2005 IEEE/RSJ International Conference on Intelligent Robots and Systems

A middleware for a large array of cameras

Lee Middleton, Sylvia C. Wong, Michael O Jewell, John N. Carter, Mark S. Nixon
School of Electronics and Computer Science
University of Southampton, United Kingdom
Email : {ljm,sw2,moj,jnc,msn}@ecs.soton.ac.uk

***Abstract*— Large arrays of cameras are increasingly being employed for producing high quality image sequences needed for motion analysis research. This leads to the logistical problem with coordination and control of a large number of cameras. In this paper, we used a lightweight multi-agent system for coordinating such camera arrays. The agent framework provides more than a remote sensor access API. It allows reconfigurable and transparent access to cameras, as well as software agents capable of intelligent processing. Furthermore, it eases maintenance by encouraging code reuse. Additionally, our agent system includes an automatic discovery mechanism at startup, and multiple language bindings. Performance tests showed the lightweight nature of the framework while validating its correctness and scalability. Two different camera agents were implemented to provide access to a large array of distributed cameras. Correct operation of these camera agents was confirmed via several image processing agents.**

I. Introduction

Increasingly, people are using large arrays of high quality cameras for capture and analysis of motion. Kanade et al. mounted 49 cameras in a room to capture motion for virtualised reality [6]. Wilburn et al. built a dense camera array to achieve high resolution and high framerate capturing of image data using many low resolution CMOS cameras [13]. Zhang and Chen built a large self configuring camera array capable of rendering novel views of scenes in near real time [16]. In all these works multiple cameras are employed to produce a higher quality image than would be possible with a single camera. Specifically, we seek to deploy high resolution video rate data captured from multiple cameras for the analysis of human gait.

In this paper we propose a middleware framework for a camera data acquisition system. The aim of this framework is to allow transparent and reconfigurable access to visual data and image processing software while minimising maintenance

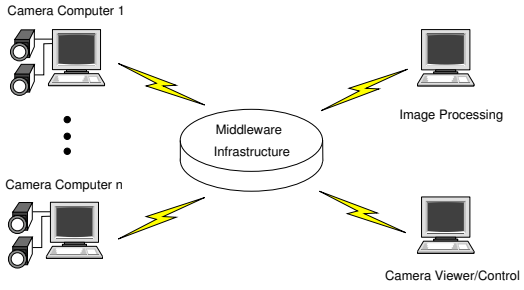


Fig. 1. Overview of how the middleware and camera systems fit together.

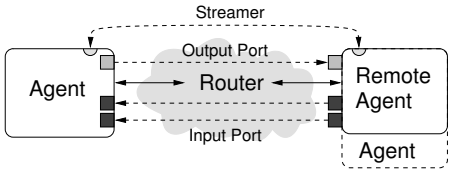


Fig. 2. An overview of how the middleware and agents fit together.

by encouraging code reuse. Fig. 1 shows the system overview of the proposed framework. The middleware facilitates communication between dedicated camera computers and image and gait processing software. It has the following features:

- *Zeroconf* [14]: This allows agents to automatically locate middleware components in a TCP/IP network.
- *Multi language support*: This allows users to exploit the benefits of different languages.
- *Lightweight*: The algorithms used in gait analysis are CPU intensive, thus the middleware must not be an additional drain on resources.
- *Service discovery*: Agents can query the middleware to discover and utilise services provided by other agents.
- *Locking*: Cameras are stateful devices. It is important that processes cannot be interrupted mid-session.

Hori et al. [4] also implemented a middleware for networks of computers with attached cameras. In their work, the cameras were accessed as if they were a local device with commands such as `camera.capture()`. This is similar to player/stage [2], a popular middleware among the robotics community. Player/stage provides software abstraction for a large variety of robot sensors, such as pan/tilt cameras, sonar and laser range finders, and wheel encoders. However, these systems are not suitable for our application as their goals are to provide direct access to sensor data over a network. In contrast, we want intelligent agents in our framework, where researchers can provide agents that do, for example, background subtraction or image mosaicing.

