means that when a buffer of marker locations is received, the location for marker one was collected furthest in the past and hence has the most lag. Frames of data are internally numbered. When *RequestData* and *GetData* calls occur, the next available frame is provided.

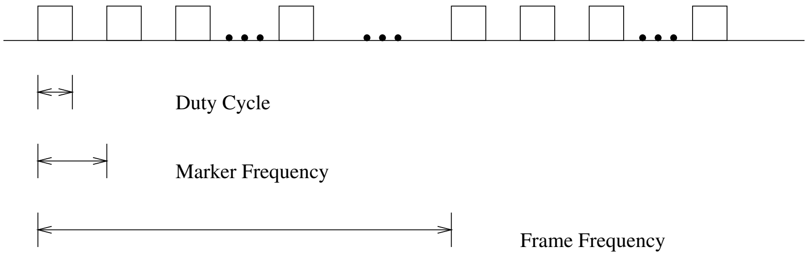

Figure 17: *Markers are activated at intervals of Marker Frequency for Duty Cycle periods of time. Activation of the first marker occurs at the Frame Frequency.*

While the *FrameFrequency* cat be set to track at very high rates (up to 2400 Hz), the SCSI cannot provide data any faster than 60 Hz, thereby limiting the rate at which the SGI can receive data to 60 Hz. In order to not miss any frames of data, the data must be received by the software at that rate. Consequently the complete cycle, from *RequestData* to *Swapbuffers* must be completed in 1/60th of a second. The scene is also displayed at the same rate. Since the left and right eyes must be displayed separately, each eye must be displayed in half that amount of time, or in 1/120th of a second. The left eye is drawn in the top half of the buffer, and the right in the lower half. The subject's display is interlaced, so the first pass of the electron beam draws takes the left eye view and draws it on every second line from top to bottom of the subject's screen. The beam then returns to the top of the monitor screen, but continues down the same buffer, drawing the right eye view. This means that the view experienced by the subject's right eye is older and has more lag than the left. This is an unfortunate hardware limitation, since the right eye is the dominant eye in most people.

The worst lag occurs at the bottom of the screen, for the right eye using data obtained with marker one. The most accurate display is at the top of the left eye view using data from the last marker.

Although the timing difference between markers is extremely small, we put the early markers on the slowest moving objects in order to minimize lag. Since the head generally moves less quickly than manually manipulated objects, we place markers one through three on the head.

For markers attached to a rigid body, the distance between markers should be constant. We can measure this distance in each frame to validate the data. Again, while the distances would be small, we choose consecutively numbered markers for each rigid body. Independent of any small technical advantage, the interface is easier to use when markers are consistently assigned to objects.

5.3.1 Timing Induced Error Analysis

Peak velocity for pointing to a virtual object, without transporting any additional objects is 721 mm/s [Graham, 1995]. Fast, tangential wrist motion can occur at rates up to three meters per second [Atkeson and Hollerbach, 1985].

When objects are moved at low velocities, static error dominates. At high velocities, temporal error dominates. The system tracks objects at 60 Hz, and updates the visual display at 60 Hz stereo (120 images per second, 60 to each eye). Mean static accuracy is $\pm$ 0.47 mm (width), $\pm$ 1.73 mm (up/down) and $\pm$ 1.33 mm (depth). Absolute value, worst case errors are: 1 mm, 4 mm, and 3 mm respectively [Summers et al., 1999]. These values are better than other systems running at similar rates.

Temporal accuracy is approximately 1.5 frame lag time. Combining spatial and temporal errors, we have at worst 22 mm of error. At peak pointing velocity, the worst case temporal error is 18 mm.

We can examine the temporal accuracy in more detail by analyzing the timing differences induced by strobing. The Optotrak collects marker location measurements sequentially, but when the VHL uses this data, it assumes that the data was collected simultaneously. This *simultaneity assumption* is made by most current tracking systems [Welch, 1996]. *MarkerFrequency* can range from 100 to 4000 Hz, with the default value typically set at 2500 Hz [Northern Digital Inc., 1992].

At peak velocity and the default marker frequency, a moving object would be recorded at a position 0.29 mm different by two consecutively strobed markers. For maximum and minimum marker frequencies, this ranges from 0.18 mm to 7.2 mm. The error between three consecutive markers is thus 0.58 mm on average. If an experiment used 9 markers, the error between the first and last would be 2.60 mm.

Since we use distance errors between markers on rigid bodies to identify erroneous data, we can set tighter bounds by minimizing the timing induced errors.

5.3.2 Discussion

Peak velocities reported for head movements in virtual environments are lower than peak velocities for hand movement. However, the head movement times are for head-mounted displays which inhibit movement due to their weight and inertia. Observation suggests that the velocity is still lower when non-intrusive light-weight stereo goggles are worn while peering into a small desktop virtual model display. It would be interesting to verify this empirically.

This ordering of velocities seems to hold for non-peak velocities too. Some studies indicate that people naturally hold their heads still while performing precise alignment tasks. Since the total accuracy increases at lower velocities, this would support the placement of the most temporally accurate markers on the objects with highest velocity, namely the manipulated objects and not the head.

