Campus Navigator

ICS 212 project, Winter 2002
by

Introduction

In the past few years we have witnessed an exciting race for miniaturization of computing devices. It is possible nowadays to build a computer equivalent in computational power with the personal computer that sat on our desk five years ago. This evolution may change the way we relate to our computers. And computers already have changed the way we live. Perhaps in another ten years we will be as dependent on our handheld computers as we are today on our email.

For our project we will explore the capabilities and power of a handheld device. We will do this by designing and implementing a real world application. Every product we ever build should be economically viable. In other words, it should offer enough functionality that a (significant) number of people will be willing to open their pockets and purchase that product. Hence the first analysis should be from the user point of view: How the usage of a handheld device is different that the usage of a traditional computer? What additional benefits the user may get from using a handheld device? Next, we should make sure that we could offer that functionality. The second analysis would be: What amount of computation a handheld may support? How to perform such computations on the handheld?

Goal

The concrete scope of our project will be a campus navigator. Suppose Alice Inahurry just arrived on campus and have an appointment with Bob Somewhereon. But she does not know exactly where Bob's office is located. Even with proper maps and directions it is difficult to get to the right spot. Here is where we come in for help. We will attempt to build a system that will automatically direct her to the right door. If we will succeed, then we will have saved her time and energy, thus producing a useful product.

We will build our system upon existing infrastructure. We will be using a handheld (Compaq PocketPC) to support user interaction and we will assume that the handheld has access to a wireless 802.11 network.

Problem components

Let's take a closer look to our problem. We want to give Alice navigation directions. For that we will need to offer Alice a display of geographical data. People have been doing this for ages in form of maps. So our first important component will be a map renderer.
To guide Alice through the labyrinth of buildings and roads, we should provide her accurate navigation information. We should be able to compute this information somehow, so our second important component will be a path finder.

Alice's handheld will have to be in contact with the physical world. It should be able to tell where Alice is and what direction is she looking at. We will have to build a sensor component.

We will need to let Alice interact with her automated guide, so we will need to build an user interface.

Alice should be able to request direction anytime anywhere, so we will have to be able to provide her both the software and the data to do that through a solution deployer.

**Requirements**

We know have a broad overview of our system. But what are the parameters we would like to have met? What is the level of responsiveness and what amount of information we should present to Alice? The best way to answer these questions is to sit in Alice's shoes and think a little how is she going to use our system.

We would like to be able to locate her position with a precision of 1m or even more accurate. Why? Because we need to distinguish between doors. And on corridors doors may be 3m apart. We also need to know on which side of a wall she stands, because if we think she's outside when she actually is inside then we will most probably give wrong directions.

We would like to give her oriented maps. Even if you have a map of a place, you still need to have it oriented and this may be tricky even for experienced trekkers, especially in unfriendly conditions. How good of a precision? Perhaps somewhere near 1° will be enough.

Now we would like to be able to update her map in real time, at least several times per second. Why? First, because she may want to interact with the map display and request different areas or more details. Second, because she may turn around and we need to update the orientation. Perhaps 5fps should be enough for our purpose.

We would also like to preserve a decent amount of detail, no matter how close look she choose to take on our maps. If she wants to see all campus, perfect, but if she wants to see only the interior of CECS building, then we should be able to show her all relevant details inside the building.

We would like also to update the path display often. Even for a pedestrian walking at 1.5 m/s in one second you may have passed by the door you are looking for and you are
already aiming for the next one. So we will need to make the update at 1fps, matching our location precision requirement.

We also will like to be supportive for Alice. We should provide a friendly user interface that allows her to play around and customize the information we present.

Last, but not least, we should make our system easily available. At an extreme, it should run out of standard web browsers and not require anything else.

**Background**

We have defined our problem in terms of components and their expected behavior. We may now proceed and look at existing solutions. Maybe someone had already built something similar and we may just sit back and relax.

