In the IEE International Workshop, Intelligent Environments 2005 (IE05), Colchester, UK, 28-29th June 05

PROBLEMS FACED IN DEVELOPING AN AMORPHOUS COMPUTER AS AN INTELLIGENT SKIN FOR PERVASIVE COMPUTING

A M King, V Callaghan, G Clarke

University of Essex, United Kingdom

Abstract - Futurists predict the use of Smart Matter, nano-technology with computational ability, as pervasive. It will be painted on the surfaces of our environments, and the futurists suggest that these surfaces will be used as video displays, user interfaces, and sensor arrays. The implication is that a coat of paint will be enough to add this functionality to a surface.
The field of Amorphous Computing provides one possible realisation of this vision; an amorphous computer is a multitude of identical tiny computers with local communication capability, and can be painted onto a surface to form an ad hoc network. An iSurface is a specialised version of an amorphous computer, using a simulator with a physical grounding. In this paper we argue that such a vision of Intelligent Surfaces is fundamentally flawed. The load, both in terms of processing and communication, on the individual elements will always be a hindrance to responsiveness of iSurface applications, and we present experimentation on several applications to demonstrate this.

1. INTRODUCTION

"Today, people's domestic spaces are becoming increasingly 'decorated' by electronic or computer based artefacts (gadgets) varying from mobile phones, through CD players, to transport systems and beyond"[1]. This conjures the vision of a rich, dynamic ecosystem of interacting devices with computational capabilities. The advent of nanotechnology opens up new possibilities for pervasive computing; mass-manufactured nano-devices could literally saturate the environment. "Smart matter" [2] could be everywhere; sewn into the fabric of our clothes, or painted on the surfaces of our homes as an "intelligent" skin. Future visions [3] of the home suggest that applications such as video and user interfaces will be commonplace on these surfaces, and that they will also act as composite sensors such as video cameras. The implication is that a simple coat of nano-scale device bearing paint will be enough to add this type of functionality to a surface. Such an "intelligent surface" (iSurface) can be seen as a specialisation of an Amorphous Computer [4] - a multitude of identical tiny computers (particles), each with a CPU, memory and communication capability. A possible precursor to the nano-scale particles can be seen in the Smart Dust mote [5], a MEMS based sensor and processing node. We refer to these elements of an iSurface as iCells.

An amorphous computer is a massively parallel system limited to local communication between neighbouring particles (iCells). Amorphous computing is the development of organisational principles and programming languages for obtaining coherent behaviour from the interaction between large numbers of unreliable particles that are interconnected in unknown, random, and time-varying ways. Investigations into massively parallel systems, such as cellular automata, are one source of ideas for dealing with amorphous systems. An alternative is research into self-organising systems which Abelson et al [6] suggest offers a possible solution which is to embed all the required code into the particles at the time of manufacture. With an amorphous computing system where each particle has this code preloaded, the particular functions that are activated within any particle depends on the messages it receives from the local environment. The specific issue that makes this a hard problem, is predicting the range of functions that must be pre-coded given the vast number of possible states the system can be in. Butera argues that: "A programming model employing a self-organising ecology of mobile process fragments supports a variety of useful applications on an [amorphous computer]." [7]. In support of this, he offers a distributed programming methodology, known as 'process self-assembly'. A programming model is introduced, as is the "process fragment" - the atomic element of process self-assembly. His project demonstrates the feasibility of a mobile agent paradigm which we have used as part of our system.

Returning to the future vision, we see that what are proposed as applications for an iSurface - video, composite sensors and user interfaces - are all data-heavy, time-critical applications. For this to work, an iSurface needs to be able to display images and transfer large amounts of data. It needs to be able to locate elements of an interface and allow the ability to click buttons and to drag and drop interface elements. It also needs to support connections between these functional features of the iSurface. The question is whether an iSurface, derived from the work on Amorphous Computing and Smart Dust, would be a feasible platform for these applications. If so, then what is necessary in terms of capabilities and software? If not, then what are the stumbling blocks that prevent the

concept being a success? This paper reports on experiments, based on an essential time-critical application for an iSurface, which suggest that this vision is unrealisable now, or in the future, because of a number of severe problems. Specifically run-time damage, poor response times, and, most importantly, the load on the iCell's resources.

1.1 Physical Grounding and the Transmission Bottleneck

If we based an iCell on the last developed version of a Smart Dust mote, we would end up with an 8 MHz processor and 10kbps communication speed. Moore's law suggests that, by 2020 (a realistic target date for iSurface manufacture), an equivalent processor would be operating at 8 GHz with a communications speed of approximately 10Mbps.

