OSGAR: A Scene Graph with Uncertain Transformations

Enylton Machado Coelho
Blair MacIntyre

GVU Center, College of Computing
Georgia Institute of Technology
Atlanta GA
{*machado, blair*}*@cc.gatech.edu*

Simon J. Julier

Virtual Reality Laboratory
Advanced Information Technology
Naval Research Laboratory
Washington DC
julier@ait.nrl.navy.mil

Abstract

An important problem for augmented reality is registration error. No system can be perfectly tracked, calibrated or modeled. As a result, the overlaid graphics will not align perfectly with objects in the physical world. This can be distracting, annoying or confusing. In this paper we propose a method for mitigating the effects of registration errors that enables application developers to build dynamically adaptive AR displays. Our solution is implemented in a programming toolkit called OSGAR. Built upon OpenSceneGraph (OSG), OSGAR statistically characterizes registration errors, monitors those errors and, when a set of criteria are met, dynamically adapts the display to mitigate the effects of the errors. Because the architecture is based on a scene graph, it provides a simple, familiar and intuitive environment for application developers. We describe the components of OSGAR, discuss how several proposed methods for error registration can be implemented, and illustrate its use through a set of examples.

1. Introduction

Registration errors can have a profound impact on the effectiveness of an augmented reality (AR) system. The purpose of many AR systems is to provide information to the user about objects by aligning graphics with those objects in the physical world. However, no AR system is perfect. Tracking systems cannot measure the pose of their sensors exactly. Internal system calibration parameters cannot be known perfectly and the world cannot be modeled precisely. As a result, the graphics will not align perfectly with the objects in the physical world. In some situations these errors can be little more than an annoyance. However, in other situations the annotations could be ambiguously placed (it is not clear what object they refer to) or appear to be placed on the wrong object altogether.

The conventional approach to registration errors is to consider them as a type of tracking problem. Apart from a few notable exceptions, none of them recent (e.g., [1], [13]), they are rarely addressed directly. The prevailing assumption seems to be that, given better tracking and faster computers, the major causes of registration errors will be overcome. However, we do not believe that this is the case, especially when one considers mobile augmented reality systems where one cannot rely on accurate, fixed infrastructure in carefully controlled settings.

We believe that a better approach is to assume that registration errors will be inevitable, and provide application developers with tools to help them understand and deal with these errors. In particular, we believe that any AR toolkit should also help developers choose and display annotations in such a way that the effects of registration errors are minimized.

Our first attempt at developing an adaptive user interface was to introduce the concept of a *Level of Error* 3D scene graph node [10]. Analogous to Level of Detail (LOD) nodes (that switch between different representations of an object based on the projected size of the object), an LOE node is a switch node that uses an estimate of the registration error of an object to select the appropriate annotation style for a particular object in a scene. In [11] we described an implementation of the LOE which considered the problem of estimating and adapting to the registration error of a single object. The only sources of error were due to the tracker and calibration errors.

However, despite its appeal to conventional graphics scene graphs, the LOE is not sufficient to handle all of the possible strategies that are needed to adapt to registration error. For example, the LOE considers each object individually. Furthermore, the LOE only allows a finite set of fixed

Proceedings of the Third IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2004)
0-7695-2191-6/04 $20.00 © 2004 **IEEE**

(a) (b) (c)

Figure 1. In (a) two drawers are labeled, but the target of the labels is ambiguous. In (b) all drawers are labeled, reducing ambiguity but cluttering the space. In (c) the two drawers are labeled with callout lines that point to unambiguous locations on the appropriate drawer.

displays, limiting it to handling a small number of cases.

In this paper, we present the design and implementation of a programming toolkit for AR, called *OSGAR*, that provides a general framework for propagating error estimates from an arbitrary collection of sources, and creating adaptive interfaces based on these estimates. It consists of three main components: an error propagation mechanism (which calculates the uncertainty at any point in the scene graph), a set of components for adaptation (such as replacing models of objects with callout labels and lines), and a set of common facilities required for many types of AR applications (including support for trackers, video-in-the-background and fiducial tracking).

The structure of this paper is as follows. Section 2 provides several motivating examples. Section 3 describes the sources of registration error we are concerned with. Section 4 describes the uncertainty representation and the mathematical framework used to compute the estimate of the registration error. The architecture and implementation of OSGAR is described in Section 5. This section also describes some of the techniques which can be used to improve the quality of the augmentation presented to the user as well as the quality of interaction. The limitations and future directions of OSGAR are discussed in Section 6 and conclusions are drawn in Section 7.

2. Motivating examples

While some AR domains require precise registration (e.g., AR-guided surgery [13]), there are many domains where AR could be usefully applied that do not require precise registration. For example, consider a system that tries to label two empty drawers in the tool cabinet in Figure 1(a). While the labels are small enough to fit within the projected area of the drawers, a small amount of registration error makes it unclear which drawer the labels are referring to. Furthermore, because all drawers look the same there are no obvious visual clues a user could employ to resolve the ambiguity. One solution, shown in Figure 1(b), is to label all the drawers, so the user can infer the registration error offset. However, even for this relatively small amount of registration error, adding all thirty labels unnecessarily clutters the user's view of the world. A better solution in this case, shown in Figure 1(c), would be to offset each label so it does not cover any part of the target drawer, and use a callout line to point to a location on the display that is likely to overlap some part of the drawer.

