An Augmented Vision System for Industrial Applications

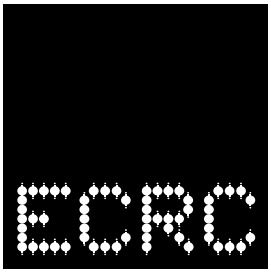
Klaus Ahlers
David Breen
Chris Crampton
Eric Rose
Mihran Tuceryan
Ross Whitaker
Douglas Greer

ECRC-94-39

Technical report ECRC-94-39

An Augmented Vision System for Industrial Applications

Klaus Ahlers
David Breen
Chris Crampton
Eric Rose
Mihran Tuceryan
Ross Whitaker
Douglas Greer



European Computer-Industry Research Centre GmbH (Forschungszentrum)
Arabellastrasse 17
D-81925 Munich
Germany
Tel. +49 89 9 26 99-0
Fax. +49 89 9 26 99-170
Tlx. 52 69 10

For more information please contact : Douglas S. Greer
dsg@ecrc.de

Abstract

This paper describes the major components of the Grasp augmented vision system. Grasp is an object-oriented system written in C++, which provides an environment both for exploring the basic technologies of augmented vision, and for developing applications that demonstrate the capabilities of these technologies. The hardware components of Grasp include video cameras, 6-D tracking devices, a frame grabber, a 3-D graphics workstation, a scan converter, and a video mixer. The major software components consist of classes that implement geometric models, rendering algorithms, calibration methods, file I/O, a user interface, and event handling.

1 INTRODUCTION

Although it is easy to believe that the human brain is devoted largely to some form of abstract reasoning, the majority of our central nervous system is actually involved primarily with vision and motor control. These two functions are concerned with processing information about three-dimensional space, and although we are generally unaware of it, we are extremely proficient at understanding and working with spatial information. For this reason, computer graphics and three-dimensional user interaction devices are a very natural and efficient medium for human-computer interaction.

One of the new paradigms for making $3D$ information quickly accessible, immediately understandable and easily modifiable is called augmented vision (AV)*. The goal of augmented vision is to take advantage of human spatial reasoning proficiency and to enhance the capabilities of the human visual system through the combination of computer generated graphics, computer vision and advanced user interaction technology. This concept is realised by superimposing information in the form of a three-dimensional computer generated image on top of a "real-life" visual scene, and allowing a user to interact with both. Information stored in a database or potentially derived from a computer vision system can then be used to provide the human viewer with additional information about the scene that would not otherwise be apparent.

This paper describes the current effort to build the Grasp augmented vision system at the European Computer-Industry Research Centre [1]. Grasp is an object-oriented system written in C++, which provides an environment both for exploring the basic technologies of augmented vision, and for developing applications that demonstrate the capabilities of these technologies. The hardware components of Grasp include video cameras, 6-D tracking devices, a frame grabber, a 3-D graphics workstation, a scan converter, and a video mixer. The major software components consist of classes that implement geometric models, rendering algorithms, calibration methods, file I/O, a user interface, and event handling. It is this combination of trackers, cameras, video mixing, and real and virtual world calibration, along with conventional real-time graphics, that distinguishes Grasp from other $3D$ computer graphics systems.

There are numerous possible industrial applications for an augmented vision system. The remodeling of an existing room may involve an AV system for locating pipes and wires, and laying out new communication lines or structural supports. An interior design application may utilize augmented vision for the selection and placement of furniture, lighting fixtures, drapes, and wall hangings in an empty room. In computer-aided design, AV may be used to modify the design of an aircraft, a car or a train, or to plan a new construction site. A fashion designer could interactively design clothing and have the

*We use the term "Augmented Vision" instead of "Augmented Reality", because the major focus of our work is the integration computer vision techniques into this developing technology.

pattern transmitted to a tailor who would "see" the new pattern superimposed on a piece of material. Augmented vision could also allow a customer to "try on" any piece of clothing from an extensive catalogue, or have the results of a tailor's alteration seen, before any cloth is cut. Repair crews or operators of large earth-moving machinery could "see" the location of underground pipes and power-lines. Mechanics could receive additional information when repairing complicated mechanical devices. Any task that requires or utilizes 3-D spatial information could conceivably benefit from augmented vision.

2 PREVIOUS WORK

Several research groups are currently exploring Augmented Reality (AR) for a variety of applications. Feiner et al. [7] have developed a knowledge-based AR system for maintenance and repair instruction at Columbia University. Lorensen et al. [11] of General Electric have focused more on AR for medical applications. At Boeing [13], AR is being developed to assist in manufacturing processes. A group at the University of North Carolina [4, 5] has also explored medical applications and has conducted extensive work in tracking technologies for AR. Deering [6] of Sun Microsystems has studied the problems associated with achieving high resolution head-tracked stereo display for AR. Grimson et al. [9] of MIT present a method for automatically registering clinical data from MRI or CT scans with a patient's head on an operating table.

3 HARDWARE CONFIGURATION

Developing a complete augmented vision system involves bringing together a number of technologies, including distributed computing, nomadic head-mounted-displays, large databases and geometric constraints. While these are all very important components, our first prototype is more concerned with the necessary underlying computer graphics, computer vision and user interaction foundation. In the initial stage of our work, the mechanical design of the user head-mounted-display was factored out. In its place, a standard video camera was used and the computer-generated graphics combined with video signal input; the result of which is presented on a normal video display. This temporarily eliminates the need for eye tracking and alignment, but the basic computer graphics and spatial tracking problems are the same in either case. This type of system, where the output is displayed on a standard television monitor rather than special "glasses", worn by the user, may actually be preferable in some applications.