6 Consistency

Consistency refers to a technique whereby comprehension is aided by the use of naming conventions [Griswold, 1998]. In software development this is a commonly used technique to aid the comprehension of programmers by using stylized names for data, functions and modules. The conventions chosen are often arbitrary, and consistency is enforced through compliance rather than programming tools.

Consistency in the VHL is used not only to facilitate code development, but to aid users of the system. It aids in communication among users and between users and developer since it provides a common vocabulary or local jargon. Consistency reduces the time for a user to execute programs since they can easily identify the programs they are responsible for executing. It reduces set-up times as it is easier to remember something short than to type in some criteria each time.

Some of the conventions used are listed below. The meaning and intended usage of the items identified by the conventions listed here will be made clear in later sections. The conventions are provided early in this document to aid comprehension for the reader in later sections of this document.

Conventions that help developers only:

- Class names are prefixed by the letter ‘o’ to avoid name conflicts with other systems and libraries.
- Enumerated types are always upper case.
- Class member data is prefixed by ‘m’ as in m<Data>, static data by ‘s’ as in s<Data>. Data is modified by the routines set<Data> and queried by the routines qry<Data>.

Conventions that help users and communication:

- The files most commonly changed by the users are the attribute files. Prefixing them with “AA” places them at the top of the directory listings.
- All scripts that the users execute are prefixed by “run”.
- All files produced by the point of view calibration are prefixed with “ex” since they are in a tab delimited format suitable for analysis in Excel (tm). Users can ensure that they are using the correct calibration results by checking the dates on these files.
- Files produced by the object calibration have names composed of the prefix “calib” followed by the type of the object. Users can ensure that they are using the correct calibration results by checking the dates on these files.

- Markers 1 – 3 are always used to track the goggles.
- For a given class o<Class> we use the following naming conventions:

  declaration o<Class>.h
  definition o<Class>.c
  default attribute file AA.<class>
  target keyword in trial specification <Class>
  configuration file keyword -<class>

7 Graphical Object Inheritance

Graphics performance and scene complexity are two requirements that must be balanced for every augmented reality system. A scene is complex if it takes a long time to render it. This can happen if the scene requires a large number of polygons to represent it or requires a time consuming algorithm (like ray-tracing). By contrast, an impoverished scene has only a few simple objects. Graphics performance refers to the time required to render the scene, relative to the data the scene represents. Some systems, like Azuma, build the desired complexity first, then extract performance [Azuma and Bishop, 1994]. Others, such as flight simulators focus on performance first. The VHL uses an impoverished environment and focuses on graphics performance since we are testing human performance.

To ensure graphics performance, we have low complexity scenes with only a few simple objects. Only objects that were requested by the experimenters have been implemented. Migration to faster hardware will enable us to expand the repertoire of objects.

Augmented objects are composites of physical and virtual objects. This remainder of this section deals with the description of the virtual objects.

7.1 Base Class

There are a number of graphical objects in our system which derive from the top level abstract base class *oGrObject*. Each graphical object instance is uniquely identified by the combination of class type and name.

The tracking system provides three degrees of freedom (XYZ location) for each marker. By using three or more rigidly attached markers, we can also determine orientation (yaw pitch and roll), and hence determine all six degrees of freedom (DOF). Unlike most other systems, we do not always want the movement of the graphical object to mimic the 6DOF movement of the associated physical object. For example, we might wish to constrain the graphical portion of an augmented object to the floor of the workspace, independent of the height of the associated physical object.

In order to decouple the movement of the physical and virtual components, we provide each object with both a movement source and a movement type. Movement sources include real-time data from the tracking system, simple pre-defined mathematical models and the default – no movement. Movement types include various degrees of freedom motion: (2DOF,...,6DOF), STATIC (or no movement), OFF (invisible or not drawn), and TRIAL (moves under experimental trial control).

Specifying the movement type allows us to reduce the degrees of freedom of the graphical object by projecting the mapping onto a lower dimensional space. This mapping is not unique however. When we project from six to five dimensions, we can choose any five of the six degrees of freedom. This is similar for the other lower dimensions. We only provide one instantiation for each lower dimensional mapping. We arbitrarily chose the degree of freedom to constrain on a per class basis. For example, when we constrain a cube to 2DOF, then the virtual cube is always displayed on the desktop, oriented according to the workspace, no matter how its physical counter-part is moved. Conceptually, it is like a shadow. We could also have implemented the 2DOF case to be orientation around two axes while the center of the cube remained in the same position.

This unusual decoupling of the physical and virtual components of augmented objects has two benefits. First, it allows experimenters to determine what impact the decoupling has on human performance. Second, by allowing constrained motion, we can reduce the number of positional markers required per object. For instance, at least three location markers are required to define 6DOF motion, but only one location marker is required to define one, two or three DOF motion (if we pick location not orientation as the degrees of freedom). While this is not important for our system as the current experiments use an impoverished environment with few objects, this could increase the number of tracked objects supported in other (or future) augmented reality systems.

7.2 Abstract Subclasses

Augmented objects are a hybrid of physical objects and computer-generated virtual objects. Graphical object hierarchies for augmented reality environments must consider both of these facets. Unlike typical graphical object hierarchies, we first create subclasses according to a physical property of the augmented objects – their level of deformation. Objects are considered either "rigid" or "flexible" as seen in Figure 18. For example, the augmentation of a block of wood is classified as rigid whereas the augmentation for a finger is classified as flexible. Manipulation techniques bifurcate for these two types of objects, so creating subclasses based on the deformation property allows us to encapsulate functionality.