Another example is the recent evaluation (by Honda and Microvision[1]) of a wearable maintenance system that uses a non-tracked see-through heads-up display to present *in-situ* automotive maintenance informations to trained technicians. This system was demonstrably useful (resulting in a quoted 38% improvement over the non-wearable version) despite the fact that the graphics were not registered with the physical world. This system raises some interesting questions for the AR community: would an AR version of the system be even more effective? Would precise registration be necessary, or would a coarsely registered version using moderately accurate tracking be as effective (e.g., by allowing the current system's graphics to be generated from technician's approximate viewpoint, even if they aren't registered)? Would some tasks benefit

1 For more information, see `http://www.microvision.com/nomadexpert/field.html`

Proceedings of the Third IEEE and ACM International Symposium on Mixed and Augmented Reality (ISMAR 2004)
0-7695-2191-6/04 $20.00 © 2004 **IEEE**

greatly from precise registration, while others would not? When one considers practical issues of creating, deploying and maintaining such a system, such as cost and robustness, these questions become critical.

To continue with the above example, perhaps some repair shops would have good tracking, and others would not. Perhaps tracking quality would vary with the model of the car (e.g., new cars might have "embedded trackers", old ones would not), or the location in the shop or parking lot. Perhaps trackers break occasionally. For whatever reason, a commercially viable repair system such as this would need to function in a variety of situations, and ideally adapt automatically as the situation changes (e.g., as the technician walks around the shop). As an application designer, it is easy to imagine many different display modes for such a system, based on different amounts of registration error. What is hard is actually computing reasonable estimates of registration error for the different graphical objects in the system, and creating a graphical display system that uses these estimates to select the appropriate display modes. OSGAR is designed to address this problem.

3. Sources of uncertainty

For OSGAR, we consider the following classes of uncertainty [8]: tracking, calibration, and modeling.

Tracking. Tracking systems estimate, in real-time, the pose of a tracked object. There are literally hundreds of papers which describe different tracking methods based on a variety of sensing technologies (e.g., magnetic, ultrasonic, inertial, computer vision, etc.), as well as hybrid systems that combine more than one of these technologies. However, as noted by Welch and Foxlin, there is no "Silver Bullet" that is likely to provide perfect tracking [16]. Therefore, any tracking system should be assumed to return error-corrupted estimates of the true pose of the sensor. These errors can be modeled statistically. However, it is often very difficult to provide precise, high-order statistical descriptions of these errors, especially since many tracking systems are closed black-boxes, making it impossible to know what signal processing is carried on within them.

Therefore, we assume that the measurement from a tracker can be considered to be the mean of the distribution and the uncertainty can be represented by the covariance. Some tracking systems (such as the Intersense VisTracker and GPS receivers that support the NMEA GST message) provide covariance information directly. However, many tracking systems only provide performance specifications and the covariances must be approximated from these.[2]

2 For example, if the specifications consist of a hard bound on the errors, the standard deviation can be set to be a third or a quarter of this value.

Calibration. An AR system consists of a tracking system and a display system and the calibration of these systems and the relationship between them must be known. For example, in video-mixed AR systems the intrinsic parameters of the camera (such as its optical distortion) must be computed. Calibration parameters can be accurately computed off-line for a camera with a specific focus and zoom setting by looking at a static scene with a set of calibration patterns in it. However, there is no guarantee that these parameters are correct for a moving camera in a scene with different camera settings. The problems are exacerbated in see-through AR systems because current methods require the user to align objects on the display with those in the physical world (e.g., SPAAM [14] or the alignment framework described in [2]). Issues such as fatigue, the finite sampling space and user error can lead to inaccuracies. The errors can be calculated using perturbation methods.

Modeling. Models are approximations of the physical objects they are meant to represent. Models can be acquired in many ways, from a tape measure to 3D scanning laser range finders. However, measurements always contain errors. The environment can change in unmodeled ways. The GIS community has been cognizant of the effects of errors for a great deal of time [15] and the Geography Markup Language (GML) includes a schema for data quality which is expressed using means and covariances. Therefore we assume that the raw model constitutes a mean estimate and each vertex in the model contains errors.

Several issues should be noted. First, each of these error sources include *temporal* elements that OSGAR does not currently address. Latency throughout all components of the system, time synchronization across multiple devices, and the discrete update times of displays and the rendering subsystems create errors which can be considered to increase the error in the tracker [8]. Second, some of these errors are *view dependent* and some are *view independent*. The error in the model is not, for example, a function of the position and orientation of the user's head. However, the projection of the model onto the user's display is a function of the users view. As described below, this distinction is used to optimize the error propagation mechanism. Finally, we are not aware of any widely-used modeling format which includes information about the imprecision of the model, so we do not currently support models with errors on each vertex. However, it would be straightforward to modify the system to support such models if they existed.

4. Error representation and propagation

Uncertainty in OSGAR is represented by adding a covariance matrix to the transformation nodes in the scene graph. Any transformation matrix is considered to

be the mean of a probability distribution function (PDF) for that transformation, instead of just a single discrete transformation. If no uncertainty information is specified for a transformation, it is assumed to be exact. The mean of the PDF (i.e., the original transformation) is used for culling, rendering, and so on, as before.