The hardware configuration is illustrated in Figure 3.1. The graphical image is generated by the workstation hardware and displayed on the workstation's high resolution monitor along with auxiliary control information. A scan converter takes the relevant portion of the graphical image and converts it to

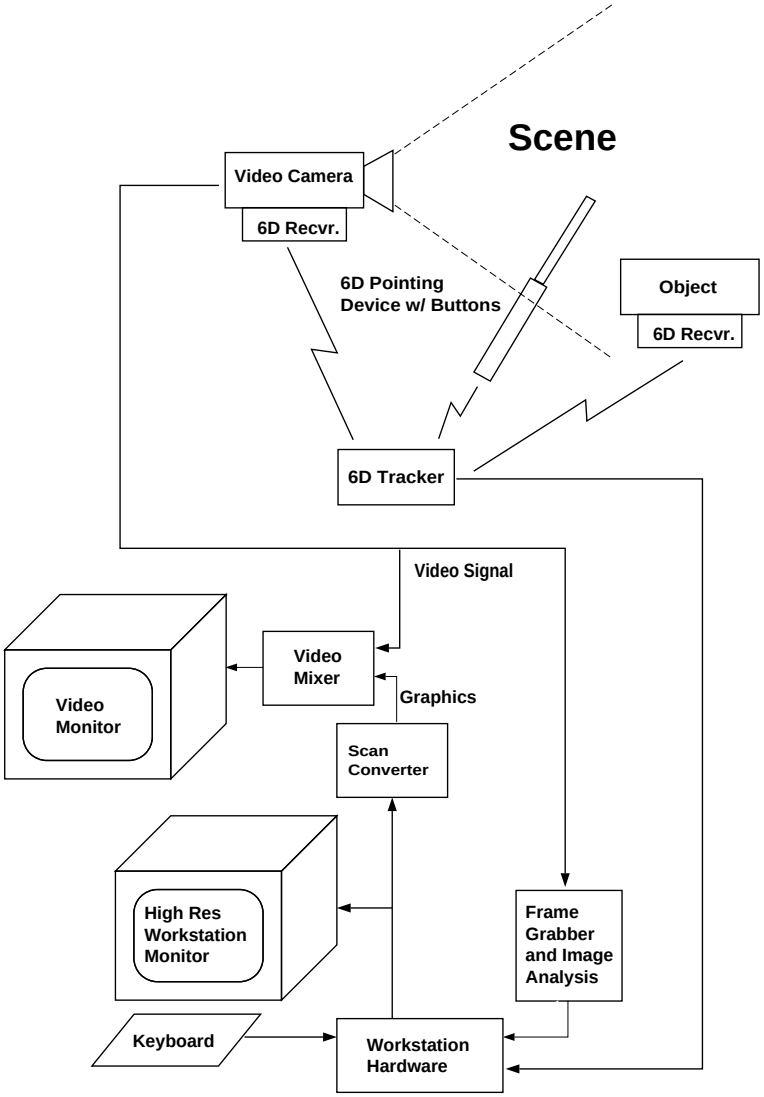


Figure 3.1: Hardware Configuration

standard video resolution and format. The scan converter also mixes this generated video signal with the video signal input from the camera. A six degrees-of-freedom ($6D$) tracker, which is capable of sensing three translational and three rotational degrees of freedom, provides the workstation with continually updated values for the position and orientation of the video camera, a tracked object and the pointing device. The information about the camera's position and viewing angle is used to keep the overlaid graphics in correct alignment with the visual image. A frame grabber is used during the initial calibration procedure as part of the process to determine the optical characteristics of the camera, and the initial transformations of the various objects in the scene.

The workstation used for the first prototype is a SPARCStation 10 ZX, supplied by Sun Microsystems, Inc. The scan converter is an "Otto" Graphics Converter Model 9500 made by Folsom Research Inc. This scan converter model includes

a video key input that is used for mixing the two video signals. The $6D$ tracker is an Ascension Technology Corporation "Flock of Birds". The frame grabber is a VideoPix SBus card, also from Sun Microsystems, Inc.

4 SOFTWARE COMPONENTS

The Grasp system is written using the C++ programming language [15] and follows the principles of object-oriented design. This section describes the classes used to encapsulate the functionality of the system, and the functions and methods that operate on their data. The classes define geometric and scene models, rendering methods, calibration techniques, file I/O, a user interface, and event handling.

4.1 Geometric and Scene Modeling

At its core, Grasp is an interactive $3D$ computer graphics system. Therefore Grasp provides methods for representing and viewing $3D$ geometric models. Grasp supports a wide variety of geometric representations at several levels of complexity. Points, vectors, and matrices comprise the lowest level representation. Points may be combined to create different types of polygons, and to define higher-level curved surfaces. Numerous solid primitives are also available. These primitives may be hierarchically organized into more complex structures. The attributes that define their color, surface and shading properties may be modified. Once the geometry and its shading properties have been defined, lights and cameras provide the lighting and viewing information needed by the renderers for image generation.

A general purpose graphics system requires a basic set of objects to represent position, direction, and spatial transformations. The *Point3* class is used to represent a position in three-dimensional space using the standard X, Y and Z Cartesian coordinates. The *Direct* class, a subclass of *Point3*, is used to describe three-dimensional vectors that indicate direction or orientation. Homogeneous three-dimensional coordinates are represented with the class *Point4*. *GfxBBox* is a 6-value bounding box. 4×4 matrices and their associated matrix and vector operations are implemented by the class *Matrix4*. The *GfxTransform* class, which is a subclass of *Matrix4*, is used for representing spatial transformations of three-dimensional homogeneous coordinates where the bottom row of the 4×4 matrix carries the translation component and upper-left 3×3 sub-matrix is the rotation and scaling component. The *Quaternion* class provides an efficient and compact representation of rotation operations using a four-dimensional vector.

The *GfxCamera* class is an implementation of a digital camera based on the RenderMan model [16]. It stores not only the camera position, but information

about camera parameters such as the aspect ratio and focal length. The camera also maintains clipping information, and some information about the $2D$ window used for rendering. A *GfxScene* stores information necessary to render a scene – the lights, the camera position, and a geometric object hierarchy. The scene also coordinates the picking operation between the renderer and the geometry tree.

Lights are necessary to enhance an object's $3D$ appearance, for example, with diffuse shading and specular highlights. Four kinds of lights are supported by the Grasp system: ambient, point, distant, and spot. The light objects for a particular scene are collected together using a *GfxLights* container object; scene objects have one reference to a *GfxLights* container. Within the geometry-tree, the light states are controlled using *GfxLightState* objects, associated with the node-attribute objects. Light objects incorporate a transform, although this only affects lights that have a position or direction. The effect of lights on a scene is renderer-dependent. Some renderers may not support all varieties of lights, but in such cases a reasonable approximation will be made.

