Prop-Free Pointing Detection in Dynamic Cluttered Environments

Pyry Matikainen,† Padmanabhan Pillai,* Lily Mummert,* Rahul Sukthankar,*† Martial Hebert†
*Intel Labs Pittsburgh, †Robotics Institute, Carnegie Mellon University

***Abstract*—Vision-based prop-free pointing detection is challenging both from an algorithmic and a systems standpoint. From a computer vision perspective, accurately determining where multiple users are pointing is difficult in cluttered environments with dynamic scene content. Standard approaches relying on appearance models or background subtraction to segment users operate poorly in this domain. We propose a method that focuses on motion analysis to detect pointing gestures and robustly estimate the pointing direction. Our algorithm is self-initializing; as the user points, we analyze the observed motion from two cameras and infer rotation centers that best explain the observed motion. From these, we group pixel-level flow into dominant pointing vectors that each originate from a rotation center and merge across views to obtain 3D pointing vectors. However, our proposed algorithm is computationally expensive, posing systems challenges even with current computing infrastructure. We achieve interactive speeds by exploiting coarse-grained parallelization over a cluster of computers. In unconstrained environments, we obtain an average angular precision of 2.7°.**

I. INTRODUCTION

Pointing gestures are a natural modality for specifying locations or designating objects in the environment. Commodity systems now offer pointing interfaces that require users to employ props; however, prop-free, vision-based pointing remains an elusive goal. Prior work has shown that real-time pointing detection can be made robust for single users operating in controlled settings but such techniques fail in cluttered environments, particularly for multiple users in dynamic backgrounds.

This paper proposes a functional model for pointing that estimates the pointing direction without assuming a human kinematic model. Our algorithm analyzes scene motion and identifies arm lines corresponding to multiple users without requiring segmentation or background subtraction. The system is self-initializing; as users point, we analyze the optical flow and infer rotation centers that best explain the observed motion. From these, we group pixel-level flow into dominant pointing vectors that each originate from a rotation center.

While our algorithm is robust to visual clutter, it is also computationally expensive. Because it is intended for use in interactive applications, fast response times are essential [3]. To provide the needed responsiveness, we exploit the coarse-grained parallelism inherent in the algorithm to develop a distributed implementation that processes multiple high-resolution and high data rate video streams in real time. The resulting system, PinPoint, can be used as a real-time pointing interface in natural environments. We have developed several applications that use PinPoint such as those shown in Figure 1.

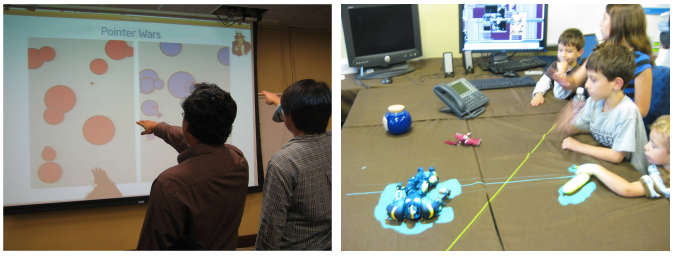

Fig. 1. PinPoint recovers 3D pointing directions of multiple users in a cluttered, dynamic scene, enabling a variety of applications. (Left) players pop balloons by pointing at them; (Right) A projector illuminates objects as users point to them.

The remainder of this paper is structured as follows. Section II surveys the related work in this area. Section III introduces a functional model for pointing and details key aspects of our algorithm including detection of rotation centers and robust recovery of the 3D pointing vector. Section IV describes how the proposed method is implemented using the Sprout distributed stream processing framework to achieve low-latency processing of video. Section V presents an experimental methodology and results that evaluate PinPoint in terms of seek time, pointing precision and latency. Finally, Section VI concludes the paper.

II. RELATED WORK

Pointing has been studied from an interface perspective for many years, primarily to analyze the effectiveness of systems employing laser pointers and similar props [17]. Thirty years ago, MIT's "put-that-there" system [2] combined a 6DOF (Polhemus) sensor mounted on the user's arm with a speech recognizer to allow a user to interact with a large display by pointing at specific objects while giving voice commands. Commodity technologies such as the Nintendo Wii Remote controller have popularized prop-based pointing interfaces. However, the goal of being able to point without props or markers, in an unstructured, crowded environment has remained elusive. Microsoft's Kinect (formerly Project Natal) [22] aims to offer prop-free gestural interfaces using a depth-based sensor in conjunction with strong priors on human body configurations. Bellucci et al. provide a recent survey of several touchless technologies that can be used in distant interaction [1].

Prop-free vision-based pointing detection falls into the category of computer vision for human-computer interaction, which has been an active research area for many years

(see [12] for a recent survey). The majority of systems that use gestures for human computer interaction focus on gesture recognition, where the goal is to determine whether a specified gesture was observed in the given video stream. In its simplest form, such systems can operate on an entire video clip and perform recognition without localizing the gesture in space or time. Under benign conditions, where the background does not generate distracting motion, such whole-clip gesture recognition can be used to drive user interfaces.

The primary difference between gesture recognition and pointing is that the former is a classification problem ("Did the user wave his hand?") while the latter is more akin to regression ("At what location on the table (x, y) is the user pointing?"). The remainder of this section discusses the latter.

Early work relevant to vision-based finger pointing focused on close-ups of the user's hand (e.g., [6], [10], [20]) against a clean workspace. Later, these ideas were extended to whole-body interfaces and typical approaches required segmenting the user from the background using techniques such as skin-color masks (e.g., [7], [26]), face detection (e.g., [19], [23]) or binary silhouette processing (e.g., [7], [14], [25]).

