MIT Campus Walkthrough

Michael B. Craig

August 24, 2005

Contents

Chapter 1

Introduction and Background

The MIT Campus Walkthrough project (MITWalkthru) aims to allow highly visual, virtual "walkthroughs" of the MIT campus. MITWalkthru aims to help users visualize paths they may take between different locations on the MIT campus. The information needed by MITWalkthru, including 2-D and 3-D geometry and connectivity of the MIT campus, is provided by the Building Model Generation (BMG) Project[2].

1.1 The Building Model Generation Project

The BMG Project includes numerous tools that together build a rich description of the MIT campus. The only input data to BMG are 2-D floorplans (provided by MIT's Department of Facilities), and a "base map" of the outside areas of campus. From this information, BMG obtains 2-D geometry and naming information about the spaces in all of MIT's buildings, as well as information about the connectivity between the spaces. Furthermore, BMG extrapolates the floorplans into full 3-D geometry of the buildings and the base map.

Owing to the complexity and difficulty of such an undertaking, as well as to irregularities in the 2-D floorplans, the data produced by the BMG pipeline–that is, the aforementioned "numerous tools"–are not perfect. The pipeline itself can be run regularly (daily, say), so that as incremental improvements are made to the individual tools, the quality of the output improves.

In the future, BMG may provide new types of information to augment its current output. Some examples include: automatic placement of furniture in rooms, based on room type; textures and other highly specific visual information for the 3-D models (as, currently, only simple, untextured polygons are generated for these models); and "intelligent" information about landmarks or particular features of a space that may be interesting to a user.

1.2 BMG-related Projects

The ultimate goal of the BMG pipeline is to produce MIT campus information that can be used by other projects that have more tangible interest to users. The Location-Aware Active Signage project, for example, attempts to provide route-planning capabilities to users, incorporated with information about various events at MIT[7]. The project makes significant use of the 2-D geometry and connectivity information to find and display routes.

1.3 MITWalkthru

MITWalkthru is similar to the Active Signage project in that respect: it attempts to provide users with meaningful access to the data provided by BMG. The essential goal of MITWalkthru is to allow users to explore the MIT campus by virtually "walking" around the geometry information provided by BMG. This typically takes one of two forms. In the "free walk" form, the user controls an avatar and is allowed to freely navigate through the campus. In the "path walk" form, the user requests a path between two locations on campus, and then moves the avatar along this path.

Because the collective data output by BMG are quite large–with at least 113 buildings and 46,000 spaces–it is impractical to expect the casual user to have direct access to all of it. MITWalkthru takes a client/server approach, in which a central server stores the massive dataset output by BMG, and a client–the software with which the user interacts–accesses the data from the server remotely. Thus, to perform

either of the walkthroughs described, we assume that the user should only be able to "see" a small fraction of the entire campus (namely, the area immediately surrounding, or visible to, his avatar). Thus, the client and server must interact so that as the client moves, the server provides it with the relevant local information.

Thus, MITWalkthru is a solution to a specific instance of a more general problem: that of providing meaningful access, from a user's local computer–which likely has limited memory and computational ability–to a very large data set, stored remotely on a more powerful computer. Furthermore, we are most interested in this problem when it also requires access by multiple users, typically on different computers; the server, storing the data set, services these users concurrently. But MITWalkthru relies on another property of the data: that it is arranged and stored spatially, with some decent distance metric, so that "movement" in the data makes sense, and small movements in the exposed data translate to small movements in the underlying storage structure. Leverage of this property allows MITWalkthru to provide smooth "walkthroughs" of the data.

Chapter 2

Related Work

Much work has been done on "walkthrough" visualizations of virtual environments. It is no surprise, for example, that modern video games allow "free walking" of detailed 3-D environments with little trouble. MITWalkthru presents a comparative challenge because of the massive size of the MIT campus. It would be impractical, or at the least, unwieldy, to force the user to download and store data for the entire campus, when he might normally wish to use MITWalkthru for a short period of time, and from any of a variety of computers scattered around campus.

Maneesh Agrawala has done significant work on visualization of virtual environments and paths through them. In [1], he presents research on what users find most important about route maps, and describes LineDrive, a route-mapping tool which takes this research into account. In [6], he and Niederauer and others present work on visualization of architectural environments that allows users to see all floors of a building at once. Unlike these projects, MITWalkthru places less emphasis on global visualization of routes; its focus is on traversing the environment with only local information.

Chapter 3

Motivation

The ultimate goal of MITWalkthru is to allow members of the MIT community to learn about a path through campus virtually, before the path is taken physically. In other words, if the user wants to find his way around a particular building or floor on campus, or wants to find the best route between two locations on campus, he can explore either of these, in very little time, with MITWalkthru.

