IMPROVING USABILITY OF PEDAGOGICAL COMPUTER EMULATION INTERFACES

By

Stephen D. Williams (sdw@lig.net)

Submitted to the

Faculty of the College of Arts and Sciences

of American University

in Partial Fulfillment of

the Requirements for the Degree

of Master of Science

In

Computer Science

Chair: ______________________
Michael Black, PhD

____
Mohammad Owrang Ojaboni, PhD

____
Chris Powell, DM

____
Dean of the College

____
Date

2013
American University
Washington, D.C. 20016

IMPROVING USABILITY OF PEDAGOGICAL COMPUTER EMULATION INTERFACES

By

Stephen D. Williams

Submitted to the

Faculty of the College of Arts and Sciences

of American University

in Partial Fulfillment of

the Requirements for the Degree

of Master of Science

In

Computer Science

Chair: [illegible] 8/6/13

Michael Black, PhD

[illegible] 8/6/13

Mohammad Owrang Ojaboni, PhD

[illegible] 8/6/13

Chris Powell, DM

[illegible]

Dean of the College

August 7, 2013

Date

2013
American University
Washington, D.C. 20016

IMPROVING USABILITY OF PEDAGOGICAL COMPUTER EMULATION INTERFACES

by
Stephen D. Williams (sdw@lig.net)

ABSTRACT

Computer emulations, simulating real or imagined computer systems, are a valuable tool to quickly gain understanding of computer architecture and software. Existing computer emulation systems offer useful but limited visualization and interaction. This paper addresses improving usability of pedagogical computer emulator interfaces with the application of published design principles informed by research into visuospatial ability. The results include a survey of promising techniques addressing similar problems and suggestions for application. Along with supporting work extending a publicly available Java-based PC emulator to enable use of the popular Processing visualization development environment, this provides a well-developed design and implementation framework for future improvements by interested parties.

ACKNOWLEDGEMENTS

I want to thank my committee for their time, diligence, and attention. I want to particularly thank Dr. Michael Black for attention, patience, flexibility, and enthusiasm for this subject. I appreciate the opportunity to be involved in an interesting project that promises to develop into an effective, useful, and fun asset to computer architecture, computer science, and human computer interaction education and practice.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1 Purpose

The purpose of this effort is to enable widespread and informed participation in the improvement of the Emumaker 86 and other computer emulators, both to provide a better tool for users and also to efficiently enable worthwhile exploration.

In this document, a design, visuospatial, visualization, and functionally informed critique of existing pedagogical computer architecture emulation systems is described. This critique creates the context for developing a broad requirements definition, anticipating powerful but currently unavailable visualization and interaction capabilities. A variety of potentially relevant and highly successful design & visualization techniques and technologies are described and characterized relative to this problem. Next, a series of well known and newly derived design principles are explored relative to this problem, informed by these visualization techniques and novel derivations. A number of interactive visualization solutions are described to fit these constraints. In support of creating an open ecosystem of rapidly evolving visualization, the transformation of the EmuMaker 86 simulator from a directly coded Java Swing API to a set of plugin Processing sketches(Reas and Fry, 2007) is detailed. Finally, next steps in both design and implementation of visualizations and evolution of the EmuMaker 86 simulator are discussed.

This work is related to the continued development of the EmuMaker 86 simulator, described as follows, from the NSF proposal:

> “ This proposal describes our plan to develop a web based, graphical microprocessor simulator tool to teach modern computer architecture to undergraduate students. Our simulator will allow students to view and experiment with the flow of data through a multicore x86 processor pipeline without needing to rewrite simulator source code. If granted, we plan to use this funding to develop this simulator and write laboratory exercises for its use in computer architecture courses.
>
> Computer architecture, as currently taught at the undergraduate level, presents a highly simplified and often outdated model of computers. More current processor models, such as multicore and superscalar, are difficult for students to visualize. Research simulators that accurately model processor architectures are difficult to use, require edits to source code, and are poorly documented. These tools, impractical for an undergraduate course, create a barrier for students who wish to learn about modern processors.
>
> We propose a new processor simulator intended specifically for undergraduate education. Unlike existing pedagogical tools, this simulator will cycle accurately model a full 32 bit x86 system, including processor and memory microarchitecture. Modern architectural concepts, such as multicore, a superscalar pipeline, and branch prediction will be included. The simulator will also model input/output and storage devices so that it can execute a simple real-world operating system, such as Minix.
>
> Our simulator will include a graphical user interface visually depicting microarchitecture components and the flow of data. It will model the architecture at varying levels of abstraction, allowing students to work at their comfort level. It will also allow students to use the graphical interface to make modifications to the microarchitecture.” – Black (2011, 1)

Visualization in computer science education has long been employed and has been evolving in various ways to assist instruction and learning.(Fouh et al., 2012) Computer architecture simulators provide opportunities not only for understanding computer organization, operation, and architecture, but also for thorough understanding of software algorithms and characteristics.

Building on the implementation of that system, this work identifies usability requirements and explores gaps with what exists to enable identification of rational constraints and design elements that characterize the most promising solutions.

1.2 Research Questions

The key research questions addressed here were arrived at by first considering the high-level goal of improving usability of a computer architecture simulator for a series of specific tasks, including:

1. Uses of the simulator in the classroom for:
    (a) For operating systems, computer organization, and computer architecture classes: Running created operating systems.
    (b) For computer organization: Assist in the process of learning assembly language.
    (c) For computer organization: Assist in the teaching of simple digital logic.
    (d) For advanced computer architecture class: illustrating pipelining and computer architecture.
2. Uses of the simulator for software and systems development:
    (a) Understanding application, operating system, and device driver boundaries and interaction.
    (b) Understanding algorithms in concrete terms, including instruction and memory usage.
    (c) Finding performance bottlenecks.

Then, data was gathered and analyzed about difficulties experienced by computer science students using the EmuMaker 86 simulator for these purposes. Background for understanding perceptual limitations was gathered to inform the analysis of this data. Existing solutions to similar problems and applicable system design principles were used as a basis for synthesizing solutions. These results are closely related to and supported by Shaffer et al. (2013). The following summarizes the research questions and the results obtained:

1. What are the problems, gaps, and limitations that users could be expected to experience when using an existing computer emulator to understand machine instructions, registers, flags, processor state, memory and memory use patterns, I/O, and specific implementation details of software and computer architecture?

    (a) *Answer:* Student feedback has shown with existing emulators, users are only able to observe and understand a handful of instructions, system changes, and memory modifications at a time. There is no sense of flow, temporal variance, or large granularity organization of software. Models of datapath definition of processor architectures quickly become difficult to see in their entirety or to reason holistically about connectivity.

2. While exploring the integration of the most popular interactive visualization environment into the research emulator, what conclusions were reached that informs and enables future development?

    (a) *Answer:* During this effort, it was found that emulators tend to be laid out in a common architectural pattern and that the need to add flexible visualization hooks is common to all of these emulators, if they even have visualization of a particular area. The ability to see the value of registers is common to every emulator pedagogical and debugging emulator. The value of a particular memory location is not available on every emulator, e.g. DosBox; this requires linear memory model emulation rather than just compatible system calls. Visualizing microcode can only be done in those emulators that model microcode at all. Even for those that do, there may not be consistency in how this is done. Many just have a large if / switch statement for the instruction implementation rather than instruction decoding to a microcode queue.

3. According to the literature, what human perceptual and cognitive limitations do users tend to have and how does that affect the selection of solutions for computer architecture simulator user interfaces?

    (a) *Answer:* The human perceptual system has a number of strengths along with weaknesses such as limited working memory, alternate mental model and reasoning systems that introduce systematic errors, body-space related metaphorical reasoning about spatial, relational, connectedness, and controllable aspects of both the real world and conceptual and virtual worlds.

4. What existing interactive visualization methods work well for somewhat problems analogous to simulator CPU, memory, and other simulator interfaces and what principles can be derived from those methods?

    (a) *Answer:* Visualizations utilizing icons, flow graphs, space filling proportions, dynamic maps of color, visual vibration, motion, and field discontinuity are some of the key solutions for allowing highlight or pattern recognition of important information when size, complexity, and multivariate nature causes unbounded complexity.

5. What published visualization methods and design principles can be rationally applied to select potential solutions for this problem domain?

    (a) *Answer:* A number of visualization methods have solved certain constraints with improvement over then existing methods. These include sparklines, Sankey Diagrams, Choropleth Maps, Treemaps, Table Lens, and Layer-Time graphs. Many design principles can provide insight, constraint, and sources of innovative solutions. Some key principles include using, manipulating, or sensing "closure", consistency, contour bias, desire lines, figure-ground relationships, Fitts's Law, Forgiveness, Hick's Law, proximity, wayfinding, and others.(Lidwell et al., 2003)

6. What are some new solutions for improving visualization?

   (a) *Answer:* Several specific visualization solutions were designed to satisfy these constraints, and several were implemented.

1.3 Significance of the Study

This study is important because it focuses on diagnosing and improving on what is, after correctness, the most important aspect of computer architecture simulators: The user experience, including the effectiveness and quality of the emulator experience. The effectiveness of an emulator is limited by both a lack of features and a user interface that prevents or doesn't actively support the full range of actions a user may want to perform. By referencing key background material, performing analysis of emulation systems and user requirements, and synthesizing informed solutions, this work advances emulator design and implementation.

This report, and the collected sources used, should also be useful as a template for analysis of other, unrelated problems.

1.4 Concepts and Background

A computer emulator duplicates the function of one computer system on another.(Wikipedia, 2013b) A pedagogical or debugging computer emulator, which will be referred to here as a computer architecture simulator (CAS)[1], provides exposed simulation of the internal operations of a computer system, often allowing various

[1] "Computer simulator" is generally understood to describe a computer simulating anything, while "computer emulator" is generally taken to mean a computer simulating another computer.

views and forms of direct manipulation. A CAS allows direct observation of the logical mechanics of operation of one or more aspects of a computer system. By internalizing an effective, representative mental model as a framework for further understanding, the user of an emulator may gain a more rapid and thorough understanding of many important computing concepts and their relationships. The degree of understanding both in breadth of experience and qualitative development of accurate intuition can enable rapid evolution of an explorer.

Unfortunately, this potential is only weakly realized to date. Existing computer emulators provide minimal, literal visualization and very low-level emulator and visualization controls. A small number of instructions, memory locations, and machine state changes can be observed while a very small set of instructions executes. Operations at the very small are difficult to understand in the context of an overall system. Multi-scale aspects of systems are difficult to discern. Relationships between applications, libraries, operating systems, device drivers, memory, and I/O are unclear. Additionally, adding interactive visualization to a CAS system is generally a difficult, involved task.

Numerous advances in visualization methods, libraries, and subsystems, efficient “big data” analysis, and new developments in human visuospatial performance characteristics provide a basis for addressing these weaknesses. This gap between what is conceived, implemented, and easily implementable, and what is clearly possible for interactive visualization in enhanced CAS systems is the central problem set addressed.

1.4.1 Usability

Usability is a measure of how pleasant and easy an interactive system is to use to achieve some useful goal. Here, we will consider pleasant and easy with respect to goals that the user of a system may have and characterize the net benefit of a system as a measure of efficiency. A difficult and annoying system discourages and otherwise minimizes or thwarts continued use by users. According to Cockton (2013), the following are propositions about usability evaluation that describe the ideal the Human-Computer Interaction field aspires to:

> Things arent this simple at all though, but lets start by considering the following propositions about usability evaluation:
>
> 1. Usability is an inherent measurable property of all interactive digital technologies.
> 2. Human-Computer Interaction researchers and Interaction Design professionals have developed evaluation methods that determine whether or not an interactive system or device is usable.
> 3. Where a system or device is usable, usability evaluation methods also determine the extent of its usability, through the use of robust, objective and reliable metrics.
> 4. Evaluation methods and metrics are thoroughly documented in the Human-Computer Interaction research and practitioner literature. People wishing to develop expertise in usability measurement and evaluation can read about these methods, learn how to apply them, and become proficient in determining whether or not an interactive system or device is usable, and if so, to what extent.

While these ideals may never be fully realized, significant progress has been made, allowing rational comparative evaluation of methods and systems. The usability of a system can be measured directly and indirectly and through formal and informal methods. Often, there are specific metrics for performance of a task or measurement of knowledge gained that can be related to alternative approaches.

1.4.2 Related Disciplines

The problems and solutions for CAS interactive visualization potentially involve these disciplines:

1. Computer Engineering - Understanding how computers and computer elements are designed, built, and function.

2. Computer Science in general - Understanding computer science principles.
3. Software Engineering - Supporting good software engineering practices, methods, and continuous improvement through better understanding of concrete results.
4. Computational Linguistics - Developing a more firm grounding in a bottom to top understanding of function and layering.
5. Pedagogy - Using iterative, rich, and optimally responsive learning and teaching techniques.
6. Social Sciences - Considering the social dynamics of effective learning, development, and professions.
7. Graphics & Graphics Arts - Designing and creating iterative and useful graphics.
8. Cognition Sciences - Using our current understanding of cognition to improve methods.
9. Machine Learning / Artificial Intelligence - Gaining automated assistance in understanding, interpretation, and interactive intent and feedback.
10. Human-Computer Interaction - Using our current knowledge of what works and promising avenues to improve our interaction with computers.
11. Interaction Design - Specific study of the dynamics and characteristics of interaction with systems.
12. Neuro-Psychology - Taking into account our current knowledge of fundamental and developmental mental abilities, strengths, tendencies, and weaknesses.

Attention to these aspects of problem solving design can help lead to more useful and optimal results.

1.4.3 Visuospatial Thinking & Mental Models

Visuospatial thinking research is an exploration of known knowledge, spatial, perception, internal representation inference, and the resulting limitations, and characteristics of the human thought. Many experiments have incrementally added to knowledge about how people apparently learn, represent, and reason about knowledge. These insights are characterized by commonality and variance. Working conclusions, continually challenged by new theories and experiments, have become useful at some levels of design and problem solving involving human interaction. In many cases, people may have no conceptual framework for making design choices, or they may have their own model of how human perception and memory work. By considering current research results, designers can create and validate their working assumptions to produce more effective designs.

The following are some conclusions that are likely to be salient in designing rich user interfaces for computer architecture simulators and other similarly difficult to solve design problems.

Cognitive Mental Spaces

People use more than one kind of mental space depending on the function they serve.(Shah and Miyaka, 2005, 1) There are at least four well-studied spaces. These include:

1. The space of the body, involving prioperception and action, divided by body parts. This space is experienced volumetrically, mainly as a series of connected parts and the space around them.(Shah and Miyaka, 2005, 2) It also intrinsically models what is and is not possible in terms of movement, often constraining understanding of observations.(Shah and Miyaka, 2005, 3)

2. The space around the body for perception and action. It is primarily divided into

front/back, head/feet, left/right. Reasoning is often relative to these general vectors and related rotations. Head/feet, and therefore up/down and high/low, are asymmetric and have more perceived stability while the other axii are mostly symmetric and equivalent.

3. The space of navigation, constructed in memory from different types of sources, normally experienced as a plane. The collage of sources causes systematic errors. This models the space of potential travel, which is often too large to observe or consider at once.

4. The space of external representations, such as pictures, maps, charts, and diagrams, which act as aids to memory and cognition. These representations often schematize[2] and distort information. These external representations indirectly represent some other space, and are then represented internally in ways that depend on how they are perceived.

Furthermore, attention tends to be focused on foreground objects: "For human cognition, the void of space is treated as background, and the things in space as foreground. They are located in space with respect to a reference frame or reference objects that vary with the role of the space in thought or behavior."(Shah and Miyaka, 2005, 1) Careful consideration of foreground vs. background is needed to avoid conflicting with natural assumptions.

These spaces are not exact representations, but rather are selective models, evolved for particular purposes.(Shah and Miyaka, 2005, 2) Depending on context, habit, input form, and even explicit suggestion, a person can switch representations and interpretation, often substantially changing performance.(Shah and Miyaka, 2005, 9) These different spaces allow us to reason about our bodies in an envi-

[2]Schematize: to reduce to or arrange according to a scheme.(Random House, 2005)

ronment with obstacles while accomplishing goals through possible actions. When work to understand and interact with a virtual environment, such as that created in a computer, we're often operating metaphorically so as to utilize these mechanisms.