GfxGeomObj is the base class for objects that comprise the geometric object tree. Geometric objects are stored in a tree-structure, with leaf objects (*GfxLeafObj* and its subclasses) and composite objects, *GfxCompObj*. Composite objects store a list of children and are used to group objects together that share a common transform and attribute set. A transform object, *GfxTransform*, can be attached to a composite object and applies to all of the node's children. Attaching an attribute object (*GfxAttrObj*) sets attributes that, similarly, apply to the node's children. For example, if the *Color* attribute is set in a node, then all of the node's children will inherit this color unless a child-node overrides the color attribute.

Colors, shading, and other rendering parameters are called *Attributes* in Grasp. Some may be represented by simple types: boolean, int, unsigned, etc. Others are more complex and require a collection of information; for example, the *GfxColor* class. *Attributes* allow the user to define many effects produced during rendering, for example, color, flat vs. smooth-shading, opacity, texture mapping, geometric approximation level, illumination, and shading effects

Grasp supports a wide variety of geometric entities, derived from the *GfxLeafObj* class and called graphical objects, that may be rendered. The definitions of the three dimensional objects are based on the geometric primitives of the RenderMan system [16]. Two-dimensional primitives have also been included in Grasp. The graphical objects can be divided up into five categories:

- Lines — Objects that can be represented as lists of points.
- Polygons and Triangle Strips — Triangle strips and various types of polygons.

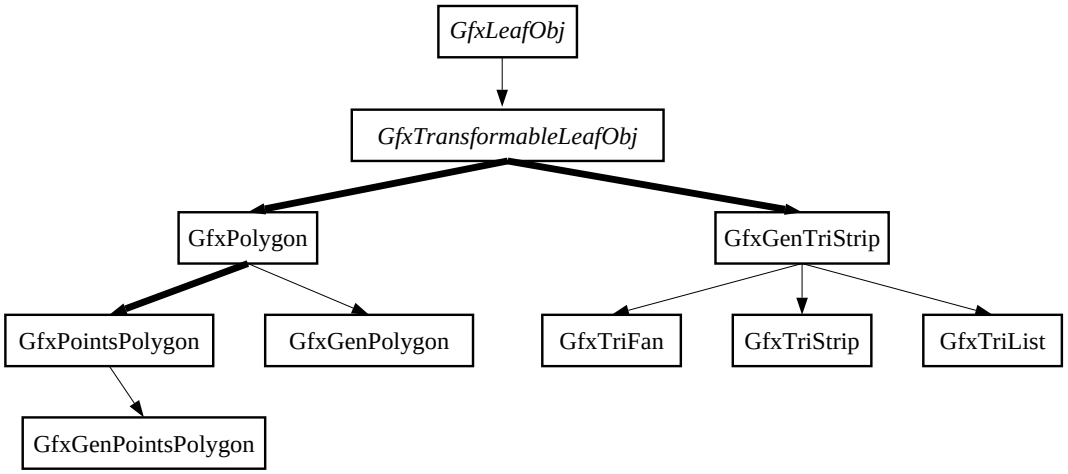


Figure 4.1: Polygon and TriStrip Class Inheritance Hierarchy

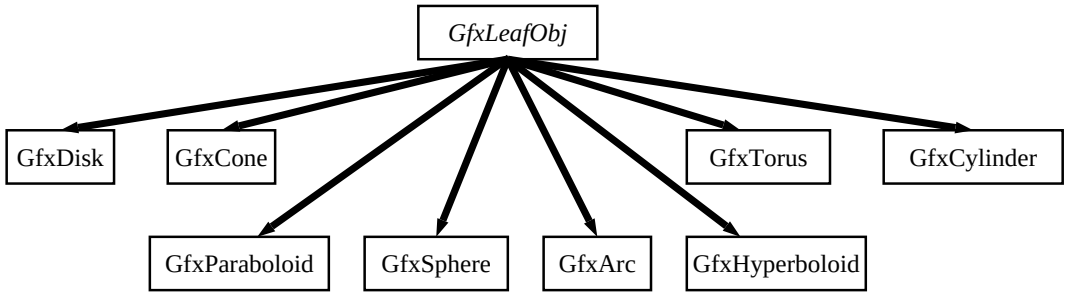


Figure 4.2: Analytical Objects Class Inheritance Hierarchy

- Analytical Objects — Objects such as spheres, cones, or tori.
- Patch and Patch Mesh Objects — Different types of patch surfaces.
- Miscellaneous Objects — For example, text.

Lines are represented by the *GfxPolyline* class. A simple polygon (*GfxPolygon*), a simple polygon with holes (*GfxGenPolygon*), a set of polygons with (*GfxPointsPolygon*) and without holes (*GfxGenPointsPolygon*) have all been implemented. Collections of triangles have also been implemented as strips (*GfxTriStrip*), fans (*GfxTriFan*), and lists (*GfxTriList*). The class hierarchy for these graphical objects is presented in Figure 4.1. *Analytical* objects are those whose shape can be represented by an equation. This includes traditional conic sections such as cones, hyperboloids and paraboloids, as well as other quadratic representations such as spheres and tori. The class hierarchy for these graphical objects is presented in Figure 4.2.

Patches and Patch Meshes are important objects particularly with respect to modeling realistic shapes. In general, a patch mesh is a collection of patches with shared control points, forming a single surface. Within Grasp a variety of surfaces are available. These surfaces may defined with either bilinear