Furthermore, because the relevant MIT campus data set is very large, it is unreasonable to expect a user to store and manipulate it, in its entirety, on his/her local computer. Instead, we would like to store the data set on a powerful, centralized computer, and have it serve individual pieces of the data to potentially many different users. Thus, we would like a client/server architecture in which small pieces of data are served to the user over the network. Since a user is expected to only need access to a very small subset of the entire campus at any one time, such an architecture seems sufficient.

Though MITWalkthru is currently only designed to allow 2-D visualizations, its structure easily allows for extensions in the data. The most natural upgrade to MITWalkthru would be visualization of a 3-D first-person walkthrough, along with the 2-D walkthrough. Future iterations of MITWalkthru should incorporate more information as it becomes available from BMG, such as more detailed 3-D models, or landmark indications.

Chapter 4

Architecture and Design

As described, MITWalkthru is fundamentally based on a client/server architecture. The server runs on a single computer and stores all of the relevant output from the BMG pipeline. The client runs separately, on any other networked computer, and connects to the server to retrieve pieces of this information. The client only ever knows about a small subset of the server's entire data set, but employs a cache so that movement within the locally available data is always smooth.

4.1 Campus Data Structure

The data set is stored in a large connected graph structure. The vertices and the edges of the graph can be treated as generic entities for much of the discussion; specific implementations will be discussed further on. It suffices, for now, to think of each vertex v as a renderable object which has some spatial connection to the vertices to which v is connected; and every 3-D point in space is located within some vertex.

The server loads the entire graph into RAM. When serving a client, it tells the client which graph vertices and edges it should store locally in its cache. In fact, the client always stores a small, connected subset of the entire graph. As the client "moves" through this sub-graph, it updates the server regarding its position, and the server then tells the client which nodes to add to, and which nodes to remove from, its cache. The cache, then, is not automatic; it is "pushed" by the server, rather than

"pulled" by the client. In other words, the server always tells the client exactly what it should store in its cache. Thus, the server itself stores a keeps track of a mirror of the client's cache.

The size and shape of the cached sub-graph is determined by the client's *draw parameters*. The draw parameters are a way of determining, given the client's *position* in any vertex in the graph, which surrounding vertices should be drawn by the client. The draw parameters may be as simple as "every vertex no more than n edges away," and depend upon the specific implementation of the graph. The *visible set* for any position, then, is that which consists of the vertex v containing the position, along with the vertices given by the draw parameters for that position.

We make a strict requirement, based on the visible set, about what cached set the server must tell the client to store. First, given the vertex v in which the client's position is located, the server should tell the client to always cache a superset of the visible set. Second, for every vertex v' connected to v by one edge, the server should also tell the client to cache v''s visible set. In fact, the bigger the cache, the better; but these requirements, at least, ensure that if the client is not moving at a rate of more than one edge per cache-update message, then the client will always store the entire visible set.

Of course, it certainly might occur that the client moves very quickly into vertices for which it does not have the entire visible set cached–the client must also store a *visible set available* (VSA) flag indicating whether or not the entire visible set is cached. But it is always the case that, as long as the client tells the server its position (and its current vertex) often enough, the server will send back the proper cache updates in time.

4.2 Event-Driven Architecture

Both the client and the server work on an *event-driven* architecture. Essentially this means that, by default, they do nothing, but when *events* occur, they then perform the necessary work. There are only four crucial events which underlie the system:

user input to the client (e.g. from the mouse or keyboard); a redraw request to the client; a cache update to the client, from the server; and a position update to the server, from the client. We examine each event, and the work necessary when each occurs, below.

4.2.1 Client: User Input Event

The *user input event* occurs when the client's user performs some action with the keyboard or mouse, which is recognized by the client's user interface. For example, pressing the "up arrow" key may indicate that the client should move the position forward.

When this event occurs, the client first calculates the updated position, and determines if it lies in a different vertex (from that of the original position), which we call v'. If it does, and if v' is available in the client's cache, then the client sends a *position update* message to the server (expecting an eventual *cache update* message in response); if v' is not available in the client's cache, no action is taken: we know that a *position update* message has already been sent to the server, and a *cache update* message will be received in time. If the updated position does not in fact lie in a different vertex, no special action is taken.

This handler employs an optimization, though: if the client has sent a *position update* message in the past and has not received the corresponding *cache update* message, it need not immediately send another *position update*. Instead, it need only set a flag indicating that another *position update* should be sent after the next *cache update* is received. This way, if the client traverses numerous edges while waiting for a *cache update*–an unfortunate, but possible condition–it need only send a *position update* for the last vertex it reached.

Finally, if movement did occur, a redraw is scheduled for the client.

4.2.2 Client: Redraw Event

The *redraw event* occurs when some other module of the client has requested that it perform a redraw of its data. This essentially occurs as a result of one of the other three event's handlers.

