Brigham Young University
BYU ScholarsArchive

All Theses and Dissertations

2010-10-08

Pixel Based Note Taking through Perceptual Structure Inference

Mitchell Kent Harris
Brigham Young University - Provo

Follow this and additional works at: https://scholarsarchive.byu.edu/etd

Part of the Computer Sciences Commons

BYU ScholarsArchive Citation

Harris, Mitchell Kent, "Pixel Based Note Taking through Perceptual Structure Inference" (2010). *All Theses and Dissertations*. 2282.
https://scholarsarchive.byu.edu/etd/2282

This Thesis is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in All Theses and Dissertations by an authorized administrator of BYU ScholarsArchive. For more information, please contact scholarsarchive@byu.edu, ellen_amatangelo@byu.edu.

Pixel Based Note Taking through

Perceptual Structure Inference

Mitchell K. Harris

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Masters of Science

Dan R. Olsen Jr., Chair
Michael Jones
Dennis Ng

Department of Computer Science

Brigham Young University

December 2010

Copyright © 2010 Mitchell K. Harris

All Rights Reserved

ABSTRACT

Pixel Based Note Taking through

Perceptual Structure Inference

Mitchell K. Harris

Department of Computer Science

Masters of Science

Knowledge workers need effective annotation tools to assimilate information. Unfortunately many digital annotators are limited in the range of document that they accept. Those that do accept many different documents do so by converting documents to images, thus losing any awareness about the original content of the document. We introduce a digital note taker that is both universal and content aware. By constructing a hierarchical context tree of document images, the structure of a document is inferred from the image. This hierarchical context tree is shown to be useful by demonstrating how it facilitates selection of document elements, reflowing documents to accommodate inserted notes, and expanding the context of links. PixelJot, and implementation of these ideas, demonstrates their feasibility.

Keywords: Annotation, perception, HCI

Table of Contents

INTRODUCTION 1
Universality 2
Content Aware 3
Selection 4
Layout 6
Annotation pages and Visual links 8
Context expansion 10
Synopsis pages 12
Context 17
Thesis Statement 18
PRIOR WORK 19
PIXELJOT 27
IMPORTING 29
SEGMENTATION 31
Contextual Structure 31
Getting Contextual Structure 34
Quickly Finding Whitespace 38
Crop Cut 41
Grid Cut 44
Over segmentation 48
Hierarchically inferred cutting 51
LAYOUT 63
SELECTION 68
Selection by Bounding Box 68
Selection by Stroke 72
VISUAL LINKS AND CONTEXT EXPANSION 77
In Place Expansions 78
What to Expand to 79
Insignificant expansions 80
Surging expansions 81
Expansion results 83
Perceptual Scaling 84
Continuity histograms 88

SYNOPSIS PAGES 97
Index 98
Search 99
Summaries 100
Conclusion 107
Content Awareness 107
Usefulness of Content Awareness 107
FUTURE WORK 109
BIBLIOGRAPHY 110

INTRODUCTION

Knowledge workers gather and assimilate information. Since they represent an important and growing portion of the work force, simplifying and improving their tasks is a worthwhile goal. In past decades, their tools were paper, note cards, and white boards. More recently, physical media has been succeeded by digital document annotation applications. A digital annotation tool is a program that imports pre-existing documents and allows for gathering and aggregation of the users observations on the document. Highlighting and creating comments are common tools in annotators for gathering observations, while summaries, indexes, and searches are examples of aggregation tools. While digital annotation applications have many advantages, their inflexibility has limited their adoption. Many existing annotators are unable to universally work with all kinds of documents. Those annotators that do universally import documents do so by converting them to images. Once converted to pixels, these annotators are able to display the documents but cannot manipulate them or understand their structure. The structure of a document is entirely opaque to them. We introduce a method of making a digital document annotator that is able to both universally import all kinds of documents and understand their internal structure. Our implementation, shown in Figure 1, is called PixelJot. In the introduction we discuss the nature of the universality versus structure awareness impasse, and how our method claims to resolve it.

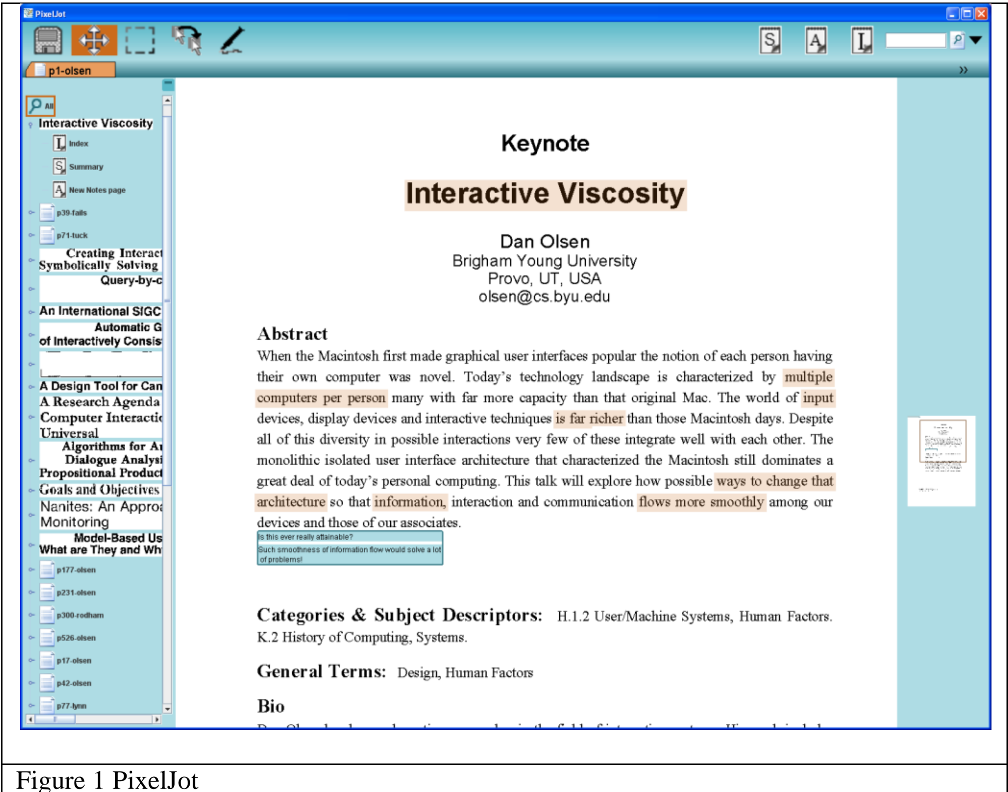


Figure 1 PixelJot

Universality

Annotating a document means adding comments, highlights, cross references, and other such notes on top of an existing document. This requires direct access to the source document. Documents that people wish to annotate come in many formats, as witnessed by their many extensions. Just a few are ".pdf", ".doc", ".docx", ".odf", ".txt", ".rtf", and ".xls". There are many more formats and undoubtedly in the future more formats will be created.

Unfortunately, this plethora of document formats makes importing difficult for annotators. For each additional format, the annotator must determine how to read and display it. This means that

many annotators are limited to working only with a limited set of document types, locking out many documents as sources.

Instead, we must have an annotator that is universal. A universal annotator is able to import all kinds of documents, regardless of format, and lay them side by side each other for inter-document cross referencing. Reading all kinds of document formats is a difficult problem. A special adapter could be written for each different format, but this would not support future formats, would take extensive expertise and special knowledge of the various formats, and may be impossible for proprietary formats.

To achieve universality, some existing annotators use a clever work-around. Rather than attempting to read all formats, they install a virtual printer and then "print" documents to their application. This virtual printer converts documents to images, which are easily imported by the annotator. All printable documents can be imported, allowing direct on-the-document note-taking.

Content Aware

There is a trade off involved with converting documents to images. Documents have an internal structure, storing where figures, paragraphs, and columns are. Images are entirely structureless. This lack of structure means that an annotator that converts documents to images loses its ability

to be aware of content of the document. Content awareness is when a document-to-image converting annotator is aware of the content or structure of it. Failure to be content aware limits an annotator's ability to assist with annotation tasks. For example, assisting in selection by snapping selection boxes to nearby words or paragraphs would require an annotator to be aware were words and paragraphs are. We focus on four operations that are very difficult or not possible on images, and thus are out of reach for non-content aware annotators. These are:

1. Selection
2. Layout
3. Context expansion
4. Synopsis pages

Selection

Selecting an element in a document is the first step for most annotation tasks. Highlighting a region is an example of a task that requires having a selection region. Without a selection technique, highlights would run outside of the text their intended for as in Figure 2.

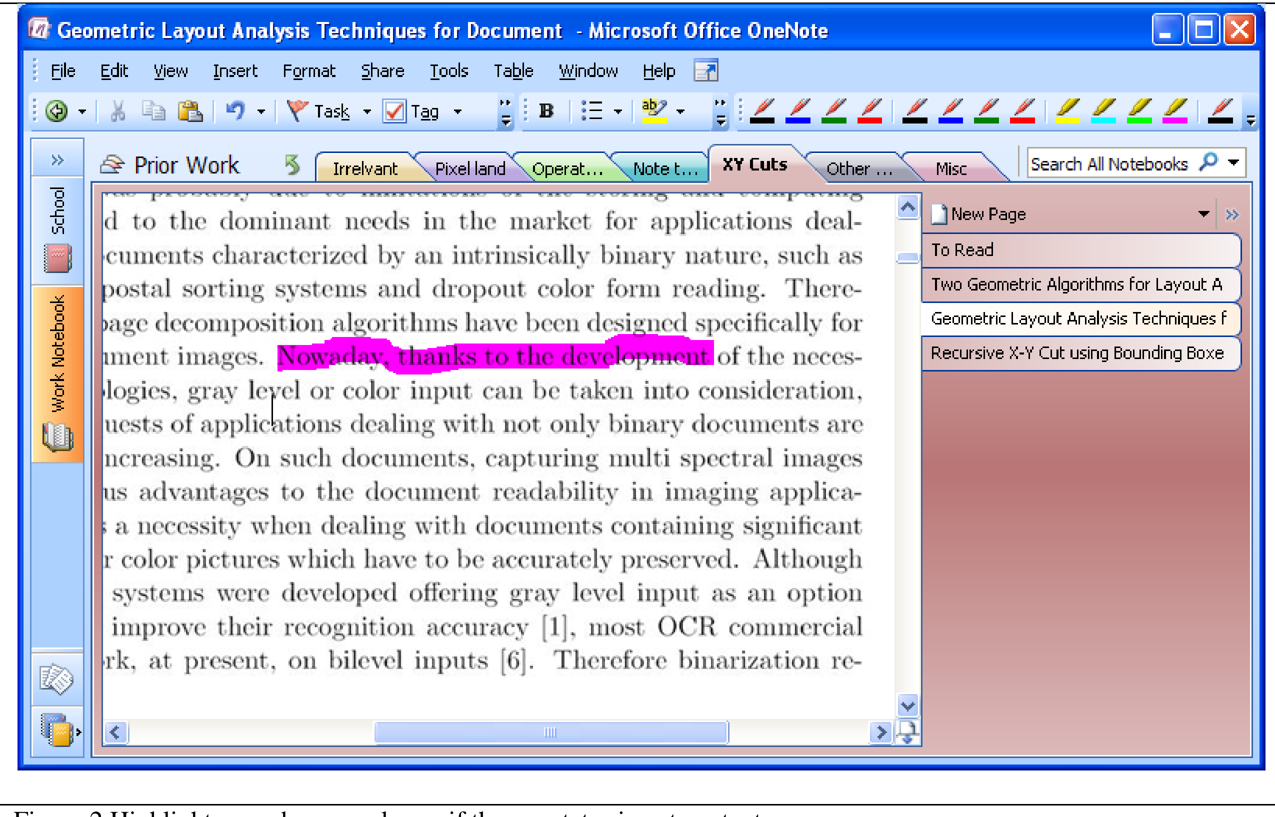


Figure 2 Highlights may become sloppy if the annotator is not content aware.

While a selection algorithm is necessary, selection of regions of images is a difficult problem. Existing selection techniques such as the lasso, bounding box, min-graph cut, and intelligent scissors are ill-suited for document based selection because, since they work on pixels, they are unaware of lines, paragraphs, and columns. They may select only half of a character or shave off an edge of a paragraph as seen in Figure 3. Instead document selection requires a technique that is aware of document primitives such as the characters, lines, paragraphs, or columns. Take Figure 3 and Figure 4 as an example. If a hasty user were to provide the selection rectangle shown in Figure 3, a good content aware selecting algorithm would automatically snap to the paragraph, the most likely element of the document that is being selected.

ortions of topics for . with a an supply described highlight otations as nformation ; image	rendered onto paper. In this context the process of annotating the paper received much less attention despite its importance in actual use. The advent of very cheap storage, cheap communication via the Internet, standard formats such as PDF or HTML and the pervasive availability of computing has caused a shift in our usage of documents. For an increasing number of people, the majority of their reading experience is digital rather than paper. Email has rapidly replaced the paper letter for much correspondence. Scholars increasingly subscribe to digital libraries rather than print journals. Technical manuals and promotional materials increasingly come through the web. Adler et al [1] have reported that reading occupies 70% of document-related activity. However, for many subjects a	ortions of topics for . with a an supply described highlight otations as nformation ; image	rendered onto paper. In this context the process of annotating the paper received much less attention despite its importance in actual use. The advent of very cheap storage, cheap communication via the Internet, standard formats such as PDF or HTML and the pervasive availability of computing has caused a shift in our usage of documents. For an increasing number of people, the majority of their reading experience is digital rather than paper. Email has rapidly replaced the paper letter for much correspondence. Scholars increasingly subscribe to digital libraries rather than print journals. Technical manuals and promotional materials increasingly come through the web. Adler et al [1] have reported that reading occupies 70% of document-related activity. However, for many subjects a
Figure 3 A selection bounding box drawn by a user		Figure 4 A good selection algorithm would snap the box to select the paragraph	

Unfortunately, because existing document image annotators are not content aware, they are unable to determine where the elements of the document lie and assist with selection. This operation is beyond the reach of non-content aware annotators.

Layout

Inserting comments into a tightly packed document can cause some readability issues. If the comment won't fit into the whitespace of a document, then either the comment would have to over flow over the body of the text as in Figure 5, or it would have to be resized until it fits in the whitespace. If the whitespace is small, the resized comment may become too small to be readable.

Instead of these two unacceptable options, it would be better if the document could be reformatted, flowing the paragraphs and columns to make room for the comment. Figure 5 demonstrates a rather lengthy comment that a user wishes to insert in the whitespace of a paragraph. The comment does not fit, but if a layout algorithm were to be included, the

comment could still be inserted and the paragraph below would be moved downwards to make room for the comment as in Figure 6.

Laser Pointer Interaction

Travis Nielsen
Computer S… Young University, Provo, UT
…cs.byu.edu

This is an idea that I have
Is there an easier way of doing this?
It does seem like thay did this pretty cheaply
Haven't WebCam's dropped in price since this paper
I'm glad that this comment block will make it into this area.

ABSTRACT
Group meetings and other non-desk situations require that people be able to interact at a distance from a display surface. This paper describes a technique using a laser pointer and a camera to accomplish just such interactions. Calibration techniques are given to synchronize the display and camera coordinates. A series of interactive techniques are described for navigation and entry of numbers, times, dates, text, enumerations and lists of items. The issues of hand jitter, detection error, slow sampling and latency are discussed in each of the interactive techniques.

Keywords
Laser pointer interaction, group interaction, camera-based interaction.

INTRODUCTION
A very interesting setting for interactive computing is in a meeting where the display is projected on the wall. Projection of the large image allows all participants sitting

equipment consists of a computer attached to a projector and a camera to detect the laser pointer position. We used a standard 1024 x 768 projector connected to a laptop PC. For the camera we used a $500 WebCam that can deliver up to 7 frames per second over TCP/IP. This camera connection is very slow, but adequate for our initial tests.

In addition to meeting situations, this technique is useful wherever the user is in a situation for which a large projected display is possible, but a local personal display would be awkward. Examples include a repair shop with service information displayed on the wall, a laboratory where instrument controls are displayed on the wall, or as an alternative to the traditional television IR remote. In situations where the hands are occupied, the laser could be mounted on the back of a half-finger glove with the actuator switch on the side of the glove. This would require use of the hand to point, but would eliminate searching for and grabbing the pointer.

Figure 5 The comment in blue needs to be inserted at the arrow point

Laser Pointer Interaction

Dan R. Olsen Jr. and Travis Nielsen
Computer Science Department, Brigham Young University, Provo, UT
{olsen, nielsent }@cs.byu.edu

ABSTRACT
Group meetings and other non-desk situations require that people be able to interact at a distance from a display surface. This paper describes a technique using a laser pointer and a camera to accomplish just such interactions. Calibration techniques are given to synchronize the display and camera coordinates. A series of interactive techniques are described for navigation and entry of numbers, times, dates, text, enumerations and lists of items. The issues of hand jitter, detection error, slow sampling and latency are discussed in each of the interactive techniques.

Keywords
Laser pointer interaction, group interaction, camera-based interaction.

INTRODUCTION
A very interesting setting for interactive computing is in a meeting where the display is projected on the wall. Projection of the large image allows all participants sitting in their chairs to see the information under discussion. This provides a shared environment that can ground the discussion and provides an equal discussion point for everyone. However, if the information is interactive, only one of the participants has control of the changes.

equipment consists of a computer attached to a projector and a camera to detect the laser pointer position. We used a standard 1024 x 768 projector connected to a laptop PC. For the camera we used a $500 WebCam that can deliver up to 7 frames per second over TCP/IP. This camera connection is very slow, but adequate for our initial tests.

This is an idea that I have
Is there an easier way of doing this?
It does seem like thay did this pretty cheaply
Haven't WebCam's dropped in price since this paper
I'm glad that this comment block will make it into this area.

In addition to meeting situations, this technique is useful wherever the user is in a situation for which a large projected display is possible, but a local personal display would be awkward. Examples include a repair shop with service information displayed on the wall, a laboratory where instrument controls are displayed on the wall, or as an alternative to the traditional television IR remote. In situations where the hands are occupied, the laser could be mounted on the back of a half-finger glove with the actuator switch on the side of the glove. This would require use of the hand to point, but would eliminate searching for and grabbing the pointer.

Figure 6 The paragraphs flow to make room for the inserted comment

An algorithm that reflows a document must know what each element of the document is and how they relate to each other. For example, in the case of inserting a comment that we just introduced, the layout algorithm needed to know the bounds of each of the paragraphs below the insertion point. If this paper has two columns the layout algorithm would also need to know to which column each paragraph belonged, so as to move the elements of one column but ignore the other. Since most document image annotators have only images of documents, but no structure about them (that is they are not content aware), most annotators are unable to reflow documents.

Annotation pages and Visual links

When reading a single document, most readers want to put annotations directly on the source document. When reading several documents, a reader may want to pool common ideas together on a separate page. PixelJot supports annotation pages. An annotation page is essentially a blank slate, ready for users to insert comments or drag in visual links from source documents. Figure 7 shows an annotation page with several comments and visual links.

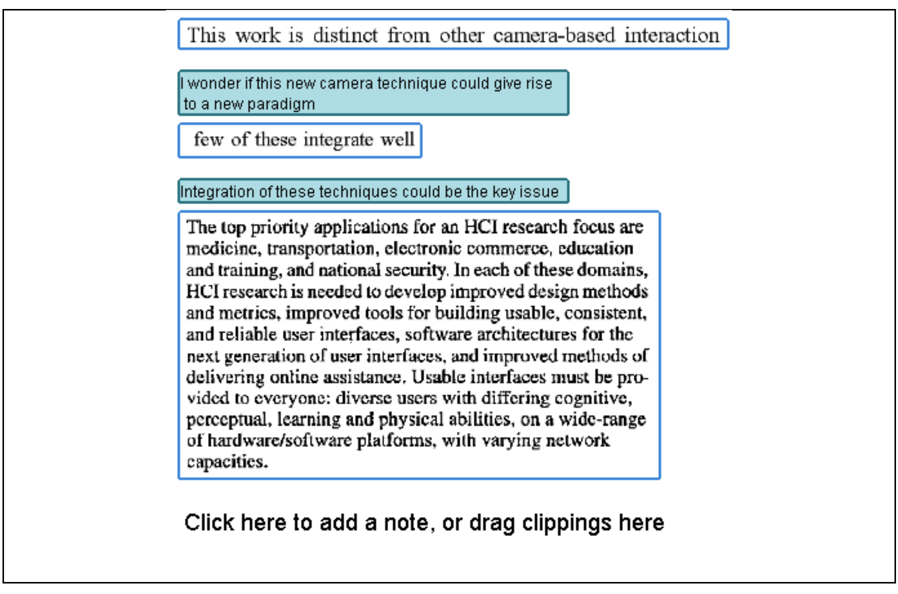


Figure 7 An annotation page with several notes (blue background) and clippings.

A visual link is a cross reference or link between or within documents that uses a visual portal into the target document rather than a text tag to visualize itself. To add a new visual link, the reader selects the area on a source document they wish to add (Figure 8), drags it to where they want to be placed in the annotation page (Figure 9).

Figure 8 A region is selected

The previously mentioned layout algorithm handles making room for the new visual link. In this case it shifts the text downward (Figure 9).

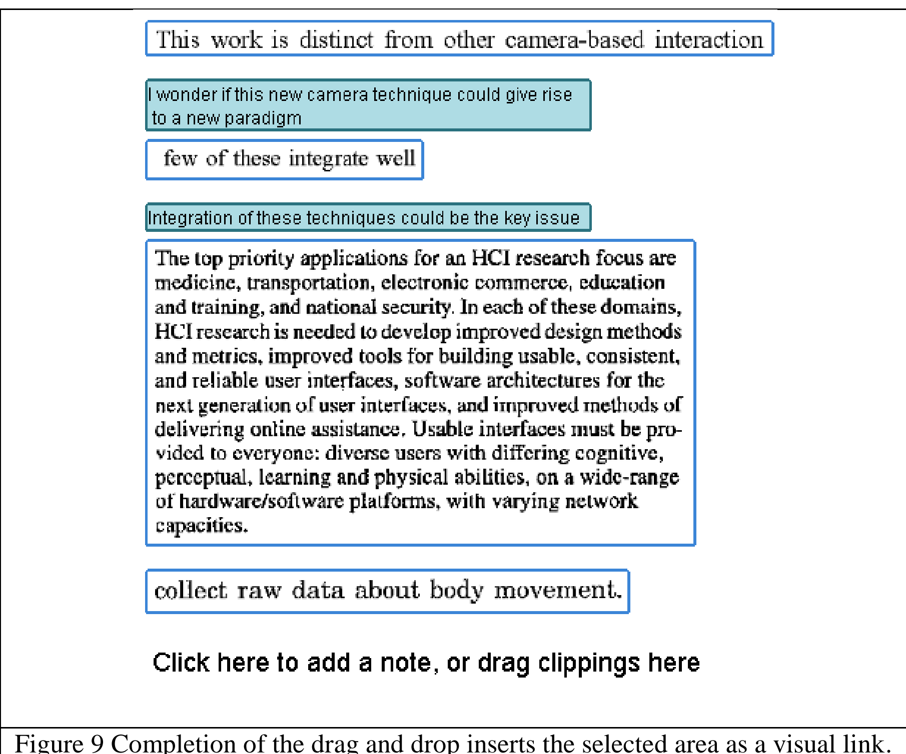


Figure 9 Completion of the drag and drop inserts the selected area as a visual link.

Double clicking on any of these links navigates back to the referenced document.

Context expansion

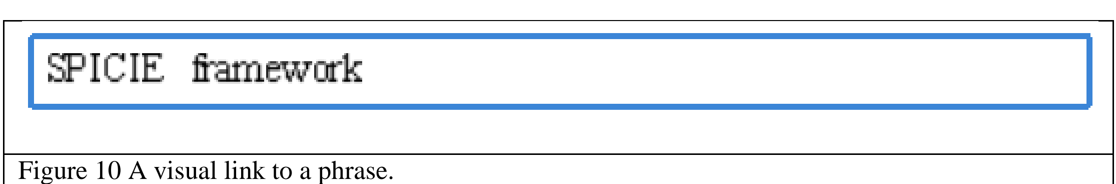


Figure 10 A visual link to a phrase.

PixelJot allows for an interesting operation on visual links called context expansion. This is one of the four operations that are not possible for non-content aware annotators. Context expansion means that rather than just navigating to an original document by clicking on a link, a user may expand the link to view the target's context. For example, clicking once on the link shown in

Figure 10 would expand the visual portal to show the entire line of text (Figure 11), clicking again would reveal the paragraph (Figure 12), and a third click would expand the visual portal to show the column (Figure 13). Subsequent clicks would show both of the columns together and then the entire page. Once expanded to the entire page, an additional click cycles the link back to its original portal.

This paper introduces the SPICIE framework for annexing

Figure 11 The visual link to a phrase shown in Figure 10 is expanded to show the entire line. The original phrase is selected with brown boxes.

ABSTRACT

This paper introduces the SPICIE framework for annexing display servers and sharing content on available screens. SPICIE allows a user carrying a portable device, such as a laptop or tablet, to annex additional screen space for her device. She then selects windows on her device to share with the annexed screens. In particular, overlapping windows on her personal device may be spread out (de-multiplexed) so that they do not overlap on the annexed screens. SPICIE protects user privacy by ensuring that only pixels generated by explicitly shared windows are transmitted to the display server. Multiple users may also simultaneously annex the screens to share content.

Figure 12 The line is expanded to show the paragraph.

ABSTRACT
This paper introduces the SPICIE framework for annexing display servers and sharing content on available screens. SPICIE allows a user carrying a portable device, such as a laptop or tablet, to annex additional screen space for her device. She then selects windows on her device to share with the annexed screens. In particular, overlapping windows on her personal device may be spread out (de-multiplexed) so that they do not overlap on the annexed screens. SPICIE protects user privacy by ensuring that only pixels generated by explicitly shared windows are transmitted to the display server. Multiple users may also simultaneously annex the screens to share content.

Author Keywords
de-multiplex pixels, annex screen, protect privacy

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
People are increasingly nomadic, and must manage large quantities of information. In particular, people need their personal information with them wherever they are. An effective technique for keeping information available is to carry that information with the user on a portable computer such as a laptop or tablet. These *personal devices* have portable form factors, but have limited screen space. Consequently, users are challenged to manage the increasing amount of information on a restricted screen space.

Most laptops and tablets have a Video Graphics Array (VGA) connector that can be used to add a single screen. A VGA connector increases the number of pixels available, but limits users to one additional screen and to the number of pixels on that screen that both the personal device and

Figure 13 The paragraph is expanded to show the column.

This context expansion is impossible for non-content aware annotators because they have no way of finding what the appropriate context of an element is. Given a bounding box on an image, a non-content aware annotator would not know how far to expand the bounding box to capture the next level up of context.

Synopsis pages

The last operation that is impossible for non-content aware annotators is generating sophisticated synopses of annotated documents. A synopsis page is any kind of condensation of an entire document. An index or summary are a kind of synopsis. Synopsis are based on tags or comments. While annotating documents, a user will often wish to select an element and tag it. A

tag associates a region with a string for later search. Figure 14 demonstrates what a tag looks like in PixelJot.

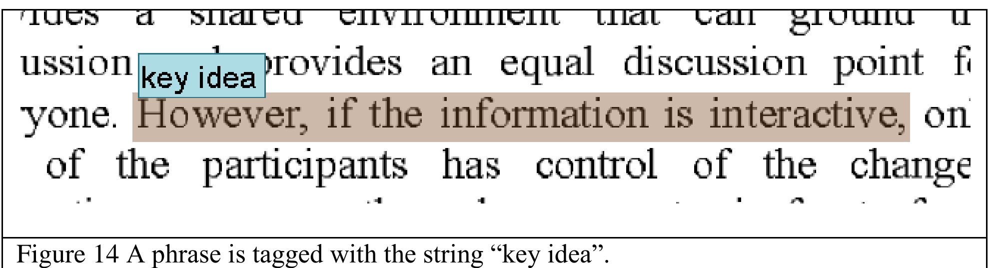


Figure 14 A phrase is tagged with the string “key idea”.

There are three kinds of synopses in PixelJot.

1. Index Page
2. Search Page
3. Summary page

To show an example of each of these pages, we’ll need a sample annotated document. Figure 15 provides a document with several tags spread throughout.

title

De-multiplexing Pixels: Wirelessly Expanding Portable Screen Space

Richard B. Arthur, Mitchell K. Harris, Dan R. Olsen, Jr.
Brigham Young University
3361 TMCB, Provo, UT, 84602-6576, USA
startether@startether.com, heneryville@gmail.com, olsen@cs.byu.edu

ABSTRACT key idea

This paper introduces the SPICIE framework for annexing display servers and sharing content on available screens. SPICIE allows a user carrying a portable device, such as a laptop or tablet, to annex additional screen space for her device. She then selects windows on her device to share with the annexed screens. In particular, overlapping windows on her personal device may be spread out (de-multiplexed) so that they do not overlap on the annexed screens. SPICIE protects user privacy by ensuring that only pixels generated by explicitly shared windows are transmitted to the display server. Multiple users may also simultaneously annex the screens to share content.

Author Keywords

de-multiplex pixels, annex screen, protect privacy

ACM Classification Keywords

screen support.

Desktop machines, however, can support extra graphics hardware, allowing users to greatly increase the screen space available for applications. Unfortunately, desktop machines are not very mobile, so personal devices are still limited to a single additional screen.

Inst[…] a single screen limit, a user should be able to key idea to meet her need for space. This paper introduces Spaces for Interactive Computing In Education (SPICIE): a wireless screen annexation protocol that allows users to utilize multiple screens via a p Unimportant s overcoming the cable limitations. SPICIE facilitates *de-multiplexing pixels*, which allows users to spread out windows from their personal device to the attached screens. Spreading these windows provides users context while

Figure 15 An example document with various tags.