Modern processors can effectively perform one operation / instruction per processor cycle and thus we can assume that 8 MHz gives 8 MIPS (Millions of Instructions Per Second) and 8 GHz gives 8000 MIPS. Converting several lines of C++ code into their assembly language equivalent suggests an average of approximately 15 instructions per line. The core of an iCell as it currently stands is approximately 300 lines of code in size. Using our average iCell core of 4500 instructions, this would take 0.0005625 seconds to execute on an 8 MHz processor and 0.0000005625 seconds at 8 GHz. If we arbitrarily choose a message size of 50 bytes (400 bits) then this would take 0.04 seconds to fully transmit on a 10kbps connection. On a feasible current iCell, one cycle of the core process on an 8 MHz processor would take 0.6 milliseconds and the transmission time for 50 byte message at 10kbps would be 40.0 milliseconds. We can immediately identify the transmission times as the factor most likely to cause a bottleneck in a system comprising of iCells simply running their core program. Assuming the same is true for the futuristic version of the iCell, which has an improved performance over the Smart Dust specification by a factor of 1000, we arrive at the figure for one cycle of an iCell core on an 8 GHz processor as 0.0006 milliseconds. The associated transmission time for a 50 byte message at 10Mbps would be 0.04 milliseconds. It is clear that the transmission time is the likely bottleneck for any system using messages of this size. In order for transmission time to cease being the bottleneck, a message size of six bytes or less would be required.

1.2 The iCell refractory period

Using an undirected broadcast method of propagating agents and messages across a surface, it is inevitable that messages will eventually be passed back to the iCells that spawned them. The result is an iSurface with a spreading activity that never actually stops. Taking inspiration from excitable media [8], the iCell adopts a refractory period for dealing with messages. Basically this is a rest period during which the iCell is no longer capable of receiving messages of a certain type. Internally the iCell maintains a finite short-term memory store for these periods. A memory entry is recorded as a message ID and duration. When a message enters an iCell it is checked against the memory; should the entry for that message exist, and the duration hasn't expired, the message is deleted and never makes it to the agents or iCell core. As this memory store is only finite, the iCell can only remember a relatively small number of messages so a huge number of uniquely identified messages entering the system at the same time would have a very negative effect. Experimentation has demonstrated that, by introducing a number of unique messages exceeding the available memory limits for an iCell, the propagation of the messages will never cease. The result is a hyperactive surface with constantly increasing levels of data.

1.3 Overlaying a Gridwork onto an Ad Hoc iSurface

One of the aims of an iSurface is that it is an ad hoc system, where iCells are randomly sprayed onto a surface and self-organising into a working system. This involves successfully assigning channels of communication such that contention is avoided and each iCell can form full-duplex communication with its neighbours. This problem is conceptually identical to that found in cellular phone networks. There are a limited number of frequencies available and many "users" wanting to communicate. Mobile phone companies solve this by dividing their coverage area into cells centred on base stations. Each cell has a range of frequencies available to it that adjoining cells lack. This allows the re-use of frequencies across the system without significant fear of contention. However, a cellular phone network relies on full-duplex connections made between the handsets and the base unit, not between handsets. We can apply the idea to the iSurface by saying each iCell is its own base unit and its neighbours are within its communication cell. However, with this system the communication cells overlap to an extreme extent due to the massive combination of possible neighbourhoods. We need to ensure that every one of the 10,000 iCells used in the experimentation can communicate with up to eight other iCells, without contention over channel usage becoming too much of an issue. This problem is similar to that of map colouring; a closed graph's areas need colouring so that no neighbouring segments (sharing an edge) share the same colour. However, channel assigning is far more complicated; "neighbours" can be defined as iCells related by a single degree of separation (a common physical neighbour). Most maps can be completed with

three or four colours but due to the complexity of using 10,000 iCells with so many different combinations of neighbourhood (and the need to communicate with 8 other iCells) this obviously isn't true for this problem. Existing techniques for map colouring generally work on a centralised basis, where the system has the advantage of a global view. Our distributed system must make use of self-organisation to assign the channels.

Experiments with the iSurface have been carried out until no contention exists. Firstly the system loads the pre-constructed physical layout of 10,000 iCells and calculates the neighbours for each iCell. The communication range used gives an average neighbourhood size of 9, greater than the desired size of 8 but useful for our intended goal of redundancy to aid robustness. The iCells are then assigned random channels to broadcast on for each of the 8 possible output connections. The main loop of the system begins with every iCell in the system evaluating what inputs clash with its outputs or with other inputs. Wherever a clash is detect the iCell broadcasts a "disruption signal" on that channel to allow its neighbours to realise they're causing a conflict. Then each iCell goes through all eight of its outputs and checks if there is a disruption signal being broadcast on that channel. If there is then that output goes through all possible channels starting from a random point in the spectrum. It chooses the first channel that doesn't have a disruption signal. If none are available it will choose a channel at random. Allowing this system to execute for long enough will allow it to enter a more or less stable condition where the number of collisions doesn't vary significantly from one cycle to the next.

Experimental results showed that a significant number of channels were necessary to remove contention completely from the system. For anything less than 575 channels, the system was observed to descend to a stable level of contention and fluctuate around that point. For 575, or greater, channels, the system entered a state of complete stability with zero contention. The more channels that were available, the quicker the stabilisation of the system.