No restrictions are placed on the form of the transformation matrices; the scene graph is assumed to be composed of arbitrary nodes with arbitrary affine transformations between them. Specifically, let $\mathbf{M}_i^j$ be the true relative transformation from node i to node j ,

$$\mathbf{M}_i^j = \begin{bmatrix} m_{11} & m_{12} & m_{13} & m_{14} \\ m_{21} & m_{22} & m_{23} & m_{24} \\ m_{31} & m_{32} & m_{33} & m_{34} \\ m_{41} & m_{42} & m_{43} & m_{44} \end{bmatrix}$$

Each element can take arbitrary values.[3] To parameterize such a general matrix, a number of authors have developed methods to decompose an arbitrary matrix into a set of primitive operations including rotation, translation and scale [12, 6]. However, these decompositions are constructed by applying potentially expensive nonlinear operations (such as single value decomposition). Because the graph can be extremely large, a significant number of these decomposition operations might be performed leading to significant computational costs. Therefore, to simplify the implementation, all errors are expressed directly in terms of the elements of the transformation matrix. In other words, the uncertainty is a 16-dimensional state which corresponds to each element in the transformation matrix. This is a straightforward generalization of the approach of Bar-Itzhack for direction cosine matrices [3].

$\mathbf{M}_r^n$ is the cumulative transformation matrix from the root node $\mathbf{R}$ to an arbitrary node $\mathbf{N}$. This transformation is given by:

$$\mathbf{M}_r^n = \prod_{\forall i \in P} \mathbf{M}_{i-1}^i, \tag{1}$$

where p is the path from the root node $\mathbf{R}$ to an arbitrary node $\mathbf{N}$ and i are nodes in this path.

However, the system does not have access to these true values. Rather, it only has access to the estimated relative transformation $\hat{\mathbf{M}}_i^j$. The difference between the two is due to the sources of uncertainty outlined in Section 3. As a result, the cumulative transformation calculated in the graph is:

$$\hat{\mathbf{M}}_r^n = \prod_{\forall i \in P} \hat{\mathbf{M}}_{i-1}^i \tag{2}$$

Therefore, the problem is to estimate the statistics of $\hat{\mathbf{M}}_r^n$ given that error can be introduced at any transformation in the tree.

3 It is not even possible to assume that $m_{44} = 1$.

The error introduced at a node is assumed to be an additive matrix,

$$\hat{\mathbf{M}}_i^j = \mathbf{M}_i^j + \delta\mathbf{M}_i^j. \tag{3}$$

Therefore, the error propagation equation is

$$\begin{aligned} \mathbf{M}_r^i + \delta\mathbf{M}_r^i &= \left(\mathbf{M}_{i-1}^i + \delta\mathbf{M}_{i-1}^i\right)\left(\mathbf{M}_r^{i-1} + \delta\mathbf{M}_r^{i-1}\right) \\ &= \mathbf{M}_{i-1}^i\mathbf{M}_r^{i-1} + \mathbf{M}_{i-1}^i\delta\mathbf{M}_r^{i-1} \\ &\quad + \delta\mathbf{M}_{i-1}^i\mathbf{M}_r^{i-1} + \delta\mathbf{M}_{i-1}^i\delta\mathbf{M}_r^{i-1} \end{aligned} \tag{4}$$

Assuming that the error introduced at a node is independent of the error introduced at preceding nodes, the expected value of the last term will always evaluate to $\mathbf{0}$ and thus can be neglected. Therefore, the equation which propagates the error down the scene graph is as follows:

$$\begin{aligned} \hat{\mathbf{M}}_r^i &= \hat{\mathbf{M}}_{i-1}^i\hat{\mathbf{M}}_r^{i-1} \\ \delta\mathbf{M}_r^i &= \mathbf{M}_{i-1}^i\delta\mathbf{M}_r^{i-1} + \delta\mathbf{M}_{i-1}^i\mathbf{M}_r^{i-1} \end{aligned} \tag{5}$$

However, this representation has two main difficulties:

- It is more computationally expensive. If one assumed, for example, that the matrix only encoded translation rotation and scale then only 9 or 10 parameters would be required. However, this is at the cost of introducing complicated nonlinear transformations at each node to recover the parameters (and their uncertainties) after a transformation is applied.
- The representation does not capture the nonlinear constraints which exist between matrix elements. For example, large orientation errors are not simply additive. These could be partially overcome by using more sophisticated models. For example, the error could be treated as being multiplicative and of the form $\mathbf{I} + \delta\mathbf{M}_{i-1}^i$ where $\mathbf{I}$ is the identity matrix. A recursive relationship exists in this case.[4] However, any representation is always an approximation and, for this paper, we chose the simplest usable approximation.

Despite these limitations, we believe this representation is appropriate for the needs of OSGAR:

- The transformation operations on each node are simple. The transformation consists of a single matrix multiplication which only involves basic arithmetic operations.
- The complexity of specifying nonlinearities (e.g., tracker errors) are only introduced at the nodes where the errors occur.

4 The mean term is the same but the error propagation term becomes $\delta\mathbf{M}_r^i = \delta\mathbf{M}_r^{i-1} + \left(\mathbf{M}_r^{i-1}\right)^{-1}\delta\mathbf{M}_{i-1}^i\mathbf{M}_r^{i-1}$.