An index is a page composed of a visual link to each tag in the document in sorted in alphabetical order of the tag string. Figure 16 shows an index page based on the document from Figure 15.

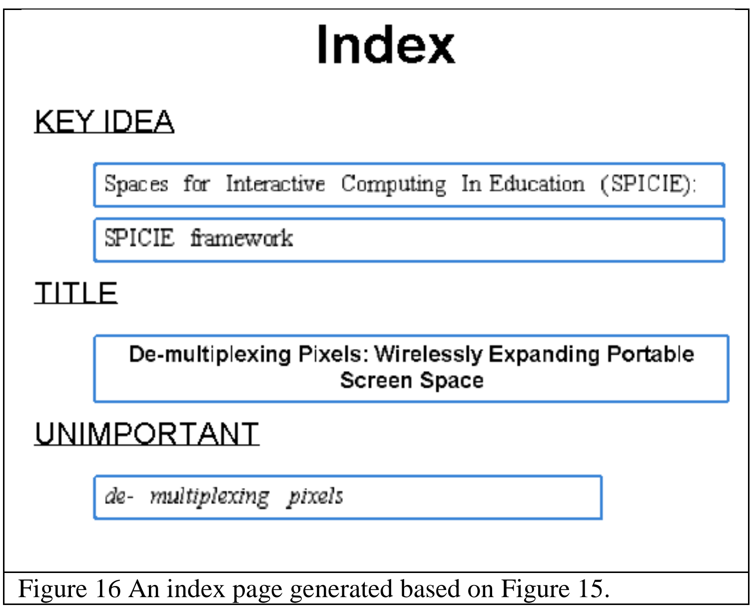


Figure 16 An index page generated based on Figure 15.

A search page allows a user to search for tags within a document, folder, or all documents for a given string. Figure 17 demonstrates a search of the document in Figure 15 for the string "Key Idea".

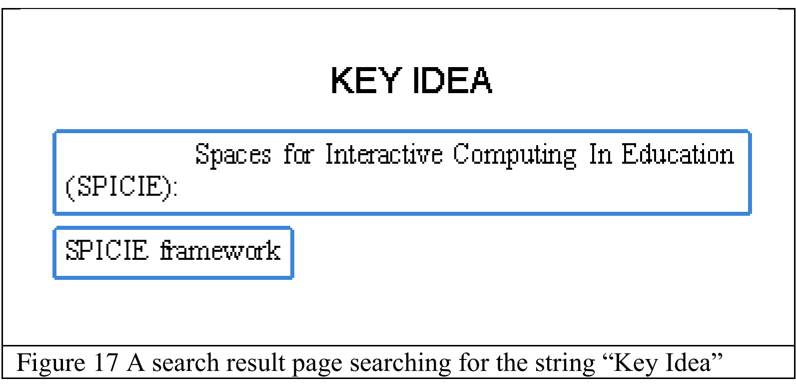


Figure 17 A search result page searching for the string "Key Idea"

A summary page lists visual links to tags, highlights, and comments of a document in order of appearance in the document.

Tag: title

De-multiplexing Pixels; Wirelessly Expanding Portable Screen Space

Tag: key idea

SPICIE framework

Tag: key idea

Spaces for Interactive Computing In Education (SPICIE):

Tag: Unimportant

de- multiplexing pixels

Figure 18 A summary page for the document in Figure 15.

Need for content awareness in synopsis pages

Synopses are impossible for a non-content aware annotator to use. The most obvious reason is that synopses depend on tagging and visual links. Tagging requires a good selection algorithm, which, as discussed earlier, requires content awareness. Visual links require content awareness to make context expansion work. Thus an annotator that is not content aware would be unable to offer synopsis pages.

Summaries have a special and unique need for content awareness. Summaries list tags, highlights, and comments in a document in the order that they appear. Ordering annotations is not possible without understanding the flow order of a document. Look again at Figure 15 and Figure 18. Notice that this summary listed the tag in the left column before the tags in the right, and the tag of the title before all of these. To achieve this, the algorithm that created the summary needs to understand the flow order, and therefore the structure, of a document. Creating synopsis pages, especially the summary page, requires content awareness.

Selection, layout, context expansion, and synopsis pages are all useful operations that are out of reach for annotators that store documents only as images. This introduces a tradeoff. Either the annotator can universally import documents by converting them to images, and thus lose the power to do these four operations, or it may attempt some other format-specific method of importing, and lose its ability to be universal.

Context

Let us suppose that there is a system for universally importing documents *and* getting structure with them. What kind of structure is needed? Clearly in order to support selection the structure is going to need to be aware of the fundamental elements of a document, such as the characters, lines, paragraphs, and columns. In order to support context expansion, the structure needs to know the contextual hierarchy of a document. To expand the link in Figure 10 to the scope shown in Figure 11 a context expansion algorithm needs to know the line to which the phrase belongs. To expand again to Figure 12 it would need to know the paragraph to which the line belongs.

In fact, a closer look at all four of these operations shows that they all need to know contextual hierarchy. Layout requires knowing the column to which a paragraph belongs if it is to move only the paragraphs within an affected column. Again, this requires knowing the hierarchy of the document. For the synopsis page algorithm to create summaries it needs to know the flow

order of the page. That requires knowing how the document is organized, and how each element is ordered compared to another. That, in short, is a kind of contextual hierarchy. In order to make these four operations possible, an annotator needs to know the contextual hierarchy of a document.

Thesis Statement

We introduce a method that allows annotators to be both universal and content aware. By importing documents as images this system is universal. We introduce a document understanding algorithm for segmenting the document image into individual atomic pieces. Segmentation of the document results in a hierarchical context tree (HCT) which is a data structure that stores the segmentation of elements of a document, and their relationship to each other. The HCT allows for content aware operations such as selection, layout, context expansion, and synopsis pages while still being a universal annotator. We have created PixelJot, an implementation these ideas, to demonstrate the feasibility of creating an HCT and the tools that use it.

PRIOR WORK

There are many commercial note taking applications [1-4, 6]. Most allow for text input, arbitrary attachments, tagging pages, hyperlinks between pages or to websites, and hierarchal grouping of notes into notebooks, sections, tabs, and pages. However, most of these note taking applications are not good annotators. In order to understand why, we will need to define the difference between the two. Like an annotator, a note taker is a tool that allows for gathering and aggregation of observations. However, unlike an annotator, a note taker does not import the object that is the subject of the observations; it simply records the observation apart from the subject. While this frees a note taker from the complexities of importing documents, it also is the cause of their most fundamental flaw. That is that note takers lead to a high degree of duplication, the burden of which is placed on the user. Note takers force the user to first create a "pointer" to the content they wish to annotate before recording observations. Typically this "pointer" is a recreation of part of the document, such as a brief description. Take for example Zoho notebook. To takes notes on a paper, the user in Figure 19 first had to write what the paper was about (the first paragraph, which is the "pointer"), then write his observation in the second paragraph. Obviously this is tedious and inefficient. Because annotators import documents and allow for annotation directly on top of the document, this eliminates inefficiency by not forcing users to recreate the elements they wish to annotate.

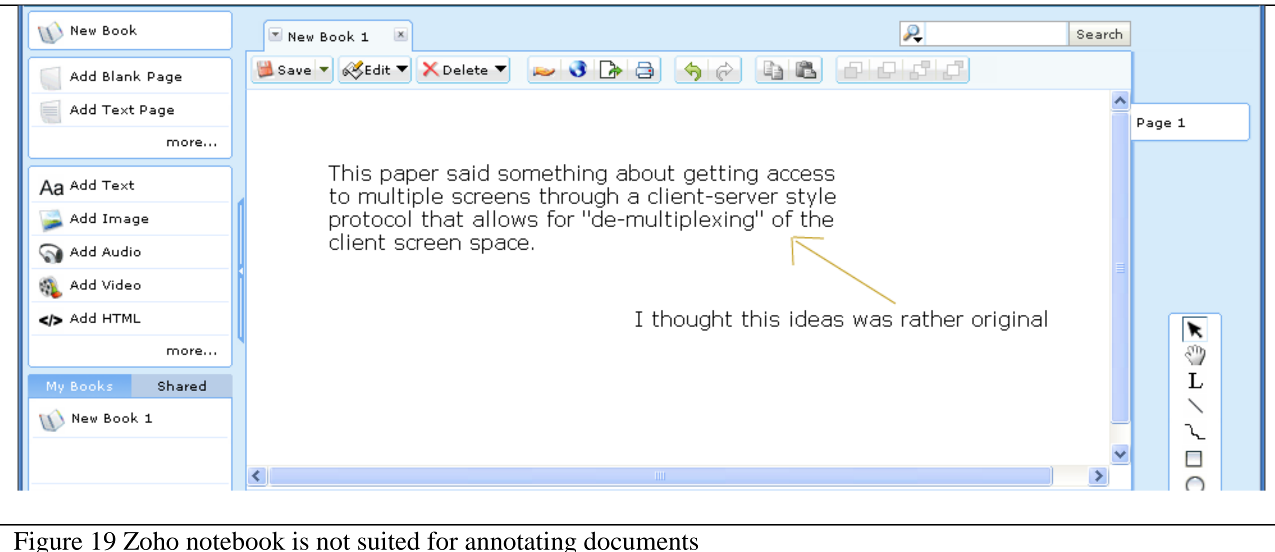


Figure 19 Zoho notebook is not suited for annotating documents

All of the commercial applications the author has encountered do little to nothing to be content aware. Images are often entirely immutable. A representative example of these commercial annotators is Evernote [1]. As seen in Figure 20, Evernote has embedded a PDF document in the top page, but is unable to select its text or interact with its content in anyway. In the lower page of Figure 20, Evernote has imported a spreadsheet as an image. This image is not editable beyond basic image processing operations such as contrast and brightness levels. Once in image form, the content and context are inaccessible. OneNote [4] does offer limited interactions with images such as search indexing via OCR and digital inking over the top. While OCR implies a slight understanding of the content of images because it can find the words, it is agnostic to the context of the words within the document.

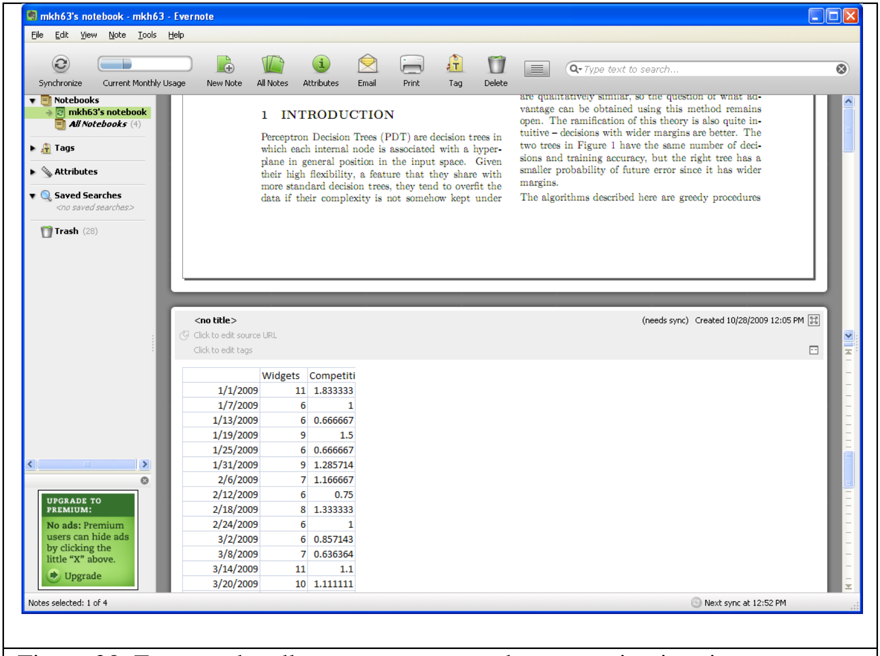
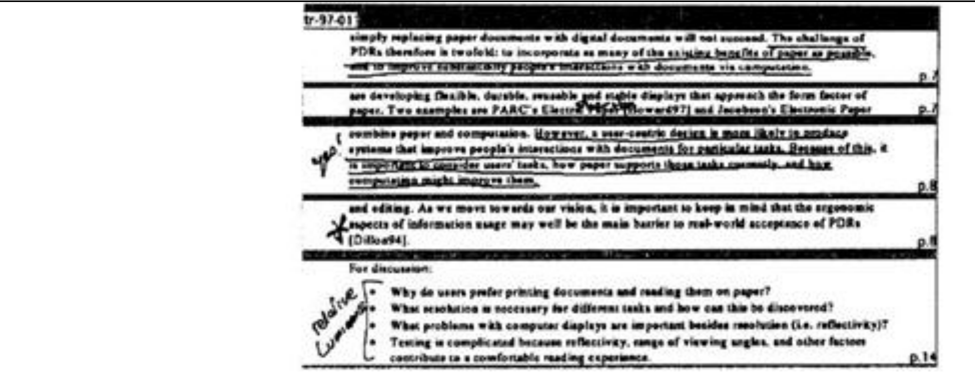


Figure 20 Evernote handles non-text content by converting it to images.

The research community has also offered several solutions to the limitations of annotators. XLibris [14] is a system for annotating digital documents as if they were paper. It is meant to be used on a tablet device where the user writes with freeform digital ink annotations to mark over images of documents. Xlibris is able to annotate nearly any kind of document because of its image representation. XLibris introduces clippings as a visual cutout of sections of a document. These clippings are a precursor to visual links. XLibris will auto-generate clippings (as seen in Figure 21) for each document by finding the bounding boxes of annotations and extending them across the width of the page. Clippings are thus always extended to the full width of the paper; causing the bounding box to extend beyond their column in multi-column documents, capturing content irrelevant to the annotation (see Figure 22). Clippings are added to the beginning of documents as a summary of annotations. Clicking on a clipping will jump the XLibris reader to that part of the document. XLibris has no support for being aware of words, lines, paragraphs, or

columns. Clipping pages are not editable documents themselves and may not be generated by users for any purpose besides a pre-document summary.


tr-97-01

simply replacing paper documents with digital documents will not succeed. The challenge of PDRs therefore is twofold: to incorporate as many of the existing benefits of paper as possible, and to improve substantially people's interactions with documents via computation. p.7

are developing flexible, durable, reusable and stable displays that approach the form factor of paper. Two examples are PARC's Electric Paper [Howard97] and Jacobson's Electronic Paper p.7

combine paper and computation. However, a user-centric design is more likely to produce systems that improve people's interactions with documents for particular tasks. Because of this, it is important to consider users' tasks, how paper supports those tasks currently, and how computation might improve them. p.8

and editing. As we move towards our vision, it is important to keep in mind that the ergonomic aspects of information usage may well be the main barrier to real-world acceptance of PDRs [Dillon94]. p.8

For discussion:

- Why do users prefer printing documents and reading them on paper?
- What resolution is necessary for different tasks and how can this be discovered?
- What problems with computer displays are important besides resolution (i.e. reflectivity)?
- Testing is complicated because reflectivity, range of viewing angles, and other factors contribute to a comfortable reading experience. p.14

Figure 21 XLibris allows for clippings as visual summaries of documents. Image as provided by XLibris.

Laser Pointer Interaction

Dan R. Olsen Jr. and Travis Nielsen
Computer Science Department, Brigham Young University, Provo, UT
{olsen, nielsent }@cs.byu.edu

ABSTRACT
Group meetings and other non-desk situations require that people be able to interact at a distance from a display surface. This paper describes a technique using a laser pointer and a camera to accomplish just such interactions. Calibration techniques are given to synchronize the display and camera coordinates. A series of interactive techniques are described for navigation and entry of numbers, times, dates, text, enumerations and lists of items. The issues of hand jitter, detection error, slow sampling and latency are discussed in each of the interactive techniques.

equipment consists of a computer attached to a projector and a camera to detect the laser pointer position. We used a standard 1024 x 768 projector connected to a laptop PC. For the camera we used a $500 WebCam that can deliver up to 7 frames per second over TCP/IP. This camera connection is very slow, but adequate for our initial tests.

In addition to meeting situations, this technique is useful wherever the user is in a situation for which a large projected display is possible, but a local personal display would be awkward. Examples include a repair shop with service information displayed on the wall, a laboratory

Figure 22 In XLibris, an annotation like the red circle will project a clipping across the entire page. The green box would be the resultant clipping.

ScanScribe [13] is another note taking application made by the research community. ScanScribe is intended primarily for sketches, whiteboard images, and scribbled text. ScanScribe only accepts images and has no method of converting documents into images. Users must take screen shots, save them to file, and open them with ScanScribe. ScanScribe's main contribution is that it is content aware, even better, it is able to determine a limited degree of context for elements in

images. ScanScribe thresholds and segments documents upon import and discovers connected components. ScanScribe will then attempt to group connected components to assist selection. For example, clicking on an entry in the table in Figure 23 selects a single digit. Subsequent clicks select the entire entry (as shown) then the entire table. If the naturally inferred groupings are not sufficient, ScanScribe will learn groupings by users' explicit selections.   Many natural groupings are not discovered by the automatic grouper. ScanScribe's context discovery is optimized for informal sketch-like note taking rather than document annotation, and thus performs very poorly on images of documents. It does not understand that documents are composed of characters, words, lines, paragraphs, columns, etc. ScanScribe will normally successfully group characters into words, but it never groups words into lines, lines into paragraphs, or paragraphs into columns. It is thus fundamentally limited when working with documents.

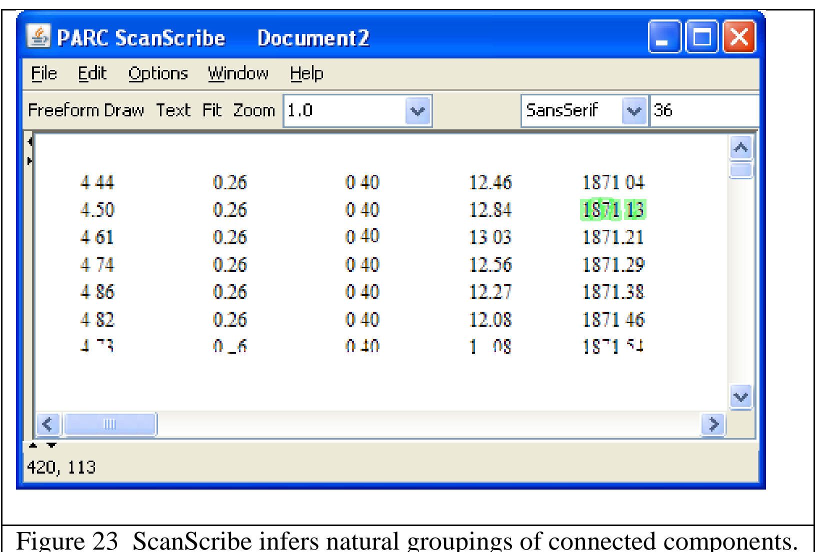


Figure 23 ScanScribe infers natural groupings of connected components.

ScreenCrayons [12] is an annotator that takes screen shots of the desktop and allows for marking these images with highlights, underlines, circles, and margin bars. ScreenCrayons is universally

able to work with any application or document because it imports documents and applications as images. It does not provide for linking documents, but, like XLibris, does provide a method of summarizing documents using clippings of annotations. ScreenCrayons only allows for interacting with images in the form of inking marks or typed tags. However, ScreenCrayons is somewhat content aware because it examines underlying images to assist selection and highlighting. When users use ink or selection, the document image is examined and the ink or selection is adjusted to snap to elements of the document. The context of these elements is never infered so ScreenCrayons cannot expand links (or clippings in the case of ScreenCrayons) or re-layout pages. We will use the whitespace identification algorithm found in ScreenCrayons in our work.

XEROX operates a private long-distance tele-
phone system called Intelnet. The netword con-
sists of 12 interconnected AT&T telephone switch-

Figure 24 Image Emacs segments individual characters.

Image Emacs [7] is an image based text editor that works entirely on images of documents. It is not a note taking application, but it does demonstrate novel interaction on document images. When a document is loaded by Image Emacs, it will segment the document into columns, lines and characters. As seen in Figure 24 the recognition each character has been surrounded by a tight bounding box. Image Emacs allows text editing operations such as copy, paste, typing and reflowing by moving or replicating images of characters within the document. It never uses OCR, but is able to function on images of documents almost as easily as Microsoft Word [5] does on ".docx" files. This reflowing demonstrates a high degree of content awareness in

documents. However, ImageEmacs is not an annotator. Instead it is an editing and authoring tool. Its purpose is not to gather and aggregate observations on a document, but rather to open an existing document and modify it.

A similar system to ImageEmacs is Paper to PDA [8] which converts documents into a series of word-sized images and then embeds the images in an HTML document. The result is an HTML document that flows to match display devices of any size.

UpLib [9] is a digital library that stores documents in many different "projections". Some of these are an image of the document, OCR text, or metadata. The published work is unclear about the granularity of linking or the degree of content awareness of this solution. The document capture system seems robust enough to at least partially capture arbitrary documents, but it is unclear if it will capture applications.

The paper "Hierarchical Representation of Optically Scanned Documents" [10] is a founding work in document understanding using recursive cutting. Recursive cutting repeatedly divides a document into smaller and smaller chunks by slicing along whitespace areas. The smaller chunks are sliced again and again until the document is indivisible. Their system is optimized for optically scanned documents, and includes cuts that are not axis aligned. While a precursor to our own context inference system, this work makes no attempt to create a structure suitable for note taking. Often its document representation varies from how we, as humans, would hierarchically segment a document. While it isolates each of the pieces of a document, it does

not stack them the way we would by putting words as members of lines and lines as members of paragraphs.

Thus none of the existing annotators adequately interpret document content to support annotation. Most do not discover the context of document elements at all, and those that attempt to work poorly with documents, being more specialized for ink style notes. What is needed is an annotator that can universally import all kinds of documents, and then be sufficiently content aware to discover the context of document elements. This annotator should be specially designed to support annotations tasks.

PIXELJOT

Our solution to the content awareness issue is called PixelJot. It is able to be both universal and content aware. By being universal we can annotate any document, regardless of the file format of the source file. Because it is content aware, it can understand and manipulate the internal elements of documents. For example, PixelJot can use this content awareness to assist in selection by snapping selection rectangles to nearby elements in the document. PixelJot can insert comments or visual links into a document and reflow lines, paragraphs, and columns to accommodate the insertion. PixelJot can expand the scope of visual links to show surrounding context. Additionally PixelJot can create synopsis pages that summarize the contents of a document respecting the document flow order. This chapter introduces the core components that makes our solution work, thus enabling selection, layout, context expansion, and synopsis pages. Subsequent chapters will expound on each of these core components.

PixelJot uses the previously introduced trick of installing a virtual printer driver and printing documents to a series of images. This step makes PixelJot universal. Importing documents as images introduces a memory versus resolution tradeoff, which the Importing chapter explains and resolves.

Becoming content aware is a much more difficult issue. Recall that the goal is to obtain a contextual hierarchy of a document, but all that is available is the image of the document. PixelJot uses perceptual clues to mine out contextual structure from images of documents. We

call this dividing of the document segmentation. The Segmentation chapter introduces the concept of a hierarchical context tree (HCT), how it can be used to find the contextual hierarchy of a document, and how it is constructed.

Once the document image is segmented and an HCT is created, our four key operations become possible, namely: layout, selection, context expansion, and synopsis pages. The Layout chapter discusses how PixelJot is able to use an HCT to reflow documents when accommodating insertions of notes. The Selection chapter introduces a selection algorithm that uses an HCT for snapping user selection bounding boxes to nearby elements of a document, therefore assisting in selection. It also introduces a selection technique that accepts highlighter-like strokes as input. The Visual Links chapter formalizes the concept of visual links first encountered in the introduction, and then pioneers an algorithm for context expansion of links using the HCT as a guide. The Synopsis Pages chapter further explores how the index, search, and summary pages are generated.

IMPORTING

PixelJot is able to universally import all kinds of documents by using a special printer driver. This virtual printer driver, created by Code-Industry[2], allows users to “print” their document to images, which will then be imported by PixelJot. This is an established method for note-takers to import arbitrary documents employed by applications such as Microsoft OneNote [4]. Converting documents to images introduces a major memory constraint. The virtual printer we use allows for four quality settings: 100x100, 200x200, 300x300 and 600x600 dots per inch (DPI). Of course more DPI provides greater quality, but it also consumes significantly more memory. The table in Figure 25 shows the amount of memory necessary for each of these resolution settings.

DPI	Pixels Width	Pixel Height	Total Pixels	MB per Image
100 x 100	850	1100	935,000	3.6
200 x 200	1700	2200	3,740,000	14.3
300 x 300	2550	3300	8,415,000	32.1
600 x 600	5100	6600	33,660,000	128.4

Figure 25 Table describing memory requirements of images

These values are for a single standard 8.5”x11” page, but many documents contain several pages. For example, a research conference paper can be about 10 pages long and thus at 600x600 DPI it will require 1284 MB to hold in memory. A journal article of 40 pages could take as much as 5.1 GB at 600x600 DPI. Memory is clearly a constraining factor.

However, quality of the image must be balanced as well. The table in Figure 26 shows a phrase rasterized at various levels of quality. By looking very closely at the 100 x 100 DPI you may see some blockiness. This blockiness almost entirely disappears at the 200 x 200 DPI level and above.

Sample Text	DPI
This paper	100 x 100
This paper	200 x 200
This paper	300 x 300
This paper	600 x 600

Figure 26 Visual results of various printing resolutions.

PixelJot uses the 200 x 200 DPI resolution setting.

SEGMENTATION

Converting documents into images allows PixelJot to universally import any kind of document. However, this comes at the price of "flattening" documents into pixels. Once projected to pixels, the structure of the document is lost. Content aware annotators need structure to make some features work such as reflowing documents and context expansion of links. The immediate questions are what kind of structure is needed and how can we get it?

Contextual Structure

The level of structure that PixelJot needs is to know what the various pieces of a document are, and how the pieces relate to each other. It needs to know how the pieces combine together to compose other larger pieces. For example, suppose a user selects a portion of a document, then drags it to another document thus making a visual link similar to that seen in Figure 27. Clicking on the link once will expand the link to show not just the word, but also the entire line as shown in Figure 28. Another click would expand the line to show the entire paragraph as in Figure 29. To do this PixelJot needs to know several things. First, it needs to know that the word is an atomic, selectable piece of the document. Second, it needs to know that this particular word belongs to the line shown in Figure 28. Finally, it needs to know that the line of Figure 28 belongs to the paragraph shown in Figure 29.

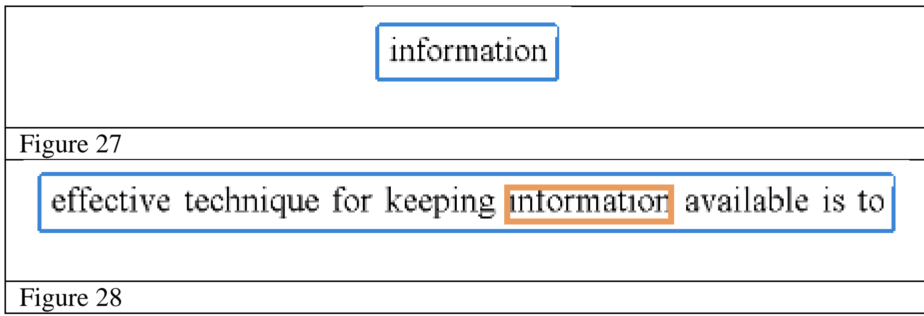


Figure 27

Figure 28

People are increasingly nomadic, and must manage large quantities of information. In particular, people need their personal information with them wherever they are. An effective technique for keeping information available is to carry that information with the user on a portable computer such as a laptop or tablet. These *personal devices* have portable form factors, but have limited screen space. Consequently, users are challenged to manage the increasing amount of information on a restricted screen

Figure 29

Note that PixelJot does not need to identify each piece. To expand context from a line to the paragraph it is not necessary to know that one is a line and the other a paragraph, just that one belongs to another. This is called the contextual structure. PixelJot stores this context in a data structure called a hierarchical context tree (HCT). An HCT is a tree whose root is the entire document, and leaves are the atomic and most basic pieces of the document: the individual characters or figures. Each level in-between represents a grouping of these basic pieces. Let's look at the document in Figure 31 with its corresponding HCT in Figure 30. Regions in Figure 31 have been colored to match their corresponding nodes in Figure 30. The root of this tree is the entire document.

The document is composed of four margins (shown in green) and the document body. The document body is composed of the title section, text body, and a gutter (shown in orange) which divides the title and text body. Further segmentation of the text body reveals a layer that separates the columns and another layer for separating the paragraphs. The paragraphs are

broken into lines, which are broken into words. While not shown in this tree, the words are divided into characters. This HCT allows for each element of a document to be placed in context of its neighbors.