A small set of approaches performed pointing detection with a single camera (e.g., [4], [15], [21], [27]). These systems circumvented the underconstrained nature of the problem by relying on the rigid link kinematic constraints of the human body in conjunction with strong expectations of likely poses. However, due to the inaccuracy in estimating out-of-plane pointing gestures using a monocular camera, the majority of related work has employed multiple cameras, typically in completely calibrated configurations. PinPoint uses two (or more) cameras but requires calibration only in the final stage of processing.

Most of the earlier work is designed to work with only a single user. A few systems (e.g., [25], [28]) do track multiple users, but their approach requires a structured, static environment while our goal is specifically to detect pointing in cluttered scenes despite distracting background motion.

Existing approaches to pointing typically achieve real-time performance by employing efficient implementations of computationally-inexpensive algorithms, such as background subtraction, on single machines. However, a few recent systems utilizing large numbers of cameras (e.g., [25] with 12 feeds) do use clusters to distribute the workload. PinPoint differs from those in that we require a cluster not due to the number of feeds but rather due to the computational cost of motion analysis.

PinPoint is intended to allow users to point to arbitrary objects in 3D. In this respect, our work is closer in spirit to systems for human-robot interaction (e.g., [18], [21]) and intelligent rooms (e.g., [13], [28]) than systems designed as interfaces for 2D or multi-planar Cave displays.

Finally, Gavrila and Davis [11] and Mikic et al. [16] are examples of modeling and tracking humans using a full 3D multiple camera approach. Our work differs from these in that we neither use full 3D models nor aim to recover human kinematics. In summary, PinPoint is the first vision-based multi-user 3D pointing detection system designed specifically for cluttered environments, where reliable segmentation is not possible.

III. MOTION-BASED POINTING

PinPoint's physical setup consists of a pair of cameras typically mounted above the users and on either side of a large workspace. Each camera is extrinsically calibrated with respect to the workspace by placing fiducials at known location in the space. Unlike systems that operate on a stereo/disparity map, most of the processing in PinPoint involves 2D operations on individual camera streams with late merging to recover 3D vectors when necessary. This allows PinPoint to extend naturally (and efficiently) to configurations with three or more cameras, or to function in a monocular configuration when recovery of arm lines alone is sufficient.

Appearance-based approaches that seek to separate the user from the background fail in cluttered environments because of inaccuracies in segmentation. In contrast, we propose a functional model of pointing that focuses on the observed motion in the scene without segmentation. The basic idea is to examine the flow vectors in the video stream, determine which ones are associated with pointing actions and hypothesize arm lines in the image (rays emanating from a rotation center) for each such pointer. Given two or more camera views, we can then intersect the planes defined by these rays to recover a 3D vector corresponding to each pointer. We describe each of these steps in greater detail in the subsections below.

A. Functional model of pointing

A typical pointing gesture consists of a sweeping arm motion that ends with the limb extending towards the object of interest followed by finer motions that manipulate the object over the next several seconds (e.g., steering a cursor on a large display or refining a selection). Thus, our system's goals are two-fold: to robustly detect the occurrence of a pointing gesture in the middle of a crowded environment and then to accurately track where the user is pointing (i.e., recover an arm line for each user in each camera view).

Our model of pointing is functional in that it is agnostic to the appearance of the "arm" and to its actual kinematic structure. We assume that the observed motion generated by pointing gestures is instantaneously dominated by rotation about a single axis (typically about the user's shoulder), and that this rotation center moves slowly during the task. This allows our system to recognize pointing gestures performed by both adults and children, at any distance from the camera, or even pointing gestures made using objects such as a rolled-up magazine (where appearance-based techniques fail dramatically). In all of these cases, the lifting of a user's arm as it executes a pointing gesture can be modeled by a circular motion around the shoulder (here in a vertical plane). As discussed below, identifying a center of rotation is challenging for several reasons, including out-of-plane rotations (a circle in 3D appears as an ellipse), perspective distortion, distracting motion in the scene, and noisy estimates of optical flow.

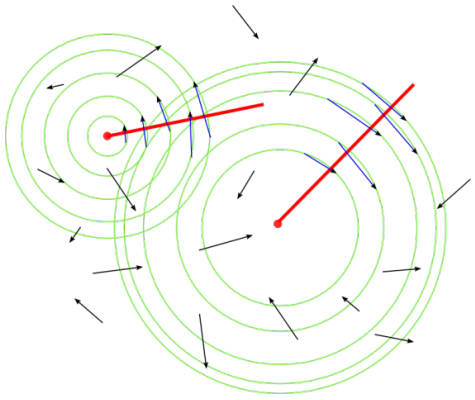

Fig. 2. Example of rotation centers and estimated pointer configurations recovered from a typical image. In this scene, the observed motion was best explained by two pointers; remaining moving points are treated as outliers.

Figures 2 and 3 show rotation centers extracted from noisy flow data and pointing models for each of two detected users superimposed over their camera images, respectively.

B. Identifying rotation centers

In the PinPoint system, localizing the presence of a new rotation center is equivalent to detecting a pointing event. Unlike earlier work, where users must explicitly initialize the system using a predetermined gesture, PinPoint is self-initializing and tracks new pointing events as they occur.

