An Efficient Representation for Multi-Application Accessible Visual Information

by

Alton Jerome McFarland

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

~~August 9, 2002~~
[September 2002]

Author____
Department of Electrical Engineering and Computer Science
August 9, 2002

Certified by [illegible]
Boris Katz
Thesis Supervisor

Accepted by [illegible]
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

An Efficient Representation for Multi-Application Accessible Visual Information

by

Alton Jerome McFarland

Submitted to
the Department of Electrical Engineering and Computer Science

August 9, 2002

in Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

As computers become more and more pervasive in our society, one of the problems that arises is how to efficiently integrate different types of systems and the information that they collect. Through cooperation between multi-modal systems, a much broader range of applications become available. In an attempt to attack this problem on a small scale, this project demonstrates the coupling of a natural language processing system to a camera-based person tracker for the purpose of being able to ask informally worded questions about what the tracker has observed in the MIT Intelligent Room.

Thesis Supervisor: Boris Katz
Title: Principal Research Scientist, MIT Artificial Intelligence Laboratory

Acknowledgements

I would also like to thank David Demirdjian, Neal Checka, and Kevin Wilson for their help with the Vision Interfaces Group's person tracker.

I would especially like to thank Boris Katz and the entire InfoLab Group for their help and support with my thesis (and in general). Special thanks go to Jimmy Lin for helping me get started with the GUI code and to Luciano Castagnola, Ali Ibrahim, and Jacqueline Lai for being nice enough to traipse back and forth throughout the Intelligent Room when I needed help testing the system.

Table of Contents

1 Introduction

The problem of mapping the complexities of human perception into useful, quantifiable data is an issue that continues to intrigue the artificial intelligence community. For instance, the seemingly simple task of determining who, if anyone, is in a room at a given time becomes significantly more complicated when viewed from a computer's perspective. First, the ability to process visual information is necessary. Distinguishing where a person ends and a wall begins, while trivial for a human being, is not an easy determination for a computer program to make. And even when such determinations can be made, the passage of time introduces a daunting element of instability, as the positions and orientations of objects in the room constantly change. In order to gain as much information as possible from such a system, some concept of temporal state must be maintained. The cameras can provide the static representation of a particular scene, but in order to answer questions like "What is George doing?", not only must the system be able to determine that the blob of pixels in the center of the room is George; it must also be able to decide from the correlation of temporal data relating to the scene, that George's recent actions indicate that he is using the telephone.

Another advantage of storing temporal data is the gained ability to perform post-processing on that data. The aforementioned problem of object-tracking is a very difficult one. The determination of an object's actual motion can be made more accurate by taking into account its previous motion and the motion of the objects around it. For instance, the data might show Blob 0 moving steadily from position (0,0) to position (0, 2) and then suddenly disappearing just as Blob 1 appears at position (0,2) and continues moving in the same direction that Blob 0 had been going. In such an instance, it isn't likely that Blob 1 suddenly appeared in the room at that position just as

Blob 0 exited (perhaps through changes in position that occurred faster than the person-tracker could update). It is much more reasonable to assume that the object was simply lost for a moment and was reacquired, and subsequently, renamed. Post-processing of the data can reveal such glitches and smooth them out by equating the concerned objects. In this case, after the post-processing had occurred, Blob 1's information would ultimately be stored as additional information for Blob 0. Here the concept of time can pay great dividends, for only through the examination of ranges of data could such decisions be made.

In a joint project between my own group, the InfoLab Group, and the Vision Interfaces Group, I implemented a system for the maintenance and dynamic processing of such temporal data called the Multi-application Accessible Visual Information Storage system (or MAVIS). Positional information from two cameras located in the Intelligent Room at the AI Lab is coordinated by the Vision Interfaces group's person-tracking system [1] and output continuously. That visual data was dynamically stored in a database to which START [3,4], our natural-language processing system, had access. START allows users to ask questions in natural language. Numerous functions were written to allow the system to make objective determinations (i.e., "Where is George?"). Eventually, that objective knowledge (time, position, etc.) will enable MAVIS to make subjective determinations like those previously mentioned (i.e., "What is George doing?"). Stated simply, the ultimate aim was to dynamically take input from cameras in the Intelligent Room and to effectively represent the transient nature of that input, thereby allowing useful inferences to be made as to what real-world actions the data corresponds to.

MAVIS tries to integrate the abilities of START and the person-tracker in a way that allows for distributed access by other programs. By creating a simple, easily-accessible representation for the state of the Intelligent Room, I hope to make it easier for more complicated interactions to occur.

2 Related Work

As mentioned previously, many researchers are interested in marrying the capabilities of differing human-centric systems to attain new and interesting functionality. Over time it has become apparent that one of the most difficult aspects of that endeavor is how to effectively share information between those systems. One approach that has gained significant popularity is the "blackboard model" [7]. In a blackboard architecture the shared information, or "problem state", is stored in a single location. Accesses to that location are the only means by which the distributed systems interact. Information can be added, deleted, or modified by those systems, so a means of controlling access is necessary.

Koelma and Smeulders [6] proposed an infrastructure for image interpretation that utilizes an object-based blackboard architecture. An implementation of their system would utilize blackboard "levels" to allow for ordering of data on the blackboard. Those levels could be distinguished by level of abstraction, data type, or whatever fits the needs of the particular application.

The Intelligent Room has its own agent-based means of communicating between its components [2]. The infrastructure of the Room itself is composed of user and multi-modal interfaces and systems for context management and resource management. Context management refers to the maintenance of information about an entity's "situation" (i.e., its current characteristics). Context management plays a large role in design and operation of the Intelligent Room since proper responses to user needs and input require contextual information. Meeting those needs is at the heart of the Intelligent Room's purpose and is behind most of the work on Human-Centered Interaction (HCI). Resource management is performed by a system called Rascal, which is responsible for providing a buffer between the demands of individual applications and the functionality of the workspace as a whole. Without such a buffer, uncontrolled contention for resources could quickly cause significant problems.

The goals of the Stanford project on Interactive Workspaces [8] are very similar to those of the Intelligent Room project. They also hope to enhance HCI through research involving the integration of multiple devices and applications in a single room. As in the Intelligent Room, the Stanford project attempts to incorporate context-based interpretation in its design. Another key focus of their proposed architecture is the development of separately maintained action-perception couplings. Action-perception couplings, as the name implies, are the relationships between actions and perceptions. For instance, the perception that a remote-controlled car is responding smoothly to your manipulation, is an example of an action-perception coupling. The action is what you actually do with the remote and the perception is how you see the car responding. Separating the maintenance of those couplings allows for multiple couplings to be simultaneously maintained (i.e.,on separate processors).

MAVIS, while meant to be extensible, is intended to serve a more specific purpose than the ones discussed above. My focus was mainly to achieve a coordination of vision and language to an extent that the Intelligent Room itself could be made to appear sentient when naturally queried about what is going on inside it.

3 Background

3.1 Person Tracking System

The Vision Interfaces Group's person-tracking system [1] was used to generate the visual data necessary to answer useful queries. In their system, information from multiple cameras, positioned at different angles, is coordinated through a central server before finally being output to external applications (i.e., my system). Their system uses learned knowledge of a particular static background (in this case, the Intelligent Room) to aid in the determination of object position. Having an idea of the background allows the tracking system to more accurately determine which elements of a particular camera image are in the foreground (e.g., do not correspond to the learned background). Contiguous clusters of such foreground points are detected dynamically and labeled as blobs (i.e., people).

3.1.1 Foreground Detection

To allow for rapid and efficient object tracking, rapid detection of a frame's foreground is necessary. The person-tracking system does this by computing differences between the pixels in the current image and the learned background image. If the values at particular pixel place it at a more shallow depth than the corresponding pixel in the learned background image, the pixel is labeled as foreground. [1]

Once the system has decided which pixels are in the foreground and which are in the background, the trajectories of the tracked objects must be determined. This determination is made through the maintenance of quality-weighted possible trajectories. These trajectories are compared with the pixel foreground information of each successive frame and reevaluated to determine the most accurate description of the object's motion. [1]

3.1.2 Camera Server

All of the aforementioned calculations take place for each camera in the Intelligent Room. The information from those cameras is then sent to a server where it can be coordinated. There, information acquired from differing camera angles can be correlated, yielding a much more accurate and precise idea of what's going on in the room. It is the server's processed determination of motion in the room (based on the data from multiple cameras) that is finally output for use by systems such as MAVIS. [1]

Specifically, the person-tracker assigns a unique number (starting with 0) to each blob it detects in the room. If Blob 0 already exists in the room, the next blob will be named Blob 1, then Blob 2, etc. The positional information about each blob, coordinated from each camera, is output as a tab-separated string containing the object's x-position, y-position, and z-position. The x-position is measured laterally outward from a plane passing through the center (hence it can be positive or negative) of the room's far wall (as viewed from the main entrance to the Intelligent Room), and the y-position is the straight-line distance from any point in the room to that far wall. So the point (0,0), in the person-tracker's coordinate system would correspond to a point in the center of the far wall, (-3,0) would be the far right corner, (3,0), would be the far left corner, and so on. The z-position (or blob "height") is the blob's height above an imaginary horizontal plane in the room, located well above the floor. The reason for taking measurements from such a reference point is that moveable objects in the room (e.g., chairs) clutter the lower regions of the images. Taking measurements directly from the floor would also necessitate telling the difference between a rolling chair and a moving person. Therefore, only objects whose heights exceed a certain value are tracked.

Information about the positions of all blobs in the room is sent out continuously at a rate that is usually around 12Hz. If there are two blobs detected in the room (for instance, blobs "0" and "1") the system will repeatedly output a line about Blob 0 and a line about Blob 1. If at any point, Blob 0 disappeared (from having left the room or being lost by the camera) the system will simply begin sending continuous updates about Blob 1 alone.

3.2 START

START [3,4] is a Natural Language Processing System created by Boris Katz and used by the InfoLab Group of the MIT Artificial Intelligence Laboratory. START parses naturally formulated (i.e., normal English) questions and statements into an efficient representation and allows for the construction of similarly natural answers and responses.

3.2.1 T-expressions

Ternary expressions (T-expressions) are the most basic form of knowledge representation within START. T-expressions store information in a directed subject-relation-object format that allows for easy retrieval and search. When posed with a query, START translates the input into appropriate ternary expressions. For instance, the query "Does Jim own a car?" would be parsed into the T-expression (Jim - own - car). Those expressions would then be matched against the information currently stored in START's knowledge base. If the statement "Jim owns a red car." had been entered previously, thereby creating the T-expressions (Jim - own - car) and (car - is -

red), the query would find a positive match and START would formulate an answer such as "Yes. Jim owns a red car." Since T-expressions are directed, however, the query "Does a car own Jim?" would not find a match and START would respond with an "I don't know." answer, indicating that it has no knowledge to support such a ludicrous assertion.

Through the creation of an extensive knowledge base of stored T-expressions, START is able to answer questions about a wide range of topics. The only requirements for matching are that the formulated query follows grammatical rules and that the information contained in a previously stored T-expression corresponds to the desired information. Once a match has been achieved, START has several methods of actually answering the posed question. Some information is stored natively in START, some might be retrieved from external databases, and some might even be pulled live from the Web. [5]

3.2.2 Native Linguistic Capabilities

In addition to their usefulness as an input interface for information retrieval, START's natural language processing capabilities also allow an underlying system to output the relevant information in a user-friendly format. Questions like "Is John in the room?" can be answered with natural responses like "Yes, John is in the room. *He* has been in the room for 2 hours." As that example demonstrates, START has the ability to perform informed lower-level language processing in addition to simply wrapping the answer in natural text. Since START knows that "John" is a masculine name, it is able to use the pronoun "He" to refer to "John" if so desired. This is necessary as the use of "John" over and over in every sentence, would quickly become repetitive and sound very unnatural.

In addition to knowledge about names' gender, START has several other types of native knowledge that allow it to process language more effectively. Parts of speech, pluralizations, synonyms and other valuable grammatical information are also able to be stored within START. These useful bits of word-level information allow START to better understand the true meaning of a sentence/question and to construct appropriate and variable responses.

3.2.3 Knowledge Sources

For some queries, the best strategy is for START to store knowledge natively for use in question answering. Storing the parsed representation of "Jim owns a red car." allows for the answering of several different questions about that statement. To facilitate answering of even more queries, START uses a system for outside information retrieval called Omnibase. [5]

The core of Omnibase consists of scripts that, when given an appropriate input, can independently (of START) retrieve information. These scripts are separated into classes, so that, based on the type of question posed to START, an appropriate script may be chosen to find the answer. For instance, when asked "Who directed Gone with the Wind?", START would, knowing that 'Gone with the Wind' is an IMDB (Internet Movie DataBase) movie, access the DIRECTOR script for the class imdb-movie and pass it the movie title as input. In this particular case, the script would go to the IMDB webpage for Gone with the Wind, and parse out the relevant information from the HTML. START would then take that information and wrap it in a natural sounding sentence:

"Gone with the Wind (1939) was directed by George Cukor, Victor Fleming, and Sam Wood."