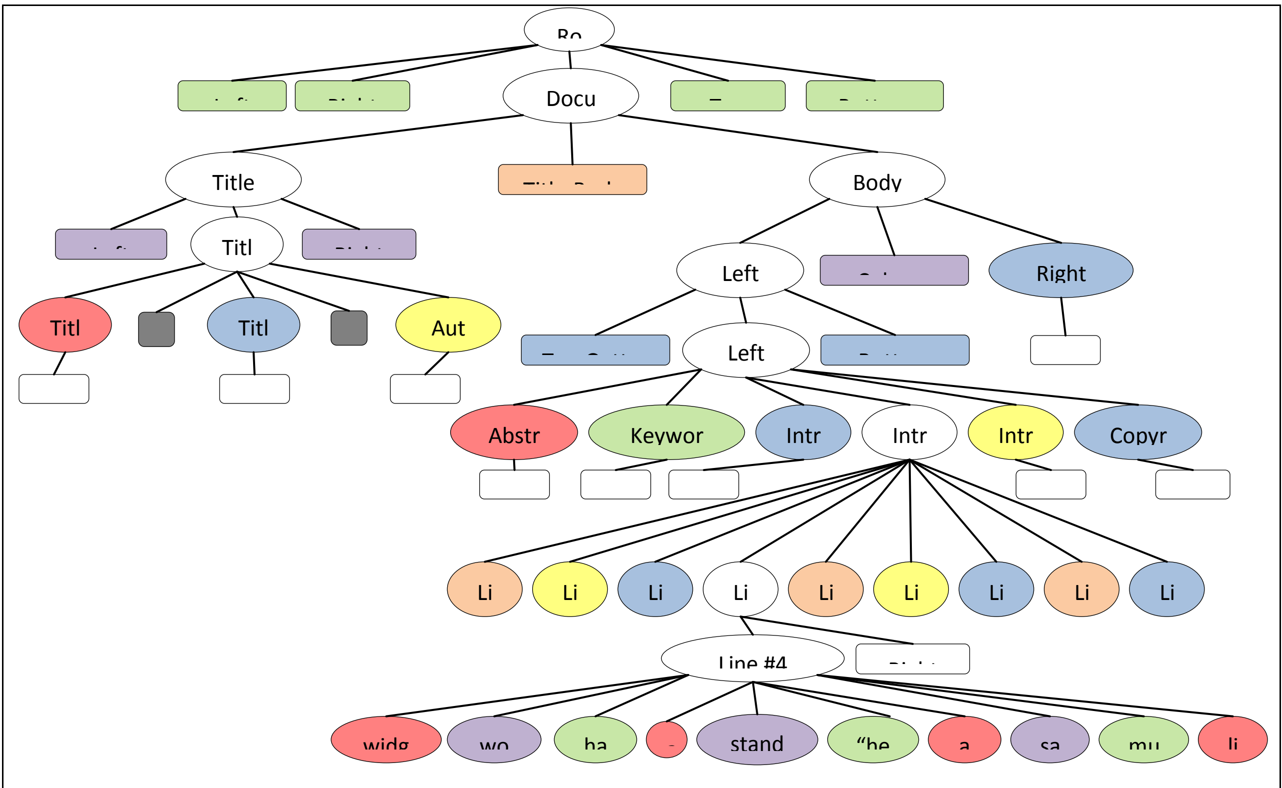


Figure 30 The hierarchal context tree segmented from the document in Figure 31. The nodes are colored equivalently to their corresponding region.

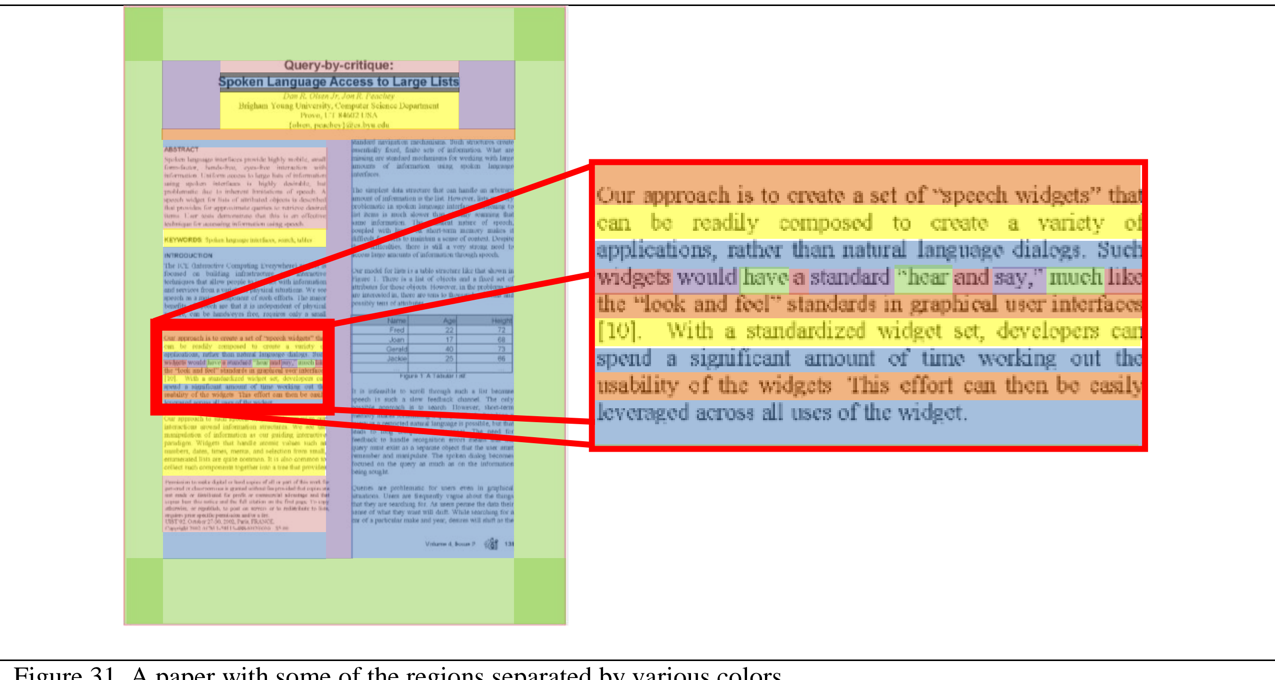


Figure 31 A paper with some of the regions separated by various colors

Getting Contextual Structure

The next issue is how to create an HCT from images. Images, in and of themselves, have no structure whatsoever. However, by simply glancing at a document we can see some clues. A human is able to look at a document and determine where the paragraphs are or to which paragraph a line belongs. This is because documents are prepared for human consumption, and thus contain visual clues to the human perceptual system about its structure. By utilizing perceptual cues, the contextual structure of a document may be determined. Dividing an image of a document up into its parts to determine structure is called segmentation.

The perceptual cue that we utilize in segmentation is whitespace. The human eye naturally divides up regions by the amount of whitespace separating various elements. Take for example Figure 32. While looking at this pattern of circles our brain groups them into various levels.

Rather than seeing sixty-four independent circles, we see the circles in tight groups of four, then more loose groups of four, etc.

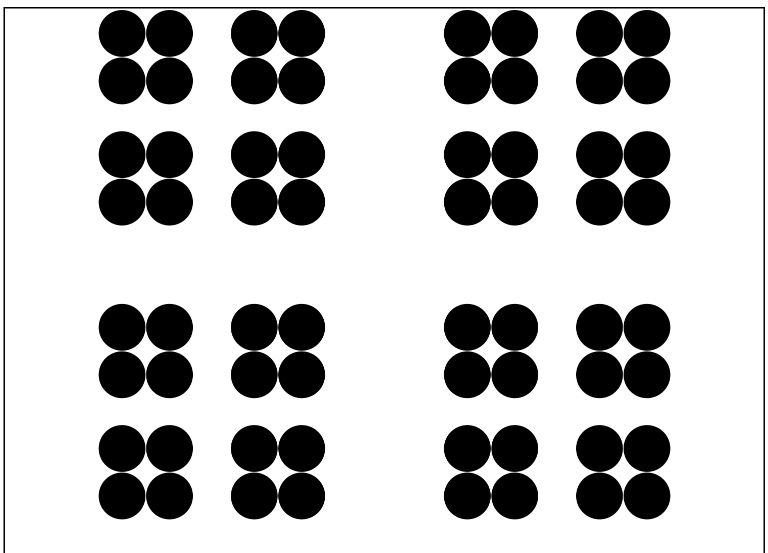


Figure 32 Our brain naturally groups these circles on various levels.

The amount of whitespace between each grouping of circles gives clues to our brain on how to divide the circles. Note that the background need not even be white. Any color will do as long as it stands in strong contrast to the color of the circles and is uniform throughout.

Just as our brain groups these circles based on the amount of whitespace between them, documents use whitespace to denote separations between elements. Take for example

Figure 33. The whitespaces between the lines designate each line as a separate entity. If the whitespace were removed, as in Figure 34, then the paragraph becomes unreadable. PixelJot uses this whitespace to inform its segmentation.

Our approach is to create a set of "speech widgets" that can be readily composed to create a variety of applications, rather than natural language dialogs. Such widgets would have a standard "hear and say," much like the "look and feel" standards in graphical user interfaces [10]. With a standardized widget set, developers can spend a significant amount of time working out the usability of the widgets. This effort can then be easily leveraged across all uses of the widget.	Our approach is to create a set of "speech widgets" that can be readily composed to create a variety of applications, rather than natural language dialogs. Such widgets would have a standard "hear and say," much like the "look and feel" standards in graphical user interfaces [10]. With a standardized widget set, developers can spend a significant amount of time working out the usability of the widgets. This effort can then be easily leveraged across all uses of the widget.
Figure 33 The lines of a paragraph are separated by whitespace.	Figure 34 Without whitespace between lines the text becomes unreadable.

PixelJot segments documents into an HCT by recursively using cuts. A cut is an algorithm that slices along whitespace to divide the document up into smaller pieces. Each smaller part is then cut up again, and those divided regions are cut yet again, so on and so forth. The recursive cutting continues until the document is divided into small atomic pieces with no more whitespace that affords slicing. We introduce two kinds of cutting algorithms: the crop cut and the grid cut. Intuitively the crop cut works by slicing away whitespace *around* objects. In Figure 35 this results in slicing away the margins of the logo and thus isolating the BYU lettering.

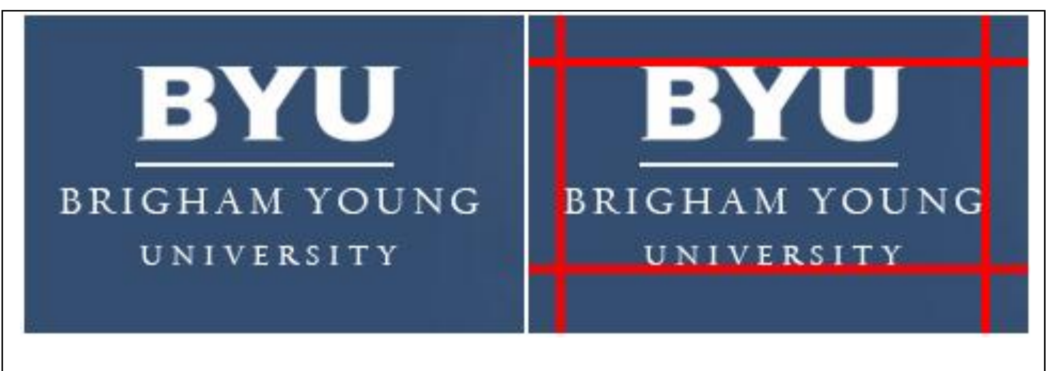


Figure 35 A crop cut before and after.

Where the crop cut is designed for isolating objects, the grid cut is designed for dividing objects up into smaller pieces. As seen in Figure 36, the grid cut slices along the whitespace *between* objects, generating a grid of "cells" or smaller pieces.

4.44	0.26	0.40	12.46	1871.04
4.50	0.26	0.40	12.84	1871.13
4.61	0.26	0.40	13.03	1871.21
4.74	0.26	0.40	12.56	1871.29
4.86	0.26	0.40	12.27	1871.38
4.82	0.26	0.40	12.08	1871.46
4.73	0.26	0.40	12.08	1871.54

Figure 36 A table

4.44	0.26	0.40	12.46	1871.04
4.50	0.26	0.40	12.84	1871.13
4.61	0.26	0.40	13.03	1871.21
4.74	0.26	0.40	12.56	1871.29
4.86	0.26	0.40	12.27	1871.38
4.82	0.26	0.40	12.08	1871.46
4.73	0.26	0.40	12.08	1871.54

Figure 37 The table is divided by a single grid cut using 4 vertical and 6 horizontal slices.

Notice that the HCT in Figure 30 can be created by applying these cuts at each level of segmenting Figure 31. First, the entire document is separated into four margins and the document body using a crop cut. Then the grid cut divides the document into the title section and the text body. If we examine the right branch of this tree we would see that a grid cut divides the document again into the left column and the right column. Cuts can be continually applied to generate the remainder of the HCT. Recursively applying both the crop cut and the grid cut is how PixelJot creates a HCT from an image.

Segmentation needs three separate algorithms:

1. An algorithm for quickly finding whitespace in an image.
2. A crop cut algorithm.
3. A grid cut algorithm.

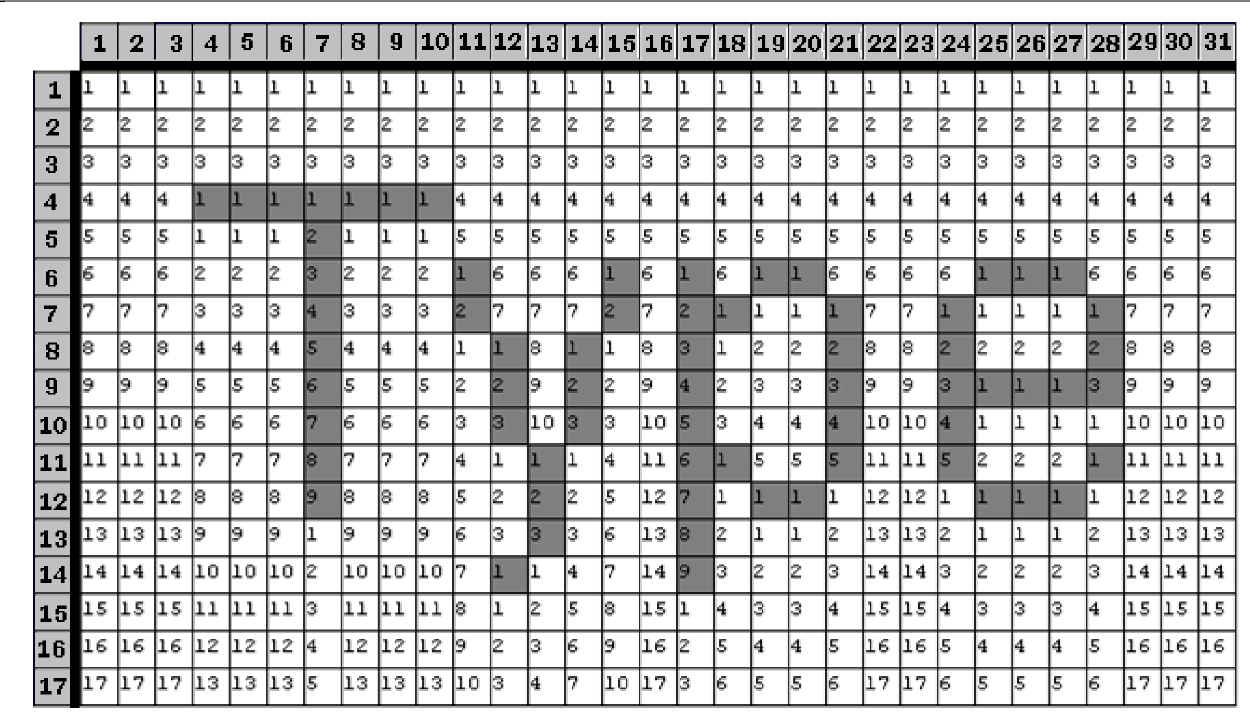
Quickly Finding Whitespace

Quickly finding whitespace in images is a problem that has been previously addressed by ScreenCrayons [12]. ScreenCrayons introduced the continuity map, a data structure that, once built, allows for constant time determination of the size of uniform areas. Continuity maps are similar to integral images, except that values reflect the count of the number of pixels in a given direction (typically to the left or to the top) since a significant difference occurs. PixelJot uses a horizontal continuity map that counts from left to right, and a vertical continuity map that counts from top to bottom. The formal definition of a vertical continuity map is in Figure 38

```
1.  if pixel(x,y) is similar to pixel(x,y-1)
2.    continuity(x,y) = continuity(x,y-1)+1
3.  else
4.    continuity(x,y) = 1
```

Figure 38 Definition of a vertical continuity map

Figure 39 demonstrates an image with its vertical continuity map laid on top.



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	1	1	1	1	1	1	1	4	4	4	4	4	4
5	5	5	5	1	1	1	2	1	1	1	5	5	5	5	5	5
6	6	6	6	2	2	2	3	2	2	2	1	6	6	6	1	6
7	7	7	7	3	3	3	4	3	3	3	2	7	7	7	2	7
8	8	8	8	4	4	4	5	4	4	4	1	1	8	1	1	8
9	9	9	9	5	5	5	6	5	5	5	2	2	9	2	2	9
10	10	10	10	6	6	6	7	6	6	6	3	3	10	3	3	10
11	11	11	11	7	7	7	8	7	7	7	4	1	1	1	4	11
12	12	12	12	8	8	8	9	8	8	8	5	2	2	2	5	12
13	13	13	13	9	9	9	1	9	9	9	6	3	3	3	6	13
14	14	14	14	10	10	10	2	10	10	10	7	1	1	4	7	14
15	15	15	15	11	11	11	3	11	11	11	8	1	2	5	8	15
16	16	16	16	12	12	12	4	12	12	12	9	2	3	6	9	16
17	17	17	17	13	13	13	5	13	13	13	10	3	4	7	10	17

	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5	5	5	5	5	5	5
6	1	6	1	1	6	6	6	6	1	1	1	6	6	6	6
7	2	1	1	1	1	7	7	1	1	1	1	1	7	7	7
8	3	1	2	2	2	8	8	2	2	2	2	2	8	8	8
9	4	2	3	3	3	9	9	3	1	1	1	3	9	9	9
10	5	3	4	4	4	10	10	4	1	1	1	1	10	10	10
11	6	1	5	5	5	11	11	5	2	2	2	1	11	11	11
12	7	1	1	1	1	12	12	1	1	1	1	1	12	12	12
13	8	2	1	1	2	13	13	2	1	1	1	2	13	13	13
14	9	3	2	2	3	14	14	3	2	2	2	3	14	14	14
15	1	4	3	3	4	15	15	4	3	3	3	4	15	15	15
16	2	5	4	4	5	16	16	5	4	4	4	5	16	16	16
17	3	6	5	5	6	17	17	6	5	5	5	6	17	17	17

Figure 39 A vertical continuity map laid over an image of the phrase “Type”.

The numbers on each column count upward until a major change appears, at which point the count starts over. This divides each column into runs, or continuous areas or similar pixels. The run in the first column is full length because it stretches across the entire map. Column four has three separate runs.

The segmentation algorithms we introduce make two different queries against continuity maps. The first query is GetContinuityAt(x,y). This simply returns the value of the continuity map at a given location and takes $O(1)$ time. For example, in Figure 39 GetContinuityAt(7,7) returns four.

The second query is GetRunEnd(x,y). This returns the location in the row or column of the last pixel belonging to the same run as the location passed in. For example in Figure 39 GetRunEnd(7,7) returns 12, the last location in the column that belongs to the same run as location (7,7). This value is found using a binary search algorithm. Binary search requires having an indicator function that determines if the value being searched for lies before or after a given queried location. Fortunately, there is such an indicator. Since continuity maps increments once per pixel and restart count when a break is encountered, then it is always possible to know if a break exists between two locations. If the difference in count between two locations is equivalent to the distance between them, there could have been no break. If it is less then there must have been a break. In the first condition the binary search should look farther, since the break hasn't occurred yet on the lower end. In the second condition the binary search should look nearer.

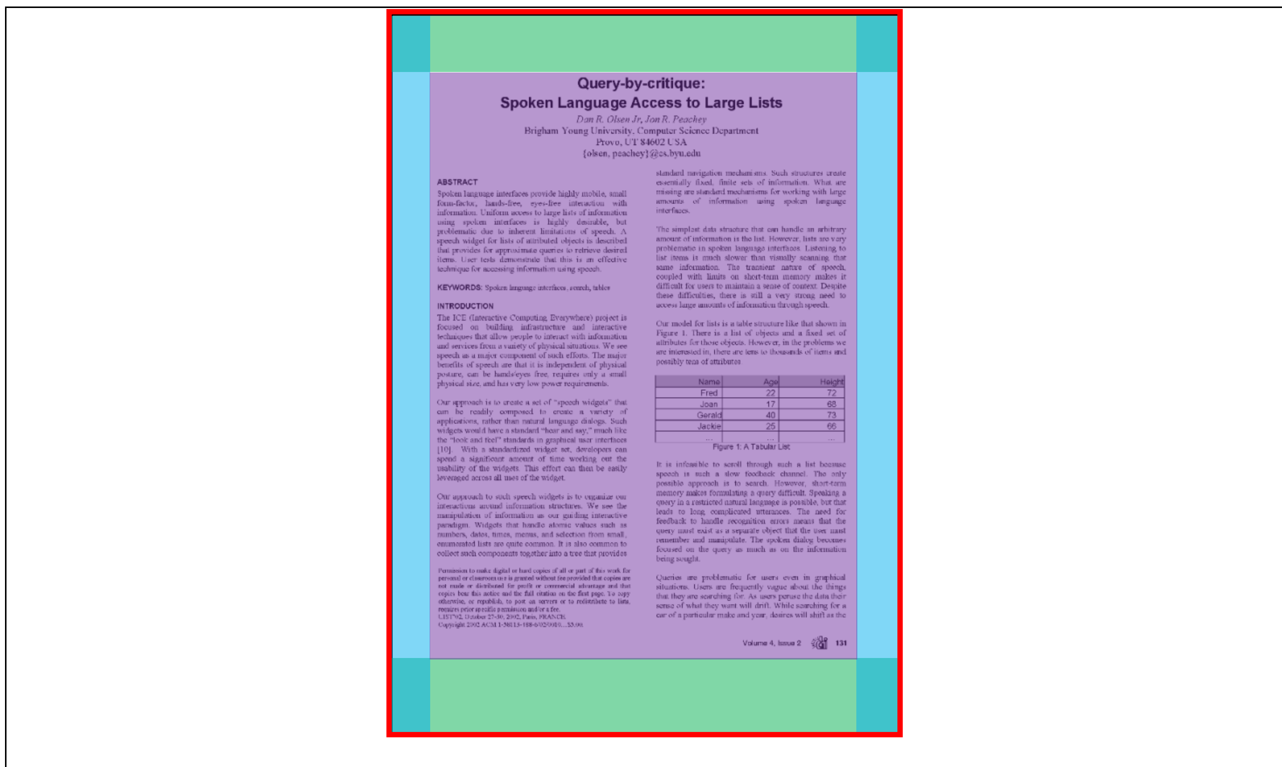
```
1.  int GetRunEnd(ContinuityMap map, int x, int y)  {
2.    int startValue = map.getContinuityAt(x,y) //The value that the query starts at
3.    int lowerBound = y
4.    int upperBound = map.Height
5.
6.    while(lowerBound < upperBound) {
7.      int nextCheck = (lowerBound+upperBound)/2;
8.      int distanceFromStart = nextCheck-y
9.      int value = getContinuityAt(x,nextCheck)
10.     if(value>startValue+distanceFromStart)  assert false //Impossible condition
11.     if(value==startValue+distanceFromStart)  {
12.       if(lowerBound == nextCheck)
13.         return lowerBound
14.       lowerBound = nextCheck
15.     }
16.     else if {
17.       upperBound = nextCheck
18.     }
19.   }
20.   return lowerBound;
21. }
```

Figure 40 Implementation of GetRunEnd(x,y).

These two queries, GetValueAt(x,y) and GetRunEnd(x,y) allow for PixelJot's segmentation algorithm to quickly find the location and extent of whitespace on an image.

Crop Cut

The goal of a crop cut is to slice away whitespace on the perimeter of a region of interest (ROI), and thus isolate an interior object. Figure 41 demonstrates the result of a crop cut against an entire document. The margins of the document are shown as blue and green regions, and the central area, shown in purple, is isolated. The crop cut thus receives as parameters an image, with its vertical and horizontal continuity maps, and a ROI. It returns no more than five regions: one region for each of the top, bottom, left, and right margins (if such margins exist), and another region for the center.



Query-by-critique:
Spoken Language Access to Large Lists

Dan R. Olsen Jr, Jon R. Peachey
Brigham Young University, Computer Science Department
Provo, UT 84602 USA
{olsen, peachey}@cs.byu.edu

ABSTRACT
Spoken language interfaces provide highly mobile, small form-factor, hands-free, eyes-free interaction with information. Uniform access to large lists of information using spoken interfaces is highly desirable, but problematic due to inherent limitations of speech. A speech widget for lists of attributed objects is described that provides for approximate queries to retrieve desired items. User tests demonstrate that this is an effective technique for accessing information using speech.

KEYWORDS: Spoken language interfaces, search, tables

INTRODUCTION
The ICE (Interactive Computing Everywhere) project is focused on building infrastructure and interactive techniques that allow people to interact with information and services from a variety of physical situations. We see speech as a major component of such efforts. The major benefits of speech are that it is independent of physical posture, can be hands/eyes free, requires only a small physical size, and has very low power requirements.

Our approach is to create a set of "speech widgets" that can be readily composed to create a variety of applications, rather than natural language dialogs. Such widgets would have a standard "hear and say," much like the "look and feel" standards in graphical user interfaces [10]. With a standardized widget set, developers can spend a significant amount of time working out the usability of the widgets. This effort can then be easily leveraged across all uses of the widget.

Our approach to such speech widgets is to organize our interactions around information structures. We see the manipulation of information as our guiding interactive paradigm. Widgets that handle atomic values such as numbers, dates, times, menus, and selection from small, enumerated lists are quite common. It is also common to collect such components together into a tree that provides standard navigation mechanisms. Such structures create essentially fixed, finite sets of information. What are missing are standard mechanisms for working with large amounts of information using spoken language interfaces.

The simplest data structure that can handle an arbitrary amount of information is the list. However, lists are very problematic in spoken language interfaces. Listening to list items is much slower than visually scanning that same information. The transient nature of speech, coupled with limits on short-term memory makes it difficult for users to maintain a sense of context. Despite these difficulties, there is still a very strong need to access large amounts of information through speech.

Our model for lists is a table structure like that shown in Figure 1. There is a list of objects and a fixed set of attributes for those objects. However, in the problems we are interested in, there are tens to thousands of items and possibly tens of attributes.

Name	Age	Height
Fred	22	72
Joan	17	68
Gerald	40	73
Jackie	25	66
...	...	...

Figure 1: A Tabular List

It is infeasible to scroll through such a list because speech is such a slow feedback channel. The only possible approach is to search. However, short-term memory makes formulating a query difficult. Speaking a query in a restricted natural language is possible, but that leads to long complicated utterances. The need for feedback to handle recognition errors means that the query must exist as a separate object that the user must remember and manipulate. The spoken dialog becomes focused on the query as much as on the information being sought.

Queries are problematic for users even in graphical situations. Users are frequently vague about the things that they are searching for. As users peruse the data their sense of what they want will drift. While searching for a car of a particular make and year, desires will shift as the

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST'02, October 27-30, 2002, Paris, FRANCE.
Copyright 2002 ACM 1-58113-488-6/02/0010...$5.00.

Volume 4, Issue 2 131

Figure 41 A crop cut has created four overlapping margins (green and blue) and a single central area (in purple).

The crop cut functions by moving vertical lines horizontally inward from the left and right edges of the document, and horizontal lines vertically inward from the top and the bottom edges. Each line is moved inward until it reaches a strip that is not visually uniform. Once the four margin slices are found, deriving the margin and center regions is trivial. The trick then is finding these stopping points. The crop algorithm determines when to stop moving a line inward by checking the continuity image. A vertically uniform line will have a run that extends across the full length of the ROI, but if there is a break in the line its continuity value at the bottom of the run will be less than the height of the ROI. Figure 42 demonstrates how the left slice stopping point is found.

```
1.  int findLeftSliceStoppingPoint(Image img, ROI roi) {
2.    int leftStoppingPoint = roi.Right
3.    for(int x=roi.Left; x<=roi.Right; ++x) {
4.      if(img.VerticalMap.getValueAt(x,roi.Bottom) < roi.Height) {
5.        leftStoppingPoint = x
6.        break
7.      }
8.    }
9.    return leftStoppingPoint
10. }
```

Figure 42 Naïve crop cut algorithm

Finding the stopping point for the top, right, and bottom slices is similar. The top and bottom algorithms utilize a horizontal continuity map rather than the vertical. The heart of finding stopping points comes from the condition of the IF statement on line 4. If the run at the bottom of the ROI is longer than the height of the ROI, then that run *must* have extended the entire length of the ROI. Therefore that line is considered to be entirely uniform. However, if the value at the bottom of the ROI is *less* than the height of the ROI, then a count must have restarted within ROI, and thus there must have been a break.

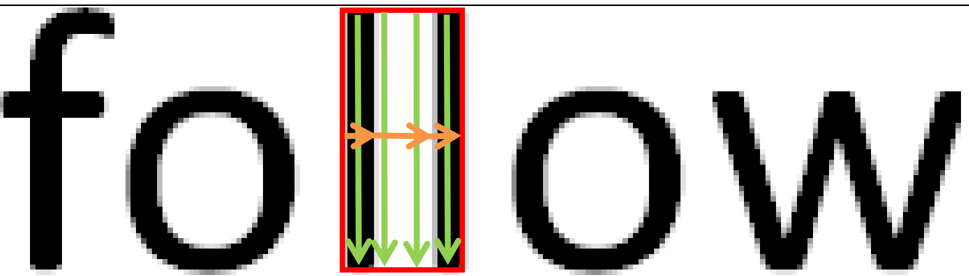


Figure 43 A crop cut attempting to segment two characters. Green arrows are vertical continuities, orange arrows are horizontal continuities.