Extension of the Mind and Body Through Tools

At a certain point of proficiency and neuromotor integration, or the internalized model equivalent, a tool becomes an extension of the body, and therefore the prioperceptive mind. With experience, we adapt to the spatial presence, limitations of movement, capabilities, and sensory feedback of a car to the extent that we have little explicit perception between the desire and action of making something happen. We become, to some extent, one with the vehicle. While this type of internal/external modeling and extension happens frequently, it often is more subtle. A good way to experience this gap acutely is to take begin aircraft pilot training: Adding control of altitude along with a new interface, new limbs with new rules essentially, and new types of boundaries and simultaneous cognitive demands rapid cognitive growth. Good examples of tool use, beyond actual tools, include musical instruments: "leads to the exploration of designed artifacts as extensions of human cognition as scaffolding onto which we delegate parts of our cognitive processes. ... it is possible to describe the digital instrument as an epistemic tool: a designed tool with such a high degree of symbolic pertinence that it becomes a system of knowledge and thinking in its own terms."(Magnusson, 2009)

The same extension of the body and addition of new senses and limbs occurs with coherent, well thought out interactive visualization and representational models. 3D game environments have experimented with a multitude of solutions here with some being particularly effective:

> "Black and White 2 (along with its predecessor, Black and White (2001)) is a much more (literally) hands-on game. The Black and White games turn

> a standard PC cursor into a hand with an extraordinary range of capabilities and even expressions, which affects all kinds of interactions. Black and White embodies the interaction with its super-cursor-hand, such that to move a unit of troops, the player literally picks up the collection of soldiers in her or his divine hand and drops (or throws) them at the desired location. The interaction is strangely diegetic, because the hand is no longer merely a cursor, but a special kind of avatar in the world. This fundamentally changes the relationships among the player, the avatar, and in-world content. The meanings of the interactions between player and in-world objects also change; often they take on a degree of humor not seen in other strategy games..." (Olli et al., 2008, 205)

This depiction of a hand, with very hand- and arm-like interaction, albeit at a god-like level, are almost immediately understood and used by users:

""Such behaviors, of course, lead to Black and Whites central mechanic, which is the moral feedback the game provides, based on the ways the player, through the hand avatar, interacts with her or his people."(Olli et al., 2008, 205)."

The human mind can expand and extend the working model of limbs, segments, joints, and capabilities. When properly trained, this allows users to accomplish tasks as automatically and comfortably as throwing a ball or steering a vehicle. A key design goal for many systems should be to replace the use of menus and other artificial interaction with mechanisms that can be experienced as an extension of the user's body model.

Mental Models

A mental model is some representation of the world that is used to predict and interact. There are a number of views of the different natures of this, including the logical, syllogistic model(Johnson-Laird, 1983, 64) of propositions and models that variously model the spatial and temporal aspects of the real world.

> Mental models owe their origin to the evolution of perceptual ability in organisms ... David Marr ... outlined a computational theory of vision that largely accounts for the derivation of the perceptually based models of the world. The theory postulates three principal forms of representation: first, the 'primal

> sketch', which is a symbolic representation of the disposition ... local geometry, and the structure ...; second, the '2.5-D sketch', which is a viewer-centred representation fo the depth and orientation of surfaces, including contours and discontinuities; and third, the three-dimensional model of an object, which is based on an object-centred set of coordinates and primitives that make the space-filling shape of the object explicit. The account of 3-D models applies only to certain classes of objects, but it is clear that the basic ideas should be extensible to all objects and to scenes in which there are spatial relations between objects. ... It is therefore safe to assume that a primary source of mental models – three-dimensional kinematic models of the world – is perception.(Johnson-Laird, 1983, 64)

It is clear that people employ a mosaic of mental models to understand, reason about, and react to systems. When creating solutions, a designer should give explicit thought to the form and structure of information so as to enable the most appropriate mental model. In some cases, that will be logical constraints, rules, propositions, and definition of logic of a system. In more advanced situations, this can be thought of as an 'algebra' or 'calculus' of some kind, metaphorically consistent with algebra or calculus.

A good example of this is Allen's Interval Algebra(Allen, 1983), which defines thirteen basic relations between time intervals that are "distinct, exhaustive, and qualitative".(Alspaugh, 2013) These relations are: precedes, meets, overlaps, finished by, contains, starts, equals, started by, during, finishes, overlapped by, met by, and preceded by. Note that this algebra can be interpreted spatially, temporally, and logically or symbolically.

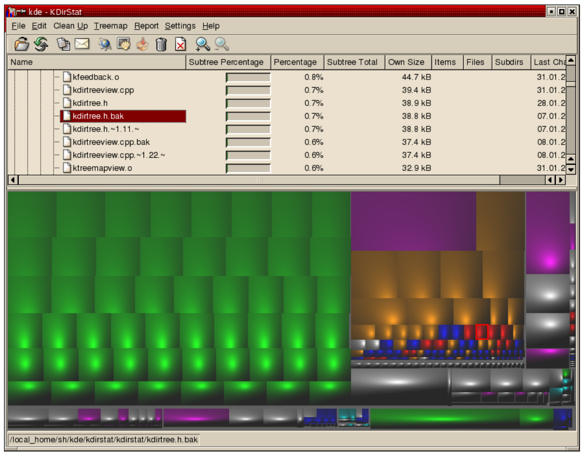
1.4.4 Successful Interactive Visualization Methods

These visualization methods elegantly solve key information density and representation problems in ways that are generally applicable to a wide range of situations. Each has a certain range of greatest success and each has limitations and problems. Considering when they are useful and why they are limited provides both building blocks and the basis for surpassing those limitations for particular problem domains.

Treemap

Treemaps provide the ability to directly represent hierarchy and at least two key scalar or categorical attributes about many items in a single graphical, navigable and explorable display. For example, direct access to more than 100,000 directories and files in a file system is not unusual, with large files being very apparent.

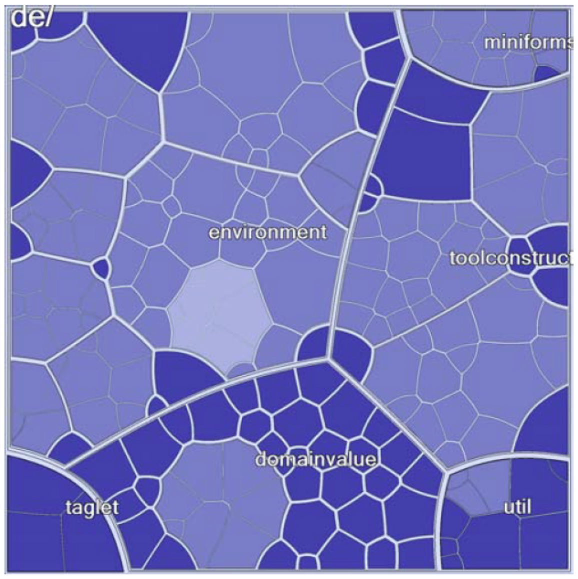
Figure 1.1. Kdirstat treemap



Name	Subtree Percentage	Percentage	Subtree Total	Own Size	Items	Files	Subdirs	Last Cha
kfeedback.o		0.8%		44.7 kB				31.01.2
kdirtreeview.cpp		0.7%		39.4 kB				31.01.2
kdirtree.h		0.7%		38.9 kB				28.01.2
kdirtree.h.bak		0.7%		38.8 kB				07.01.2
kdirtree.h.~1.11.~		0.7%		38.8 kB				07.01.2
kdirtreeview.cpp.bak		0.6%		37.4 kB				08.01.2
kdirtreeview.cpp.~1.22.~		0.6%		37.4 kB				08.01.2
ktreemapview.o		0.6%		32.9 kB				31.01.2

/local_home/sh/kde/kdirstat/kdirstat/kdirtree.h.bak

Figure 1.2. Voroni Treemap



(Balzer et al., 2005)

Table Lens

A table lens is a technique where a two dimensional table uses variable width columns and rows to highlight certain cells and compress other cells while keeping fields in context.

Bubble Chart

Bubble charts fill available space with various sized bubbles, clustered together, usually using a pseudo-phsyics collision algorithm.

Figure 1.3. NY Times Budget Proposal bubble chart, showing stratified grouping.

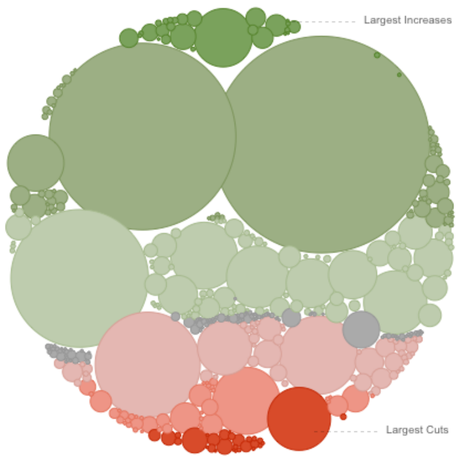


Figure 1.4. Bubble Chart Showing Group Membership and Relative Size.

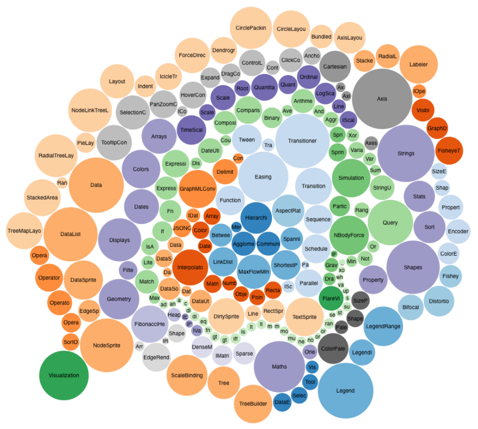
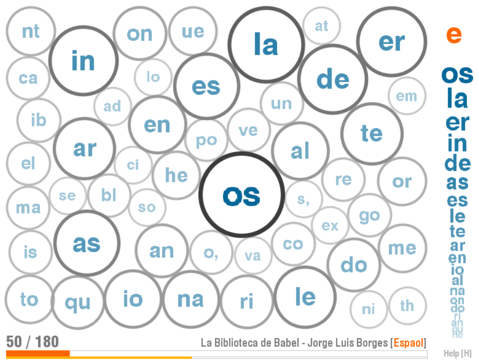


Figure 1.5. Bubble Chart Letter-Pair Analysis of Document.

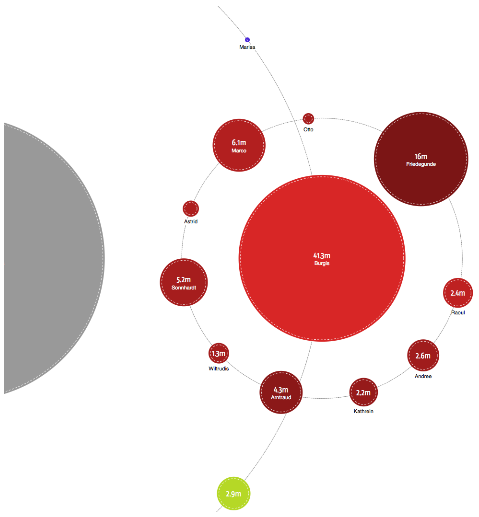


(Bereciatua, 2005)

1.4.5 Radial Bubble Tree

A radial bubble tree supports a local view into data that may be highly complex showing variables like type and relative magnitude. The bubble tree supports navigating up or down in a hierarchy.

Figure 1.6. Minimal BubbleTree Demo.

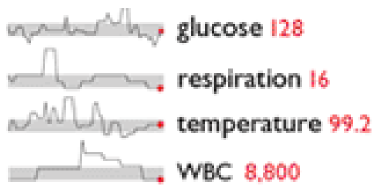


(Aisch, 2011)

Sparkline

A Sparkline provides an understandable graph, usually a line chart, in the size and placement of about a word or two. It is a very effective way to embed or aggregate knowledge about the nature of a trend or the position of a spike.

Figure 1.7. Edward Tufte's Sparklines.



Sankey Diagram

A Sankey Diagram shows proportional flows with multiple sources and sinks through multiple phases, including feedback loops and closed or partially closed systems. The direct proportions, ease in labeling and splitting streams proportionally can be very clear and concise. Although named after Irish engineer M. H. P. R. Sankey, this type of diagram was created by C.J. Minard who said about his creation: "The aim of my carte figurative is less to express statistical results, better done by numbers, than to convey promptly to the eye the relation not given quickly by numbers requiring mental calculation."(Riehmann et al., 2005) These diagrams can be interactive, allowing zooming in and filtering for relevant data. A key feature is the ability to do flow tracing by selecting an edge or node in a data graph which can highlight contributing flows.

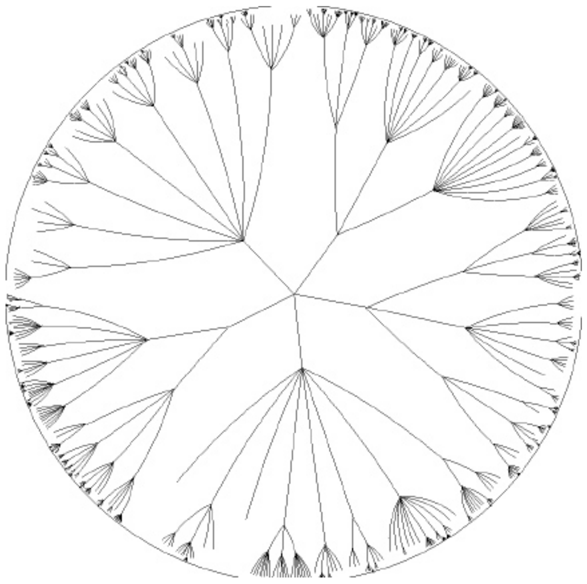
Hyperbolic Tree

A hyperbolic tree is one of several methods that map a uniform planar space to a non-uniform, focused view somewhat like a fisheye lens. This is shown as circular view of a tree mapped onto a hyperbolic plane. Elements gradually grow smaller as the view is centered away from them.(Lamping et al., 2013)

Choropleth Map

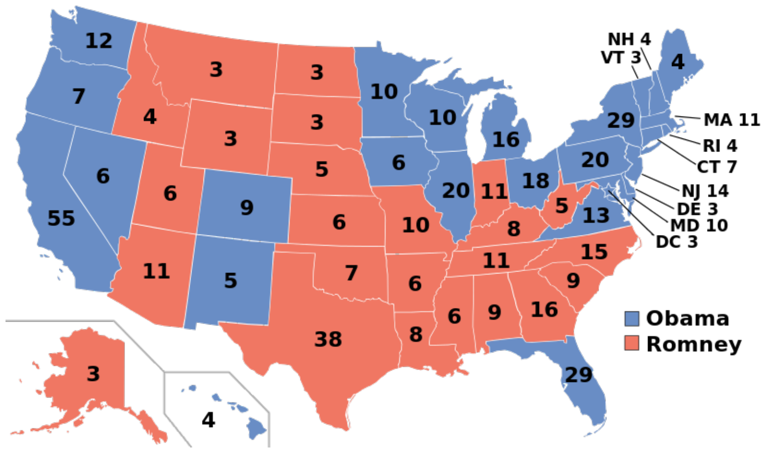
Choropleth maps are extremely common, frequently being used at elections and for illustrating census and survey data. "A choropleth map is a thematic map

Figure 1.8. Hyperbolic Tree



in which areas are shaded or patterned in proportion to the measurement of the statistical variable being displayed on the map, such as population density or per-capita income."(Wikipedia, 2008)

