UC San Diego
Technical Reports

Title

A Systems Architecture for Ubiquitous Video

Permalink
https://escholarship.org/uc/item/8gc255s1

Authors
McCurdy, Neil J
Griswold, William G

Publication Date
2005-02-04

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

A Systems Architecture for Ubiquitous Video

Neil J. McCurdy
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
nemccurd@cs.ucsd.edu

William G. Griswold
Computer Science and Engineering
University of California, San Diego
La Jolla, CA 92093-0114
wgg@cs.ucsd.edu

Abstract

Realityflythrough is a telepresence/tele-reality system that works in the dynamic, uncalibrated environments typically associated with ubiquitous computing. By opportunistically harnessing networked mobile video cameras, it allows a user to remotely and immersively explore a physical space. This paper describes the architecture of the system, motivated by the real-time and dynamic requirements imposed by one application domain: SWAT team command and control support.

RealityFlythrough is analogous to an operating system in that it provides abstractions to the user that hide inherent limitations in the underlying system. Just as an operating system provides the illusion of infinite processors and infinite memory, RealityFlythrough provides the illusion of complete live camera coverage in a physical environment.

1 Introduction

Ubiquitous computing is often described as computers fading into the woodwork [3]. Ubiquitous video, then, is cameras fading into the woodwork, and is captured by the expression, "the walls have eyes." Ubiquitous video is characterized by wireless networked video cameras located in every conceivable situation. The data is transmitted either to a central server or simply into the ether for all to view [2]. While many believe that such an environment is inevitable, we do not have to wait for the future to take advantage of ubiquitous video. There are a number of situations that could benefit from having live, situated access to ubiquitous video streams using today's technology.

For example, police Special Weapons and Tactics (SWAT) teams [6] are routinely involved in high risk tactical situations in which the Incident Command Post (command and control) is situated some distance from the incident site. It is the responsibility of the command post, and specifically the team commander to direct the field operations, but this activity is often done "blind", without the aid of visuals from the scene. The commander forms an internal spatial model of the scene generated from either prior knowledge, maps, or reports from the officers in the field, and must update and reference this model throughout the event. Commands must be issued to field officers from their point of view, further straining the commander's conceptual model [6].

Introducing video feeds to the team commander's arsenal would be of obvious benefit. A naive solution would equip each field officer with a head or shoulder mounted camera and have the video streams displayed on an array of monitors similar to those used in many building security systems today. An ideal solution would present the team commander with infinite cameras allowing the commander to "fly" naturally around the scene viewing the operations from any desired vantage point. A more practical solution provides the illusion of the ideal system while operating under the constraints imposed by the real environment, including the constraint that the resulting displays should not be misleading.

We have created RealityFlythrough [10] [11], a system that uses video feeds obtained from mobile ubiquitous cameras to present the illusion of an environment that has infinite camera coverage. The use of illusion in RealityFlythrough is analogous to the illusions (abstractions) that an operating system provides to a programmer. Programming a raw computer is challenging because of its finite, yet untamed, resources. An operating system provides convenient programmability through the abstraction of an unlimited number of sequential processors and unlimited shared storage. Ubiquitous video is analogously limited (few cameras) and untamed (imprecise position and orientation). RealityFlythrough, then, provides abstractions for the sensible viewing of ubiquitous video streams, thus easing the task of making inferences from multiple separate video streams. Stitching the multiple video streams together into a single scene is a straightforwardly sensible abstraction of numerous video streams. With such an abstraction, the user need only understand one integrated scene, as in a video game, rather than multiple feeds, as in a building security system. However, limited resources as well as the untamed elements of ubiquitous video make such an abstraction non-trivial to construct.

The key *limitation* of ubiquitous video is the incom-

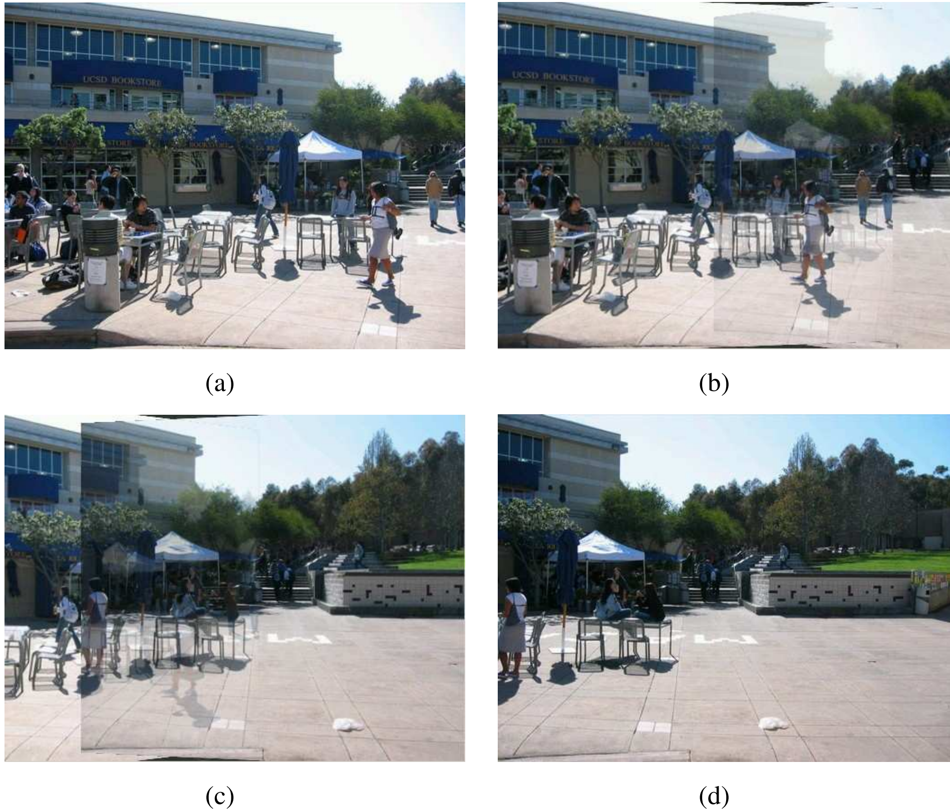

Figure 1: A transition from image (a) to image (d). Images (b) and (c) show the transition in progress as image (a) moves off the screen to the left and image (d) moves in from the right. This transition corresponds to a rotation to the right.

plete coverage of the live video streams–every square foot of a space cannot be viewed from every angle with a live video stream at any chosen moment. For two cameras pointing in two rather different directions, when the user switches from viewing one camera to another, it is often not obvious how the subject matter in the two views relate to each other, nor is it obvious what is in the intervening space between the two cameras. To address this limitation, RealityFlythrough fills the intervening space between two cameras with older imagery (captured from the live camera feeds), and provides segues (i.e., transitions) between the two live cameras that sequences and blends the imagery in a way that provides the sensation of a human performing a walking camera pan.

The key *untamed element* of ubiquitous video is the imprecision of both the location and orientation sensed for a camera (due to both sensor latency and sensor inaccuracy). Such imprecision gives misleading cues to the user about how the subject matter seen in one camera relates to the subject matter in another. For example, the images might appear farther apart than they really are. Under certain assumptions, offline vision techniques could perform seamless stitching [13]. To achieve real-time flythrough, this problem is instead handled by stitching together the live and still imagery in an abstraction that shows the misregistration in overlapping images (with an alpha blend), rather than hiding it through blending or clipping. Although this sacrifices aesthetics, it increases sensibility through full disclosure. For this technique to work, images must overlap. This property is sought by the mechanism that captures the older still images for filling.