Multi-camera tracking systems [8], [9] also attempt to look at the problem of controlling a large array of cameras. Here, the camera agents not only act as capture devices, but also perform processing on the image data. The middleware is highly focused on the task of tracking. For instance, the messages in Sato et al. [9] are high level commands like location of outgoing object and measured height. The middleware is also responsible for coordinating the movement and focus of the cameras to achieve the system goal. In comparison, the

middleware in our application has to be more general purpose. This is because researchers have different objectives from the image data. For example, in our research group, there are people who work with raw image data, silhouettes [11], and 2D and 3D models [12].

A final approach to this camera coordination problem is to leverage an existing middleware. The most common is CORBA [3]. It was originally developed for business applications but has since been used in many problem domains [15]. The middleware is very generic. A stub is written in an interface description language and used to create a client to the middleware. CORBA additionally offers many services such as brokers and a directory. However, a CORBA orb (the centre of the middleware) requires a lot of resources to run. Many features not required in our application are included by default, and they cannot be optionally switched off. Also, there is a steep learning curve before researchers can add their existing code to this framework. Consequently, CORBA was not suitable for our application.

The solution we propose contains some of the features from all the approaches outlined above, while being easy to use and lightweight. This paper extends an earlier version of this work published in [7]. It is organised as follows: Section II provides an overview of the Lightweight Agent Framework (LAF). This is followed by an introduction to two agents essential to our camera data acquisition system, camera and super camera, in Section III and Section IV. Section V presents results from simple performance tests. Finally, Section VI describes four agents that employ the services provided by the camera and super camera agents.

II. System Overview

Fig. 2 shows an overview of LAF. The system has been simultaneously developed on C++, Java, and Python. Central to the system is the *router*. It is the main point of communication and coordination. All *messages* between components (except streamers) are sent via the router. Also, it acts as a broker for agents providing and requiring services. Agents are providers of services. Remote agents are clients of services provided by agents. To use the service provided by an agent, a remote agent requests a *lock* on the agent from the router. However, LAF is not restricted to a simple model of clients (remote agents) and servers (agents). An agent can contain one or more remote agents, thus allowing it to be both a client and a server at the same time. An example of an agent which is also a remote agent is described in Section IV. Agents and remote agents communicate via ports and streamers. Ports are inputs and outputs of agents. Streamers are direct socket connections, mediated via the router. This allows video information to be sent directly between agents without the traffic passing through the router.

A. Ports and Streamers

There are two types of ports in LAF: input and output. Input ports are used to pass data to an agent. It can be optional or non-optional. Processing (via the `CALL` command, see Section II-B) cannot proceed until all non-optional ports are *set*. Output ports are used to pass data to remote agents. As an example, if a user wants an agent to add two strings together. The agent should have, as a minimum, two input ports and one output port. The input ports should be non-optional, as concatenation is not possible without both inputs *set*. The concatenated string can be read from the output port.

In applications involving a large array of cameras, a large amount of data is generated. If this data is to be sent via the router, this could result in more traffic than the router can deal with. For this reason, streamers were implemented. A streamer is a direct socket connection between an agent and a remote agent. The router is responsible for instigating this connection, so address (IP and port) information is not required ahead of time.

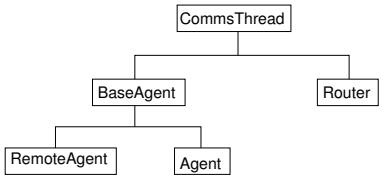


Fig. 3. Diagram showing the interrelation of the software objects.

B. Messages

LAF employs XML messages for communication between the router, agents and remote agents. The structure of the message is defined using XML schema [1]:

```
<complexType name="message">
  <sequence>
    <element name="command" type="string"/>
    <element name="sender" type="string"/>
    <element name="target" type="string"/>
    <element name="name" type="string" minOccurs="0"/>
    <element name="data" type="string" minOccurs="0"/>
  </sequence>
  <attribute name="id" type="nonNegativeInteger" use="
      required">
  <attribute name="replyto" type="nonNegativeInteger">
</complexType>
```

Broadly, there are three classes of messages – status, router and agent. There are two status messages, `OK` and `NOK`, which give feedback about the success of an action. Router messages are actions that only the router can perform. Two router commands are defined in LAF: `PING` and `SHOW`[1]. Agent messages are communications, via the router, between agents and remote agents. They deal with control and communication. Agent messages recognised by LAF are `SUBSCRIBE`, `UNSUBSCRIBE`, `LOCK`, `UNLOCK`, `SETPORT`, `GETPORT`, `GETSTREAMER` and `CALL`[2]. The use of some of these commands will be explained in subsequent sections.