We employ the Kanade-Lucas-Tomasi (KLT) tracker [24] to recover a sparse sampling of optical flow in each image. A pointing gesture is detected whenever we observe significant motion that could be explained by a new center of rotation.

To describe our algorithm, let us consider the ideal case, where a single pointer rotates in a plane parallel to the image plane to generate circular arcs of motion in the image. These result in a set of features of the form $\{(x, y, f_x, f_y)\}$, denoting an optical flow vector $\mathbf{f} = (f_x, f_y)^\mathsf{T}$ observed at a location $\mathbf{x} = (x, y)^\mathsf{T}$ in the image. Clearly, the rotation center $\mathbf{r} = (x_r, y_r)^\mathsf{T}$ for the pointer is the point in the image at the intersection of the normals to these motion features — since the observed motion $\mathbf{f}$ is tangential to the radius $\mathbf{x} - \mathbf{r}$ from the center of rotation to the feature. This observation can be exploited to efficiently recover the rotation center using a straightforward variant of the Hough Transform [9]: each motion feature votes in 2D parameter space for those rotation centers located near a line normal to the local optical flow,

$$\left| \frac{(\mathbf{x} - \mathbf{r})^\mathsf{T} \mathbf{f}}{||\mathbf{x} - \mathbf{r}||\,||\mathbf{f}||} \right| < \epsilon. \tag{1}$$

By contrast, a traditional circular Hough transform would vote in 3D parameter space for all circles $(\mathbf{r}, \rho)$ passing through the feature location $\mathbf{x}$ irrespective of the local flow. With non-maxima suppression, the proposed method extends naturally to scenes containing multiple centers of rotation, and is reasonably robust to noisy features and distracting motion features caused by moving objects.

The pointing gestures observed by our system rarely occur in ideal settings, so we refine the approach outlined above in several ways. First, since optical flow estimates are notoriously noisy, rather than inferring rotation centers from instantaneous flow, we track features using KLT over a number of frames, k and analyze the resulting trajectories to retain only those that exhibit sufficient displacement: $||\sum_{i=t-k+1}^{t} \mathbf{x}_i|| > \theta$. A further refinement is to fit circles directly to such trajectories using linear least squares [8] and for each trajectory to cast its votes — as a Gaussian centered at the best fit (x_r, y_r) — rather than allowing each individual flow vector to independently vote on rotation centers. These votes persist from frame to frame but decay exponentially.

Second, we recognize that real-world motions are not perfectly circular and that out-of-plane circles are affected by perspective distortion. However, naïvely augmenting the rotation center algorithm to examine the space of all possible ellipses would result in an unacceptable number of spurious detections. We compromise by initializing on circular motion and subsequently fitting an ellipse model to motion tracked in subsequent frames. Specifically, each rotation center is parameterized by its location $\mathbf{r}$, eccentricity e and major axis direction $\hat{\mathbf{l}}$, all of which are estimated using the flow from tracked features, as described below.

Finally, since the intersections of normals from closely-located points are noisier than those from well-separated points, we avoid hypothesizing new rotation centers until their locations have been confirmed by observations from a wide range of angles (i.e., until the user's arm has swept a larger arc).

C. Estimating 2D pointer configuration

Each detected rotation center in an image is assumed to correspond to a pointer (user). Attempting to recover the 3D pointing vector from a single image is an underconstrained problem. Although suitable priors or depth information can enable single-view pointing (e.g., [4], [22], [27]), we choose to ignore the foreshortening information and focus on recovering only the 2D pointing line in the image since these can be merged with information from other camera views in a subsequent stage.

A traditional approach to recovering the pointing line would be to construct an appearance model for the pointing limb and to track this over time (e.g., [4]). Instead, because appearance models can be confounded by illumination, auto-gain control, clutter and occlusion, we focus exclusively on motion information. Specifically, we robustly estimate the 2D pointing angle using each motion feature that is consistent with the given rotation center as follows.

First, we assume that each tracked feature moves along the arc of an ellipse (approximating a circular arm motion under perspective). The trajectory corresponding to each feature votes in an accumulator space of ellipse parameters $(\mathbf{r}, e, \hat{\mathbf{l}})$. Let $\{(\mathbf{x}_k, \mathbf{f}_k)\}$ denote those N features consistent with a given rotation center. Next, we estimate the pointing line using a kernel density estimate in angle space. Specifically, let θ_k be