As with operating system abstractions, the illusions are imperfect and exact a price. The filled images are still rather than moving, and are no more up to date than the last time a camera panned over a location. Since the system cannot anticipate the future, the still imagery must be captured for every conceivable location at all times, increasing memory requirements. The stitching is also not seamless. Moreover, it should not be. A SWAT commander, for example, must have awareness of the physical limitations and the staleness of any data. Consequently, static images display an age bar and our stitchings intentionally show imperfections. And in the spirit of an operating system performance monitor, additional views are provided to unmask the abstraction and reveal the underlying system structure. For example, a

birdseye view abstraction shows the positions and orientations of the live cameras, unmasking the illusion of infinite camera coverage. An additional feature of the abstraction is the ability to directly select a given camera.

The contributions of this paper are the RealityFlythrough architecture, and its evaluation along three dimensions: (1) its support for the desired abstractions for ubiquitous video, (2) scalability, and (3) its robustness to changing user requirements that is the measure of every good architecture. The architecture has three unique qualities. First, it uniformly represents all image sources and outputs as *Cameras*, supporting a rich yet simple set of operations over those elements in achieving the desired abstractions. Second, it employs a separate *Transition Planner* to translate the user's navigation commands into a sensible sequence of camera transitions and accompanying image blends. Third, it aggressively employs the Model-View-Controller design pattern to separate the world state of all the cameras from the user controls, transition planner, windowing toolkit, and even the underlying graphics library. Our experiments show good support for the desired abstractions, as well as excellent scalability in the number of live video sources and *Cameras*. Support for evolution is explored through a series of changes to the application.

The paper is organized as follows. Section 2 will describe the user experience, and Section 3 will compare our system to related work. Section 4 will outline the requirements of the system. We will present a high level architectural overview of the system in Section 5, and then drill into the RealityFlythrough engine in Section 6 to reveal how the illusion of infinite cameras is achieved. Sections 7.1 and 7.2 will evaluate the architecture's support of the system requirements, and Section 7.3 will evaluate the architecture's tolerance to change and support for future enhancements. Section 8 concludes the paper.

2 User Experience

A large element of the user experience in RealityFlythrough is dynamic and does not translate well to the written word or still photographs. We encourage the reader to watch a short video [11] that presents an earlier version of RealityFlythrough, but we do our best to convey the subtlety of the experience in this section. When observing the images in Fig. 1, keep in mind that the transformation between the images is occurring within about one second, and the two transitional frames represent only about 1/10th of the transition sequence.

The user's display is typically filled with either an image or a video stream taken directly from a camera. When a new vantage point is desired, a short transition sequence is displayed that helps the user correlate objects in the source image stream with objects in the destination image stream. These transitions are shown in a first person view and provide the user with the sensation that she is walking from one location to another. The illusion is imperfect, but the result is sensible and natural enough that it provides the necessary contextual information without requiring much conscious thought from the user.

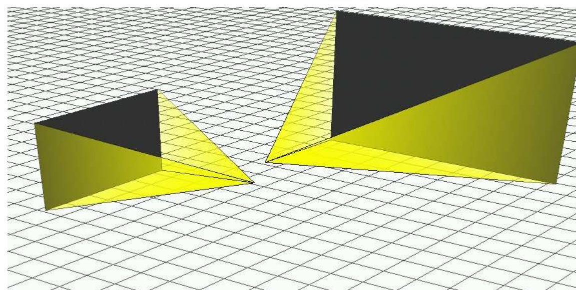

Figure 2: An illustration of how the virtual cameras project their images onto a wall.

RealityFlythrough works by situating 2d images in 3d space. Because the position and orientation of every camera is known, a representation of the camera can be placed at the corresponding position and orientation in virtual space. The camera's image is then projected onto a virtual wall (see Fig. 2). When the user is looking at the image of a particular camera, the user's position and direction of view in virtual space is identical to the position and direction of the camera. As a result, the entire screen is filled with the image. Referring to Fig. 1, a *transition* between camera A (image (a) in the figure) and camera B (image (d) in the figure) is achieved by smoothly moving the user's position and view from camera A to camera B while still projecting their images in perspective onto the corresponding virtual walls. By using OpenGL's standard perspective projection matrix to render the images during the transition, the rendered view situates the images with respect to each other and the viewer's position in the environment. By the end of the transition, the user's position and direction of view are the same as camera B's, and camera B's image fills the screen.

It may be easier to understand how RealityFlythrough works by envisioning the following concrete example. Imagine standing in an empty room that has a different photograph projected onto each of its walls. Each image covers an entire wall. The four photographs are of a 360 degree landscape with one photo taken every 90 degrees. Position yourself in the center of the room looking squarely at one of the walls. As you slowly rotate to the left your gaze will shift from one wall to the other. The first image will appear to slide off to your right, and the

second image will move in from the left. Distortions and object misalignment will occur at the seam between the photos, but it will be clear that a rotation to the left occurred, and the images will be similar enough that sense can be made of the transition. RealityFlythrough operates in a much more forgiving environment: the virtual walls are not necessarily at right angles, and they do not all have to be the same distance away from the viewer.

3 Related Work

There have been several approaches to telepresence with each operating under a different set of assumptions. Telepresence [8], tele-existence [14], tele-reality [13] [7], virtual reality and tele-immersion [9] are all terms that describe similar concepts but have nuanced differences in meaning. Telepresence and tele-existence both generally describe a remote existence facilitated by some form of robotic device or vehicle. There is typically only one such device per user. Tele-reality constructs a model by analyzing the images acquired from multiple cameras, and attempts to synthesize photo-realistic novel views from locations that are not covered by those cameras. Virtual Reality is a term used to describe interaction with virtual objects. First-person-shooter games represent the most common form of virtual reality. Tele-immersion describes the ideal virtual reality experience; in its current form users are immersed in a CAVE with head and hand tracking devices.

RealityFlythrough contains elements of both tele-reality and telepresence. It is like telepresence in that the primary view is through a real video camera, and it is like tele-reality in that it combines multiple video feeds to construct a more complete view of the environment. RealityFlythrough is unlike telepresence in that the cameras are likely attached to people instead of robots, there are many more cameras, and the location and orientation of the cameras is not as easily controlled. It is unlike tele-reality in that the primary focus is not to create photo-realistic novel views, but to help users to internalize the spatial relationships between the views that are available.

All of this work (including RealityFlythrough) is differentiated by the assumptions that are made and the problems being solved. Telepresence assumes an environment where robots can maneuver, and has a specific benefit in environments that would typically be unreachable by humans (Mars, for example). Tele-reality assumes high density camera coverage, a lot of time to process the images, and extremely precise calibration of the equipment. The result is photorealism that is good enough for movie special effects ("The Matrix Revolutions" made ample use of this technology). An alternative tele-reality approach assumes a-priori acquisition of a model of the space [12], with the benefit of generating near photo-realistic live texturing of static structures. And finally, RealityFlythrough assumes mobile ubiquitous cameras of varying quality in an everyday environment. The resulting system supports such domains as SWAT team command and control support.