Once a distribution of channels for each iCell, without contention, was obtained, the system entered a phase of linking iCells into full-duplex connections with each other. Each iCell waits a random period before attempting to connect to its neighbours. When it activates, it goes through each of its incoming frequencies and assigns one of its own outgoing frequencies to it, thus forming a full-duplex connection. If it finds it has already made connections before its own activation (by another iCell linking to it) then it bypasses the affected frequencies. The success of this system is entirely dependent on communication range. Range determines neighbourhood size, so, for small neighbourhoods, iCells are unlikely to make the 8 reciprocal connections that are possible. Large neighbourhoods result in a large number of fully connected iCells, but also an increasing number of isolated iCells with no connection whatsoever. The consequences of the increasing neighbourhood sizes extend further than simply having more redundant iCells. Due to the increased neighbourhood sizes, iCells have more potential partners to connect to. However, an iCell is limited to 8 connections, and it chooses these from the available input channels. These are guaranteed to be the 8 closest iCells. Increasing the neighbourhood size beyond the immediate physical neighbours results in a fragmented network. Here we can have iCells that don't actually connect to their physical neighbours, sometimes resulting in areas cut off from the rest of the surface, although this is very rare. To avoid this problem, the range, and hence neighbourhood size, needs to be selected so that the 8 closest physical neighbours are the most likely candidates for connection. The value for this range depends on the density of the iCells in the iSurface.

1.4 The Remainder of this Document

The remainder of this paper describes the experimentation performed on the iSurface. Firstly, we describe the iSurface simulator, the physically grounded application used to perform the experimentation. Secondly, we discuss the application of Emergent Pathways, the growing of lines between two points, and the problems encountered when doing so. Thirdly, we examine the use of the iSurface to create a system of buttons, and the associated difficulties. Fourthly, we investigate the trouble inherent in developing a video system for the iSurface. Fifthly, we consider the obstacles facing the use of the iSurface in a simple gesture recognition system. Finally, we deliver our conclusions.

2. EXPERIMENTATION ON THE iSURFACE

2.1 iSurface

The iSurface has been designed to work on a grid basis with a definite notion of up, down, left and right. This could be accomplished by manufacturing the iSurface in a sheet or tile format where all the iCells are pre-connected on a grid. The iSurface simulator on which all experimentation was performed simulates a grid of 65536 (256 by 256) instances of an iCell. The iCell is a completely self-contained simulation of a "real" iCell that maintains its own message buffers for I/O and its own list of Agents. It is capable of communicating separately with each of its neighbours, assuming exclusive full-duplex communication with each. The majority of messages in this simulation are the same size, typically 8 bytes, and each cycle length is equal to the time it takes to transmit a message of this size; the

result being that an iCell can transmit one message to its neighbours per cycle. Any other messages that the iCell tries to send during the cycle are added to a queue for transmission in future cycles. In this simulation, the message size, and hence communication time and cycle duration, are determined by the gradient information message. The Agents used in this simulation are stored as hard-coded classes that are instantiated by each iCell. They spread across the iSurface by replicating themselves on startup to all the "uninfected" neighbours of their host iCell. Each Agent has access to the sensing, effecting, processing, and communications capabilities of their host iCell. However, Agents on the same iCell are not necessarily aware of each other's existence.

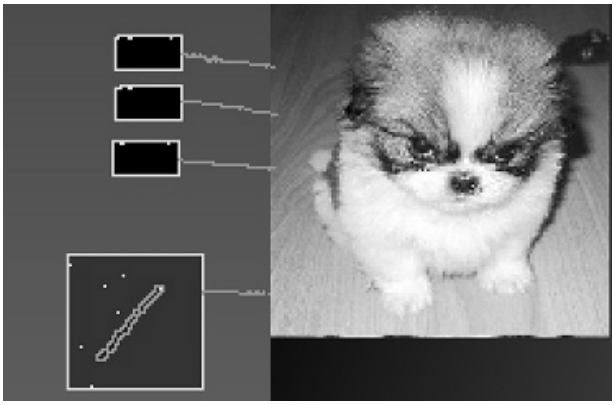

Figure 1. Screenshot of the iSurface simulator. The three black boxes are buttons, the grey panel detects gestures, and the picture of the dog is a video display.

2.2 Emergent Pathways

One major problem that underlies the entire iSurface concept is the reliable transmission of data from one area of the iSurface to another. Communications on an iSurface are generally undirected and a method involving propagation of a signal throughout the network is inefficient, albeit guaranteed to reach its target if any path exists. Directed routing of data is possible. Use of a coordinate system allows messages to be passed in the right direction by comparing the current node's coordinates to those of the target node. However the problem lies with the reliability of such coordinate systems and the requirement of a signal knowing its target coordinates. Another solution is the creation of "channels" for the signals to pass along. A channel consists of an uninterrupted line of iCells between two or more points that relays signals along its length. It is possible for a user to explicitly define a channel between two points but should damage occur and this channel be interrupted, a user is needed to make manual repairs.