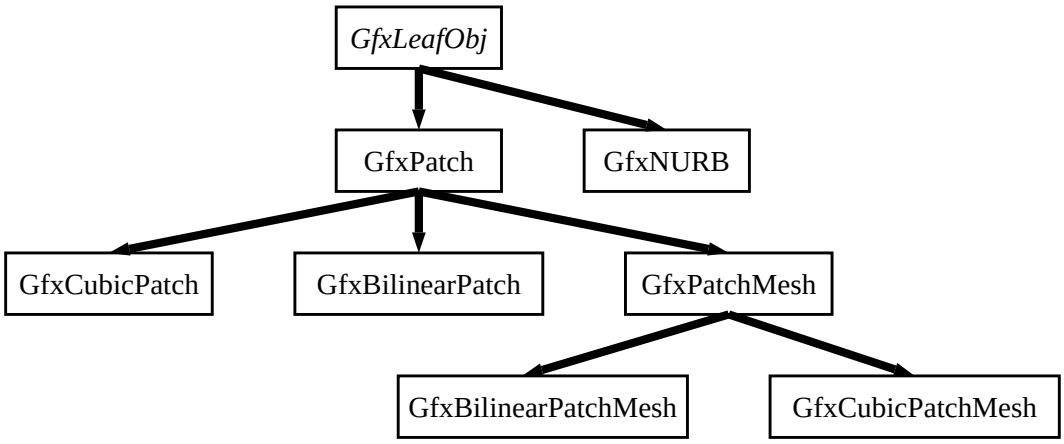


Figure 4.3: Patch Object Class Inheritance Hierarchy

(*GfxBilinearPatch*) or bicubic (*GfxCubicPatch*) patches, and may be collected into meshes (*GfxBilinearPatchMesh*, *GfxCubicPatchMesh*). The class hierarchy of these objects is presented in Figure 4.3. As in RenderMan, bicubic surfaces may be defined in a variety of bases, including Bezier, B-spline, Beta-Spline, Catmull-Rom, Hermite, power, and Cardinal.

4.2 Rendering

The Grasp rendering system is implemented as a separate module with a clearly defined interface. This interface has been designed to provide access to the underlying functionality without exposing any details of the implementation. The reason for this careful design is to ensure that new renderers can be implemented and added to the system without requiring any changes outside of the rendering system itself. Currently three renderers have been implemented, a RIB renderer, a stream renderer, and an XGL renderer.

Care has been taken to ensure that the rendering mechanism can function as efficiently as possible, which is an important consideration for the interactive applications intended to be supported by the Grasp system. In particular, it is assumed that the rendering operation is applied many times to the same objects thereby implying that the caching of any intermediate state will yield a better performance.

Viewed from the outside, the rendering process can be broken up into two main phases: firstly, the leaf objects (i.e. geometric primitives) use a renderer object to construct a renderer-dependent representation of themselves (termed the *set-up* operation). This phase usually involves tessellating complex primitives into simpler entities. Secondly, the leaf objects use the same renderer object and the intermediate representation to perform the actual output operation (termed the *drawing* operation).

The renderer interface is defined by the abstract class *Renderer*. This interface can be sub-divided into the following components: control, context, attributes, transforms, rendering, interaction support, and renderer-dependent representations. A small number of methods are used to control the internal operation of the renderers. The camera methods are used to control the rendering viewport, position and attributes for a particular rendering of a scene. Other methods control double buffering, and internal state initialization. The context methods are used to step the renderer through the various levels of initialization, reflecting the hierarchical structure of the internal model. Numerous attributes are set and reset during the rendering of a scene with the attribute methods. As with the attributes, the current transform can be controlled using a group of methods which sets the transform directly or which saves the current transform so that it can be later restored. For each of the geometric primitive types, there is a corresponding rendering method available in the renderer, but this does not mean that all renderers must provide support for all primitives. The principle is that each renderer should provide support for as reasonable a subset of the primitives as is appropriate with respect to the underlying rendering system being used. For example, a simple renderer may only implement triangle strips and lines and this will suffice for most applications as the more complex geometric classes "know" how to break themselves up into the more basic primitives. These methods do not perform any rendering as such; their function is to generate a renderer-specific representation of the primitive which can be cached for use by the actual rendering operation.

In order that a particular class of renderer can be used within an interactive application, a set of methods that implement picking must be available. The picking process is very similar to the rendering process except that the renderer only checks for objects that intersect with the picking volume (the objects are **not** actually drawn). If a primitive intersects with the pick volume, the current pick identifier information is added to the internal pick buffer. Frequent reference has been made to the renderer-dependent representations of the geometric primitives that are cached inside the primitive objects. The principle here is that a renderer will need to create some intermediate representation of a primitive before it can be rendered. Typically, this would be a renderer-specific data structure corresponding to the primitive. Once cached, this representation can be used repeatedly as long as the renderer *type* does not change. When a different renderer type is encountered the cached representation must be deleted (by the primitive object, not the renderer) and a new renderer-dependent representation created and cached.

The *RIB Renderer*, implemented by class *GfxRibRenderer*, produces a RIB [12] representation of a scene. The RIB data is generated in a textual form and this text is written by the renderer to an output stream, as defined by the *C++ IOStream* library [15]. Output streams can be connected to files (`ofstream` objects) or may perform output directly into a memory buffer (`ostrstream` objects). The RIB renderer supports all of the geometric primitives with the

exception of lines, text and triangle strips, as these are not supported by the RIB format per se. The *Stream Renderer*, implemented by class *GfxStreamRenderer*, is a subclass of the *RIB Renderer*. It additionally provides support for the primitives not supported by RIB and generates a slightly modified form of RIB. No new methods are added to the public interface with respect to the RIB renderer.

The *XGL Renderer* provides an interactive rendering capability within Grasp. XGL is the graphics library supplied by Sun Microsystems Inc. that provides a $2D$ and $3D$ rendering environment for graphical applications. XGL is an appropriate rendering interface for the Grasp system as it efficiently and transparently utilizes any underlying hardware graphics accelerators. XGL provides a set of basic $3D$ objects, including lines, polygons and triangle strips. Several more complex geometric primitives, such as arcs and disks, are supported, but as XGL breaks these down internally into the simpler (optimized) primitives, they are not as efficient to use in an interactive application. Altogether, the capabilities of XGL fit well to the Grasp rendering model and when the potential efficiency of XGL applications is taken into account (given appropriate graphics accelerator hardware), it is clear that XGL is very suitable for the interactive applications supported by Grasp. The modeling and rendering constructs of our system map cleanly into the attribute, light, rendering primitive, vertex information, bounding box, shading, text, and bounding box constructs supported in XGL.

4.3 Calibration

There are a number of coordinate systems that are defined in Grasp, each is tailored to a specific modeling task. For example, there is a coordinate system defined for objects, the use of which makes object-modeling natural and self-contained. Similarly, there are coordinate systems defined for the camera, the $6D$ tracker and the world. As in the case of the object-centered coordinate system, defining these local coordinate systems makes the usage of the corresponding objects, devices, etc. easier in a particular context. The mathematical models of each of these coordinate systems and their respective geometries must be rigorously defined and, in addition, they must be related to each other. This final process of registering the different coordinate systems within the Grasp system with each other and real world objects is the goal of calibration. Calibration is one of the fundamental components of augmented vision, and provides some of its most challenging problems.