When this event occurs, the current visible set is retrieved and drawn. No other special action needs to be taken.

4.2.3 Client: Cache Update Event

The *cache update event* occurs when the client receives a *cache update* message from the server. A cache update message consists of two lists: names of vertices that the client should remove from its cache; and vertices that the client should add to the cache.

The client, then, simply performs these relevant removals and additions. If the VSA flag indicated that the visible set was not fully available, then the visible set is re-calculated, and a redraw event is triggered.

4.2.4 Server: Position Update Event

The *position update event* occurs when the server receives a *position update* message from the client. A position update message consists of the vertex in which the client's user is located.

Upon receiving this message, the server re-calculates the client's cached set of vertices (again, based on the draw parameters). Call the new cache C', where the old one is C. The server compares C with C' to determine the sets of vertices, V_r and V_a, such that $C' = C - V_r + V_a$. In other words, it determines what vertices should be removed from the old cache, and which should be added to it, to form the new cache. Finally, the server sends a *cache update* message to the client; this consists of the names of the vertices in V_r, and the full data of the vertices in V_a.

4.3 Specific Graph Implementation

It is clear that the essential architecture, as described, is independent of the specific meaning for the graph's vertices and edges. A variety of specific choices might be tested, without significant change to either the client or server.

For our purposes, the 2-D space and portal information, as output from the BMG pipeline, was used: the spaces correspond to vertices, and the portals correspond to edges. The draw parameters for any vertex v consist of merely a number n, indicating all vertices not more than n edges away from v. In other words, the visible set consists of all spaces that the user might reach through some limited number of portal traversals.

This implementation has a number of benefits: the graph is straightforward to build from the BMG pipeline's output; the vertices have distinct, meaningful names; and the visible set generally has direct, relevant meaning to the user.

Chapter 5

Implementation

Here we describe the specific implementation choices used. Where appropriate, these choices are contrasted with other, competing options.

5.1 Basic Network Protocol

Before the server and client implementations are discussed, it is important to understand the lowest layer of the network protocol that they use to communicate. TCP/IP is used, and data are sent in individual, typed packets, or "messages." Each message begins with a header, containing simply its type and its length, and a payload, whose format is type-specific.

It might seem more natural to use UDP instead of TCP, since the main transfer of data occurs in a stream-like fashion, as described below; however, cache updates in particular need to be sent reliably to the client. Using an underlying protocol other than TCP would severely complicate the implementation, without any clear benefits.

5.2 Server Implementation

The structure of the server was kept as simple as possible. It has two main parts: the data structure storing the campus data, and the client connection handlers. The client connection handlers access the campus data structure, to provide only the necessary

information to clients. Because the campus data structure is immutable after it has been initialized, its design could be kept nearly completely separate from that of the client connection handlers.

5.2.1 Campus Data Structure

As mentioned, the campus data set is stored as a read-only graph of spaces linked by portals. Since (indoor) spaces have human-meaningful names, they can be accessed directly by them; once a space is located, its adjacent spaces can be located in the usual way.

Because the graph is immutable, common searches such as depth-first or breadth-first search must employ auxiliary maps (typically hash tables) to identify "marked" spaces. But this immutability means that multiple threads can access the graph concurrently without a problem.

5.2.2 Client Connection Handlers

Each client connection is handled by a single thread. Many such threads, then, may exist at any time. Each thread operates in a very simple "state machine" which dictates the server's behavior in a highly modularized way. Pseudo-code of the main loop for the thread illustrates this:

```
1  state ← initialState
2  while TRUE
3        do msg ← READ-MESSAGE()
4           state ← state.HANDLE-MESSAGE(msg)
```

Once a message is read from the client, it is handled by the current state's message handler. This message handler likely sends a message back to the client. The state machine is updated, depending on the current state, and the message received; thus we have a simple state machine determining the behavior of the connection handler. Note that our pseudo-code does not illustrate failures, which may occur either when

reading a message or when handling a message. In either case, the loop is broken, and the connection to the client is closed immediately (though in future versions, we may wish to at least inform the client of the reason for a disconnection).

Given that the only messages mentioned previously are *position update* and *cache update* messages, it might seem unnecessary to build such a generalized message-handling loop. But it is important to realize that there are numerous other message types which are not crucial to the core design: for example, choosing an initial space, setting the draw parameters, sending errors, and resetting the server, all require different messages, and thus different states to handle these messages.

Note, finally, that there is one extra "server socket" thread, which simply accepts connections from clients and spawns them off into the client-handling threads described.

5.3 Client Implementation

There are two main parts to the client, as well: the user interface, and the connection handler. But unlike with the server, the two parts cannot be kept completely separate. It is often the case that interaction from the user interface compels the connection handler to send messages to the server, and messages received on the connection should change the behavior of the user interface. Given the success of the server's state machine, however, both of these components are modeled as similar state machines.