From Amorphous Computing, Clement and Nagpal [9], proposed a process of growing connections between two endpoints and for these connections to be self-repairing in the event of damage. As an iSurface can be seen as a specialisation of an amorphous computer, this process should be implementable on the iSurface. Clement and Nagpal's aims are based on the creation of shapes rather than developing connections for communication, but their approach provides the foundations for a possible solution. This Amorphous Computing approach builds upon *gradient fields* emitted from one of the endpoints of a line. The *source endpoint* uses a high value for the gradient to begin on. This value is broadcast to all neighbours in the form of a message tuple {Processor ID, Gradient Value, State, Successor ID}. Processor ID is a value randomly chosen by the particle (iCell) as a unique identifier. Gradient Value is the gradient at that particle. State denotes whether that particle is part of a line. Finally, Successor ID is the ID of the particle that provided the highest gradient value to this particle. When a particle receives this message, it compares the gradient value with the one it has stored. Should this new value be higher, the particle decrements it and stores it along with the ID of the sender as the Successor ID. The successor node allows the gradient field to exist as a chain from each particle back to the gradient source; no data aside from the gradient value and the successor ID need be stored.

When the predetermined *target endpoint* of the line receives a gradient value, it changes its State to on (meaning it considers itself to be part of the line). It then broadcasts a message including the new state. Its successor node will detect that it is identified in the message, switch itself to an on state and then broadcast its own message. The line forms by a process of backtracking through the successor nodes from the endpoint to the gradient source point.

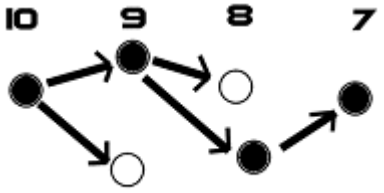

Figure 2. The node with a value of 10 is the gradient source. Reversing the flow of the arrows identifies successor nodes. The node with a value of 7 is the endpoint. The solid circles show where the line has backtracked through the successor nodes.

2.2.1 Self-repair Using Active Gradient. As the line is entirely dependent on the chain of successor nodes in the gradient field, any alteration of the field will alter the successor chains and thus alter the line. This is the reasoning behind the self-repair section of the Amorphous Computing approach. This is called an Active Gradient approach - a gradient field that can adapt itself to situations such as surface damage thus causing the line to re-route accordingly. iCells need to periodically broadcast their state and gradient

information and remember a timestamp based on the processors internal clock. When an iCell hears from its successor it will update the timestamp. Should this timestamp become too old, the stored values for the gradient and successor ID are considered to be unreliable. In this event, the iCell will decrement its stored gradient value. As time passes with no signal from its successor this gradient value will continue to decrease. Eventually another neighbour's broadcast gradient signal will be higher than the stored gradient, thus meaning that the neighbour is now closer to the source. This gradient is adopted and the successor link replaced accordingly. Using this approach causes the gradient to slowly adapt itself to damage. According to the theory this will also cause the line to adapt to this new gradient i.e. self-repair.

2.2.2 Types of Gradient Creation. There are two main approaches to gradient creation. The algorithm for the first approach is exactly as described earlier. This approach generates a gradient in a single pass. The second, multi-pass approach is identical to the single-pass system except that when it receives an update message from its successor node, it will update its information to match this input, even if the new information is worse than its current value. This has the effect of breaking down the data-heavy single-pass result and creates an incomplete gradient, refined by the periodic update messages.

To begin with, the multi-pass system creates an incomplete gradient for much less bandwidth cost than the single-pass approach. However, the update passes carry a fixed cost as well as costing more bandwidth when newly updated iCells send their updated gradient message to their neighbours. After several passes the total cost far surpasses that of the single-pass approach. This implies that a single pass approach is more efficient, in terms of total resources used, at creating a gradient.

2.2.3 Dealing with Surface Damage. One of the stated aims for this application area is the ability for the line to adapt to damage to the iSurface. For the purpose of experimentation it was decided to use a similar form of damage to that presented by Clement and Nagpal. All damage repair experiments utilised a large, solid rectangle cut out of the centre of the surface, bisecting any line present there. The multi-pass system is the only approach that was able to successfully deal with gradual repair. The difference being that gradient change is passed down the successor chain and Agents update themselves according to their successor data instead of waiting to adopt the lowest available gradient as a successor. In this way the altered gradient propagates through the successor network and thus aids adaptation. However, comparison of the bandwidth used by a single-pass complete reconstruction (experimental results provide an average of about 1,400,000 bytes across the surface) and that used by the multi-pass repair (77,177,375 bytes) suggest it is more efficient to scrap the damaged gradient and start from scratch. Unfortunately, detection of the damage in order to reinitialise the gradient would be a problem. The gradient needs to be reinitialised at the source. This means that the source needs to determine if damage has occurred. This could be accomplished by regular signals being sent along the wire between the source and the destination. Should a predetermined length of time expire without such a signal, the source may determine that the line is broken and start a re-initialisation. There are various issues of responsiveness with this. Primarily the distance between the start and end points. A timeout needs to be tailored to allow a signal to cover this distance. The larger the distance, the longer the timeout needs to be. For larger distances, the responsiveness suffers as a result of this longer timeout to determine if a break has occurred.