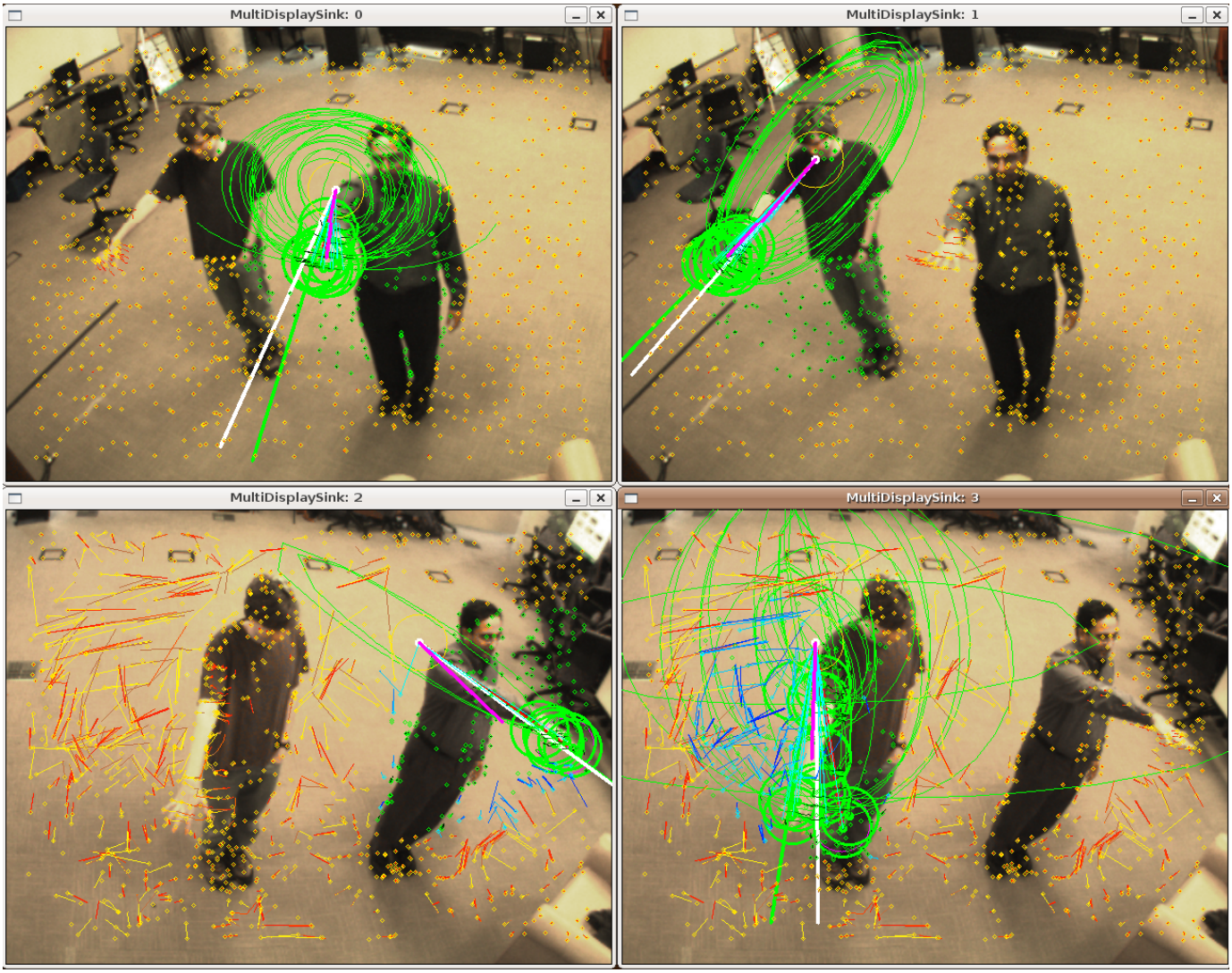

Fig. 3. Example of pointing detection and tracking. Display rows show different camera views; display columns show detection for the different player positions. Features returned by the KLT tracker are shown in green if corresponding to a player position, and gold otherwise. Feature trajectories are shown in shades of blue if corresponding to a player position, and shades of red to yellow otherwise. Pointing has been detected for both players. Rotation centers (shoulders) are shown with yellow circles, and pointer (arm) features are shown with green circles. Ellipses indicate the elliptical fit of feature trajectories with respect to the motion center. A large number of noisy feature trajectories are apparent in the lower display row; most of these are filtered and are not used to estimate pointer position. However, in the lower right view, the two large ellipses with a horizontal major axis are likely due to noisy feature trajectories. Pointer estimates are shown in several ways: green lines are based on current feature density, magenta lines are based on feature cluster centroids, and white lines show smoothed estimates.

the bearing from r to $\mathbf{x}_k$. The angle of the pointing line in the image is given by:

$$\hat{\theta} = \arg\max \sum_k N(\theta_k, \sigma^2), \tag{2}$$

where $N(\mu, \sigma^2)$ denotes a normal distribution with mean μ and variance σ^2. These pointing lines are shown as the red lines in the optical flow visualization (Figure 2) and magenta overlays in the camera views (Figure 3). Finally, we obtain a temporally-smoothed estimate, by applying an exponentially-weighted moving average to the pointing line measurements over time, resulting in the white lines in Figure 3.

D. Recovering 3D pointing vectors

As described above, we identify new rotation centers and the 2D configuration for each tracked pointer independently in each image. The next step is to integrate this information from multiple camera views to recover 3D vectors corresponding to each pointer in the scene.

To achieve this, we exploit several characteristics of PinPoint's physical setup. First, we employ a pair of cameras that can observe the scene from significantly different viewpoints; we calibrate them both with respect to each other and to the application environment. Second, we address the data association problem introduced by multiple users by exploiting knowledge of the scene geometry: e.g., the left pointer in one

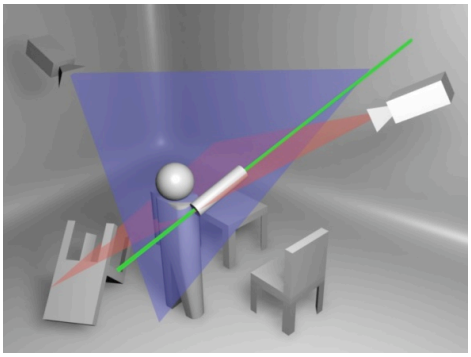

Fig. 4. Recovering the 3D pointing vector from multiple views.

camera view corresponds to the left pointer in the second view. Finally, we assume that the camera streams are sufficiently synchronized in time so that we can treat the two observations of the scene as simultaneous.