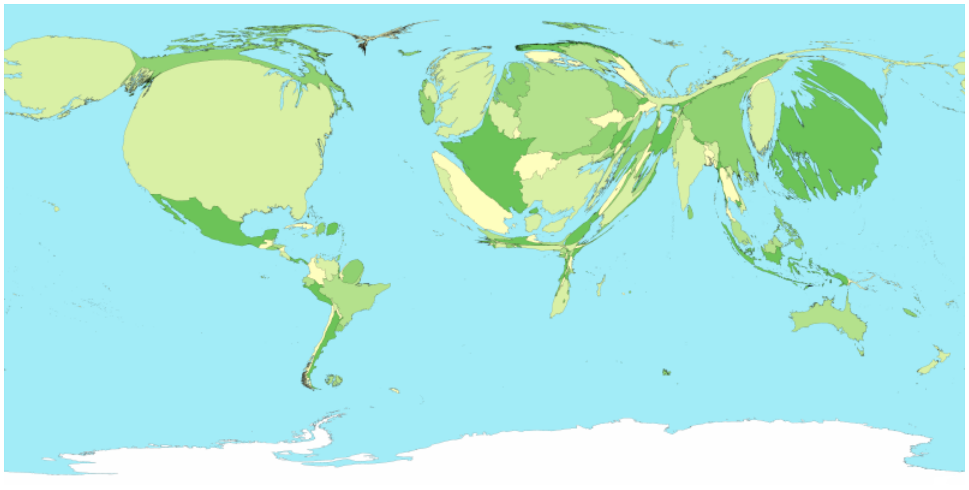
Figure 1.9. Choropleth Map



Cartogram

A cartogram transforms a standard map of some kind by adjusting the size of each identified area according to some variable. The resulting map retains very similar relative positioning constraints among elements, but can be highly distorted by the variable being illustrated. The following diagram displays GDP of countries. Election results

Figure 1.10. Cartogram



Expand-Ahead

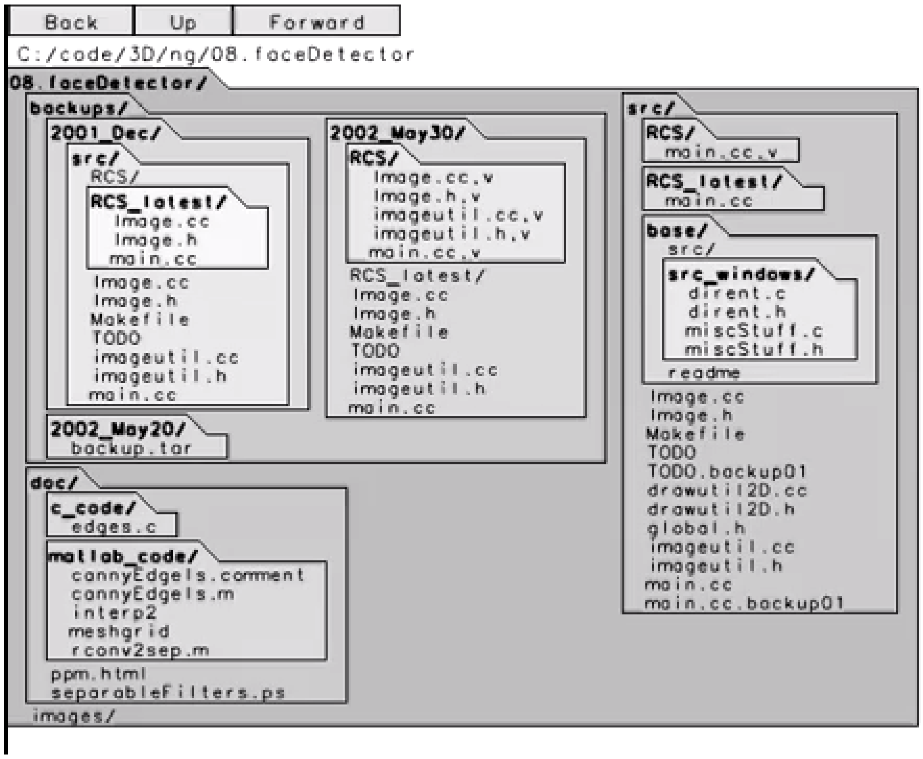
Expand-Ahead(McGuffin et al., 2009) uses unused whitespace on screen to expand certain levels of a hierarchy to show as much information as possible while maintaining context.

The animation scrubbing interactive visualization, attained by holding a control key down while moving the mouse, is a useful paradigm for replay and pause of animations.

Layer-Time

The layer-oriented time-series data(Lopez-Hernandez et al., 2010) (layer-time) iterative visualization method applies a powerful but rarely used method of simultaneously displaying many temporal instances while allowing the user to highlight specific cases easily. As the user moves the pointer around, a single instance is clearly

Figure 1.11. Expand-Ahead

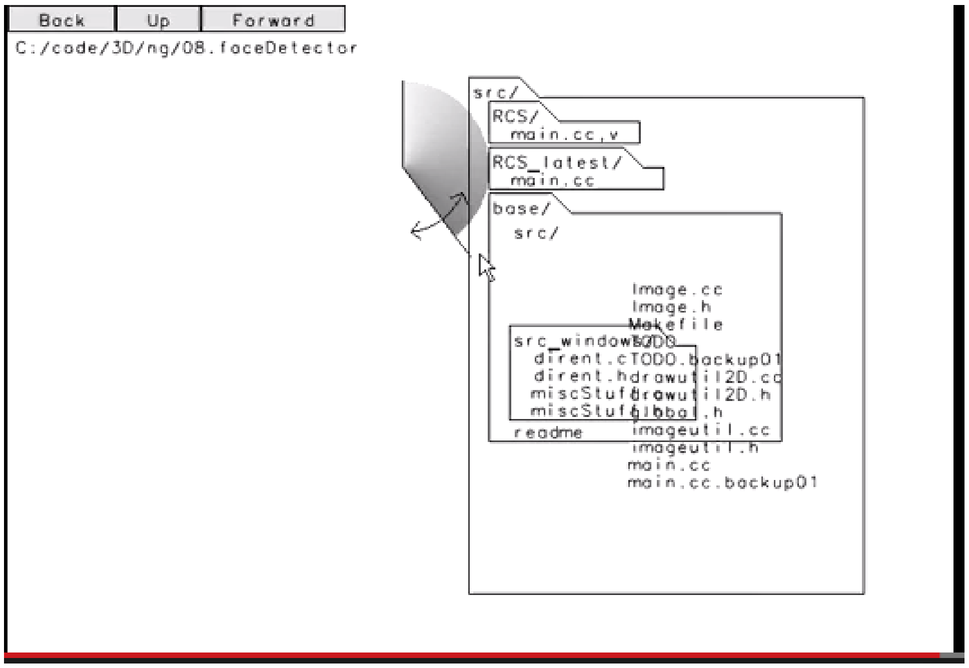


highlighted while the background shows the aggregate paths of all passes.

MindMap

Mindmap software, such as FreeMind(Assorted, 2013b), implements interactive editing of a hierarchical outline of arbitrary information, automatically laid out in a two dimensional pattern. Mindmaps, a type of spider diagram, are meant to directly map to a user's schematic mental model for information. Modern mindmap software allows direct manipulation and editing of nodes in the map. This representation of knowledge, direct manipulation, and collapsible nature are all potential features of an information tracking subsystem in a CAS.

Figure 1.12. Expand-Ahead Animation Scrubbing



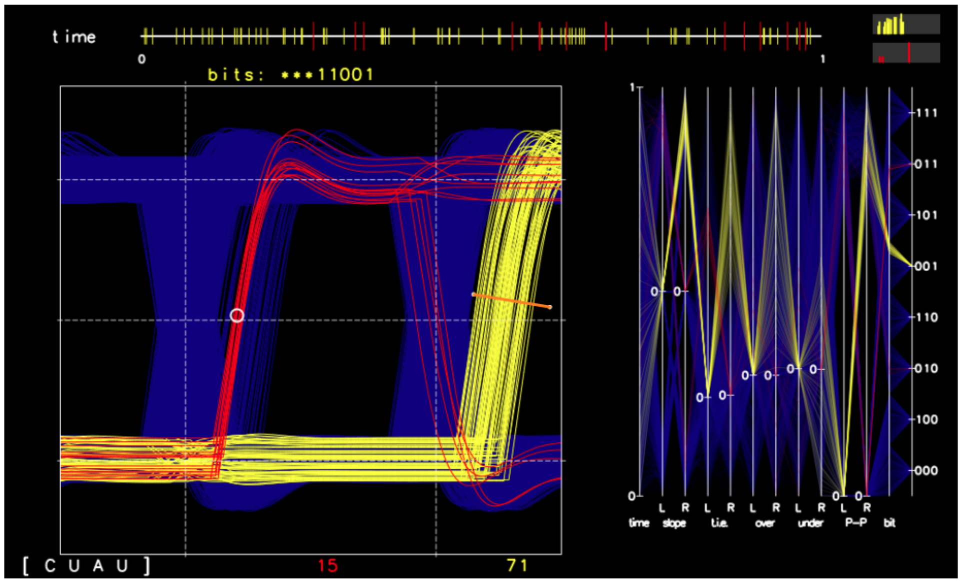
Kanji

Kanji, the Japanese and Chinese pictograph based writing system, is built from a small number of radicals, arranged in a sometimes meaningful pattern. Each radical has an original meaning as a word along with a number of usually related sense that it can be used as part of a Kanji character. For instance, a radical that is a stylized bow is used as part of the word for ’pull’ and contributes a ’pull’ sense to other words. Kanji are usually taught by wrote learning. Japanese and Chinese writing systems are very difficult to learn because Kanji are irregular, based mostly on historical evolution rather than careful design. However, Kanji can be more concise than an equivalent alphabetical writing system.

Emoji

Icons for types of people, places, objects, or actions, used as an abbreviated language to communicate on Asian mobile networks. They can be used for small,

Figure 1.13. Layer-Time visualization



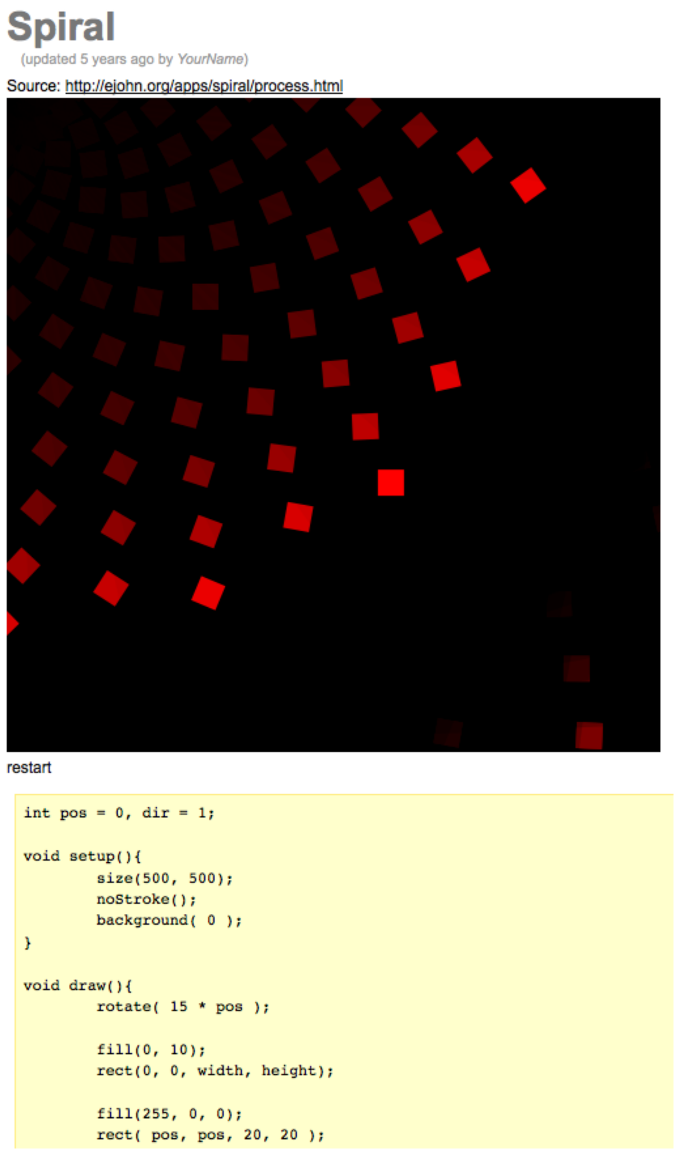
highly dense messages or for whole conversations.

TiddlyWiki & TiddlyProcessing

TiddlyWiki is a one-page-application, written in Javascript and running in a web browser. This application implements something like a full Wiki contained within that one page. Newly created nodes, 'tiddlers', are saved to memory and the whole page is saved back into the HTML file that was originally loaded. The user can open one or more tiddlers, shown consecutively on the web page. Everything is searchable and the page actively manages certain data. This is meant for personal note or record taking, editing and then publishing a web page, and similar capabilities. This system has potential use as a knowledge / data capture, sharing, and note taking mechanism.

TiddlyProcessing(Assorted, 2013d) (Assorted, 2013c) combines TiddlyWiki with ProcessingJs, the Javascript implementation of the Processing environment. This combination allows a user to create, edit, or run multiple processing sketches on a single web page. This is both an easy way to become productive quickly, and an easy way to share and explain related concepts. This is useful for prototyping or demonstrating visualization methods and ideas. It could also be used as an active and interactive snapshot, program trace, or statistical summary of a segment of operation of an emulator. In 1.14, both the running sketch and the source code are visible at once. The user can edit and immediately see the result. Multiple sketches can be open and in edit or run on the same page.

Figure 1.14. TiddlyProcessing


Spiral
(updated 5 years ago by YourName)
Source: http://ejohn.org/apps/spiral/process.html
restart

```
int pos = 0, dir = 1;

void setup(){
        size(500, 500);
        noStroke();
        background( 0 );
}

void draw(){
        rotate( 15 * pos );

        fill(0, 10);
        rect(0, 0, width, height);

        fill(255, 0, 0);
        rect( pos, pos, 20, 20 );
```

Limitations

Each of these visualizations is a useful improvement, but with limited application. Others have noted the limitations of these techniques:

> Attack graphs are usually displayed as node-link graphs. An excellent review of 15 different general approaches to displaying node-link graphs is available... We explored many of these ... Hand-drawn graphs are too time-consuming. The Graphviz dot tool(Graphviz, 2013) and force-directed approaches lead to excessively complex displays unrelated to the underlying network structure. Techniques to expand and collapse parts of large graphs such as space trees(Plaisant et al., 2002) and hyperbolic trees(Lamping et al., 2013) cause global context to be lost when part of a network is expended and, as a result, are difficult to follow. Treemaps(Johnson and Schneiderman, 1991) are excellent when summarizing data for a small set of hosts, but they do not represent a network's hierarchical structure well. Finally, multilevel cell matrices are difficult for system administrators to interpret and relate to actual networks.(Goodall et al., 2007)

Successful, optimal interfaces may combine elements and ideas from more than one of these explorations to more optimally address a particular problem. This recombination can be difficult to see, but through using these different visualization methods, an algebra of design elements can be drived.

1.4.6 Node-Link Graphs

In addition to value quantification and classification, grouping and matrix / field pattern recognition, and iconography, node-link graph structures provide a rich mechanism to illustrate relationships, membership, temporal flow, and dependency. Many types of visualization of graphs have been created to solve the representation and interpretation needs for various applications. A wide-ranging and somewhat comprehensive examination of node-link graph visualization methods and constraints can be found in Munzner (2006a). These methods include:

1. Visual Channels - separable vs. integral, position or attribute.

2. Traditional Graphs
   (a) Visual External Representation - common node/link graph, offloads working memory and cognition to visual system.
   (b) Hand-drawn - high density, takes very long to create.
   (c) Dot - automatically computed, low information density, tends to be difficult to read labels.
   (d) Graph Layout Criteria - minimize crossings, area, curves, maximize angular resolution and symmetry.
   (e) Force-Directed Placement - magnet/spring simulation, doesn't scale.
   (f) TopoLayout - detect topological features and layout by data or cluster type.
   (g) Multilevel Hierarchies - scalable, not useful if features can't be detected.
   (h) Animated Radial Layouts - dynamic graphs for dynamic data, minimizes visual changes, shows current state.(Yee et al., 2001)
   (i) Animation - polar interpolation, maintaining neighbor order.
   (j) Constellation - information density / visual salience tradeoff.
   (k) Selective Emphasis - highlight sets, avoid misperception and hidden state.
3. Nontraditional Representations
   (a) Treemaps - showing structure with containment, but not connection; good for extreme values; poor for structure.
   (b) Themescapes - cluster stability, noise for interpretation, spatial analysis of non-visual information.
   (c) Multilevel Call Matrices - link matrix vs. node-link.
4. Focus + Context - single detail view vs. multiple linked windows

(a) SpaceTree - supports expand/contract interaction, animated transitions, only shows a small fraction of detail at once.