Figure 4.4 shows the main coordinate systems used in the Grasp system, and the major relationships or transforms between them. For example, the arc labeled "A" represents the object-to-world transform. The central reference coordinate system is the "World" coordinate system which is at a fixed and known location relative to the containing environment. The reference system used by the six-degrees-of-freedom ($6D$) tracker is referred to as the "Tracker"

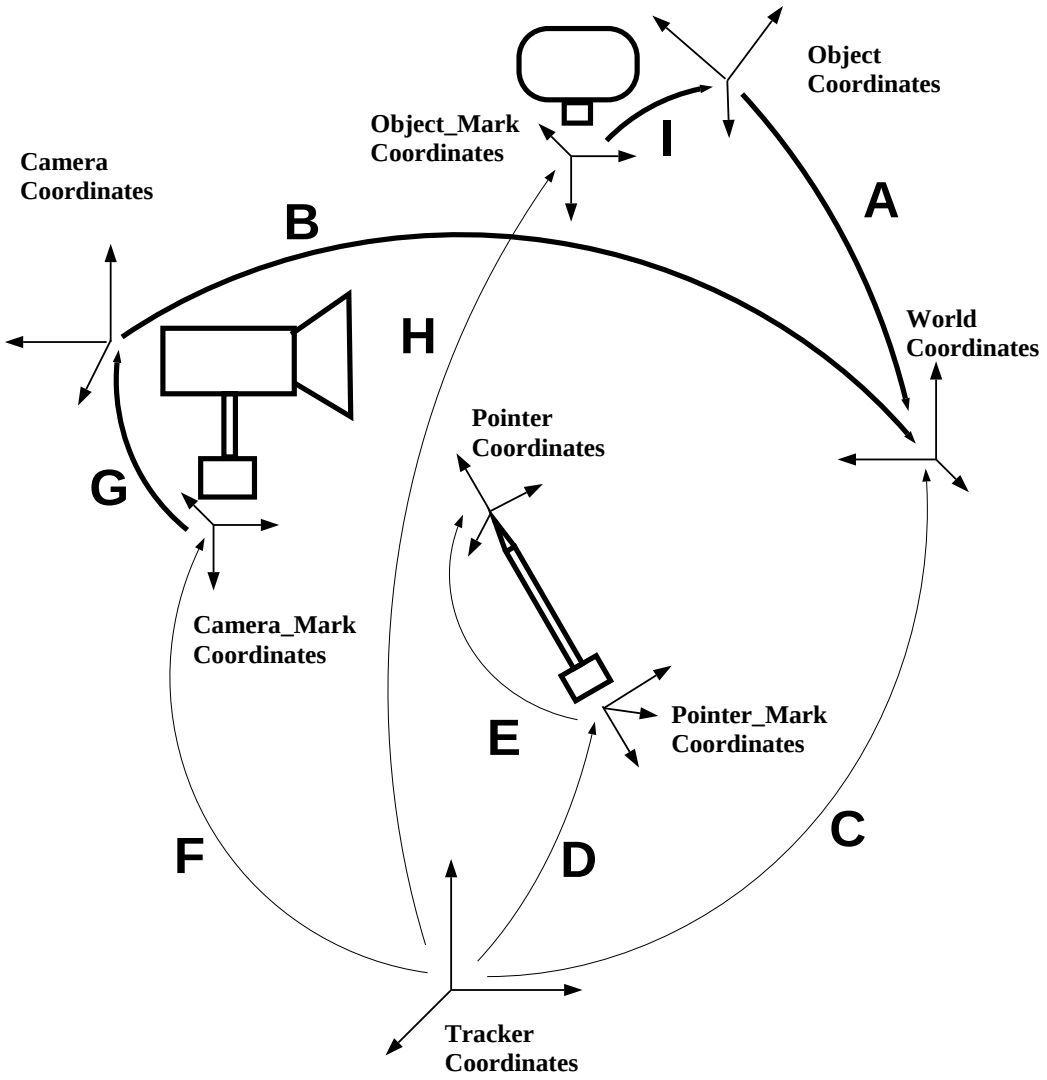


Figure 4.4: Coordinate System Transformations

coordinate system which also remains at a fixed location while the system is operating. Its relationship to world coordinates is defined by the tracker-to-world transform which is labeled “C” in Figure 4.4. This transform is computed during the tracker calibration procedure described below.

The 6D tracker is capable of determining the position and orientation of multiple receivers or “Marks”. In the Grasp system, marks are rigidly attached to the camera, the pointing device and an object, and their locations define the time-varying (non-fixed) camera-mark, pointer-mark, and object-mark coordinate systems. The 6D tracker hardware can determine their instantaneous location and orientation, and from this information the tracker-to-camera-mark transform (labeled “F”), the tracker-to-pointer-mark transform (labeled “D”), and the tracker-to-object-mark (labeled ”H”) can be computed.

The locations of most interest to applications are not the marks themselves, but

rather the position of the camera, the end of the pointing device and the actual position of the object. The relationship between these two, the pointer-mark-to-pointer transform, labeled “E” in Figure 4.4 and the object-mark-to-object transform, labeled “I”, are calculated during the tracker and object calibration procedures.

The extrinsic camera parameters, determined during the camera calibration procedure described below, can be used to compute the camera-to-world transformation which is the arc labeled “B”. The camera coordinates are located somewhere within the camera itself and consequently, the camera-mark-to-camera transform cannot be directly measured. However, when both the camera and tracker calibration procedures are completed, it can be computed indirectly by composing (multiplying) known transformations.