4 Requirements

In earlier work [10], we built a proof of concept system which revealed a number of rich requirements for harnessing ubiquitous video. Ubiquitous video is challenging because the cameras are everywhere, or at a minimum can go anywhere. They are inside, outside, carried by people, attached to cars, on city streets, and in parks. Ubiquity moves cameras from the quiet simplicity of the laboratory to the harsh reality of the wild. The wild is dynamic—with people and objects constantly on the move, and with uncontrolled lighting conditions; it is uncalibrated—with the locations of objects and cameras imprecisely measured; and it is variable—with video stream quality, and location accuracy varying by equipment being used, and the quantity of video streams varying by location and wireless coverage. Static surveillance-style cameras may be available, but it is more likely that cameras will be carried by people. Mobile cameras that tilt and sway with their operators present their own unique challenges. Not only may the position of the camera be inaccurately measured, but sampling latency can lead to additional errors.

Our proof of concept system revealed the need for better image quality, higher frame rates, greater sensor accuracy with faster update rates, and better support for the dynamic nature of ubiquitous video.

We used a SWAT team scenario as a concrete goal for what the system should be able to support. As mentioned in the introduction, the team commander currently maintains an internal spatial model of the incident site without the aid of visuals. The status quo places a heavy burden on the commander, and at least in the training exercises observed by Jones and Hinds [6] costly mistakes do happen. Our own discussion with a SWAT team member confirmed that the introduction of cameras in the field would be welcome. Not only would they provide the commander with the much needed visuals, but they would also reduce the amount of vocalization required by field officers, contributing to the stealth sometimes needed. Mobile cameras are a requirement in this domain because still cameras are hard to place and could easily be targeted and disabled. We should expect to have about 25 officers, and therefore 25 cameras, in the field.

Common knowledge about police operations combined with the previous description reveal minimum requirements for a system that could support SWAT: The system must work at novel sites with minimal config-

uration; cameras should be mobile and therefore wireless; the system needs to handle very incomplete camera coverage with fewer than 25 cameras in the field; and the system must work in unforgiving environments and should expect intermittent network connectivity.

Note that no SWAT teams have been involved in the design of the system thus far. SWAT teams will be consulted regarding user interface decisions after these basic requirements are met.

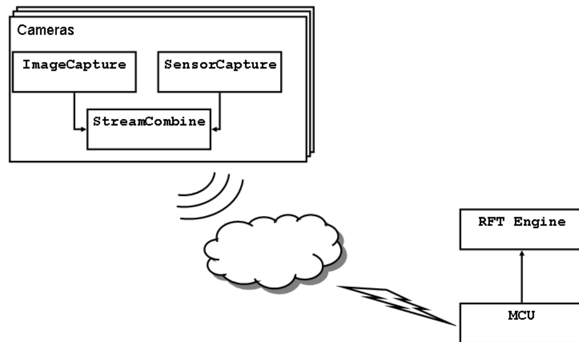

Figure 3: Component diagram showing system overview.

5 System Overview

Given the requirements just outlined, how might the system be built? First we need some cameras and location sensors. We need to capture the image data from a camera and compress it, and we also need to capture the sensor data. We call the components that do this, *Image Capture* and *Sensor Capture*, respectively. The data then needs to be combined so that we can match the sensor data to the appropriate frame in the image data. We call the component that handles this: *Stream Combine*. The resulting stream then needs to be sent across the network to a machine that decodes the data and presents it to the user. We have a modified *MCU* (Multipoint Control Unit) that does the decoding, and a *RealityFlythrough Engine* that combines the streams and presents the data to the user in a meaningful way. (Fig. 3 shows the relationships between these components.)

All of the video transmission components are based on the OpenH323 (`http://www.openh323.org`) implementation of the H323 video conferencing standard. In theory the system can support any H323 client without modification, but synchronizing the sensor data would be difficult. We have created our own client that merges the sensor data into the video stream. If the H323 standard required that clients generate a time stamp that could be used to synchronize with external data, the sensor data could be delivered by other means and synchronization could occur on the server. Without a synchronizing time stamp, though, we are forced to modify the client to get the precision we desire. It is still possible to obtain reasonable synchronization with unmodified clients as long as some assumptions can be made about network delays. We mention this only because stand-alone video conferencing units that do hardware video compression are already starting to emerge, and it was a key design decision to follow standards so that we could support third party components.

RealityFlythrough is written in C++ and makes heavy use of OpenGL for 3D graphics rendering, and the boost library (`http://boost.org`) for portable thread constructs and smart pointers. A projection library (`http://remotesensing.org/proj`) is used to convert latitude/longitude coordinates to planar NAD83 coordinates, and the Spatial Index Library (`http://www.cs.ucr.edu/~marioh/spatialindex`) is used for its implementation of the R-Tree datastructure [5] that stores camera locations. RealityFlythrough is designed to be portable and is confirmed to work on both Windows and Linux.

The *Engine* is roughly 16,000 lines of code (including comments), and the *MCU* is only an additional 2600 lines of code written on top of OpenH323.

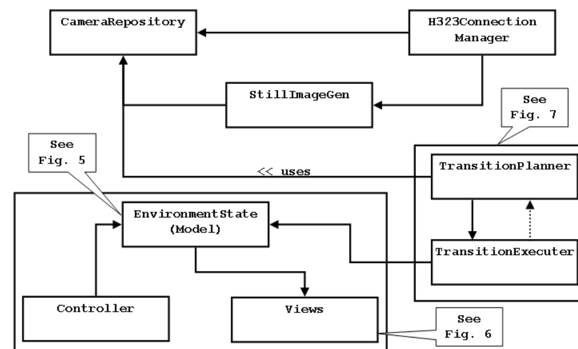

Figure 4: Component diagram showing an overview of the RealityFlythrough engine. Unlabeled arrows represent "calls" relationships. The dotted line is an event callback.

6 Engine Architecture

The *RealityFlythrough Engine* is the heart of the system. Given the available video streams and the user's intentions as input, the *engine* is responsible for deciding which images to display at any point in time, and for displaying them in the correct perspective. Fig. 4 shows the functional components of the engine. The standard Model-View-Controller design pattern [4] is used to represent and display the current system state. The *Still Image Generator* is responsible for producing and managing the still images that are generated from the live camera feeds. These still images are used to backfill transitions, but may also be worth viewing in their own right since they may not be much older than the live feeds.

The *Transition Planner/Executer* is responsible for determining the path that will be taken to the desired destination, and for choosing the images that will be displayed along that path. The *Transition Executer* part of the duo actually moves the user along the chosen path. And finally, the *Camera Repository* acts as the store for all known cameras. It maintains a spatial index of the cameras to allow for quick and targeted querying of cameras.