C. Router

The router, agent and remote agent are implemented in a common class hierarchy, as shown in Fig. 3. At the top level is `CommsThread`, which contains the low level threading and

[1] `SHOW` returns a list of all agents currently registered with the router.
[2] `CALL` executes the action of an agent.

networking. Below CommsThread are two derived classes: Router and BaseAgent. BaseAgent implements the ports mechanism and provides a connection handler. Inherited from the BaseAgent are Agent and RemoteAgent.

The router is responsible for agent subscription, message re-direction, and agent selection. It employs a plugin system which makes it simple to extend its functionality.

Upon starting, the router registers itself as a multicast DNS service (mDNS) in zeroconf. The mDNS service allows information to be passed to any subscriber of the service. In this case the port and IP address of the router are passed via mDNS. Essentially this means that connection to the router by any agent is potentially an automatic process.

The subscription process of an agent from the perspective of the router begins when a SUBSCRIBE message is received. This message contains the *type* of agent which is being subscribed. The type is something like string.concat which could be the name of an agent which concatenates strings together. The type of an agent is purely a descriptive name describing the service it provides. When the subscription is received the router assigns a unique name for the agent. This is made up of the type and a unique (for the lifetime of the router) id number. For example the name assigned by the router in the case of the string concatenation agent could be string.concat0. Once an agent is subscribed, it is added to two lists (connected agents and free agents) which is maintained in the router. If an agent unsubscribes or dies, the router removes it from both lists.

All messages pass through the router. As a result of this the router can perform filtering of the messages. For example, some messages are permitted only if the sender has a lock on the target, for example SETPORT, GETPORT and CALL. If the sender does not have the required lock, the router returns an NOK message to the sender. In most cases however the extent of the routers manipulation of the message is handling acknowledgements and passing it onward to the appropriate target.

The last function of the router is the agent selection process. This is used primarily when a remote agent is trying to lock an agent. The selection mechanism currently employed is a naïve one. Basically, a list of free agents is maintained. Acquisition of a lock involves taking the first free agent and returning it to the remote agent. This agent is then removed from the free agent list and added to a locked agent list.

D. Agents and Remote agents

Users writing agents for LAF will need to create a derived class of Agent. The Agent class provides the underlying networking and messaging required for all agents. The derived class will minimally need to (a) provide type, ports, and streamers in the constructor, and (b) overwrite the action() method. The type is a string which describes the service which the agent provides. This is used by remote agents to request its service. Ports and streamers provide the inputs and outputs of this agent. The action method is the engine of the agent. The user writes their own action method to provide the agents' functionality. In the case of a string concatenation agent this method will read the input ports and set the output port appropriately. An example of this is shown in Fig. 4.

```
class TestAgent( Agent ):
  def __init__( self ):
    Agent.__init__( self )
    self.setIdentityPort( 'type', 'string.concat' )
    self.addPort( 'a', self.INPUT )
    self.addPort( 'b', self.INPUT )
    self.addPort( 'c', self.OUTPUT )
  def action( self ):
    if self.setOutputPort( 'c', self.getInputPort('a').
        upper() + self.getInputPort('b').upper() ):
      return True
    return False

if __name__=='__main__':
  p = TestAgent( )
  if p.connect():
    p.join( )
```

Fig. 4. Python string concatenation agent.

```
int main(void) {
  RemoteAgent ra();
  if(!ra.connect_and_lock("string.concat"))
    return 0;
  ra.setInputPort( "a", "foo" );
  ra.setInputPort( "b", "bar" );
  ra.call();
  cout << ra.getOutputPort( "c" ) << "\n";
  ra.disconnect_and_unlock();
  return 0;
}
```

Fig. 5. C++ string concatenation remote agent.