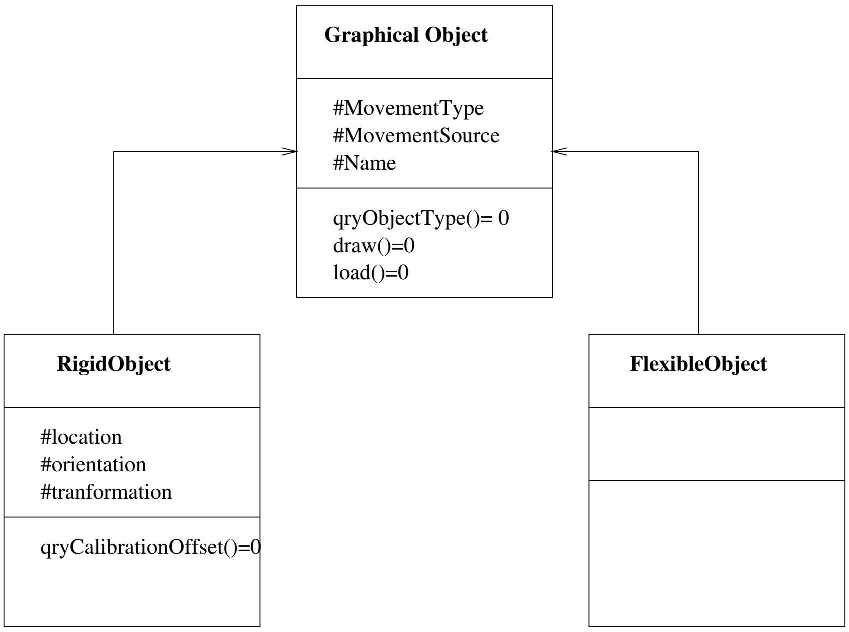

Figure 18: *The first division in the hierarchy of graphical objects is based on the physical properties of the object that they augment.*

Rigid objects are non-deformable, meaning that any markers attached to them form a "rigid body". Three or more location markers are used to form a marker coordinate system. The object has its own coordinate system. These are shown in Figure 19.

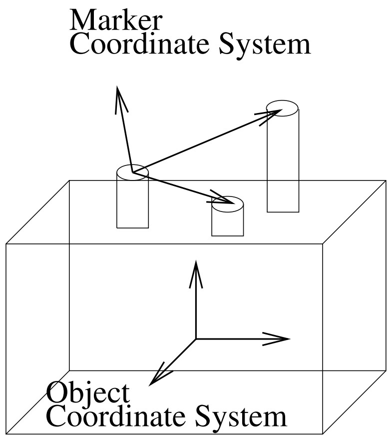

Figure 19: *The rigid object has an internal, unchanging coordinate system as does the "rigid body" formed by the markers.*

Not only are the markers and the physical object rigid bodies, but a rigid object and its attached markers also form a rigid body. Consequently we can define an exact transformation between marker and object coordinate systems. Specific details for creating this transformation are discussed in [Summers et al., 1999]. Consequently, the abstract class *oRigidObject* contains as member data a 3D location, a 3D orientation, and the transformations used to position the virtual object relative to the markers attached to the physical object.

Flexible objects are characterized by markers that are not fixed relative to one another. This means that the calibration procedure used to map rigid objects and their associated graphical components cannot be used. The mapping process for flexible objects is thus deferred to each concrete subclass.

7.3 Rigid Objects

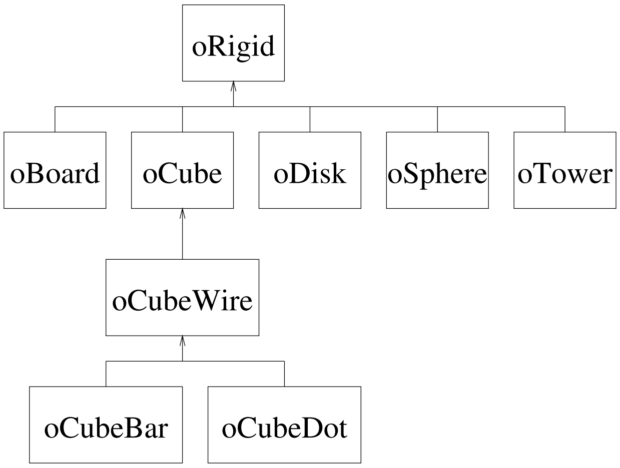

Figure 20: *Concrete subclasses deriving from abstract class oRigidObject.*

The inheritance hierarchy for concrete subclasses deriving from abstract class *oRigidObject* is shown in Figure 20. In our system, we define *simple objects* as those composed of a few graphics primitives and *complex objects* as those composed of many graphics primitives or other simple or complex objects. Simple objects such as cubes, disks and spheres derive directly from *oRigidObject*.

Complex objects are handled in different ways, with the goal of creating an easy interface for the experimenters. (Recall that experimenters set up the experiment, while subjects use the environment.) Experimenters interact with the objects in two ways: through attribute specification and through trial specification. Attributes specified include size, shape and colour. The trial specification identifies the target in each trial by listing the class type.