**Existing off-the-shelf options**

Internet mapping have been around for a while. Anybody can go to MapQuest [1], enter a pair of addresses and get driving directions from one point to another. This sounds somewhat like our problem description. On the plus side, it offers:
- path finding, driving directions
- unlimited amount of maps
- easy deployment through standard web mechanisms

A closer look we find that there are several shortcomings that make it unusable in our setup:
- we need a much finer detail, on MapQuest’s maps we can’t see any of the campus buildings
  - it is not interfaced with any sensors
  - it is not working in real time
  - it does not relate persons with locations

Perhaps MapQuest should stay on the desktop (near a printer) for the time being.

Well, but there are also portable mapping solutions available. Why not using one of those, for instance TeleType [2]. This one has
- path finding, driving directions
- interaction with location sensor (GPS)
- real-time display of paths

Unfortunately, it is still not covering all our needs:
- it has too coarse detail
- the GPS sensor may be inappropriate in some cases (see below)
- the real time is either very slow or slow with scarce detail
- it is clumsy to deploy, one have to upload her maps before the trip and setup the destination by hand
- it is not relating people with locations
Apparently we will not be able to use this one either.

It appears that none of the existing integrated solutions is solving our problem. It appears that we will have to integrate a new solution. But there is an interesting question of whether we have to integrate existing components or if we have to build new components from scratch.

**Mapping solutions**

The mapping solutions scene is quite large. We have analyzed a number of available products, both commercial and open source. We will try to see to what extent they fit for our problem.

From the commercial ones we have looked at ESRI [3] ArcIMS [4] This is probably the most complete, stable and feature rich solution in the market. Their data format is accepted as a de-facto standard for GIS. Unfortunately, it is so feature rich that it has a very steep learning curve. It appears that one need extensive training in order to use such a tools. There are semester long courses available that will teach one how to start thinking taming the beast. Clearly that cannot be done in our time budget. But perhaps somebody else can start looking at it.

There are scores of other commercial solutions [5]. Unfortunately they usual don't offer free demos and tend to be very complex. We do not need all the complexity in there, just a very little piece of it. Moreover, we'll like to customize the software to meet our requirements and mobile computing constraints, and usually commercial software does not give too many hook points (if any). So we'll not use a commercial mapping application.

What about open source software? There is quite a list of open source GIS solutions [6]. But most of them are desktop oriented. It's quite obvious that is extremely hard to map a desktop application into a handheld device and make it work in real time, so let's just look at the Internet oriented ones.

We have tried Practical Map Server [7]. It works quite well on www.scorecard.org [8]. Unfortunately it is completely undocumented, it serves only raster maps and apparently need some special data setup, both in terms of files and database support. We weren't able to figure out how to install it, so we have to drop this option.

There is another interesting option, the ALOV [9] This one supplies almost everything we require, serves both raster and vector maps, has a client/server version and it's written in java, hence it is portable enough. But it does not trim the data down when serving to client and it requires some database interaction. We decided that there is not enough time to try to install and start this program. Also, it appears that it will require heavy customization in order to adapt it to a handheld viewable area.
Yet another interesting option is GPSDrive [10]. It is already ported for PocketPC, it knows how to download maps from a server, can build path finding. Unfortunately it was released very late (two days before the deadline). Also it is quite unclear how it's path finding works because it uses raster maps. Also there might be some problems because of the need to orient the maps. Anyway, this is worth a look from now on.

We have looked also at a light desktop solution, GeoTools [11]. We wanted to use it as a map drawing application and build on top of it some map server interaction. But we discovered that is was rather slow even for a mighty desktop computer, so we finally decided to drop it, too.

Summarizing, existing mapping solution are designed with a bulky desktop computer in mind and are most likely unusable for handheld devices. They require both high resolution displays and significant computing resources. Moreover, they do not offer easy enough interface. Or they were very late to come in the picture. Or they were too slow. And most often then not, they do not trim down the maps, they simple throw the full map set to the client and expect it to handle the data. Which may be overwhelming for a handheld device.