(b) 2D Hyperbolic Trees - shows a lot of data with focused information large, other information progressively smaller, but in context.

(c) H3 - 3D fisheye from hyperbolic geometry - shows large amounts of data, but the user can get lost.

(d) TreeJuxtaposer - landmarks visibile, stretch/collapse navigation, difficult for some tasks, expensive.

(e) SequenceJuxtaposer - side-by-side comparison, accordion drawing

Of these, visual channel methods, Dot, force-directed placement, animation, selective emphasis, cushion treemaps, themescapes, and sequencejuxtaposer are most promising for application to a computer architecture simulator.

1.4.7 Taxonomies & Paradigms

Taxonomies provide a framework for understanding or spectrum of distinctive points that highlight fundamentally different alternatives. They can be useful in fully understanding a domain and particular instances and especially helpful in boosting the creative process by directly enabling the consideration of all alternatives. Paradigms illustrate a particular pattern or model of understanding or designing and building something, sometimes enabling particular kinds or effectiveness of action.

Semiotics, Affordance, & Charles Peirce's Theory of Signs

(Atkin, 2013)(Olli et al., 2008, 201) A sign is a combination of a signifier (the image) and a signified(the concept).(Olli et al., 2008, 193) There is value in establishing a consistent, widely or consistently understood shared context of signs.

An affordance is an intrinsic property of an object or a design element that a human perceives to be of potential use.(Olli et al., 2008, 194). A common example is a push or pull door: If there is a handle or bar, the door should be able to be pulled. If there is a push pad, it is clear that you should push the door. It is an error in a sense for a door that must be pushed to have a handle associated with pulling, which is only diminished by clear marking.

Charles Peirce's Theory of Signs evolved to a trichotometric vocabulary of icon, index, and symbol. The icon refers to a sign's similarity to the object it signifies or the object's intended use. An index is when there is a representation of an effect, indicating a particular cause. This involves an additional level of inference, which could be an additional cognitive burden, although it could also reuse an existing association. The symbol is an arbitrary association based on some association. This arbitrary and potentially complex association requires further memory and inference, however it can also be context free and without prior knowledge conflict.

Constraints on Affordances

In Norman (2002), three types of constraints on affordances are described: physical, logical, and cultural. A physical affordance constraint simulates some physical process with limits based on walls, weight, etc. A logical constraint involves rules or some type or combination of deduction and inference. A cultural constraint is related to cultural shared conventions or strong associations due to everyday association. A well-known example is a science demonstration in San Francisco's Exploratorium of a brand-new toilet that was turned into a drinking fountain, challenging museum goers to overcome their cultural bias.

Constraints can help define the purpose of an element, but also prevent errors or irrational operation.

Selfconscious and Unselfconscious Cultures

(Olli et al., 2008, 253)(Alexander, 1979) Creators in an unselfconscious culture follow tradition which dictates that there is a right and wrong way to do things. Nothing related to design is written down, but new builders are actively corrected so that they follow existing practice. This is observed not only in cultures that have long-standing cultural practices, but even, in many ways, in the modern video game industry.

A selfconscious culture engages in academic and industry general rules, principles, and best practices. Education is formalized, general rules allow rapid education, and creativity is enabled rather than constrained.

> “Alexander (ibid.) describes how as a self-conscious design culture develops further, change for its own sake becomes acceptable. Culture changes too rapidly for adaptation to keep up with it and factors sustaining equilibrium drop away. The master craftsman takes over the process of form-making and inventiveness becomes valued as a way of distinguishing craftsmen/artists, leading to the cultural perception of the designer as a star. Specialization underlies the establishment of design academies, and the academies make principles explicit, making them available for criticism and debate. Debate requires justification, leading to the formulation of general theories, principles and rules. Questioning leads to unrest, which leads to formal innovation and further self-consciousness.
>
> Self-conscious design culture is concerned with both the design education of novices and explicit, self-conscious debate among established and experienced designers. One of the distinctions of experienced and expert designers (as with all forms of expertise) is an increasing implicitness of knowledge, with ongoing analytical processes oriented towards making that implicit knowledge more explicit. Hence explicit design knowledge accelerates and facilitates the ongoing development of expertise, but it is always very far from fully representing that expertise.”(Olli et al., 2008, 255)

1.4.8 Elements of Designed Form

These elements are derived from the Elements of Designed Game Form.(Olli et al., 2008, 252) This layered taxonomy helps separate conceptual forms from implementation strategy and embodiment, showing how they are separate but can be used together in various combinations.

1. System Interaction
2. Logical Elements & Semantics
   - (a) Objects
   - (b) Space
   - (c) System
   - (d) Rules
3. Media
   - (a) Infrastructure
   - (b) Technology
   - (c) Interfaces
   - (d) Modes or sequences
   - (e) Skins or symbol styles
   - (f) Lighting and highlighting
   - (g) Audio
   - (h) Video
   - (i) Animations
   - (j) 2D Graphics
   - (k) 3D Graphics

1.4.9 Characteristics of Interaction Cues

(Olli et al., 2008, 195) In considering the nature of a possibly interactive visual element, these measures are one type of categorization that may be useful in considering usability with users. These were developed to evaluate video games, so some adjustment may be necessary.

Table 1.1: Characteristics of Interaction Cues Documented in the Study. Quoted from: (Olli et al., 2008, 195)

Artifact-Centered Characteristics	Human-Centered Characteristics
The time and sequence (order, duration, iteration)	What the human has to do to perceive it as a cue?
Space and setting (location, environment)	What types of audience is the cue aimed at?
Occasion	What knowledge does the cue take for granted?
Medium	What kind of response does the cue expect the player to have?
Physical attributes	What are the consequences for not responding to the cue in desired ways?
Function	
Diegetic vs. non-diegetic nature	

1.4.10 A Taxonomy of Game Cues

(Olli et al., 2008, 195) While a successful CAS innovation may or may not involve gamification or game-like features, the taxonomy taken from Olli et al. (2008) provides an informed summary of techniques in a field which has the most competitive pressure for innovative interfaces.

Interactivity

Are elements interactive or non-interactive?

Markedness

Is an element marked as interactive? Is it clear to a user what is active vs. passive? Are conventions and expectations, possibly intrinsic in some way, followed or violated?

Diegesis

A diegetic form is an element that has an in-environment representation and action, such as an unlocked door allowing access. A non-diegetic form might be control of speed or reset of the environment. This is a difference of indirect vs. direct communication and control.

Medium

What is the medium of the cue? Is it print, icons, video, flashing, audio tones, music, vibration?

Diegetic forms: Wholes versus parts

Is the diegetic form an "object whole" or an "object part"? The latter can be a subpart that is interactive. The interaction is often a form of direct control. The objects, actions, and the object states may appear natural and intuitive. The sense of a part might be both concrete, such as a simulation of a physical element, or temporal or situational. The same signal may indicate different things in different contexts.

Non-diegetic forms: 3D in-world overlays

These are cues in a 3D environment, such as flashing surface decals or columns of light or movement such as something floating, that indicate something to which attention is being drawn.

Non-diegetic forms: 2D window overlays

These are 2D overlays, possibly on a 2D or 3D environment, such as cursors, dialog boxes, mini-maps, etc. They are external to the main environment or world but may directly affect it.

1.4.11 Published Principles of Design

Listed in the following sections are selected principles from the Universal Principles of Design(Lidwell et al., 2003). These are principles that have been found to apply in a wide range of situations. These guide choices in design, both affecting designs and moderating the process of choosing a design element. Many of these principles provide proven observations on human nature and human limitations. Here, the principles are interpreted from the perspective of properly solving and optimizing for the interactive visualization problem set.

80/20 Rule

Frequently, 80% of the use of something will be spent in 20% of the features, area, or time period. This tends to indicate that the most used features and controls should be visible with less effort. It also means that the 20% of the system that will get most used should be optimized the most.

Affordance

Affordance describes the degree to which the design of a physical or virtual object indicates, agrees with, and facilitates its function. Poor affordance, such as a pull handle on a push door, cause confusion, extra cognitive load, and delay in concentration on useful areas.

Alignment

Proper visual alignment of elements can be a very important signal as to grouping and sequencing. While poor alignment can slow the eye down, aligning elements which are not related can also have a negative effect as users spend extra time disambiguating meaning.

Archetypes

Universal patterns of style, elements, and theme that tend to resonate with a wide range of people due to innate sensitivities or biases. Archetypes often refer to particular cultural elements which, while generally influenced by innate taste, are often highly evolved and differentiated by cultural Darwinism.

Chunking

Rather than a stream of details or large forests of areas to focus attention on, chunking elements into a group represented by a single, larger element tends to make the overall information easier to process and recall.

Closure

Visually, a set of individual elements placed in a pattern will tend to be perceived as part of a continuos shape at a certain distance.

Common Fate

Visual elements that are moving in the same direction, to roughly the same degree, are perceived as being part of the same item or same group. In a running emulator, if several memory locations were increasing in value in sync, a visualization of that by movement, color, or pulsing in sync would naturally be very apparent.

Comparison

Representing one or more variables in an organized and deliberate fashion allows comparison that may reveal relationships and patterns. Different representation methods have varying strengths and weaknesses. Good diagrams, matched to the problem and latent patterns at hand, can immediately reveal salient points. A number of popular methods have well-known problems with cognitive fidelity. Pie charts for instance are perceived less accurately than other methods such as bar graphs. Yet variations on a pie chart can effectively illustrate the change in multiple variables over the passage of time.(Lidwell et al., 2003, 53).

Consistency

Consistency in placement, meaning, and structure of systems and information can enhance usability. Style guidelines for placement of 'OK' and 'Cancel' in dialog boxes is a common case. Different kinds of consistency can allow both efficient understanding and flexibility and increase power of expression.

Constancy

Constancy describes the effects of perceptual compensation mechanisms that can provide misleading estimation of shade, color, size, or other characteristics. Designs should avoid using certain types of complex pattern and color combination while relying on the user to sense an absolute brightness or other feature.

Fitts's Law

Fitts's Law says that the time and effort needed to reach a target is function of the target size and distance to a target.(Göktürk, 2010) A pop-up menu can be much more efficient than a menu far away. A large drag-and-drop area can speed operations.

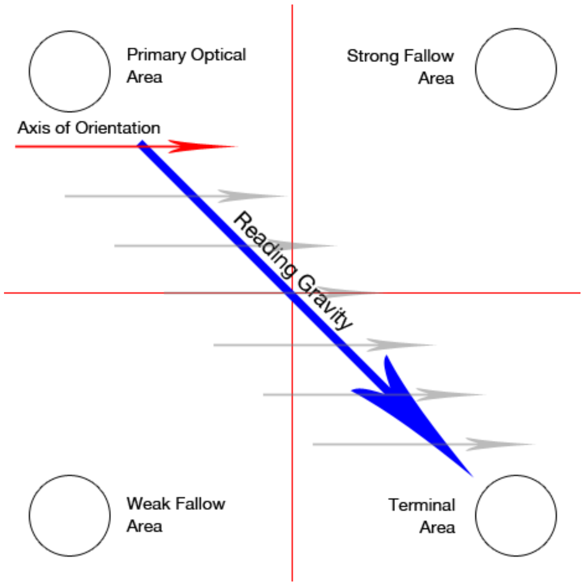
Figure 1.15. Fitts's Law, Shannon-Hartley Formulation(community, 2013a)

$$T = a + b\log_2\left(1 + \frac{2D}{W}\right)$$

Gutenberg Diagram

A Gutenberg Diagram describes the general pattern followed by a user's eyes when observing evenly distributed information. Placement of certain elements can be rationalized and optimized according to this common pattern, see: 1.16.(Bradley, 2011)

Figure 1.16. Gutenberg Diagram

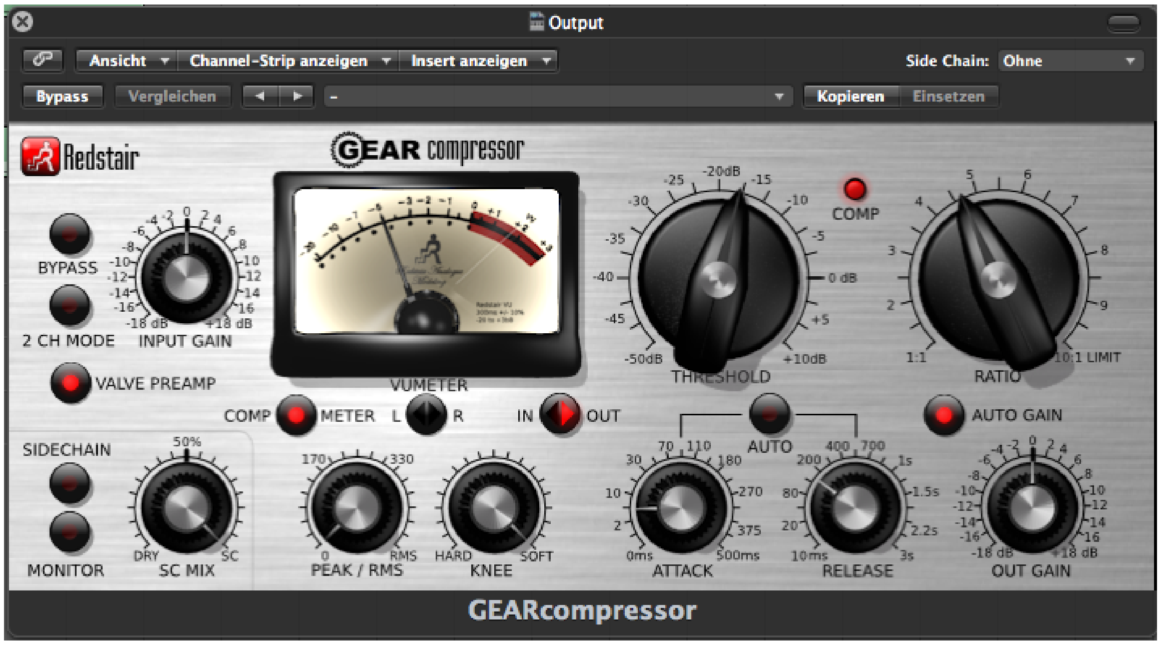


Mimicry / Skeuomorph

Designing elements to mimic other things, especially real-world objects, textures, and forms, and their action when interacted with, is a technique for which the output is called a skeuomorph. This technique is expected to make users feel immediately comfortable.(community, 2013c) This establishes an immediate shared context and mental model of what something is and how it can be used. An example of this for a software emulation of an analog audio compressor is shown in 1.17.

Critics of this method decry the constraints this places on use of space, ability to add features, and improving on real-world objects in general.

Figure 1.17. Skeuomorph for Audio Compressor(Göttling, 2013)



1.4.12 Taxonomy of Interactive Dynamics

> ”Meaningful analysis consists of repeated explorations as users develop insights about significant relationships, domain-specific contextual influences, and causal patterns. Confusing widgets, complex dialog boxes, hidden operations, incomprehensible displays, or slow response times can limit the range and depth of topics considered and may curtail thorough deliberation and introduce errors. To be most effective, visual analytics tools must support the fluent and flexible

use of visualizations at rates resonant with the pace of human thought."(Heer and Schneiderman, 2012)

Taxonomy of interactive dynamics for visual analysis, from (Heer and Schneiderman, 2012), applied to CAS:

1. Data & View Specification
   (a) *Visualize* data such as memory, code, state, and events by showing in context, in bulk when necessary, in compact understandable ways.
   (b) *Filter* data interactively through flexible controls that allow users to focus on one or more areas.
   (c) *Sort* data and metadata derived from data in multiple ways to allow highlight of important patterns.
   (d) *Derive* metadata to summarize at different granularity, by different perspectives, and at different times.
2. View Manipulation
   (a) *Select* items such as memory ranges, code points, and thresholds easily to highlight or filter.
   (b) *Navigate* in a multi-level granularity fashion to the points in code and data that are interesting.
   (c) *Coordinate* multiple views, such as CPU, memory, and system analysis, so that the overall picture can be assimilated.
   (d) *Organize* multiple views, such as CPU, memory, and peripheral views, to allow optimal use for a particular task.
3. Process & Provenance

(a) *Record* history, state, and results of emulation for later comparison, review, sharing, and testing.