2.2.4 iCell Load. iCell Load (iCL) is the overhead of an iCell's processing and communications resources generated by the Agents and messages resident within them. When significant pressure is placed on the communication system, massive backlogs appear which can be fatal for time critical applications. Messages enter the iCell, or are generated internally, such that the output cannot be transmitted away fast enough to clear the queues and so the backlogs increase. This is a direct result of transmission times; the faster an iCell can transmit, the higher load it can tolerate. Experimentation using agents that generate fake iCL was performed on the single-pass approach. An unencumbered system takes about 100 cycles to complete. Low levels of iCL had little effect on this. As iCL levels increase they have a significant delaying effect on the propagating gradient. The high data rate of the gradient data compounds the problems caused by the high data rate of the iCL making agents. A similar effect occurred to the multi-pass approach, but as the data rate of the propagating gradient waves was quite low, the iCL didn't affect it to the same extent.

2.2.5 Creating the Line. Utilising the successor chains created as part of the gradient, the line develops exactly as expected. However, attempts to get the line to respond to changing gradients cause a highly localised concentrated form of iCell load to occur. Agents in their line state retransmit while the gradient field is in flux. This causes new lines to be created. These new lines are quickly broken by the changing gradient, but not before spreading onwards. Lines can be created faster than the timeout that destroys broken lines. Coupled with the gradient information messages, these cause queues of messages that continue to increase, eventually crippling the iCells. This results in the affected section of the iSurface becoming unusable and, as the gradient information propagates, this affected section can spread.

2.3 Buttons

Buttons are useful functions for a user-interface, and can act at different levels of functionality from being stand-alone buttons, to being part of a complex hierarchy of buttons in a menu system. Buttons are the result of a single agent replicating itself across an area of the iSurface. The result is a patch of iCells with homogenous coding, in terms of the button agent. The user initiates the growth of the button specifying the size it is to grow to. The user then specifies the interactions of that button with regards to everything on the iSurface. Figure 1 shows buttons in use in an application; pressing them will alter the image.

2.3.1 How Buttons Work. Once a button agent is instantiated on an iCell, it releases a request to all neighbouring iCells to send it a "coordinate" message. Should an agent receive this request it will respond with its known coordinates as well as the specified dimensions of the final button (in terms of iCells). When an agent receives such a response (from the user or another agent) it will alter the given coordinates according to the direction it came from and store them as its own. If these coordinates lie outside of the specified dimensions, the agent will set itself into an "interface" mode, otherwise it transmits a copy of itself to all surrounding iCells (already infected iCells will ignore this) and the new agents repeat this process. The result is a rectangular region of specified size infected with generic button agents ready for programming.
The user needs to provide three pieces of data to the button to program it. The Button ID is a globally unique identifier for the iSurface and also the signal given off by the button when pressed. The Inhibition ID specifies the signal that will cause the button to become inactive, and the Activation ID the signal that will turn the button back on. These Inhibition and Activation IDs can be the Button IDs of other buttons, thus allowing the user to create hierarchies of button menus. These ID directives propagate until the entire patch is programmed.

A button transmits a signal into the iSurface at large when it is pressed. The pressed state occurs when any part of the button area detects pressure. This transfer and preservation of state is accomplished by means of regular pulses triggered by any iCell detecting pressure, and should this pressure remain in place, then the agent periodically sends out further pulses. Pulses reset an internal timeout and turn an unpressed agent into a pressed one. The resulting system allows the user to "drag" the pressure around the button and to still preserve state. One might think that leaving a long pause will result in the timeout occurring before the pulse reaches the edge of the button, but, experimentally, this does not happen. On average, the maximum time it takes a pulse to spread throughout the button is ***sqr(width^2 + height^2)***. When this pulse reaches the agents in the interface, they enter the pressed mode and transmit a message containing the Button ID. Should another pulse arrive before the interface's timeout period expires, the timeout is reset but no additional Button ID message is sent.

2.3.2 Reaction Times. The time taken for the button to fully react to pressure depends on the size of the area used for the button and the position inside the button at which pressure occurs. Assuming a button with no delays in processing or transmitting the messages, we can estimate the time to total "infection" lies between ***sqr((1/2 width)^2 + (1/2 height)^2)*** **and** ***sqr(width^2 + height^2)*** message hops. However, the timeout is specified by (width + height) and on larger areas this will be a significant value and could result in unresponsive buttons. Like all excitable mediums, elements of time critical computation are best kept highly localised [8].

2.3.3 Coping With Unreliability. Due to the way applications work on the iSurface, reliability of the buttons is dependent on the state of the iCells that make up the button area and the physical positioning performed by the user. In terms of robustness to damage, experimentally the button is observed to operate as expected. When growing, the buttons attempt to flow around non-functional elements to form as close to a rectangle as possible. However, the button's growth is confined to its intended boundaries and is incapable of taking routes around obstacles that lie outside these borders. Should damage occur after the button has been created we observe essentially the same effect. Pulses flow around damaged areas, thus they may take longer routes than calculated by ***(width + height)***. This will affect the necessary timeout period. Damage that effectively severs the button into two (or more) pieces, has the result of creating independent buttons; all but one of which will be disconnected from the functional button area.