A checkerboard is considered to be a complex object since it is composed of a large number of tiles. We handle this type of complexity by creating a separate class which has the same behaviour as the simple objects. Like other simple objects, the checkerboard class has its own attribute file, and can form the virtual component of any rigid physical object one wishes to augment. While the experimenters could place a collection of cubes to create a board, encapsulating this behaviour in a class reduces the complexity for the experimenters in two ways. First, it reduces the number of attributes that must be set since every “check” is the same size, with alternating colours, in the same plane etc. as shown in Example 2. Second, the movement source and movement type are set once for the entire object, not individually for each cube or tile.

Another type of complexity is found when an object is composed of non-primitive objects. Internally, the complex object contains the non-primitive objects as member data. When the complex object is drawn, it simply calls the draw routines for each member data, and does not deal with the graphics primitives directly. The attribute file for the complex object lists the attribute files for each member data. This specification is more indirect than for simple objects that have one attribute specification file per object. However, we feel the benefits outweigh the disadvantages of having one main file plus one file per member data.

The first benefit is that we reduce cognitive overhead for the experimenters since they can use their existing knowledge of simple object attribute specification. The second benefit is that attribute specification is potentially faster since

```
// Board properties.

// Size of board in mm
Width = 220.0
Depth = 220.0

// NxN board
NSquares = 8

// Colours to alternate on checkerboard.
LightColour = 0x00ffffff
DarkColour = 0x00000000

// Centre of board
X = 0.0
Y = 0.0
Z = 0.0

Property = LMC_DIFFUSE
```

Example 2: *Flat ASCII data files are used to assign default graphical object attributes such as size, shape and colour. Comments indicate the meaning of the attribute and any necessary information such as units. Here the size of the a checkerboard is set.*

pre-existing attribute specifications for simple objects can be used. This would scale to a larger system with a database of defaults for simple objects. Third, it reduces development time since the simple objects have already been fully programmed and tested.

A tower falls into this category of complexity. A tower is composed of three stacked cubes. The attribute file for the tower lists only the attribute files for each cube as shown in Example 3. Experimenters are already familiar with the attribute specification for cubes. If we followed the format of the simple objects and provided a single file to specify the attributes for an object the attribute names would be inconsistent and hence confusing. For example, the attribute tag "RotateZ" rotates cubes, disks and sphere about the Z-axis, but would have to be called TopCubeRotateZ, MiddleCubeRotateZ etc. for the tower.

```
// The three cubes that compose the tower.
// Each file lists the attributes of that cube

TopCube    = AA.tower.top
MiddleCube = AA.tower.middle
BottomCube = AA.tower.bottom
```

Example 3: *The tower attribute specification points to the attributes files for each of its data members which also derive from the graphical object hierarchy.*

Not all of the cubes' functionality is used when they become member data of another object. They are not independently calibrated, and their movement is dictated by the tower. For example, the tower's movement type and movement source are used to control the location of each cube.

A third set of composite objects are the cube subclasses. Deriving from a solid cube is a wire-frame cube, and from that two wire-frame cubes with additional objects inside of them. CubeDot contains a small sphere inside of a wire frame cube, while CubeBar contains a bar running from diagonal corners. These variants could have easily been programmed as an attribute instead of subclasses. They were programmed as separate sub-classes to expand the experiment possibilities. The trial specification file identifies the target by class name. Creating subclasses allowed us to synchronize the class names used in the trial file, the actual class names, and the individualized attribute file names. In Example 4, specifying "CubeDot" in the first trial places a target of class *CubeDot* with attributes specified in the file "AA.CubeDot", while the second trial places a target of class *CubeWire* with attributes from "AA.CubeWire". Since each target type has its own attribute file, these objects can be very dissimilar if desired. Another solution would be to expand the format of the trial specification file to include these attributes.

7.4 Flexible Objects

The one flexible object we have in our system is a pointer. This pointer is attached to a hand. One marker is located at the tip of the index finger, another on the knuckle, and a third in a non-collinear location on the back of the hand. (See Figure 21.)

```
// Each line contains:
//      file name into which data will be logged
//      target object type
//      x y z coord of target
//      3 parameter, named A,B,C which are used differently
//      for different experiments.

//fn    type        x      y      z      A      B      C
//--    ----        ----   ----   ----   ----   ----   ---
nts010  CubeDot     -100   0      15     44     20     0
nts020  CubeWire    -50    0      15     22     6      -22.5
nts030  CubeBar     100    0      15     22     9      -45
...
```

Example 4: *The trial specification determines when and where virtual targets appear. Here three cube variants are targets in different trials. Parameters are interpreted according to the specific experiment being executed.*

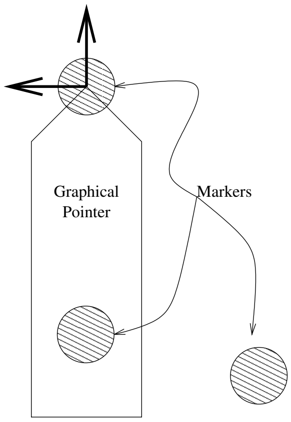

Figure 21: *Markers are attached to the finger tip, knuckle and back of the hand. The position and orientation of the graphical pointer is drawn according to the location of the markers.*

There are many possibilities for creating a mapping between a set of markers and a graphical object. Choices involve selecting the orientation, position, size and shape of the object. For the pointer in our system we made the following choices: the orientations of the marker and pointer coordinate systems were aligned; the tip of the pointer was associated with the marker on the tip of the finger; the size of the pointer remained constant; and the shape of the pointer remained fixed. Alternative choices would have been to resize the object based on the distances between markers, or to associate the position with a different marker or an averaged function of multiple markers.