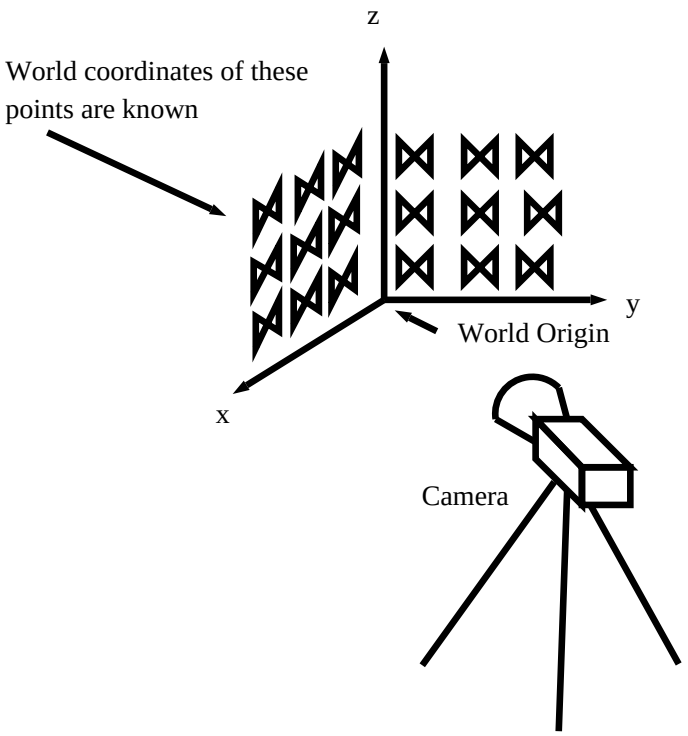


Figure 4.5: The set-up with which the camera is calibrated.

Currently camera calibration is performed at the beginning of a session and the measured model parameters are used throughout the subsequent operation. The calibration is conducted using a predefined calibration grid that is set up at a known location in the “world”, as shown in Figure 4.5. The calibration grid consists of two orthogonal planes that define a $3D$ coordinate system. These two planes are marked with “butterfly” patterns, the centers of which are at known $3D$ positions within this coordinate system.

The camera calibration process requires the following steps:

1. The camera is pointed at the calibration grid.

2. The centers of the butterfly patterns in the image are identified, thereby obtaining the $2D$ image coordinates corresponding to the known $3D$ image points. There are two ways that these image points could be identified:

   (a) Automatic detection involving image processing.

   (b) Have the user specify these image points interactively by clicking on them with the mouse.

3. Compute the camera parameters using the input data, where (x, y, z) are the world coordinates and (r_i, c_i) are the corresponding row and column coordinates of the points in the image.

The camera calibration procedure [18] amounts to collecting $3D$ points (x_i, y_i, z_i) and their corresponding coordinates in the digitized image (r_i, c_i), and computing the *extrinsic parameters* R and T (camera-to-world transformation), and the *intrinsic camera parameters* r_0, c_0 (image plane origin in pixel coordinates), f_u, and f_v (focal length of the camera in the u and v direction).

Tracker calibration uses the same calibration grid as the camera calibration, since, once again, we need to use the $3D$ coordinates of known world points. The calibration procedure itself consists of making a number of selections from the calibration grid by pointing at known locations with a pointer device. Several of the selections are made by pointing at the same calibration point with different pointer orientations. The selections are triangulated using a least-squares formulation to determine the length of the pointer. The two remaining selections are points which define the x- and z-axes. The mathematical details of this procedure and camera calibration may be found in an ECRC technical report [1].

Object calibration, calculating the object-mark-to-object transformation, may be performed with two techniques. The first technique is similar to camera calibration where known $3D$ points in the object's coordinate system are associated with their $2D$ projections in a grabbed image. The second technique involves associating the same object space $3D$ points with another set of $3D$ points in the world coordinate system. The world coordinate points are acquired using the calibrated $6D$ pointing device.

All of the calibration techniques in Grasp have been encapsulated within classes that are subclasses of *GfxCalibration*. The number and variety of these techniques are evident in Figure 4.6. There are separate procedures for calibrating a single mark, an image, a tracked object, the mark on the camera, the camera itself, and the pointing device. The calibration may be processed in world coordinates ("Wc") or the coordinate system of the entity being calibrated. The technique may be manual ("Man"), i.e. needing user intervention with a pointing device ("Ptr") or with the camera ("Cam"), or it

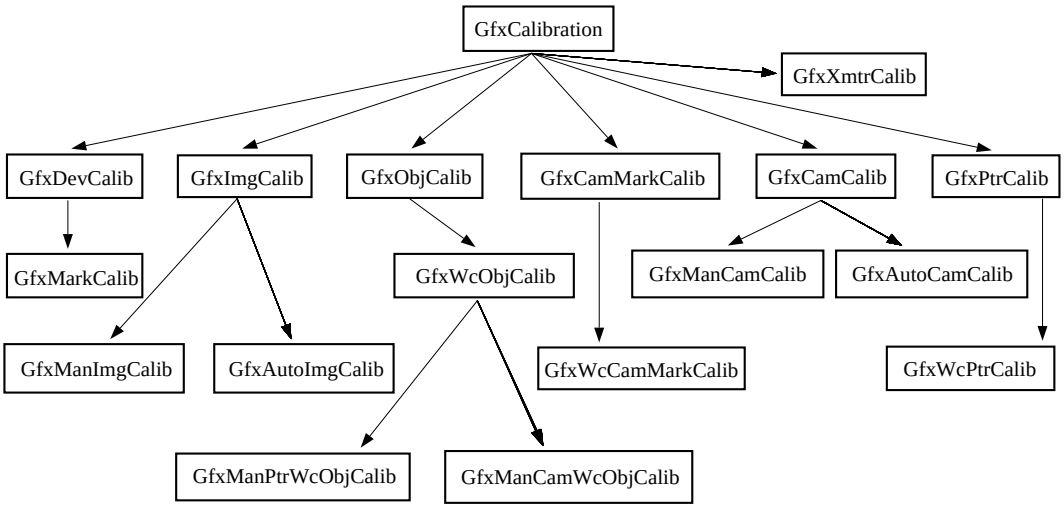


Figure 4.6: Calibration Class Inheritance Hierarchy

may be automatic ("Auto"), when the system is able to locate calibration information with vision techniques in grabbed images, without user input.