Load on the iCell is the largest obstacle to reliability of the button operation. When elements of the button are slow in passing on data, other elements will get whichever signal arrives first. However, the problem lies in the transmission from these overloaded iCells. It is observed that, when an overloaded iCell manages to transmit a pulse, its neighbours have completed their timeout period and reverted to the unpressed state. In this event, the button is likely to reactivate, despite the lack of current user pressure. Compensating for iCell load necessarily means a reduction in responsiveness for the button.

2.4 Video

Suggestions that ad-hoc networks of iCells working as iSurfaces could be used as displays for images or video have been discussed widely. At the heart of a video / picture application is a notion of positioning. That is, every iCell in the system that is to display part of the image, needs to know its position relative to some origin. This is the same origin that the image data refers to. This coordinate system is generated in exactly the same way as that described for the button agent. Figure 1 shows the video system in use. It has formed a square display area and contains a fully formed picture.

2.4.1 Encoding the Image. The image used for the experimentation is 256 pixels by 256 pixels and uses a greyscale palette (8-bit). This means there is a total data size of 65536 bytes (or 64k). If we choose to transmit the entire picture from iCell to iCell, we automatically take a crippling performance hit. On a Smart Dust based surface with a 10kbps transfer rate, we would have transmission times of about 53 seconds. Flood filling this data across a 256 x 256 surface would take between 13420 seconds (about 224 minutes) and 26840 seconds (about 447 minutes). The only advantage of this system is the ease with which an iCell can obtain its correct colour by simply indexing the image data at the correct point. We would argue that the only feasible alternative is to release a stream of pixel data into the system. Each message takes the form of a coordinate ('x' & 'y'), a pixel value (from 0 to 255), and takes up a total of 7 bytes. This results in a transmission time of about 0.007 seconds per message, on a Smart Dust based iCell. Although the amount of data entering the system has increased by a factor of 7, we have broken the data up into more manageable chunks. Flood filling a single pixel across the surface is relatively trivial, between 1.8 seconds and 3.6 seconds, but attempting this for all 65,536 pixels would literally take hours. In addition, each message would require a unique identifier to prevent message “flowback” and surface hyperactivity.

2.4.2 Directed Message Routing. The ideal solution would be to route the pixel data from the entry point(s) to the target iCell using a coordinate system and having coordinate values stored in the message. As each iCell on our iSurface can transmit messages in specific directions, an agent only needs to decide the proper direction from itself to the target and retransmit in one of the 8 available directions.

2.4.3 Streaming Data Onto The iSurface. Placing all the data on the surface in a single pass is unrealistic for a video application. Instead, we suggest that video would be streamed onto the iSurface and then routed to its destination. The problem we then face is how to get the data into the system in large enough quantities to be useful; that is, how many messages do we place into the system at once, and where do we place them? We have a total of 65536 pixels to place into the system. If we select a single iCell as an entry point and input one pixel message per cycle, it will obviously take 65536 cycles to completely enter the data into the system, let alone route the data to its destination. If we follow the quadrant approach to the iSurface and use a separate entry point for each quadrant, we get 16384 cycles. This is still unacceptable. Instead of targeting single iCells, we could target multiple iCells in patches, each iCell effectively becoming an additional channel into the iSurface.

In terms of performance, we have established experimentally that a system divided into quadrants, with strips of 128 iCells in each, and using a data encoding to take advantage of this structure, is the best approach, with a "score" of 128 cycles to form the image. As a cycle is the time taken to transmit one pixel message, we can calculate how long this would be in "real-time". On a Smart Dust based iCell we have a transmission time per cycle of 0.007 seconds. This gives us a time of 0.896 seconds to render a single frame; way over the 0.04 seconds required for convincing animation. Obviously, an iSurface based on Smart Dust couldn't cope with this application. However, we can calculate the communication speed necessary to run the application as it currently stands which would be 224,000 kbps, 22 times faster than Smart Dust and using Moore's Law we can guess at a target date of about 2012.

2.4.4 The Effect Of iCell Load and Damage on the Video System. The video system is time critical by nature. Data has to be reliably transmitted at a rate of 24 frames a second. iCell load is lethal for this system. Even a slight delay on a single pixel of the video area means delays for all pixels it has to pass data to. The result is that video data gets backlogged and compounds the problem. Compression solutions ease the data rate, but cause problems for the iCell processor and thus contribute to iCell load. Obviously improving bandwidth, and thus transmission times, will help relieve the load, as backlogs occur when data is entering the cell at a higher rate than it can clear it. However,

damage is more of a problem in this application than iCell load. Because the application utilises a simplified directed message routing system, there is no compensation for bypassing dead iCells. If an iCell in a column is dead, every iCell past it will no longer receive messages. This destroys a section of the display area, thus rendering the video system essentially useless.