(b) *Annotate* patterns to record user analysis for later use.

(c) *Share* results to leverage value and use of system.

(d) *Guide* users through setup, next steps, and tasks, both generically and for pedagogical purposes.

1.4.13 Gamification and Flow

In "Distinguishing Games and Simulation Games from Simulators" (Narayanasamy et al., 2006), the authors identify key features distinguishing a game, a simulation game, and a simulator. They also discuss the subtle difference between a game and a serious game. This exploration is potentially very useful for developing gamification strategies of formal or enthusiast learning activities, contests, or problem solving. Closely related to the what the authors discuss as game-play gestalt(Narayanasamy et al., 2006, 3) is the psychological phenomena of flow(Wikipedia, 2013c)(Csikszentmihalyi, 2013)(Csikszentmihalyi and Csikszentmihalyi, 1988). These related phenomina refer to hyper focused, optimal states of interaction with a system or activity. Good designs foster and enhance the ability for an interested user to achieve these states while a poor system disrupts a user's focus with broken paradigms, tedious interfaces, and the inability to present information at the right granularity in an optimal way.

Figure 1.18. Flow: Challenge vs. Skill

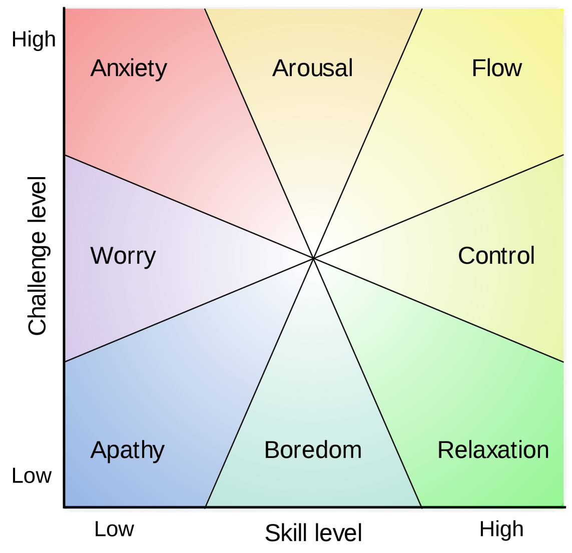
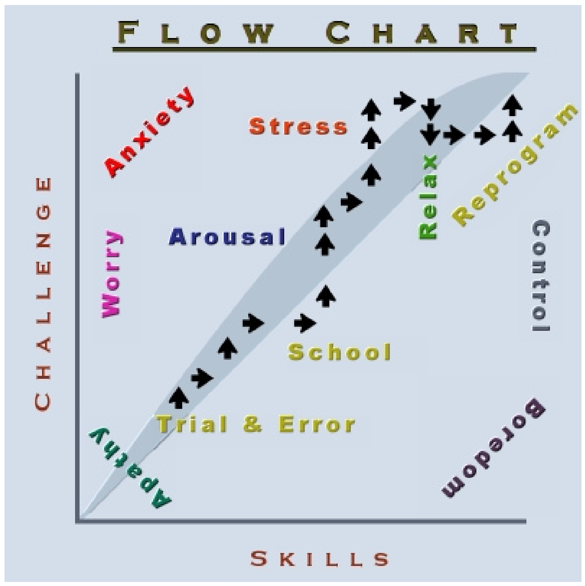


Figure 1.19. Flow Chart



Programming and users of a CAS will likely have a lot of overlap in their tasks, thinking, and concentration. It is well documented that interrupting a task that requires a large amount of concentration and working memory can cause a large amount of lost time and effort. In Parnin (2013), this working memory is described in terms of a number of distinct types:

1. Prospective memory - holds reminders to perform future actions in certain situations.
2. Attentive memory - holds conscious memories that are randomly accessible.
3. Associative memory - holds a set of implicit links between co-occuring elements.
4. Episodic memory - memory of past events.
5. Conceptual memory - range of memory between perceptions and abstractions.

A system that specifically supports and assists the user in offloading or reinforcing these types of memory would greatly improve the efficiency and capability of a user. Existing examples of these include a debugger remembering breakpoints and a programming editor automatically showing all of the lines where a 'todo' comment was inserted. Prospective examples might be 'painting' memory with a certain type or purpose or marking certain code as interesting or uninteresting to filter and remind in future passes.

1.4.14 Universal Application

It is well-known that novel application of a technique in one field or area originally discovered in another is one of the most common forms of innovation. Yet there continues to remain many cases where a core idea, well known in one field, is only much later applied to others. Often, even realizing there is an opportunity for a much better solution prevents timely realization of such solutions. Often, successful ideas can be somewhat universally applied to many other fields, although often with transformation or mapping that may not be immediately obvious. A field may be said to be analogous if a significant number of the subproblems can be mapped to each other as having similar form, elemental components, and some existing commonality in solutions. While transferable ideas and paradigms may be noticed serindipidously upon occasion, leveraging systematic summaries of the best ideas in somewhat analogous fields is often a way to produce candidate methods. These candidate methods and attempts to creatively apply them can lead not only to successful reapplication, but also to derivative, analogous, or even unrelated ideas.

1.4.15 Code and Algorithm Understanding

A user of a computer architecture simulator will likely find that they need to interpret the actions of some range of code and understand how associated memory is used. An inexperienced user will likely not immediately or easily understand any significant sequence of operations. An experienced developer may quickly recognize a pattern that matches well-known structures, practices, and algorithms. However, even in the latter case, correlating different sequences of code with associated data to reach a conclusion, and then remembering these results for later reuse, is something not widely available or yet fully evolved. An ideal system would allow layered, side-by-side review and annotation. While some support is available in special use commercial packages such as IDA Pro(Eagle, 2011), this capability can be greatly expanded to support exploratory investigation and capturing of knowledge and conclusions for operating systems and applications running in an emulator.

In addition to enabling human investigation and analysis of running systems, automated filtering, transformation, and recognition methods could be developed or adapted. This would allow greatly accelerated understanding and navigation of an existing system while enabling more advanced meta-analysis. An example of a useful meta-analysis is determining which programs employed similar overall structure, algorithms, data flow, and data formats. There exist a number of systems that statically or dynamically trace systems structure, data flow, and algorithmic nature. In Taherkhani et al. (2012), freshman sorting algorithms are recognized by their characteristics at an overall accuracy of 81% using a system called Aari.

Daikon(Group et al., 2013)(Ernst et al., 2007) detects program invariants in C/C++, Eiffel, Java, and Perl programs. A program invariant is a property that holds at certain points in a program; i.e. a relationship or state that is detected which likely explains what a program is doing. Examples would include the relationship between variables or the fact that a certain array of integers has become sorted. The Daikon system is extensible and adaptable, suitable for integration into a CAS for analysis of the effect of code segments on memory.

1.4.16 Processing

Processing(Reas and Fry, 2007), created by Ben Fry and Casey Reas in 2001 (Maeda, 2009) while in the MIT Media Lab Aesthetics and Computation Group to improve upon the Design by Numbers(DBN) system by John Maeda. Both systems were created to enabled artists and designers to learn and use programming in their work to create innovative interactive visualizations. Processing has since grown in capability and popularity so that it is used for a variety of purposes on many platforms. While originally only Java-based, Processing currently supports running 'sketches' (a complete Processing program) in Javascript, in an environment that is similar. Until the most recent versions, the Processing IDE supported export of the sketch as a Java applet. Now, Processing supports export of packaged Java-based executables for Linux, MacOSX, and Windows and a separate Javascript-based mode for browsers. Unfortunately, not all useful libraries have been ported from Java to Javascript and developers will have to maintain two versions of their own libraries to support all platforms. An additional export mode to Android, being Java-based, is similar to the desktop environment.

For this work, Processing is an ideal environment to rapidly prototype ideas.

This is due to popularity which provides both widely available familiarity and widespread documentation and examples, concise power, ease of development, and ability to integrate into Java-based or Javascript-based applications. Processing supports both 2D and 3D visualization, including OpenGL shader programs. It also provides simple integration of mouse and keyboard input and integration with other external devices, such as lab instruments, through various publicly available libraries. Once an interactive visualization method is working in isolation, it can be integrated with the emulator efficiently.

Other somewhat related work includes FreeStyler(of Engineering University of Dulsburg-Essen, 2013) and JeLSIM(Milligan and Thomas, 2009).

Characteristics of Adaptable Design

The key components of an adaptable design include maintaining separation of concerns, employing a flexible, modular architecture, supporting temporal models that include consideration of timing, sequencing, and speed of emulation. A key aspect of Java platform is the ability of the just in time (JIT) compiler to perform optimizations about how the system is currently running. If properly controlled through the injection patter, to virtual machine is free to optimize for the current configuration. Therefore, if a particular visualization is disabled or minimized, this minimal path can be highly optimized, allowing the emulator to run much faster.

1.4.17 Personas

A computer architecture emulation system, such as the EmuMaker 86 simulator (aka Graphical PC Simulator) (EmuMaker 86 simulator)(Black and Komala, 2011), provides emulation of an entire computer, a 386 PC with standard peripherals in this case. A pedagogical computer emulator provides multiple views into the operation of this simulated computer. In the case of the EmuMaker 86 simulator, this includes a text-mode and graphics mode screen, keyboard, memory, int erupts, timers, floppy drives, hard drives, IO ports, and the CPU including registers, ALUs, and flags. The EmuMaker 86 simulator provides separate panes, implemented in Java Swing, to allow viewing and interaction for each of these components. A menu is provided to allow configuration, initialization, and control of the running of the system. An extended ability is present to allow definition of datapaths for a custom CPU architecture which can be run directly. The following diagrams illustrate the user interface for these components. Other computer emulators exist for various similar and disparate purposes. Some other emulators useful for pedagogical purposes are listed in (Black and Komala, 2011). These systems share a certain range of literal representation of block diagrams of CPU components, grid layout of memory summary cells, and lists of actual machine instructions represented as assembly language statements.

Users of a CAS can be summarized by a small set of “personas”. [3] The main personas considered here include: Computer science (CS) students, computer architecture (CA) students, computer science/architecture professors (PROF), learning software developers (LSWD), and professional-level software developers (PSWD). These personas have a large degree of overlap in needs from a system, but differing in significant ways including interest in: areas of the system, levels of detail of operation and data, and importance of different types of patterns and statistics.

Computer science and computer architecture professors need a system that allows rapid proficiency to allow focusing on specific concepts and problems along with a wide range of features and operation flexibility. Students generally will have an interest in learning new concepts, becoming progressively more proficient at reasoning

[3] A persona is “A technique that employs fictitious users to guide decision making regarding features, interactions, and aesthetics.” (Lidwell et al., 2003)

about computer systems and software development, and avoiding expenditure of uninteresting, unrewarding effort. Progress will begin at some layer and progress incrementally. Software developers are generally working to solve specific problems. They may use a CAS to provide insight about performance, bugs, or to understand the structure and operational characteristics of a hardware or software system. Any of these may be working to extend the emulator or interactive visualizations for the emulator.

1.5 Specific Accomplishments

These accomplishments illustrate contributions which may enable further research and development of: computer architecture simulator (CAS) improvements, emulator related research projects, and methodology of constraining and developing innovative interactive visualizations.

1. Created a curated, core set of design constraints and concerns relevant to CAS visualization.

2. Created a catalog of relevant design elements, paradigms, and existing methods, relating these to CAS visualization.

3. Placed some of the most key relevant literature in the context of solving CAS visualization, providing a framework for many types of further exploration.

4. Developed sense of missing features, inefficiencies, and user requirements needed to move significantly toward optimal CAS use.

5. Reviewed and critiqued existing CAS and CAS-like systems with respect to usability, features, and modifiability.

6. Modified Emumaker 86 structure and visualization implementation method to support pluggable Processing modules, completely separating emulator engine and visualization components.
7. Designed several methods for further emulator development and debugging through integration with a reference emulator.
8. With constraints, concerns, and existing methods in mind, developed a set of new CAS related design elements.
9. Implemented interactive visualization elements in Processing sketches to demonstrate usability features and implementation method.
10. Developed user scenarios with postulated interactive visualization assistance to demonstrate the need, utility, and context for these features.
11. Summarized the most promising future research opportunities.

CHAPTER 2

REVIEW OF EXISTING EMULATORS

2.1 Existing Computer Emulators

This research incorporates insight from long experience managing and running VMWare(VMWare, 2013), VirtualBox(Oracle, 2013), Parallels(GmbH, 2013), Xen(Foundation, 2013a), KVM(Foundation, 2013b), UML(Dike, 2013)(Dike, 2006), OpenVZ(Project, 2013), QEMU(Bellard et al., 2013), SoftPC, and other virtualization systems.

Pedagogical emulators considered, as noted in Black and Komala (2011, 2):

Table 2.1: Pedagogical Computer Emulators considered

Emumaker 86	2013
WebMIPS	2004
ProcSim	2005
SimuProc	2004
K Scalar	2001
WinMips64	1992

WinDLX	1997
PCSpim	1990
Microprocessor Simulator	2003
Simulator	2003
MiniMIPS	2004
HDLDLX	2004
IECS	2000
p88110	2009
IASSim(Fagin and Skrien, 2013)	2012

These pedagogical emulators were examined directly by running and examining source code or indirectly through screen shots and published literature. There were major differences in which peripherals and computer features were supported, and in what could be effectively visualized. EmuMaker 86 simulator goes further in supporting a wide range of peripherals and supporting visualization of each of these components.

2.1.1 Emumaker 86

Emumaker 86(Black and Waggoner, 2013) is a PC emulator, able to run DOS and Windows 3.1, that includes fairly full emulation of an Intel 8086 processor, including microcode breakdown of each instruction. It also includes objected oriented implementation of each major subsystem and peripheral, including memory, CPU, segment registers, graphics card, keyboard, interrupts, floppy drives, hard drives, serial ports, parallel ports, and timer. Each subsystem has a representative visualization that can be enabled or disabled.

Emumaker 86 is highly unusual not only for providing implementation and visualization of all standard (early) PC peripherals, but also because it includes a hardware simulator for CPU design. An object-based graphic data path builder interface allows placement and connection of the datapath of logic units. A table interface implementing a control path builder allows definition of the control logic. The resulting CPU can then be run by the emulator user. This CPU logic also can have access to the main PC emulator which results in a co-processor relationship.

Figure 2.1. Emumaker 86 2013 Main Interface.