Other scripts might simply return a piece of HTML that was cut directly from the page. The level of complexity involved in generating the answer format is completely variable. The constant in the Omnibase-START relationship is that START deals with language while Omnibase deals with information retrieval. However, for START to know which things Omnibase has information about, there has to be a coupling between the two systems. That coupling takes the form of Omnibase symbols. Stored in Omnibase, these symbols ("Gone With the Wind", "Tom Cruise", "France", etc.) constitute the set of things (movies, people, countries, etc.) that Omnibase has information about. START has access to those symbols and when parsing queries, it tries to determine whether or not a question is being asked about one of those symbols. For instance, it would be silly to load the names of all movies into START, a language processing engine. However, for START to efficiently determine that "Gone with the Wind" is a single entity about which the user is asking a question, those symbols need to be stored somewhere. The logical place to store those names is in Omnibase, an information retrieval engine. In essence, Omnibase is like an encyclopedia in that it is a gateway to large amounts of information. START simply has access to the index to that encyclopedia.

3.2.4 Annotations

The T-expressions that match a particular query are generated by system inputs called annotations. Annotations are a way to encase the relationships of T-expressions in a more intuitive structure. An annotation could look something like "us-state's flag". After parsing these annotations, START stores the generated T-expressions with pointers back to the original information segment. The aforementioned annotation implies that there is a possessive relationship between the symbol "us-state" and the noun "flag". So if the question "Does us-state have a flag?" were asked, START would give a positive response. In practice, though, "us-state" will be a variable that maps to the names of any of the 50 US states. So START would be able to confirm that Florida, Texas, Utah, etc., all have flags. Synonymy/hyponymy, ontologies, and structural transformation rules are all utilized when trying to match user queries to stored T-expressions. Through its use of annotations, START is able to bridge the gap between language and understanding. [4]

4 System Design

4.1 Design Considerations

4.1.1 Ease of Access

One of the significant problems that arises when dealing with large amounts of data, such as the data that comes from the Intelligent Room's cameras, is how to efficiently store that data while allowing for easy access. One of the project's goals is to create a sort of "blackboard" from which multiple distributed systems could read. Though use of visual data is the primary focus here, any number of applications could potentially benefit from such information. For instance: a graphical user interface could show the current (or past) activity in the room, an audio system might utilize the stored positional information to better determine who was speaking at a given moment, and, of course, a natural language processing system could use the data to answer questions about things that happen in the room. Since the cameras are constantly outputting positional data, current information will be constantly changing. That volatility requires that there be a way to retrieve static information, so that numbers aren't changing while they're being read. Essentially, changes in room state must be atomic so that readers can trust that the information they're receiving is valid and complete, even if that means the data may be a few milliseconds behind the most recent data. In addition to designating some of the data as the room's "current state", the sheer volume of the total input must be dealt with. With the person-tracking server sending updates several times a second for multiple blobs, the amount of stored information will get very large, very quickly. It seemed desirable to minimize the amount of data stored, while retaining the pertinent overall information, to save space, save processing time, and avoid redundancy as much as possible.

4.1.2 Accuracy

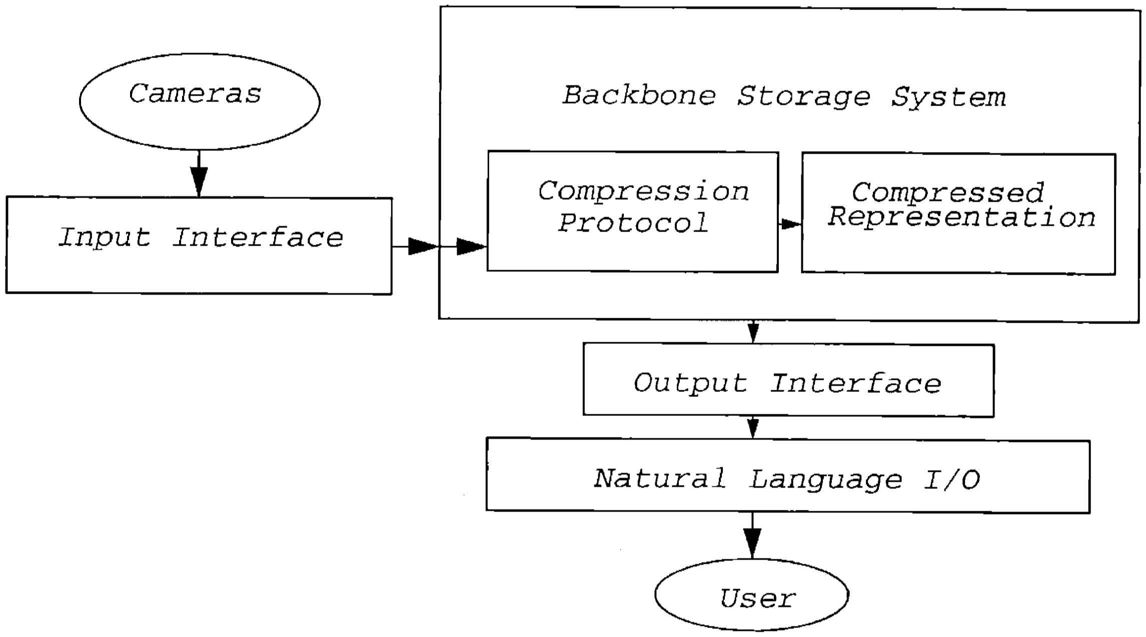
Another concern that needed to be addressed was the accuracy of the information received from the person-tracker. Sometimes non-existent people, or artifacts, are detected and positional updates about them are output erroneously. The occasional appearance of such artifacts, coupled with the fact that, on occasion, the tracker can lose a person for a period of time, can cause significant problems. When the tracker loses a person, aside from the obvious issue of not having current information for a particular person in the room, identification errors can occur. Upon reacquisition of a temporarily lost person, the system sometimes mislabels that person. Sometimes it might decide that it has detected a new, different person. For instance, assume there are two people in the room, labeled "1" and "0". If the tracker loses Person 1 just as an artifact is erroneously detected, the artifact will be labeled "1" and the second person, upon reacquisition, will be labeled "2". These sorts of glitches can cause serious problems when trying to use the stored data to determine what was happening in the room. As Person 1 is mistakenly changed to Person 2, aside from the mysterious appearance of a third person, the corresponding data would suggest that Person 1 made a sudden and unexpected jump in position. Also, from that point on, the information that should have been associated with Person 1 will now be associated with Person 2. This sort of error makes it difficult to draw conclusions based on a person's actions over time. The data associated with Person 1 would, at some point, become useless as it begins to correspond to an artifact. It is impossible to tell that such a thing has occurred when looking at each bit of data separately. In order to catch such errors, the decision was made to try and build in an expectation of what future data for a particular person should look like, based on what data had already been received.

To discern what the data actually represented, it became clear that some sort of "replay" feature would be needed. Live data could be simulated easily enough by writing programs that simply pretend to be the person-tracking server and send updates to my system. What was most important, though, was to be able to select portions of real data and to rerun them and see (graphically) exactly what sort of room activity they correspond to. This, of course, should be able to be done without the added complexity involved in checking the data for changes, checking it for accuracy, or storing it for later use.

Also, for the system itself to be practical it seemed clear that it was important to detect when a person had left the room. This required both a mechanism for determining whether or not a person who had entered the room had left and a means for making the person's exit clear in the stored data.

4.2 System Architecture

To effectively coordinate and execute the various tasks that I felt were integral for useful operation, I originally envisioned a system with five main components.



1. *backbone storage system* - the data from the cameras (and any other input devices) goes here.

2. *input interface* - it would be impractical to store the raw data "as-is" for any long period of time. Such storage would quickly become unmanageable. The cameras in the Intelligent Room output x-y position information at a rate of around 12Hz, but not all of that information needs to be stored permanently.

3. *compression protocol* - Since permanent storage of the raw visual data is undesirable, some other processing has to be done to avoid wasting time and space. The stored data will need to be compressed into a more efficient representation. For the types of inferences I hoped to make, I felt that the important features of the data could be captured by representing changes in the information and keeping track of the times that those changes occur. So, if the system's interface with the

camera had some idea of the "current room state", it need only modify its representation when that state changes.

4. *compressed representation* - the data will be stored in a compressed format to save space and processing time.

5. *output interface* - external applications must be able to easily access the information in the storage system.

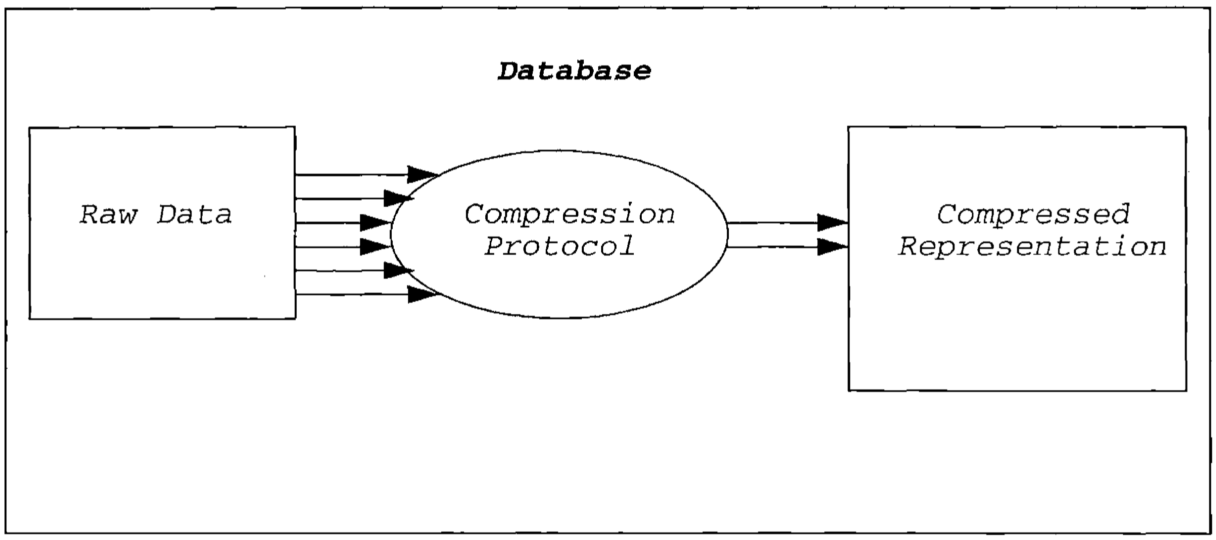
6. *natural language I/O* - questions and answers concerning the stored data will be input to and output from the system in natural language.

The obvious choice for the backbone storage system was a relational database. Their optimized searchability and ease of manipulation made databases clearly preferable to files or other possible options. Simply put, databases are used for the storage and coordination of large amounts of information, which is exactly what I needed.

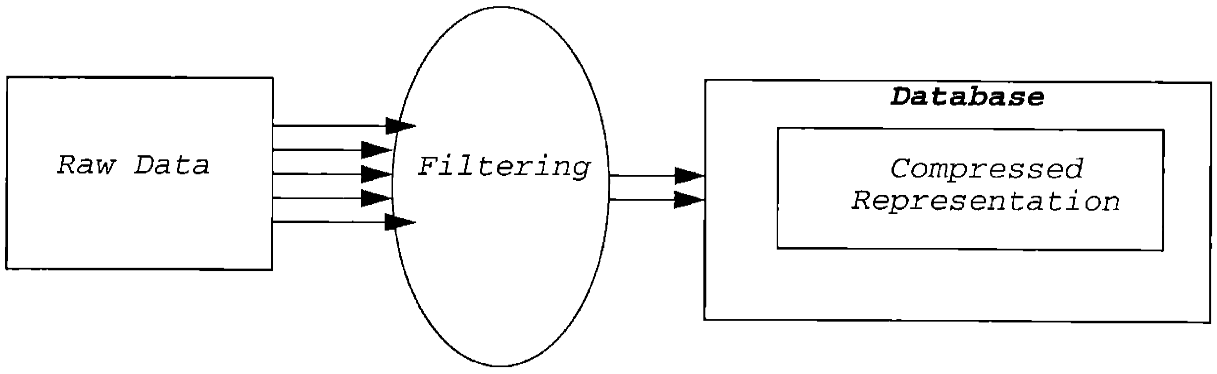
To ease that coordination before the actual storage, I decided that, as part of the input interface, I would create a central server to interface with the other parts of the system. This server is the controlling force behind the entire system. It has access to the current state of the room and is the only entity that can update the official "locked" version of the current state. The central server is responsible for listening to the input interface and, when alerted that a change (or changes) has occurred in the room, it must accordingly update the "locked" current state and store the new data permanently in the backbone storage system.