In the end we decided to go and build yet another map distribution application. As you will see, this is not quite your regular mapping application, it is heavily biased towards minimizing client computations as well as client server communication.

**Sensors**

Our system will have to provide a decent amount of interaction with real world. We'll make a brief survey of the available options.

**Location determination**

GPS [11] Global Positioning System. Based on a network of 24 satellites which broadcast periodic signals. The receiver gets those signals and triangulates a position. Inexpensive receivers have precisions within 10m. Differential ones may get within 1-3m. Advanced GPS systems will give centimeter accuracy. But it need line of sight with six satellites, so it will not work in buildings and may be shielded near tall buildings.

RADAR [12] Based on wireless signal triangulation. The signal strength from several simultaneously visible bases is interpolated and gives the position. They attempted a mathematical model of signal propagation, but they achieved only a 4.3m accuracy for 50% of experiments. The system yields better results if an empirical signal strength map is used - 2.94m in 50% of cases. Drawback: require coverage of all campus point with several wireless access points, which may be overly expensive at the current range of those devices (50m). It requires extensive effort for building the signal strength maps. It is not accurate enough.
We've mentioned only systems that we thought we might use. A good survey of all location determination systems can be found in [13]. Just for the record, those systems may use infrared, wireless 802.11, ultrasound, pressure plates or computer vision. We have also a couple of ideas, but we had no time to even consider them more seriously.

First, we may try to use miniaturized accelerometers and integrate the results. Combining such a system with a GPS receiver may give reasonable results indoors for a while, then we may resynchronize with the GPS when outside again.

Another idea is to measure more precisely radio signals emitted from some base stations. If we could determine not only the strength, but also the direction where the signal came from, then we might hope to get more accurate results. To determine the direction one can use some differential antennae (if we aren't completely off course). On top of that we may try to use a frequency that is not altered so much by the environment (walls, peoples).

**Orientation**

Digital compass. There are a number of commercial chips available that will give accurate results (within degree precision) while working satisfactory in any environment.

GPS. GPS systems also offer some sort of orientation data, but this is quite imprecise for pedestrian use. It is based on tracking the movement of the device and interpolating the course. But a pedestrian may change direction suddenly or may play around with handheld, so this is not a viable option.

**Implementation**

**Analysis**

In building our system there are a number of strategic decisions we have to make. Because of a tight time budget we were careful to make conservative decisions, trying to take a path that will lead to a palpable result when the time ends.

First of all, there is a question of managing the complexity of a full map. Usually geographical information tends to be huge, especially when we require having tiny details available. We don't believe that it's realistically to expect a handheld to manage all the complexity of map handling. Especially since map data will have to be in floating point format and usually handholds computers come without a floating-point coprocessor (for power reasons). So at early stages of our development we decided to split our solution into a client component that will handle map rendering and a server component that will handle the full map and extract only relevant information to be sent to the handheld. The
client will manage only a small piece of the map surrounding the visible window and only the amount of detail that can be visible on the window (for instance, if full campus map is displayed, no room detail will be sent).

We had run a number of tests comparing integer versus double performance on Compaq PocketPC under Java, the result is that integer arithmetic goes 20 times faster than double arithmetic. So we decided that all client computations will be made in fixed precision arithmetic. We will have to use 16 bit of precision because we want to support multiplications in a 32 bit space. But, since we are concerned only with rendering in a 200x200 area that is enough to accommodate all possible detail we'll ever need.