5.3.1 Connection Handler

The connection handler thread on the client side runs in a loop very similar to the server's. But because the thread may be interrupted (by the user interface) to send messages to the server, we make an exception when in the special state *idleState*. Instead of waiting to read a message, when in this state, the thread sleeps until it is woken by the user interface. When the user interface wakes the thread, it provides a functor for the thread to run as soon as it has awoken, and the next state for it to change to, before it waits for a message from the server. The functor typically sends

a message to the server. More simply: when the user interface wants to wake the connection thread from its idle state, it first wants the client to send a message to the server, and then be in the proper state to handle the reply from the server.

5.3.2 User Interface

The user interface is also based on a state machine. Each "state" contains methods to handle user input, and to draw to the screen. Thus we achieve the same modularization of function that we did with the connection handlers. It is also important to realize, though, that the user-interface state may be changed not only from within a user-interface method, but by the connection handler. Care is taken in the implementation to avoid synchronization problems.

5.4 Extended Display Features

5.4.1 Cache Behavior Visualization

Rather than just displaying the visible set, our implementation provides a more complete visualization: it also displays the rest of the cache, the spaces that have most recently been removed from the cache, and the spaces in an "extended" portion of the graph, which the client would not normally store. Each of these distinct sets is displayed in a different color, so that the user can track the cache behavior exactly.

We should note, however, that in order to accomodate these visualizations, some minor modifications needed to be applied to the overall design. In particular, for the client to view the "extended" set of spaces, the server sends them along with every *cache update* message. In other words, these spaces are not cached by the client; the client keeps them stored only until it receives a new *cache update* message, when it replaces them with the new set. Clearly, this is far from efficient. But since the "extended" set is merely provided for visualization purposes, and would not be expected for inclusion in a production version of the software, this simpler approach was chosen.

5.4.2 Free Movement

A further extended feature is provided in which the user can move freely around a floor, without being bound by space boundaries. In other words, movement is no longer restricted to occur across edges in the graph structure. When the user starts a walkthrough on a building's floor, and then begins "free movement," he/she can directly navigate to any space on the floor. If the user moves outside of any space, the entire floor will be shown (in gray, as a normal part of the "extended" set described above). When the user moves back into a space, normal behavior resumes.

To accomodate this free movement, the server was changed to recognize *position update* messages which do not specify a vertex, but which do specify a 3-D position. Upon receiving such a message, the server scans the client's current floor for a space in which the 3-D position lies. If no such space exists, all spaces in the floor are sent back as an "extended" set to the client. If such a space does exist, the server resumes normal behavior. Thus, free movement is not expected to work as responsively as normal movement, but it is provided for extra flexibility in visualization.

5.5 Language Choice

The entire MITWalkthru project is written in Java. Specifically, the code relies on the J2SE platform, version 1.4.2 or higher. The primary reason for choosing Java was to facilitate delivery of the client application to the user; J2SE 1.4.2 affords a relatively easy way to load the applet from a public WWW page. Java 1.4 also provides relative ease for working with threads, network sockets, and synchronization; because we have chosen a synchronous network protocol, blocking sockets handled by threads are a natural choice for implementation, and very easily manipulated with Java.

Other possible choices for the client that were rejected (at least for the time being) include a Flash application, and a combination VRML and dynamic HTML WWW site. Flash was rejected because of the author's unfamiliarity with it, and because it is a proprietary platform. VRML and DHTML were rejected because of the relative unpopularity (and dearth) of VRML plug-ins. Also, it was unclear whether or not

VRML and DHTML would afford the flexibility needed for the client. But, the emergence of X3D[8] may make this a more viable option in the future.

5.6 Security

The MITWalkthru server and client are both designed to be run only on computers located at MIT. At present, the network traffic is not protected from eavesdroppers (by encryption or any other means), but the server only accepts connections from MIT's IP addresses.

5.7 3-D Graphics and Java

One reason that 3-D walkthroughs have been avoided for the time being is the lack of a suitable toolkit to render 3-D graphics with Java. While JOGL[4], a Java binding for the OpenGL API, is likely to be the eventual choice for MITWalkthru, it has several problems: it does not run on all platforms supported by Java; and it depends on OpenGL drivers, so it cannot be used in an applet–instead, Java Web Start, which is significantly more unwieldy to the casual user, must be employed.

Java3D, a more standardized API, is known for having unfortunately buggy implementations and being difficult to use. Furthermore, Java3D is not a standard part of J2SE distributions, making it another impediment to users who have not already installed it.