Another integral part of what I term the “input interface” is the portion of the system that actually interfaces with the person-tracking server. This “camera interface” directly receives all of the updates that the person-tracker sends out. It is here that the checking is done to determine whether or not changes have been made to the current room state. If the interface determines that changes have been made, it alerts the central server, which then propagates those changes to the locked current state and to the permanent storage. The two types of changes that the camera interface can detect are the appearance of a new person in the room and an update to an existing person’s physical attributes (i.e., x-position, y-position, or height). Since the person-tracker itself only sends updates for the people it currently detects in the room, a person leaving the room must be detected by an entity that has a sense of the larger picture (i.e., time between updates, trajectories, etc.), hence such detection is done in the central server.

In effect, the camera interface acts as both a portion of the input interface and as the compression protocol. Originally I had envisioned a scenario in which compression of the data would be performed asychronously to data storage.



In that scenario, every bit of information received from the person-tracker would be stored, regardless of redundancy. I had intended to write a program that would crawl over the database after the initial storage and filter out the redundant entries, leaving only the entries that designated a change of some sort. I eventually realized, however, that there was no need to do that post-processing when I could simply utilize the existing notion of "current state" and filter the input for changes against that state while the system was running. The camera interface performs that filtering and, as a result, the central server ends up storing only essential information.



Another issue that had to be addressed was how to best allow external applications to access the current room state. I decided that it was reasonable to assume that only the person-tracker would need to actually modify the information. All other applications would simply be allowed to view the "locked" current state (i.e., output interface). At any point in time, that state would simply consist of the people in the room at that time and the physical and temporal (time first seen and time last modified) values associated with them.

For debugging and demonstration purposes, and to test the output interface, I decided to create two graphical user interfaces to interact with the system. The first is a live display, which shows "person" movement in the room as soon as the information is passed to the system. The second is a replay display, which enables a user to select a segment of data (presumably copied from the storage system) and to replay that data in sequence. These interfaces allow a user to ask questions about a person's position, time in the room, etc., and be able to verify the correctness of the returned answers by simply watching what's happening with their own eyes, either live or through replay.

Since creating a connection between the Intelligent Room and START is the main goal of my system, I of course needed to interface the data collection and storage portions of my system with natural language. Starting from the START-Omnibase model that I described earlier, and given that Omnibase already allows access to SQL databases, I felt that I could modify that existing implementation to use the Intelligent Room, rather than the Web, as the source for external information. This worked out well, as the Intelligent Room can be viewed as a large, specifically focused, data-collection system.

5 System Implementation

I used the JAVA programming language to implement several portions of the system. I chose JAVA for two reasons. Besides ease of programming, I have previous experience interfacing with my own group's natural language system (written in Scheme), from JAVA. This decision was made to speed the eventual front-end integration between my system and START.

5.1 VisualBlob

To ease the manipulation of "people" in the room, I created a class called VisualBlob to contain all the essential information pertaining to that person. Besides the person's name, VisualBlob contains both temporal and physical information. The temporal information consists of the time the person was first seen, the time the person was last seen, and the time the person's physical information was last modified. The physical information consists of the person's x-position, y-position, z-position, and a boolean flag that shows whether or not the person's physical state has been updated since the last time it was seen. Equality between people is determined by equivalence between name and each of the physical attributes. Since it was easy to do so, I decided to store the number anyway, in case it were to be utilized once again.

```
class VisualBlob implements Serializable {
    String dateAndTimeCreated;
    String dateAndTimeModified;
    long createdMillis;
    long modifiedMillis;
    long lastSeen;
    String name;
    float xPos;
    float yPos;
    float zPos;
    boolean stateUpdated;
```

```
        .
        .
        }
```

5.2 Blob Vector

For the central server to know when to update the "locked" room state, it needed to know both when changes had occurred and what those changes were. To facilitate this, I decided to create the BlobVect class as a container for the constantly updated state of the room. It contains all the VisualBlobs that currently exist in the room. The camera interface is responsible for modifying the state of those blobs as it receives new information from the person-tracking server. When it does so, it marks the BlobVect as having been updated. As with individual blobs, that marking is done via an internal boolean called *stateUpdated.* I implemented the BlobVect class as a remote JAVA object so that it could be shared by separate programs. Through its access to that remote object, the central server is able to check the *stateUpdated* boolean and, if the state has in fact been updated, make the appropriate changes to the "locked" room state.

5.3 Database

A PostgreSQL database is where both the "locked" current room state and the accumulated room data are stored. The database contains seven tables. When the central server iterates through the shared BlobVect and decides that a new blob should be added or that an existing blob's information should be updated, the new blob is copied into tables called TempLastUpdates and TempCurrentData. Once the server has finished its iteration, the entire TempLastUpdates table is copied to the table LastUpdates and cleared. LastUpdates is then copied to the table RoomData for permanent storage. The main reason for all the copying involved is to avoid incomplete updates. Copying information for final storage in RoomData should be an atomic operation. Since the

RoomData table can potentially be very large, it is preferable to access it once for a large copy command than multiple times for smaller copy commands. The intermediate step involving copying to the LastUpdates table is performed so that a listing of which blobs most recently changed can be quickly retrieved without doing an intensive search through the RoomData table. Since LastUpdates is only cleared just before it is modified, that listing can be retrieved at any point.

Just as when updating RoomData, updates to CurrentData (the "locked" representation of the current room state) should be atomic. The aforementioned copying of new or updated blobs to TempCurrentData serves the same purpose as copying them to TempLastUpdates. Similarly, when the server's iteration through the BlobVect is complete, the data from TempCurrentData is atomically copied to the CurrentData table. The reason for having two separate "Temp" tables is that all blobs, even when not updated, are copied to TempCurrentData. This is so that the table will always have complete information on all the blobs in the room. In effect, a new snapshot of the room is added every time. This keeps the CurrentData from having "in between" times where only part of the data for a particular update (i.e., central server iterating through the BlobVect) has been added.

The tables I have yet to mention, Symbols and Attributes, play an important role in the language portion of the system and will be discussed later.

5.4 Camera Interface

The CameraInterface class is the direct interface between the person-tracker and my central server. When idle it simply sits on a port listening for connections. Once the person-tracker connects and starts rapidly sending updates in String form, the camera interface is responsible for parsing, packaging, and forwarding the information it receives.

Each line read from socket represents a "blob" being detected in the Intelligent Room. The first thing the person tracker does with a received line is to parse out the necessary information. The string is tokenized and used to create a new VisualBlob based on the values retrieved.

The camera interface then tests to see whether or not that blob already exists in the BlobVect.

```
// reading line of input from camera server
    String currentBlobString = infoReader.readLine();
    VisualBlob currentBlob = parseBlobFromCameraString(current-
BlobString);
    int blobIndex = remoteBlobs.blobExists(currentBlob);

    if (blobIndex == -1) {
//blob didn't exist previously
//so add it to the set of blobs currently in the room
      currentBlob.setUpdated(true);
      remoteBlobs.add(currentBlob);
      remoteBlobs.setUpdated(true);
      System.out.println("added blob");
    }
    else {
//blob already exists
//so retrieve old version of this blob to test for differences
      VisualBlob oldBlob = (VisualBlob)(remote-
Blobs.vec()).get(blobIndex);
                  .
                  .
                  .
```

This is done by comparing the new blob's name to the names of those in the BlobVect. If the blob doesn't exist, it is simply added to the BlobVect and both it and the BlobVect are marked as having been updated. If the blob did exist, then the BlobVect version of that blob is retrieved for more comparisons and the time that the blob was last seen is updated for both versions. If the two blobs are equivalent (using the VisualBlob notion of equality), nothing is done. Hence, updates to my representing of the room state are only made when actual changes occur. If the two blobs are different, however, the permanent temporal information (i.e.,the time created) of the old blob (the one stored in the BlobVect) is copied into the new blob (which contains the latest positional information) and the old blob is replaced by the new one. Both the blob and the BlobVect are marked as having been updated.

5.5 RoomInfoServer

As stated previously, the purpose of the central server (class RoomInfoServer) is to monitor the shared BlobVect for changes and to make the appropriate updates to the database. In my implementation, the central server queries the BlobVect every 20 milliseconds to find out if the BlobVect's state has been updated. If it has, the server begins an iteration through the blobs. For each VisualBlob, the blob is first tested to see whether or not it is still inside the room. This is determined by checking the current system time against the time that blob was last seen (by the camera interface). If the difference is larger than a threshold value, which I set at 30 seconds, the blob is marked as being outside the room. When this happens, the String "OUTSIDE ROOM" is stored in the positional attributes of the blob's database representation and the live version is given large, normally unachieveable negative values for all of its location info. The blob is then checked for existence in the database. If it doesn't exist, the blob's name is added to the database's Symbol table (again, the significance of this table will be discussed later). Finally, each blob is individually tested to find out whether its state has been updated.

```
if (!(postgresConnection.symbolExists(currentBlob.getName()))) {
    //if the symbol didn't exist previously, add it to the "sym-
bols" table
    postgresConnection.addSymbol(currentBlob.getName());
}

if (currentBlob.stateUpdated()) {
    //if blob has been updated, the new data gets inserted into
    //both the "permanent data" table and the
    //"current room state" table.

    postgresConnection.insertBlobIntoTempLastUpdates(current-
Blob);

    currentBlob.setUpdated(false);

}
else {
    //if the blob hasn't been updated, the data gets
    //inserted only into the "current room state" table
    //this allows the server to keep track of when the
    //blob was last detected for making determinations
    //like when a person has left the room
    currentBlob.setModifiedMillis(oldMillis);
    postgresConnection.insertBlobIntoTempCurrentData(current-
Blob);
```

If it has, it is inserted into both the TempLastUpdates and TempCurrentData tables. If it hasn't been updated, it is only inserted into TempCurrentData. Again, this ensures that CurrentData will contain a complete snapshot of the current room state while RoomData will only be updated when changes are made to a particular blob. Once it has iterated through all the blobs, the server initiates all the appropriate transfers of information between database tables.

If the BlobVect's state was not updated, the server simply iterates through each blob and tests whether or not the blob is still considered "inside the room". If it isn't (because too much time has passed since it was last seen), the BlobVect entry for that blob is updated with the aforementioned large negative positional information. Of course, the next time an update for that blob is received, that information will be overwritten with new, valid positional data.

5.6 Displays

For both debugging purposes and aesthetics I also implemented a simple GUI to show what is going on in the room at a particular moment. The display consists of a blank area, meant to represent the Intelligent Room, where every 100 milliseconds the display draws whatever blobs are current in the shared remote BlobVect. The display writes the blobs' names at the positions specified by their internal data. If data is coming in live, the blobs can be seen moving around the display as their positions change and are constantly updated on the screen.

In order to be able to perform "replays" of chunks of saved data, I also implemented a modified version of my display that takes input from a file. The file takes the same format as an SQL dump of the permanently stored data. As it is read in to my replay program, each line (representing a particular blob's characteristics at that moment) is written to a port where it is used to update a remote BlobVect (a different one than is used by the live display and other external programs). Those lines are written to the port at intervals determined by the times specified in the blobs' information. For instance, if a particular data line of the file corresponds to blob "1" being seen at time t and the succeeding line corresponds to blob "2" being seen at time q, then there will be $q - t$ milliseconds between the update for blob "1" and the update for blob "2". That way the replay display shows blob movement at the same rate that the information was originally received.

5.7 Language Interface

To allow questions to be asked about the room's information I modified the existing START-Omnibase interaction to allow for interactions between START and MAVIS. Basically, instead of getting information from Omnibase, I had START get its information from MAVIS' database. To accomplish this I first wrote scripts (in Scheme, the language used for Omnibase scripts) to return what I considered useful information from the database. For example, given a particular blob's name, I wrote scripts to return stored information such as its position, its height, the time it was first seen, etc. Here is the script to retrieve the x-position of a particular blob:

```
(lambda (name)
  (list (get-currentdata-column name "xpos")))
```

where *get-currentdata-column* is a general function that accesses the "current data" table in the database and "xpos" is the name of the column containing the x-position.

```
(define (get-currentdata-column name column)
  (let* ((a (sql (string-append
                 "select "
                 column
                 " from currentdata"
                 " where name = " (sqlstr name)))))
    (if (null? a) #f (caar a))))
```

Then I moved on to writing slightly more complicated scripts to calculate information like the distance between two blobs, the distance between a blob and a stationary object in the room (like a telephone), the amount of time a blob has been in the room, etc. What those scripts output constitutes the information that I can give in response to a user query from START. As mentioned earlier, schemata are necessary to match those queries to the information that answers them. I wrote schemata to answer questions like "Where is X?", "How long has X been in the room?", "Is X near the telephone?", etc. Here is an example schema:

```
;;Answers questions of the form:
;;"where is jerome?"
;;"where is jerome located?"
;;"what is jerome's position?"
;;"is jerome in the room?"

(def-schema
  :Phrases
  '(
    "any-smartroom-blob's position"
    )
  :Sentences
  '(
    "any-smartroom-blob is located near the wall"
    "any-smartroom-blob is in the room"
    )
  :Long-Text
  '((show-smartroom 'any-smartroom-blob 'get-current-position))
     .
     .
     .
      )
```

Also, consistent with the way Omnibase handles symbols, I implemented a Symbols table to allow START to know what "blobs" it has information about. When detailing the operation of the RoomInfoServer I mentioned that newly detected blobs would be added to the Symbols table. This allows the system to dynamically "learn" about new blobs as they are first seen. This is necessary for the system to be able to answer questions about things happening live in the room. The answers to those questions, as retrieved from the scripts, are then wrapped in natural-sounding text by functions I wrote which utilize START's language capabilities.

```
===> who is in the room?
<P>jerome and bill are currently in the room.

===> where is jerome?
<P>jerome has been in the room for 00:00:49.
<HR>
<P>jerome's current position is (-2.4, 0.2).
```

6 Evaluation

In testing and using MAVIS, I was able to attain the basic functionality that I had hoped for. When people move around in the Intelligent Room, MAVIS can answer questions about their number, position, persistence, etc. It is also simple to combine those bits of information to make more subjective assessments about things like "nearness" to a particular object. Although I was happy with the system's eventual performance, I do believe that with a few key changes, it could be made to work significantly better.