The second big question is what kind of software environment we are going to use. We have a lot of requirements to meet, what tradeoffs should we make in order to design a decent decision? We need to have our system running in real time, but also to have it easy deployable. We decided to use Java for both client and server for a number of reasons:
- we believe that is possible to obtain real-time performance even if using Java, if carefully used. We will have to reduce computations on client side to a minimum. We will also have to be able to trim down the data coming to the client to a minimum, reducing computational as well as communication load stress.
- we need a portable solution. In the current market handhelds come and go in matter of an year. It will be somewhat wasteful to design and implement a solution that works on the current generation of handhelds from a single vendor, having the prospect of being completely outdated in less than a year.
- we need to have a solution deployable through internet. This requirement can be met only using Java.
- last, but no least, it will be easier to prototype our solution if we were to use the same environment on both client and server. We may short-circuit the communication part and see how thing work without even setting up a real server. This may prove invaluable, because it will give us a chance to steer our project before investing too much effort into a solution or another. Having a short-circuited version it also precious because allows shortest and fastest debug cycles.

Next, we'll have to decide if the client will deal with vector maps or with raster maps. Raster maps are faster for rendering, but have several drawbacks: are big, hence slow to load. As we think of the user interacting with the system, we discover that it will be a good idea to let the user have some map playing without requiring server interaction. In a raster solution this will mean a larger raster that will make the communication more of a problem. There is another interesting question: how are we going to draw a raster rotated clockwise with 36 degrees? This is not that easy and may require substantial amount of computation (hence slower response) [14]. On the other hand, using vector data will lead to smaller amounts of data. Using fixed precision arithmetic and already trimmed data the rendering procedure may become fast enough. We may also let the user play around with pan and zoom and silently load new map pieces when necessary from the server. Orienting is also a fast procedure (we have to rotate only a relatively small number of vertices opposed to all image's pixels). So we will use vector maps on client side.
For the path computation problem there are again two possible solutions: use a raster map and find a path based on colors or use a layer of paths vectors and build a graph of them. The raster approach seems seductive because it's simplicity. But it does not scale well - what if we need to find a path across one side of the campus to the other, but only display the last final meters in detail? We will be missing the detail. Still we've implemented the raster solution, but we've found it also slow, running in 5 seconds for a 1000 pixel long path on a 1Ghz PC. So we decided to use a graph of path vectors.

The path could be build by the client or by the server. If we want the client to be able to build it, then we should be able to trim down the vector path down in the same way as we did to the map. But we cannot cut a piece of the graph without the risk of disconnecting the graph and/or miss the shortest path (Fig. 1).

We decided to keep the graph on the server and do the path computations there. In case of server failure the client can still track the user along the last path computed by the server (see figure).
Architecture

We've decided for a client/server approach, we've pretty much placed our modules on client or on the server. The final architectural picture is:

Client

We implemented the client using Jeode JVM [15]. This is a fairly implementation of PersonalJava [16] for the Compaq PocketPC, albeit we found it incomplete. The implementation is real-time oriented. This means:
- we reduce the computations as much as possible, in size and data types
- we try to have everything preallocated in order to reduce the pressure on the memory management system of our JVM.

The client consists of a number of threads, each thread implementing a module. We choose this approach because we need user responsiveness and this is the way to give fast user feedback while performing time consuming operations in the background. We found useful to model our threads in the worker paradigm (see ThreadWorker). At any moment a worker may execute at most a job, while accepting new requests. After the current job is done, it will notice all interested parties (through Observer/Observable mechanisms) and start processing the last request received. All other requests are discarded. We've modeled in this way the Comm module which does the server communication (ThreadLoader), as well as the Render module which renders the map.

The central module is the Render. It receives requests for update the screen from the Sensor or from the UI. If necessary it calls the server to get an updated map/path. It is using a double buffering scheme, so at any point there is a current map ready for being placed on the screen as well as a drawing map in which it renders the new map image. When done, it switches the two images and calls the Java AWT thread to repaint the screen.
We will describe how we organized the geometry. There are mainly three coordinate systems, real, area and screen (Fig. 3). If no other stated we will be speaking about real coordinates.