Finally, software-only APIs such as jGL[3] avoid the system-dependency problems of JOGL and Java3D, allowing them to be used in applets without issue. But because they do not make use of hardware acceleration, and in fact must be written entirely in Java, there is significant doubt about how well they would perform for anything but very simplistic 3-D models.

Chapter 6

Results

MITWalkthru successfully allows visual navigation of the 2-D data output from the BMG pipeline without overburdening the user's computer. Smooth walkthroughs can be achieved, and the data can be thoroughly investigated at the user's will. Moreover, the extended visualization allows the user to see exactly how the client/server caching behavior works.

MITWalkthru has also been useful in exposing problems in the data produced by the BMG pipeline. Missing portals and missing spaces become immediately apparent using MITWalkthru; hopefully it can thus aid as a tool for improving the pipeline.

Three screenshots are included to illustrate the look of MITWalkthru. Figure 6-1 is an example showing all of the different cache visualization elements: visible set spaces, cached spaces, just-removed spaces, extended spaces, and a space with a vertical portal.

Figure 6-2 shows a zoomed-out view of an entire floor; its particular topology implies that most of it will be cached, for most of the rooms.

Finally, figure 6-3 shows an example of "free movement," in which all of the spaces are grayed out.

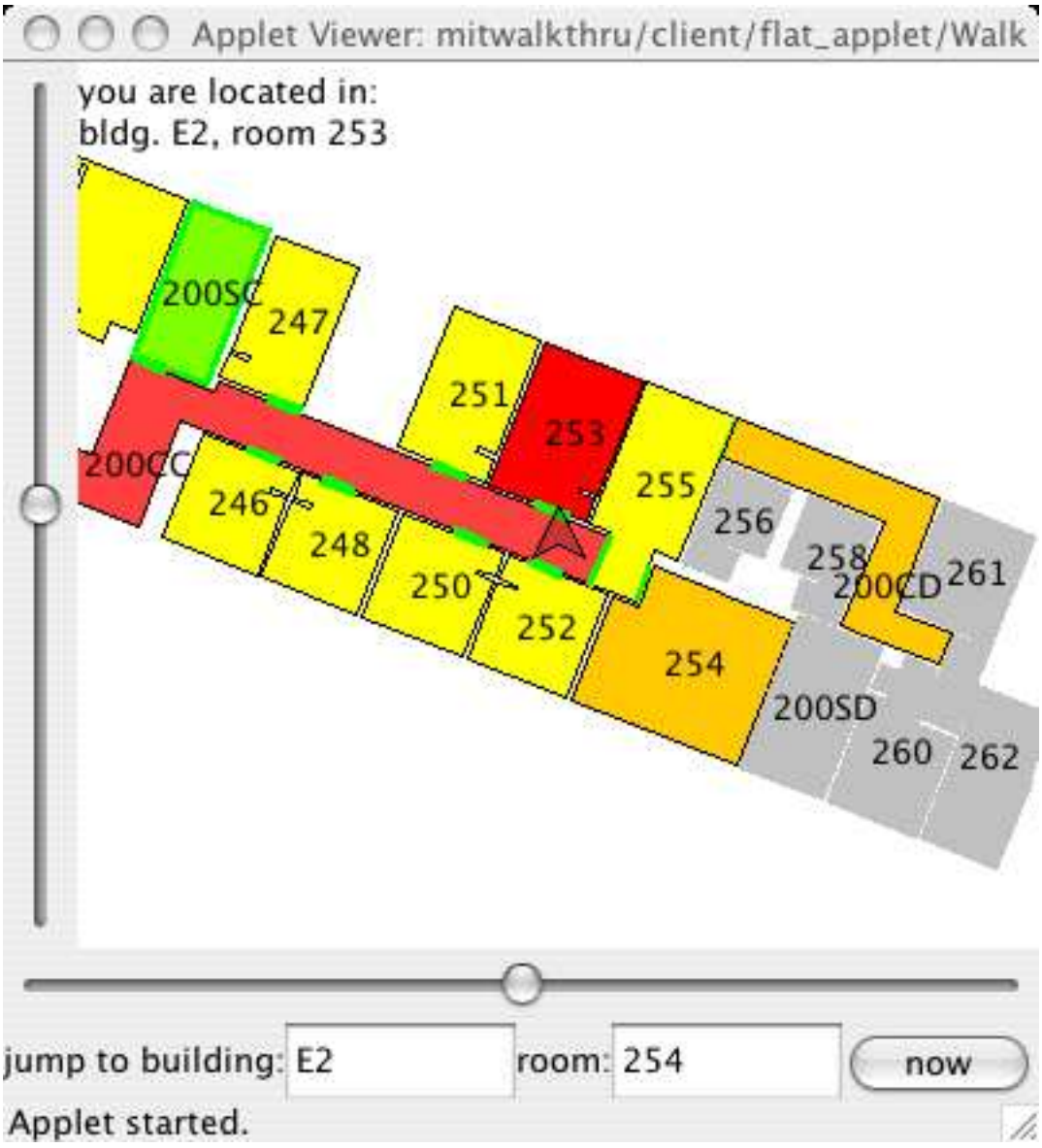

Figure 6-1: An example screenshot: the current space is shown in bright red; the visible set is unsaturated red; cached spaces are yellow; spaces just removed from the cache are orange; spaces in the "extended" set are gray; and the space with the vertical portal has a green overlay.