6.1 Data Transfer

The most essential part of MAVIS is the connection to the Intelligent Room's person-tracker. Though the constant receipt of string updates was sufficient for my purposes, a system that would hope to track large amounts of people might be better served with some other method of information transfer. The main problem with the current method is that it is difficult for me to tell when the information from a particular snapshot of the Intelligent Room has been completely sent. Since the receipt of the strings is sequential and there is no delimiter between sets of strings, it's impossible to know when you've gotten everything. I was able to get around this problem by updating so quickly that the differences in timing weren't noticeable. For instance, a string update for one blob might be read in at 00000001 milliseconds and another blob's update might be read in at 00000005 milliseconds. Those two blobs were most likely seen by the camera (at their respective positions) at the same moment. When they are sent serially to my server, however, I have no way of knowing for sure whether they were actually seen at the same time or 4 milliseconds apart. All that I can know is the times that I received them. For a system on a larger scale, this could become an important issue. I tried to design the system to be as independent as possible, but I believe that this problem could best be solved through closer integration with the person-tracker. More information transfer, and likely a different method of transfer, between the storage system and the tracker itself would lead to more accurately stored information.

In keeping with the need for more effective information transfer between the person-tracker and MAVIS, I believe that the movement of data between my central server and the database could also be improved. Though designed partially to cope with the aforementioned temporal uncertainty concerning the receipt of updates, my implementation is vulnerable to performance degradation if a lot of information is being transferred. Copying and clearing multiple tables in the database is slow, so the less frequently that needs to be done, the better. I believe that this could be remedied both through the aforementioned closer interaction with the person tracker
and through implementation of the data "locking" in the central server rather than in the database itself.

6.2 Data Smoothing

I had originally intended to build in a "smoothing" system that would crawl over the data and attempt to correct what seemed like obvious glitches in tracker detection. In the end, I wrote a program that attempted to perform that function, but didn't incorporate it as an automatic portion of the overall system for fear of making mistakes and ruining the original data. Instead, I left it as an external program that could be run over portions of the data if so desired.

7 Future Work

The most important aspect of a system like MAVIS is accuracy. Whether temporal or positional, storing the correct numbers for the correct objects is essential. I previously mentioned that my system could be improved through more efficient receipt of information from the person-tracker, but that is only a beginning. To really maximize the potential of the interaction, there should be a feedback loop between the person-tracker and the database. Having access to previously stored information could allow the tracker to make smarter decisions about whether or not the sudden appearance of a person in the center of the room is a momentary glitch or whether there is existing data to support it. I experimented with post-processing stored information in an attempt to smooth the data (i.e., correlate sudden appearances of new blobs on a previously seen blob's trajectory to the old blob), thereby allowing me to display more sensible results on retrieval. It would be much more effective, however, to allow the person-tracker to access that information for its own use.

Also, more generally, any number of systems could potentially be connected to make use of the stored information. For that matter, more and different types of information could be stored (for instance: audio, video, etc.). The integration of multi-modal systems is a complicated task, but there are worthwhile benefits to be attained from a scenario in which all systems are aware of, and accessible to, one another.

References

[1] Darrell, Trevor, David Demirdjian, Neal Checka, and Pedro Felzenswalb. *Plan-view Trajectory Estimation with Dense Stereo Background Models*. Proceedings of the International Conference on Computer Vision, 2001.

[2] Hanssens, Nicholas, Ajay Kulkarni, Rattapoom Tuchinda, and Tyler Horton. *Building Agent-Based Intelligent Workspaces*. To appear in Proceedings of The International Workshop on Agents for Business Automation. Las Vegas, NV, 2002

[3] Katz, Boris. *Using English for indexing and retrieving*. In Proceedings of the 1st RIAO Conference on User-Oriented Content-Based Text and Image Handling (RIAO '88), 1998.

[4] Katz, Boris. *Annotating the World Wide Web using natural language*. In Proceedings of the 5th RIAO Conference on Computer Assisted Information Searching on the Internet (RIAO '97), 1997.

[5] Katz, Boris, Sue Felshin, Deniz Yuret, Ali Ibrahim, Jimmy Lin, Gregory Marton, Alton Jerome McFarland, and Baris Temelkuran. *Omnibase: Uniform access to heterogeneous Data for question answering*. In Proceedings of the 7th International Workshop on Applications of Natural Language to Information Systems (NLBD 2002), 2002.

[6] Koelma, Dennis and Arnold Smeulders. *A blackboard infrastructure for object-based image interpretation*. Computing science in The Netherlands, eds: E.Backer, CWI, Amsterdam, 1994, 136-147.

[7] Nii, Penny H. *The Blackboard Model of Problem Solving and the Evolution of Blackboard Architectures (Parts 1 & 2)*. The AI Magazine, 1986.

[8] Winograd, Terry. *Towards a Human-Centered Interaction Architecture*. Working paper for Stanford project on Interactive Workspaces. Stanford University, Version of April, 1999.

Appendix

Relevant classes only, reproducible utilities (displays, frames, filehandlers, socket listeners, etc.) not included

```
(in-package :start)

(make-neuter-proper-nouns
 (any-smartroom-blob2
  :gens (any-smartroom-blob)
  :prop ((matching-symbol t))
  )
 (any-smartroom-object
  :gens (table chalkboard)
  :prop ((matching-symbol t))
  )
 )


;;ANSWERS QUESTIONS
;; who is in the room?

(def-schema
 :Phrases
 '(
  )
 :Sentences
 '(
  "someone is located in the room"
  )
 :Long-Text
 '((show-smartroom 'fake 'GET-PEOPLE-IN-ROOM))
 :Sons
 '(*no-db-links*)
 :Liza
 '()
 :function-call T
 )

;;ANSWERS QUESTIONS
;; where is jerome?
;; what is jerome's position?
;; is jerome in the room?

(def-schema
 :Phrases
 '(
  "any-smartroom-blob's position"
```

```
   )
  :Sentences
  '(
    "any-smartroom-blob is located near the wall"
    "any-smartroom-blob is in the room"
   )
  :Long-Text
  '((show-smartroom 'any-smartroom-blob 'GET-CURRENT-POSITION))
  :Sons
  '(*no-db-links*)
  :Liza
  '()
  :function-call T
  )

;;ANSWERS QUESTIONS
;; how far is jerome from mary?
;; what is the distance between mary and jerome?

(def-schema
  :Phrases
  '(
    "the distance between any-smartroom-blob and any-smartroom-blob2"
   )
  :Sentences
  '(
    "any-smartroom-blob is very far away from any-smartroom-blob2"
   )
  :Long-Text
  '((reply-current-distance-between 'any-smartroom-blob 'any-smartroom-blob2))
  :Sons
  '(*no-db-links*)
  :Liza
  '()
  :function-call T
  )

;;ANSWERS QUESTIONS
;; has jerome been in the room for a long time?
;; how long has jerome been in the room?

(def-schema
  :Phrases
  '(

   )
```

```
 :Sentences
 '(
"any-smartroom-blob has been in the room for a very long time"
   "any-smartroom-blob has been in the room for quite long"
)
 :Long-Text
 '((show-smartroom 'any-smartroom-blob 'GET-CURRENT-TIME-IN-ROOM))
 :Sons
 '(*no-db-links*)
 :Liza
 '()
 :function-call T
 )

;;ANSWERS QUESTIONS
;; is jerome sitting?
;; is jerome standing?

(def-schema
 :Phrases
 '(

   )
 :Sentences
 '(
   "any-smartroom-blob is sitting"
   ;;"any-smartroom-blob is seated"
  "any-smartroom-blob is standing"
)
 :Long-Text
 '((reply-sitting 'any-smartroom-blob))
 :Sons
 '(*no-db-links*)
 :Liza
 '()
 :function-call T
 )

;;ANSWERS QUESTIONS
;; is jerome near the telephone?
;; is jerome near the chalkboard?

(def-schema
 :Phrases
 '(
  )
 :Sentences
 '(
  "any-smartroom-blob is near the any-smartroom-object"
)
 :Long-Text
 '((reply-nearness-to-object 'any-smartroom-blob 'any-smartroom-object))
 :Sons
 '(*no-db-links*)
 :Liza
 '()
 :function-call T
 )

;;ANSWERS QUESTIONS
;; when did jerome enter the room?
;; did jerome enter the room yesterday?
(def-schema
 :Phrases
 '(

  )
 :Sentences
 '(
  "any-smartroom-blob entered the room yesterday"
  )
 :Long-Text
 '((show-smartroom 'any-smartroom-blob 'GET-CREATED-TIME))
 :Sons
 '(*no-db-links*)
 :Liza
 '()
 :function-call T
 )
```