2.5 Gesture Recognition

A simple gesture is defined as a straight line across the panel. It is composed of a start point and an end point, allowing us to obtain a two-dimensional vector from the start point. This form of gesture recognition tests the feasibility of propagating and collating information in a directed manner across the iSurface. This would be crucial to form the basis for a drag and drop interface. Figure 1 shows a gesture recognition panel in use. The drawn shape is identified and used to select an image for the video system to display.

2.5.1 Obtaining Gestures From The iSurface. Gestures are performed on a panel feature on the iSurface. The panel is seeded in the iSurface as a single agent, and grows a coordinate system in exactly the same way as the button and video agents. The result is a rectangular region of iSurface of specified dimensions, infested with gesture panel agents.

At the heart of this gesture recognition system is the "path list", a list detailing exactly which iCells have been touched in the process of making the gesture. When an iCell is first touched, the resident agent checks to see if it already possesses a path list. If it does, it appends its own coordinates to this list. If it doesn't already have a list, it constructs its own one with its coordinates being the sole member. The path list is transmitted to all surrounding iCells.

For the purpose of these experiments, only the first and last members of the path list form the vector from which the gesture is derived. This means that the gesture is considered to be a straight line between the start and end points, regardless of whether the path actually followed a straight line at all.

Having constructed a path list, we now need to determine whether the user has finished tracing their gesture. To do this, we need to give agents a notion of being the last iCell pressed. To this last iCell pressed, we delegate the responsibility of identifying the gesture and telling the rest of the panel about its findings. To determine the last iCell pressed, we automatically let each iCell that detects pressure, believe itself to be the last. When pressed, each iCell starts a timer. Once this expires, it processes the path list and sends the relevant results to the rest of the panel. To prevent every iCell that's been pressed doing this, an iCell will believe itself "not last" if it receives the path list from another iCell after it has sent out its own. This means that there has definitely been another iCell pressed after itself, and so its timer stops. The final iCell pressed will obviously not receive an additional path list, and will therefore wait until the timer expires and then process the path list. After the path list has been processed, a reset signal is sent and the panel returns to a receptive state.

2.5.2 Fragmentation of Gestures.

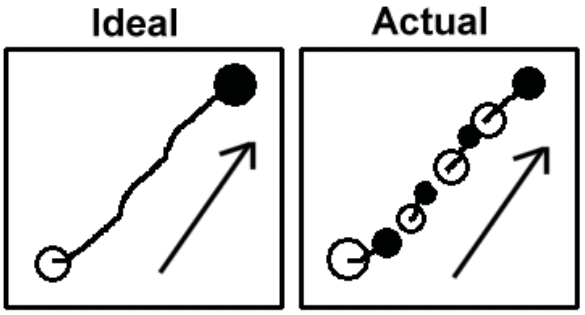

Figure 3. The left diagram shows an ideal gesture. It has a single start point (hollow circle) and end point (solid circle). In actuality, the line can become fragmented and have many start and end points.

As demonstrated in figure 3, a problem that becomes apparent when using the simulator on slower machines, is that strokes end up becoming fragmented. This occurs when the mouse (which controls pressure on the iCells) moves faster than the simulation can update, thus skipping over many iCells. However, this is also a problem likely to affect any real world application of the iSurface; a user may gesture too quickly, or fail to provide enough pressure along the length of the gesture. The result is a series of independent gestures, that know nothing about previous or subsequent gestures. This results in multiple "last pressed iCells" (one per gesture) trying to activate individually and potentially conflicting with each other. The problem is twofold; firstly, we need to stop all but the final pressed iCell from activating, and secondly, we need to obtain the original starting point so an uninterrupted gesture may be constructed. We solve this by adding an additional timer to the iCells, with a timeout period far longer than the existing one. Now, when a "last pressed" iCell reaches the end of its existing timer, it propagates the start point of its path list throughout the panel (this could be altered to just a small radius). All iCells that have been registered as "pressed" will replace their existing start point with the new one. If these iCells have already transmitted their start point in this fashion, they stop their timers and register themselves as "not last". If a "last pressed" iCell does not receive another start point after it transmits its own, it will wait until its longer timeout expires and will then process the gesture using its current start and end points. In theory, the oldest end point should trigger first, updating all other points. The next oldest will then trigger and "switch off" the oldest.

This continues until the newest point switches off the final false end points, and processes the gesture between the newest end point and the oldest start point.

We observed that, for smaller numbers of segments, the error is non-existent, but as the line becomes more fragmented the system becomes very unreliable. These failures are caused by the end points sending out their updates within short periods of each other. When these updates travel as a spreading message front, they collide and cancel out. The result is that an end point may actually never receive an update message because its own cancelled them out. The less fragments, the less chance of this happening.

The direct deciding factors on how well this system works, are the speed of iCell processing and communication; heavily affected by the iCell load. If either of these factors is unable to keep up with a user, the result is severe fragmentation. Increasing the longer timeout significantly will aid with this at the expense of responsiveness.

Damage to the iSurface would also cause fragmentation. If a path list can't be passed on through a dead iCell then a split in the gesture occurs. Again, dealing with fragmentation requires a decrease in responsiveness.