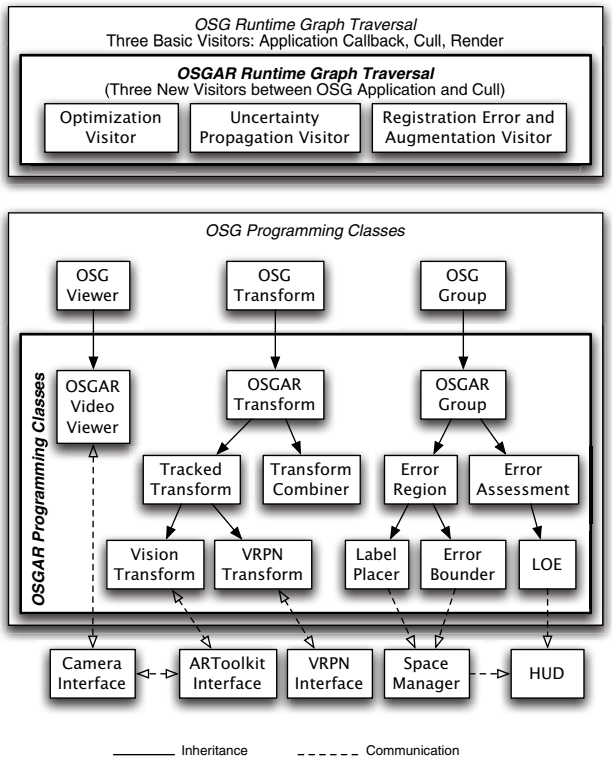


Figure 2. The major components of OSGAR.

- If nodes do not introduce their own sources of error, the first term in the error propagation is not needed, reducing complexity even further.
- Any type of matrix operation can be supported. Therefore, not just errors in the location of objects, but also their sizes, the position and projection properties of the viewer, etc., can be handled in a uniform way.

5. Architecture

OSGAR is an extension to OSG, a full featured 3D scene graph library. The key difference between OSG and OSGAR is that transformations is OSGAR can have uncertainty associated with them; most of the functionality of OSGAR builds on this conceptually simple change. OSGAR also provides programmers with access to basic AR technologies (e.g., live video-in-the-background, real-time tracking devices, marker-based tracking in the video stream).

The components of OSGAR are shown in Figure 2. The core components are the three runtime *Visitors* that propagate uncertainty and registration error estimates through the scene graph (see Section 5.1), plus the subclass hierarchies under *OSGAR Transform* (see Section 5.2) and *OSGAR Group* (see Section 5.3). The Transform classes are used to define nodes with uncertain transformations in the scene graph. The Group classes control how the scene graph reacts to registration errors (see Section 5.4 for examples of specific Group classes).

The remaining classes provide facilities needed by AR applications, and are described briefly in Section 5.5. The *Video Viewer* is a specialized 3D viewer that takes video from a camera and inserts it in the background of the scene. The *Camera*, *ARToolkit*[5] and *VRPN*[6] interfaces provide access to basic AR technology. The *Space Manager* and *Heads-up Display (HUD)* are utility classes designed to manage space on the display, and layout augmentations in a 2D HUD.

One deviation OSGAR makes from OSG is that we require the camera location to be specified as a node in the scene graph. Setting the camera location to a node in the graph is a straightforward way to generate the uncertainty estimate of the model-view matrix defining the camera's pose in the world.

5.1. Scene graph traversal

OSG is based on the notion of using a set of specialized *Visitors* to traverse the scene graph each frame. OSGAR adds three Visitors (i.e., three passes through a subset of the scene graph) to the per-frame update loop, after application callbacks are run, but before culling and rendering. The three Visitors together implement the OSGAR runtime functionality, as follows:

1. Optimization. This Visitor initializes the nodes in the graph for error propagation, determines the subset of the scene graph that needs to be examined by the other two Visitors, and sets flags on the nodes to control those traversals.

The Visitor uses a standard recursive, bottom-up damage propagation technique to mark the nodes:

1. When a node is reached, its "requires error processing" flag is set to false.
2. The children are visited.
3. If the node, or any child of the node, needs to be processed, then this node is also marked as requiring processing.

2. View-Independent Uncertainty Propagation. This Visitor propagates uncertainty as it traverses the graph from the root of the scene through all of the nodes marked by the optimization visitor, using the algorithm described in Section 4 (see Section 5.2). At each step, the accumulated estimate is combined with the uncertainty at the current

5 Available from `http://www.hitl.washington.edu/artoolkit/`

6 Available from `http://www.cs.unc.edu/Research/vrpn/`

node using Equation 5, giving an estimate of the uncertainty of that node expressed in the local coordinates of that node. This computation is done for both OSG and OSGAR transformation objects, with the OSG transformations assumed to be precisely specified with no error.

Each time one of the OSGAR Group nodes is visited, the accumulated uncertainty is stored at that node, for use in the next traversal. (Note that since OSG supports directed acyclic graphs, a single node could lie on multiple paths from the root. The propagated uncertainty is stored for each path to the root of the graph, and tagged with a path identifier so it can be recovered when needed.) Any Transformation Combiner nodes encountered during traversal are also handled by this visitor (see Section 5.2.2).