Given a pair of corresponding pointers in the two views, we recover the 3D pointing vector by intersecting them as follows. Given the extrinsic calibration for a camera, the estimated 2D pointer state uniquely defines a 3D plane in the environment, as illustrated by the red and blue triangles in Figure 4. These planes intersect at a line in the environment that corresponds to the 3D pointing vector.

E. Adding robustness

We enhance the method described above along several dimensions to add robustness:

- **Track filtering.** The raw output from the KLT tracker includes a significant number of noisy tracks generated by spurious features in addition to those generated by pointing gestures. PinPoint eliminates these on the basis of two criteria: (1) we reject tracks that are too short in length, since such non-discriminative arcs are consistent with almost any ellipse; (2) we filter out tracks that exhibit significant jitter (high average second derivative), since they are likely to have been generated by noisy features.
- **Rotation center migration.** To accommodate the user's natural motion during the pointing interaction, we allow the rotation center to migrate according to the observed motion in a local window around its detected center $\mathbf{r}$.
- **Priors on user location.** When available, we exploit prior knowledge of user locations. In particular, given the camera and workspace geometry, we eliminate regions in the image where users cannot stand and generate masks that prohibit rotation centers from being detected in those regions, resulting in increased robustness and better throughput.

IV. LOW-LATENCY IMPLEMENTATION

PinPoint is intended to be used in live, interactive applications rather than *post hoc* pointing detection in stored video. It is, therefore, necessary to have both high throughput to keep up with the video stream, and low end-to-end processing latency (100–200 ms) to ensure the system and any feedback to the user feels responsive [3]. Unfortunately, KLT tracking step alone on standard definition video takes over 250 ms per frame.

Rather than simplify the algorithm, or sacrifice resolution or fidelity to meet the response time goals, we instead developed a parallel, distributed implementation of PinPoint, using the Sprout runtime system [5]. In this section, we provide background on Sprout, and then describe the implementation details of PinPoint.

A. Sprout runtime system

Sprout is a runtime system that enables the creation of interactive perception applications by exploiting coarse-grained parallelism. It provides an easy-to-use framework for developing applications that can be distributed over a cluster of commodity multi-core servers while hiding much of the complexity of parallel and distributed programming.

Sprout represents applications as data flow graphs. This model is a good fit for the structure of PinPoint and other perception applications. The vertices of the graph are coarse-grained processing steps called *stages*, and the edges are *connectors* that represent data dependencies between stages. Sprout provides an easy-to-use API based on a set of C++ class hierarchies for defining application stages. Our code inherits from the base `Stage` class, specifies inputs and outputs, and extends the class with an `exec()` method. The implementation of this method is relatively unrestricted; it can be written as either parallel or sequential code, and can freely use external libraries, even non-thread-safe ones. Sprout performs all inter-stage communication transparently, including serialization of data structures, and automatically makes use of appropriate network and local transport mechanisms.

Data is delivered to a Sprout application by specialized stages called data *sources* and consumed by data *sinks*. Source stages provide the input data to the application, for example as a stream of video from a camera. A sink is simply a stage with no outputs in the data flow graph, although it may generate outputs to external components, such as a file or a display. Common sources and sinks that deal with video cameras, video files, and displays come predefined in Sprout.

Sprout lets one form parallel constructs in an application's data flow graph using *splitter* and *joiner* stages. A splitter divides or copies data. The number of outputs can be fixed (for task parallelism) or variable (for data or pipeline parallelism). Similarly, a joiner merges data, and can have fixed or variable inputs. Common splitters and joiners, such as copiers, image tilers, and round-robin splitters are predefined in Sprout. Many of these are implemented as templated classes, so can be readily adapted to operate on custom data types.

At run time, a management process called the *configuration server* performs the initial and ongoing configuration of an application based on a human-readable description of the application data flow graph. The configuration server launches a set of *stage server* processes on a set of compute nodes.

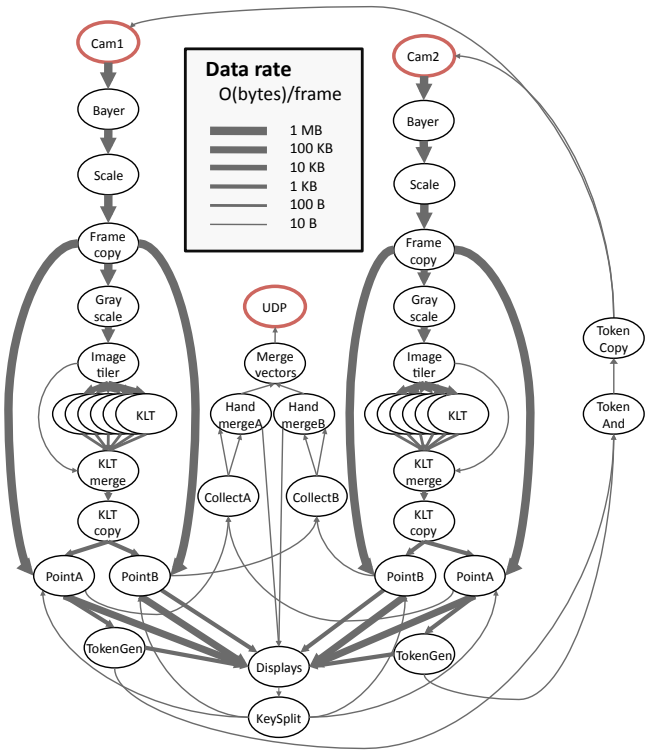

Fig. 5. Application graph for a two camera, two player game using PinPoint. Nodes are application stages, and edges are data dependencies. Edge widths indicate the amount of data transmitted between stages. Sources and sinks are outlined in red.