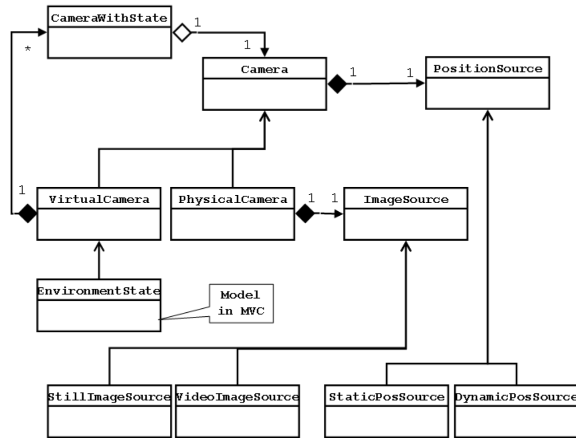

Figure 5: Class diagram showing the relationship of classes that are directly related to the Model in the MVC design pattern. For all class diagrams, open arrows represent inheritance, and arrows that have diamonds at the base represent containment. Filled in diamonds indicate that the contained object is created and destroyed by the container. Open diamonds indicate that the container only has a reference to the object.

6.1 Model-View-Controller

The objects that comprise the Model-View-Controller support the abstraction of infinite camera coverage that we are attempting to achieve. In Dijkstra's THE operating system [3], each layer in the layered architecture provides an additional level of abstraction on and insulation from the raw computer. We use the notion of a virtual camera (Fig. 5) to support the abstraction of infinite camera coverage. A virtual camera is simply a location, an orientation, a field of view, and a list of the "best" cameras that fill the field of view. The notion of "best" will be explored in depth in Section 6.3 where we discuss the Transition Planner/Executer, but for now it is sufficient to think of it as the camera that most closely matches the user's intentions. A virtual camera, then, can be composed of multiple cameras, including additional virtual cameras. This recursive definition allows for arbitrary complexity in how the view is rendered, while maintaining the simplicity suggested by the abstraction: cameras with an infinite range of view exist at every conceivable location and orientation.

Model. The concept of a virtual camera is extended all the way down to the *Environment State* (Fig. 5) which is the actual *model* class of the Model-View-Controller. The user's current state is always using the abstraction of a *Virtual Camera* even if the user is hitchhiking on a *Physical Camera*. In that particular case the *Virtual Camera* happens to have the exact position, orientation, and field of view of a *Physical Camera*, and hence the physical camera is selected as the "best" camera representing the view. The current state of the system, then, is represented by a *Virtual Camera*, and therefore by a position, an orientation, and the physical cameras that comprise the view. Changing the state is simply a matter of changing one of these three data points.

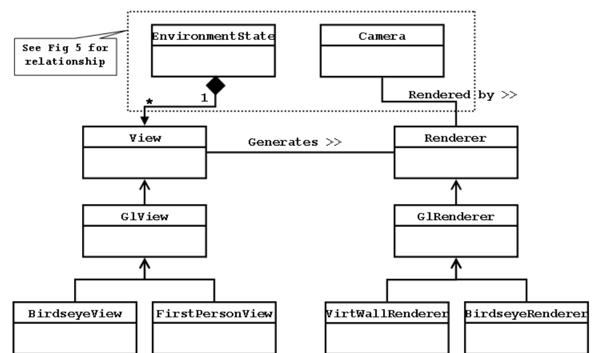

Figure 6: Class diagram for the classes involved in the View relationship of the MVC. The "Gl" in class names indicates that the classes are OpenGL-specific.

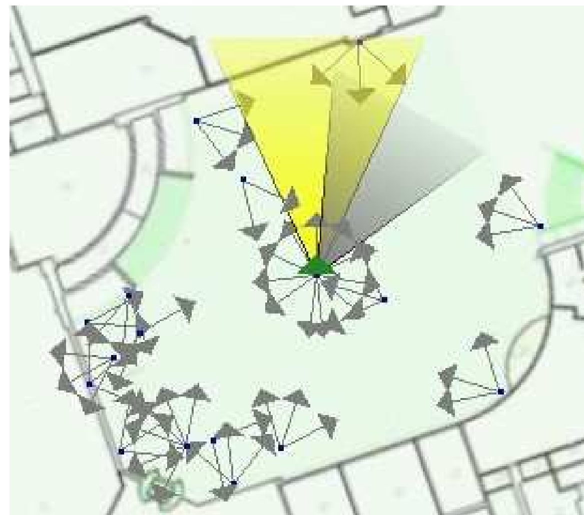

Figure 7: The birdseye view. The arrows represent the camera locations and directions of view. This picture corresponds to the transition in Fig. 1.

View The Model-View-Controller design pattern nat-

urally supports multiple views into the system state. There are currently two views (Fig. 6), but we envision more (see Section 7.3). The two views are the *First Person View* and the *Birdseye View*. The *First Person View* is the primary view that displays the images from a first person immersive perspective. This is the view that was described in Section 2. The *Birdseye View* shows a top down perspective on the scene, with cameras rendered as arrows and the field of view of active cameras displayed as cones emanating from the arrows (7).

The two views described above happen to both use OpenGL for rendering, but the use of OpenGL is not a requirement. Although it may seem that RealityFlythrough is heavily reliant on OpenGL, other than the views, there is only one class in the system that knows anything about OpenGL. That is the *Physical Camera GL* class which contains the OpenGL specific parts of a *Physical Camera*. By separating these concerns we have made it relatively simple to use other rendering platforms if desired, but more importantly other display methods can be used in concert with OpenGL.

As mentioned in the introduction, the *Birdseye View* not only provides a wide-area map view of the scene, but also reveals some of the rawness of ubiquitous video that is being abstracted away by the *First Person View*. The birdseye view makes the live camera coverage (or lack thereof) obvious and it reveals the ages and density of the still images that are used for backfill (see Section 6.2). There are currently three display modes available: (1) show all cameras, (2) show only the cameras that have been updated within some user specifiable interval, and (3) show only the live cameras. In an ideal environment, the user could ignore the information presented in the birdseye view because a live image would be present at every vantage point. A more typical scenario, and the one we adopted in the experiment described in Section 7, presents the user with the birdseye view that shows only the locations of the live cameras. The assumption, then, is that the intervening space is fully populated with still imagery. In this mode, the illusion of infinite camera coverage is still present, but the user is given some extra insight into where live camera coverage is available.

Each view instantiates one or more renderers to actually render the cameras that are involved in the current state. Since the definition of a *Virtual Camera* is recursive, there may be multiple cameras that need to be rendered. Each of these cameras has a state associated with it: the opacity (intensity) at which the camera's image should be drawn for the alpha blend. There are currently two types of renderers: *Virtual Wall Renderer* and *Birdseye Renderer*.

The *Virtual Wall Renderer* is used by the *First Person View*. It renders images using the virtual wall approximation described in Section 2. The images are rendered in a specific order, on the appropriate virtual walls, and with the opacity specified in their state.

The *Birdseye Renderer* simply draws either the camera arrow or the frustum cone depending on the current state of the camera.

Controller The controller is a typical MVC controller and does not require further comment.

6.2 Still Image Generation

Key to the success of the infinite camera abstraction is the presence of sufficient cameras. If no imagery is available at a particular location, no amount of trickery can produce an image. To handle this problem, we take snapshots of the live video feeds and generate additional physical cameras from these. A *Physical Camera* consists of an *Image Source* and a *Position Source* (Fig. 5). The *Image Source* is a class responsible for connecting to an image source and caching the images. The *Position Source*, similarly, is responsible for connecting to a position source and caching the position. A camera that represents still images, then, is simply a camera that has a static image source and a static position source. This is contrasted with live cameras that have a *Video Image Source* that continually updates the images to reflect the video feed that is being transmitted, and a *Dynamic Position Source* that is continually updated to reflect the current position and orientation of the camera.