3. Registration Error Visitor. Before starting this Visitor, the uncertainty of the camera location (obtained from the node representing the camera in the scene graph) is combined with the projection matrix (which may also include uncertainty). As the Visitor traverses the graph, the camera uncertainty is combined with the uncertainty at each OSGAR Group node, creating a view-dependent estimate of uncertainty. This uncertainty estimate is used to compute screen-space estimates of registration error needed by the OSGAR Group nodes (see Section 5.3).

For efficiency, this Visitor collects the vertices of all geometry in the subtree under each Group node (whenever any part of the subtree is damaged) and stores a 3D convex hull of those points in the Group itself. The points in the 3D hull are used by the Group nodes when computing the view-dependent registration error estimates.

5.2. Transform nodes

The OSGAR Transform base class extends OSG Transform by using its existing 4×4 transformation matrix as the mean of the distribution of the transformation, and associating uncertainty (in the form of a 16×16 covariance matrix) with this 4×4 transformation. A collection of utility methods are provided for setting and retrieving the uncertainty information in various forms. The covariance matrix can take on the special values *perfect* (to indicate that there is no uncertainty associated with this transformation) and *infinite* (to indicate that the value of the transformation is unknown, and the link should not be followed). The latter value could be used, for example, when a tracker is not reporting (e.g., the user is out of range, or a fiducial marker is not currently seen).

A programmer can use OSGAR Transform directly, to create a node in the graph with some fixed uncertainty. For example, if the location of an object in the environment was measured relatively carefully, a position of +/- a few millimeters and an orientation of +/- a few degrees could be specified via method calls to *setPositionCovariance(dx, dy, dz)* and *setOrientationCovariance(dh, dp, dr)* (where *dh*, *dp* and *dy* are the covariances for heading, pitch and roll respectively).

Several subclasses of the Transform node exist.

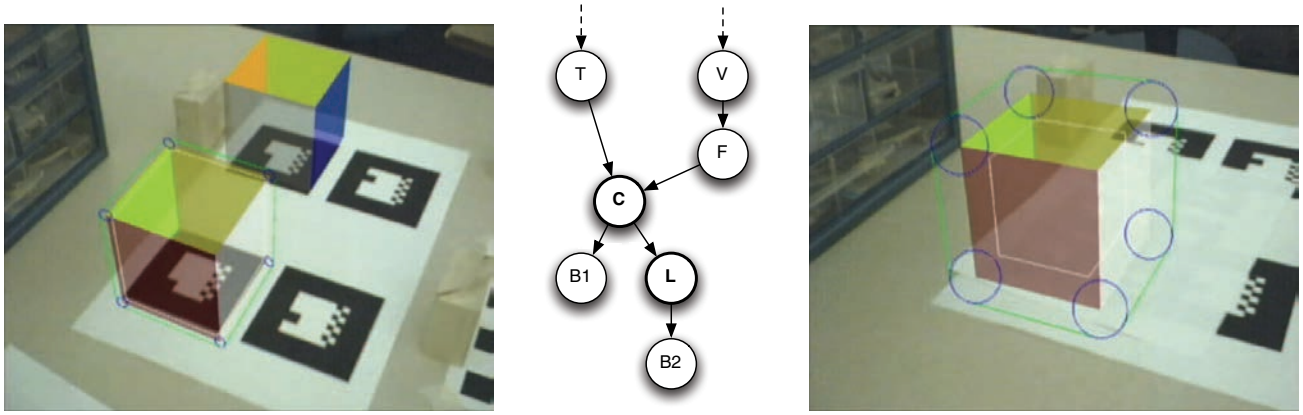
5.2.1. Tracked transform nodes One common source of uncertain transformations is external tracking systems. OSGAR TrackedTransform is a subclass of OSGAR Transform whose values are updated automatically from some tracking system. This class has methods (implemented by each subclass) to determine if the tracker is currently reporting or not. The value of the transformations on each TrackedTransform is updated by the *tracker handler* (see section 5.5). The current implementation of OSGAR supports two subclasses of TrackedTransform:

- *vrpnTransform* creates a VRPN tracker client to connect to a sensor of a local or remote VRPN tracker server. The application developer specifies the tracker name and network address, and the specific sensor to get transformations from.
- *visionTransform* implements a vision-based tracker, currently using the ARToolkit. The application developer specifies the marker that this Transformation should be attached to. The transformations received are the position of the fiducial relative to the camera, so the visionTransform is typically attached to the scene graph node representing the location of the camera.

Currently, neither VRPN nor ARToolkit provide uncertainty estimates for their trackers, so we set the uncertainty manually on these nodes, typically based on the manufacturers specifications and our experience. We are in the process of adding uncertainty support to VRPN, and enhancing specific VRPN servers to support it (e.g., many GPS units provide an estimate of their current accuracy, and one major tracking company is providing us with an SDK to retrieve covariance information from their trackers). Similarly, we are creating an accuracy estimator for the ARToolkit.

5.2.2. Transformation combiner nodes The *Transform-Combiner* class is designed for situations where multiple pose estimates are available for an object. The programmer can set up a Combiner to automatically combine the pose estimates, can leverage application or tracker specific knowledge to improve the accuracy of the fusion, or can simply choose between the poses.