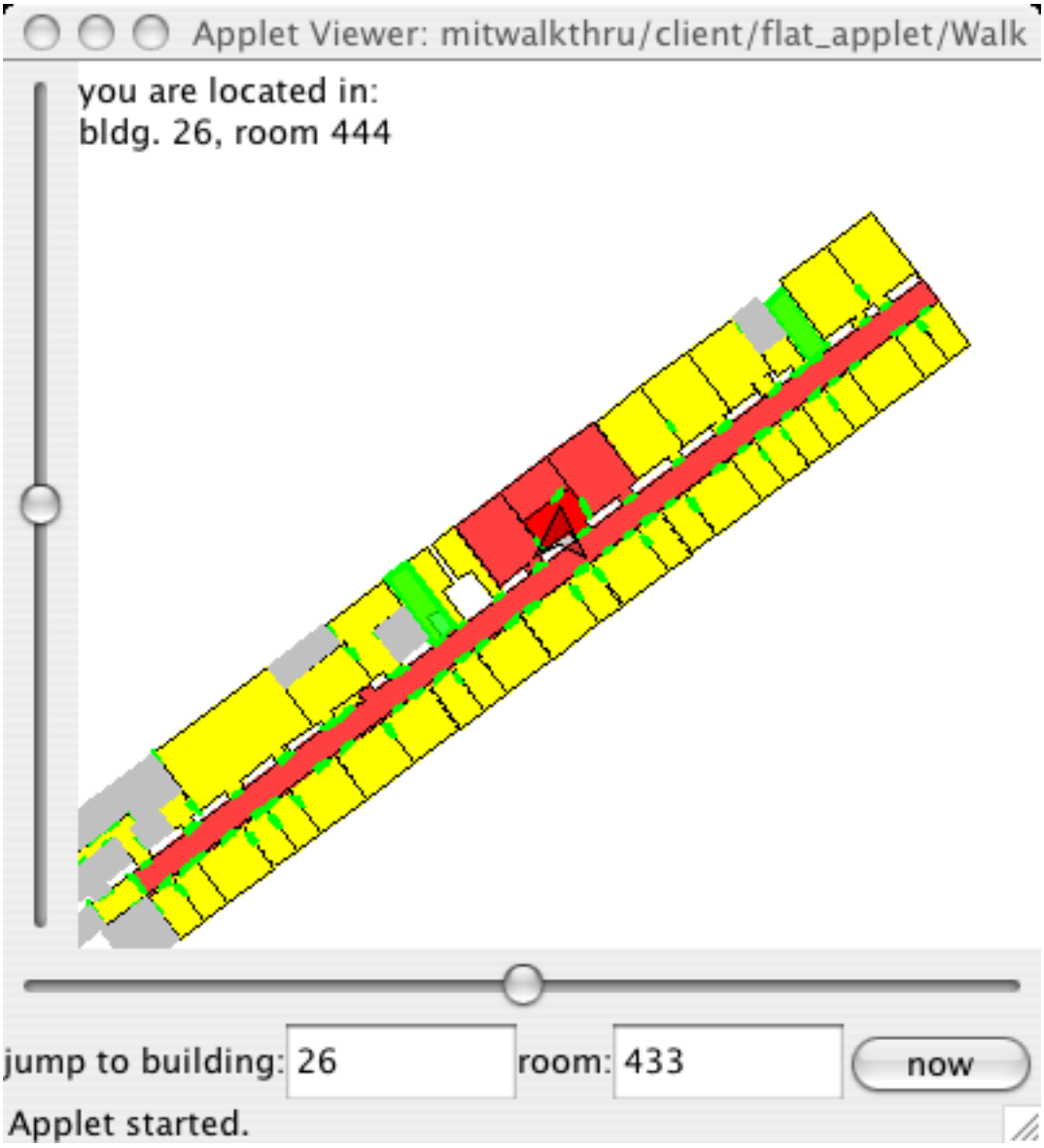

Figure 6-2: A zoomed-out view; nearly all of the spaces on the floor are cached.