Stage servers are container processes for application stages, each of which runs on a separate thread. The configuration server activates stages dynamically as needed among the set of stage server processes for the application.

B. PinPoint implementation

Figure 5 shows the data flow for PinPoint used as an interface to the two-player game shown in Figure 1. Data sources Cam1 and Cam2 are read according to a token-based scheme that controls the number of video frames in the processing pipeline. Data is read from the sources when a token is available. Our implementation uses two token generators, one per camera, and synchronizes the sources using the TokenAnd and TokenCopy stages. The UDP sink is a generic adapter that sends data to a UDP port, which in this case is a control input to the game.

Raw frames are captured from the cameras, interpolated using a Bayer filter, and optionally reduced in size. Features are extracted from gray-scaled frames using a KLT tracker. Because feature extraction is the most computationally expensive stage in the application, the frame is tiled into six overlapping subregions, and up to 200 features are extracted from each subregion in parallel.

The stages PointA and PointB identify rotation centers and estimate pointer configurations, if any, for each player position. These stages maintain the trajectory of each feature. Rotation

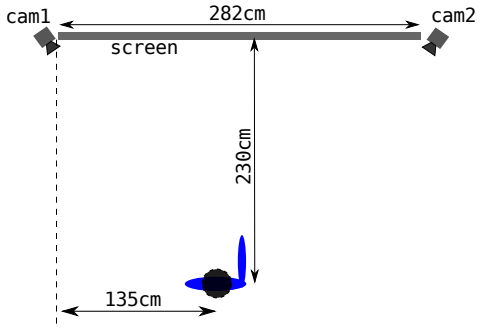

Fig. 6. Top down schematic of experimental setup. Player stands approximately centered in front of a large projected display, with two cameras positioned on the upper corners of the screen.

centers are determined by fitting circles to trajectories using linear least squares [8]. To eliminate noise, only trajectories with sufficient history, length, and consistency of motion are used. A voting scheme based on Gaussian distributions about each center is used to select the most likely location. The features are then clustered about their nearest centers. To find the pointer configuration and account for out-of-plane rotation, ellipses are fitted to trajectories of sufficient length and consistency of rotational direction. The pointing angle is estimated from the density of features about the rotation center. Once identified, features corresponding to the pointer are tracked explicitly from frame to frame. Rotation centers are tracked using features in their regions. Features along the pointer are tracked using local motion, excluding duplicates and those that appear stationary. The stage output consists of two points: the location of the rotation center and and a position along the pointer that represents the mode of the density estimate.

The final stages collect the pointer configurations from each view, and calculate the 3D pointing vector for each player position. As described in Section III-D, the pointing line in each view defines a 3D plane, and the intersection of the two planes corresponds to the 3D pointing vector. The intersection of each pointing vector and the destination surface is then calculated, and these points are communicated to the display application through the UDP sink. Various stages send data to the Displays stage, which shows camera views and diagnostic images.

V. EVALUATION

In this section, we evaluate the implementation of PinPoint described in Section IV. We examine performance of both the pointing algorithm and the underlying runtime system.

A. Experimental environment

Our experiments are performed using the setup illustrated in Figure 6. We use two Point Grey Research Firefly MV 03MTC cameras mounted on the top corners of a 282×200 cm vertically-mounted projected display. The display is 1024×768 resolution, while the cameras operate at 640×480. The user

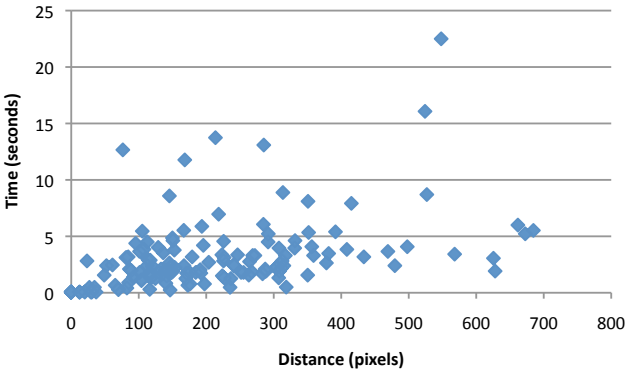

Fig. 7. Seek time vs. distance for target radius of 80.

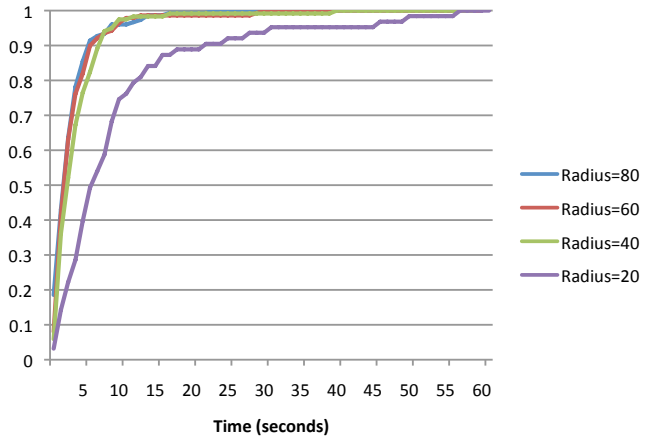

Fig. 8. CDF of seek time for different target sizes.

stands approximately centered in front of the screen and points to projected targets of varying sizes.