Users writing remote agents will need to create an instance of RemoteAgent (or a derived class). The instance will need to specify the name(s) of the agent(s) it wishes to lock, set the agent's input ports and execute the CALL message. The CALL message runs the locked agent's action() method. An example is shown in Fig. 5.

Agents and remote agents can connect to LAF either automatically or manually. Automatic connection is performed using zeroconf. Manual connection uses environment variables ROUTERHOST and ROUTERPORT.

III. Camera Agent

Since the goal of LAF is to coordinate an array of cameras, the first agent implemented was naturally one that provides transparent access and control of cameras over the network. Currently, we are using Point Grey Dragonfly firewire cameras in our laboratory. However, the agent is not camera platform dependent and other cameras can be controlled as long as drivers for accessing the camera are available.

With multiple cameras connected to the firewire bus, the capture process happens in one operation. By this it is meant that data from each camera is transferred via the bus to the user in a single read operation. Therefore a single agent is used to abstract all cameras on the bus. Study of usage patterns reveals two distinct modes of behaviour that the camera agent has to encapsulate – grabbing image sequences

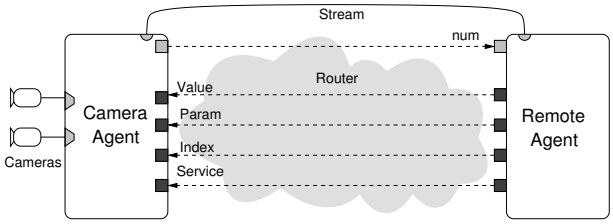


Fig. 6. Configuration of a camera agent.

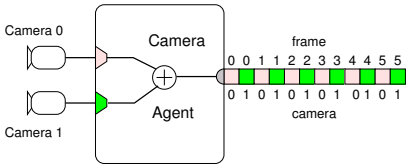


Fig. 7. Sending image data via a streamer in the camera agent.

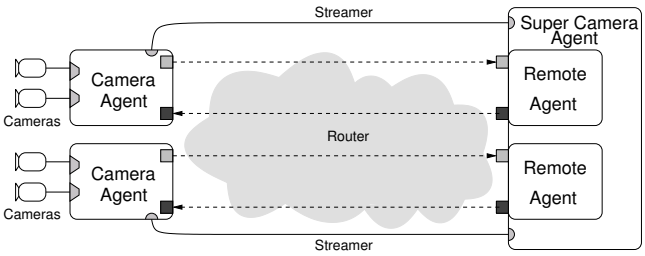


Fig. 8. Configuration of a super camera agent.

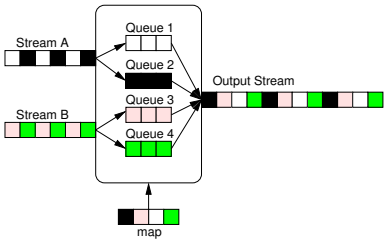


Fig. 9. Demangling of image streams in super camera agent.

and configuring camera features. Transmission of image data is achieved with a *streamer*. Fig. 6 illustrates the specific ports and general configuration for a camera agent and how it interacts with a remote agent. The agent was implemented in C++ as it required access to device level calls to control the camera.

Of the illustrated input ports, only *service* was compulsory. The service port exposes the features of the cameras controlled by the agent, such as grab image, set shutter speed, and turn on auto white balance. The other input ports were employed to set required parameters, such as the value of the shutter speed.

The streamer is used to send the video data directly to a remote agent. The video data is sent a frame at a time with the frame from each camera on the bus interleaved as shown in Fig. 7.

Two different sorts of camera agents have been implemented. The first delivers the raw bayer information and the second delivers colour information. When transmitting image data the colour camera agent requires 3 times the bandwidth of the bayer one. This is a consequence of colour data requiring 3 bytes to represent each pixel (one for each of the colour channels). There is scope to write other sorts of camera agents which could potentially perform more complex image processing operations. For instance to save on transmission bandwidth an image which is cropped about a region of interest could be sent. Alternatively compression could be used to save bandwidth. Note that any such modifications would result in different data being sent via the streamer.