Our choice for the pointer was primarily based on a requirement to incorporate a previous 2-dimensional experiment in our testbed. Different choices than those we chose for the pointer may be more appropriate for other objects. One criterion to consider when making a choice of representation is the focus of the user. The virtual object should be constant sized when we do not want the object to distract the user from a task. In our case, we would not want a changing pointer representation to distract the user when the task was to point to another object. Conversely, the object should be resized if the task involves feedback about the object, such as size or distance changes. An (unimplemented) example is stretching a virtual rubber band, the ends of which are attached to different fingers. Here the focus is on the manipulated virtual object itself, so it is crucial that the size and shape of the virtual object be non-constant.

8 Experiment Flow Control

An experiment consists of a sequential series of trials. Each trial in turn consists of a predetermined sequence of phases. Trials may be repeated within an experiment just as phases may be repeated within a trial. The selection and execution order of the phases is determined by the experiment type. The number and order of the trials is defined by the experiment. A schematic is shown in Figure 22.

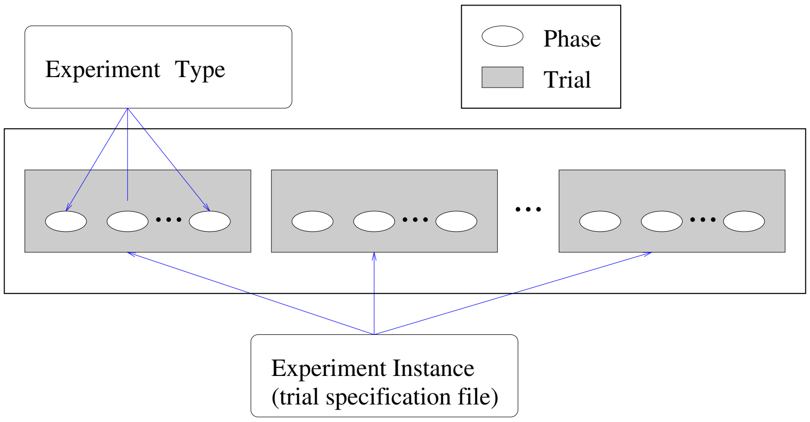

Figure 22: *An experiment is composed of trials which are subdivided into phases. The experiment type determines the phase ordering while the number of trials is determined by the specific experiment instance.*

8.1 Example – Force Experiment

The *Force Study* examined the effect that visual changes to size had on lifting and gripping forces when objects were manipulated by hand. One experiment changed the size of the objects physically, while the other (discussed here) changed the object graphically. Subjects were asked to lift a tower to a height of approximately 2 cm, maintain the lift for 8 seconds, then place the tower on a 2D replacement target on the table. After the tower was replaced, all visual images were eliminated, a random amount of time elapsed and then a newly sized tower appeared and the process was repeated.

The tower is composed of a physical grip apparatus which measures lifting and gripping forces exerted by the subject. Overlaying the physical apparatus is a virtual tower. The virtual tower is composed of three cubes. The size of the top cube matches the grip apparatus and remains a constant size throughout the experiment. The width and depth of the middle and bottom cubes are changed on a per trial basis. The 2D replacement target corresponds in size to the bottom cube during that trial while its orientation anticipates the next trial.

Table 2 shows the order of the phases, and the machine and subject events in each phase. This specification formed the basis of discussion between the experimenter and developer. A clear understanding of "what would happen when" greatly reduced programming time for the developer and testing for the experimenter.

A more detailed description of the phases for the Force Experiment follows.

Set Trial Attributes The width and depth of the middle cubes and the rotation of the replacement target are set for the virtual tower.

Tower On The virtual tower is made visible with its new attributes.

Log0 The virtual tower is visible to the subject who lifts the grip apparatus to a height of 5 cm. The motion data of the grip apparatus is recorded. When the lift has been held for an adequate time to stabilize, the experimenter presses the middle mouse key and data logging stops.

Replacement The replacement target appears and the subject places the tower on the target. The size of the target matches the size of the bottom of the tower, and the orientation of the tower matches the orientation of the next lift. The experimenter presses the left mouse button once the tower has been replaced. While this phase could have been implemented to continue logging the data into the same file as the previous phase, the experimenter was not interested in this data, and requested that it not be logged to simplify subsequent data analysis.

Tower Off The virtual tower is turned off so that it is not visible.

Write The logged data is written to the file specified by the experimenter in the trial specification file.

Idle To ensure that the subject does not anticipate the reappearance of the tower, the experimenter waits a random amount of time. The experimenter signals the end of this wait by pressing the right mouse key.