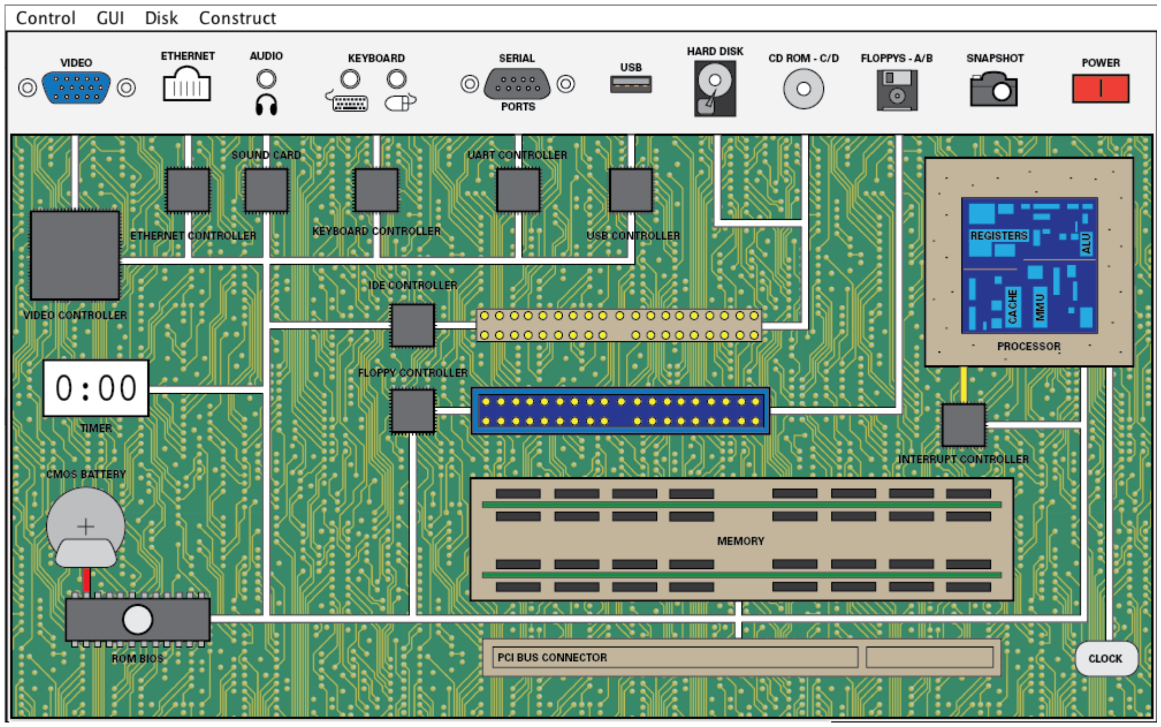
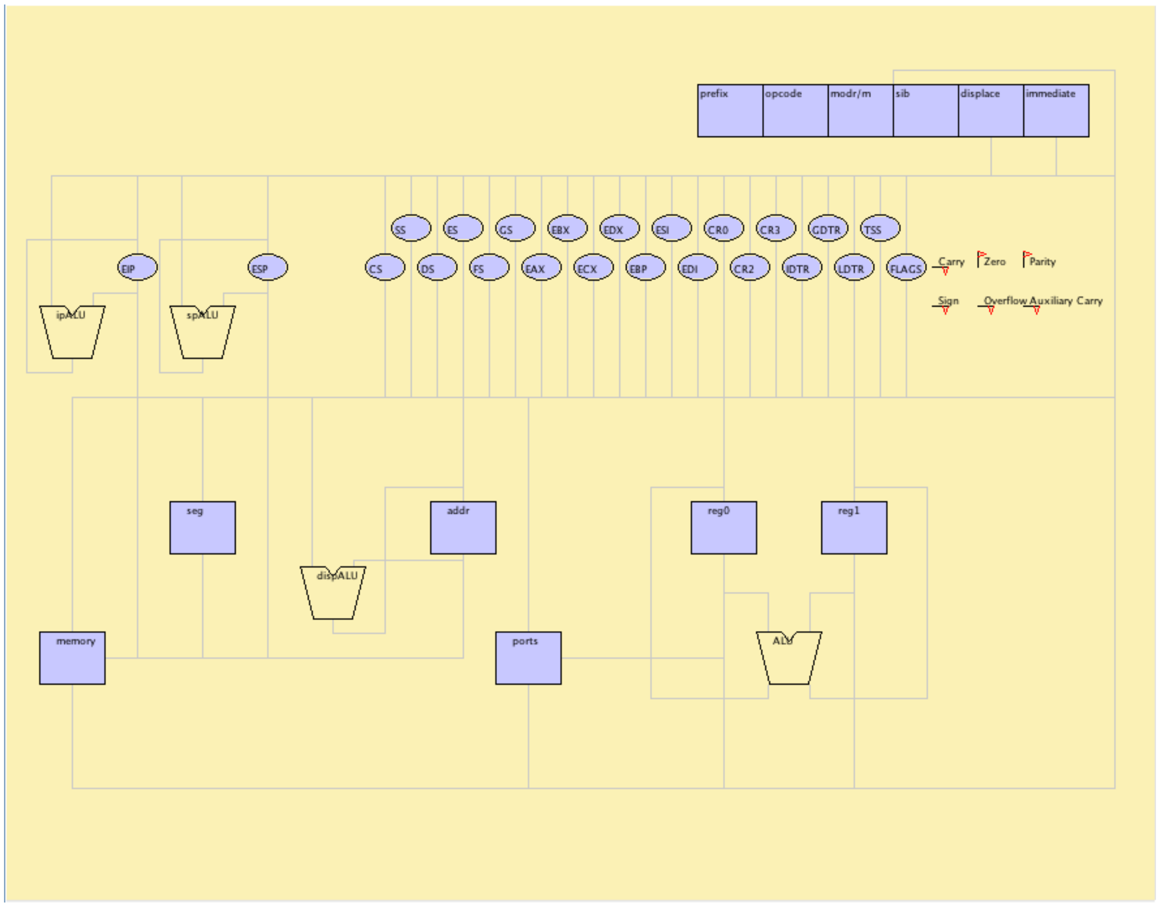


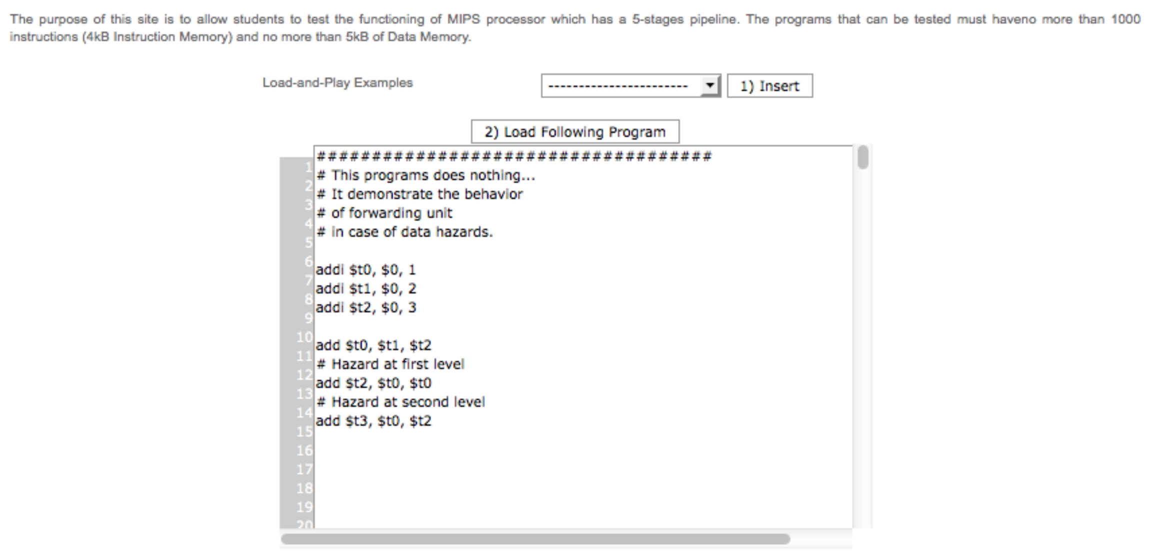
Figure 2.2. Representation of the CPU in Emumaker 86.



2.1.2 WebMIPS

WebMIPSPereira (2013)(Branovic et al., 2003) is a web-based MIPS CPU emulator, implementing only the CPU, memory, and registers. This emulator takes assembly as input and then allows a user to step through the CPU to see the effects.

Figure 2.3. WebMIPS Program Entry

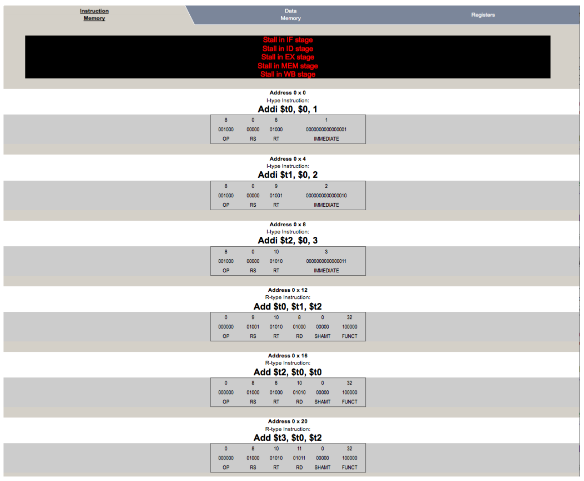

The purpose of this site is to allow students to test the functioning of MIPS processor which has a 5-stages pipeline. The programs that can be tested must haveno more than 1000 instructions (4kB Instruction Memory) and no more than 5kB of Data Memory.

```
#################################
# This programs does nothing...
# It demonstrate the behavior
# of forwarding unit
# in case of data hazards.

addi $t0, $0, 1
addi $t1, $0, 2
addi $t2, $0, 3

add $t0, $t1, $t2
# Hazard at first level
add $t2, $t0, $t0
# Hazard at second level
add $t3, $t0, $t2
```

Figure 2.4. WebMIPS CPU



Stall in IF stage
Stall in ID stage
Stall in EX stage
Stall in MEM stage
Stall in WB stage

Address 0 x 0
I-type Instruction:
Addi $t0, $0, 1

8	0	8	1
001000	00000	01000	0000000000000001
OP	RS	RT	IMMEDIATE

Address 0 x 4
I-type Instruction:
Addi $t1, $0, 2

8	0	9	2
001000	00000	01001	0000000000000010
OP	RS	RT	IMMEDIATE

Address 0 x 8
I-type Instruction:
Addi $t2, $0, 3

8	0	10	3
001000	00000	01010	0000000000000011
OP	RS	RT	IMMEDIATE

Address 0 x 12
R-type Instruction:
Add $t0, $t1, $t2

0	9	10	8	0	32
000000	01001	01010	01000	00000	100000
OP	RS	RT	RD	SHAMT	FUNCT

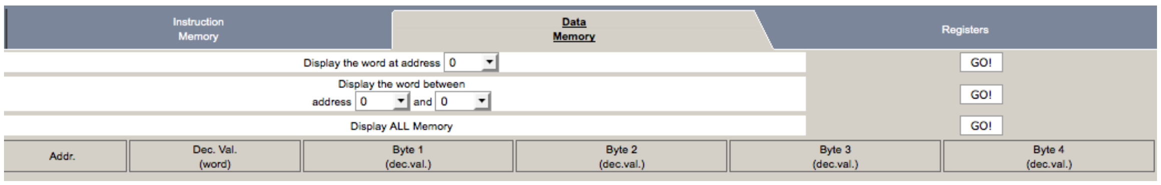
Address 0 x 16
R-type Instruction:
Add $t2, $t0, $t0

0	8	8	10	0	32
000000	01000	01000	01010	00000	100000
OP	RS	RT	RD	SHAMT	FUNCT

Address 0 x 20
R-type Instruction:
Add $t3, $t0, $t2

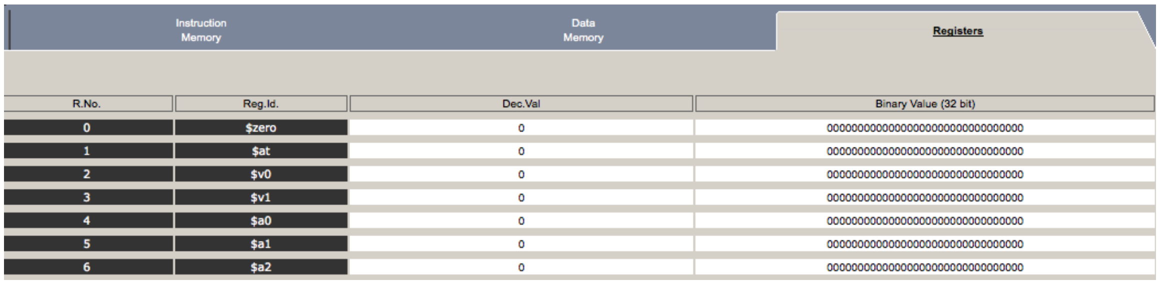
0	8	10	11	0	32
000000	01000	01010	01011	00000	100000
OP	RS	RT	RD	SHAMT	FUNCT

Figure 2.5. WebMIPS Memory


Instruction Memory
Data Memory
Registers
Display the word at address 0
GO!
Display the word between address 0 and 0
GO!
Display ALL Memory
GO!

Addr.	Dec. Val. (word)	Byte 1 (dec.val.)	Byte 2 (dec.val.)	Byte 3 (dec.val.)	Byte 4 (dec.val.)

Figure 2.6. WebMIPS Registers

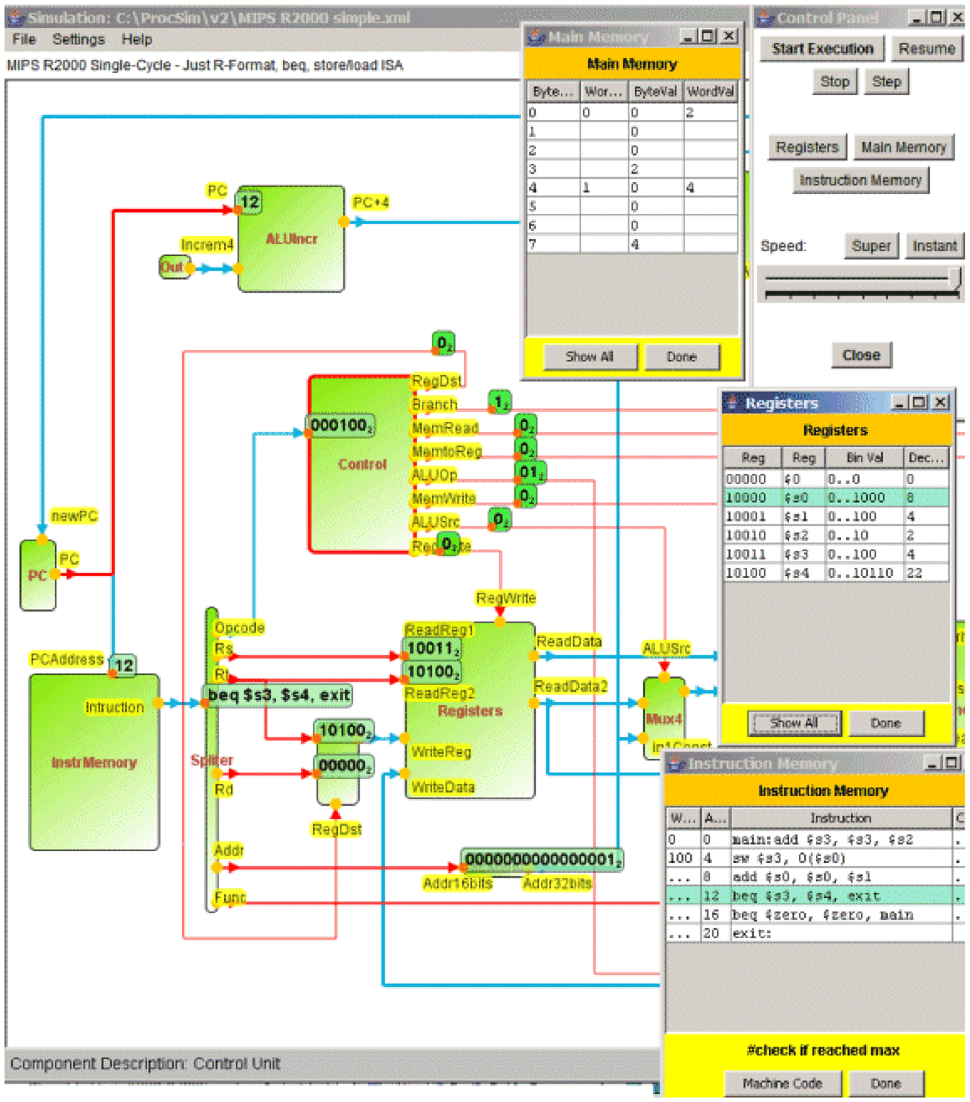


R.No.	Reg.Id.	Dec.Val	Binary Value (32 bit)
0	$zero	0	00000000000000000000000000000000
1	$at	0	00000000000000000000000000000000
2	$v0	0	00000000000000000000000000000000
3	$v1	0	00000000000000000000000000000000
4	$a0	0	00000000000000000000000000000000
5	$a1	0	00000000000000000000000000000000
6	$a2	0	00000000000000000000000000000000

2.1.3 ProcSim

ProcSim is a visual MIPS R2000 processor simulator. The main features include input assembly, controllable animation, different datapaths, memory reading and writing, and defining your own processor datapaths in XML and via a diagram editor.