3. CONCLUSIONS

Clement and Nagpal's line drawing system can be successfully ported over to a "real-world" Amorphous Computer derivative such as the iSurface. The general principle of using this system to directly link two areas is sound; the resulting gradient and line are robust when dealing with pre-damaged surfaces and the gradient itself is successful at dealing with repairing itself. Clement and Nagpal themselves say that responsiveness is determined by the timestamps used to control the updating of the gradient. This has been shown to be true in this system. However, using this system as a communication channel between two or more points requires reliability and this would entail quick responses to damage, both diagnosis and repair. The obvious solution would be to increase the rate of gradient update passes. However, as we have seen, high data rates contribute heavily towards iCell load, which in turn lowers responsiveness of the system. To compound the problem, increased responsiveness means the timestamps expire sooner. iCell load delays the necessary messages to renew the timestamp, and so it expires. This is a false result caused by data being delayed.

Features, such as Buttons, can be successfully grown on the iSurface, and do provide some level of functionality. The buttons, created using the pulse based method, need to be physically small in order to retain some degree of responsiveness. iCell load is a major problem in iSurface applications and this system is no exception. An iCell in the button that doesn't retransmit within its neighbours' timeout period, risks causing message flowback and thus false button presses. Even if the retransmission occurs within the timeout, the delay may significantly affect the responsiveness of the button.

Time-critical applications, such as the video system, are at severe risk from iCell load. We can see that the backlog of image data caused by even a slight delay will ruin the quality of the video and compound the problem. Applications such as these can be rendered unusable by load. Damage is even more of a problem than load in structure dependent applications such as the video. As the very underlying concept relies on a specific constant network structure, the slightest change can cause massive problems; complete loss of functionality in an area not even damaged, for example.

Gesture recognition again demonstrates the problems of transmitting time-critical data across the iSurface. In this case, the physical action on the iSurface (the pressure) was acting too fast for the iSurface to build a cohesive model of the action. Compensating for this problem, which is exacerbated by iCell load, reduces responsiveness. The gesture recognition system, by relying on flood filling for parts of its communications, is not as badly at risk from damage as completely structure dependent applications, such as the video system. The systems involving directed message movement, as dictated by a coordinate system, fail when the underlying structure is changed. Undirected systems are able to cope with the network topology changing. However, with the undirected systems we have the problem of releasing large amounts of data into the system, and this contributes to iCell load.

A possible solution to the problem of iCell load would be to prioritise tasks on the iCell. This would involve the host iCell allocating its resources to the agents based on need, and there are established methods that could be adapted to do this. The result would be that areas of the iSurface specialised towards certain applications, which is certainly feasible. However, much of the common functionality, such as the gradient system presented in this paper, is time-critical. By assigning priority to one agent, the others will suffer, and, following the examples of this paper, these time-critical systems will have to become less responsive to compensate for the lower allocation of resources. The result may be optimal given the system, but it probably will not be acceptable in terms of responsiveness.

This problem is typical of applications proposed for an iSurface or similar devices. These applications demand responsiveness. To obtain this, the solution is usually to increase data rate. However, an increase in data rate can lead to iCell load and compound the error. A simple answer is to increase the capabilities of the iCells in terms of processing speed and communication throughput. Every time a similar problem occurs, the

answer would be to increase these capabilities. As has been suggested as a solution for iCell load in these experiments, increasing communication capability, and thus decreasing transmission time, will enable an iCell to clear its queue faster, and thus reduce backlogs. Eventually the capabilities required get so high, and thus the devices they require are so far into the future, that the applications themselves become redundant, or are achieved by other means, before the hardware needed becomes available.

4. REFERENCES

1. Callaghan, V., Clarke, G., Colley, M., and Hagras. H., 2001, "Embedding Intelligence, Research Issues for Ubiquitous Computing", The 1st Equator IRC Workshop on Ubiquitous Computing, Nottingham, UK
2. PARC, "MEMS / Smart Matter Research at PARC"
3. Masens. C., 2004, "Smart walls, smart windows, smart bricks and tiles..", SmartHouse magazine, Australia
4. Katzenelson, J. , 2000, "Notes on Amorphous Computing", MIT Artificial Intelligence Laboratory, USA
5. Pister, K., Kahn, J., and Boser, B., 1999, "Smart Dust: Wireless Networks of Millimeter-Scale Sensor Nodes." Highlight Article in 1999 Electronics Research Laboratory Research Summary",
6. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight, T., Nagpal, R., Rauch, E., Sussman, G., and Weiss, R., 2000, CACM, 43(5), 74-82
7. Butera, W., 2001, "Programming a Paintable Computer", PhD Thesis, MIT Media Lab, USA
8. Adamatzky, A., 2001, "Computing in Nonlinear Media and Automata Collectives", Institute of Physics Publishing, UK
9. Clement, L., and Nagpal, R., 2003, "Self-Assembly and Self-Repairing Topologies", Workshop on Adaptability in Multi-Agent Systems - RoboCup Australian Open, Australia