If multiple poses are available for a node in a scene graph, there will be a path from the root to the node for each pose. Normally, if a node is linked to a scene graph via more than one path, the scene graph semantics dictate



(a) (b) (c)

Figure 3. The TransformCombiner *C* chooses which path to cube *B1* (the near cube) to use for each frame, based on the error propagated from *T* (a receiver for a fixed tracker) and *F* (a fiducial marker, measured relative to the video camera *V*). In (a) the fiducial markers are visible, and has much lower estimated error than the fixed tracker, so the branch through *F* is used. The second cube is attached to *C* via an LOE *L* that only displays it when the error is small. In (b) the fiducial is not tracked, so the fixed tracker is used (and the second cube is hidden by the LOE).

that it is rendered once for each path, usually at a different location in the 3D world.

The TransformCombiner has different semantics. A programmer-supplied callback function determines which single incoming path to use for the subtree under the Combiner. The function is given the pose estimates for all of the paths leading into the Combiner, and returns a new estimate along with an indication of which incoming path to use for all subsequent passes. The effect is equivalent to the children of the Combiner being attached to that path from the root, and the other paths terminating at the Combiner.

The most obvious scenario is when an object is tracked by multiple sensors, and the Combiner should fuse the poses using an approach such as the PDF intersection method proposed by Hoff [7]. However, there are other more mundane scenarios in which the combiner turns out to be a very powerful way of structuring an AR scene graph.

For example, consider a user and object that are tracked by some reasonably accurate tracker (such as an Intersense IS600 or an RTK GPS system, both of which give 1-2cm positional accuracy), and the object can also be tracked using the camera on the user's video-mixed AR display. If the system wants to add augmentations to the object, sensor fusion is not particularly useful — the absolute accuracy of a system such as the ARToolkit is not particularly good, but the registration obtained when using it is quite good. (While both the translation error along the direction of projection and the rotation error are large, the translation error is small perpendicular to the direction of projection.) In this situation, using the vision tracker when it reports, and falling back to the less accurate tracker otherwise, is probably the right thing to do, as illustrated in Figure 3.

Perhaps more interestingly, suppose an object is tracked intermittently (such as by a vision tracker), but the object always remains within a certain area, such as on a desk that it is tethered to by a cable. In most AR prototype systems, when the object is not tracked, its last known position is used. OSGAR supports more systematic solutions to this problem.

One simple solution would be to use a static Transform to specify a second pose for the object, and use a TransformCombiner to merge it with the tracked pose. The Transform would be given a mean in the middle of the desk and a large uncertainty, to capture the full range of the object's possible location.[7] The Combiner would choose this static transformation when the object is not tracked, but use the tracked path when possible. In this case, the Combiner would propagate the fixed transformation with a large error estimate or the tracked transformation with a very small error estimate. If the programmer simply

7 More complex solutions could also be implemented that take into account the time since the object was last tracked, or leverage other application-specific semantics.

uses the provided transformation (i.e., the mean of the distribution) to render the augmentation, the resulting display would be nonsensical (the object would jump between the tracked location and the arbitrary "middle of the desk" location specified by the fixed transformation).

However, if the programmer makes use of the error estimation nodes discussed below, the augmentation itself can change automatically when the magnitude of the error changes, and do so in a way that adapts to other aspects of the viewing situation. If the fixed area is, say, the top of a desk and the user is very close to the desk, the registration error (the projection on the 2D display) will be huge, requiring alternative display methods (such as using a 2D inset window containing descriptive text). However, if the user is far from the desk (say, on the other side of a large lab), the magnitude of the registration error *on the 2D display* may be quite small, allowing a 3D augmentation to be used. By specifying a range of augmentations to use, based on registration error on the 2D display, the system can adapt automatically to these very different situations. This example illustrates the power of OSGAR, and the approach programmers should take when building applications using it.

Implementation. The Optimization Visitor counts the number of paths that enter any Combiner node, and saves the counter in the Combiner. Then, during the view-independent uncertainty propagation traversal,[8] the Combiner collects the error estimates for every path into it. When this visitor enters the Combiner via the final path, it activates the user specified callback function to determine the final pose estimate and the path to use for the subtree during the remaining traversals. Subsequent phases (registration error computation, culling, rendering) only traverse the subtree "under" the Combiner when they arrive along this single path.

We are designing a range of sample callback functions for common Combiner functions. Currently, we have implemented a *SmallestCovarianceCombiner* that always chooses the smallest covariance, and uses it and its path as the values for the subtree.

5.3. Basic group nodes

The OSGAR Group class is abstract, and provides accessor methods to retrieve the view-independent uncertainty estimates that are computed by the propagation Visitor. The *Assessment* and *Region* subclasses provide access to two different representations of the registration error estimate of the objects in the tree under them. Recall from Section 5.1 that the registration error Visitor computes the 3D convex hull of the geometry in the subtree underneath all OSGAR Group nodes. The OSGAR Group nodes use the view-dependent error estimate to compute a 2D view-dependent convex hull of the points in this 3D hull. The points in the 2D hull are used to compute the registration error estimates.

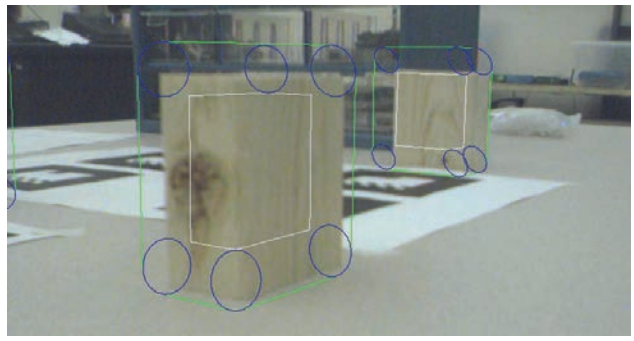


Figure 4. The registration error of cubes in the world. The blue ellipses represent the registration error around the vertices of the cubes. The green "outer" region is the convex hull representing the area the entire box will fall within, and the white "inner" region is the convex hull representing the area that some part of the box should occupy.