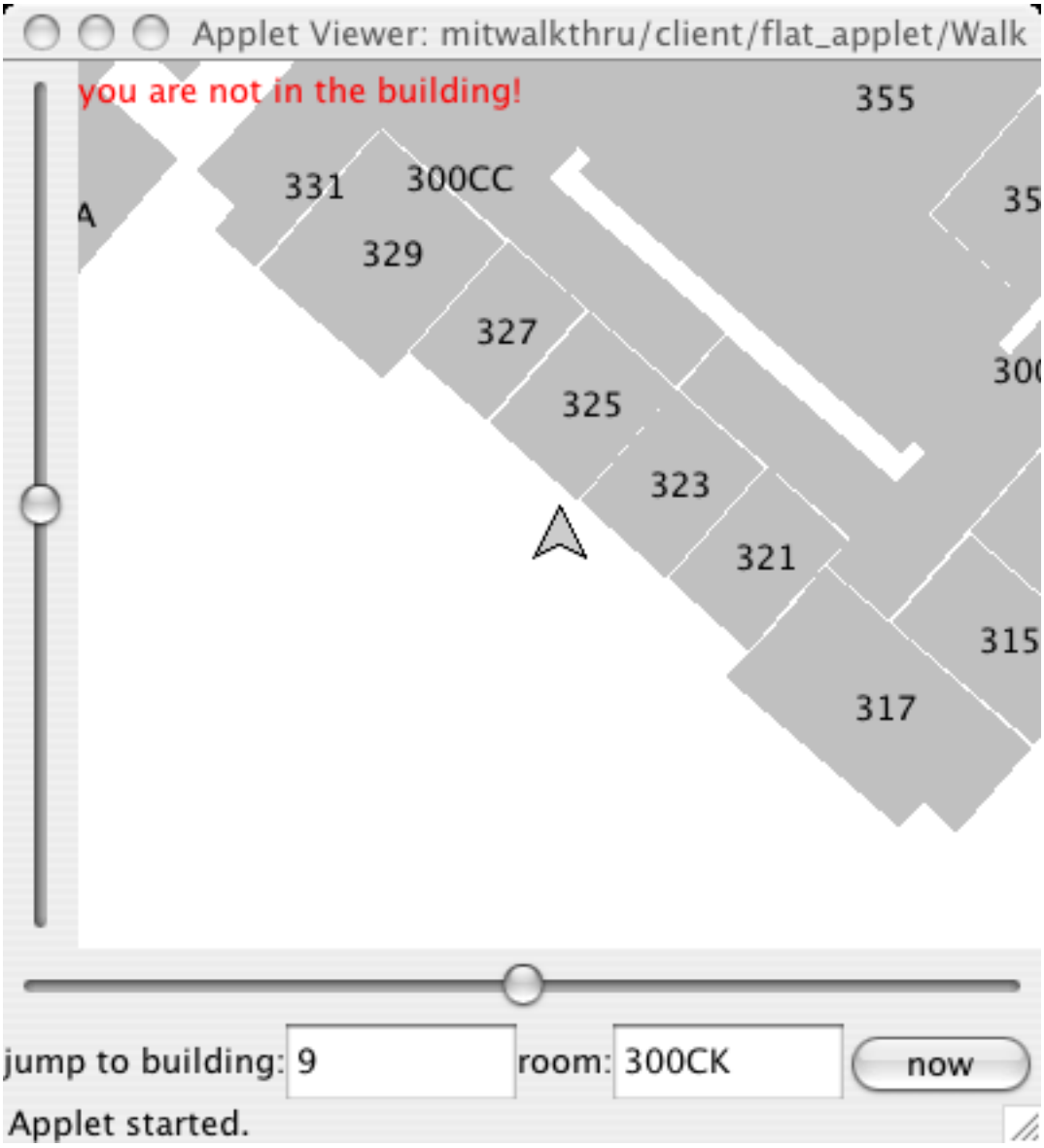

Figure 6-3: An example screenshot of "free movement:" all of the spaces are gray.

Chapter 7

Future Work

MITWalkthru provides a solid platform for addition of new features that are compatible with the graph-structure of MIT's campus data. The most immediate extension would likely be inclusion of 3-D data into the spaces. Also, a different implementation of the graph-structure could be provided, which would give the user fully correct visibility.

Additionally, it may be possible to augment the graph-structure by adding levels-of-detail to the vertices. The client's cache may contain a larger set of vertices at any time, if more of those vertices have a lower LOD; the LOD for a vertex would be upgraded as the user moves closer to it. This is one method by which to alleviate stalling, if the user moves too quickly.

Appendix A

Usage Instructions

Here we describe how to obtain, build, and execute the MITWalkthru project. The instructions given should be run on a computer in the MIT Computer Graphics Group. The instructions also assume that the cited commands (including those related to SVN, Java, and make) are available and properly setup. Finally, the commands listed verbatim are given for the tcsh shell.