Figure 2.7. ProcSim Animation Screen



Main Memory

Byte...	Wor...	ByteVal	WordVal
0	0	0	2
1		0	
2		0	
3		2	
4	1	0	4
5		0	
6		0	
7		4	

Registers

Reg	Reg	Bin Val	Dec...
00000	$0	0..0	0
10000	$s0	0..1000	8
10001	$s1	0..100	4
10010	$s2	0..10	2
10011	$s3	0..100	4
10100	$s4	0..10110	22

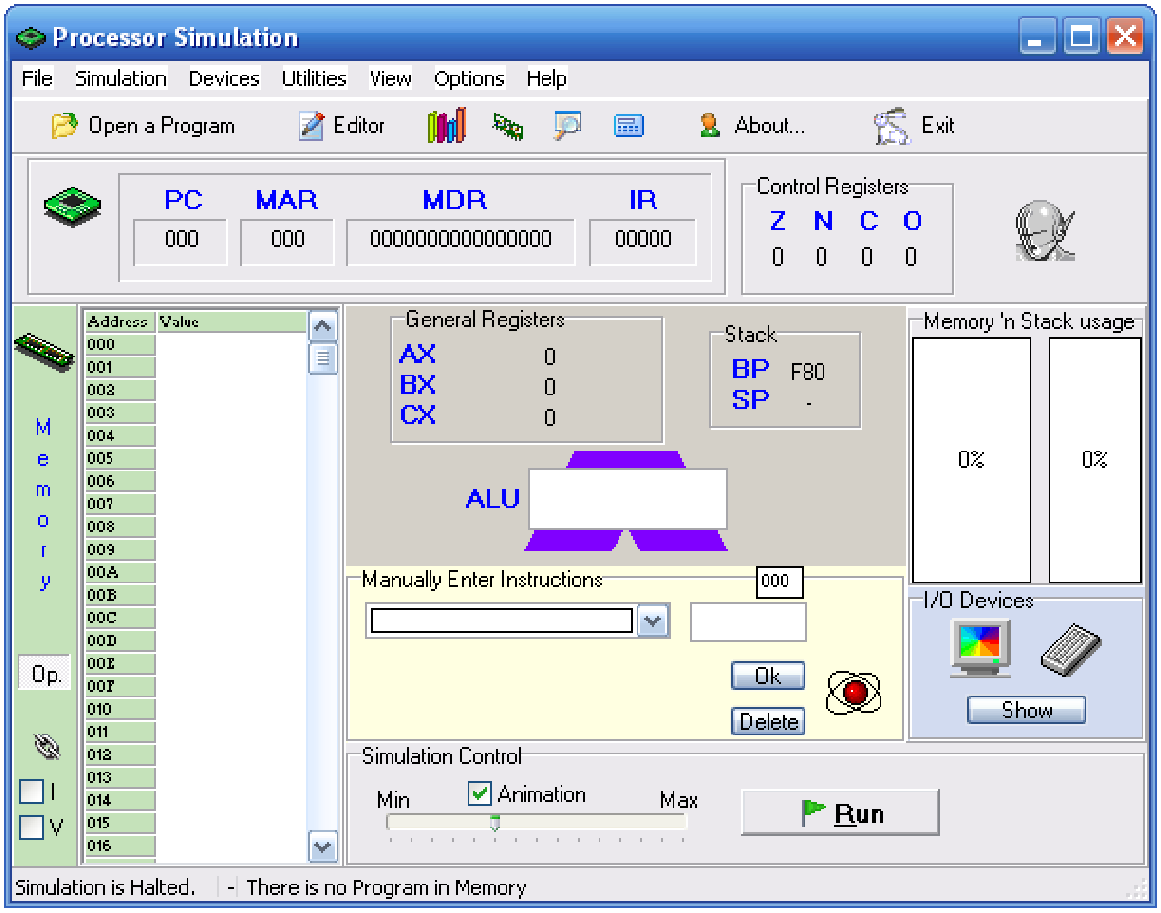
Instruction Memory

W...	A...	Instruction	C
0	0	main:add $s3, $s3, $s2	.
100	4	sw $s3, 0($s0)	.
...	8	add $s0, $s0, $s1	.
...	12	beq $s3, $s4, exit	.
...	16	beq $zero, $zero, main	.
...	20	exit:	

2.1.4 SimuProc

SimuProc is a "hypothetical processor simulator", showing how each assembly instruction executes internally. SimuProc supports about 50 instructions.

Figure 2.8. SimuProc



PC 000
MAR 000
MDR 0000000000000000
IR 00000

Z 0 N 0 C 0 O 0

AX 0
BX 0
CX 0

BP F80
SP -

0% 0%

000

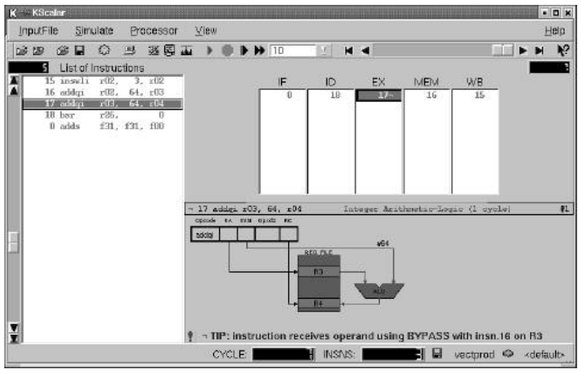
Animation ✓

Simulation is Halted. - There is no Program in Memory

2.1.5 KScalar

KScalar(Moure et al., 2002) is a graphical simulation tool intended to support study of processors using the Alpha AXP instruction set. It supports examination of CPUS "from a very simple in-order, scalar pipeline, to a detailed out-of-order, superscalar pipeline with non-blocking caches, speculative execution, and complex branch prediction".

Figure 2.9. KScalar CPU Simulation



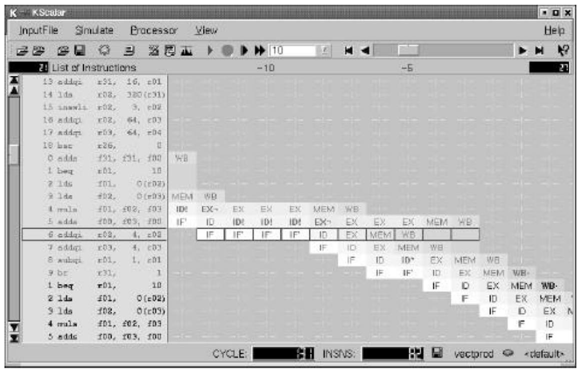
```
15 inswli   r02,   3, r02
16 addqi    r02,  64, r03
17 addqi    r03,  64, r04
18 bsr      r26,       0
 0 adds     f31, f31, f00
```

```
IF   ID   EX    MEM   WB
 0   18   17¬   16    15
```

```
¬ 17 addqi r03, 64, r04          Integer Arithmetic-Logic (1 cycle)
```

¬ TIP: instruction receives operand using BYPASS with insn.16 on R3

Figure 2.10. KScalar Pipeline Cycle Diagram

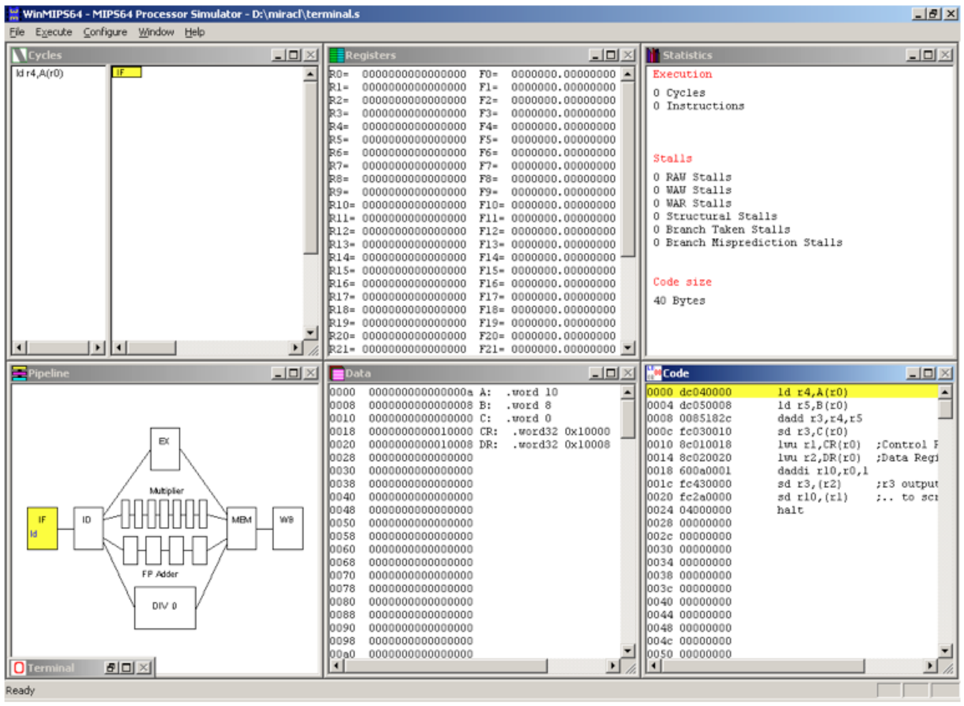


```
13 addqi    r31,  16, r01
14 lds      r02, 320(r31)
15 inswli   r02,   3, r02
16 addqi    r02,  64, r03
17 addqi    r03,  64, r04
18 bsr      r26,       0
 0 adds     f31, f31, f00   WB
 1 beq      r01,      10
 2 lds      f01,   0(r02)
 3 lds      f02,   0(r03)   MEM  WB
 4 muls     f01, f02, f03   ID!  EX¬  EX   EX   EX   MEM  WB
 5 adds     f00, f03, f00   IF'  ID   ID!  ID!  ID!  EX¬  EX   EX   EX   MEM  WB
 6 addqi    r02,   4, r02        IF   IF'  IF'  IF'  ID   EX   MEM  WB
 7 addqi    r03,   4, r03                            IF   ID   EX   MEM  WB
 8 subqi    r01,   1, r01                                 IF   ID   ID*  EX   MEM  WB
 9 br       r31,       1                                       IF   IF'  ID   EX   MEM  WB·
 1 beq      r01,      10                                                 IF   ID   EX   MEM  WB·
 2 lds      f01,   0(r02)                                                     IF   ID   EX   MEM
 3 lds      f02,   0(r03)                                                          IF   ID   EX
 4 muls     f01, f02, f03                                                               IF   ID
 5 adds     f00, f03, f00                                                                    IF
```

2.1.6 WinMips64

WinMIPS64 is an instruction set simulator, designed as a replacement for WinDLX.(Scott, 2012)

Figure 2.11. WinMips64 Main Emulation Window


WinMIPS64 - MIPS64 Processor Simulator - D:\miracl\terminal.s
File Execute Configure Window Help
Cycles
Registers
Statistics
Pipeline
Data
Code
Terminal
Ready
IF
ID
EX
Multiplier
FP Adder
DIV 0
MEM
WB

```
ld r4,A(r0)   IF
```

```
R0=  0000000000000000  F0=  0000000.00000000
R1=  0000000000000000  F1=  0000000.00000000
R2=  0000000000000000  F2=  0000000.00000000
R3=  0000000000000000  F3=  0000000.00000000
R4=  0000000000000000  F4=  0000000.00000000
R5=  0000000000000000  F5=  0000000.00000000
R6=  0000000000000000  F6=  0000000.00000000
R7=  0000000000000000  F7=  0000000.00000000
R8=  0000000000000000  F8=  0000000.00000000
R9=  0000000000000000  F9=  0000000.00000000
R10= 0000000000000000  F10= 0000000.00000000
R11= 0000000000000000  F11= 0000000.00000000
R12= 0000000000000000  F12= 0000000.00000000
R13= 0000000000000000  F13= 0000000.00000000
R14= 0000000000000000  F14= 0000000.00000000
R15= 0000000000000000  F15= 0000000.00000000
R16= 0000000000000000  F16= 0000000.00000000
R17= 0000000000000000  F17= 0000000.00000000
R18= 0000000000000000  F18= 0000000.00000000
R19= 0000000000000000  F19= 0000000.00000000
R20= 0000000000000000  F20= 0000000.00000000
R21= 0000000000000000  F21= 0000000.00000000
```

```
Execution
0 Cycles
0 Instructions

Stalls
0 RAW Stalls
0 WAW Stalls
0 WAR Stalls
0 Structural Stalls
0 Branch Taken Stalls
0 Branch Misprediction Stalls

Code size
40 Bytes
```

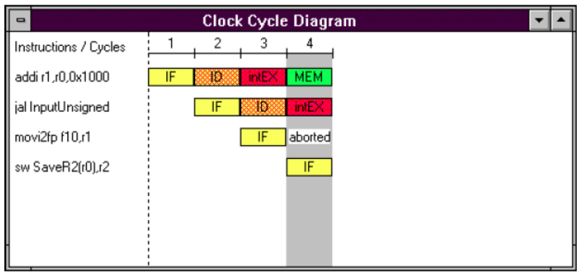
```
0000  000000000000000a A:   .word 10
0008  0000000000000008 B:   .word 8
0010  0000000000000000 C:   .word 0
0018  0000000000010000 CR:   .word32 0x10000
0020  0000000000010008 DR:   .word32 0x10008
0028  0000000000000000
0030  0000000000000000
0038  0000000000000000
0040  0000000000000000
0048  0000000000000000
0050  0000000000000000
0058  0000000000000000
0060  0000000000000000
0068  0000000000000000
0070  0000000000000000
0078  0000000000000000
0080  0000000000000000
0088  0000000000000000
0090  0000000000000000
0098  0000000000000000
00a0  0000000000000000
```

```
0000 dc040000       ld r4,A(r0)
0004 dc050008       ld r5,B(r0)
0008 0085182c       dadd r3,r4,r5
000c fc030010       sd r3,C(r0)
0010 8c010018       lwu r1,CR(r0)   ;Control F
0014 8c020020       lwu r2,DR(r0)   ;Data Regi
0018 600a0001       daddi r10,r0,1
001c fc430000       sd r3,(r2)      ;r3 output
0020 fc2a0000       sd r10,(r1)     ;.. to scr
0024 04000000       halt
0028 00000000
002c 00000000
0030 00000000
0034 00000000
0038 00000000
003c 00000000
0040 00000000
0044 00000000
0048 00000000
004c 00000000
0050 00000000
```

2.1.7 WinDLX

WinDLX is a RISC processor architecture emulator created in the mid-1990's to simulate a simplified MIPS CPU.

Figure 2.12. WinDLX Clock Cycle Diagram

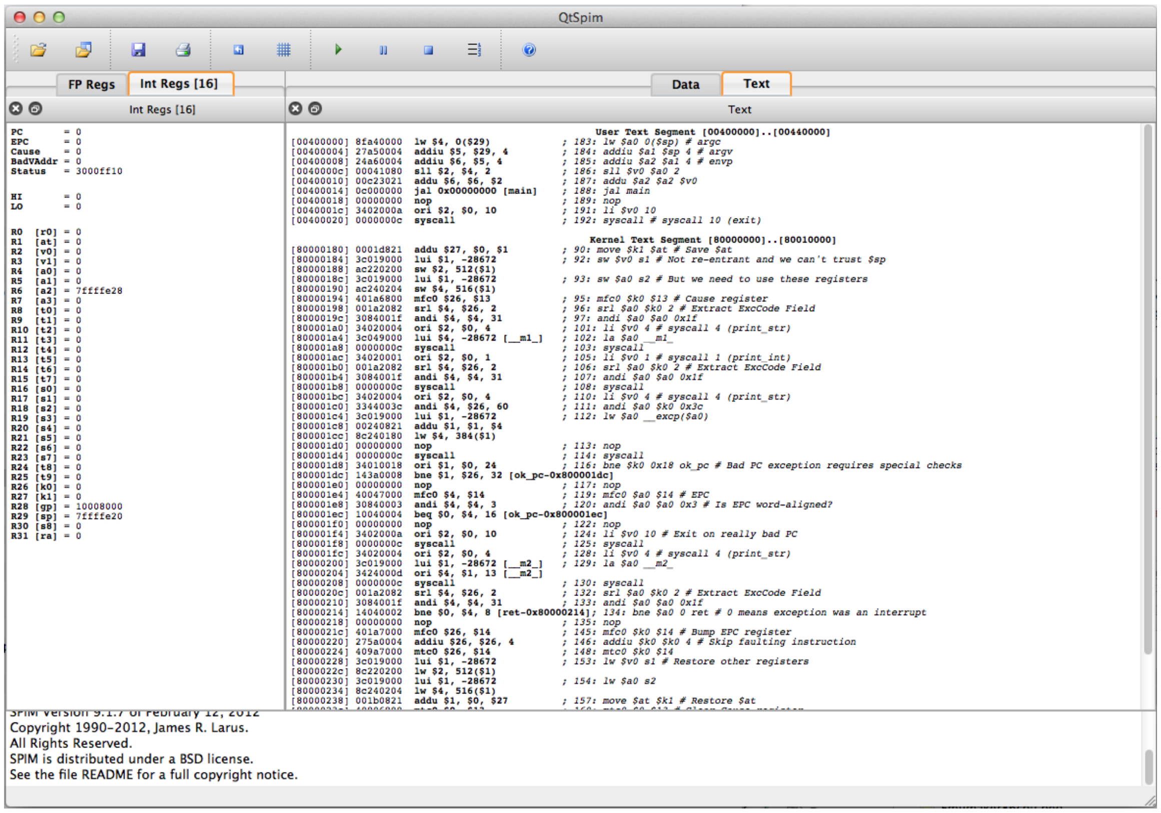


Instructions / Cycles	1	2	3	4
addi r1,r0,0x1000	IF	ID	intEX	MEM
jal InputUnsigned		IF	ID	intEX
movi2fp f10,r1			IF	aborted
sw SaveR2(r0),r2				IF

2.1.8 Spim

Spim was created in 1990 to emulate a MIPS processor. The current version implements almost the entire MIPS32 assembler-extended instruction set.