To keep the still imagery as fresh as possible, the images are updated whenever a camera pans over a similar location. Rather than just update the *Image Source* of an existing camera, we have chosen to destroy the existing camera and create a new one. This makes it possible to do a transitional blend between the old image and the newer image, without requiring additional programming logic. The images fit neatly into our *Camera* abstraction.

The use of still imagery to help achieve the abstraction of infinite camera coverage is of course imprecise. There are two ways that the limits of the abstractions are disclosed to the user:

First, an age indicator bar is attached to the bottom of every image. The bar is bi-modal to give the user both high resolution age information for a short interval (we currently use 60 seconds), and lower resolution age information for a longer interval (currently 30 minutes). With a quick glance at the bottom of the screen, it is very easy for the user to get a sense of the age of an image. We originally used a sepia tone for older images, but in addition to this not giving the age granularity that was required, it also contradicted our aim to not mask reality. The user should see the images exactly as they were captured from the cameras. It is quite possible that information crucial to the user may be hidden by that kind of image manipulation.

The second way the system provides additional dis-

closure is by giving the user the option to never see older images. The user's preferences are used in the "best camera" calculation, and if no camera meets the criteria, the virtual camera will simply show a a virtual floor grid.

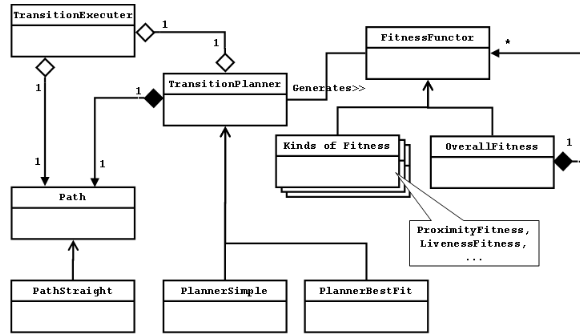

Figure 8: Class diagram showing the relationship of the classes involved in transition planning.

6.3 Transition Planner/Executer

When the user changes views, the *Transition Planner* (Fig. 8) is responsible for determining the path through space that will be taken and the images that will be shown along this path. The *Transition Executer* is responsible for moving the user along the chosen path. There is a high degree of coupling between the *planner* and the *executer* because of the dynamic nature of ubiquitous video. Consider a typical case where the user wishes to move to a live camera. A naive approach would determine the location and orientation of the live camera, compute the optimal trajectory to get to the target location and orientation, determine the images to be shown along the path, and finally execute the plan that was just developed. This approach does not work in a ubiquitous video environment for several reasons. The primary problem is that the destination camera may change its position and likely its orientation in the interval between when the plan was computed and when the execution of the plan has completed. The result will be a plan that takes the user to the wrong destination. Another problem is that the images that are selected along the path may not be the optimal ones. This is because the cameras that provide the intervening imagery may be live cameras as well, in which case their locations and orientations may have changed in the time since the plan was created. The result is that a live image that could have been shown is missed, or perhaps worse, a live image is shown that can no longer be seen from the current vantage point, so instead no image is displayed. Another possibility is that the dynamically generated still imagery is updated after the plan is generated, but the older image is displayed instead.

To account for all of these problems the transition planning needs to be done dynamically and interleaved with the execution. There are a number of competing issues that need to balanced when doing dynamic planning. It would seem that the ideal is to construct a plan at every time step, but some parts of the planning process are computationally expensive and need to be done sparingly. Also, the user needs to be given time to process the imagery that is being displayed, so even if a better image is available, showing it immediately may actually reduce comprehension.

The solution is to first introduce a dynamic *Path* object that takes a *Position Source* rather than a *Position* as its destination. The destination is now a moving target. At every time step, the *Path* can be queried to determine the current trajectory. With this trajectory, the *Transition Planner* can look ahead some interval and determine the best image to display. This image (camera, really) is added to the end of the camera queue. Each *Virtual Camera*—and since the *Transition Planner* acts on the *Environment State* remember that the *Environment State* is a virtual camera—maintains a fixed-length queue of cameras. When the queue is filled and a new camera is added, the camera at the front of the queue (the oldest or least relevant camera) is popped off the queue and thus removed from the *Virtual Camera*. The new camera is added with a time intensity, which means that the opacity gradually increases with time. We currently have the image blend to full opacity in one second.

This approach results in what appears to be a transition from one image to another, but along a dynamically changing path and with images that were used earlier still being displayed (if in view) to provide additional contextual information. The piece of the puzzle that is still missing is how the plan is constructed and adjusted dynamically. The *Transition Executer* (Fig. 8) is responsible for querying the *Path* at every time step and moving the user along the desired trajectory. It is also responsible for notifying the *Transition Planner* at time intervals set by the *planner*. These notification events give the *planner* the opportunity to determine which image (if any) to display next. Time is being used for signaling instead of "destination reached" because having the *Path* be dynamic means the destination may never be reached. Time is an adequate approximation of this signal point.

To determine the images to show during a transition the *Transition Planner* applies a series of *Fitness Functors* to each camera in the neighborhood. The *Fitness Functors* are weighted based on user preference. Some of the fitness dimensions are: proximity (how close is the camera to the specified position), rotation and pitch (how well do the orientations match), screen fill (how much of the screen would be filled with the image if it

were displayed), recency (how recently was the image acquired), and liveness (is the camera live or not).

To further increase the sensibility of transitions, three heuristics are used to decide which images to display: (1) The current image should stay in view for as long as possible, (2) once the *to* image can be seen from the current position, no other images should be displayed, and (3) there should be a minimum duration for sub-transitions to avoid jumpiness. The first two items are handled by always applying the *Fitness Functors* to the current camera and the ultimate target camera regardless of whether they pass the "in the neighborhood test", and then boosting the fitnesses by a configurable scalar value. This has the effect of giving extra weight to the current and target cameras, thus indirectly satisfying our heuristics. The third item is handled by adjusting the time interval used for *Transition Planner* callbacks.

6.4 Camera Repository

The *CameraRepository* is simply a container for all of the cameras (including the still cameras) that are known to the system. To support efficient spatial querying of the cameras, an R-Tree [5] is used to store the camera locations. The exact locations of the live cameras are not stored in the index because this would cause continuous updates to the index, and such precision is not necessary when doing "get cameras in neighborhood" queries. Instead, only location updates that are greater than a configurable threshold result in a replacement in the spatial index.

Each *physical camera* has certain fixed memory costs. To minimize the use of limited OpenGL resources, the cameras share a pool of texture maps. We have to store the image somewhere, though, so each camera (*Image Source*, really) allocates 768KB to store a 512x512 image (the size is dictated by OpenGL's texture map size requirements) at a depth of 24bits. After a period of inactivity, the *Image Source* frees memory by storing the image to disk. Under normal loads, there is no perceptible difference in performance when an image is read from disk.

7 Evaluation

An architecture must be evaluated along two dimensions: does it work, and will it work in the future? In this section we first present a user study that captures the essence of the user experience and shows that the abstractions presented are compelling and useful. Second, we examine performance to get insight into the scaleability of the system. Third, to evaluate how well the architecture will accommodate future changes to the application, we examine its robustness against a set of significant changes and extensions.