IV. Super Camera Agent

The camera agent provides access to the camera array on a per PC basis. However, accessing and controlling a large number of cameras can be cumbersome using camera agents alone. For instance to access 6 cameras which are connected in pairs to three PCs, the user needs to maintain three separate camera agents. Moreover, when cameras are connected to the firewire bus they do not necessarily get allocated in the order of connection but in the order that the operating system first polls them. This can lead to inconsistencies with experimental set up from one day to the next. Primarily to solve these issues the super camera agent was written in C++.

The configuration of the super camera agent is shown in Fig. 8. It contains a number of remote agents which control all camera agents connected to the router. Thus in terms of the middleware system the super camera agent is both a remote agent and an agent. The image data is transmitted to the super camera agent via a number of streamers - one for each camera agent connected. An output streamer is provided for remote agents to read the collated image data. Ports are used to control features of the camera agents. Similar to the camera agent, the only compulsory input port is the *service* port.

To handle the incoming streams and the outgoing stream a design pattern known as a producer-consumer model is employed. This is illustrated in Fig. 9. Each incoming stream is assigned a producer which appends each completed frame as it arrives onto a queue. In this way each queue corresponds to a unique camera on each of the camera agents. The consumer then takes one frame from each queue in turn and appends it to the output stream. The order in which it takes the items from the queues depends on an internal map. As each camera has a unique serial number then this serial number can be associated with one of the queues. The map is just a lookup table which says what order the queues should be emptied in terms of the serial numbers.

V. Performance Testing

Four tests were carried out to evaluate the performance of LAF. The first test evaluates scalability. Fig. 10 shows that connection and disconnection times to LAF do not increase with an increasing number of registered agents. The second test measures the overhead of messaging. One hundred separate operations were invoked against a string

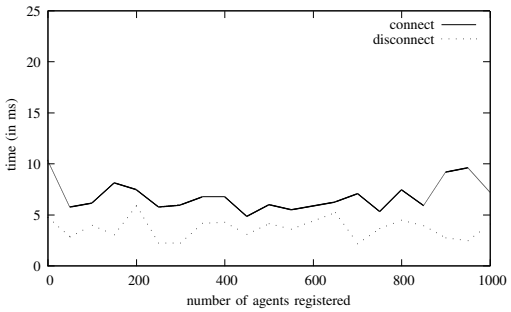


Fig. 10. Connection and disconnection time against number of agents registered.

	remote agents	
agents	C++	Python
C++	943	1771
Python	1596	2158

TABLE I

TIME (IN MS) TO PERFORM 100 STRING CONCATENATION OPERATIONS.

concatenation agent. The result is shown in Table I. The third test measures the average startup times of agents and is illustrated in Table II. The slower start up times of the remote agents is the overhead due to locking. The messaging test and the startup time test demonstrate the lightweight nature of LAF. The fourth test evaluates the streaming performance of streamers. For a camera agent we achieved an average of 661 Mbit/s with a gigabit network. This means we can directly stream video data from 9 cameras[3].

VI. APPLICATION AGENTS

As an example of the system in operation several application agents which have been developed are described here. The first allows the cameras to be configured remotely, the second is for performing a simple mosaicing operation of the images of all the cameras on the network, and the final one performs background subtraction on images.

A. Camera control application

Scientists in our laboratory often require different camera configurations. Thus, an application for configuring the cameras was written. This application used the locking mechanism of LAF to stop simultaneous camera access. The application interface and current camera view are shown in

[3]Our cameras run at a resolution of 640×480 with a framerate of 30 frames per second.

	time (ms)
C++ agent	263
C++ remote agent	291
Python agent	512
Python remote agent	604

TABLE II

STARTUP TIME.

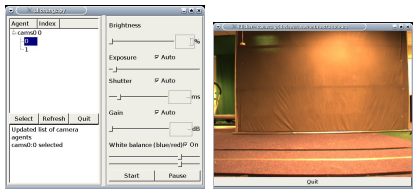


Fig. 11. The gui application controlling a camera.