Region classes. Regions provide the programmer with a collection of closed regions (illustrated in Figure 4): an error ellipse for each vertex in the 2D hull, an outer region (the convex hull of all points in the error ellipses) representing the region that the object *might* intersect, and an inner region representing the region the object *should* intersect (see [11] for an explanation of how these regions are computed).

Assessment classes. Assessments provide the programmer with a single floating point value representing an assessment of the magnitude of the registration error. The assessment is computed using two user-defined methods: *metric* is run on each of the vertices in the 2D hull, giving a floating point value for each. An *aggregator* is run on the set of vertices and floating point values, and returns the single value for the object. There are many possible metrics that could be used by an Assessment object, such as the maximum of the main axis of the ellipses, the area of the ellipse, etc. As an aggregator function, one can consider the closest vertex, the average of all vertices, etc.

8 This decision should be made in the registration error visitor, using view dependent estimates, rather than in the view-independent uncertainty propagation visitor, using the view independent estimates. This will be changed in the near future when we re-implement the visitors to solve a set of related problems, but the current implementation is sufficient for illustrating the desired functionality.

5.4. Specialized group subclasses

We have implemented one subclass of Assessment (*LOE*) and two subclasses of Region (*Bounding Regions*, *Label Placer*) as examples of how to extend the basic Group classes.

Bounding Regions. Used to display graphical representations of the 2D convex hull: the vertex ellipses, the inner region, the outer region, or any combination of them. It can be used for prototyping, debugging or to create a simple highlight of the region an object is expected to occupy. Figure 4 shows two cubes with all regions displayed.

Level of Error (LOE) Switches. The LOE (originally discussed in [10]) automatically chooses between different subgraphs within the scene graph, allowing the application developer to specify different augmentations corresponding to the same physical object. At any time, one of the specified subgraphs is enabled (visible) and the rest are disabled (invisible). The location of the LOE in the scene graph is used to determine the registration error estimate, but the subgraphs do not need to be in the same part of the graph. In our current implementation, the LOE chooses between its children and children attached to the HUD (discussed in Section 5.5), but it could easily be extended to include subgraphs elsewhere in the scene graph. During each traversal, one augmentation will be chosen based on to the metric computed from the registration error estimate at the LOE node.

Label Placer. The Label Placer computes where to position the labels for a given object based on the computed inner and outer regions of the model. A programmer can choose to keep the label overlapping the object (by keeping it inside the inner region) or guarantee that the label will never block the object (by keeping it outside the outer region). The Label Placer computes where to place the labels each frame to enforce one of these constraints. The Label Placer uses a callback that specifies how to position the labels, for which we have implemented three simple examples. The first implementation positions the label where there is the most space available, computed from the sides of the object to the limits of the screen. The second implementation always tries to position the label in this order: right, top, bottom and then left. If the label does not fit on the right side, then it tries to position it at the top, and so on. A third implementation favors edges closer to the screen sides, to keep the middle clear. Labels that are not positioned by the Label Placer are marked as not *anchored* and passed to the *Space Manager* (described in section 5.5) to be handled there.

5.5. Additional AR components

As mentioned above, OSGAR provides a collection of facilities necessary for AR application development.

Video. Our current focus is on video-mixed AR experiences, so the OSGAR *Video Viewer* class allows a video stream to be texture mapped onto the background of the window. The *Camera Interface* also feeds video to the ARToolkit for fiducial recognition.

Trackers. OSGAR supports both the tracking of fiducial markers and a wide variety of spatial trackers via the VRPN tracker package. Both are handled internally by a centralized *tracker handler* that performs the marker detection on each new video image, polls VRPN once per frame, and updates the values of the associated Tracker Transformations in the scene graph when necessary. Trackers will eventually provide uncertainty estimates with their reports, although we have not finished extending VRPN and the ARToolkit to do this.

HUD. The HUD class is used for displaying 2D augmentations. It is implemented as an orthographic projection attached to the camera position. Any kind of OSG subgraphs can be attached to the HUD.

Space Manager. Currently a stand-in for a more powerful space manager, such as that proposed by Bell and Feiner [4]. The class collects the regions created by the Error Region classes into a set of *Hull* objects, and uses the HUD to display those that should be visible. Hull objects store all the vertex error ellipses, the inner and outer hulls, the path on the scene graph, and the object's name. The Space Manager is also responsible for positioning labels that were not positioned explicitly by the Label Placers.

6. Discussion

Our method deals with the effects of dynamically changing static uncertainty on spatial registration error; OSGAR does not yet take into account temporal aspects of such errors, nor does it try to take into account other influencing factors, such as illumination.