Unfortunately, this naïve algorithm contains a critical flaw, while it works very well for the larger blocks of a document, it begins to fail on the word and character level. Take for example the double 'L's in the word 'follow' shown in Figure 43. Note that the ROI is set to a tight bounding box around both of the 'L's. The green lines represent vertical runs, all four of which run full length. The orange lines represent three separate runs along one line in the horizontal direction. While only one horizontal line is shown, all of the horizontal runs would be the same in this ROI. The naive algorithm would find each line vertically uniform and therefore when moving inwards from the left would continue all the way to the right. Although each vertical line is uniform, they are not the same *kind* of uniform. A horizontal continuity check shows that there *are* breaks in this ROI.

This issue is solved by the cross checking algorithm shown in Figure 44.

```
1.  int findLeftSliceStoppoingPoint(Image img, ROI roi) {
2.    int leftStoppingPoint = roi.Right
3.    for(int x=roi.Left; x<=roi.Right; ++x) {
4.      if(img.VerticalMap.getValueAt(x,roi.Bottom) < roi.Height) {
5.        leftStoppingPoint = x
6.        break
7.      }
8.    }
9.    leftStoppingPoint = Min(leftStoppingPoint
10.                           ,img.HorizontalMap.getRunEnd(roi.Left,roi.Top))
11.   return leftStoppingPoint
12. }
```

Figure 44 Cross-checking crop cut algorithm

The cross checking algorithm consults with the continuity map of the opposite direction. In the case of Figure 43, the cross check asks the horizontal continuity map where its first run ends. Since the very first run (the first of the orange arrows) is only one pixel long in Figure 43, this value would be one. The minimum of the result from the naïve algorithm and this cross check yields the final stopping point that the crop algorithm uses.

Once these stopping points are created, the crop algorithm uses them to divide the image up into the margins and the center region, then passes the center region on for further segmentation. The margins are known to be entirely uniform areas, so they do not require any recursion. The center area may then be cropped again, or a grid cut may be applied to it.

Grid Cut

Where the crop cut is designed to isolate interesting areas in a document, the grid cut is designed to divide up already cropped regions. The grid cut looks for divisions *inside* the ROI. The grid cut results in cells (interesting areas) divided by gutters (uniform areas). For example Figure 45

and Figure 46 demonstrate a grid cut operating on the body of a document and on a paragraph respectively.

Query-by-critique:
Spoken Language Access to Large Lists

Dan R. Olsen Jr, Jon R. Peachey
Brigham Young University, Computer Science Department
Provo, UT 84602 USA
{olsen, peachey}@cs.byu.edu

ABSTRACT
Spoken language interfaces provide highly mobile, small form-factor, hands-free, eyes-free interaction with information. Uniform access to large lists of information using spoken interfaces is highly desirable, but problematic due to inherent limitations of speech. A speech widget for lists of attributed objects is described that provides for approximate queries to retrieve desired items. User tests demonstrate that this is an effective technique for accessing information using speech.

KEYWORDS: Spoken language interfaces, search, tables

INTRODUCTION
The ICE (Interactive Computing Everywhere) project is focused on building infrastructure and interactive techniques that allow people to interact with information and services from a variety of physical situations. We see speech as a major component of such efforts. The major benefits of speech are that it is independent of physical posture, can be hands/eyes free, requires only a small physical size, and has very low power requirements.

Our approach is to create a set of "speech widgets" that can be readily composed to create a variety of applications, rather than natural language dialogs. Such widgets would have a standard "hear and say," much like the "look and feel" standards in graphical user interfaces [10]. With a standardized widget set, developers can spend a significant amount of time working out the usability of the widgets. This effort can then be easily leveraged across all uses of the widget.

Our approach to such speech widgets is to organize our interactions around information structures. We see the manipulation of information as our guiding interactive paradigm. Widgets that handle atomic values such as numbers, dates, times, menus, and selection from small, enumerated lists are quite common. It is also common to collect such components together into a tree that provides standard navigation mechanisms. Such structures create essentially fixed, finite sets of information. What are missing are standard mechanisms for working with large amounts of information using spoken language interfaces.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
UIST'02, October 27-30, 2002, Paris, FRANCE.
Copyright 2002 ACM 1-58113-488-6/02/0010...$5.00.

The simplest data structure that can handle an arbitrary amount of information is the list. However, lists are very problematic in spoken language interfaces. Listening to list items is much slower than visually scanning that same information. The transient nature of speech, coupled with limits on short-term memory makes it difficult for users to maintain a sense of context. Despite these difficulties, there is still a very strong need to access large amounts of information through speech.

Our model for lists is a table structure like that shown in Figure 1. There is a list of objects and a fixed set of attributes for those objects. However, in the problems we are interested in, there are tens to thousands of items and possibly tens of attributes.

Name	Age	Height
Fred	22	72
Joan	17	68
Gerald	40	73
Jackie	25	66
...	...	...

Figure 1: A Tabular List

It is infeasible to scroll through such a list because speech is such a slow feedback channel. The only possible approach is to search. However, short-term memory makes formulating a query difficult. Speaking a query in a restricted natural language is possible, but that leads to long complicated utterances. The need for feedback to handle recognition errors means that the query must exist as a separate object that the user must remember and manipulate. The spoken dialog becomes focused on the query as much as on the information being sought.

Queries are problematic for users even in graphical situations. Users are frequently vague about the things that they are searching for. As users peruse the data their sense of what they want will drift. While searching for a car of a particular make and year, desires will shift as the

Figure 45 The body of a document is separated into two columns (the blue regions) and the column gutter (the green region) by a grid cut.

The simplest data structure that can handle an arbitrary amount of information is the list. However, lists are very problematic in spoken language interfaces. Listening to list items is much slower than visually scanning that same information. The transient nature of speech, coupled with limits on short-term memory makes it difficult for users to maintain a sense of context. Despite these difficulties, there is still a very strong need to access large amounts of information through speech.

Figure 46 A paragraph is sliced into its various lines and the leading (a printers term for the space between lines) between them by a grid cut.

In Figure 45 a grid cut has divided the two columns of the paper and identified the column gutter between them. In Figure 46 the grid cut has divided a paragraph into nine lines of text by placing a gutter between each line along the text leading. For succinctness, we will explain the grid cut as a vertical list cut. It is left to the reader to generalize the vertical list cut algorithms to the corresponding grid cut algorithms.

In order to setup the grid cut algorithm, we will need to define three helper functions. The heart of the grid cut algorithm is a predicate (shown in Figure 47) that determines if a row of pixels belongs to a gutter or not. This predicate is called isGutter. Similar to the margin of the crop cut, a gutter row of pixels is defined as a row with no breaks in it across the full width of an ROI.

```
1.  boolean isGutter(Image img, ROI roi, int y)  {
2.    return img.HorizontalMap.getValueAt(roi.Right,y) >= roi.Width
3.  }
```

Figure 48 The isGutter predicates determines if a given row belongs to a gutter or not.

Most of the work in the grid cut algorithm is searching across rows of the image looking for the next row that does or does not belong to a gutter row. Figure 49 and Figure 50 define two algorithms that help with this task. Both of them simply scroll through rows, finding the next row that does, or does not, satisfy the isGutter condition.

```
1.  int getNextGutter Row(Image img, ROI roi, int y) {
2.    for(int cursorY=y; cursorY <= roi.bottom AND NOT isGutter(img,roi,y); ++cursorY) { }
3.    return cursorY;
4.  }
```

Figure 49 The getNextGutterRow returns the y index of the next row considered a gutter.

```
1.   int getNextNonGutterRow(Image img, ROI roi, int y)  {
2.     for(int cursorY=y; cursorY <= roi.bottom AND isGutter(img,roi,y); ++ cursorY) { }
3.     return cursorY;
4.   }
```

Figure 50 The getNextNonGutterRow returns the y index of the next row not considered a gutter.

With these three helper functions defined, we're ready to explore the body of the grid cut algorithm. The object of the grid cut is to separate an ROI into several smaller ROIs, called cells, separated by gutters. Once the locations and extent of the gutters are discovered, finding the location and extent of the cells is trivial. It is simply the area in-between gutters. The trick then is to find where the gutters lie. Figure 51 demonstrates this algorithm. In essence this is simply a grouping algorithm. Each row that is considered as part of a gutter is grouped with all of its adjacent gutter rows to make one larger gutter object.

```
1.   List<Gutter> findGutters(Image img, ROI roi)  {
2.     List<Gutter> gutters = new List<Gutter>
3.     int y = roi.Top

4.   //Scroll through until we hit something interesting, start counting from
     //there
5.   y = getNextGutterRow (img,roi,y)
6.   for(; y<=roi.bottom; ++y) {
7.     if(isGutter(img,roi,y)  { //Starts a gutter
8.        int startY = y++   //Store the start of the gutter
9.        //Start looking for the end of the gutter at the next pixel
10.        y = getNextNonGutterRow(img,roi,y)
11.       gutter.add(new Gutter(startY,y)) //This gutter extends from startY to y
12.    }
13.    return gutters
14.  }
```

Figure 51 findGutters determines where the gutters are in a given ROI.

Over segmentation

While simple, this algorithm has a serious flaw. Like the crop cut's original algorithm, this works well on the macro-scale of the document, column, paragraph, or line level, but begins to fail on the word and character level. Particularly, this grid cut algorithm segments characters too deeply. Figure 52 demonstrates a short sequence of characters. Figure 53 shows where we would like the grid cut to place a slice. Slicing between the characters separates them into two distinct elements. Unfortunately, the naïve grid algorithm stated thus far yields the result shown in Figure 54.

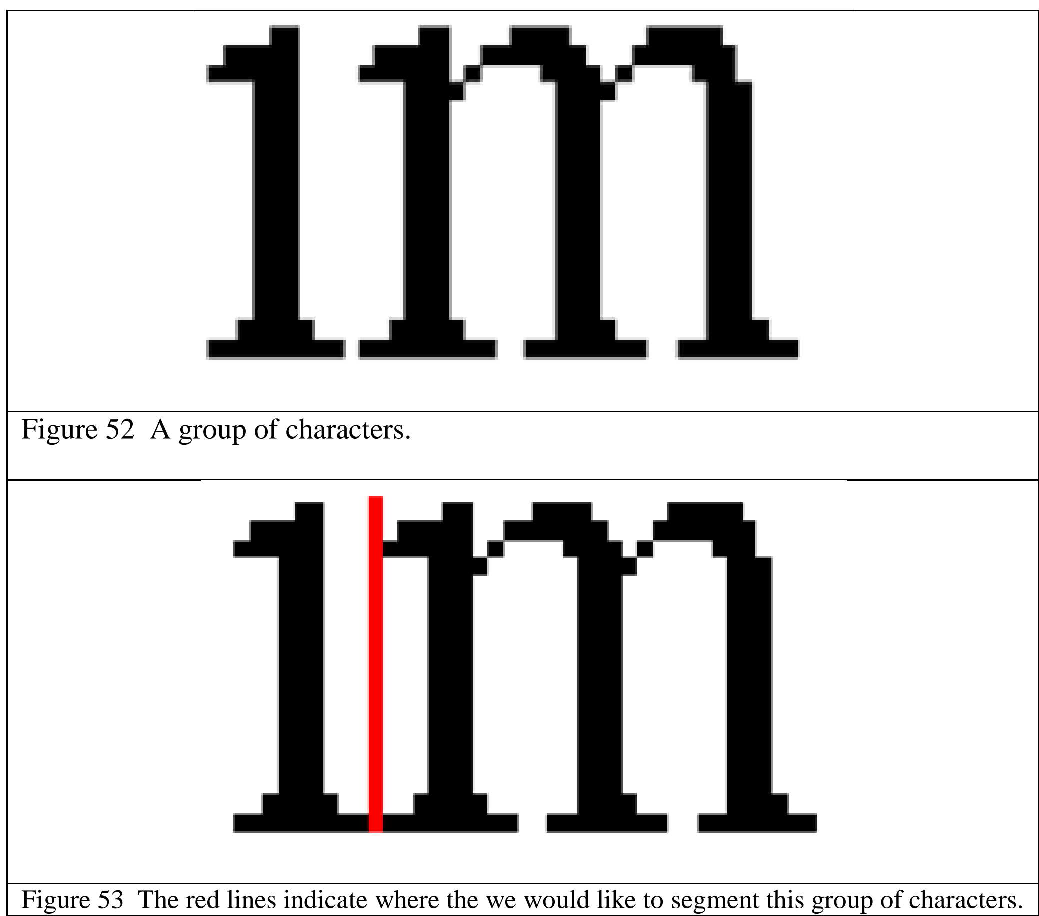


Figure 52 A group of characters.

Figure 53 The red lines indicate where the we would like to segment this group of characters.

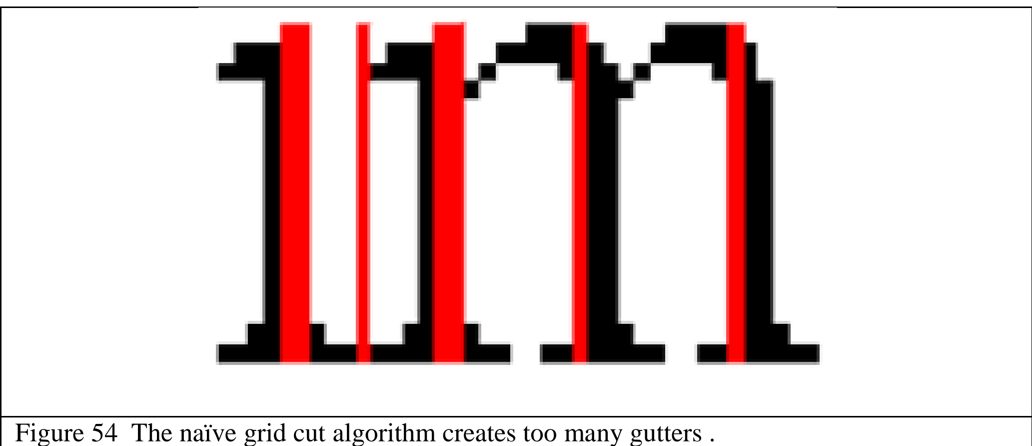


Figure 54 The naïve grid cut algorithm creates too many gutters .

In addition to one slice separating the ‘i' and the ‘m’, we get other slices within each of these characters. This is because the black vertical lines composing the ‘i' and the ‘m’ are uniform and full length and thus are considered valid gutter columns.

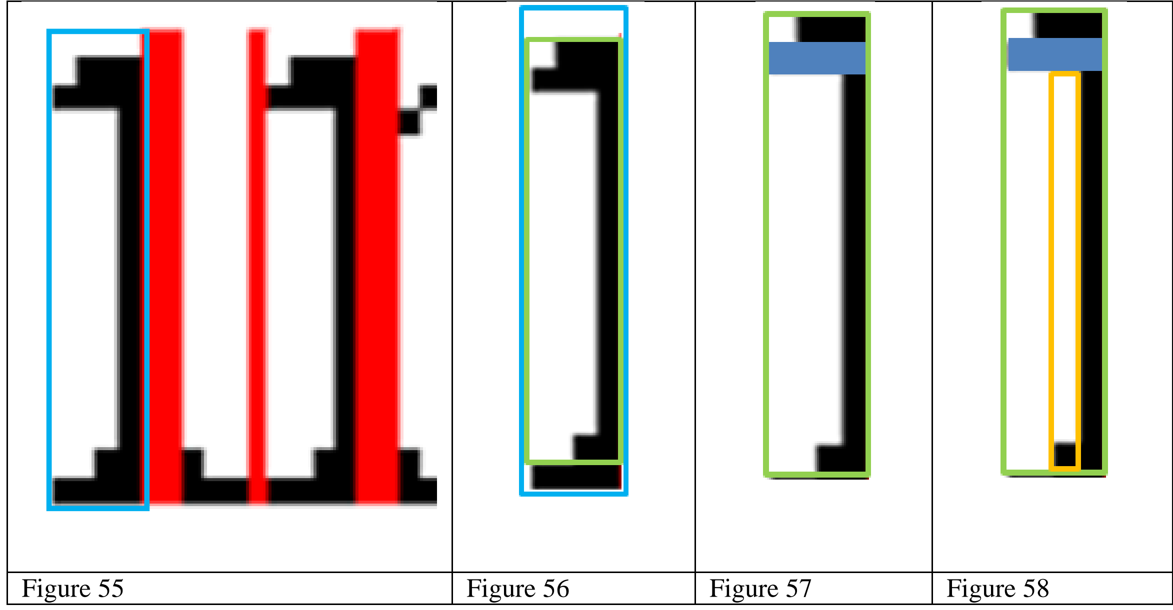


Figure 55 Figure 56 Figure 57 Figure 58

The largest drawback to this over segmentation is that it may, in turn, cause even more bad segmentation. Let’s take for example the left most region created in Figure 54, which is shown by the blue rectangle in Figure 55. This new region would be further segmented by cropping out

the white strip on the top, and the black strip on the bottom, yielding the green region in Figure 56. This crop region would be divided again with a grid cut cutting along the four black pixels near the top of the green rectangle. This gutter is shown by the solid blue box in Figure 57. If we were to examine the lower cell of this grid cut, we'd see that the two white columns on the left, and the single right column on the right would be removed by a crop cut, yielding the orange region in Figure 58. From here the ROI would be reduced to the single black pixel at the bottom of the orange region by another crop cut. As we can see, all of these cuts below the character level are entirely inane. Often these trivial slices continue until the document image is sliced up into single pixels, resulting in a massive HCT. This extremely large tree is burdensome to hold in memory and repaint efficiently. Clearly some kind of stopping condition is needed on grid slices.

We found that the most effective method of stopping this over segmentation is to enforce that grid nodes slice *only* along background colored pixels. Figure 59 demonstrates the necessary modification to the isGutter predicate that requires gutters to be of a background color.

```
1.  boolean isGutter(Image img, ROI roi, int y)  {
2.    return img.HorizontalMap.getValueAt(roi.Right,y) >= roi.Width
3.             AND isBackgroundColor(img.getPixel(roi.Right,y))
4.  }
```

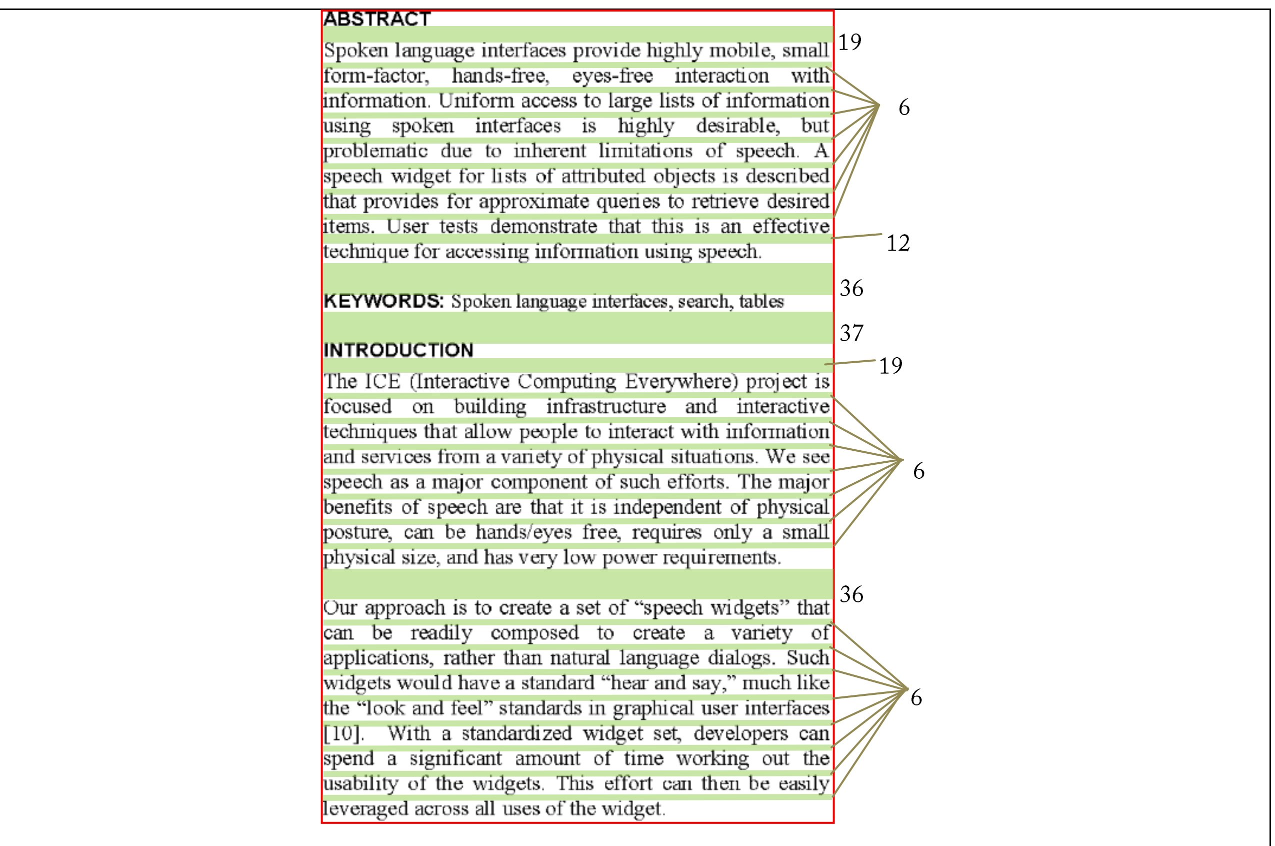
Figure 59 The background-aware isGutter predicate.

In PixelJot the background is discovered globally for each page by finding the mode color. Other methods could be employed to find the background color locally for a given ROI, which would allow segmentation to work on pages with several different background colors. However,

since this is atypical of most documents, one global background color per page has sufficed. Slicing only along background colored pixels yields results like Figure 53 with no further segmentation possible.

Hierarchically inferred cutting

Another important modification to the naïve grid cut algorithm is *not* immediately taking all available cuts. The object of segmentation is to build a hierarchal context tree, and there may be times that deferring a grid cut until later actually constructs a better hierarchy. Take for example Figure 60. In this figure there are forty-three horizontal slices possible, the gutters of each is shown in green. We would expect a segmentation of this column to appear as Figure 61.



ABSTRACT

Spoken language interfaces provide highly mobile, small form-factor, hands-free, eyes-free interaction with information. Uniform access to large lists of information using spoken interfaces is highly desirable, but problematic due to inherent limitations of speech. A speech widget for lists of attributed objects is described that provides for approximate queries to retrieve desired items. User tests demonstrate that this is an effective technique for accessing information using speech.

KEYWORDS: Spoken language interfaces, search, tables

INTRODUCTION

The ICE (Interactive Computing Everywhere) project is focused on building infrastructure and interactive techniques that allow people to interact with information and services from a variety of physical situations. We see speech as a major component of such efforts. The major benefits of speech are that it is independent of physical posture, can be hands/eyes free, requires only a small physical size, and has very low power requirements.

Our approach is to create a set of "speech widgets" that can be readily composed to create a variety of applications, rather than natural language dialogs. Such widgets would have a standard "hear and say," much like the "look and feel" standards in graphical user interfaces [10]. With a standardized widget set, developers can spend a significant amount of time working out the usability of the widgets. This effort can then be easily leveraged across all uses of the widget.

Figure 60 A column of a research paper with possible gutters shown in green. The height of each gutter is indicated to its right.

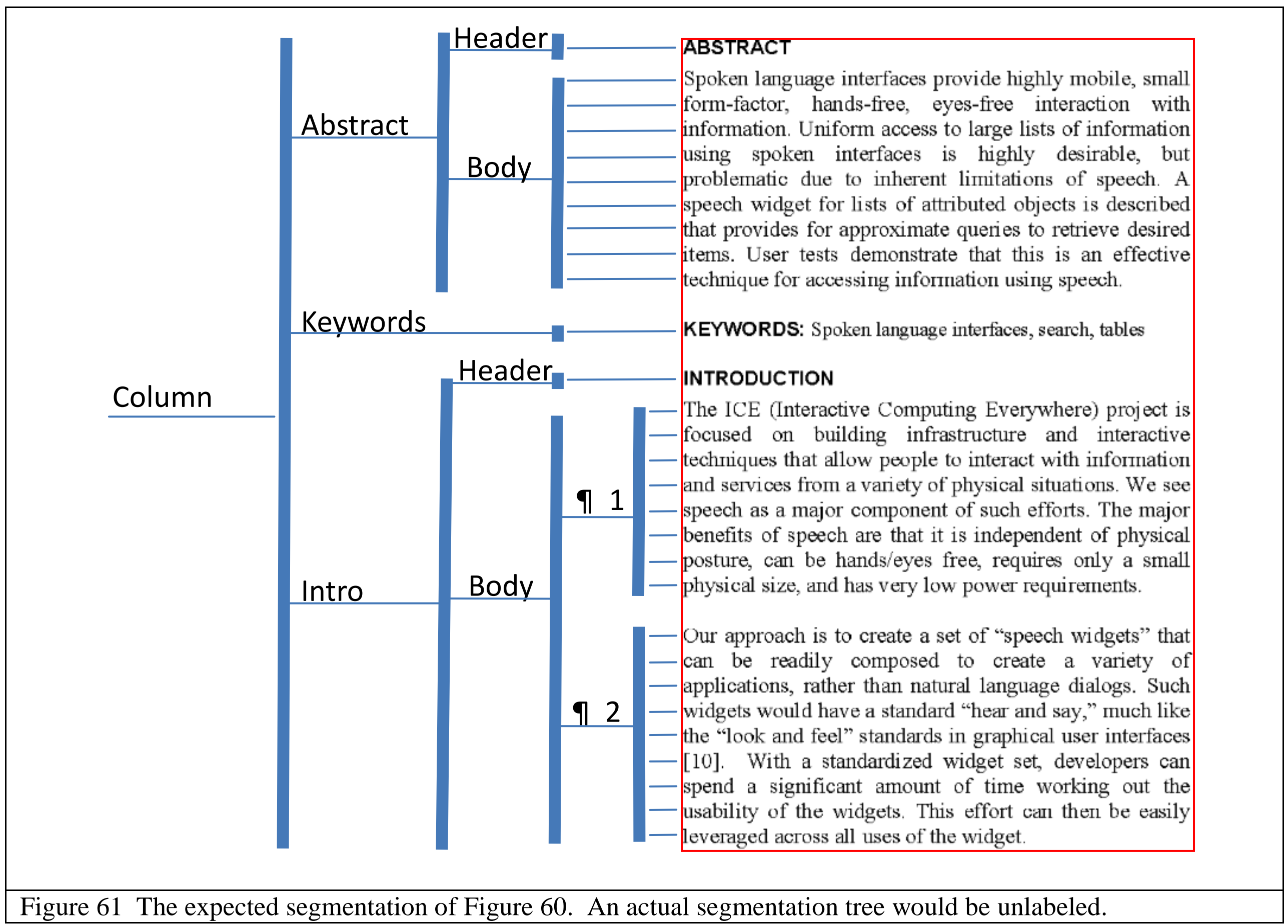
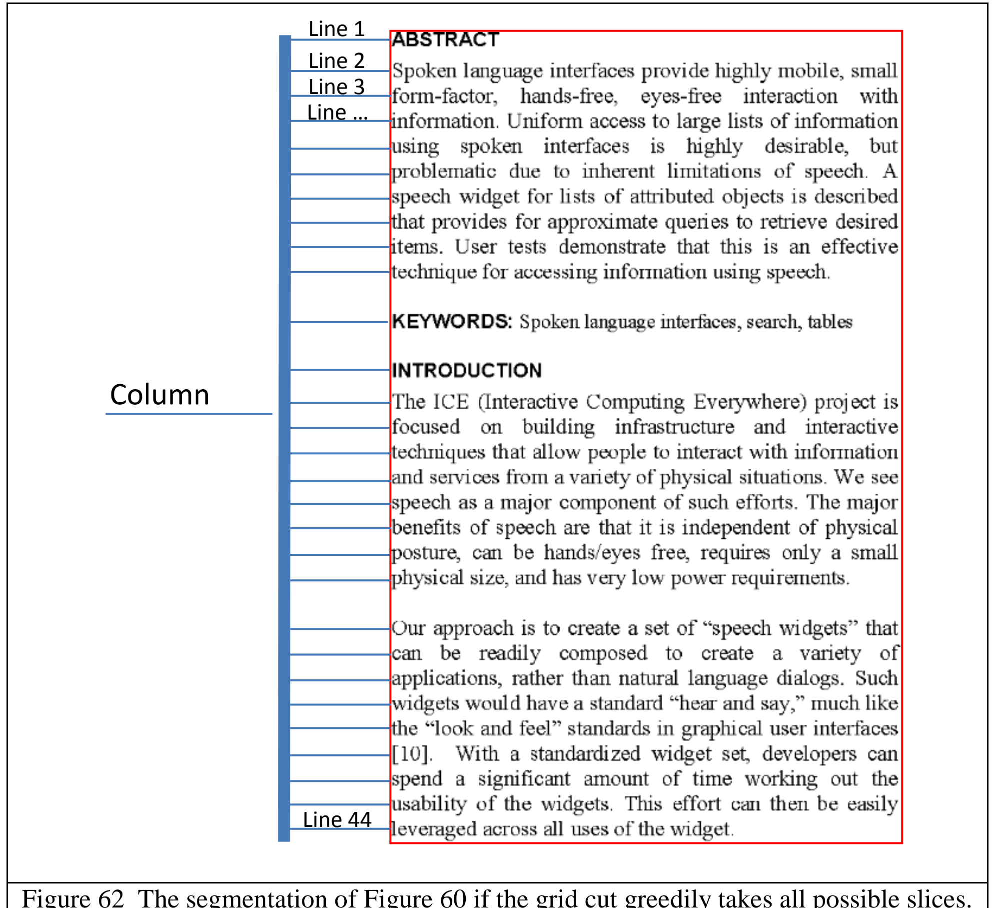


Figure 61 The expected segmentation of Figure 60. An actual segmentation tree would be unlabeled.