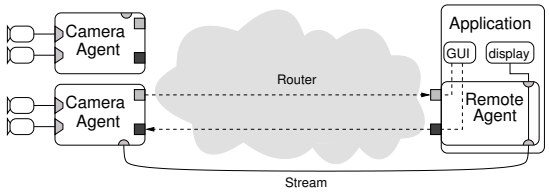


Fig. 12. Schematic showing the design of the camera control application.

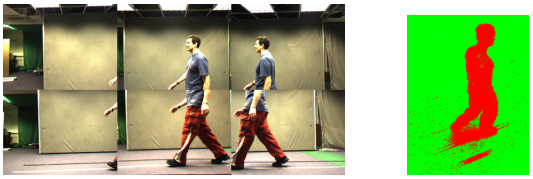
Fig. 11. This application was written using the C++ bindings. The design of this application is similar to the camera agent as shown in Fig. 12. Notice that only one remote agent is needed. This remote agent is locked to a particular camera agent via the router and unlocked when the user selects another camera. If the newly selected camera is also controlled by the same camera agent then the unlocking process is not performed. In order to send the video data a streamer is used. When the camera is changed, the current streamer is shut down and a new one is reconnected. To control the various features of the camera the service input port to is set to the feature of interest.

B. Image mosaicing

An image mosaicing agent was written in Python. The output from one frame of an example run is given in Fig. 13(a). This was achieved with 3 PCs each with 2 cameras connected. The program locked a super camera agent and then grabbed a few seconds worth of data. From this data a large composite image was created. Notice that this program requires no knowledge of the camera arrangement (as evidenced by the mosaic), or the numbers of cameras connected as this is handled automatically by the super camera agent.

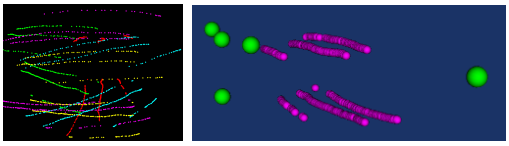
C. Background subtraction

A common first step of many of image processing algorithms is background subtraction. This process divides an image to foreground (moving) and background (static) parts



(a) (b)

Fig. 13. (a) Creation of an uncalibrated image mosaic from six cameras. (b) Removal of the background from a single camera.



(a) (b)

Fig. 14. (a) Path of laser pointer viewed from five cameras. (b) Calibrated reconstruction of the original path.

and remove the background. In gait analysis, this effectively highlights the gait subject. An agent was written in Python to perform background subtraction. Fig. 13(b) shows an example output frame. This agent locks a single camera and statistically computes a reference background when no subject is in view. When the subject is in the scene, the reference is used to find portions of the image that have changed.

D. Camera calibration

3D processing of image data requires cameras to be calibrated. Calibration establishes the parameters of the cameras, eg pose and CCD configurations. Projective factorisation is a simple method for calibration [10]. As input, this requires a known set of 3D correspondences such as the path of a laser pointer viewed from multiple cameras. To perform this, a camera agent was extended to process raw video data and return coordinates of the laser pointer. A remote agent was written to collect these coordinates and perform projective factorisation. Figure 14 shows the path of the laser pointer as viewed by 5 different cameras.

VII. Conclusions

This paper described the development of a middleware, LAF, for the control of a large array of cameras. LAF consists of a router, and superclasses for agents (service providers) and remote agents (clients). Since agents can contain remote agents, LAF is not limited to a simple client/server model. LAF provides transparent access to camera control. Additionally, it facilitates access and reuse of intelligent agents that provide a variety of image processing operations. Other features of LAF include minimal computational overhead, zeroconf for automatic discovery of components in the framework, locking, and multiple language support. LAF has been shown to be easy to use via the `string.concat` example. In this paper, LAF has been used to control an array of cameras. However, the framework is general purpose and is not solely limited to controlling camera arrays [5].