7.1 Effectiveness of the Abstraction

An earlier paper on RealityFlythrough [10] showed that users of the system had a positive experience and felt that they had more of feeling of "being there" than they would have had with the naive security monitor approach. We had these same subjects re-evaluate the experience with a system based on the architecture presented in this paper. This allows for a direct comparison between the two systems both in terms of usability and performance. While an architecture does impact usability and the kinds of interfaces that can be designed, our intent was not to evaluate the user interface. The appropriate user interface for the system depends on the application domain, and we expect to design one specific for SWAT teams.

To determine how the system was perceived by users, we repeated the earlier experiment as closely as possible. We used the same subjects, the same equipment on the user end, the same location for the flythrough. The camera operators were asked to behave as they did in the first experiment.

There were three hand-carried camera units in the field. They consisted of a standard logitech web camera (˜$100), a WAAS-enabled Garmin eTrex GPS (˜$125), a tilt sensor manufactured by AOSI (˜$600), and an 802.11b equipped laptop. The tilt sensor provides compass, tilt, and roll readings at 15hz. The video streams were transmitted using the OpenH323 video conferencing standard at CIF (352x288) resolution.

The subjects' task was: explore with the goal of getting a sense of what is happening, see if there is anyone they know, and determine if there is anything to draw them to the site for lunch. The experiment was run twice because some problems with the system were encountered on the first run. We discuss this first experiment not only because the problems are revealing, but also because the subjects' negative reactions underscore their frank views.

The first run of the new experiment was very positive from a technical standpoint. Three video streams connected successfully, and a large number of still images were automatically generated, quickly filling the entire region with cameras. Only 61 still cameras were used in the earlier version of the experiment, but 100's were generated in this one, greatly increasing the camera density. Despite the extra overhead incurred by auto-generating the images and by planning transitions on the fly, the system performance felt about the same. In fact, the subjects made the statement that the "performance was definitely much nicer." The new H263 video codec proved to be far superior to the H261 codec used previously. The frame rate varied by scene complexity, but appeared to average about 6-8 frames per second. The frame size was the same as was used previously, but the

image quality was better and the colors were much more vivid. The generated still images were clear and of good quality, validating the algorithm used to select frames to be converted into still images. On several occasions the subjects rapidly pointed out the age of images, indicating the success of the age indicator bar.

Even with all of these improvements, though, the subjects were not left with a positive impression and had to conclude that "from a usability standpoint, it went down." Transition sequences were met with comments like "it seems like it's awkward to move through several of those stills", and "[that] transition wasn't smooth." Post-experiment analysis identified three sources for the problems: (1) Too many images were being presented to the user, not allowing time for one transition to be processed mentally before another one was started. (2) The attempt to acquire a moving target resulted in an erratic path to the destination, causing disorientation. And, (3) no attempt was made to filter the location data by sensor accuracy. Still images were being generated even when the GPS accuracy was very low, so transitions involved nonsensical images detracting from scene comprehension.

Fortunately, none of these problems were difficult to handle. In Section 7.3 we will discuss the actual modifications made because these unplanned changes exemplify the architecture's robustness to changing requirements.

The experiment was repeated with much more positive results. Despite worse conditions at the experiment venue (we shared the space with a well attended Halloween costume contest), the subjects had much more positive comments such as, "Let's try one in the completely opposite direction. That was pretty nice.", and "It's pretty accurate where it's placing the images." "That was kind of cool. They weren't quite all in the same line, but I knew and felt like I was going in the right direction."

The costume contest placed some restrictions on where the camera operators could go, and also forced them to be in constant motion. The subjects found the constant motion to be annoying ("they're all over the map"), and the motion placed quite a strain on the new algorithm used to home in on a moving target. The subjects actually preferred the calmness of the still images. Midway through the experiment, we asked the operators to slow down a bit, and the experience improved dramatically: "Yeah, that's what it is. So long as [the camera operators'] rotation is smooth and slow, you can catch up to it and have smooth transitions."

The camera operators' motion was probably more erratic than normal, but the algorithm used to home in on dynamic cameras still needs to be improved. It only takes a fraction of a second for a person to turn her head 90 degrees, and people do this enough that we have to be able to handle it. There are two possible solutions to the homing problem: (1) Save the last 10 or so frames of a video stream and always display them with the live frame rendered on top. This would ensure that as the user homed in on the target, relevant imagery would be displayed in the vicinity. Part of what is disconcerting with the current system is that there is a clear boundary between the image and the background so the image appears to bob around as it comes into sight. Displaying earlier frames would help blur that boundary. (2) Modify the rotation speed of transitions to lock in on the destination early, and thus prevent the bobbing effect. Both of these possibilities will be discussed further in Section 7.3.

Server Capacity (# Live Cameras)

Decoding Only

- Roughly 1 Connection for 1% of CPU
- 100 Connections ⇒ 100% CPU
- > 113 Connections ⇒ packet loss

Full Experience @ 15fps

# Conn	CPU (%)	Fps achieved
10	85	15
15	**95**	**14**
20	100	10

Table 1: Summary of results.

7.2 System Performance

By measuring the performance of the system we hope to provide some insight into the scalability of the architecture. Raw performance metrics mainly measure the speed of the hardware and the quality of the compiler. Seeing how the raw numbers vary under certain conditions, however, reveals important details about the architecture.

The experiments with RealityFlythrough described thus far have only been run using at most three video streams. To determine the maximum number of simultaneous streams that can be handled by the server, we ran some simulations. The capacity of the wireless network forms the real limit, but since network bandwidth will continue to increase, it is instructive to determine the capacity of the server. We should estimate the capacity of a single 802.11b access point to give us a sense of scale, however. For the image size and quality used in the user studies, the H263 codec produces data at a rel-

atively constant 200Kbps. Empirical study of 802.11b throughput has shown that 6.205Mbps is the maximum that can be expected for applications [15]. This same study shows that the total throughput drops drastically as more nodes are added to the system. With more than eight nodes, total throughput decreases to roughly 2Mbps. This reduction means we cannot expect to have more than 10 streams supported by a single 802.11b access point.

The bottleneck on the server is the CPU. As more compressed video streams are added to the system, more processor time is required to decode them. Some of the other functional elements in RealityFlythrough are affected by the quantity of all cameras (including stills), but the experimental results show that it is the decoding of live streams that places a hard limit on the number of live cameras that can be supported.

The machine used for this study was a Dell Precision 450N, with a 3.06Ghz Xeon processor, 512MB of RAM, and a 128MB nVidia QuadroFX 1000 graphics card. It was running Windows XP Professional SP2. The video streams used in the simulation were real streams that included embedded sensor data. The same stream was used for all connections, but the location data was adjusted for each one to make the camera paths unique. Because the locations were adjusted, still image generation would mimic real circumstances. No image processing is performed by the engine, so replicating the same stream is acceptable for this study. The image streams were transmitted to the server across a 1Gbit ethernet connection. Since the image stream was already compressed, very little CPU was required on the transmitting end. A 1Gbit network can support more than 5000 simultaneous streams, far more than the server would be able to handle. Network bandwidth was not a concern.