![Coordinate Systems](image)

**Figure 3. Coordinate Systems**

The data used for rendering is a piece of the map in fixed-point integer representation using 16 bits. It represents the screen window, as well as the surroundings. We have chosen to have more information that strictly necessary in order to support small range panning/zooming directly from client data without a new server request, thus improving responsiveness. There are also two thresholds for the detail. If the screen window goes below the min or above the max threshold, a server request will be issued to retrieve a new map with less or respectively more detail.

All 16 bits of precision are fully used. In area coordinates, the minimum coordinate of a point is - MAX_PREC and the maximum is MAX_PREC. In this way we make sure that no limited precision artifacts will alter the rendering quality.

To map from area to screen coordinates we use a series of transformations:
- clip all shapes to screen window
- recenter data to match screen widow center
- zoom data so the screen window covers the full area (- MAX_PREC, MAX_PREC), using multiplication and shift, no division
- rotate the map using fixed point sin and cosine
- map data into screen coordinate, multiplication by screen size and shift by precision

The UI module is composed of two components: map display and stylus/mouse interaction. The map display is done in the standard Java AWT thread. More interesting is the stylus interaction. We have tried to make it as intuitive as possible. There are three main actions a user may request: recenter the map to see the full path, pan the map or zoom the map. Panning and Zooming are done by dragging the stylus on the screen and make map updates in real time. We have also used some visual indicators of what action is in progress. As an implementation detail, one will need an additional thread to handle related mouse events, leaving the AWT thread to generate those events.
The Sensor module reads the sensors and makes update requests for the display. At this point we are only interfacing the GPS sensor. The GPS communicates with the outside world through a serial port protocol, NMEA [15]. A description of this protocol may be found in [16]. The serial communication is a java extension, the javax.comm package. It is not yet available for the Jeode JVM, so we have written a native code application that reads the GPS port and writes relevant information into a file. That file is periodically polled from our client application to retrieve location information. Still it is possible to write our own JVM extension. An introductory description on how to use the javax.comm package and how to interface the JVM with a serial port using native methods may be found in [17].

Server

We implemented the server using the Tomcat servlet engine [20]. All requests are made and served through standard HTTP protocol. This choice offers us reliable server software to support our needs. The server is connected to an Oracle [21] database used to store the person-location mappings.

The first problem for our server is how to efficiently trim down possibly huge map information down to manageable levels. We will use Quad trees to quickly trim down most of the data. A quad tree is a geometrically organized tree. Each node corresponds to a geometrical area and may have four children corresponding to four geometrical slices of it's area (Fig. 4).

We also use Level Of Details (LOD) in order to send only the relevant information. This is a very simple concept: a request has a certain size. All invisible elements for that size

![Figure 4. Quad tree](image-url)
(for instance they have sub pixel dimension) will simply be filtered out. We maintain a list of precomputed Quads for each possible LOD.

To make the geometrical data usable by the client, we clip the shapes in the requested window using a standard clipping algorithm, the Sutherland-Hodgman polygon clipping. We adapted the implementation from [22]. Afterwards we transform real coordinates into area coordinates and send the result to the client.

Geographical data is stored on the disk in ESRI shape file format [23]. We have chosen this format because it is the most widespread GIS data format. The availability of editing tools [24] is also tempting. Moreover, we could pick a Java module which reads this data format from [7], without bothering rewriting everything from scratch.

For the path problem we create a graph of line segments based on a special layer(s) containing path information. We link segments based on edge proximity. The path finding mechanisms is divided in two steps. First we have to determine where in the graph the desired path should start and end. This is not as trivial as it looks because it is clearly wasteful check all segments for proximity. We have build structure similar to a Quad tree, but which records which areas a line goes through as opposed to what areas a bounding box lies in. We called it LineQuad.

After we determined where our path starts and end, we have a classical graph problem. We will solve it using A* algorithm [25]. This is a Dijkstra shortest path algorithm guided by a heuristic. At each step of the Dijkstra algorithm the next node to be chosen is the most promising one, i.e. the one with $f = g + h$ minimum. By $g$ we denote the cost of getting to that node and by $h$ we denote the heuristic estimated cost to reach our goal. For geometrical path finding the h function may be chosen naturally as the Euclidian distance between the current node and the destination node. This has the nice property that $g + h \leq$ total cost for any node, which guarantees that the A* algorithm will succeed in finding the shortest path.