We designed OSGAR to use several distinct Visitors because we hope to eventually decouple the registration error computation from the display loop. Even though the system has proven to be sufficiently fast (we exceed 60 frames per second on a dual 2GHz Pentium4 Xeon with an NVIDIA Quadro graphics card in our example and test programs), we are concerned that a toolkit designed to reduce the impact of registration error should not increase the latency of the system (and thus increase registration errors). Fortunately, the metrics we compute do not generally need to be synchronized with the display loop; if a Level-of-Error object or a Label Placement algorithm works with data that is a few frames (i.e., a fraction of a

second) old, the result should be almost imperceptible to the user. Even the computation of the 2D convex hulls does not need to be done synchronously, as long as the resulting graphics are translated on the screen with the objects. If an object is changing quickly, the computed regions may be slightly wrong, but most of the uses we propose for such regions would not be adversely affected by such latency.

7. Conclusions

There has been increasing interest in the AR community on how to engineer solutions that support the real deployment of AR applications [9, 5]. We believe that the ability to adapt to error estimates will form the foundation of AR systems that are not tied to specific tracking and sensing hardware, and are thus more robust and deployable in a wide variety of situations.

In this paper, we have introduced an architecture that integrates the uncertainty associated with the physical world into 3D computer graphics. We use this information to automatically and efficiently estimate the registration error associated with each object in an AR system in real time, and present a collection of sample programming structures that demonstrate how these estimates can be used to improve the quality of the information being conveyed by the system.

We believe OSGAR represents an important step toward the creation of real, robust AR systems in complex, mobile environments. By allowing programmers to deal with tracking technology (and other uncertainty) in a methodical and structured way, they can focus on what the application should do in different conditions, rather than tightly coupling the system to a particular collection of devices.

8. Acknowledgments

The authors would like to thank Brendan Hannigan for many of the low-level AR libraries, Mike Ellison for the VidCapture wrappers for DirectShow (which are the basis for our video library), researchers at UNC for VRPN, and the many authors of the ARToolkit (which we use for fiducial marker tracking). This work was supported by the Office of Naval Research under Contract N000140010361.

References

[1] R. Azuma and G. Bishop. Improving static and dynamic registration in an optical see-through HMD. In *Computer Graphics (Proc. ACM SIGGRAPH '94)*, Annual Conference Series, pages 197–204, Aug. 1994.

[2] Y. Baillot, S. Julier, D. Brown, and M. Livingston. A tracker alignment framework for augmented reality. In *Proceedings of the International Symposium on Mixed and Augmented Reality (ISMAR '03)*, pages 142–150, Tokyo, Japan, 7–10 October 2003.

[3] I. Y. Bar-Itzhack and J. Reiner. Recursive attitude determination from vector observations: Dcm identification. *Journal of Guidance, Control and Dynamics*, 7(1):51–56, January – February 1984.

[4] B. Bell, S. Feiner, and T. Hollerer. View management for virtual and augmented reality. In *ACM Symposium on User Interface Software and Technology (UIST '01)*, pages 101–110, Nov 11–14 2001.

[5] E. M. Coelho and B. MacIntyre. High-level tracker abstractions for augmented reality system design. In *The International Workshop on Software Technology for AR Systems (STARS '03)*, October 2003.

[6] R. Hartley and A. Zisserman. *Multiple View Geometry*. Cambridge University Press, 2nd edition, 2003.

[7] W. Hoff. Fusion of data from head-mounted and fixed sensors. In *Proceedings of the First International Workshop on Augmented Reality (IWAR '98)*, pages 167–182, 1998.

[8] R. L. Holloway. Registration Error Analysis for Augmented Reality. *Presence: Teleoperators and Virtual Environments*, 6(4):413–432, August 1997.

[9] G. Klinker, T. Reicher, and B. Brugge. Distributed user tracking concepts for augmented reality applications. In *International Symposium on Augmented Reality (ISAR '00)*, pages 37–46, Oct 5–6 2000.

[10] B. MacIntyre and E. M. Coelho. Adapting to dynamic registration errors using level of error (LOE) filtering. In *International Symposium on Augmented Reality (ISAR '00)*, pages 85–88, Oct 5–6 2000.

[11] B. MacIntyre, E. M. Coelho, and S. Julier. Estimating and adapting to registration errors in augmented reality systems. In *IEEE Virtual Reality (VR '02)*, pages 73–80, March 2002.

[12] K. Shoemake and T. Duff. Matrix animation and polar decomposition. In *Proceedings of Graphics Interface (GI '92)*, pages 258–264, Vancouver, BC, Canada, May 11–15 1992.

[13] A. State, M. A. Livingston, G. Hirota, W. F. Garrett, M. C. Whitton, H. Fuchs, and E. D. Pisano. Technologies for augmented-reality systems: realizing ultrasound-guided needle biopsies. In *Proceedings of SIGGRAPH 96*, pages 439–446, New Orleans, LA, August 4–9 1996.

[14] M. Tuceryan and N. Navab. Single point active alignment method (SPAAM) for optical see-through HMD calibration for AR. In *International Symposium on Augmented Reality (ISAR '00)*, pages 149–158, Oct 5–6 2000.

[15] H. Veregrin. Error modeling for the map overlay operation. In M. Goodchild and S. Gopal, editors, *Accuracy of Spatial Databases*. Taylor and Francis, London, UK, 1994.

[16] G. Welch and E. Foxlin. Motion tracking: No silver bullet, but a respectable arsenal. *IEEE Computer Graphics and Applications*, 22(6):24–38, 2002.