To obtain a baseline for the number of streams that could be decoded by the server, we decoupled the *MCU* from the *engine*. In the resulting system, the streams were decoded but nothing was done with them. With this system, we found that each stream roughly equated to one percent of CPU utilization. 100 streams used just under 100 percent of the cpu. The addition of the 113th stream caused intermittent packet loss, with packet loss increasing dramatically as more streams were added. The loss of packets confirmed our expectation that the socket buffers would overflow under load.

Having confirmed that the addition of live cameras had a real impact on CPU utilization, we added the *RealityFlythrough engine* back to the system. We did not, however, add in the still image generation logic. To determine the load on the system we looked at both the CPU utilization and the system frame rate as new connections were made. The system frame rate is independent of the frame rates of the individual video feeds; it is the frame rate of the transitions. It is desirable to maintain a constant system frame rate because it is used in conjunction with the speed of travel to give the user a consistent feel for how long it takes to move a certain distance. As with regular video, it is desirable to have a higher frame rate so that motion appears smooth. To maintain a constant frame rate, the system sleeps for an interval between frames. It is important to have this idle time because other work (such as decoding video streams) needs to be done as well.

For this experiment, we set the frame rate at 15fps, a rate that delivers relatively smooth transitions and gives the CPU ample time to do other required processing. As Table 1 indicates, fifteen simultaneous video feeds is about the maximum the system can handle. The average frame rate dips to 14fps at this point, but the CPU utilization is not yet at 100 percent. This means that occasionally the load causes the frame rate to be a little behind, but in general it is keeping up. Jumping to 20 simultaneous connections pins the CPU at 100 percent, and causes the frame rate to drop down to 10fps. Once the CPU is at 100 percent, performance feels slower to the user. It takes longer for the system to respond to commands, and there is a noticeable pause during the transitions each time the path plan is re-computed.

To evaluate the cost of increasing the number of cameras, still image generation was turned on when the system load was reduced to the 15 connection sweet spot. Recall that still images are generated in a separate thread, and there is a fixed-size queue that limits the number of images that are considered. Still images are replaced with newer ones that are of better quality, and there can only be one camera in a certain radius and orientation range. What this means is that there are a finite number of still images that can exist within a certain area even if there are multiple live camera present. The only effect having multiple live cameras may have is to decrease the time it takes to arrive at maximum camera coverage, and to decrease the average age of the images. This assumes, of course, that the cameras are moving independently and all are equally likely to be at any point in the region being covered.

The live cameras were limited to a rectangular region that was 60x40 meters. A still image camera controlled a region with a three meter radius for orientations that were within 15 degrees. If there was a camera that was within three meters of the new camera and it had an orientation that was within 15 degrees of the new camera's orientation, it would be deleted.

We let the system get to a steady state of about 550 still images. The number of new images grows rapidly at first, but slows as the density increases and more of the new images just replace ones that already exist. It took roughly 5 minutes to increase from 525 stills to 550. At

this steady state, we again measured the frame rate at 14fps and the CPU utilization at the same 95 percent. The system still felt responsive from a user perspective.

These results indicate that it is not the increase in cameras and the resulting load on the R-Tree that is responsible for system degradation; it is instead the increase in the number of live cameras, and the processor cycles required to decode their images. This shows that the architecture is scalable. Since the decoding of each video stream can be executed independently, the number of streams that can be handled should scale linearly with both the quantity and speed of the processors available. Depending on the requirements of the user, it is possible to reduce both the bandwidth consumed and the processor time spent decoding by throttling the frame rates of the cameras not being viewed. This would reduce the number of still images that are generated; a tradeoff that only the user can make.

7.3 Robustness to Change

The investment made in an architecture is only warranted if it provides on-going value; in particular it should be durable with respect to changing user requirements, and aid the incorporation of the changes dictated by those new requirements. Below we discuss several such changes, some performed, others as yet planned. Only one of these changes was specifically anticipated in the design of the architecture.

7.3.1 Planned Modification

The hitchhiking metaphor has dominated our design up to this point. Another compelling modality for RealityFlythrough is best described as the virtual camera metaphor. Instead of selecting the video stream to view, the user chooses the position in space that she wishes to view, and the best available image for that location and orientation is displayed. "Best" can either refer to the quality of the fit or the recency of the image.

It should come as no surprise that the virtual camera metaphor inspired much of the present design, so there is a fairly straight forward implementation to support it. The *Virtual Camera* is already a first class citizen in the architecture. To handle a stationary virtual camera, the only piece required is a *Transition Planner* that runs periodically to determine the "best" image to display. Part of the virtual camera metaphor, though, is supporting free motion throughout the space using video game style navigation controls. The difficulty we will face implementing this mode is in minimizing the number of images that are displayed to prevent the disorienting image overload. This problem was easily managed with the hitchhiking mode because a fixed (or semi-fixed) path is being taken. The path allows the future to be predicted. The only predictive element available in the virtual camera mode is that the user will probably continue traveling in the same direction. It remains to be seen if this is an adequate model of behavior.

Another measure of a good architecture is that it is no more complicated than necessary; it does what it was designed to do and nothing more. The plan to support a virtual camera mode explains why the *Camera* is used as the primary representation for data in the system. Once still images, video cameras, and "views" are abstracted as cameras, they all become interchangeable allowing for the simple representation of complicated dynamic transitions between images.

7.3.2 Unplanned Modifications

In Section 7.1 we described three modifications to the system that needed to be made between the first and seconds runs of the experiment. Since these modifications were unplanned, they speak to the robustness of the architecture.

Reduce Image Overload. The goal of the first modification was to reduce the number of images that were displayed during transitions. This change had the most dramatic impact on the usability of the system, making the difference between a successful and unsuccessful experience. The modification was limited to the *Transition Planner*, and actually only involved tweaking some configuration parameters. In Section 6.3 it was revealed that the current and final destination cameras are given an additional boost in their fitness. Adjusting the value of this boost does not even require a re-start of the system.

Moving Camera Acquisition. The second modification also involved transition planning, but in this case the change occured in the *Path* class. The goal was to improve the user's experience as she transitions to a moving target. The partial solution to this problem—implemented for the second experiment—adjusts the path that the user takes so that she first moves to the destination camera's original location and orientation, and then does a final transition to the new location and orientation. This makes the bulk of the transition smooth, but the system may still need to make some course corrections during the final transition. We presented some possible solutions to this "last mile" problem in Section 7.1, and we explore them here. One option is to display the last 10 or so frames of the destination camera in addition to the current frame. This would have the effect of filling in the scene around the current field of view and reducing the jumpiness that the user experiences. Despite the apparent complexity of this suggestion, the modification is relatively straightforward because of the *Virtual Camera* abstraction. A *Physical Camera* can be created from each of the previous frames (as is done during still image generation), and these new cameras can be added onto

the camera queue that is a part of every *Virtual Camera*. The queue is a fixed size, so the older frames will naturally be pushed off and destroyed. All of the logic for rendering these additional cameras is already in place so no other modifications would be required. The other alternative suggested for handling the "last mile" problem was to boost the speed of the transition to make the final aquisition happen faster. This would create a Star Wars style "warp speed" effect. This modification can easily be handled by modifying the *Transition Planner* to boost the speed of the transition when it detects that it is in the "last mile" phase.