**Deployment**

We would like to have our client be deployed directly form the web server in form of a Java applet. Our applet will need to access local data, so we have to use a signed applet. But the Jeode JVM does not support signed applets. We decided that we will install the software on the handheld and we will launch it using a link (.lnk) computed on the fly by the server. Since there is no shell command line on our PocketPC running WindowsCE, this is the only way to pass the necessary parameters.

We have build a web database infrastructure to hyperlink a person with his location. There is a database that stores the person location information. A web server reads the information from the data bases and for each person presents a hyperlink to the start point of the navigator system, labeled with the person’s name.
Implementation limitations

After finally getting a working system we tried it in real conditions. We have found a number of limitations of our approach:
- it is very hard to handle the constraints imposed by walls. The problem lies in detecting the position in the path graph of the start and end points. Sometimes an outside path is closer to an inside position than an inside path, fooling our algorithm to think the user it's outside. This problems may be overcome increasing the number of available paths, for instance laying a path near every wall. It is unclear though how much data increase we will have for such an approach.

Because of time constrains we were forced to drop some parts of our planned implementation:
- we haven't implemented the independent client path following
- we haven't integrated the digital compass
- we haven't tried in building location sensors
- we haven't build an detailed and/or accurate campus map
- the current position is
- auto centering functionality is missing

Because of third party software limitations, we failed to implement some aspects:
- we can't deploy the software form a web browser - no signed applet support
- we can't read directly the sensors data - no serial communications support
- we cannot draw the paths in more visible way, the AWT API allows us to draw only 1 pixel wide lines

Experimental results

We have made a comparison of our system using as a client a 1 GHz PC and a 206MHz PocketPC.

<table>
<thead>
<tr>
<th></th>
<th>IPAQ, ARM 206MHz</th>
<th>PC, Duron 1Ghz, Sun JVM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Map render</td>
<td>300 ms</td>
<td>10 ms</td>
</tr>
<tr>
<td>Map load</td>
<td>3000 ms</td>
<td>100 ms</td>
</tr>
<tr>
<td>Path load</td>
<td>1000 ms</td>
<td>40 ms</td>
</tr>
</tbody>
</table>

While the PocketPC solution runs 20-30 time slower, it is still fast enough to meet our initial requirements.

We have also analyzed the communication load and we have found that it is quite light.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Map load</td>
<td>4 Kb</td>
</tr>
<tr>
<td>Path load</td>
<td>0.25 Kb</td>
</tr>
</tbody>
</table>
We didn't have complex maps at hand, so our experimental results should be considered preliminary.

**Future work**

The main unsolved problem remains the problem of accurate in building location sensors. This is still a challenging problem in terms of finding an accurate cheap solution.

We have built an adhoc infrastructure for map deployment mechanism that is rather rudimentary. It may be a useful thing to try to use an existing map deployment infrastructure and try to customize it for the problem at hand.

We are using Java for portability reasons. Still, it might be interesting to write the client in native code and increase the rendering performance (yet the main bottleneck seems to be the network distribution of maps).

**Conclusions**

Recent advances in miniaturization technology permit us to carry in our pocket a fully functional personal computer. We have analyzed how application design and implementation is different for such a device as opposed to traditional desktop computers. We have focused our research into one application area - geographical information systems. In the process we have build a prototype of a real application.

We have shown that it is possible to build non-trivial real-time applications for a handheld using Java. This may be contrary to the conventional wisdom Java => virtual machine => hopelessly slow. For current developments in handheld computing portability of applications becomes even more important that it were in the Wintel dominated desktop computing scene. Being able to execute and deploy an application to a wide range of machines becomes crucial.

**References**