smartroom-functions.lisp

```
(in-package :start)

;;Creating smartroom service

#+genera
(net:define-protocol :start-smartroom (:start-smartroom :byte-stream)
  (:invoke (service-access-path)
    (apply 'invoke-omnibase-service service-access-path
   (neti:service-access-path-args service-access-path))))
#+(or lucid allegro)
(define-protocol :start-smartroom (:start-smartroom :byte-stream)
  (:invoke #'invoke-omnibase-service))
(eval-when (:load-toplevel :execute)
  (add-tcp-port-for-protocol :start-smartroom 8064))


(use-start-host 'kiribati)
(setq *omnibase-service* :start-smartroom)


;; SMARTROOM-SPECIFIC FUNCTIONS

(def-omnibase-class smartroom-blob
  :source-name "Intelligent Room"
  :gender :neuter
  )

(defun show-smartroom (matching-word field)
;;  (pvalues matching-word field)
  (let* ((blobname (get-matching-value-root-singular matching-word))
 )
    (show-smartroom-aux matching-word blobname field 0)))

(defun show-smartroom-aux (matching-word blobname field counter)
;;(pvalues blobname counter)
  (let* ((value (omnibase-get "smartroom-blob"                              (string-downcase
blobname) field :use-cache nil)))
    (if (equal (car value) 'NIL)
(if (equal counter 5)
    'NIL
    (progn
     (sleep 1)
     (show-smartroom-aux matching-word blobname field (+ counter 1))))
(cond ((eq field 'GET-PEOPLE-IN-ROOM)
      (reply-people-in-room value))
     ((eq field 'GET-CURRENT-XPOS)
      (reply-current-xpos blobname value))
     ((eq field 'GET-CURRENT-YPOS)
      (reply-current-ypos blobname value))
     ((eq field 'GET-CURRENT-POSITION)
      (reply-current-position blobname value))
     ((eq field 'GET-CURRENT-TIME-IN-ROOM)
      (reply-current-time-in-room blobname value))
     ((eq field 'GET-CREATED-TIME)
      (reply-time-entered-room blobname value))
     (else "didn't work")))))


(defun get-distance-between-two-points (point1 point2)
  (let* ((xpos1 (car point1))
 (xpos2 (car point2))
 (ypos1 (cadr point1))
 (ypos2 (cadr point2))
 (straight-line-dist (sqrt (+ (expt (- xpos1 xpos2) 2) (expt (- ypos1 ypos2) 2)))))
;;   (pvalues xpos1 ypos1 xpos2 ypos2 straight-line-dist)
   straight-line-dist))


;;NATURALLY FORMULATED REPLIES


(defun reply-people-in-room (lst)
;;  (pvalues lst)
  (recording-query-reply ('t :p t)
   (cond
    ((and (eq (length lst) 1) (equal (car lst) "ROOM IS EMPTY"))
     (format t "No one is currently in the room."))
    ((endp (cdr lst))
     (format t "~A is currently in the room." (car lst)))
    (t
     (format t "~A are currently in the room." (gen-np lst :how :bare :stream nil))))))


(defun reply-current-xpos (blobname value)
   (recording-query-reply ('t :p t)
    (gen-np blobname :how :bare :stream t)))

(defun reply-current-ypos (blobname value)
   (recording-query-reply ('t :p t)
    (gen-np blobname :how :bare :stream t)))
```

```
(defun reply-current-position (blobname value)
;;  (pvalues blobname value)
  (if (and blobname value)
      (recording-query-reply ('t :p t)
(if (or (equal (car value) "OUTSIDE ROOM") (equal (cadr value) "OUTSIDE ROOM"))
    (format t "~A is currently outside the room." (gen-np blobname  :how :bare :stream nil))
    (format t "~Acurrent position is (~A, ~A)." (gen-np blobname :case 'genitive :how :bare
:stream nil) (car value) (cadr value))))
      'NIL))

(defun reply-current-distance-between (matching-word1 matching-word2)
  (let* ((name1 (string-downcase (get-matching-value-root-singular matching-word1)))
 (name2 (string-downcase (get-matching-value-root-singular matching-word2))))
 (if (or (equal name1 "nil") (equal name2 "nil"))
    'nil
    (let* ((values (omnibase-get "smartroom-blob" (string-append name1 "&" name2) "GET-
CURRENT-DISTANCE-BETWEEN" :use-cache nil))
   (pos1 (list (car values) (cadr values)))
   (pos2 (list (caddr values) (cadddr values))))
;;      (pvalues pos1 pos2)
      (if (member 'NIL values)
  (reply-current-distance-between-aux name1 name2 0)
  (recording-query-reply ('t :p t)
    (if (member "OUTSIDE ROOM" pos1 :test 'equal)
 (format t "~A isn't in the room right now." name1)
 (if (member "OUTSIDE ROOM" pos2 :test 'equal)
    (format t "~A isn't in the room right now." name2)
    (format t "~A and ~A are currently ~A apart." name1 name2 (sqrt (+ (expt (- (car values)
(caddr values)) 2) (expt (- (cadr values) (cadddr values)) 2))))))))))))

(defun reply-current-distance-between-aux (name1 name2 counter)
  (let* ((value (omnibase-get "smartroom-blob" (string-append name1 "&" name2) "GET-CUR-
RENT-DISTANCE-BETWEEN" :use-cache nil))
 (pos1 (list (car values) (cadr values)))
 (pos2 (list (caddr values) (cadddr values))))
   (if (member 'NIL values)
(if (equal counter 5)
    'NIL
   (reply-current-position-aux name1 name2 (+ counter 1)))
(recording-query-reply ('t :p t)
  (if (and (member "OUTSIDE ROOM" pos1 :test 'equal) (member "OUTSIDE ROOM" pos2
:test 'equal))
     (format t "Neither ~A nor ~A is in the room right now." name1 name2)
 (if (member "OUTSIDE ROOM" pos1 :test 'equal)
 (format t "~A isn't in the room right now." name1)
 (if (member "OUTSIDE ROOM" pos2 :test 'equal)
    (format t "~A isn't in the room right now." name2)
    (format t "~A and ~A are currently ~A apart." name1 name2 (sqrt (+ (expt (- (car pos1) (car
pos2)) 2) (expt (- (cadr pos1) (cadr pos2)) 2))))))))))

(defun reply-current-time-in-room (name1 value)
;;(pvalues name1 value)
 (if (and name1 value)
    (recording-query-reply ('t :p t)
(format t "~A has been in the room for ~A."  (gen-np name1 :how :bare :stream nil) (car value)))
    'NIL))

(defun reply-time-entered-room (name1 value)
;;(pvalues name1 value)
(if (and name1 value)
 (recording-query-reply ('t :p t)
  (format t "~A entered the room at ~A."  (gen-np name1 :how :bare :stream nil) (car value)))
 'NIL))

(defun reply-sitting (matching-name)
 (let* ((blobname (string-downcase (get-matching-value-root-singular matching-name)))
(value (omnibase-get "smartroom-blob"
   blobname "GET-CURRENT-HEIGHT" :use-cache nil))
(status-string (if (> (string-to-number (car value)) 3)
  " is standing."
  " is sitting.")))
;;   (pvalues blobname value status-string)
  (if (and blobname value status-string)
(recording-query-reply ('t :p t)
 (format t status-string (gen-np blobname :how :bare :stream t)))
'NIL)))


(defun reply-nearness-to-object (matching-name matching-object)
 (let* ((blobname (string-downcase (get-matching-value-root-singular matching-name)))
(object (string-downcase (get-matching-value-root-singular matching-object)))
(chalkboard-pos '(0.0 0.0))
(table-pos '(0.0 3.0))
(blob-pos (omnibase-get "smartroom-blob" blobname "GET-CURRENT-POSITION"))
(object-pos (cond ((string-equal object "chalkboard")
  chalkboard-pos)
 ((string-equal object "table")
  table-pos)
 (t 'nil)))
(dist-between (if (or (eq object-pos 'nil) (eq blobname 'nil))
 'nil
```

```
  (get-distance-between-two-points blob-pos object-pos))))
;;   (pvalues blobname blob-pos object object-pos dist-between)
   (if (and blobname object dist-between)
(recording-query-reply ('t :p t)
    (format t "~A is ~A from the ~A." (gen-np blobname :how :bare :stream nil) dist-between
(gen-np object :how :bare :stream nil)))
'nil)))
```

```
;; returns the names of all the people currently in the room
smartroom-blob  GET-PEOPLE-IN-ROOM
(lambda (fake)
(let* ((a (sql "select name from currentdata where xpos!='OUTSIDE ROOM'")))
  a))

;;returns the last time the positional data changed for a particular blob
smartroom-blobGET-LAST-MOVEMENT-TIME
(lambda (name)
  (list (get-currentdata-column name "modifiedtime")))

;;returns the time a particular blob was created
smartroom-blobGET-CREATED-TIME
(lambda (name)
  (list (get-currentdata-column name "createdtime")))

;;returns the current "unknownfloat" for a particular blob
smartroom-blobGET-CURRENT-UFO
(lambda (name)
(list  (get-currentdata-column name "unknownfloat")))

;;returns the current x-position for a particular blob
smartroom-blobGET-CURRENT-XPOS
(lambda (name)
(list  (get-currentdata-column name "xpos")))

;;returns the current y-position for a particular blob
smartroom-blobGET-CURRENT-YPOS
(lambda (name)
(list  (get-currentdata-column name "ypos"))

;;returns the current height for a particular blob
smartroom-blobGET-CURRENT-HEIGHT
(lambda (name)
(list   (get-currentdata-column name "zpos")))

;;returns the current coordinate position for a particular blob
smartroom-blobGET-CURRENT-POSITION
(lambda (name)
 (let* ((xpos-str (car (get "smartroom-blob" name "GET-CURRENT-XPOS")))
 (ypos-str (car (get "smartroom-blob" name "GET-CURRENT-YPOS")))
 (xpos (if (equal? xpos-str "OUTSIDE ROOM")
   "OUTSIDE ROOM"
   (string->number xpos-str)))
(ypos (if (equal? ypos-str "OUTSIDE ROOM")
  "OUTSIDE ROOM"
  (string->number ypos-str))))
   (list xpos ypos)))

;;returns the current distance between two blobs
smartroom-blobGET-CURRENT-DISTANCE-BETWEEN
(lambda (inputstr)
 (let* ((name1 (match:prefix (match "&" inputstr)))
 (name2 (match:suffix (match "&" inputstr)))
 (pos1 (get "smartroom-blob" name1 "GET-CURRENT-POSITION"))
 (pos2 (get "smartroom-blob" name2 "GET-CURRENT-POSITION"))
       (values (append pos1 pos2)))
 values))

;;returns the length of time a particular blob has been in the room
smartroom-blob GET-CURRENT-TIME-IN-ROOM (lambda (name)
 (let* ((a (sql (string-append "select age('now', (select min(createdtime) from roomdata where
name='" name "'))"))))
   (car a)))
```

```
package smartroom;

import java.io.*;
import java.util.*;

class VisualBlob implements Serializable {

   //container class for room "blobs".  Each blob represents a person.

   String dateAndTimeCreated;
   String dateAndTimeModified;
   long createdMillis;
   long modifiedMillis;
   long lastSeen;
   String name;
   float unknownFloat;
   float xPos;
   float yPos;
   float zPos;
   boolean stateUpdated;

   //constructor
   public VisualBlob(String n, float ufo, float x, float y, float z) {
dateAndTimeCreated = getDateAndTime();
createdMillis = getCurrentMillis();
modifiedMillis = createdMillis;
dateAndTimeModified = dateAndTimeCreated;
name = n.toLowerCase();  //store names as lowercase
unknownFloat = ufo;
xPos = x;
yPos = y;
zPos = z;
lastSeen = System.currentTimeMillis();
   }

   //constructor
   public VisualBlob(String date, long dateMillis, String dateMod, long dateModMillis, String n,
float ufo, float x, float y, float z) {
unknownFloat = ufo;
xPos = x;
yPos = y;
zPos = z;
```

```
dateAndTimeCreated = date;
createdMillis = dateMillis;
dateAndTimeModified = dateMod;
modifiedMillis = dateModMillis;
name = n.toLowerCase(); //store names as lowercase
lastSeen = System.currentTimeMillis();
   }

   //sets updated state for this
   public void setUpdated(boolean x) {
stateUpdated = x;
dateAndTimeModified = getDateAndTime();
   }

   //returns true if this has been updated
   public boolean stateUpdated() {
return stateUpdated;
   }

   //sets state variables for this
   public void setXPos(float x) { xPos = x; }
   public void setYPos(float y) { yPos = y; }
   public void setUnknownFloat(float ufo) { unknownFloat = ufo; }
   public void setZPos(float z) { zPos = z; }
   public void setDateModified(String d) { dateAndTimeModified = d; }
   public void setDateCreated(String d) { dateAndTimeCreated = d; }
   public void setCreatedMillis(long m) { createdMillis = m; }
   public void setModifiedMillis(long m) { modifiedMillis = m; }
   public void setLastSeen(long l) { lastSeen = l; }

   //retrieves state variables for this
   public float getXPos() { return xPos; }
   public float getYPos() { return yPos; }
   public float getZPos() { return zPos; }
   public float getUnknownFloat() { return unknownFloat; }
   public String getDateCreated() { return dateAndTimeCreated; }
   public String getDateModified() { return dateAndTimeModified; }
   public long getCreatedMillis() { return createdMillis; }
   public long getModifiedMillis() { return modifiedMillis; }
   public long getLastSeen() { return lastSeen; }
   public String getName() { return name; }

   //returns String representation of this
   public String toString() {
   String s = name + "" + dateAndTimeCreated + "" + createdMillis + "" + dateAndTimeModified
+ "" + modifiedMillis + "" + unknownFloat + "" + xPos + "" + yPos + "" + zPos + "\n";
```

```
return s;
   }

   //tests for equivalece between this and b
   //if the name, x-position, y-postion, and z-position are all
   //the same then this and b are equal
   public boolean equals(VisualBlob b) {
if (!(name.equals(b.getName()))) {
   return false;
}
if (!(xPos == b.getXPos())) {
   return false;
}
if (!(yPos == b.getYPos())) {
   return false;
}
if (!(zPos == b.getZPos())) {
   return false;
}

return true;
   }

   //sets status of this as being outside the room
   public void setOutsideRoom() {
System.out.println("SET BLOB OUTSIDE ROOM");
setUnknownFloat(-1);
setXPos(-200);
setYPos(-200);
setZPos(-200);
   }

   //returns current date and time
   public String getDateAndTime() {
GregorianCalendar cal = new GregorianCalendar();
int month = cal.get(Calendar.MONTH) + 1;
int day = cal.get(Calendar.DAY_OF_MONTH);
int year = cal.get(Calendar.YEAR);
int hour = cal.get(Calendar.HOUR_OF_DAY);
int minute = cal.get(Calendar.MINUTE);
int second = cal.get(Calendar.SECOND);
java.util.Date calDate = cal.getTime();
TimeZone zone = cal.getTimeZone();
boolean daylight = zone.inDaylightTime(calDate);
String timezone = "";
```

```
if (daylight) {
   timezone = "EDT";
}
else {
   timezone = "EST";
}
String currentdatetime = year + "-" + month + "-" + day + " " + hour + ":" + minute + ":" + second
+ " AD " + timezone;

return currentdatetime;
   }

   //returns current milliseconds
   public long getCurrentMillis() {
return System.currentTimeMillis();
   }

}
```