If the grid cut were to greedily take every possible slice, the result would appear like the tree in Figure 62. This is because *all* of the green gutters of Figure 60 would be used simultaneously, and thus lines of different paragraphs or even sections would find themselves peers of each other under the same grid cut in the HCT.



ABSTRACT

Spoken language interfaces provide highly mobile, small form-factor, hands-free, eyes-free interaction with information. Uniform access to large lists of information using spoken interfaces is highly desirable, but problematic due to inherent limitations of speech. A speech widget for lists of attributed objects is described that provides for approximate queries to retrieve desired items. User tests demonstrate that this is an effective technique for accessing information using speech.

KEYWORDS: Spoken language interfaces, search, tables

INTRODUCTION

The ICE (Interactive Computing Everywhere) project is focused on building infrastructure and interactive techniques that allow people to interact with information and services from a variety of physical situations. We see speech as a major component of such efforts. The major benefits of speech are that it is independent of physical posture, can be hands/eyes free, requires only a small physical size, and has very low power requirements.

Our approach is to create a set of "speech widgets" that can be readily composed to create a variety of applications, rather than natural language dialogs. Such widgets would have a standard "hear and say," much like the "look and feel" standards in graphical user interfaces [10]. With a standardized widget set, developers can spend a significant amount of time working out the usability of the widgets. This effort can then be easily leveraged across all uses of the widget.

Figure 62 The segmentation of Figure 60 if the grid cut greedily takes all possible slices.

Rather than taking all available slices, the grid cut needs to determine that some slices should occur before other. The less important slices would be deferred to occur further down the HCT. How then can the grid cut infer the relative importance, and thus the correct order, of slices?

Prioritizing slices using maximum gutter

Looking again at the pattern of circles that inspired perceptually inferred segmentation shows how to prioritize slicing. The pattern is shown again in Figure 63 for convenience. Earlier we discussed that our brain is able to segment this pattern into various groupings of circles, and that

the reason it is able to do so is because of the whitespace between the circles. What then allows us to designate the importance of one grouping from the other? Clearly the *amount* of whitespace matters. The more the whitespace between groups, the higher we put them in our mental hierarchy.

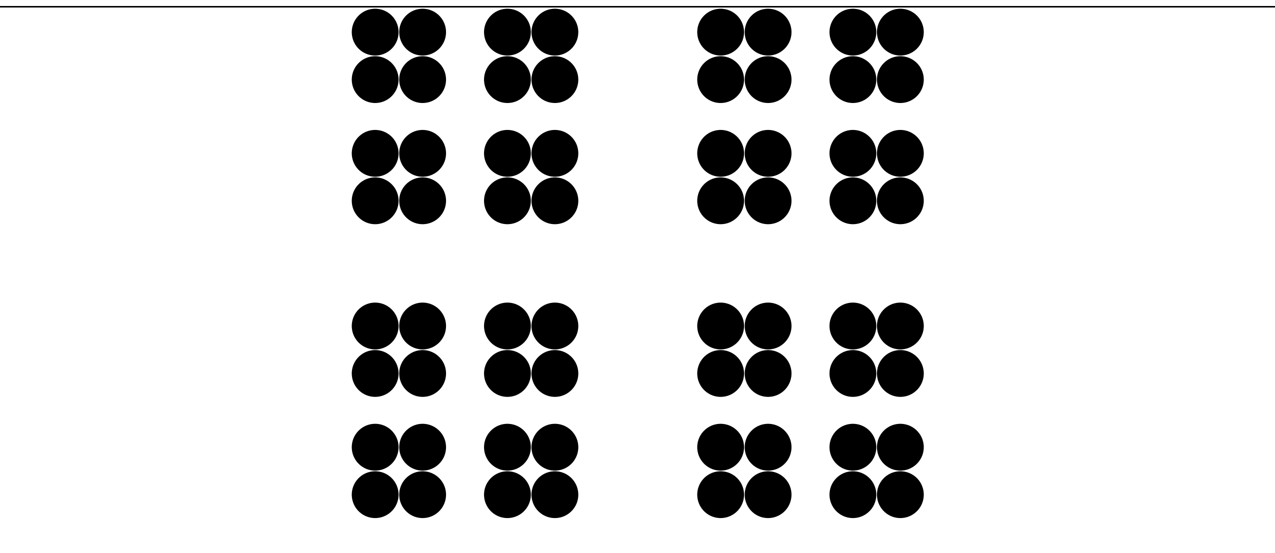


Figure 63 Our brain naturally groups these circles on various levels.

Remember that documents are designed for human consumption, and thus lines, paragraphs, and sections are all given gaps of varying sizes to designate importance separation. In Figure 60 notice that the largest gaps represent breaks between sections, smaller gaps breaks between paragraphs, and the smallest breaks between lines. Thus by prioritizing slices based on the amount of whitespace between elements, a better hierarchy can be inferred.

```
1.    List<Gutter> findGutters(Image img, ROI roi)  {
2.      List<Gutter> gutters = new List<Gutter>
3.      int y = roi.Top

4.    //Scroll through until we hit something interesting, start counting from
      //there
5.    y = getNextGutterRow (img,roi,y)
6.    for(;y<=roi.bottom; ++y) {
7.      if(isGutter(img,roi,y)  {//Starts a gutter
8.        int startY = y++  //Store the start of the gutter
9.        //Start looking for the end of the gutter at the next pixel
10.        y = getNextNonGutterRow(img,roi,y)
11.       gutter.add(new Gutter(startY,y)) //This gutter extends from startY to y
12.     }
13.     return GetPriorityGutters (gutters)
14. }
```

Figure 64 Modified findGutters algorithm filtering of found gutters to the priority gutters.

The algorithm in Figure 64 modifies the original findGutters routine by adding a filter on the results (see line 13). Rather than yielding all possible gutters, only those gutters considered to be top priority are returned. Figure 65 contains the pseudo-code for GetPriorityGutters.

```
1.    List<Gutter> GetPriorityGutters(List<Gutter> input) {
2.      List<Gutter> output = new List<Gutter>()
3.      int maximumGutterSize = -1

4.      for each(Gutter g in input)  {
5.        int gutterSize = g.EndY – g.StartY
6.        maximumGutterSize – Max(maximumGutterSize,gutterSize)
7.      }

8.      for each(Gutter g in input)  {
9.        int gutterSize = g.EndY – g.StartY
10.       if(gutterSize== MaximumGutterSize)
11.         output.add(g)
12.     }
13.     return output;
14. }
```

Figure 65 GetPriorityGutters filters the input to only gutters the same height as the largest.

This implementation of GetPriorityGutters simply finds the size of the maximum gutter, and returns all gutters of equivalent size. This change in the grid cut algorithm yields the HCT shown in Figure 66.

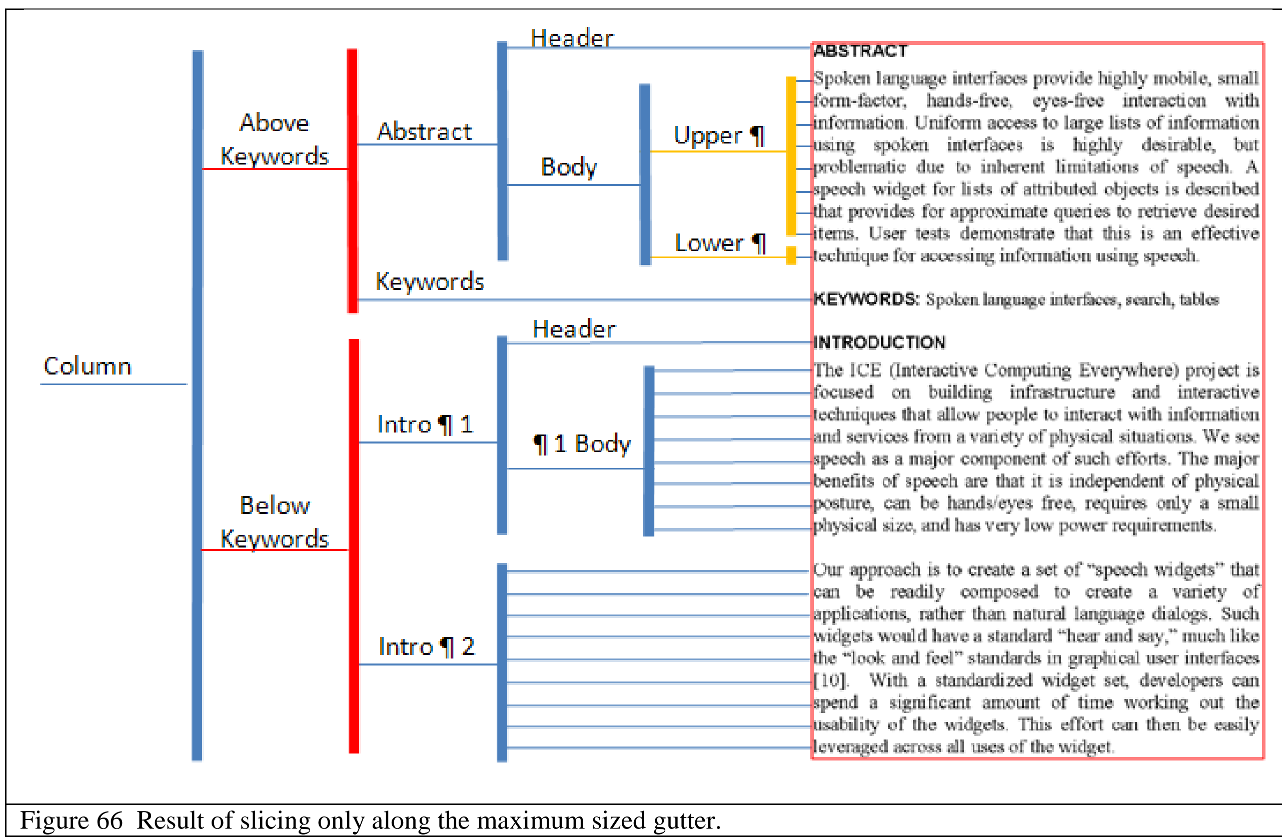
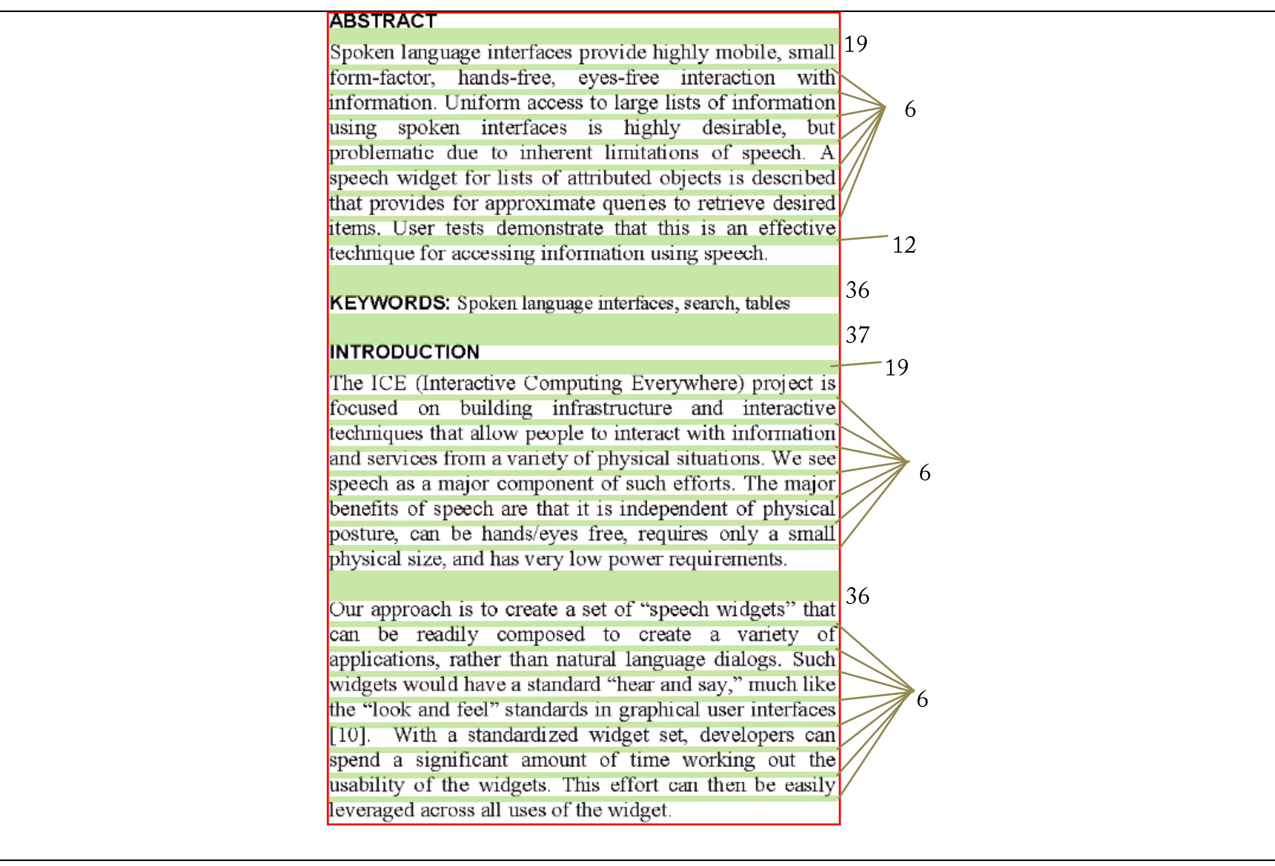


Figure 66 Result of slicing only along the maximum sized gutter.

There are two anomalies on this HCT. The first is designated by the red nodes. Like the HCT in Figure 61, we would expect the column to be broken into the separate sections. Instead the first division breaks the column into a before keywords, and after keywords grouping, and only afterwards reduces into sections. If we examine the original column with its breaks (Figure 67), we can see why.



ABSTRACT

Spoken language interfaces provide highly mobile, small form-factor, hands-free, eyes-free interaction with information. Uniform access to large lists of information using spoken interfaces is highly desirable, but problematic due to inherent limitations of speech. A speech widget for lists of attributed objects is described that provides for approximate queries to retrieve desired items. User tests demonstrate that this is an effective technique for accessing information using speech.

KEYWORDS: Spoken language interfaces, search, tables

INTRODUCTION

The ICE (Interactive Computing Everywhere) project is focused on building infrastructure and interactive techniques that allow people to interact with information and services from a variety of physical situations. We see speech as a major component of such efforts. The major benefits of speech are that it is independent of physical posture, can be hands/eyes free, requires only a small physical size, and has very low power requirements.

Our approach is to create a set of "speech widgets" that can readily composed to create a variety of applications, rather than natural language dialogs. Such widgets would have a standard "hear and say," much like the "look and feel" standards in graphical user interfaces [10]. With a standardized widget set, developers can spend a significant amount of time working out the usability of the widgets. This effort can then be easily leveraged across all uses of the widget.

Figure 67 Gutters in a column of a research papers. Gutters are shown in green with their heights indicated to the right.

Notice that the largest gutter is between the keywords and the introduction sections at thirty-seven pixels tall. The other two breaks we'd expect to be taken are only thirty-six pixels tall. We do not notice a difference so slight as one pixel, but the revised grid cut algorithm does. Because of this it introduces an odd intermediate layer of structure. Taking the absolute maximal gutters is brittle in the face of slight differences in whitespace sizes.

The second anomaly designated by the orange nodes, will be addressed later on.

Prioritizing slices using near maximal gutters

To fix this anomaly, we introduce a threshold value called the "equivalent importance ratio". Any gutter whose size is within this ratio to the size of the maximal gutter is also considered equivalent to the maximal gutter and thus included as a priority gutter. This ratio reflects the degree of imprecision in our eyes and is normally set to about 0.75.

```
1.  List<Gutter> GetPriorityGutters(List<Gutter> input) {
2.    List<Gutter> output = new List<Gutter>()
3.    int maximumGutterSize = -1

4.    for each(Gutter g in input) {
5.      int gutterSize = g.EndY – g.StartY
6.      maximumGutterSize – Max(maximumGutterSize,gutterSize)
7.    }

8.    for each(Gutter g in input) {
9.      int gutterSize = g.EndY – g.StartY
10.     if(gutterSize>= MaximumGutterSize*EquivalentImportanceRatio)
11.       output.add(g)
12.   }
13.   return output;
14. }
```

Figure 68 The modified GetPriorityGutters filters the input to only gutters the *near* same height as the largest.

Since $0.75 \times 37 = 27.75$, and 36 is greater than 27.5, then the two cuts that the original GetPriorityGutters algorithm deferred will indeed be taken in this modified version. The updated grid cut algorithm yields the HCT shown in Figure 69.

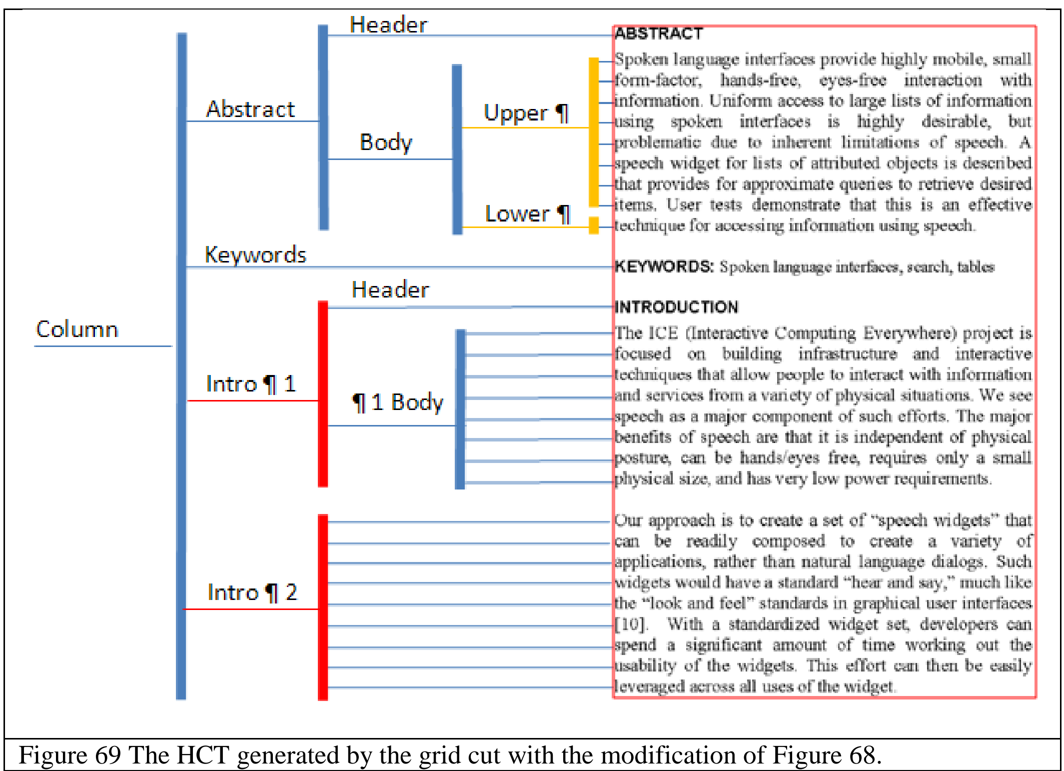


Figure 69 The HCT generated by the grid cut with the modification of Figure 68.

This does solve the problem of the previous grid cut algorithm, but there are two anomalies. The first, shown in red, is that rather than grouping the entire introduction into one section, and later dividing the intro into its constituent paragraphs as does the ideal HCT of Figure 61, the modified algorithm immediately divides the intro into paragraphs with no intermediate section step. Unfortunately, if we examine the column that we're trying to segment (Figure 67), we'll see that there is no way to infer this break by looking at spacing alone. This particular paper is formatted such that the breaks between sections and the breaks between paragraphs are the same size.

The second anomaly (also present in Figure 66) is shown in orange. Rather than breaking up the body of the abstract directly into its constituent lines, the grid cut has introduced an intermediate

layer of structure, breaking the paragraph into an upper and lower section. Again, examining Figure 67 will show us why. Notice that in-between most lines of the paragraph the gutters are six pixels high. However, between the last and the penultimate line the gutter is twelve pixels high. A closer examination shows that this is because the penultimate line of text has no descenders. Descenders are any characters that extend below the base line of text such as the letters ‘g’, ‘q’, ‘p’ or ‘y’. Because these lines *don’t* extend below the base line, but most applications still give room on each line for descent, there is more space between a descenderless line and the next than between a line with descenders and its next line. Figure 70 illustrates this issue. The descenderless line has as gap with twice the height as the line with descenders. Modifying the equivalent importance ratio to be small enough to keep these lines from being grouped separately puts it too low for quality cutting elsewhere.

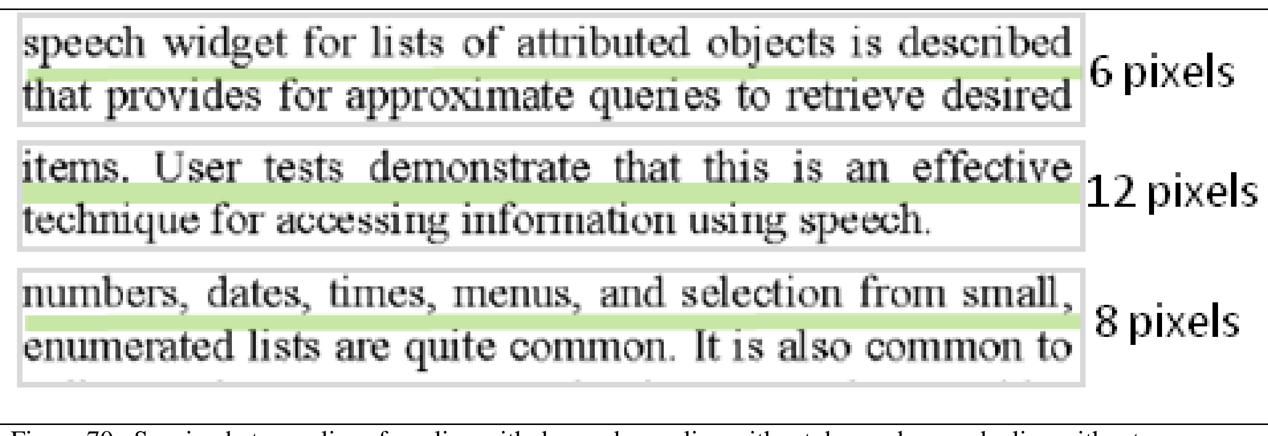


Figure 70 Spacing between lines for a line with descenders, a line without descenders, and a line without descenders but with commas.

However, this additional layer of structure may not be an entirely grievous offense for two reasons. First of all, it doesn’t occur often. By informal survey, double column research papers have a descenderless line once every four paragraphs. Single column papers nearly never have a

descenderless line. The author has never yet seen a line of text in any paper which has a line with no ascenders, though this would cause a similar issue.

The second reason to disregard this error is that the tools we develop are rarely hindered by this slight degree of extra hierarchy. Take for example selection. In selection, the HCT is used for snapping the user's selection rectangle to a nearby element in the tree. Selecting a paragraph is a reasonable task. Selecting a line is also a reasonable task. However, selecting the lower portion of a paragraph is not likely to ever happen. The arbitrary additional layer introduced by the decenderless line will probably never be discovered because no user is likely to try and select it.

Segmentation of document images leverages perceptual structure to infer an appropriate hierarchy. Segmentation composes a hierarchical context tree by recursively applying crop cuts and grid cuts to regions of a document. Once divided into an HCT, PixelJot can use this hierarchy to reflow pages, offer link context expansion, and improve selection accuracy.

LAYOUT

With the HCT available, several key operations become possible, including layout. When annotating a document, a user will often want to insert comments or insert a visual link as a cross reference. However, the document may not have room in its whitespace to accommodate these insertions. Suppose the comment block shown in Figure 71 needs to be inserted at the space between the two paragraphs. Clearly it will not fit. Ideally, rather than blocking the areas of text, the page should reflow, making room for the comment as in Figure 72.

Figure 71 The comment in blue needs to be inserted at the arrow point.

Laser Pointer Interaction

Dan R. Olsen Jr. and Travis Nielsen
Computer Science Department, Brigham Young University, Provo, UT
{olsen, nielsent }@cs.byu.edu

ABSTRACT
Group meetings and other non-desk situations require that people be able to interact at a distance from a display surface. This paper describes a technique using a laser pointer and a camera to accomplish just such interactions. Calibration techniques are given to synchronize the display and camera coordinates. A series of interactive techniques are described for navigation and entry of numbers, times, dates, text, enumerations and lists of items. The issues of hand jitter, detection error, slow sampling and latency are discussed in each of the interactive techniques.

Keywords
Laser pointer interaction, group interaction, camera-based interaction.

INTRODUCTION
A very interesting setting for interactive computing is in a meeting where the display is projected on the wall. Projection of the large image allows all participants sitting in their chairs to see the information under discussion. This provides a shared environment that can ground the discussion and provides an equal discussion point for everyone. However, if the information is interactive, only

equipment consists of a computer attached to a projector and a camera to detect the laser pointer position. We used a standard 1024 x 768 projector connected to a laptop PC. For the camera we used a $500 WebCam that can deliver up to 7 frames per second over TCP/IP. This camera connection is very slow, but adequate for our initial tests.

This is an idea that I have
Is there an easier way of doing this?
It does seem like thay did this pretty cheaply
Haven't WebCam's dropped in price since this paper
I'm glad that this comment block will make it into this area.

In addition to meeting situations, this technique is useful wherever the user is in a situation for which a large projected display is possible, but a local personal display would be awkward. Examples include a repair shop with service information displayed on the wall, a laboratory where instrument controls are displayed on the wall, or as an alternative to the traditional television IR remote. In situations where the hands are occupied, the laser could be mounted on the back of a half-finger glove with the actuator switch on the side of the glove. This would require use of the hand to point, but would eliminate searching for and grabbing the pointer.

Figure 72 The paragraphs flow to make room for the inserted comment.

Knowing where to move elements of a document can be a difficult problem. Take for example the gray block in Figure 73. If we were to insert it where the arrow indicates, where would the paragraphs, title, and columns flow? We, as humans, understand that the result should look something like Figure 74. The column on the left flows vertically, causing paragraphs below the insertion point to shift downwards. The right column and title should be unaffected.

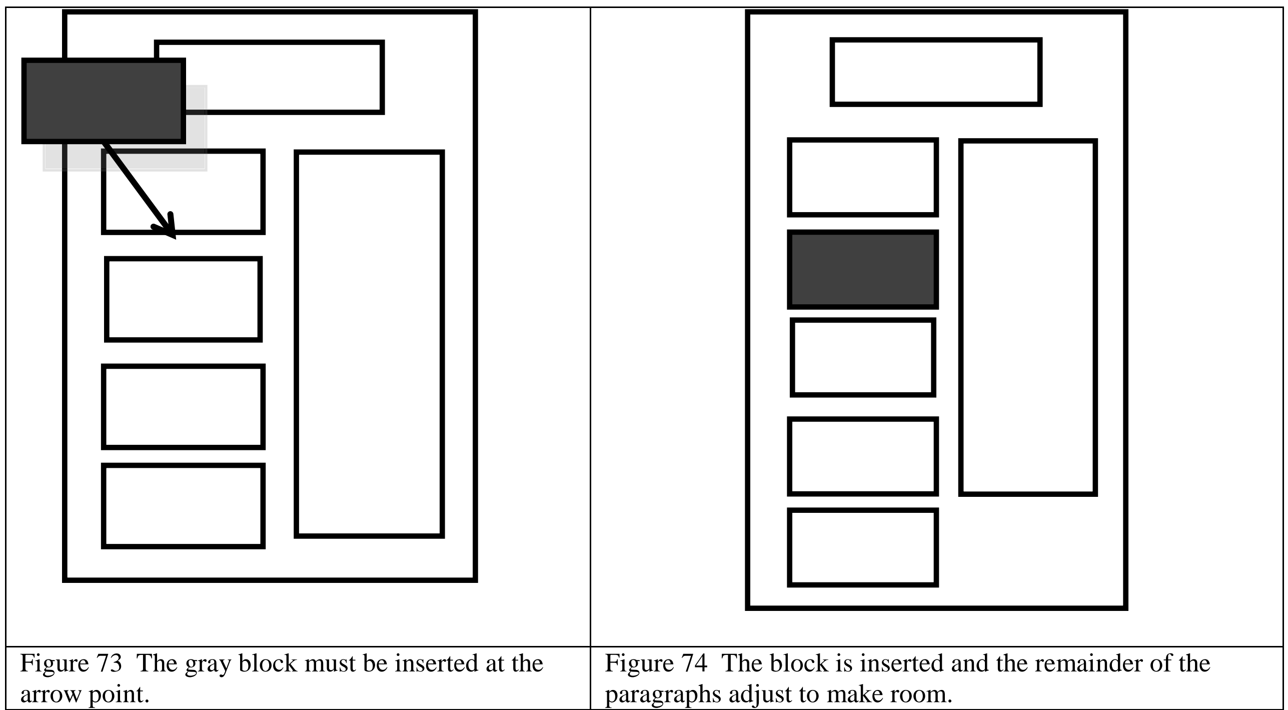


Figure 73 The gray block must be inserted at the arrow point.

Figure 74 The block is inserted and the remainder of the paragraphs adjust to make room.

However, if instead the insertion were to occur only a little farther to the right, as in Figure 73, we would expect the totally different result of Figure 74.