Location Accuracy Filtering. The final modification to the system was a little more substantial since it required modification to both the client and server software. The goal was to filter the still images on location accuracy. This change would have been trivial if we were already retrieving location accuracy from the sensors. As it was, the *Sensor Capture* component on the client had to be modified to grab the data, and then on the server side we had to add a location error field to our *Position* class. Now that every *Position* had an error associated with it, it was a simple matter to modify the *Still Image Generator* to do the filtering.

7.3.3 Future Modifications

Better High Level Abstraction. Forming continual correlations between the first-person-view and the 2d birdseye representation takes cognitive resources away from the flythrough scene and its transitions. We hope to be able to integrate most of the information that is present in the birdseye view into the main display. Techniques akin to Halos [1] may be of help.

This modification to the system should only affect the *First Person View*. Since we want to present the state information that is already available in the *Birdseye View*, that same information need only be re-rendered in a way that is consistent with the *First Person View*. If we want to create a wider field of view we could increase the field of view for the virtual camera that makes up the view. Another possibility is to generate additional views that are controlled by other virtual cameras. For example a window on the right of the display could be controlled by a virtual camera that has a position source offset by 45 degrees.

Sound. Sound is a great medium for providing context, and could be an inexpensive complement to video. By capturing the sound recorded by all nearby cameras, and projecting it into the appropriate speakers at the appropriate volumes to preserve spatial context, a user's sense of what is going on around the currently viewed camera should be enhanced.

Sound will be treated like video. Each *Physical Camera* will have a *Sound Source* added to it, and new views supporting sound will be created. There might be a *3D Sound View* which projects neighboring sounds, and a regular *Sound View* for playing the sound associated with the dominant camera.

Scale to Multiple Viewers with Multiple Servers. Currently RealityFlythrough only supports a single user. How might the system scale to support multiple users? The *MCU* component currently resides on the same machine as the *engine*. One possibility is to move the *MCU* to a separate server which can be done relatively easy since the coupling is weak. The problem with this approach, though, is that the *MCU* is decompressing the data. We would either have to re-compress the data, which takes time, or send the data uncompressed, which takes a tremendous amount of bandwidth. A better approach would be to leave the *MCU* where it is and introduce a new relay *MCU* on the new server layer. The purpose of the relay *MCU* would be to field incoming calls, notify the *MCU* of the new connections, and if the *MCU* subscribed to a stream, forward the compressed stream.

With the latter approach we could also support connecting to multiple servers. The *MCU* is already capable of handling multiple incoming connections, so the main issue would be one of discovery. How would the viewer know what server/s could be connected to? What would the topography of the network look like? We leave these questions for future work.

It is not clear where still image generation would occur in such a model. The easiest solution is to leave it where it is: on the viewing machine. This has the additional benefit of putting control of image generation in the individual user's hands. This benefit has a drawback, though. Still images can only be generated if the user is subscribed to a particular location, and then only if there are live cameras in that location. What if a user wants to visit a location at night when it is dark? It's possible that the user wants to see the scene at night, but it is equally likely that she wants to see older daytime imagery. If the still images are captured server side, this would be possible.

Since server-side still image generation may stress the architecture, we consider it here. The *engine* would not have to change much. We would need a *Still Image Generated* listener to receive notifications about newly generated cameras. A corresponding *Still Image Destroyed* listener may also be required. The camera that is created would have a new *Image Source* type called *Remote Image Source*. The *Position Source* would remain locally static. The *Remote Image Source* could either pre-cache the image, or request it on the fly as is currently done. Performance would dictate which route to take.

8 Conclusion

Each of the modifications presented is limited to very specific components in the architecture. This indicates that the criteria used for separating concerns and componentizing the system was sound. Our experiments showed good support for the desired abstractions, as well as excellent scalability in the number of live video sources and *Cameras*.

We have presented an architecture for a system that harnesses ubiquitous video by providing the abstraction of infinite camera coverage in an environment that has few live cameras. We accomplished this abstraction by filling in the gaps in coverage with the most recent still images that were captured during camera pans. The architecture is able to support this abstraction primarily because of the following design decisions: (1) The *Camera* is the primary representation for data in the system, and is the base class for live video cameras, still images, virtual cameras, and even the environment state. Because all of these constructs are treated as a camera, they can be interchanged, providing the user with the best possible view from every vantage point. (2) The *Transition Planner* is an independent unit that dynamically plans the route to a moving target and determines the imagery to display along the way. New imagery is displayed using an alpha blend which provides the illusion of seamlessness while at the same time revealing inconsistencies. The system is providing full disclosure: helping the user make sense of the imagery, but revealing inconsistencies that may be important to scene comprehension. Because the *Transition Planner* is responsible for path planning, image selection, and the blending of the imagery, it has a huge impact on the success of RealityFlythrough. Having the control of such important experience characteristics in a single componenent and having many of those characteristics be user controllable is key to the sucess of the current design. (3) Agressive use of the Model-View-Controller design pattern made the addition of the very important *Birdseye View* trivial, and allows for future expandability. Many of the enhancements outlined in the previous section will be accomplished by simply adding new views.

References

[1] P. Baudisch and R. Rosenholtz. Halo: a technique for visualizing off-screen objects. In *Proceedings of the conference on Human factors in computing systems*, pages 481–488. ACM Press, 2003.

[2] D. Brin. *The Transparent Society*. Perseus Books, 1998.

[3] E. W. Dijkstra. The structure of the "THE"-multiprogramming system. *Comm. ACM*, 11(5):341–346, 1968.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[5] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 47–57. ACM Press, 1984.

[6] H. Jones and P. Hinds. Extreme work teams: using swat teams as a model for coordinating distributed robots. In *Proceedings of the 2002 ACM conference on Computer supported cooperative work*, pages 372–381. ACM Press, 2002.

[7] T. Kanade, P. Rander, S. Vedula, and H. Saito. Virtualized reality: digitizing a 3d time varying event as is and in real time, 1999.

[8] H. Kuzuoka, G. Ishimo, Y. Nishimura, R. Suzuki, and K. Kondo. Can the gesturecam be a surrogate? In *ECSCW*, pages 179–, 1995.

[9] J. Leigh, A. E. Johnson, T. A. DeFanti, and M. D. Brown. A review of tele-immersive applications in the CAVE research network. In *VR*, pages 180–, 1999.

[10] N. J. McCurdy and W. G. Griswold. Harnessing mobile ubiquitous video. http://www.cse.ucsd.edu/users/wgg/CSE118/rtf-percom-sub.pdf, 2004.

[11] N. J. McCurdy and W. G. Griswold. Tele-reality in the wild. UBICOMP'04 Adjunct Proceedings, 2004. `http://activecampus2.ucsd.edu/~nemccurd/tele_reality_wild_video.wmv`.

[12] U. Neumann, S. You, J. Hu, B. Jiang, and J. Lee. Augmented virtual environments (ave) for visualization of dynamic imagery. http://imsc.usc.edu/research/project/virtcamp/ave.pdf, undated.

[13] R. Szeliski. Image mosaicing for tele-reality applications. In *WACV94*, pages 44–53, 1994.

[14] S. Tachi. Real-time remote robotics - toward networked telexistence. In *IEEE Computer Graphics and Applications*, pages 6–9, 1998.

[15] T. I. Wlans. An empirical characterization of instantaneous throughput in 802.11b wlans.