A.1 Obtaining the Source Code

The MITWalkthru source code is stored in the CSAIL SVN repository. To check out the source code, the following command should suffice:

```
% svn checkout svn+ssh://${USER}@${SERVER}/${REPOSITORY}
```

with the given three variables: `USER` is the username; `SERVER` is the SVN server to use, and typically just `svn.csail.mit.edu` works; and `REPOSITORY` is the path to the repository, which is `/afs/csail/group/rvsn/Repository/walkthru/mit`. Note, then, that from CSAIL machines, you can typically also just run:

```
% svn checkout file:///${REPOSITORY}
```

Either of these commands will check out the entire BMG source tree.

From the current directory, the source code for MITWalkthru specifically will be stored in `walkthru/mit/src/mitwalkthru`.

A.2 Building the Source Code

Once the source code has been retrieved from CVS, it is fairly straightforward to build. Move into the MITWalkthru directory:

```
% cd walkthru/mit/src/mitwalkthru
```

To build the entire project, including the server and the client applet, simply execute:

```
% make
```

Individual components can be built on their own; examine the `Makefile` for details.

A.3 Executing MITWalkthru

To use MITWalkthru, both the server and client components must be executed appropriately. It should be noted that the client applet will only work properly if the server has already been executed and has finished its initialization stage. Also note that the applet *must* be served by a WWW server running on the same machine as the MITWalkthru server. Finally, port 8080 is bound by the server, and used for client-server communication; the user might need to ensure that any intervening firewalls are configured properly.

A.3.1 Executing the Client Applet

The `flat_applet` is the only implemented applet at the moment. After a successful build of the source code, the file `flat_applet.jar` will be placed in the directory `mitwalkthru/client/flat_applet/`. The included HTML file, `flat_applet.html`, is in the same directory. Only these two files need to be served by the WWW server; the user can move them to whatever WWW-served directory is appropriate. But

the `server` parameter in `flat_applet.html` needs to be changed to reflect the hostname/IP address of the WWW server (which, again, is also the hostname of the MITWalkthru server).

Now, the applet can be loaded by accessing this slightly changed `flat_applet.html`, in whatever way is most appropriate: `appletviewer` or a favorite Java-enabled WWW browser should work fine.

A.3.2 Executing the Server

The server has several dependencies that must be accomodated. For the time being, the safest way to run the server is from the mitwalkthru directory, as follows:

```
% java -Xmx150000000 -Dorg.xml.sax.driver=org.apache.xerces.parsers.SAXParser
  -Dbmg.data.path=${RUN} -classpath ..:../xerces-2_6_2/xercesImpl.jar
  mitwalkthru/server/WalkthruServer
```

where the `RUN` variable gives the path of the `run` directory, generated from the BMG pipeline output. Note also that the `run_basemap` directory (also from the pipeline output) is expected to exist; this should be located in the same directory as `run` itself.

This command runs the WalkthruServer with a max. of 150MB (less than which has been observed to cause problems), with a pointer to the Xerces JAXP implementation (which should be checked out from the SVN repository), and with a pointer to a pristine BMG-pipeline output directory. As there is not yet a clean way to generate this output from BMG, using the copy in `/afs/csail/group/rvsn/www/mitwalkthru/run` may be the safest way to run the WalkthruServer for now. An addendum will be published giving simple build instructions to generate this output.

A.3.3 Navigating with the Client Applet

Instructions for using the applet are included on its WWW page. Here we provide a list of interesting starting points, where the BMG output data are relatively complete:

- bldg. E2, room 254
- bldg. 12, room 005
- bldg. 26, room 433
- bldg. 9, room 300CK
- bldg. E18, room 500CB

Bibliography

[1] Agrawala, Maneesh. *Visualizing Route Maps*. Ph.D. Thesis, Stanford University, Stanford, CA, 2002.

[2] MIT Building Model Generation (BMG) Project. http://city.lcs.mit.edu/bmg/.

[3] jGL 3D Graphics Library for Java. http://www.cmlab.csie.ntu.edu.tw/ robin/JavaGL.

[4] JOGL API Project. http://jogl.dev.java.net/.

[5] Kulikov, Vitaliy Y. *Building Model Generation Project: Generating a Model of the MIT Campus Terrain*. M.Eng. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[6] Niederauer, Christoper, et al. *Non-Invasive Interactive Visualization of Dynamic Architectural Environments*, Proceedings of the 2003 Symposium on Interactive 3D Graphics, Monterey, CA, 2003.

[7] Nichols, Patrick James, II. *Location-Aware Active Signage*. M.Eng. Thesis, Massachusetts Institute of Technology, Cambridge, MA, 2004.

[8] Web3D Consortium - Open Standards for Real-Time 3D Communication. http://www.web3d.org/.