4.4 File I/O

The need for some external data representation from which models can be imported and to which data can be exported, is two fold. Firstly, although the current Grasp applications are not editors as such, they do provide some limited editing functionality, namely the modification and manipulation of attributes and objects. Therefore, it follows that there is a requirement for users to be able to save and restore the internal state of their applications. Secondly, as the current Grasp system is not intended to provide a fully functioning editor of $3D$ scenes, graphical models will have to be created using external tools. If the exchange format used for importing data from external packages is chosen carefully, this mechanism will allow users to access a wealth of existing graphical models.

As a RenderMan-like model has been generally adopted for the internal model of the Grasp system, it was a logical step to adopt a file format resembling the RenderMan Interface Bytestream (RIB) [12] as the principal format for data interchange. RIB provides a rich set of primitives and operators while having a well defined, hierarchical syntax that is easy to read and parse. Simple models can be created using a text editor and it is possible to manually fine-tune more sophisticated models. In addition to the "standard" RIB constructs, the Grasp RIB reader supports a number of Grasp-specific extensions.

However, while RIB is a suitable format for use by the Grasp system, few third party CAD-like tools provide direct support for RIB as a data export format. Consequently, a different format has to be used for importing CAD data into the Grasp system and for this purpose the DXF format [3] was selected as being

the most suitable. DXF has been defined by AutoDesk, Inc. and is the preferred format for the import and export of data to and from the AutoCAD program as well as a large number of other CAD tools and drawing packages. Unfortunately, DXF is not well suited as the principal format for the Grasp system, as it is not hierarchical and supports only a limited subset of the primitives, attributes and operators required by Grasp applications. There is no support for exporting DXF data from the Grasp applications.

4.5 User Interface

User interface toolkits for window-based environments provide a framework for the construction of interactive components of application programs. The support often goes beyond providing just simple interface objects and methods for input handling — a more comprehensive toolkit can be provide assistance in presentation layout, dialog control, user customization, and the operating system interface.

Although the core of the Grasp system is independent of any window system toolkit, it is likely that most applications will require some means of presentation for interactive graphics and visualization. Grasp supports this by including some user interface toolkit independent concepts and building blocks that are useful for such applications. However, viewing and event handling have to be integrated with the underlying window system, which requires a toolkit of some kind, and application programs will also use this toolkit for building their user interface. Therefore, at some level, a degree of dependence on the window system toolkit is unavoidable. Grasp is currently implemented using the InterViews toolkit [10]. Hence, the functionality offered by InterViews is implicitly available to Grasp applications.

Window system toolkits often impose a style for the design of application programs and the use of library objects, and this is true for the implementation of Grasp based on InterViews. Going against the grain of the toolkit is sometimes possible but may lead to difficulties later. Figure 4.7 depicts a run-time structure composed of Grasp user interface components that represents the "recommended" *conceptual* design for the architecture of an application.

The root of the application architecture is an instance of class *GfxSystem*, one of which has to be created for every Grasp session, and it is beneath this object that the *GfxApplication* objects are attached. One *GfxApplication* instance corresponds to each distinctive graphical editing or visualization task that coexists in a single session. *GfxViewer* objects represent graphics viewers that cooperate within one application. Viewers can either occupy part of the application window itself or use other top-level windows created as *GfxFrame* objects.

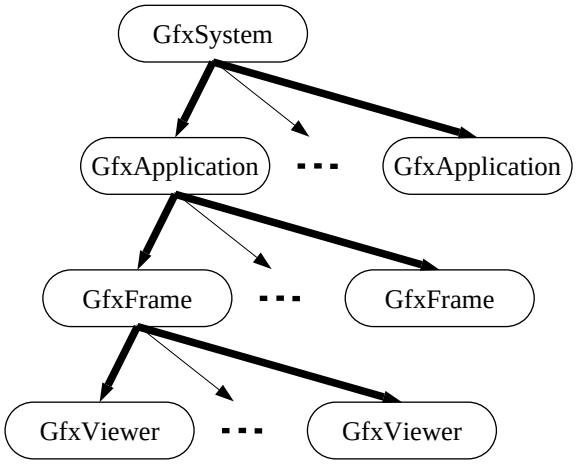


Figure 4.7: Logical view of runtime architecture.

The logical structure of an application does not specify the physical layout of the windows and interface objects. For Grasp applications, there are two things to remember when building the InterViews glyph hierarchy that describes the window contents. Firstly, *GfxViewer* objects do not implement event handling; for interaction tasks the viewer object has to be placed in the body of a *GfxViewerInput* object. Secondly, independence of the toolkit from the $3D$ renderers makes it necessary for viewers to possess their own (sub)window that can be handed over for the exclusive use of the rendering system.

4.6 Event Handling

Events are objects that represent external, asynchronous happenings and are usually generated as a result of user interaction. Grasp is able to handle a variety of events and peripheral devices. Grasp uses the InterViews user interface toolkit which on receipt of user input actions, as well as system events, compiles them into input event objects that are forwarded to the appropriate application modules. The class *Event* defines a structure that contains the information describing a window system event. Depending on the type of the event, this information may include a window identifier, some device status information, a time stamp, some pointer (mouse) location information, some key codes, etc., or some combination thereof. An *InputHandler* is a monoglyph that processes events directed at its body glyph and translates them into more meaningful operations such as *press*, *release*, or *keystroke*. InterViews automatically forwards input events directed at a display window to the input handler appropriate for the corresponding display area. Class *ActiveHandler* is subclassed from *InputHandler*. In addition to the functionality of an input handler, it uses the methods `enter()` and `leave()` to define any action that should be performed when motion events indicate that the pointer focus has entered or left the area of the body. There is one, unique *Dispatcher* object per InterViews session. The *Dispatcher* integrates window

system events with timeout signals and events generated by asynchronous communication channels. Objects that are interested in any such events register themselves with the dispatcher and are notified, via a callback mechanism, whenever a relevant event is generated. For applications interested in the events generated by a file descriptor, the *IOHandler* class is provided, whose `inputReady()` method, for example, is called when input is available from the file descriptor.

Tool objects translate a sequence of user input actions into the desired command or effect. They allow the user to manipulate graphical objects directly. Viewer input handlers use *Tool* objects to execute a single interaction task such as selecting or moving an object. The handler either has a tool object already available, or requests the current tool from the application. When the tool has been found, it is first used to create a *Manipulator* object that monitors the whole interaction sequence. The manipulator divides an interaction sequence into an initial trigger event, a sequence of manipulation events, and a terminating event. For mouse input this usually translates directly into press, drag, and release. The tool finally *evaluates* the manipulator to produce a command that represents the result of the interaction. The use of tool objects to implement interaction in the framework of graphics viewers is similar to the tool abstraction used in Unidraw [17].