	Phase Sequence						
	Set Trial Attributes	Tower On	Log0	Replacement	Tower Off	Write	Idle
Display	tower	tower	tower	replacement target, tower	-	-	-
Log	-	-	all	-	-	-	-
Successful Termination	finished set	finished set	M mouse	L mouse	finished set	finished write	R mouse
Unsuccessful Termination	-	-	Esc Key	Esc Key	- failure	i/o	Esc Key
Sees	tower but not grip apparatus	tower	tower	replacement target, tower	-	-	-
Feels	-	-	grip apparatus	tower	-	-	-
Does	-	-	lifts tower 5 cm	replaces tower on mark	-	-	-

Table 2: *The phases composing the Force Experiment are listed horizontally in the order they are executed. Vertically, for each phase, the machine events are listed followed by the subject events. In this experiment, the subject lifted a grip apparatus augmented with a graphical tower to a height of 5 cm, held it for a short period of time, then replaced it on a 2D target mark. The replacement target was sized according to the size of the tower base which changed from trial to trial.*

8.2 Phases

Phases are specified by a combination of machine generated events and subject generated events. Machine events prescribe: which graphical targets are displayed, whether data is logged, what data is accumulated, the condition for successful completion of the phase, and the condition for unsuccessful completion. Currently all the phase transitions are controlled by mouse or keyboard events. Another method would be to create transitions based on tracking data such as velocity, position, orientation, etc. Subject events prescribe what the subject sees, feels and does. Unlike traditional graphics or even virtual reality, the subject may see more than just the objects displayed by the computer. In augmented reality, the subject also sees physical objects in the scene, such as his/her own hand or other props. By comparing both what is displayed and what the subject sees, one can decide whether the experiment is run in virtual reality mode or augmented reality mode.

While this decomposition seems obvious now, it actually involved considerable iteration with the experimenters. Once this event structure was identified, experiment development time was greatly reduced, primarily by reducing the number and duration of communication/programming/evaluation cycles. Less time was needed to communicate ideas since both developer and users understood the information needed by the other group. Less time was spent (re)programming as the experimenters could more easily visualize the sequence of events and determine if the sequence met their requirements without seeing a programmed demo. Since less prototypes were required, less time was spent by the experimenters to evaluate the software.

Phases are independently defined, so they may be combined in any order for an experiment. A given experiment type simply iterates through a pre-determined sub-set of the phases for each trial. The phases currently used by experimenters are summarized in Table 3.

Phase reuse varies according to the commonality of the experiment types. Some phases such as *Write* are used in every experiment, whereas others such as *Idle* have only been used in one experiment. There are five phases that set attributes for different experiment types. Again, some of these have higher reuse than others. *SetTrialAtttrTransport* has been used in a series of four experiments, all dealing with the transportation of a physical object to a virtual target [Wang, 1999].

Software reuse is a goal of object oriented software development as it can reduce development time, facilitate maintainability (since modifications are more localized), and can increase reliability through the use of previously tested components. In our case, the creation of a set of phases which can be recombined in any order reduces the time and effort required to develop future experiments. Most new experiments reuse at least some of the phases already developed for other experiments. In some situations, new experiments have been created with no programming changes.

Phase Name	Displayed Target(s)	Data Logged	Successful Termination	Unsuccessful Termination
SetTrialAttr<Exp>	-	-	finished set	-
Start	start	-	L mouse	Esc Key
Log	end	all	M mouse	Esc Key
Log0	-	all	M mouse	Esc Key
Write	-	-	finish write	i/o error
Replacement	replacement	-	L mouse	Esc Key
Idle	-	-	R mouse	Esc Key
Tower On	-	-	object visible	-
Tower Off	-	-	object invisible	-

Table 3: *Each phase is defined by the graphical objects displayed, the data logged, and the criteria for successful and unsuccessful termination. Phases may be divided into those that modify objects, those that affect the timing of trial controlled targets, and those that affect the timing of system controlled objects.*

8.3 Graphical Object Specification

One goal of any graphics program, not just experimental systems, is to ensure that objects are displayed as expected by the user. The graphical objects displayed in the VHL are a function of what objects are selected, where they are located, when they appear and disappear, and how they look.

what What object or objects are selected to be included in the display.

where The initial position (location and orientation) in which an object is displayed is specified by user modifiable defaults provided to the system during initialization. Subsequent positions are either a continuation of the default, or based dynamically on the movement type and movement source.

when Not all objects are continuously visible in the system. Some objects may appear only once during system use, while others may disappear and reappear many times.

how The appearance of an object can be modified by changing some or all of its attributes. While an object cannot morph into a completely different class, attributes such as size, shape, colour, and level of detail can be modified.

8.4 Graphical Object Control

We separate the graphical objects in our system into two groups – experiment objects and system objects. These groups have separate initialization interfaces and default behaviours although objects in both groups can interact with each other. Experiment objects are the targets specified by the trial specification file and system objects are the objects specified by the system configuration. When the system is run in non-experimental mode, only the system objects are available. For example, the tower is a system object and can be used whether there is an experiment or not. The replacement target is an experiment object, and would not exist without an experiment.