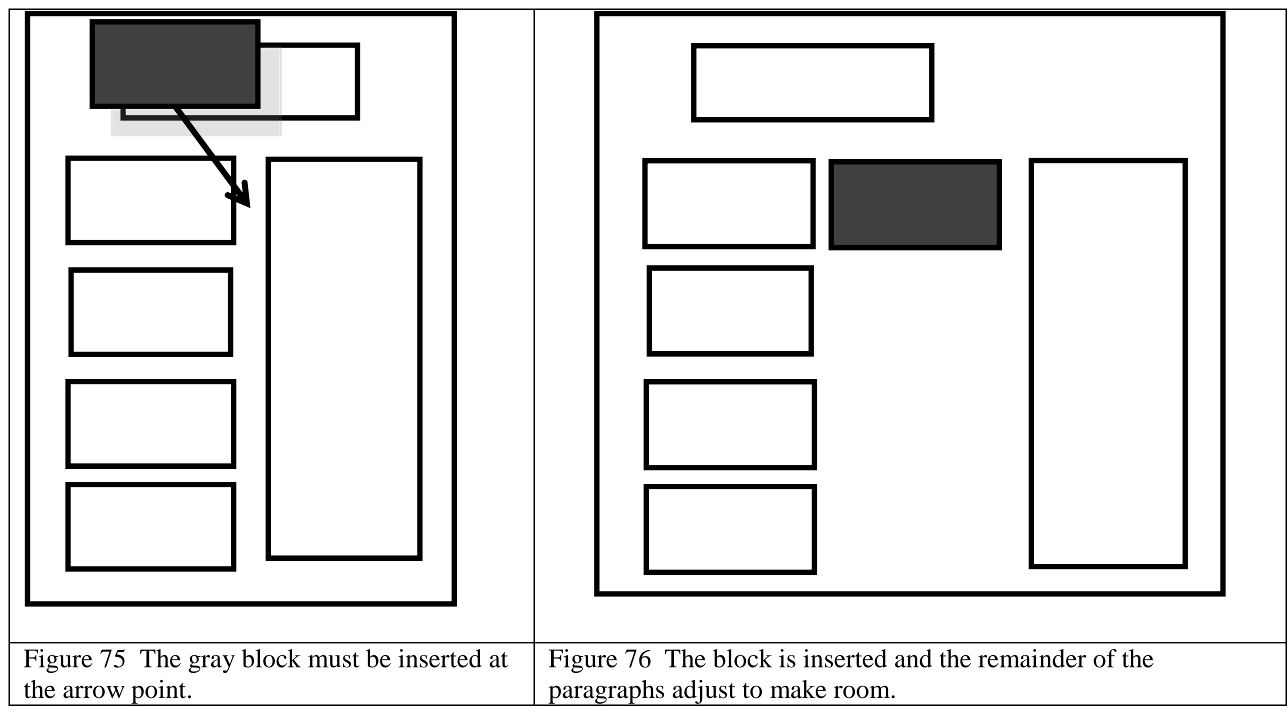


Figure 75 The gray block must be inserted at the arrow point.

Figure 76 The block is inserted and the remainder of the paragraphs adjust to make room.

Therefore, determining what elements should move where is a fairly difficult problem for a computer, but very natural for humans. Combining an HCT with a well established widget layout algorithm provides a good programmatic solution. Variable Intrinsic Size [11] (VIS) is a layout algorithm originally designed for placing hierarchical widgets in variable size windows. It has the trait of allowing for some components to be rigid and others to expand or contract, filling extra space. The original VIS algorithm provides for three classes of size: minimum, preferred, and maximum.

Non leaf nodes in the HCT are simply composites of their children. However, leaf nodes must define a value for each of the three size classes. Some leaves in the tree contain only whitespace where others have content. We define the values for each of the four size classes differently for these two kinds of leaves as shown in Figure 77.

	Minimum	Preferred	Maximum
Whitespace leaf	A small constant	Natural size	Infinity
Non-whitespace leaf	Natural size	Natural size	Natural size

Figure 77 The values of each size class for whitespace and non whitespace nodes

This makes the non-whitespace leaves rigid, unable to scale up or down, while the whitespace leaves can grow from a small constant to infinitely large. The makes it so that characters and figures are never be distorted due to resizing, but instead the whitespace will expand to fill up gaps. Whenever any change is made, the entire page is repacked to its preferred size.

SELECTION

Selecting image objects is a difficult task. There are many existing algorithms designed to assist users in image selection tasks such as intelligent scissors, min-graph cut, geodesic selection, and the humble lasso. While many of these techniques are powerful for foreground-background tasks, users would be frustrated to have to use them on our rasterized documents. These techniques are not well specialized for documents. In document selection users want to select elements of the document, such as the title or a paragraph, not pixels. Clearly there is a need for a selection algorithm that can operate against the HCT rather than only the document image. We introduce two different kinds of selection techniques. One employs a bounding box as its user-given input while the other works with a highlighter-like stroke. The output of both algorithms is a collection of HCT tree nodes to be selected.

Selection by Bounding Box

Bounding box selection is triggered by the user rubber banding out an area of a document. Ideally, a good bounding box algorithm will snap to elements of the document. For example, suppose a reader wants to select several lines of a document. The user has provided the bounding box shown in Figure 78. We would expect the two lines shown in Figure 79 to be selected.

product design, the focal point of an interactive session is to produce an information artifact that meets the goals of the user. The purpose of the user interface is to provide browsing and editing functions for manipulating and modifying the artifact until it reaches the state desired by the user.
The centerpiece of such an interactive system is the data being manipulated by the user through the user interface (UI). The data might be fairly

Figure 78 A bounding box provided by a user to select two lines.

product design, the focal point of an interactive session is to produce an information artifact that meets the goals of the user. The purpose of the user interface is to provide browsing and editing functions for manipulating and modifying the artifact until it reaches the state desired by the user.

The centerpiece of such an interactive system is the data being manipulated by the user through the user interface (UI). The data might be fairly

Figure 79 A selection rectangle like that above should select these two lines.

We expect this result even though the bounding box in Figure 78 doesn't even entirely contain both of these lines. Instead it clips off the top and right edge of the first line and the bottom of the second. A good selection algorithm should work even in the face of misdrawn input rectangles.

Users often misdraw the bounds of a selection for two reasons. First, they may be selecting quickly and thus haste causes a crude selection rectangle. Second, users are often unaware of the full bounds of an element because they fail to notice the full reach of ascenders and descenders in text. The selection rectangle of Figure 78 that clips both the top of the 'f' on the first line and the bottom of the 'y' on the second would be typical even of a fairly careful user.

A naïve selection algorithm is to select all nodes contained within the selection bounds. This algorithm would appear as in Figure 80.

```
1.  List<Node> BoundingBoxSelectAgainst(ImageArea roi, Node root) {
2.    List<Node> output = new List<Node>()
3.    RecursiveSelect(output,roi,root)
4.    return output
5.  }

1.  void RecursiveSelect(List<Node> selected,ImageArea roi,Node node) {
2.    ImageArea nodeBounds = node.Bounds
3.    if IsSelected(roi,nodeBounds,node) {
4.      selected.add(node)
5.    }

6.    if roi.intersects(nodeBounds)
7.      for each(Node child in node.Children)
8.        RecursiveSelect(selected,roi,child)
9.  }

1.  boolean IsSelected(ImageArea roi, ImageArea nodeBounds, Node node) {
2.    return roi.contains(nodeBounds)
3.  }
```

Figure 80 Naïve selection algorithm.

However, as seen in Figure 81 and Figure 82 this leads to two issues.

1. If a node is contained in the selection bounds, so are all of its children. Thus all nodes in the tree from the first contained node down to all of its leaves are selected. This causes the result to be cluttered.
2. If the user misses the bounds of an element by even a single pixel, that element is no longer contained and therefore not selected.

approach called Nanites that is designed to simplify the task of monitoring complex data structures.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms: Design, Human Factors, Performance

Additional Key Words and Phrases: CSCW, multiuser interfaces, Nanites, user interface

Figure 81 A bounding box intended for selection against some elements of text.

approach called Nanites that is designed to simplify the task of monitoring complex data structures.

Categories and Subject Descriptors: D.2.2 **Software Engineering**: Tools and Techniques; H.5.2 **Information Interfaces and Presentation**: User Interfaces

General Terms: Design, Human Factors, Performance

Additional Kev Words and Phrases: CSCW. multiuser interfaces. Nanites. user interface

Figure 82 Result of selection using containment algorithm.

The first issue is simple enough to remedy. When a parent node is selected, simply do not select the children because they are implicitly selected through the parent. Figure 83 codifies the modification.

```
1.  void RecursiveSelect(List<Node> selected,ImageArea roi,Node node) {
2.    ImageArea nodeBounds = node.Bounds
3.    if IsSelected(roi,nodeBounds,node) {
4.      selected.add(node)
5.      return
6.    }

7.    if roi.intersects(nodeBounds)
8.      for each(Node child in node.Children)
9.        RecursiveSelect(selected,roi,child)
10. }
```

Figure 83 Modification of RecursiveSelect to prevent selecting children of an already selected parent.

The second issue, that of snapping the bounding box to nearby elements, takes a little more consideration. The key to resolving this issue is, rather than enforcing full containment, to allow partial containment. We'll consider a node selected if it is "mostly contained" by the selecting rectangle. This calls for a threshold, labeled the ContainmentRatio, which determines the degree to which areas of the selection bounding box and a node's bounding box must overlap to be selected. A value of 1 degenerates to full containment (the node must be entirely inside the

selection bounding box), where a value of 0 means only intersection of a node with the selection bounds is required to be considered selected. Figure 84 shows the new IsSelected routine.

```
1.  boolean IsSelected(ImageArea roi, ImageArea nodeBounds, Node node) {
2.    if NOT roi.intersects(nodeBounds)
3.      return false
4.    if roi.contains(nodeBounds)
5.      return true

6.    ImageArea intersection = roi.intersection(nodeBounds)
7.    float intersectionArea = intersection.Area
8.    float nodeArea = nodeBounds.Area
9.    return (nodeArea-intersectionArea)/nodeArea <= (1-ContainmentRatio)
10. }
```

Figure 84 Modified IsSelected algorithm allows for nodes to be only partially contained by the selection bounds.

In practice setting ContainmentRatio to about 0.85 gives good results. Figure 85 demonstrates the results of this algorithm with similar input bounds as in Figure 81.

approach called Nanites that is designed to simplify the task of monitoring complex data structures.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; H.5.2 [**Information Interfaces and Presentation**]: User Interfaces

General Terms: Design, Human Factors, Performance

Additional Key Words and Phrases: CSCW, multiuser interfaces, Nanites, user interface

Figure 85 Result of selection using the partial containment algorithm.

Notice that while the user has clipped off the left and right edges in their selection, the partial containment algorithm has snapped to selecting the whole section. Figure 79 was also generated by the modified selection algorithm with Figure 78 as the input.

Selection by Stroke

Stroke based selection is motivated by highlighter-like interactions. Rather than using a bounding box to select an element, the user strikes through the element with the highlighter.

Highlighters are more natural to annotation based reading than bounding boxes and are simpler to use. When a user is drawing a bounding box, they must continually recheck the entire selection area to make sure that they have what they want in the bounds. However, with a highlighter, once drawn, the path does not change. Thus a user only needs to look where they're going next. These differences are sufficient to motivate including highlighter-like selection, and thus require a stroke based selection algorithm.

Sometimes people use highlighter strokes to diagonally strike through an entire paragraph [12], effectively treating the highlighter like a bounding box. Two naïve algorithms seem initially plausible. The first is to use the minimal bounding box of the selection stroke as the input into the bounding box selection algorithm.

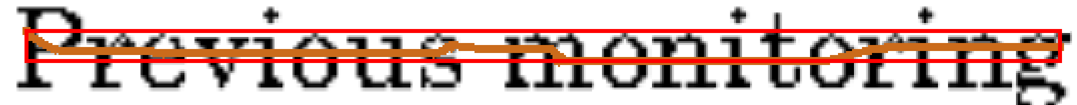


Figure 86 A stroke (shown in brown) is intended to select these letters. The red rectangle represents the minimal bounding box of the stroke.

Figure 86 demonstrates a stroke used to strike through a pair of words. We would expect this stroke to select the two words. However, since the bounding box of the stroke is very small it would make a poor input into the bounding box selection algorithm. A highlighter stroke need only cross through an element. In this case a horizontal line is sufficient to touch all the characters. Any vertical deviation in the stroke is purely coincidental hand jitter. Thus a perfect stroke, one with no vertical deviation at all, would have no area either, making it entirely unsuitable for the bounding box selection algorithm. An input like that of Figure 86 would yield

none of the characters as selected. Clearly an algorithm that considers the unique semantics of a stroke is necessary.

Another naïve algorithm is to select any element that intersects the path of the stroke. While this appears plausible at first, remember that the root node, the entire page, would intersect any stroke on it, and thus would always be selected. The obvious modification is to only select leaf nodes that intersect a stroke as in the Stroke Intersection algorithm of Figure 87.

```
1.  List<Node> StrokeSelectAgainst(Stroke stroke, Node root) {
2.    List<Node> output = new List<Node>()
3.    StrokeIntersectionSelect(output,stroke,root)
4.    return output
5.  }

1.  void StrokeIntersectionSelect(output (List<Node> selected,Stroke stroke,Node node) {
2.    ImageArea nodeBounds = node.Bounds
3.    if IsSelected(stroke,nodeBounds,node){
4.       selected.add(node)
5.       return
6.    }

7.    if roi.intersects(nodeBounds)
8.       for each(Node child in node.Children)
9.         StrokeIntersectionSelect (selected,roi,child)
10. }

1.  boolean  IsSelected(Stroke stroke, ImageArea nodeBounds, Node node)  {
2.    return stroke.intersects(nodeBounds)  AND node.isLeaf
3.  }
```

Figure 87 The Stroke Intersection stroke selection algorithm.

However, this too has failures. Figure 88 shows a stroke that does not intersect all of the relevant leaf nodes. The dot’s in the character ‘i' are separate from their stalk, and thus are separate leaf nodes. They are isolated because the stroke does not explicitly intersect them.

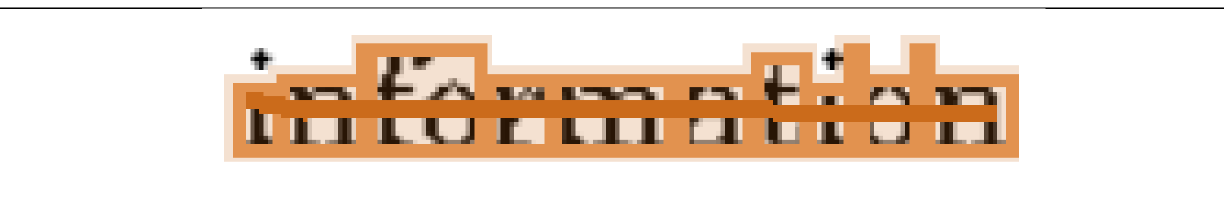


Figure 88 The highlighter stroke (dark brown) using the leaf intersection algorithm has missed the dots in the ‘i's.

Additionally the Stroke Intersection algorithm entirely fails to gracefully degenerate into bounding box selection when a paragraph is struck through as shown in Figure 89.

The focal point of many interactive systems is an information artifact being created and manipulated by one or more users through a user interface. The software components of such an interactive system perform their tasks relative to the data structures that represent the information artifact. System components interact with each other by changing these data and responding when relevant changes are made to them by other components. Perhaps the most difficult problem to be solved when building such data-centric systems is the *monitoring problem*. System components require the ability to watch for and respond to changes made to complex data structures. Previous monitoring approaches are geared toward monitoring single data items rather than entire data structures. This article describes a new monitoring approach called Nanites that is designed to simplify the task of monitoring complex data structures.

Figure 89 A highlighter stroke using the leaf intersection algorithm misses the paragraph all together when used to strike through it, but catches the leading gutters.

A compromise between the bounding box and leaf intersection algorithm gives good results. This modification, shown in Figure 90, uses two passes.

```
1.  List<Node> StrokeSelectAgainst(Stroke stroke, Node root) {
2.    List<Node> output = new List<Node>()
3.    StrokeIntersectionSelect(output,stroke,root)
4.    ImageArea bounding = GetSmallestBoundingBoxOfNodes(output)
5.    return BoundingBoxSelectAgainst(bounding,root)
6.  }
```

Figure 90 Modifying StrokeSelectAgainst makes the stroke selection algorithm more robust.

The first pass uses leaf intersection to collect a list of nodes that the stroke intersects. The second pass uses the bounding box of the results from the first pass as the selection bounds parameter in the bounding box selection algorithm. For the second pass a Containment Ratio of 0.6 is used to compensate for the additional sloppiness inherent to strokes.

The effect of this two pass algorithm is that the first pass finds the elements that designate the atomic pieces of the selection. For example in Figure 88 this is each individual character, or in Figure 89 it is the horizontal leading lines of the paragraph. The second pass widens the scope of the selection to include anything on the size magnitude of those atomic pieces. This causes the dot of the ‘i' or the paragraph to be selected because it is within the same size magnitude of the other characters, which are explicitly selected by the Stroke Intersection algorithm. Figure 91 and Figure 92 demonstrate the results of this modification. Notice in Figure 91 that the entire word is selected correctly even though the stroke does not explicitly intersect the dots of the ‘i's. Figure 92 demonstrates that this algorithm can gracefully degenerate to bounding box selection.

Figure 91 The entire word is selected with the two pass stroke selection algorithm.

The focal point of many interactive systems is an information artifact being created and manipulated by one or more users through a user interface. The software components of such an interactive system perform their tasks relative to the data structures that represent the information artifact. System components interact with each other by changing these data and responding when relevant changes are made to them by other components. Perhaps the most difficult problem to be solved when building such data-centric systems is the *monitoring problem*. System components require the ability to watch for and respond to changes made to complex data structures. Previous monitoring approaches are geared toward monitoring single data items rather than entire data structures. This article describes a new monitoring approach called Nanites that is designed to simplify the task of monitoring complex data structures.

Figure 92 The entire paragraph is selected with the two pass stroke selection algorithm.

VISUAL LINKS AND CONTEXT EXPANSION

Linking together ideas is a fundamental part of learning, making linking and cross referencing a fundamental part of annotating. The tradition method of linking is to use hyperlinks with text based tags to describe the target, as in Figure 93.

Dr. Olsen's Lab Homepage

Figure 93 A semi-opaque link shows a limited description of the target.

Unfortunately this kind of link is rather opaque, blocking most information about its target beyond a short string. PixelJot employs visual linking. A scaled down visual thumbnail of the link's target is shown as the link rather than a small blurb of text. This kind of linking allows for users to instantly see what the target is.

The HCT makes it possible to not only use visual links to show a specific target, but also to show that target in context. Links are useful because they don't bring the entire source document with them; instead they are a compact reference. However, when reviewing notes, often the context of the link may be forgotten. Of course the link can be followed in PixelJot by double clicking to open the target document in a new window. However, this distracts the reader from the task of reviewing the source work by entirely diverting attention to the new document. Instead it would be convenient to be able to find the context of a link and expand it *in place.*

In Place Expansions

PixelJot offers a method of expanding a visual link. Expansion of a link increases the view window around the target so that more of the target's document can be seen. For example, expanding the context of the link in Figure 94 would increase the scope of the target from the phrase to the entire line, as in Figure 95. Expanding the context again would reveal the entire paragraph as in Figure 96. Expanding again increases the scope to the entire column (Figure 97). Additional expansion of context after this would change the scope to the two columns together, then the entire document.

SPICIE framework

Figure 94 A visual link to a phrase.

This paper introduces the SPICIE framework for annexing

Figure 95 The visual link to a phrase shown in Figure 94 is expanded to show the entire line. The original phrase is selected.

ABSTRACT

This paper introduces the SPICIE framework for annexing display servers and sharing content on available screens. SPICIE allows a user carrying a portable device, such as a laptop or tablet, to annex additional screen space for her device. She then selects windows on her device to share with the annexed screens. In particular, overlapping windows on her personal device may be spread out (de-multiplexed) so that they do not overlap on the annexed screens. SPICIE protects user privacy by ensuring that only pixels generated by explicitly shared windows are transmitted to the display server. Multiple users may also simultaneously annex the screens to share content.

Figure 96 The line is expanded to show the paragraph.

ABSTRACT
This paper introduces the SPICIE framework for annexing display servers and sharing content on available screens. SPICIE allows a user carrying a portable device, such as a laptop or tablet, to annex additional screen space for her device. She then selects windows on her device to share with the annexed screens. In particular, overlapping windows on her personal device may be spread out (de-multiplexed) so that they do not overlap on the annexed screens. SPICIE protects user privacy by ensuring that only pixels generated by explicitly shared windows are transmitted to the display server. Multiple users may also simultaneously annex the screens to share content.

Author Keywords
de-multiplex pixels, annex screen, protect privacy

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
People are increasingly nomadic, and must manage large quantities of information. In particular, people need their personal information with them wherever they are. An effective technique for keeping information available is to carry that information with the user on a portable computer such as a laptop or tablet. These *personal devices* have portable form factors, but have limited screen space. Consequently, users are challenged to manage the increasing amount of information on a restricted screen space.

Most laptops and tablets have a Video Graphics Array (VGA) connector that can be used to add a single screen. A VGA connector increases the number of pixels available, but limits users to one additional screen and to the number of pixels on that screen that both the personal device and

Figure 97 The paragraph is expanded to show the column.

An additional click at this point will collapse the link back down to its original target shown in Figure 94. Thus expanding a link follows a cycle going from one level of context to the next higher until finally it returns to the original target. Each in-place expansion grows the size of the link to fit the new context. The layout algorithm handles pushing other elements aside to make room for the now larger link. The user can know that the cycle has restarted and the original target is being shown because this is the only time a click on the link will shrink it to a smaller size.

What to Expand to

The primary technical issue then is to actually know what the context of an element is. This is where all the effort of creating an HCT pays off. All visual links are actually a reference to a

node in the HCT. A node's context is each of its ancestors in the tree. Therefore, a simple upward traversal of the tree, starting at the node representing the target, finds the context of the linked node. Every node along the way to the top is another stop along the expansion cycle. While an upward traversal of the HCT is the heart of the actual algorithm, there are two amendments to this technique that vastly improve the quality of expansions.

Insignificant expansions

Sometimes the tree may yield an expansion that is entirely insignificant. For example, Figure 98 and Figure 99 demonstrate the expansion of a line of text by one level, adding only the small green area to the right of the line. The tree is built this way because a line above or below this line juts out slightly farther into the margin. Thus the tree cuts to the edge of that farther line. Finally, when the line shown in red is reached, the additional space on the edge is cropped away. This yields a node that contains the body of the line, and the small bit of margin (shown in green) on the right edge.   This insignificant expansion results from the target of the visual link being on the body of the line, and then expanding to the node that contains both the line body and the small margin.

widgets would have a standard "hear and say," much like
the "look and feel" standards in graphical user interfaces
[10].   With a standardized widget set, developers can

Figure 98 The target of a link as indicated by the red outline.

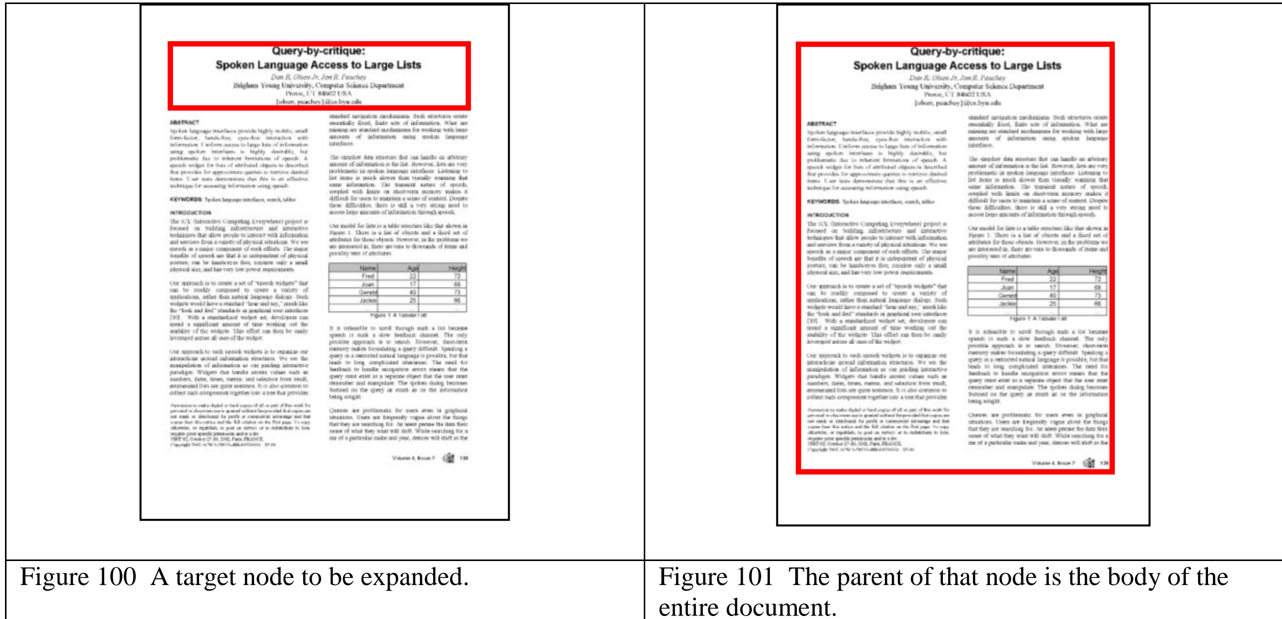
widgets would have a standard "hear and say," much like
the "look and feel" standards in graphical user interfaces
[10].   With a standardized widget set, developers can

Figure 99 The expansion of this target to its direct parent leads to the addition only of the green area on the right.

Regardless of why the tree is built this way, this diminutive degree of expansion is clearly not enough context to be helpful. Sometimes an expansion can be so slight that the user will not notice it has occurred, causing them to feel that clicking on a link sometimes works and sometimes does not. To address these insignificant expansions the algorithm defines a threshold called the Interesting Area Ratio. The ratio of the areas of the next level of expansion and the current must be greater than this ratio to be considered sufficiently interesting to include in the cycle. Parents with an area ratio below this value are simply skipped. PixelJot uses 1.5 as typical value of the Interesting Area Ratio.

Surging expansions

Opposite to the issue of some expansions being insignificant is that some are far too large. This happens when a node's parent jumps to a much larger size than the original node. Take, for example, the document in Figure 100. The title block of a paper is a likely target for a link if a user is keeping a bibliography. The parent of the title block is the entire body of the document. Expanding a link that targets the title (shown by the red area in Figure 100) expands the scope to the red area in Figure 101. Interactively this feels like the link is surging open, expanding from the relatively small title to the huge area of the document without any warning.

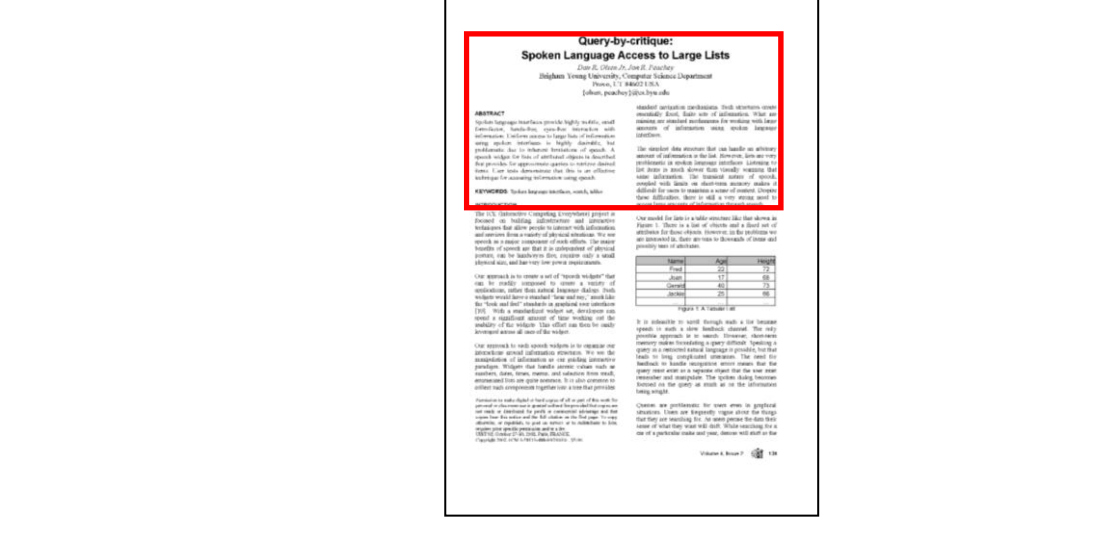


Query-by-critique:
Spoken Language Access to Large Lists

Figure 100 A target node to be expanded.

Figure 101 The parent of that node is the body of the entire document.

Introducing an intermediate step between these two expansions resolves this surging behavior. Whenever a parent node's area is much larger than its child's area, a new mid-level expansion is introduced between the two. This mid-level expansion displays an intermediate sized area between the child and the parent. Figure 102 demonstrates the intermediate expansion that would be placed between Figure 100 and Figure 101. The bounds of the intermediate are found by moving each edge of the original expansion halfway to the edge of the larger. We have found by experimentation that being about eight times bigger than ones child is a good definition of "much larger". It is about at an eight fold increase in size that an expansion begins to feel like it is surging. This intermediate step makes the expansion process feel smoother and easier to track what is happening.



Query-by-critique:
Spoken Language Access to Large Lists

Dan R. Olsen Jr, Jon R. Peachey
Brigham Young University, Computer Science Department
Provo, UT 84602 USA
{olsen, peachey}@cs.byu.edu

ABSTRACT

KEYWORDS

Name	Age	Height
Fred	22	72
Joan	17	68
Gerald	40	73
Jackie	25	66

Figure 1: A Tabular List

Figure 102 A mid-level expansion introduces an intermediate step between the expansions of Figure 100 and Figure 101.