BlobVectImpl.java

```
package smartroom;

import java.rmi.*;
import java.rmi.server.*;
import java.util.*;
import java.io.*;

public class BlobVectImpl extends UnicastRemoteObject implements BlobVect, Serializable {

   //Remote object implementation of the BlobVect interface

   private Vector blobsVect;
   private boolean stateUpdated;
   private boolean lock;

   //constructor
   public BlobVectImpl() throws RemoteException {
blobsVect = new Vector();
   }

   //adds a blob to this
   public synchronized void add(Object o) {
blobsVect.add(o);
   }

   //removes a blob from this
   public synchronized void remove(Object o)  {
int index = findFirst((VisualBlob)o);
blobsVect.removeElementAt(index);

   }

   //returns the number of blobs in this
   public int size() {
return blobsVect.size();
   }

   //returns the Vector of blobs in this
   public Vector vec() {
return blobsVect;
   }

   //returns the index of the first blob contained in this that is
   //VisualBlob.equals() to b
```

```
    public  int findFirst(VisualBlob b) {

 for (int i=0; i<blobsVect.size(); i++) {
    VisualBlob current = (VisualBlob)blobsVect.get(i);
    if (current.equals(b)) {
return i;
    }
}
return -1;
    }

    //set updated state for this
    public synchronized void setUpdated(boolean x) {
stateUpdated = x;
    }

    //returns true if this has been updated
    public boolean stateUpdated() {
return stateUpdated;
    }


    //tests for equivalence between this and another BlobVect
    //if the blobs in this are in the same order as the blobs
    //in bVect and are VisualBlob.equals() to the blob at the
    //corresponding index, then the BlobVects will be deemed
    //equivalent
    public  boolean equals(BlobVect bVect) throws RemoteException{
boolean foundBlob = false;

if (bVect.size() == this.size()) {
    return false;
}

Vector currentVec = this.vec();
Vector testedVec = bVect.vec();

for (int i=0;i<currentVec.size();i++) {
    VisualBlob current = (VisualBlob)currentVec.get(i);

    foundBlob = false;

    for (int j=0;j<testedVec.size();j++) {
VisualBlob tested = (VisualBlob)testedVec.get(j);
if (current.equals(tested)) {
    foundBlob = true;
    break;
}
    }

    if (!foundBlob) {
return false;
    }
}

return true;

    }

    //replaces the Vector of blobs in this with Vector v
    public synchronized void setVec(Vector v) throws RemoteException {
Vector clonedVec = (Vector)v.clone();
blobsVect = clonedVec;
    }

    //returns the index of the first occurance b in this or -1
    //if this doesn't contain b
    public  int blobExists(VisualBlob b) throws RemoteException {
for (int i=0; i<blobsVect.size(); i++) {
    VisualBlob current = (VisualBlob)blobsVect.get(i);
    if ((current.getName()).equals(b.getName())) {
return i;
    }
}
return -1;
    }

    //returns true if this is locked
    public boolean locked() throws RemoteException {
return lock;
    }

    //sets locked state for this
    public synchronized void setLock(boolean l) {
lock = l;
    }

    //replaces blob at specified index with b
    public synchronized void setBlobAt(int index, VisualBlob b) {
blobsVect.set(index, b);
    }
```

```
   //deletes all blobs from this
   public synchronized void clearBlobs() throws RemoteException {
blobsVect = new Vector();
stateUpdated = true;
   }


}
```

```
package smartroom;

import java.net.*;
import java.rmi.*;
import java.sql.*;
import java.io.*;
import java.util.*;
import java.lang.*;
import java.lang.reflect.*;

class SmartroomConnection {

   //maintains connection to PostgreSQL database and performs actions
   //on that database

   private String driver;
   private String url;
   private String user;
   private String password;
   private Connection con;

   //constructor
   public SmartroomConnection(String driver, String url, String user, String password) throws
java.net.UnknownHostException, IOException, SQLException, RemoteException, ClassNot-
FoundException {
this.driver = driver;
this.url = url;
this.user = user;
this.password = password;

Class.forName(this.driver);
this.con = DriverManager.getConnection(this.url, this.user, this.password);
   }

   //returns true if "symbol" exists in the symbols table
   public boolean symbolExists(String symbol) throws SQLException {
String testQuery = "select symbol from symbols where symbol='" + symbol + "'";
Statement testStmt = con.createStatement();
ResultSet returnedSet = testStmt.executeQuery(testQuery);
if (returnedSet.next()) {
   return true;
}
```

```
else {
   return false;
}
   }

   //adds a symbol to the symbols table
   public boolean addSymbol(String symbol) throws SQLException {
String ins = "insert into symbols values ('smartroom-blob', '" + symbol + "', '" + symbol + "')";
Statement insStmt = con.createStatement();
boolean inserted = insStmt.execute(ins);
System.out.println("in addSymbol");
return inserted;
   }

   //copies data from the lastupdates table to the roomdata table
   public void updateRoomData() throws SQLException {

String insertString = "insert into roomdata select * from lastupdates";

Statement ins = con.createStatement();
boolean inserted =  ins.execute(insertString);
ins.close();


   }

   //moves data from templastupdates to lastupdates
   public void updateLastUpdates() throws SQLException {

String insertString = "delete from lastupdates; insert into lastupdates select * from templastup-
dates; delete from templastupdates";

Statement ins = con.createStatement();
boolean inserted =  ins.execute(insertString);
ins.close();

   }

   //moves data from tempcurrentdata to currentdata
   public void updateCurrentData() throws SQLException {

String insertString = "delete from currentdata; insert into currentdata select * from tempcurrent-
data; delete from tempcurrentdata";
Statement ins = con.createStatement();
boolean inserted =  ins.execute(insertString);
ins.close();
```

```
    }

    //inserts a blob into lastupdates
    public void insertBlobIntoLastUpdates(VisualBlob b) throws SQLException {

String insertString = "insert into lastupdates values ('" + b.getName() + "', '" + b.getDateCre-
ated() + "', '" + b.getCreatedMillis() + "', '" + b.getDateModified() + "', '" + b.getModifiedMil-
lis() + "', '" + b.getUnknownFloat() + "', '" + b.getXPos() + "', '" + b.getYPos() + "', '" +
b.getZPos() + "')";

Statement ins = con.createStatement();
boolean inserted =  ins.execute(insertString);
ins.close();

    }

    //inserts a blob into templastupdates and tempcurrentdata
    public void insertBlobIntoTempLastUpdates(VisualBlob b) throws SQLException, RemoteEx-
ception {

String insertString = "";
if (!(RoomInfoServer.blobInsideRoom(b))) {
    insertString = "insert into templastupdates values ('" + b.getName() + "', '" + b.getDateCre-
ated() + "', '" + b.getCreatedMillis() + "', '" + b.getDateModified() + "', '" + b.getModifiedMil-
lis() + "', 'OUTSIDE ROOM', 'OUTSIDE ROOM', 'OUTSIDE ROOM', 'OUTSIDE ROOM');
insert into tempcurrentdata values ('" + b.getName() + "', '" + b.getDateCreated() + "', '" + b.get-
CreatedMillis() + "', '" + b.getDateModified() + "', '" + b.getModifiedMillis() + "', '" + b.getUn-
knownFloat() + "', '" + b.getXPos() + "', '" + b.getYPos() + "', '" + b.getZPos() + "')";

}
else {
    insertString = "insert into templastupdates values ('" + b.getName() + "', '" + b.getDateCre-
ated() + "', '" + b.getCreatedMillis() + "', '" + b.getDateModified() + "', '" + b.getModifiedMil-
lis() + "', '" + b.getUnknownFloat() + "', '" + b.getXPos() + "', '" + b.getYPos() + "', '" +
b.getZPos() + "'); insert into tempcurrentdata values ('" + b.getName() + "', '" + b.getDateCre-
ated() + "', '" + b.getCreatedMillis() + "', '" + b.getDateModified() + "', '" + b.getModifiedMil-
lis() + "', '" + b.getUnknownFloat() + "', '" + b.getXPos() + "', '" + b.getYPos() + "', '" +
b.getZPos() + "')";
}

Statement ins = con.createStatement();
boolean inserted =  ins.execute(insertString);
ins.close();
```

```
    }

    //inserts a blob into tempcurrentdata
    public void insertBlobIntoTempCurrentData(VisualBlob b) throws SQLException {

String insertString = "insert into tempcurrentdata values ('" + b.getName() + "', '" + b.getDate-
Created() + "', '" + b.getCreatedMillis() + "', '" + b.getDateModified() + "', '" + b.getModified-
Millis() + "', '" + b.getUnknownFloat() + "', '" + b.getXPos() + "', '" + b.getYPos() + "', '" +
b.getZPos() + "')";

Statement ins = con.createStatement();
boolean inserted =  ins.execute(insertString);
ins.close();

    }

    //returns a ResultSet containing the data stored in currentdata
    public ResultSet loadRoomState() throws SQLException {

Statement stmt = con.createStatement();
String query = "select * from currentdata";
ResultSet returnedSet = stmt.executeQuery(query);
return returnedSet;

    }

}
```

CameraInterface.java

```
package smartroom;

import javax.swing.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import java.rmi.*;
import java.util.*;

public class CameraInterface implements Serializable {

   //accepts input from cameras and updates the remote object
   //representation of room state

   private static BlobVect remoteBlobs;
   private static BlobVect localBlobs;
   private static Socket dataSock;
   private int socketTimeoutSeconds = 300;
   private int socketTimeoutMillis = socketTimeoutSeconds * 1000;

   //MAIN
   public static void main(String[] args) {

try {
   InetAddress localAddr = InetAddress.getLocalHost();
   String localHostName = localAddr.getHostName();
   String url = "rmi://" + localHostName + ".ai.mit.edu/";

   int infoPort = 8060;
   remoteBlobs = (BlobVect)Naming.lookup(url + "Room Blobs");

   //Create socket to receive room info

   ServerSocket infoSocket = new ServerSocket(infoPort);
   while (true) {
System.out.println("waiting for connection");

//waits for information from the cameras
dataSock = infoSocket.accept();
System.out.println("got connection");
InputStreamReader temp  = new InputStreamReader(dataSock.getInputStream());
BufferedReader infoReader = new BufferedReader(temp);

// start loop
```

```
while (true) {
   try {

String currentBlobString = infoReader.readLine();
VisualBlob currentBlob = parseBlobFromCameraString(currentBlobString);
int blobIndex = remoteBlobs.blobExists(currentBlob);


if (blobIndex == -1) {
   //blob didn't exist previously
   currentBlob.setUpdated(true);
   remoteBlobs.add(currentBlob);
   remoteBlobs.setUpdated(true);
   System.out.println("added blob");
}
else {
   //blob already existed
   VisualBlob oldBlob = (VisualBlob)(remoteBlobs.vec()).get(blobIndex);
   String oldCreatedDate = oldBlob.getDateCreated();
   long oldCreatedMillis = oldBlob.getCreatedMillis();
   oldBlob.setLastSeen(System.currentTimeMillis());
   currentBlob.setLastSeen(System.currentTimeMillis());

   //tests whether the current blob is equivalent to
   //the previously seen blob
   if (currentBlob.equals(oldBlob)) {
//if so, do nothing
   }
   else {
//if not, update remoteBlobs with new information
currentBlob.setDateCreated(oldCreatedDate);
currentBlob.setCreatedMillis(oldCreatedMillis);
currentBlob.setUpdated(true);
remoteBlobs.setBlobAt(blobIndex, currentBlob);
remoteBlobs.setUpdated(true);
   }
}



   }
   catch (Exception e) {
dataSock.close();
System.out.println("Connection closed.  Exception: " + e);
break;
   }
}

   }
}
catch (Exception e) {

   System.out.println("Error in main: " + e);
}


   }

   //parses String received from cameras and creates a
   //VisualBlob based on the data
   public static VisualBlob parseBlobFromCameraString(String s) throws Exception{
VisualBlob tempBlob = null;

StringTokenizer sTok = new StringTokenizer(s, " ");
String positionAsWord = sTok.nextToken();
String name = sTok.nextToken();
String unknownFloat = sTok.nextToken();
String xPos = sTok.nextToken();
String yPos = sTok.nextToken();
String height = sTok.nextToken();

/**
  System.out.println("name:" + name);
  System.out.println("ufo:" + unknownFloat);
  System.out.println("xpos:" + xPos);
  System.out.println("ypos:" + yPos);
  System.out.println("height:" + height);
**/
float ufo = Float.parseFloat(unknownFloat);
float x = Float.parseFloat(xPos);
float y = Float.parseFloat(yPos);
float z = Float.parseFloat(height);

tempBlob = new VisualBlob(name, ufo, x, y, z);
return tempBlob;
   }

}
```