The performance of LAF is tested by scalability, messaging and streaming tests. Agents implemented with the C++ binding were found to be faster than those using the Python binding. However, the Python binding is still very useful as it allows rapid code development. The streaming test showed that LAF is capable of operating real time streams of video data from 9 cameras. This is sufficient for our needs but if more cameras are required in the future compression at the camera agent end needs to be examined. Four application agents were also demonstrated in this paper. One which allows users to control the configuration of the cameras, one which performs a simple mosaicing of the video data, and one which performs background subtraction. These applications validated the correctness of the design and demonstrated the ease of implementation of simple applications.

VIII. Acknowledgements

The authors gratefully acknowledge support by the Defence Technology Centre 8-10 supported by General Dynamics, UK.

References

[1] David C. Fallside and Priscilla Walmsley (Editors). XML schema part 0: Primer second edition. W3c recommendation, W3C, October 2004.

[2] Brian Gerkey, Richard T. Vaungan, and Andrew Howard. The player/stage project: Tools for multi-robot and distributed sensor systems. In *Proceedings of the 11th International Conference on Advanced Robotics*, pages 317–323, June 2003.

[3] Michi Henning and Stephen Vinoski. *Advanced CORBA Programming with C++*. Addison Wesley, 1999.

[4] Toshio Hori, Yoshifumi Nishada, Nobuyuki Yamasaki, and Horishi Aizawa. Design and implementation of a reconfigurable middleware for sensorized environments. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*, volume 2, pages 1845–1850, 2003.

[5] Michael O. Jewell, Lee Middleton, Mark S. Nixon, Adam Prügel-Bennett, and Sylvia C. Wong. A distributed approach to musical composition. In *Proceedings of 9th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems (KES)*, 2005.

[6] T. Kanade, H. Saito, and S. Vedula. The 3d-room: Digitizing time-varying 3d events by synchronized multiple video streams. Technical report, Carnegie Mellon University, 1998.

[7] Lee Middleton, Sylvia C. Wong, Michael O. Jewell, John N. Carter, and Mark S. Nixon. Lightweight agent framework for camera array applications. In *Proceedings of 9th International Conference on Knowledge-Based & Intelligent Information & Engineering Systems (KES)*, 2005.

[8] J. Orwell, S. Massey, P. Remagnino, D. Greenhill, and G. A. Jones. A multi-agent framework for visual surveillance. In *IAPR International Conference on Image Analysis and Processing*, pages 1104–1107, 1999.

[9] K. Sato, T. Maeda, H. Kato, and S. Inokuchi. CAD-based object tracking with distributed monocular camera for security monitoring. In *Proceedings of the Second CAD-Based Vision Workshop*, pages 291–297, 1994.

[10] Peter Sturm and Bill Triggs. A factorization based algorithm for multi-image projective structure and motion. In B. Buxton and Roberto Cipolla, editors, *Proceedings of the 4th European Conference on Computer Vision, Cambridge, England*, volume 1065 of *Lecture Notes in Computer Science*, pages 709–720. Sprin-ger-Ver-lag, April 1996.

[11] G. Veres, L. Gordon, J. Carter, and M. Nixon. What information is important in silhouette-based gait recognition. In *Proceedings of IEEE Computer Vision and Pattern Recognition conference*, 2004.

[12] D. K. Wagg and M. S. Nixon. On automated model-based extraction and analysis of gait. In *Proceedings of 6th International Conference on Automatic Face and Gesture Recognition*, pages 11–16, 2004.

[13] Bennett Wilburn, Neel Joshi, Vaibhav Vaish, Mark Levoy, and Mark Horowitz. High speed video using a dense camera array. In *Proceedings International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004.

[14] A. Williams. Requirements for automatic configuration of ip hosts. Technical report, Zeroconf.org, March 2002.

[15] Evan Woo, Bruce A. MacDonald, and Félix Trépanier. Distributed mobile robot application infrastructure. In *International Conference on Intelligent Robots and Systems (IROS)*, pages 1475–80, Las Vegas, October 2003.

[16] C. Zhang and T. Chen. A self-reconfigurable camera array. In *Eurographics Symposium on Rendering*, 2004.