Expansion results

After pruning out insignificant expansions and creating intermediate steps between larger ones, the expansion cycle is complete. Figure 103 demonstrates an example of how the cycle of expansions is ordered. The original target, the word “standard”, is shown with a red box at each step. For each expansion step the previous expansion step is shown in green.

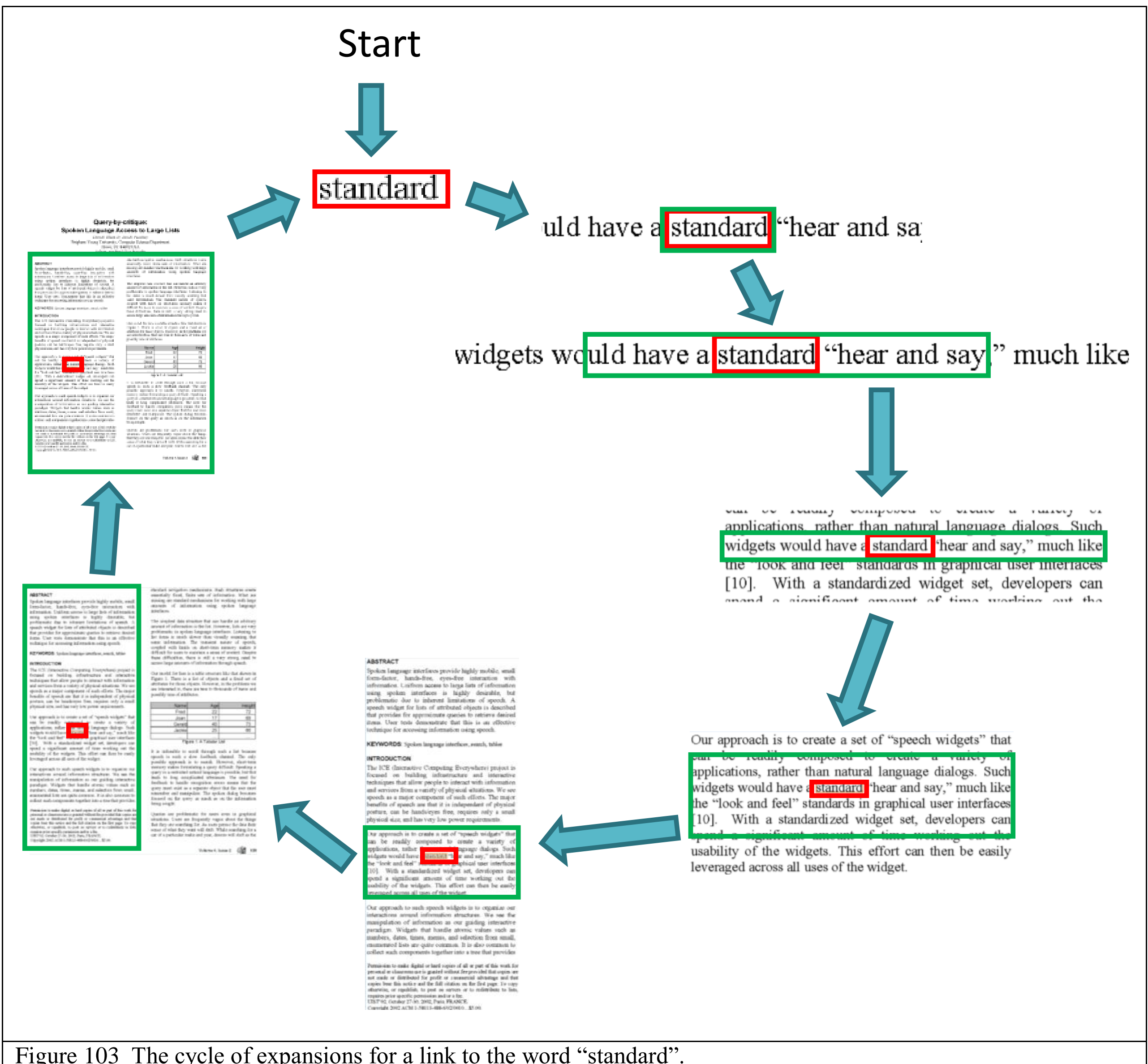


Figure 103 The cycle of expansions for a link to the word "standard".

Perceptual Scaling

Visual links, side by side, also cause some issues. In a page from a normal document, some bits of text are larger than others to reflect their importance relative to their context. Titles are often in very large font, where body text is much smaller. Removed from this context and inserted into another document, their relative size no longer makes sense. For example, in Figure 104

there is a link to the title and a link to other elements in the document. Since the title was typed in large font compared to the document body font, it seems large and out of place retaining this size in a link.

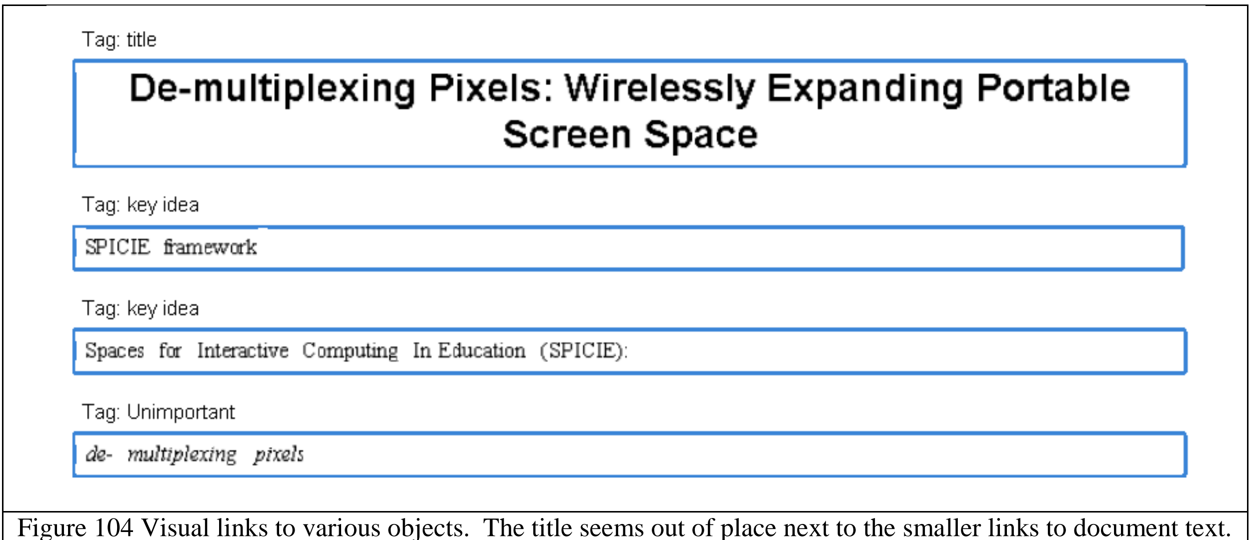


Figure 104 Visual links to various objects. The title seems out of place next to the smaller links to document text.

In the context of the entire paper the size of the title makes sense, but when it is the target of a link, the title appears far too large next to other elements. PixelJot introduces a technique for resizing elements to an ideal perceptual size. This allows links to be scaled to a size that is independent of the original context but instead is the "right" size for the content. The "right" size for an element is the size at which it can be comfortably read and understood. We call this the perceptual size. Figure 105 shows how we would expect the elements to be scaled.

Tag: title

De-multiplexing Pixels: Wirelessly Expanding Portable Screen Space

Tag: key idea

SPICIE framework

Tag: key idea

Spaces for Interactive Computing In Education (SPICIE):

Tag: Unimportant

de- multiplexing pixels

Figure 105 Visual links to various objects. The elements have been scaled to fit a comfortable perceptual size

We introduce four algorithms to rescale elements to their perceptual size. Each perceptual sizing algorithm takes an image as an input and outputs the scale that the image should be adjusted by. Typically this scale is usually less than or equal to 1 because few objects are included in papers that are already too small to be recognized. Each of these four algorithms will use the elements in Figure 106 to demonstrate their effectiveness. Figure 106 uses five different elements that we expect to be common targets of links in documents. In order these elements are a section header, a title, a paragraph, a line of text, and a single word.

ABSTRACT

ScreenCrayons: Annotating Anything

Stroke/Region Association

Having classified the highlight strokes and computed our continuity maps we now can compute the rectangular region associated with each stroke. Our algorithm searches in each of the four directions to find natural boundaries that correspond to each ink stroke. Stroke/region association has four cases: 1) area marks such as circles and scribbles, 2) horizontal underline, 3) horizontal highlight and 4) margin bars.

region associated with each stroke. Our algorithm searches

associated

Figure 106 A section header, title, paragraph, line, and word in their original size.

Please note that what is important is the *relative* size of elements to each other, not the absolute size that you see in their result figures. Each algorithm may be correct by some constant amount, and thus the absolute size of an element is irrelevant.

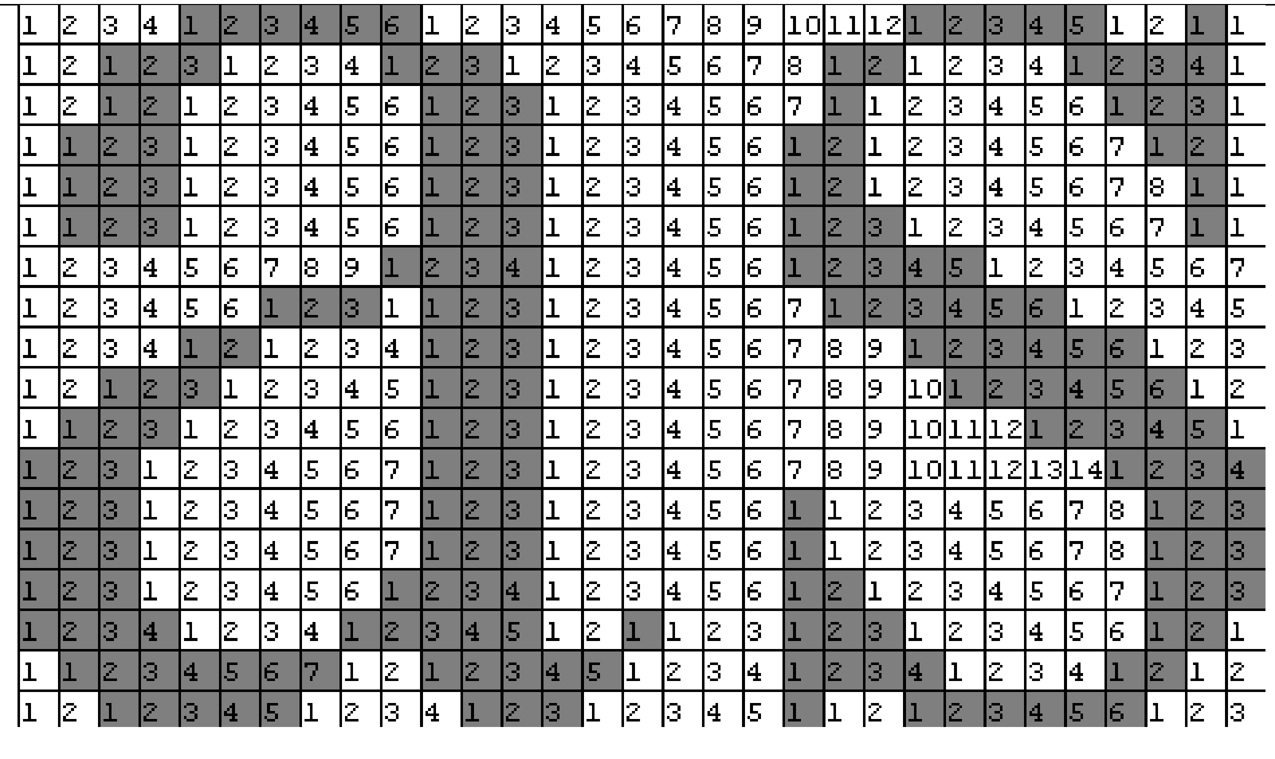


Figure 107 A horizontal continuity map of two adjacent characters.

The goal of the first two of these four perceptual sizing algorithms is to discover the dominant width of a region of an image. Since we are focusing on scaling text, this dominant width is the width of the pieces that make up text. In Figure 107 this width is about three. This is because the right stalk of the ‘a’ tends to be about three pixels wide as well as the average width of its loop. Even the diagonal of the ‘s’ is about three pixels wide if measured at an angle. Once this dominant width is found, the image is scaled to put that width at about the width appropriate to human eye acuity. Since that is about three pixels on most screen, the characters in Figure 107 wouldn’t need to be scaled at all. This is unsurprising, since they come from regular sized text. However, the character found in Figure 108, each leg of which is about five pixels wide, would be scaled by $\frac{3}{5}$ thus placing it at about three pixels wide. This technique requires an efficient method of finding the width of characters of text.

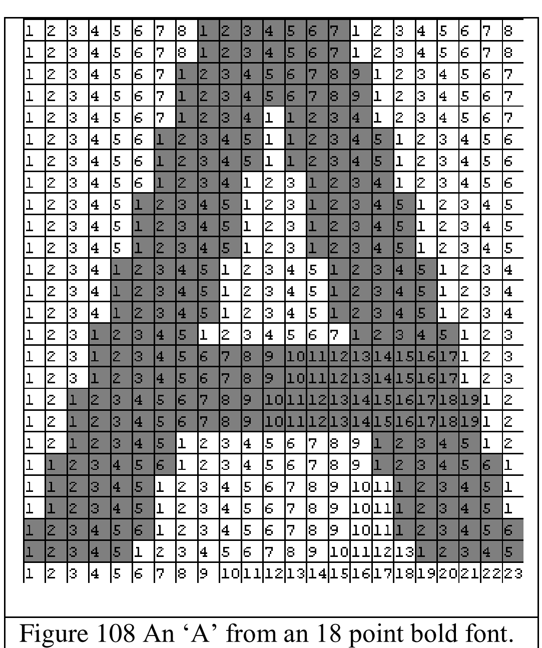


Figure 108 An ‘A’ from an 18 point bold font.

Continuity histograms

Histograms of continuity images are an efficient way gathering information about document images. Each run in a continuity image represents an uninterrupted visually uniform strip. Images of text have a great deal of breaks, and thus have, on average, many shorter runs. Images of larger or bold text will have fewer but longer runs. A histogram of a continuity image counts the number of runs of each possible length. Histograms of continuity images are very similar to the frequency spectrum of the original image. If the image has frequent breaks, it will experience high power in upper frequency areas, and therefore more histogram votes on the shorter run lengths. Fewer breaks leads to longer runs and more histogram votes in the upper range which is the same in frequency space more power in lower frequencies. Rather than using the Fourier Transform, we use the histogram technique because we can manipulate the histogram voting technique to improve results.

A histogram of a continuity image works by counting only the length of each run. For example, if a histogram were being created from the image in Figure 107, the first values to be added would be a 4, 6, 12, 5, 2, etc. When adding a run to a histogram, there are several options for weighting the strength of its vote:

1. Single vote
2. Weighted vote
3. Square Root vote

The first voting technique is to allow each run to count for only one vote. Using this voting technique against the continuity image in Figure 107, we'd get the histogram shown in Figure 109.

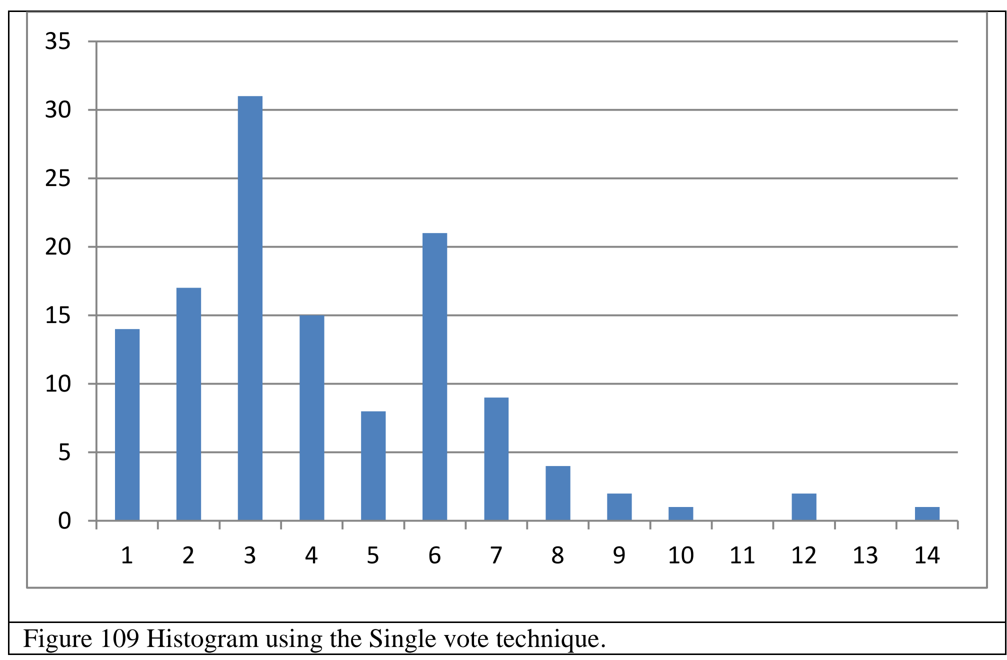


Figure 109 Histogram using the Single vote technique.

Note that the dominant three widths are: 3 (the width of the characters), 6, (the width between the characters), and 2 (the narrow neck of the serifs). These results are mostly there, however, since long runs contain many pixels, longer runs are under considered. Thus the lower end of the histogram has numerous votes, where the few longer runs only have one or two.

Naturally the opposite technique is to count each run for the number of pixels it contains. Figure 110 demonstrates the resultant histogram.

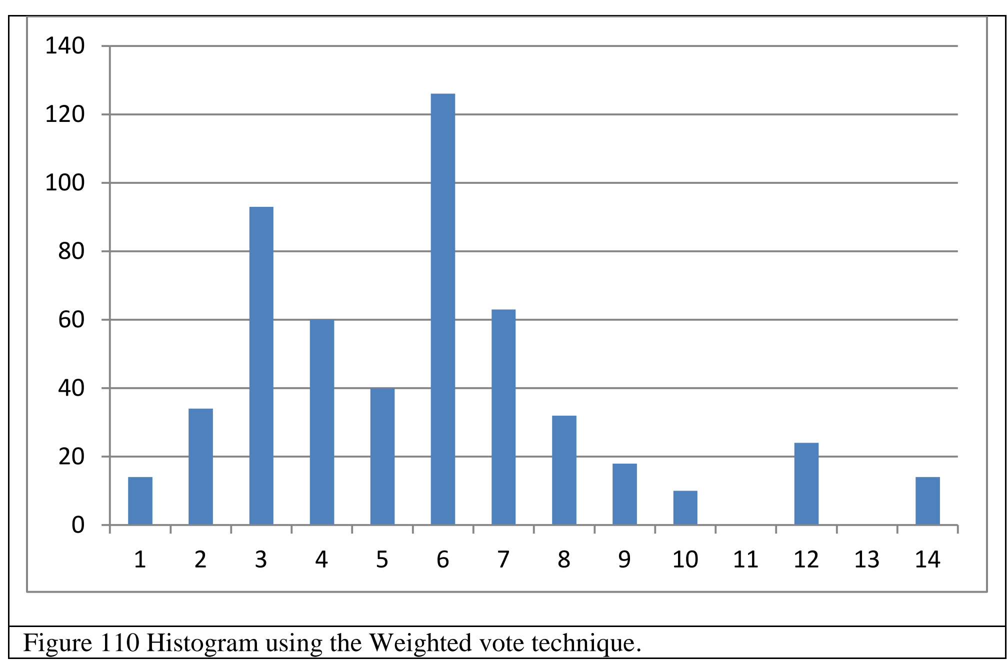


Figure 110 Histogram using the Weighted vote technique.

Notice that the larger runs seem more fairly weighted. The dominant widths here are: 6, 3, and 7 (also a common width between characters). However this technique is also a folly because we are interested mostly in the shorter runs, but this technique allows whitespace to dominate the histogram. While the sample image, Figure 107, does not have much whitespace, imagine a continuity image of a paragraph. The leading between lines of text are uninterrupted whitespace. These long stretches of whitespace, all of which are exactly the same length, would combine together to entirely dominate the more diffused vote of inter-text and intra-text runs.

This method, counting each run for the number of pixels it contains, over weights the long uninteresting runs. Counting each run for only one value allows very short runs to dominate. The best solution comes by counting each run for the square root of its length. This gives longer

runs a heftier vote, but allows the shorter and more interesting runs to collectively carry significant weight.

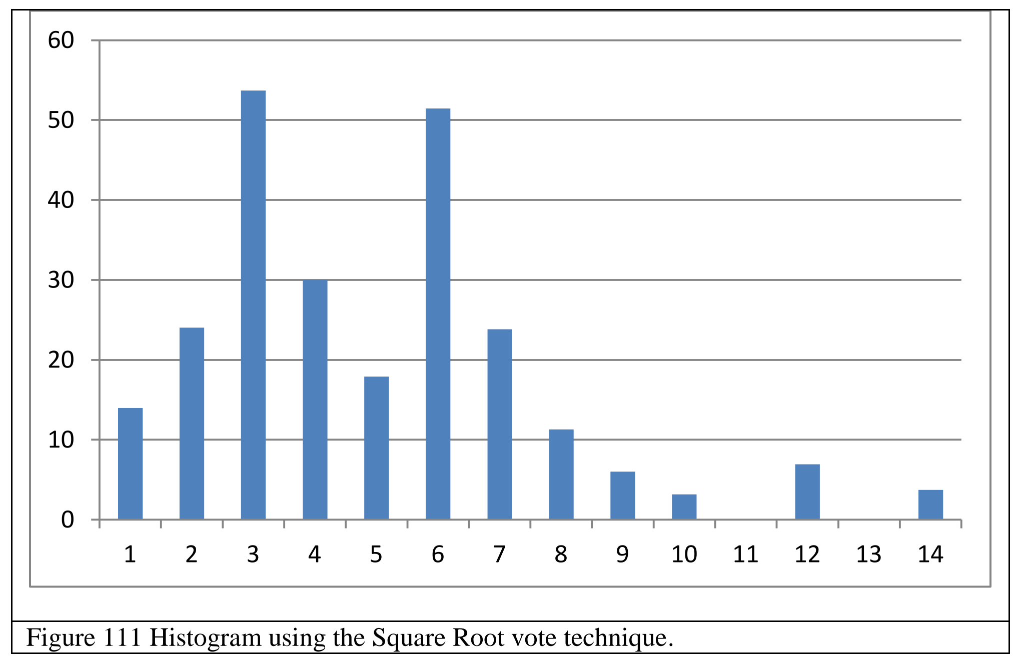


Figure 111 Histogram using the Square Root vote technique.

Figure 111 shows that the resultant histogram. Using this voting technique the dominant votes are: 3, 6, and 4 (the size of the wider necks of the serifs). The voting technique is fairer and seems to favor finding the frequency of small text like areas.

Each of four perceptual sizing procedures uses a continuity histogram with the Square Root voting technique to attempt to find the correct degree of scaling for an image. They work by using continuity histogram to find dominant width of image detail (in this case the size of text), and then scale the image so that this detail is some constant size.

Mean Run to Constant

The Mean Run to Constant procedure calculates the continuity histogram of the image, then finds the length of the mean run. The returned scale is some constant over this mean run width. The desire is that the mean run length, according to the continuity histogram, will be the dominant width of the text. Figure 112 shows each of the example clippings as sized by the results of the Mean Run to Constant procedure. Comparing Figure 112 to Figure 106 shows that Mean Run to Constant seems to favor the blocks of normal text (the paragraph, line, and word) and has reduced the bolded text (the section header and title) to be much smaller.

ABSTRACT

ScreenCrayons: Annotating Anything

Stroke/Region Association

Having classified the highlight strokes and computed our continuity maps we now can compute the rectangular region associated with each stroke. Our algorithm searches in each of the four directions to find natural boundaries that correspond to each ink stroke. Stroke/region association has four cases: 1) area marks such as circles and scribbles, 2) horizontal underline, 3) horizontal highlight and 4) margin bars.

region associated with each stroke. Our algorithm searches

associated

Figure 112 Results of the Mean Run to Constant perceptual sizing procedure.

By Rank

The By Rank procedure is very similar to the Mean Run to Constant. By Rank also uses the histogram, but this time finds the top five run lengths with the most votes. The minimum of these top run lengths is used as the denominator over some constant to serve as the scale. Figure 113 demonstrates that this procedure punishes the bold text even more strongly than Mean Run to Constant, but its results are similar.

ABSTRACT

ScreenCrayons: Annotating Anything

Stroke/Region Association

Having classified the highlight strokes and computed our continuity maps we now can compute the rectangular region associated with each stroke. Our algorithm searches in each of the four directions to find natural boundaries that correspond to each ink stroke. Stroke/region association has four cases: 1) area marks such as circles and scribbles, 2) horizontal underline, 3) horizontal highlight and 4) margin bars.

region associated with each stroke. Our algorithm searches

associated

Figure 113 Results of the By Rank perceptual sizing procedure.

Inter Runs

The Inter Runs procedure is the most complicated of the sizing algorithms. It is designed to work specifically on blocks of text and tries to determine the width of space between characters. Figure 114 and Figure 115 reveal the horizontal continuity map of some bold and non-bold characters. Notice that the bold characters seem to have a width of about five pixels where the non-bold characters have a width of about three. Both the Mean Run to Constant and the By Rank procedures worked by attempting to determine these typical character widths and scale to compensate for them. However, both of these algorithms have failed with bold characters. This is because in readability what matters is contrast. While bold characters are wider, the spacing between characters is nearly the same for bold text as normal text. Thus scaling bold text by the width of its characters, rather than the width of its whitespace, makes the resultant gaps too small to distinguish a difference between the letters. Look back at Figure 113 and Figure 112. Notice

that the reason the title is hard to read isn't that the sticks, loops, and curves of the characters are too small, but that the holes within characters and the gaps between them are too small to separate those sticks, loops, and curves.

Instead of trying to determine the typical width of a character, the Inter Runs procedure works by finding the width of space *between* characters. Examination of Figure 114 shows there to be on average ten pixels between this bold text, and Figure 115 shows about seven pixels between the normal text. Thus the bold text would be scaled down slightly relative to the normal text. But not to the extent that Mean Run to Constant or By Rank do. Inter Runs also works by using the continuity histogram, but only counts background colored runs that come between non-background colored runs.

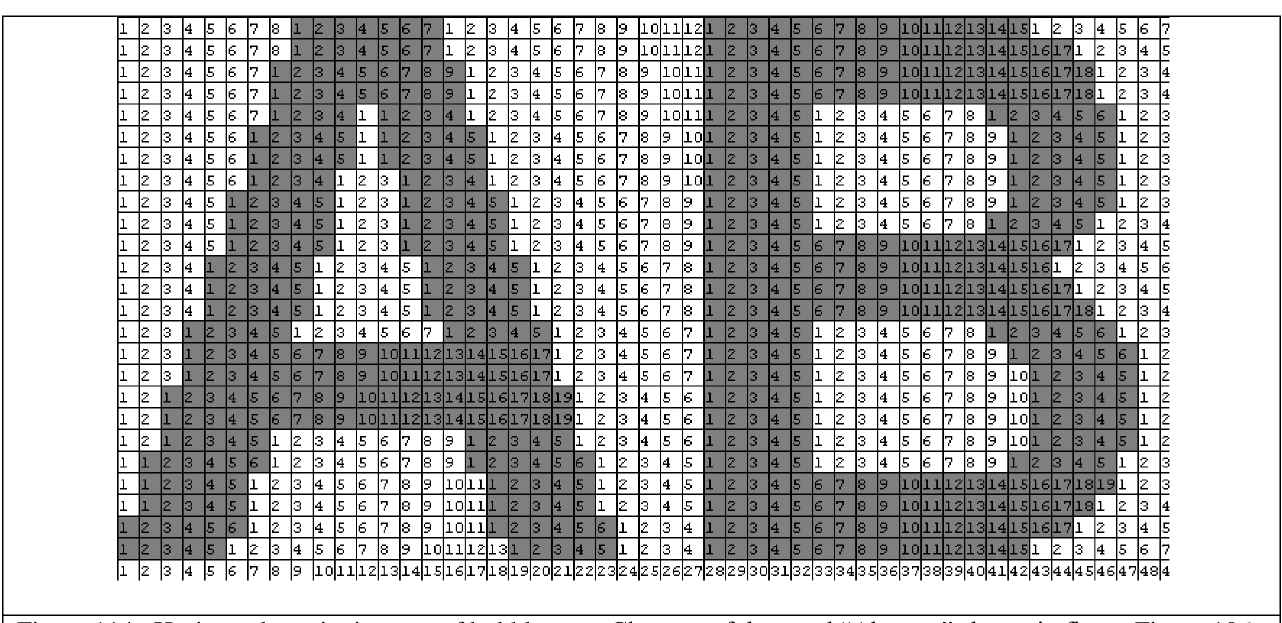


Figure 114 Horizontal continuity map of bold letters. Close up of the word "Abstract" shown in figure Figure 106.