RoomInfoServer.java

```
package smartroom;

import javax.swing.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import java.rmi.*;
import java.util.*;
import java.sql.*;

public class RoomInfoServer {

   //central server that constantly checks the remote represenation of
   //the room state.  when that state has changed, it makes the appropriate
   //updates to the database.

   public static boolean clientConnected;
   public static Socket clientSocket;
   public static PrintWriter clientWriter;
   public static ServerSocket serverSocket;
   public static String currentState;
   public static BlobVect blobs;
   public static SmartroomConnection postgresConnection;
   public static boolean markedForAnotherUpdate = false;

   //constructor
   public RoomInfoServer() throws IOException {

clientConnected = false;
clientSocket = null;
clientWriter = null;
serverSocket = null;
currentState = null;
postgresConnection = null;
try {

   String driver = "org.postgresql.Driver";
   Class.forName(driver);
   InetAddress localInetAddr = InetAddress.getLocalHost();
   String host = localInetAddr.getHostName();
   String url = "jdbc:postgresql://" + host + ":5432/smartroom";
   String user = "start";
   String password = "";
   //makes connection to database
   postgresConnection  = new SmartroomConnection(driver, url, user, password);
   blobs = new BlobVectImpl();
   loadRoomState();
   Naming.rebind("Room Blobs", blobs);
}
catch (Exception e) {
   System.out.println("Error in RoomInfoServer constructor: " + e);
}

   }

   //loads initial room state from locked version in database
   public void loadRoomState() throws SQLException {
try {
   ResultSet returnedSet = postgresConnection.loadRoomState();
   int rowCount = 0;

   while(returnedSet.next()) {
rowCount ++;
String name = returnedSet.getString(1);
String createdTime = returnedSet.getString(2);
String createdMillis = returnedSet.getString(3);
String modifiedTime = returnedSet.getString(4);
String modifiedMillis = returnedSet.getString(5);
String unknownFloat = returnedSet.getString(6);
String xPos = returnedSet.getString(7);
String yPos = returnedSet.getString(8);
String zPos = returnedSet.getString(9);



if (xPos.equals("OUTSIDE ROOM")) {
   unknownFloat = "-1";
   xPos = "-200";
   yPos = "-200";
   zPos = "-200";
}

String stateLine = name + "" + createdTime + "" + createdMillis + "" + modifiedTime + "" + mod-
ifiedMillis + "" + unknownFloat + "" + xPos + "" + yPos + "" + zPos;
loadStateLine(stateLine);

   }

   if (rowCount == 0) {
```

```
currentState = "No State Information Available\n";
   }

}
catch (Exception err) {
   System.out.println("Exception while trying to load room state.  " + err.toString());
}
   }


   //parses a String containing blob information, creates that blob
   //then loads it into the current (remotely accesible) state
   public void loadStateLine(String currentStateLine) throws RemoteException, SQLException {
try {
   StringTokenizer sTok = new StringTokenizer(currentStateLine, "");
   String name = sTok.nextToken();
   String dateAndTimeCreated = sTok.nextToken();
   String cMillis = sTok.nextToken();
   String dateAndTimeModified = sTok.nextToken();
   String mMillis = sTok.nextToken();
   String unknownFloat = sTok.nextToken();
   String xPos = sTok.nextToken();
   String yPos = sTok.nextToken();
   String height = sTok.nextToken();

   long createdMillis = Long.parseLong(cMillis);
   long modifiedMillis = Long.parseLong(mMillis);

   float ufo = Float.parseFloat(unknownFloat);
   float x = Float.parseFloat(xPos);
   float y = Float.parseFloat(yPos);
   float z = Float.parseFloat(height);

   VisualBlob currentBlob = new VisualBlob(dateAndTimeCreated, createdMillis, dateAndTime-
Modified, modifiedMillis, name, ufo, x, y, z);
   blobs.add(currentBlob);
   if (!(postgresConnection.symbolExists(name))) {
postgresConnection.addSymbol(name);
   }
}
catch (Exception e) {
   System.out.println("Error in loadStateLine: " + e.toString());
}
   }

   //iterates through each blob in the remotely accessible version of
   //the current state.  if a particular blob has been updated, it loads
   //the updated version into the locked version of the current state.
   //as the locked current state changes, the old current state is written
   //into permanent storage
   public static void writeNewState() {
try {

   //   blobs.setLock(true);
   Vector b = blobs.vec();
   long currentTime = System.currentTimeMillis();
   for (int i=0;i<b.size();i++) {

VisualBlob currentBlob = (VisualBlob)b.get(i);
long oldMillis = currentBlob.getModifiedMillis();
//change made 7/05/02 currentBlob.setModifiedMillis(currentTime);

if (!(postgresConnection.symbolExists(currentBlob.getName()))) {
   postgresConnection.addSymbol(currentBlob.getName());
}

if (currentBlob.stateUpdated()) {

if (!(blobInsideRoom(currentBlob))) {

   setBlobOutsideRoom(currentBlob);
   postgresConnection.insertBlobIntoTempLastUpdates(currentBlob);
   blobs.remove(currentBlob);
   blobs.setUpdated(true);
   markedForAnotherUpdate = true;
}
else  {
  postgresConnection.insertBlobIntoTempLastUpdates(currentBlob);
}

   currentBlob.setUpdated(false);

}
else {

  if (!(blobInsideRoom(currentBlob))) {
   setBlobOutsideRoom(currentBlob);
  }
   currentBlob.setModifiedMillis(oldMillis);
   postgresConnection.insertBlobIntoTempCurrentData(currentBlob);
}
   }
   //System.out.println("Updating set of most recent changes: " + System.currentTimeMillis());
   postgresConnection.updateLastUpdates();
   //System.out.println("Finished updating set of most recent changes: " + System.current-
TimeMillis());
   // System.out.println("Copying updates to final storage: " + System.currentTimeMillis());
   postgresConnection.updateRoomData();
   // System.out.println("Finished copying updates to final storage: " + System.currentTimeMil-
lis());
   // System.out.println("Updating current data: " + System.currentTimeMillis());
   postgresConnection.updateCurrentData();
   // System.out.println("Finished updating current data: " + System.currentTimeMillis());    //
blobs.setLock(false);
}
catch (Exception e) {
   System.out.println("Error in writeNewState: " + e);
}
   }

   //returns true if the blob is inside the room
   public static boolean blobInsideRoom(VisualBlob b)  throws RemoteException {
long lastSeenThreshSeconds = 30;
long lastSeenThreshMillis = lastSeenThreshSeconds * 1000;
long removalThreshMillis = (lastSeenThreshSeconds + 15) * 1000;
long currentTime = System.currentTimeMillis();
long diff = currentTime - b.getLastSeen();
     double diffSeconds = diff / 1000;
if (diff > removalThreshMillis) {

}

if ( diff > lastSeenThreshMillis) {
   System.out.println("HAVEN'T SEEN BLOB NAMED: " + b.getName() + " in " + diffSeconds
+ " seconds ago.LAST SEEN: " + b.getLastSeen() + " CURRENT TIME: " + currentTime);
   return false;
}
else {
   return true;
}
   }

   //retruns a String representation of all the blobs in a BlobVect
   public static String getAllBlobVectString(BlobVect vi) throws RemoteException {
Vector v = vi.vec();
String out = "";
VisualBlob rb = null;
for (int i=0;i<v.size();i++) {
   rb = (VisualBlob)v.get(i);
   out = out + rb.toString();
}
return out;
   }

   //returns a String representation of all the updated blobs in a BlobVect
   public static String getUpdatedBlobVectString(BlobVect vi) throws RemoteException {
Vector v = vi.vec();
String out = "";
VisualBlob rb = null;
for (int i=0;i<v.size();i++) {
   rb = (VisualBlob)v.get(i);
   if (rb.stateUpdated()) {
out = out + rb.toString();
rb.setUpdated(false);
   }
}
return out;
   }

   //returns a String representation of the current (remotely accessible)
   //room state
   public String getRoomState() throws RemoteException {
return getAllBlobVectString(blobs);
   }


   //MAIN
   //waits for connections and checks to see if the remotely accessible state
   //has been updated
   public static void main(String[] args) throws IOException {
try {
   RoomInfoServer rInfo = new RoomInfoServer();


   System.out.println("Awaiting connections");
   while (true) {
Thread.sleep(20);
if (blobs.stateUpdated()) {
   writeNewState();
   if (markedForAnotherUpdate) {
    markedForAnotherUpdate = false;
    writeNewState();

   }
```

```
    else {
    blobs.setUpdated(false);
    }
}
else {
    Vector b = blobs.vec();
    for (int i=0;i<b.size();i++) {
VisualBlob currentBlob = (VisualBlob)b.get(i);

if (!(blobInsideRoom(currentBlob))) {
    {
 setBlobOutsideRoom(currentBlob);
    }
}

    }
}
//allowing timer to continue ticking
    }

}
catch (Exception e) {
    System.out.println ("Error in main: " + e);
}
    }

    //marks a blob as being outside the room
  public static void setBlobOutsideRoom(VisualBlob b) throws RemoteException {
    b.setOutsideRoom();
    b.setUpdated(true);
    blobs.setUpdated(true);
  }
}
```