Since we separate these groups, we can operate our system with only one group, or with both. This allows us to create different applications. If we use only system objects and not experiment objects, we have an environment suited to demos, testing new objects etc. Using only experiment objects allows one to concentrate on the trial transitions.

Object control refers to all aspects of an object – what, where, when and how. Unlike most augmented reality systems, we layer our object control by level of detail, rather than starting with each separate aspect. Control levels are cumulative. Each higher level relies on previous levels. The system is functional at each level.

The lowest layer consists of the object's defaults set during construction. If just level 1 controls are used, only a small finite set of objects are available, and all the object attributes are set at compile time. While the experimenter can move the augmented objects around, there is little other interaction.

The second level of object control is provided by object attribute initialization files. The same attribute initialization files may be used for both experiment objects and system objects. By adding level 2 controls, the user can initialize the objects in the scene by changing the colours, sizes etc.

A third layer, also pre-set at initialization, consists of grouping structures to overlay those attributes most commonly changed in our system. There is one grouping structure for experiment objects, and one for system objects. Level 3 trial specification provides the facility for placing different objects in the scene at different times.

The fourth level of control depends on the interaction of of previous levels of control with the experiment type. Level 4, experiment type control, allows customization of the level 3 controls.

8.4.1 Level 1 – Constructor Initialization

Objects are set with default values for all attributes during construction. These defaults can only be changed by re-coding.

8.4.2 Level 2 – Attribute Initialization Files

Methods for changing individual attributes of graphical objects are available as clas methods, but these are not visible to the (non-programmer) users. Consequently, we provided a load mechanism for every graphical object through which the user can override the default values. A conceptually simple mechanism was to list one attribute per line as previously seen in Figure 2.

An attribute file can be used by multiple objects of the same class. Using common attribute files provides an easy method of ensuring consistency between objects. For example, only one line need be edited to make the top face of all cubes that share an attribute file blue instead of red. It does not even matter if some of the cubes sharing this file are system objects while others are experiment objects. If preferred each object can have its own attribute file.

8.4.3 Level 3 – Grouping Structures

It is not useful to have completely identical objects co-located in space as would happen with a shared attribute file. For our application, we identified the most common attributes that differ between objects. We included the subset that would fit our file format specification in a higher level graphical object specification. Other applications might select different attributes for this layer of attribute assignment.

Example 5 shows a grouping structure for experiment objects. Unlike the previous control level, this structure provides some timing control. Each line of the file refers to a separate trial. Only the target object listed on that line can be visible during that trial. It will not be visible for the entire trial, but that level of control will be discussed in the next section.

There are two methods of grouping system objects. The first method uses command line switches and is in the process of being phased out in favour of a file format. Currently, these methods are used concurrently. The command line format allows one to specify the movement type for a finite set of objects. The movement source is inferred from the movement type. The second method is a file that allows virtually unlimited numbers of system objects. Right now, only static objects are possible. Completing the transition to a file supporting all movement types will allow more complex scenes and object interaction. In Example 6, four cubes, a checkerboard and two disks are placed at various locations in the environment. The blue cubes share an attribute file and hence are identical except for their location.

8.4.4 Level 4 – Experiment Type

The experiment type is currently specified on the command line along with the system objects. The experiment type, in conjunction with the experiment instance and the system configuration provides another level of control. The experiment type controls the timing and appearance of graphical objects though the mechanism of phases. (Figure 23).

Phases can be divided into three major classifications: those that modify objects, those that determine when an object is included in the environment (timing), and those specific to the tracking system. The timing phases can be further subdivided into those that control when target objects are displayed and those that control when system objects are displayed. Experimenters do not know about the tracking specific phases which can be considered "private" routines.

The data format for each trial is consistent across experiments, but the interpretation of that data is not. The *SetTrialAttr<Exp>* set of phases provide a way of interpreting that data and using it to modify the attributes of graphical objects. The objects modified can be either static trial controlled targets or system controlled augmented objects or both. A particular experiment type will call the phase from this set that interprets the data appropriately. The interpretation for several different experiment types is shown in the comment section of the experiment specification (Example 5).

```
// Each line contains:
//      trial file name
//      target object type
//      x y z coord of target
//      3 parameter, named A,B,C which are used differently
//      for different experiments.
//      <exp. name> -- 3-D pointing
//              A:  target disk diameter
//              B:  time delay
//              C:  unused.
//      <exp. name> -- transportation and rotation
//              A:  target cube depth
//              A:  target cube height
//              A:  target cube width
//              B:  unused.
//              C:  rotation of target about z axis in degrees
//      <exp. name> -- orientation cues
//              A:  unused.
//              B:  unused.
//              C:  rotation of target about z axis in degrees
//      <exp. name> -- perceived dimensions
//              A:  target cube width
//              B:  target cube depth
//              C:  target cube height
//      <exp. name> -- force
//              A:  width and depth of middle cube on tower
//              B:  width and depth of bottom cube on tower
//              B:  width and depth of replacement target
//              C:  rotation of replacement target about
//                      z axis in degrees
//              ignore type and locations, since no end target.

//fn    type    x       y       z       A       B       C
//--    ----    ----    ----    ----    ----    ----    ---
nts010  Cube    -100    0       15      44      20      0
nts020  Cube    -50     0       15      22      6       -22.5
nts030  Disk    100     0       15      22      9       -45
...
```