The PinPoint code runs on a cluster of five servers running Ubuntu Linux 7.04, connected via a 1 Gbps Ethernet switch. Two of the servers act as front-end machines. An Intel® Core™ 2 2.66 GHz CPU 6700 (dual core) with 2 GB of memory runs one branch of the application data flow shown in Figure 5 from the camera source through pointing, excluding KLT feature extraction. An Intel® Core™ 2 Extreme 3 GHz CPU 9650 (quad core) with 4 GB of memory runs the remaining stages, excluding KLT feature extraction. Three Intel® Core™ i7 2.93 GHz CPU 940 (quad core) with 6 GB of memory run the KLT feature extraction stages.

B. Seek time

Measuring the accuracy of pointing detection is an ill-defined task, since the direction a human user points is itself subject to interpretation. Furthermore, in an interactive setting with feedback (e.g., a cursor marking where the system believes the user is pointing), the user quickly and subconsciously corrects for static errors (bias). Therefore, rather than directly measuring accuracy, we determine how effectively a user can control a pointing-based interface employing PinPoint.

In particular, we seek to determine how quickly the system and user can interactively move a cursor to a desired position by pointing. We collected traces of three users performing a targeting task using PinPoint. A circular target was shown on the display at a randomly selected position, and the user was to guide the cursor within the boundary of the target by pointing. We refer to the time between the appearance of the target and the registration of pointing within the boundary of the target as *seek time*. Figure 7 shows seek time for a target with a radius of 80 pixels on a display area of 1024×768. The graph shows that in most cases, the system registered pointing at the target within a few seconds. Seek time generally increases with the distance between the target and the initial position of the cursor.

Two major effects contribute to some of the long seek times. First, some spurious feature trajectories generated by noise in the camera images (as seen in the lower images of Figure 3) do make it past our filtering steps. If these noisy trajectories happen to fit ellipses centered at the shoulder, then they cause the detected pointing direction to jump erratically. Low pass filtering helps mitigate the problem, as these transient data points are greatly outweighed by the true feature trajectories on the user's arm. Secondly, the system's tracking ability varies greatly for different directions of pointing. When the target appears near the upper corners of the screen and the user is pointing almost directly into one of the cameras, ellipse fitting for that view is very poor. Therefore, there are few good data points to counterbalance the noisy data points, and the random perturbations of noisy feature trajectories are effectively magnified when the target is in such regions.

C. Pointing precision

To evaluate pointing precision, we expanded the targeting task in the previous section to include targets of different size. Figure 8 shows the cumulative distribution of seek time for four target sizes. The graph shows that as the target size shrinks, the users needed more time to accurately position the cursor, although the effect was minor until the target area fell below one quarter of its original size. The very consistent behavior at the larger target sizes and the sudden increase in seek time for the smallest size indicates that the system as configured for our experiments has an effective precision limit between 20 and 40 pixels. Although it is possible to control the cursor with greater precision, the user effort and time increases significantly. With the given screen size and user position, the upper bound of the pixel precision range corresponds to an angular precision of 2.7°.

D. System performance

We measured the latency and frame rate of PinPoint through the token generation loop shown in Figure 5. Our implementation sustained an average of 13 frames per second with a frame latency of 195±45 ms. Table I shows the latency of individual stages that averaged at least 1 ms. The primary contributors

TABLE I
STAGE LATENCY (MS)

Stage	Average	St. dev.
Camera source	7	9
Bayer	4	2
Scale	6	1
Frame copy	7	3
Gray scale	3	2
Image tiler	1	<1
KLT	52	16
KLT merge	1	<1
KLT copy	2	<1
Point	12	4
Displays	28	23

to end-to-end latency are the KLT tracker and pointing stages. The high standard deviation of the KLT tracker is due to the uneven distribution of features in image tiles. In practice, the KLT tracker will contribute more to the end-to-end latency than shown by the average in Table I, because the latency of the KLT trackers as a group is determined by the slowest instance. Performance could be improved by increasing the number of tiles. The display stage, while relatively slow, is not in the critical processing path.

VI. CONCLUSION

Pointing is a natural modality for user interaction. In this paper, we propose PinPoint, a system for multi-user, prop-free pointing detection that uses a novel analysis of motion features in conjunction with a low-latency distributed computing framework to achieve real-time performance.

This paper makes two key contributions. First, we propose a functional model of pointing that requires no explicit human model, and robustly detects pointing without background subtraction or appearance models. The model accommodates multiple users in a cluttered, dynamic environment and achieves an average angular precision of 2.7°. Second, we show that through judicious parallelization, the computation in the proposed algorithm can be distributed over a cluster of machines to achieve real-time processing of video streams while maintaining low latency.

With the recent release of the Microsoft Kinect heralding the arrival of cheap depth cameras, a natural extension of the method is to take advantage of depth information. The most basic use of depth information would be to use it to perform background subtraction, thereby reducing noise from background movement. In principle our method extends easily to three-dimensional trajectories, as the process of circle fitting can assume true circles in three dimensions since there is no warping due to perspective. However, in practice the main obstacle to this extension is the difficulty of producing accurately tracked 3D trajectories from the coarse and incomplete depth information produced by a consumer grade depth sensor such as the Kinect. Nevertheless, we believe that with a depth-aware tracking algorithm, reasonable 3D trajectories can still be extracted, and this is something we will explore in future work.