DataSmoother.java

```
package smartroom;

import java.io.*;
import java.util.*;
import java.lang.*;

public class DataSmoother {

   private static HashMap infoHash = new HashMap(100);
   private static HashMap glitchHash = new HashMap(100);
   private static HashMap equivalenceHash = new HashMap(100);
   private static HashMap linesHash = new HashMap(100);
   private static HashMap lastSuspectedGlitchHash = new HashMap(100);
   private static float errorThresh = (new Float(0.15)).floatValue();
   private static String[] linesArray = new String[500];
   private static int lineCount = 0;
   private static int resetSeconds = 3;
   private static long resetMillis = resetSeconds * 1000;
   private static int glitchTimeoutSeconds = 5;
   private static long glitchTimeoutMillis = glitchTimeoutSeconds * 1000;
   private static boolean setLastGlitch = false;

   private static boolean verbose = false;
   private static boolean superVerbose = false;

   public static void main(String[] args) throws IOException {
String verb = args[0];
String superVerb = args[1];
String filename = args[2];
if (verb.equals("true")) {
   verbose = true;
}
if (superVerb.equals("true")) {
   superVerbose = true;
}
boolean done = false;
FileReader fReader = new FileReader(filename);
BufferedReader bufReader = new BufferedReader(fReader);


while (!done) {


   String currentInput = bufReader.readLine();
```

```
    StringTokenizer sTok = new StringTokenizer(currentInput, "");
    String name = sTok.nextToken();
    String dateAndTimeCreated = sTok.nextToken();
    String createdMillis = sTok.nextToken();
    String dateAndTimeModified = sTok.nextToken();
    String modifiedMillis = sTok.nextToken();
    String unknownValue = sTok.nextToken();
    String xVal = sTok.nextToken();
    String yVal = sTok.nextToken();
    String zVal = sTok.nextToken();

    float x = Float.parseFloat(xVal);
    float y = Float.parseFloat(yVal);
    float z = Float.parseFloat(zVal);

    long modMillis = Long.parseLong(modifiedMillis);

    String outputLine = currentInput;

    if (linesHash.isEmpty()) {
if (verbose) {System.out.println("********CLEARING GLITCH MEMORY*******"); }
//equivalenceHash.clear();

glitchHash.clear();
for (int i = 0; i < lineCount; i++) {
    if (superVerbose) {    System.out.println("CLEARED EQ HASH: Printing Line: " + i); }
    System.out.println(linesArray[i]);
}

lineCount = 0;

    }

    linesArray[lineCount] = outputLine;

    if (equivalenceHash.containsKey(name)) {
// this blob is really a previously seen blob
// so update the previously seen blob's info with the
// new information
String realName = (String)equivalenceHash.get(name);
outputLine = nameSubstitute(realName, currentInput);
linesArray[lineCount] = outputLine;


InfoContainer realContainer = (InfoContainer)infoHash.get(realName);
realContainer.updateInfo(modMillis, x, y, z);
```

```
lastSuspectedGlitchHash.put(name + "" + realName, new Long(modMillis));
   }

   else {



if (infoHash.containsKey(name)) {
   //already in hash
   InfoContainer currentContainer = (InfoContainer)infoHash.get(name);
   long timeDiff = currentContainer.timeDiff(modMillis);

   float expX = currentContainer.expectedX(modMillis);

   float expY = currentContainer.expectedY(modMillis);

   if (superVerbose) {System.out.println("Name: " + name + "Time Seen: " + modMillis + "Time
Since Last Seen: " + timeDiff);
   System.out.println("XVelocity: " + currentContainer.getXVel() + "Expected X Position: " +
expX + "Actual X Position: " + x);
   System.out.println("YVelocity: " + currentContainer.getYVel() + "Expected Y Position: " +
expY + "Actual Y Position: " + y + "\n");
   }
   float xErr = abs(x - expX);
   float yErr = abs(y - expY);



   if ((xErr > errorThresh) || (yErr > errorThresh)) {
if (superVerbose) {    System.out.println("ERROR EXCEEDS THRESHHOLD: XErr = " + xErr
+ "YErr = " + yErr); }
   }


   currentContainer.updateInfo(modMillis, x, y, z);
   infoHash.put(name, currentContainer);
}
else {
   //first time seen

   InfoContainer newInfo = new InfoContainer(modMillis, x, y, z);
   infoHash.put(name, newInfo);
   if (verbose) { System.out.println("First Time Seen******"); }
}

testForGlitch(name, x, y, modMillis);


   }


   lineCount ++;

   if (!(bufReader.ready())) {
done = true;
   }
   if (verbose) {    System.out.println("lineCount: " + lineCount); }

   if (superVerbose) {    displayAllStoredLines(); }
   tidyStoredLines(modMillis);

}

//finish printing lines
for (int i=0;i<lineCount;i++) {
   if (superVerbose) {    System.out.println("END: Printing Line: " + i); }
   System.out.println(linesArray[i]);
}
   }


   public static void testForGlitch(String name, float x, float y, long modMillis) {
int newGlitchCount = 0;
Set hashKeys = infoHash.keySet();
Iterator keyIter = hashKeys.iterator();
while (keyIter.hasNext()) {
   String currentKey = (String)keyIter.next();

   String glitchKey = name + "" + currentKey;

   if (!(name.equals(currentKey))) {
InfoContainer checkedContainer = (InfoContainer)infoHash.get(currentKey);

float expectedX = checkedContainer.expectedX(modMillis);
float expectedY = checkedContainer.expectedY(modMillis);
float xErr = abs(x - expectedX);
float yErr = abs(y - expectedY);
if (superVerbose) {    System.out.println(xErr + "" + yErr); }
if ((xErr < errorThresh) && (yErr < errorThresh)) {
  // possibility of equivalence... this blob is on the same trajectory as the checked blob
  storeLine(glitchKey, lineCount);

  if (verbose) {System.out.println("****IS " + name + " REALLY " + currentKey + "???****");
}

  lastSuspectedGlitchHash.put(glitchKey, new Long(modMillis));
  if (glitchHash.containsKey(glitchKey)) {
Integer glitchCount = (Integer)glitchHash.get(glitchKey);
newGlitchCount = glitchCount.intValue() + 1;
glitchHash.put(glitchKey, new Integer(newGlitchCount));
  }
  else {

glitchHash.put(glitchKey, new Integer(newGlitchCount));

  }


}

/**
  else if ((modMillis - checkedContainer.getLastModifiedMillis()) > resetMillis) {
  // weaker test for equivalence
  storeLine(glitchKey, lineCount);

  if (verbose) {System.out.println("****IS " + name + " REALLY " + currentKey +
"???****WEAKER TEST"); }

  lastSuspectedGlitchHash.put(glitchKey, new Long(modMillis));


  if (glitchHash.containsKey(glitchKey)) {
  Integer glitchCount = (Integer)glitchHash.get(glitchKey);
  newGlitchCount = glitchCount.intValue() + 1;
  glitchHash.put(glitchKey, new Integer(newGlitchCount));
  }
  else {

  glitchHash.put(glitchKey, new Integer(newGlitchCount));

  }


  }
```

```
**/


   if ((newGlitchCount >= 2) && ((modMillis - checkedContainer.getLastModifiedMillis()) >
resetMillis)) {
// equivalence decided
// two straight glitches and it's been more than resetMillis milliseconds since the checked blob
was last seen

if (verbose) {    System.out.println("********" + name + " IS " + currentKey + "********"); }

equivalenceHash.put(name, currentKey);
changeLines(glitchKey);
clearStoredLines(glitchKey);
glitchHash.remove(glitchKey);
   }


   }
}
   }


   public static void changeLines(String glitchKey) {
StringTokenizer sTok = new StringTokenizer(glitchKey, "");
String oldName = sTok.nextToken();
String newName = sTok.nextToken();

Vector lines = (Vector)linesHash.get(glitchKey);
for (int i=0;i<lines.size();i++) {
   int currentLine = ((Integer)lines.get(i)).intValue();
   //retrieving old line
   String oldLine = linesArray[currentLine];
   if (superVerbose) {    System.out.println("Current index: " + currentLine + "Corresponding line:
" + oldLine); }
   //modifying line
   linesArray[currentLine] = nameSubstitute(newName, oldLine);
}
   }


   public static void storeLine(String glitchKey, int lineCnt) {

if (linesHash.containsKey(glitchKey)) {
   Vector lines = (Vector)linesHash.get(glitchKey);
   lines.add(new Integer(lineCnt));
   linesHash.put(glitchKey, lines);
}
else {
   Vector lines = new Vector();
   lines.add(new Integer(lineCnt));
   linesHash.put(glitchKey, lines);
}
   }

   public static void clearStoredLines(String glitchKey) {
linesHash.remove(glitchKey);
if (superVerbose) { System.out.println("removed key: " + glitchKey); }
   }

   public static void displayAllStoredLines() {
Set hashKeys = linesHash.keySet();
Iterator keyIter = hashKeys.iterator();
while (keyIter.hasNext()) {
   String key = (String)keyIter.next();
   System.out.print("Key: " + key + " Lines:");
   Vector lines = (Vector)linesHash.get(key);
   for (int i=0;i<lines.size();i++) {
int lineNum = ((Integer)lines.get(i)).intValue();
System.out.print(" " + lineNum);
   }
   System.out.println();
}
   }

   public static void tidyStoredLines(long modMillis) {
Set hashKeys = lastSuspectedGlitchHash.keySet();
Iterator keyIter = hashKeys.iterator();
while (keyIter.hasNext()) {
   String key = (String)keyIter.next();
   long glitchTime = ((Long)lastSuspectedGlitchHash.get(key)).longValue();
   long timeSinceGlitch = modMillis - glitchTime;
   if (superVerbose) {    System.out.println("Key: " + key + "Time Since Glitch: " + timeSinceG-
litch); }
   if (timeSinceGlitch > glitchTimeoutMillis) {
linesHash.remove(key);
if (superVerbose) {System.out.println("Glitch: " + key + " timed out"); }
   }
}
   }
   public static String nameSubstitute(String newName, String input) {
StringTokenizer sTok = new StringTokenizer(input, "");
String name = sTok.nextToken();
String dateAndTimeCreated = sTok.nextToken();
String createdMillis = sTok.nextToken();
String dateAndTimeModified = sTok.nextToken();
String modifiedMillis = sTok.nextToken();
String unknownValue = sTok.nextToken();
String xVal = sTok.nextToken();
String yVal = sTok.nextToken();
String zVal = sTok.nextToken();

String outString = newName + "" + dateAndTimeCreated + "" + createdMillis + "" + dateAnd-
TimeModified + "" + modifiedMillis + "" + unknownValue + "" + xVal + "" + yVal + "" + zVal;
return outString;
   }

   public static float abs(float f) {
if (f < 0) {
   f = f * -1;
}
return f;
   }

}
```

DataSmootherSimple.java

```
package smartroom;

import java.io.*;
import java.util.*;
import java.lang.*;

public class DataSmootherSimple {

   private static HashMap infoHash = new HashMap(100);
   private static HashMap glitchHash = new HashMap(100);
   private static HashMap equivalenceHash = new HashMap(100);

   private static float errorThresh = (new Float(0.15)).floatValue();
   private static int resetSeconds = 3;
   private static long resetMillis = resetSeconds * 1000;
   private static long lastSuspectedGlitchMillis;
   private static int printLineCount = 0;
   private static boolean verbose = true;

public static void main(String[] args) throws IOException {
String filename = args[0];
boolean done = false;
FileReader fReader = new FileReader(filename);
BufferedReader bufReader = new BufferedReader(fReader);


while (!done) {
   //   System.out.println("LINECOUNT: " + lineCount);


   String currentInput = bufReader.readLine();
   //   System.out.println(currentInput);
   StringTokenizer sTok = new StringTokenizer(currentInput, "");
   String name = sTok.nextToken();
   String dateAndTimeCreated = sTok.nextToken();
   String createdMillis = sTok.nextToken();
   String dateAndTimeModified = sTok.nextToken();
   String modifiedMillis = sTok.nextToken();
   String unknownValue = sTok.nextToken();
   String xVal = sTok.nextToken();
   String yVal = sTok.nextToken();
   String zVal = sTok.nextToken();

   float x = Float.parseFloat(xVal);
```

```
    float y = Float.parseFloat(yVal);
    float z = Float.parseFloat(zVal);

    long modMillis = Long.parseLong(modifiedMillis);

    String outputLine = currentInput;


    if (infoHash.containsKey(name)) {
 InfoContainer currentContainer = (InfoContainer)infoHash.get(name);
 long timeDiff = currentContainer.timeDiff(modMillis);

    }

    if (equivalenceHash.containsKey(name)) {
 // this blob is really a previously seen blob
 // so update the previously seen blob's info with the
 // new information
 String realName = (String)equivalenceHash.get(name);
 outputLine = nameSubstitute(realName, currentInput);



 InfoContainer realContainer = (InfoContainer)infoHash.get(realName);
 realContainer.updateInfo(modMillis, x, y, z);
 lastSuspectedGlitchMillis = modMillis;
    }

    else {


 testForGlitch(name, x, y, modMillis);


    if (infoHash.containsKey(name)) {
 //already in hash
 InfoContainer currentContainer = (InfoContainer)infoHash.get(name);
  long timeDiff = currentContainer.timeDiff(modMillis);

  float expX = currentContainer.expectedX(modMillis);

  float expY = currentContainer.expectedY(modMillis);

  // System.out.println("Name: " + name + "Time Seen: " + modMillis + "Time Since Last Seen: "
 + timeDiff);
 //System.out.println("XVelocity: " + currentContainer.getXVel() + "Expected X Position: " +
expX + "Actual X Position: " + x);
 //System.out.println("YVelocity: " + currentContainer.getYVel() + "Expected Y Position: " +
expY + "Actual Y Position: " + y + "\n");

float xErr = abs(x - expX);
float yErr = abs(y - expY);



if ((xErr > errorThresh) || (yErr > errorThresh)) {
   //    System.out.println("ERROR EXCEEDS THRESHHOLD: XErr = " + xErr + "YErr = " +
yErr);
}


currentContainer.updateInfo(modMillis, x, y, z);
infoHash.put(name, currentContainer);
   }
   else {
//first time seen

InfoContainer newInfo = new InfoContainer(modMillis, x, y, z);
infoHash.put(name, newInfo);
if (verbose) { System.out.println("First Time Seen******"); }


   }




   }


   System.out.println(outputLine);

   if (!(bufReader.ready())) {
done = true;
   }


}

   }
```

```
    public static void testForGlitch(String name, float x, float y, long modMillis) {
  int newGlitchCount = 0;
  Set hashKeys = infoHash.keySet();
  Iterator keyIter = hashKeys.iterator();
  while (keyIter.hasNext()) {
    String currentKey = (String)keyIter.next();

    String glitchKey = name + "" + currentKey;

    if (!(name.equals(currentKey))) {
    InfoContainer checkedContainer = (InfoContainer)infoHash.get(currentKey);

    float expectedX = checkedContainer.expectedX(modMillis);
    float expectedY = checkedContainer.expectedY(modMillis);
    float xErr = abs(x - expectedX);
    float yErr = abs(y - expectedY);
    //    System.out.println(xErr + "" + yErr);

    if ((xErr < errorThresh) && (yErr < errorThresh)) {
// possibility of equivalence... this blob is on the same trajectory as the checked blob


if (verbose) {System.out.println("****IS " + name + " REALLY " + currentKey + "???****"); }
lastSuspectedGlitchMillis = modMillis;

if (glitchHash.containsKey(glitchKey)) {
   Integer glitchCount = (Integer)glitchHash.get(glitchKey);
   newGlitchCount = glitchCount.intValue() + 1;
   glitchHash.put(glitchKey, new Integer(newGlitchCount));
}
else {

   glitchHash.put(glitchKey, new Integer(newGlitchCount));
}



if ((newGlitchCount >= 3) && ((modMillis - checkedContainer.getLastModifiedMillis()) > reset-
Millis)) {
   // equivalence decided
   // three straight glitches and it's been more than resetMillis milliseconds since the checked blob
was last seen

  if (verbose) {    System.out.println("********" + name + " IS " + currentKey + "********"); }
  equivalenceHash.put(name, currentKey);
```

```
   glitchHash.remove(glitchKey);
}




   }

   }
}
   }

 public static String nameSubstitute(String newName, String input) {
StringTokenizer sTok = new StringTokenizer(input, "");
String name = sTok.nextToken();
String dateAndTimeCreated = sTok.nextToken();
String createdMillis = sTok.nextToken();
String dateAndTimeModified = sTok.nextToken();
String modifiedMillis = sTok.nextToken();
String unknownValue = sTok.nextToken();
String xVal = sTok.nextToken();
String yVal = sTok.nextToken();
String zVal = sTok.nextToken();

String outString = newName + "" + dateAndTimeCreated + "" + createdMillis + "" + dateAnd-
TimeModified + "" + modifiedMillis + "" + unknownValue + "" + xVal + "" + yVal + "" + zVal;
return outString;
   }

   public static float abs(float f) {
if (f < 0) {
   f = f * -1;
}
return f;
   }

}
```