Example 5: *The trial specification determines when and where virtual objects appear. Parameters are interpreted according to the specific experiment being executed. The comments show the interpretations for several experiments.*

```
//object    object   defaults        X       Y       Z       mvmt     mvmt
//type      name     file                                    type     src
//------    ------   -------         ---     ---     ---     ----     ----
Cube       myCube1    AA.blue_cube    50      -10     23      static   static
Cube       myCube2    AA.blue_cube    20      -20     13      static   static
Cube       myCube3    AA.red_cube     50      -30     13      static   static
Cube       myCube4    AA.red_cube     50      -40     13      static   static
Board      myBoard    AA.board        0       0       0       static   static
Disk       myDisk     AA.disk         -10     -10     0.1     static   static
Disk       myDisk2    AA.disk         15      15      0.1     static   static
```

Example 6: *Attribute values of system objects can be set using shared or individualized files. A few commonly changed attributes such as position are individually set.*

System timing phases simply toggle the display of the system objects on or off. The state transitions for target timing phases are more complex. All targets are turned off except the one(s) appropriate to that phase. Control then passes to a wait state that continues to update the animation until the phase transition event occurs. The phase transition event depends on the particular phase.

There are two tracker specific phases that interact with the tracker data formats (though not the tracker itself). These are considered private methods that may be called by other phases, but are not included in an experimenter's phase specification, and hence are not included in Table 3. One routine alternately calls a routine to redisplay the scene and logs the data, while the other simply writes the log to a file. These routines not only do not change the attributes or movement type of any graphical routines, they have no knowledge of them. They are also unaware of which trial the experiment is currently executing.

9 Conclusions

By isolating the the timing mechanism, a modular, high-performance augmented reality environment testbed for experiments can be created. Co-location of the hand and the objects to be manipulated is easily accomplished by using a mirror as a beam-splitter and providing an off-axis head-coupled stereo display. A number of advantages accrue when one considers physical as well as the graphical components of augmented objects. For example, issues of calibration and model transformations can be isolated for rigid objects; and flexible objects can be implemented in a task dependent manner.

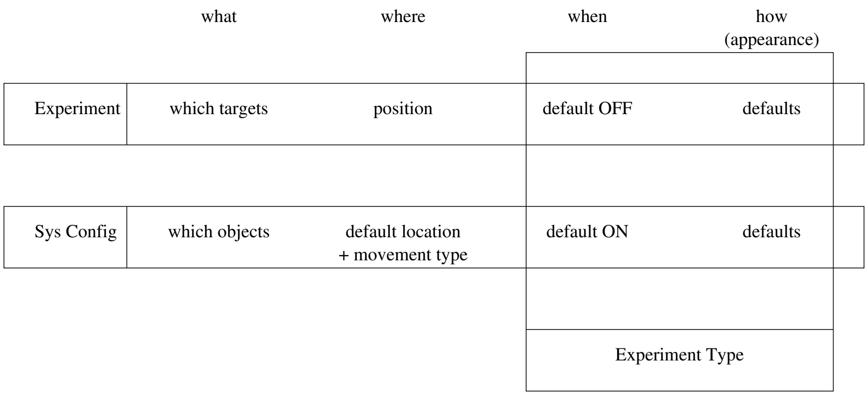

Figure 23: *The timing and appearance of graphical objects are controlled by a combination of the experiment instance, experiment type and system configuration.*

The flexibility of the experimental system is expanded when variability is provided at several levels. For example, the current apparatus supports both virtual and augmented environments depending on whether one can see through the mirror to the physical objects beneath it. The experimental control provides a variety of mechanisms to produce different experiments.

References

[Atkeson and Hollerbach, 1985] Atkeson, C. and Hollerbach, J. (1985). Kinematic features of unrestrained vertical arm movements. *Journal of Neuroscience*, 5(9):2318–2330.

[Azuma and Bishop, 1994] Azuma, R. and Bishop, G. (1994). Improving static and dynamic registration in an optical see-through HMD. In *Proceedings of SIGGRAPH '94*, pages 197–204.

[Graham, 1995] Graham, E. D. (1995). *Pointing on a Computer Display*. Ph.D. thesis, Department of Kinesiology, Simon Fraser University.

[Griswold, 1998] Griswold, W. G. (1998). Coping with software change using information transparency. Technical Report Technical Report CS98-585, Department of Computer Science and Engineering, University of California, San Diego.

[Northern Digital Inc., 1992] Northern Digital Inc. (1992). *OPTOTRAK Programmer's Guide*, version 0.9 edition.

[Summers et al., 1999] Summers, V. A., Booth, K. S., Calvert, T., Graham, E., and MacKenzie, C. L. (1999). Calibration for augmented reality experimental testbeds. In *1999 ACM Symposium on Interactive 3D Graphics, Atlanta, Georgia*, pages 155–162.

[Wang, 1999] Wang, Y. (1999). *Object transportation and orientation in virtual environments*. Ph.D. thesis, School of Kinesiology, Simon Fraser University.

[Welch, 1996] Welch, G. F. (1996). *SCAAT: Incremental Tracking with Incomplete Information*. Ph.D. thesis, Department of Computer Science, UNC-Chapel Hill.