Manipulators encapsulate the mechanics of direct manipulation. It was mentioned above that a manipulator distinguishes three phases of interaction: the initiating event that caused the creation of the manipulator, subsequent events that are part of the manipulation, and the termination event. Some manipulators animate the manipulation operation by adding feedback objects to the display. The feedback objects are used to echo the current state of the interaction and indicate what effect the manipulation has had so far.

Feedback objects are containers for $2D$ or $3D$ graphical information that is used to give the user some form of feedback during the course of an interaction. These objects are temporarily attached to a viewer object which knows that it should use them for the display of auxiliary graphics depicting some intermediate state during a direct manipulation. When a scene is rendered, the geometric components of the scene are drawn first and then, while the rendered frame is still "open", the feedback objects are added.

The Grasp system includes support for peripheral devices, in particular those that are connected via a serial link interface. The class hierarchy for the device classes matches three levels of abstraction in the design: management of the serial port, implementation of the device interface, and integration with the user interface control.

The *Spaceball* [14] is a $6D$ input device that supports the manipulation of the position and orientation of objects in $3D$ space. A software library is included with the device that implements the communications interface and assists the programmer in integrating the spaceball with application programs. This

spaceball device can be considered to consist of three separate components: the ball, a keypad, and a tone-generating beeper. All three sub-components can be accessed using the supplied spaceball library.

The graphics scan converter device [8] converts high-resolution images from the display window into a standard video format. The Grasp system interacts with this device to perform "window tracking": whenever the display window size or position changes, the scan converter input area is adjusted to reflect the new values.

Tracking is performed using a "Flock of Birds"^{TM} $6D$ tracking system [2] that was obtained from Ascension Technology Corp. The Grasp system can monitor the tracker device both with a polling and streaming mechanism. During polling, a timer event triggers at regular intervals the reading of the current tracker position and orientation which is then translated into an event object and forwarded using the event handling mechanism previously described. During streaming, the tracker device broadcasts the mark values and button states, allowing for improved interactive functionality. Grasp distinguishes between camera tracking, and object and pointer tracking: the former provides input that is used to control the camera transformation of a scene, while the last two are used to update the position and orientation of a $3D$ object displayed in a viewer.

5 CONCLUSION AND FUTURE WORK

We have described the basic architecture of an augmented vision system (Grasp) suitable for a variety of industrial applications. The hardware components of Grasp include video cameras, 6-D tracking devices, a frame grabber, a 3-D graphics workstation, a scan converter, and a video mixer. The major software components consist of classes that implement geometric models, rendering algorithms, calibration methods, file I/O, a user interface, and event handling. We are currently developing a number of applications that demonstrate its capabilities. One application involves interactively identifying the parts of a complex engine for mechanical repair. Another application may be used for interior design by supporting the interactive placement of virtual furniture in an actual room.

6 ACKNOWLEDGMENTS

This work is financially supported by Bull SA, ICL Plc, and Siemens AG.

Bibliography

[1] K. Ahlers, C. Crampton, D. Greer, E. Rose, and M. Tuceryan. Augmented vision: A technical introduction to the grasp 1.2 system. Technical Report ECRC-94-14, ECRC, Munich, Germany, 1994.

[2] Ascension Technology Corp., Burlington, VT. *The Flock of Birds Installation and Operation Guide*, 1994.

[3] AutoDesk, Inc. *AutoCAD Customization Manual*, 1992.

[4] R. Azuma and G. Bishop. Improving static and dynamic registration in an optical see-through display. In *Computer Graphics (Proc. SIGGRAPH)*, pages 194–204, July 1994.

[5] M. Bajura, H. Fuchs, and R. Ohbuchi. Merging virtual objects with the real world: Seeing ultrasound imagery within the patient. *Computer Graphics (Proc. SIGGRAPH)*, 26(2):203–210, July 1992.

[6] M. Deering. High resolution virtual reality. *Computer Graphics (Proc. SIGGRAPH)*, 26(2):195–202, July 1992.

[7] S. Feiner, B. Macintyre, and D. Seligmann. Knowledge-based augmented reality. *Communications of the ACM*, 36(7):53–62, july 1993.

[8] Folsom Research, Inc., Folsom, CA. *Installation and Operation Manual for Otto Graphics Converter Model 9500*, 1993.

[9] W. Grimson, T. Lozano-Perez, W. Wells, G. Ettinger, S. White, and R. Kikinis. An automatic registration method for frameless stereotaxy, image guided surgery, and enhanced reality. In *IEEE Conference on Computer Vision and Pattern Recognition Proceedings*, pages 430–436, Los Alamitos, CA, June 1994. IEEE Computer Society Press.

[10] M. Linton, P. Calder, J. Interrante, Steven Tang, and John M. Vlissides. Interviews reference manual version 3.1. Technical report, Stanford University, 1992.

[11] W. Lorensen, H. Cline, C. Nafis, R. Kikinis, D. Altobelli, and L. Gleason. Enhancing reality in the operating room. In *Visualization '93 Conference Proceedings*, pages 410–415, Los Alamitos, CA, October 1993. IEEE Computer Society Press.

[12] Pixar. *The RenderMan Interface*, 1989.

[13] D. Sims. New realities in aircraft design and manufacture. *IEEE Computer Graphics and Applications*, 14(2):91, March 1994.

[14] Spacial Systems Inc., Concord, MA. *Spaceball Application Developer's Reference Version 4.0*, 1989.

[15] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, MA, 1991.

[16] S. Upstill. *The RenderMan Companion*. Addison-Wesley, Reading, MA, 1990.

[17] J.M. Vlissides and M.A. Linton. Unidraw: A framework for building domain-specific graphical editors. *ACM Transactions on Information Systems*, 8(3):237–268, July 1990.

[18] J. Weng, P. Cohen, and M. Herniou. Camera calibration with distortion models and accuracy evaluation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, PAMI-14(10):965–980, 1992.