REFERENCES

[1] A. Bellucci, A. Malizia, P. Diaz, and I. Aedo, "Human-display interaction technology: Emerging remote interfaces for pervasive display environments," *IEEE Pervasive Computing*, vol. 9, no. 2, 2010.

[2] R. Bolt, "'Put-that-there': Voice and gesture at the graphics interface," in *Conference on Computer Graphics and Interactive Techniques*, 1980.

[3] C. M. Brown, *Human-Computer Interface Design Guidelines*. Intellect Books, 1999.

[4] D. Bullock and J. Zelek, "Towards real-time 3-D monocular visual tracking of human limbs in unconstrained environments," *Real-Time Imaging*, vol. 11, no. 4, 2005.

[5] M.-y. Chen, L. Mummert, P. Pillai, A. Hauptmann, and R. Sukthankar, "Exploiting multi-level parallelism for low-latency activity recognition in streaming video," in *MMSys '10: Proceedings of the First Annual ACM SIGMM Conference on Multimedia Systems*, 2010.

[6] R. Cipolla, P. Hadfield, and N. Hollinghurst, "Uncalibrated stereo vision with pointing for a man-machine interface," in *IAPR Workshop on Machine Vision Applications*, 1994.

[7] C. Colombo, A. Del Bimbo, and A. Valli, "Visual capture and understanding of hand pointing actions in a 3-D environment," *IEEE Trans. SMC-B*, vol. 33, no. 4, 2003.

[8] I. D. Coope, "Circle fitting by linear and nonlinear least squares," *Journal of Optimization Theory and Applications*, vol. 76, no. 2, 1993.

[9] R. Duda and P. Hart, "Use of the Hough Transformation to detect lines and curves in pictures," *CACM*, vol. 15, no. 1, 1972.

[10] M. Fukumoto, Y. Suenaga, and K. Mase, "'Finger-Pointer': Pointing interface by image processing," *Computers and Graphics*, vol. 18, no. 5, pp. 633–642, 1994.

[11] D. Gavrila and L. Davis, "3D model-based tracking of humans in action: a multiview approach," in *CVPR*, 1996.

[12] A. Jaimes and N. Sebe, "Multimodal human-computer interaction: a survey," *Computer Vision and Image Understanding*, vol. 108, no. 1–2, 2007, special issue on vision for human-computer interaction.

[13] N. Jojic, B. Brumitt, B. Meyers, S. Harris, and T. Huang, "Detection and estimation of pointing gestures in dense disparity maps," in *FG*, 2000.

[14] R. Kehl and L. Van Gool, "Real-time pointing gesture recognition for an immersive environment," in *FG*, 2004.

[15] M. Kolesnik and T. Kulessa, "Detecting, tracking, and interpretation of a pointing gesture by an overhead view camera," in *Pattern Recognition*, ser. Lecture Notes in Computer Science, 2001, vol. 2191, pp. 429–436.

[16] I. Mikic, M. Trivedi, E. Hunter, and P. Cosman, "Human body model acquisition and tracking using voxel data," *IJCV*, vol. 53, no. 3, 2003.

[17] B. Myers, R. Bhatnagar, J. Nichols, C. Peck, D. Kong, R. Miller, and C. Long, "Interacting at a distance: measuring the performance of laser pointers and other devices," in *Proceedings of CHI*, 2002.

[18] K. Nickel and R. Stiefelhagen, "Visual recognition of pointing gestures for human-robot interaction," *Image and Vision Computing*, vol. 25, no. 12, 2007.

[19] C.-B. Park, M.-C. Roh, and S.-W. Lee, "Real-time 3d pointing gesture recognition in mobile space," in *FG*, 2008.

[20] V. I. Pavlović, R. Sharma, and T. S. Huang, "Gestural interface to a visual computing environment for molecular biologists," in *FG*, 1996.

[21] J. Richarz, C. Martin, A. Scheidig, and H. Gross, "There you go! - estimating pointing gestures in monocular images for mobile robot instruction," in *IEEE International Symposium on Robot and Human Interactive Communication*, 2006.

[22] D. Rowan, "Kinect for Xbox 360: The inside story of Microsoft's secret 'project natal'," November 2010.

[23] J. E. V. Sébastien Carbini and O. Bernier, "Pointing gesture visual recognition by body feature detection and tracking," in *International Conference on Computer Vision and Graphics*, 2004.

[24] J. Shi and C. Tomasi, "Good features to track," in *CVPR*, 1994.

[25] A. Sridhar and A. Sowmya, "Multiple camera, multiple person tracking with pointing gesture recognition in immersive environments," in *Advances in Visual Computing*, ser. Lecture Notes in Computer Science, 2008, vol. 5358, pp. 508–519.

[26] H. Watanabe, H. Hongo, M. Yasumoto, and K. Yamamoto, "Detection and estimation of omni-directional pointing gestures using multiple cameras," in *IAPR Workshop on Machine Vision Applications*, 2000.

[27] A. Wu, M. Shah, and N. da Vitoria Lobo, "A virtual 3d blackboard: 3d finger tracking using a single camera," in *FG*, 2000.

[28] Y. Yamamoto, I. Yoda, and K. Sakaue, "Arm-pointing gesture interface using surrounded stereo cameras system," in *ICPR*, 2004.