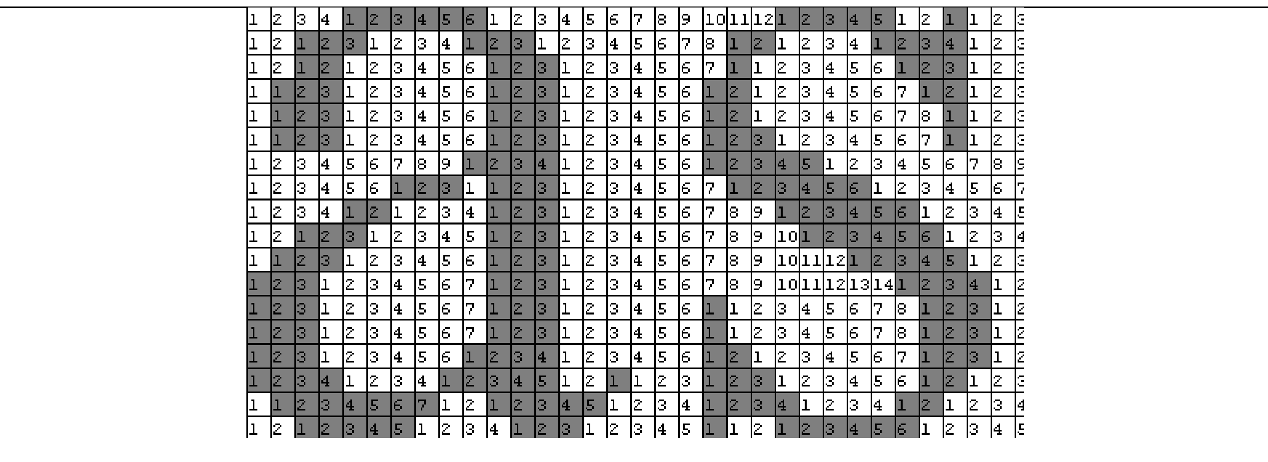


Figure 115 Horizontal continuity map of normal font letters. Close up of the word “associated” shown in Figure 106.

As seen in Figure 116, the Inter Runs procedure works more reliably with the bold section header and title font. However, sometimes this algorithm can be tricked by non-text only elements such as charts since it confuses space between text with space between chart borders.

Figure 116 Results of the Inter Runs perceptual sizing procedure.

Ensemble

The final sizing procedure attempts to merge the advantages of each of these previous perceptual sizing procedures by using a weighted average of the outputs of each. This ensemble procedure is what PixelJot uses to decide the scale applied to visual links, and generated the results we saw in Figure 105. There is a slight bias towards not scaling the results at all which allows for attention grabbing elements such as the title to retain some portion of their visual salience in links. See results in Figure 117.

ABSTRACT

ScreenCrayons: Annotating Anything

Stroke/Region Association

Having classified the highlight strokes and computed our continuity maps we now can compute the rectangular region associated with each stroke. Our algorithm searches in each of the four directions to find natural boundaries that correspond to each ink stroke. Stroke/region association has four cases: 1) area marks such as circles and scribbles, 2) horizontal underline, 3) horizontal highlight and 4) margin bars.

region associated with each stroke. Our algorithm searches

associated

Figure 117 Results of the Ensemble perceptual sizing procedure.

SYNOPSIS PAGES

Notes are only valuable if they can be easily accessed at a later date. PixelJot introduces a series of synopsis pages for reviewing documents and notes. There are various kinds of synopses possible including index, summary, and search pages. Each of these pages is created on-the-fly by examining the notes, tags, and highlights inside a document and compiling visual links to the

results into a custom built HCT. An index and search may be conducted on a single document, a folder of documents, or the entire digital notebook.

Index

An index is created by examining all of the tags in a given scope (document, folder or notebook) and sorting them in alphabetical order. Each tag is then inserted via a visual link into a custom built HCT. For example, Figure 118 demonstrates an index built from the document shown in Figure 119. If a user is interested in finding a tag based on a certain topic, they need only create the index and find the section most related to their desired topic.

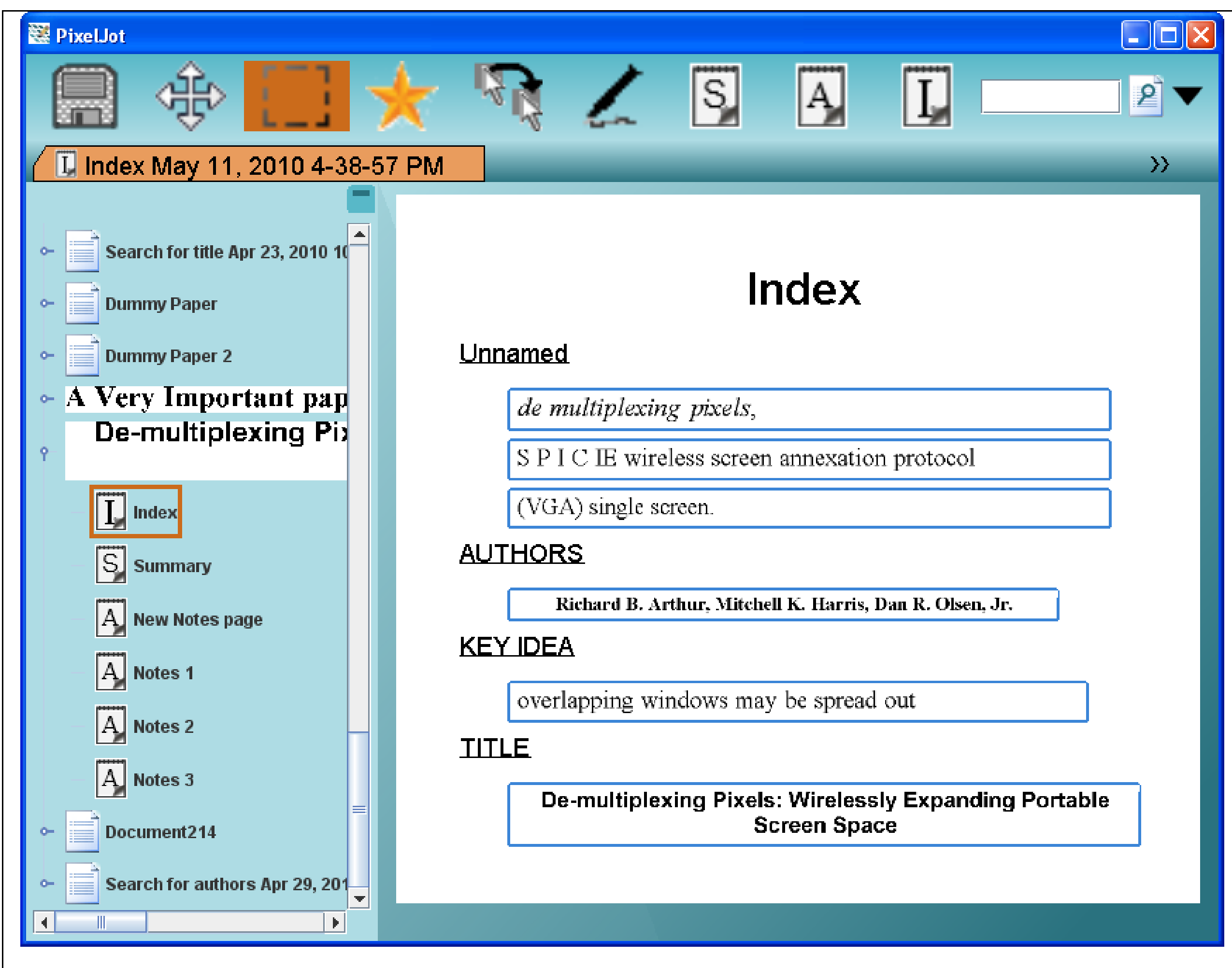


Figure 118 An index built from the document in Figure 119.

De-multiplexing Pixels: Wirelessly Expanding Portable Screen Space

Richard B. Arthur, Mitchell K. Harris, Dan R. Olsen, Jr.
Brigham Young University
3361 TMCB, Provo, UT, 84602-6576, USA
startether@startether.com, heneryville@gmail.com, olsen@cs.byu.edu

ABSTRACT
This paper introduces the SPICIE framework for annexing display servers and sharing content on available screens. SPICIE allows a user carrying a portable device, such as a laptop or tablet, to annex additional screen space for her device. She then selects windows on her device to share with the annexed screens. In particular, overlapping windows on her personal device may be spread out (de-multiplexed) so that they do not overlap on the annexed screens. SPICIE protects user privacy by ensuring that only pixels generated by explicitly shared windows are transmitted to the display server. Multiple users may also simultaneously annex the screens to share content.

Author Keywords
de-multiplex pixels, annex screen, protect privacy

ACM Classification Keywords
H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION
People are increasingly nomadic, and must manage large quantities of information. In particular, people need their personal information with them wherever they are. An effective technique for keeping information available is to carry that information with the user on a portable computer such as a laptop or tablet. These *personal devices* have portable form factors, but have limited screen space. Consequently, users are challenged to manage the increasing amount of information on a restricted screen space.

Most laptops and tablets have a Video Graphics Array (VGA) connector that can be used to add a single screen. A VGA connector increases the number of pixels available, but limits users to one additional screen and to the number of pixels on that screen that both the personal device and screen support.

Desktop machines, however, can support extra graphics hardware, allowing users to greatly increase the screen space available for applications. Unfortunately, desktop machines are not very mobile, so personal devices are still limited to a single additional screen.

Yeah! I hate moving my desktop around. Take forever!

Instead of having a single screen limit, a user should be able to add screens to meet her need for space. This paper introduces Spaces for Interactive Computing In Education (SPICIE): a wireless screen annexation protocol that allows users to utilize multiple screens via a personal device, thus overcoming the cable limitations. SPICIE facilitates *de-multiplexing pixels*, which allows users to spread out windows from their personal device to the attached screens. Spreading these windows provides users context while working in applications [1, 2, 5, 9, 15]. In addition, the SPICIE protocol supports multiple concurrent personal devices, so several users can simultaneously collaborate on the same screen space, as shown in Figure 1.

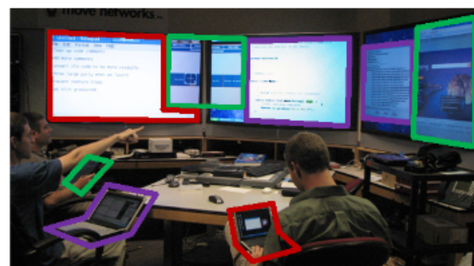


Figure 1—Collaborative group meeting. The windows and their source computer are outlined in the same color.

User Scenarios
With a trusted portable computer, most application interaction is likely to happen on the personal device (i.e. primary screen). Additional screens provide context for the

Figure 119 A document with various tags, highlights, and comments.

Search

While an index displays all of the tags with a given title, a note taker can also search for tags with a given string using the search bar. A search can be scoped to the document, folder, or notebook. A search results in a page with a visual link to every tag that bears the given string. The result of a search for the string “key idea” on the document in Figure 119 is shown in Figure 120.

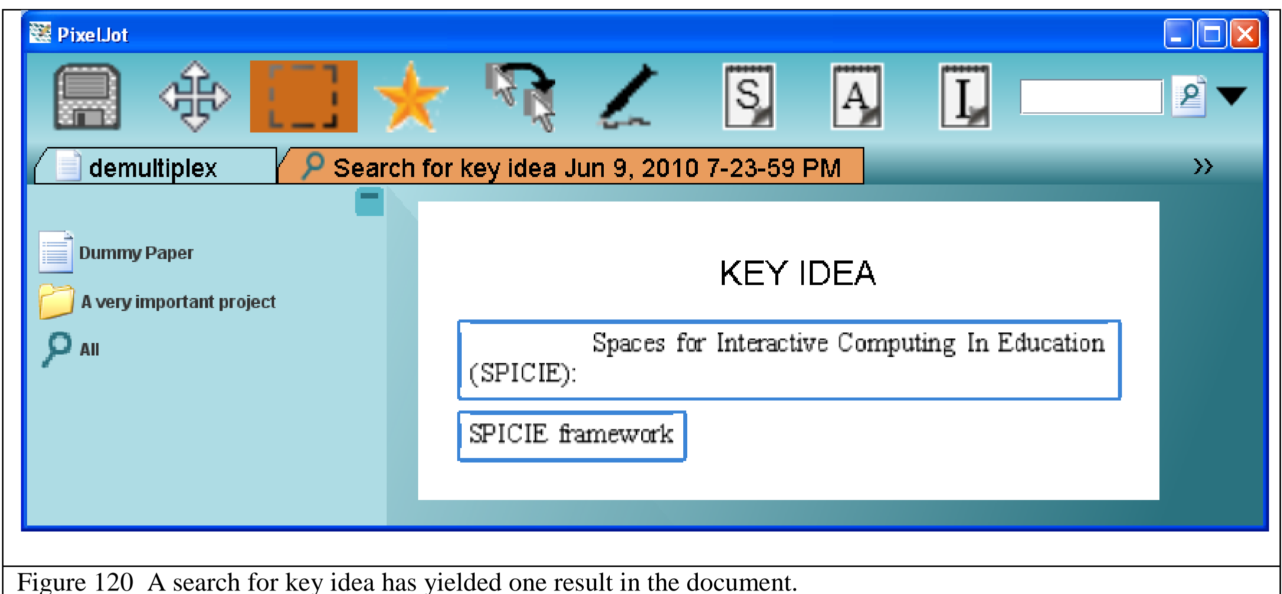


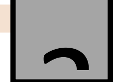
Figure 120 A search for key idea has yielded one result in the document.

Summaries

Indexes place tags in alphabetical order and searches place them in order of discovery. Neither of these synopses are conducive to summarizing documents. Summary pages present visual links to comments, tags, and highlights in the order that they appear in the document. For humans, the proper ordering of a document is easy. Figure 121 how we would order the tags, highlights, and comments of Figure 119. Figure 122 provides the corresponding summary that has listed the annotation in the correct order.

De-multiplexing Pixels: Wirelessly Expanding Portable Screen Space

Richard B. Arthur, Mitchell K. Harris, Dan R. Olsen, Jr.
Brigham Young University
3361 TMCB, Provo, UT, 84602-6576, USA
startether@startether.com, heneryville@gmail.com, olsen@cs.byu.edu



ABSTRACT

This paper introduces the SPICIE framework for annexing display servers and sharing content or ... creens. SPICIE allows a user carrying a portab... ch as a laptop or tablet, to annex additional s... for her device. She then selects windows on ... share with the annexed screens. In particular, overlapping windows on her personal device may be spread out (de-multiplexed) so that they do not overlap on the annexed screens. SPICIE protects user privacy by ensuring that only pixels generated by explicitly shared windows are transmitted to the display server. Multiple users may also simultaneously annex the screens to share content.

Author Keywords

de-multiplex pixels, annex screen, protect privacy

ACM Classification Keywords

H5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

INTRODUCTION

People are increasingly nomadic, and must manage large quantities of information. In particular, people need their personal information with them wherever they are. An effective technique for keeping information available is to carry that information with the user on a portable computer such as a laptop or tablet. These *personal devices* have portable form factors, but have limited screen space. Consequently, users are challenged ... the increasing amount of information on ... screen space.

Most laptops and tablets have a Vid... Array (VGA) connector that can be used to add a single screen. A VGA connector increases the number of pixels available, but limits users to one additional screen and to the number of pixels on that screen that both the personal device and screen support.

Desktop machines, however, can support extra graphics hardware, all... to greatly increase the screen space availab...cations. Unfortunately, desktop machines are ...bile, so personal devices are still limited to a si...al screen.

Yeah I hate moving my desktop around. Take forever!

Instead of having a single screen ... should be able to add screens to meet her ne...e. This paper introduces Spaces for Interactiv... In Education (SPICIE): a wireless screen annexation protocol that allows users to utilize multiple screens via a personal device, thus overcoming the cab...s. SPICIE facilitates *de-multiplexing pixels*, ...ws users to spread out windows from their p...ce to the attached screens. Spreading these wi...des users context while working in applications [1, 2, ...9, 15]. In addition, the SPICIE protocol supports multiple concurrent personal devices, so several users can simultaneously collaborate on the same screen space, as shown in Figure 1.

Figure 1—Collaborative group meeting. The windows and their source computer are outlined in the same color.

User Scenarios

With a trusted portable computer, most application interaction is likely to happen on the personal device (i.e. primary screen). Additional screens provide context for the

Figure 121 The expected flow order of the tags, highlights, and comments in Figure 119.

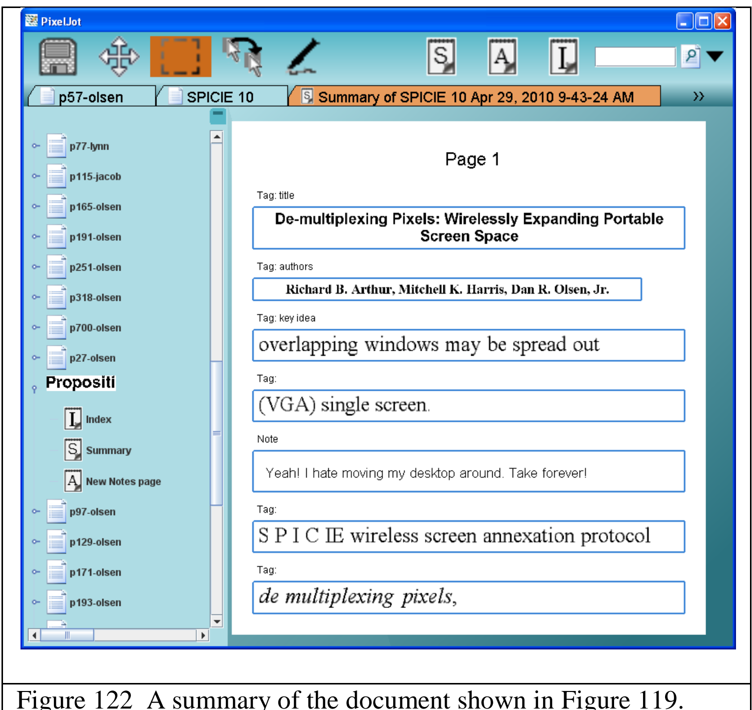


Figure 122 A summary of the document shown in Figure 119.

While it is easy for a human to know the proper flow order of a document, it is much harder for an algorithm to do so. The primary challenge in creating summaries is discovering what the relative order of each element is. Remember that the only resources available are the original image and the HCT. The naïve approach, sorting elements by their proximity to the top of the page, fails with multiple column papers. Clearly the internal structure matters in a way that a top to bottom or left to right approach cannot capture. All elements in the left column should be listed before those in the right. All elements in the title section should come before those in the body of the document.

PixelJot attempts to infer the order of elements by using the HCT. By conducting a standard depth first ordering of the HCT, each element can be compared to its neighbor. What makes this

work is that each node must visit its children in an order that follows the reading order of the document. Let's take a look at a simple block diagram of a document as an example. Figure 123 represents a block diagram of is simple document. This document has two columns, four paragraphs, and a title section.

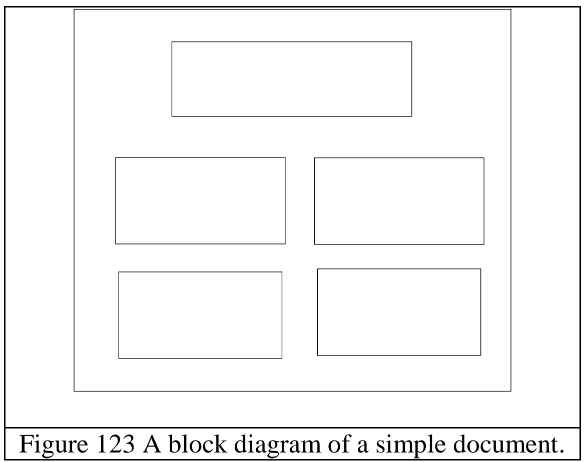


Figure 123 A block diagram of a simple document.

Figure 124 shows the HCT that would result from segmenting this block diagram. The title and the document body are first divided. Then the body is broken into columns then paragraphs. Of course the paragraphs would be divided into lines then characters, but for simplicity we only show to the paragraph level. Again, a real HCT would not be labeled, but for clarity Figure 124 has been labeled at each level.

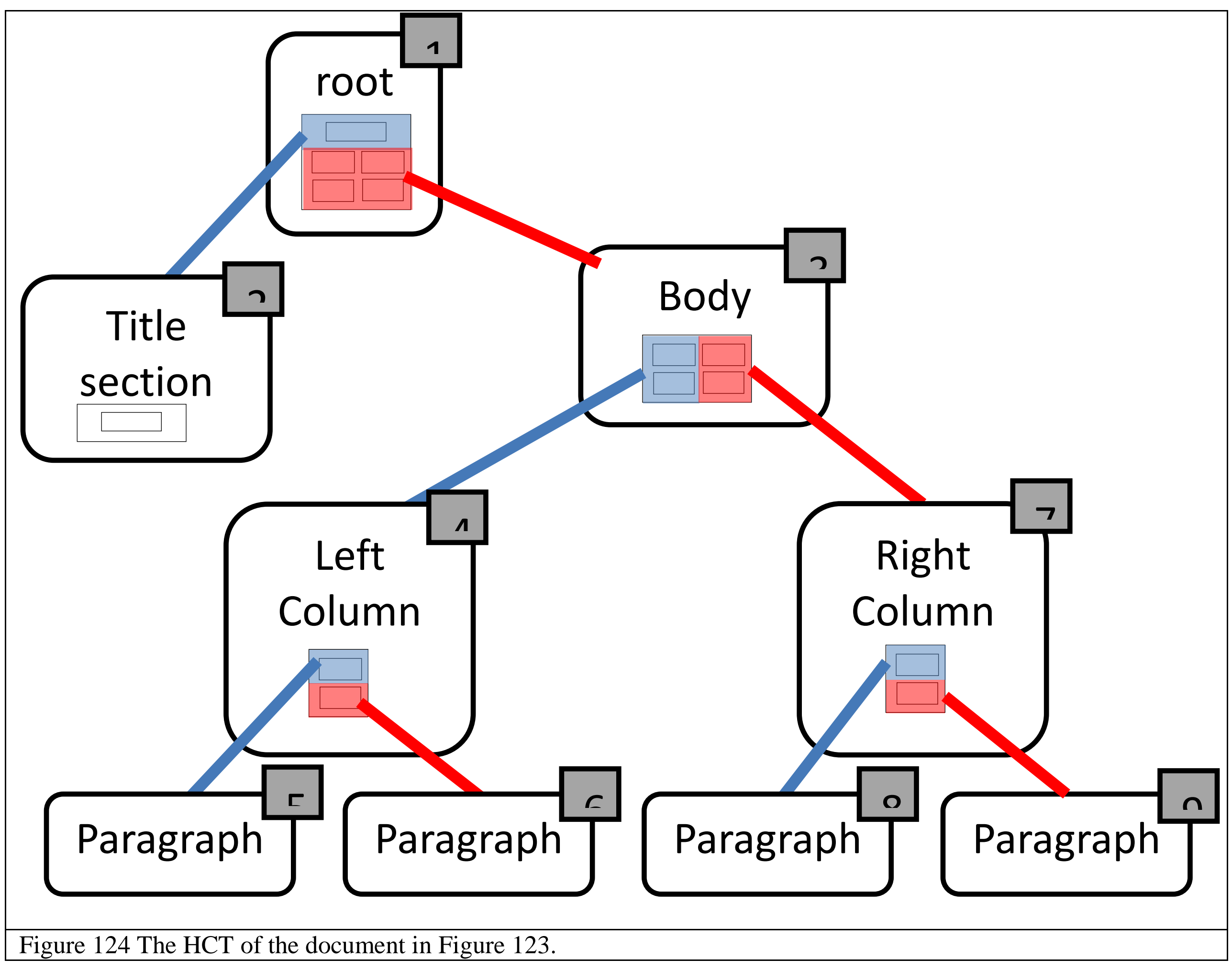


Figure 124 The HCT of the document in Figure 123.

An in-order depth first ordering of this tree would number the elements as depicted by the gray boxes. Resulting in the final paragraphs being numbered as in Figure 125.

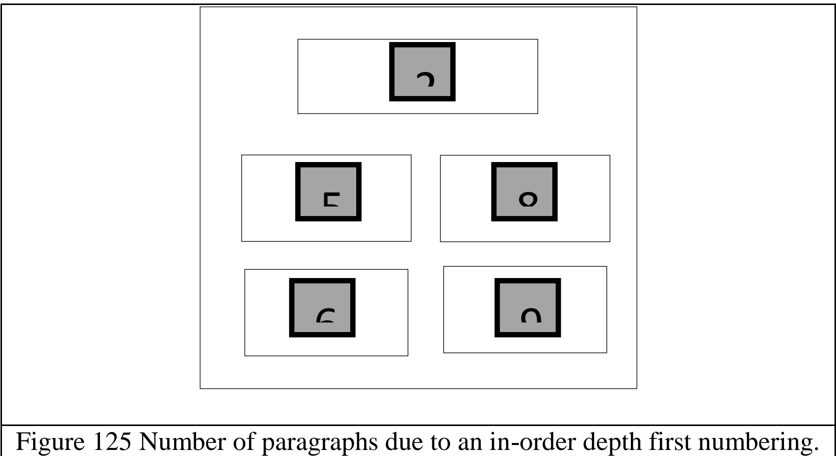


Figure 125 Number of paragraphs due to an in-order depth first numbering.

In Latin based languages documents are read from top to bottom, left to right. Thus grid nodes visit their children from top to bottom major order, left to right minor. This causes the grid node separating the two columns to visit the left column before the right, and the grid node separating paragraphs to order them from top to bottom.

We are fortunate that English is so consistent about its top to bottom, left to right reading order. Recall that the HCT is unlabeled, and thus we don't know which nodes actually represent a column break or a paragraph break. If we were to ever encounter a language that orders lines of text in a paragraph from top to bottom, but orders paragraphs from bottom of the page to the top, we would be unable to correctly order the HCT because we couldn't tell which nodes represent lines and which represent paragraphs.

Crop nodes visit each margin first and the center last. In-order depth first traversal of the HCT succeeds in ordering elements in standard single or multi-columned research papers. Examine

Figure 121 and Figure 122 again, and observe that highlights and tags are ordered exactly how we would expect them.

Conclusion

PixelJot resolves the tradeoff between universality and content awareness. We take it as self-evident that our solution is universal because the virtual printer technique can print anything into images. In the worst case scenario a screen shot could be used, meaning that anything displayable on a computer is also capturable and importable. We assert that our solution is content aware, and that content awareness is useful.

Content Awareness

PixelJot segments document images into a hierarchical context tree. This HCT divides the document into its core elements (the leaves of the tree), and stores information on how each element relates to another (the hierarchical organization of the tree). This HCT makes PixelJot content aware.

Usefulness of Content Awareness

Now let us examine utility. To demonstrate that being content aware is significant, we have demonstrated several tools that utilize the HCT. These tools then are a proof of utility. If they are helpful in annotating tasks, then they justify our contributions as significant. Selection of document elements, expanding links to view their context, reflowing documents when elements are inserted or resized, and discovering the visual order of documents for summary creation are

all examples of tools that employ the hierarchical context tree. We believe that these tools are useful for annotation tasks and justify the contribution of an HCT as significant.

We have introduced PixelJot, a universal and content aware annotator. PixelJot universally imports all document as images, uses perceptual cues to infer a hierarchical context tree, and uses this tree to offer many tools for annotating documents. We have introduced a limited set of tools to demonstrate the usability and feasibility of the HCT. There are many more operations or tools that could be invented based upon an HCT that would add even more value to annotation tasks. PixelJot, as a working prototype of a content aware annotator, is a useful and a powerful tool in annotating documents.

FUTURE WORK

Our primary contribution has been to introduce a method for making a universal and content aware annotator. The HCT is a new concept for annotators, and we only explored a limited set of tools that could be created based on the information it gives. Future work may include development of additional tools that use an HCT in annotation. The algorithm for generating the HCT we developed has been fine tuned for academic research papers. While we believe that the notion of an HCT can generalize effectively to all documents (including book pages, web pages, forms, code print outs, etc.) adjustments to our algorithm would be necessary. It may be true that it is best to develop an HCT generating algorithm for each different document type, and then develop a heuristic to determine which algorithm to use against arbitrary input.

BIBLIOGRAPHY

1. *Evernote*, Evernote Corporation, 2008.

2. *ImagePrinter*, Code-Industry, 2007-2009.

3. *Luminotes*, luminotes.com, 2008.

4. *Microsoft Office OneNote 2007*, Microsoft Corporation, 2006.

5. *Microsoft Office Word 2007*, Microsoft Corporation, 2006.

6. *Zoho NoteBook*, zoho.com, 2009.

7. Bagley, S., and Kopec, G., "Editing Images of Text," In *Communications of the ACM*, 1994, pp. 63-72.

8. Breuel, T., Janssen, W., Popat, K., and Baird, H., "Paper to PDA," In *Proceedings of the 16th International Conference on Pattern Recognition* 2002, pp. 10476.

9. Janssen, W., and Popat, K., "UpLib: a Universal Personal Digital Library System," In *Proceedings of the 2003 ACM Symposium on Document Engineering* 2003, pp. 347-349.

10. Nagy, G., and Seth, S., "Hierarchical Representation of Optically Scanned Documents," In *Proceedings of the 7th International Conference on Pattern Recognition*, 1984, pp.347-349.

11. Olsen, D., "Building Interactive Systems: Principles for Human-Computer Interaction," In *Course Technology*, 2010.

12. Olsen, D., Taufer, T., and Fails,J., “ScreenCrayons: Annotating Anything,” In *Proceedings of the 17th Annual ACM Symposium on User Interface Software and Technology*, 2004, pp. 165-174.

13. Saund, E., Fleet, D., Larner, D., and Mahoney, J., “Perceptually-Supported Image Editing of Text and Graphics,” In *Proceedings of the 16th Annual ACM Symposium on User Interface Software and Technology*, 2003, pp. 183-192.

14. Schilit, B., Price, M., and Golovchinsky, G., “Digital Library Information Appliances,” In *Proceedings of the Third ACM Conference on Digital Libraries*, 1998, pp. 217-226.