Figure 2.13. Spim Emulator



```
PC       = 0
EPC      = 0
Cause    = 0
BadVAddr = 0
Status   = 3000ff10

HI       = 0
LO       = 0

R0  [r0] = 0
R1  [at] = 0
R2  [v0] = 0
R3  [v1] = 0
R4  [a0] = 0
R5  [a1] = 0
R6  [a2] = 7ffffe28
R7  [a3] = 0
R8  [t0] = 0
R9  [t1] = 0
R10 [t2] = 0
R11 [t3] = 0
R12 [t4] = 0
R13 [t5] = 0
R14 [t6] = 0
R15 [t7] = 0
R16 [s0] = 0
R17 [s1] = 0
R18 [s2] = 0
R19 [s3] = 0
R20 [s4] = 0
R21 [s5] = 0
R22 [s6] = 0
R23 [s7] = 0
R24 [t8] = 0
R25 [t9] = 0
R26 [k0] = 0
R27 [k1] = 0
R28 [gp] = 10008000
R29 [sp] = 7ffffe20
R30 [s8] = 0
R31 [ra] = 0
```

```
                                             User Text Segment [00400000]..[00440000]
[00400000] 8fa40000  lw $4, 0($29)            ; 183: lw $a0 0($sp) # argc
[00400004] 27a50004  addiu $5, $29, 4         ; 184: addiu $a1 $sp 4 # argv
[00400008] 24a60004  addiu $6, $5, 4          ; 185: addiu $a2 $a1 4 # envp
[0040000c] 00041080  sll $2, $4, 2            ; 186: sll $v0 $a0 2
[00400010] 00c23021  addu $6, $6, $2          ; 187: addu $a2 $a2 $v0
[00400014] 0c000000  jal 0x00000000 [main]    ; 188: jal main
[00400018] 00000000  nop                      ; 189: nop
[0040001c] 3402000a  ori $2, $0, 10           ; 191: li $v0 10
[00400020] 0000000c  syscall                  ; 192: syscall # syscall 10 (exit)

                                            Kernel Text Segment [80000000]..[80010000]
[80000180] 0001d821  addu $27, $0, $1         ; 90: move $k1 $at # Save $at
[80000184] 3c019000  lui $1, -28672           ; 92: sw $v0 s1 # Not re-entrant and we can't trust $sp
[80000188] ac220200  sw $2, 512($1)
[8000018c] 3c019000  lui $1, -28672           ; 93: sw $a0 s2 # But we need to use these registers
[80000190] ac240204  sw $4, 516($1)
[80000194] 401a6800  mfc0 $26, $13            ; 95: mfc0 $k0 $13 # Cause register
[80000198] 001a2082  srl $4, $26, 2           ; 96: srl $a0 $k0 2 # Extract ExcCode Field
[8000019c] 3084001f  andi $4, $4, 31          ; 97: andi $a0 $a0 0x1f
[800001a0] 34020004  ori $2, $0, 4            ; 101: li $v0 4 # syscall 4 (print_str)
[800001a4] 3c049000  lui $4, -28672 [__m1_]   ; 102: la $a0 __m1_
[800001a8] 0000000c  syscall                  ; 103: syscall
[800001ac] 34020001  ori $2, $0, 1            ; 105: li $v0 1 # syscall 1 (print_int)
[800001b0] 001a2082  srl $4, $26, 2           ; 106: srl $a0 $k0 2 # Extract ExcCode Field
[800001b4] 3084001f  andi $4, $4, 31          ; 107: andi $a0 $a0 0x1f
[800001b8] 0000000c  syscall                  ; 108: syscall
[800001bc] 34020004  ori $2, $0, 4            ; 110: li $v0 4 # syscall 4 (print_str)
[800001c0] 3344003c  andi $4, $26, 60         ; 111: andi $a0 $k0 0x3c
[800001c4] 3c019000  lui $1, -28672           ; 112: lw $a0 __excp($a0)
[800001c8] 00240821  addu $1, $1, $4
[800001cc] 8c240180  lw $4, 384($1)
[800001d0] 00000000  nop                      ; 113: nop
[800001d4] 0000000c  syscall                  ; 114: syscall
[800001d8] 34010018  ori $1, $0, 24           ; 116: bne $k0 0x18 ok_pc # Bad PC exception requires special checks
[800001dc] 143a0008  bne $1, $26, 32 [ok_pc-0x800001dc]
[800001e0] 00000000  nop                      ; 117: nop
[800001e4] 40047000  mfc0 $4, $14             ; 119: mfc0 $a0 $14 # EPC
[800001e8] 30840003  andi $4, $4, 3           ; 120: andi $a0 $a0 0x3 # Is EPC word-aligned?
[800001ec] 10040004  beq $0, $4, 16 [ok_pc-0x800001ec]
[800001f0] 00000000  nop                      ; 122: nop
[800001f4] 3402000a  ori $2, $0, 10           ; 124: li $v0 10 # Exit on really bad PC
[800001f8] 0000000c  syscall                  ; 125: syscall
[800001fc] 34020004  ori $2, $0, 4            ; 128: li $v0 4 # syscall 4 (print_str)
[80000200] 3c019000  lui $1, -28672 [__m2_]   ; 129: la $a0 __m2_
[80000204] 3424000d  ori $4, $1, 13 [__m2_]
[80000208] 0000000c  syscall                  ; 130: syscall
[8000020c] 001a2082  srl $4, $26, 2           ; 132: srl $a0 $k0 2 # Extract ExcCode Field
[80000210] 3084001f  andi $4, $4, 31          ; 133: andi $a0 $a0 0x1f
[80000214] 14040002  bne $0, $4, 8 [ret-0x80000214]; 134: bne $a0 0 ret # 0 means exception was an interrupt
[80000218] 00000000  nop                      ; 135: nop
[8000021c] 401a7000  mfc0 $26, $14            ; 145: mfc0 $k0 $14 # Bump EPC register
[80000220] 275a0004  addiu $26, $26, 4        ; 146: addiu $k0 $k0 4 # Skip faulting instruction
[80000224] 409a7000  mtc0 $26, $14            ; 148: mtc0 $k0 $14
[80000228] 3c019000  lui $1, -28672           ; 153: lw $v0 s1 # Restore other registers
[8000022c] 8c220200  lw $2, 512($1)
[80000230] 3c019000  lui $1, -28672           ; 154: lw $a0 s2
[80000234] 8c240204  lw $4, 516($1)
[80000238] 001b0821  addu $1, $0, $27         ; 157: move $at $k1 # Restore $at
```

```
SPIM Version 9.1.7 of February 12, 2012
Copyright 1990-2012, James R. Larus.
All Rights Reserved.
SPIM is distributed under a BSD license.
See the file README for a full copyright notice.
```

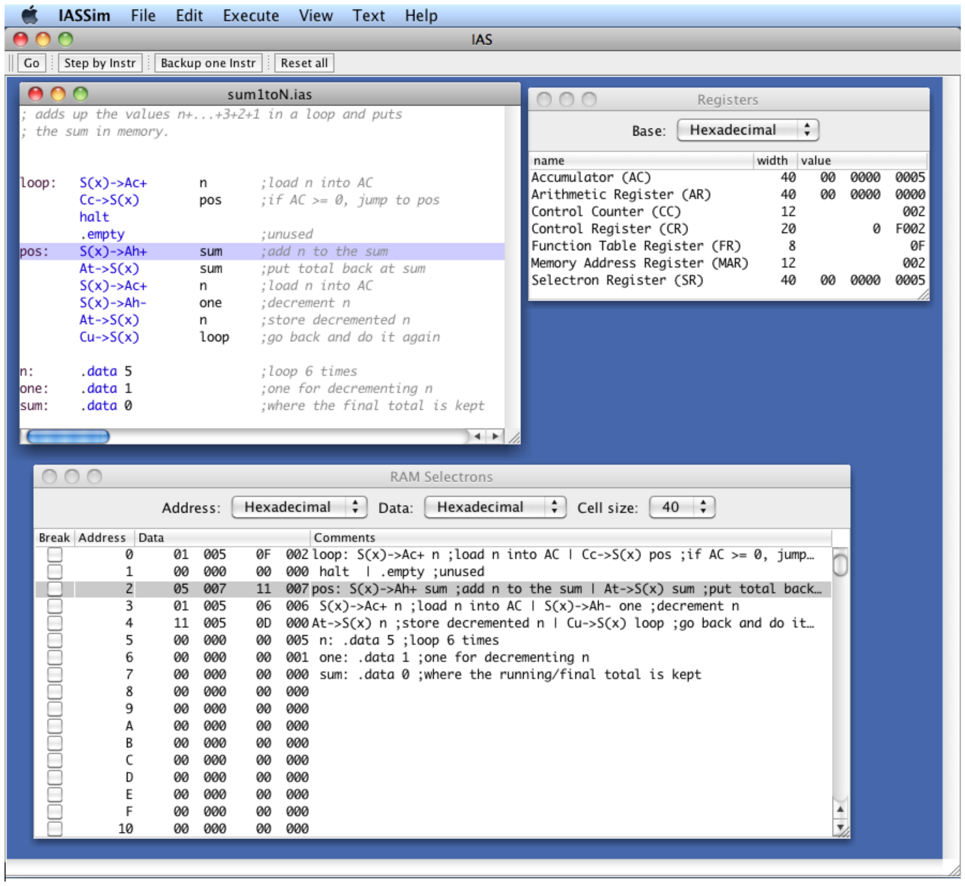
2.1.9 p88110

p88110 is a graphical RISC simulator, used since 1996 by students in computer architecture and organization.(Garcia et al., 2009)

2.1.10 IASSim

IASSim(Fagin and Skrien, 2013) is a programmable emulator for the Princeton IAS/Von Neumann Machine. IASSim is implemented in Java and

Figure 2.14. IASSim Emulator



```
; adds up the values n+...+3+2+1 in a loop and puts
; the sum in memory.


loop:   S(x)->Ac+     n       ;load n into AC
        Cc->S(x)      pos     ;if AC >= 0, jump to pos
        halt
        .empty                ;unused
pos:    S(x)->Ah+     sum     ;add n to the sum
        At->S(x)      sum     ;put total back at sum
        S(x)->Ac+     n       ;load n into AC
        S(x)->Ah-     one     ;decrement n
        At->S(x)      n       ;store decremented n
        Cu->S(x)      loop    ;go back and do it again

n:      .data 5               ;loop 6 times
one:    .data 1               ;one for decrementing n
sum:    .data 0               ;where the final total is kept
```

Base: Hexadecimal

```
name                            width  value
Accumulator (AC)                  40   00  0000  0005
Arithmetic Register (AR)          40   00  0000  0000
Control Counter (CC)              12               002
Control Register (CR)             20            0 F002
Function Table Register (FR)       8                0F
Memory Address Register (MAR)     12               002
Selectron Register (SR)           40   00  0000  0005
```

Address: Hexadecimal Data: Hexadecimal Cell size: 40

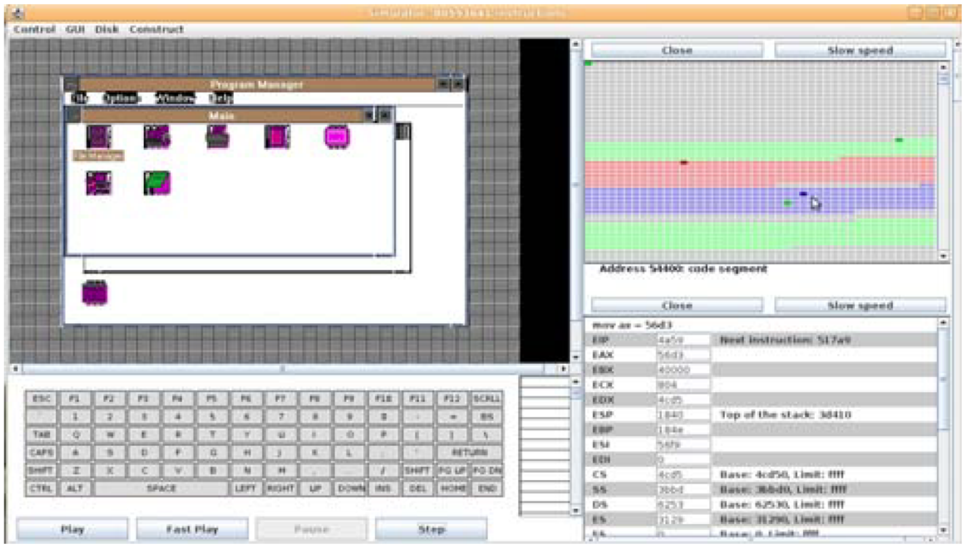
```
Break Address Data              Comments
 ☐       0    01  005   0F  002 loop: S(x)->Ac+ n ;load n into AC | Cc->S(x) pos ;if AC >= 0, jump…
 ☐       1    00  000   00  000  halt  | .empty ;unused
 ☐       2    05  007   11  007 pos: S(x)->Ah+ sum ;add n to the sum | At->S(x) sum ;put total back…
 ☐       3    01  005   06  006  S(x)->Ac+ n ;load n into AC | S(x)->Ah- one ;decrement n
 ☐       4    11  005   0D  000 At->S(x) n ;store decremented n | Cu->S(x) loop ;go back and do it…
 ☐       5    00  000   00  005  n: .data 5 ;loop 6 times
 ☐       6    00  000   00  001  one: .data 1 ;one for decrementing n
 ☐       7    00  000   00  000  sum: .data 0 ;where the running/final total is kept
 ☐       8    00  000   00  000
 ☐       9    00  000   00  000
 ☐       A    00  000   00  000
 ☐       B    00  000   00  000
 ☐       C    00  000   00  000
 ☐       D    00  000   00  000
 ☐       E    00  000   00  000
 ☐       F    00  000   00  000
 ☐      10    00  000   00  000
```

2.2 Computer Architecture Emulator Components

A user of one of these systems is presented with interfaces that will be foreign at first to students, but functional and usable for a certain range of exploration. However, the effectiveness and types of exploration are severely limited. They are limited by the presentation and visualization methods available, by the user interaction model, and by the range of operations that a user can invoke. The following provides critiques and raw requirements for each subsystem as an illustration in context of the need for better solutions.

Computer Emulator

Figure 2.15. Emumaker 86(Black and Komala, 2011)



Address 54400: code segment

mov ax = 56d3

EIP	4a59	Next instruction: 517a9
EAX	56d3	
EBX	40000	
ECX	804	
EDX	4cd5	
ESP	1840	Top of the stack: 38410
EBP	184e	
ESI	5ef9	
EDI	0	
CS	4cd5	Base: 4cd50, Limit: ffff
SS	38bd	Base: 38bd0, Limit: ffff
DS	6253	Base: 62530, Limit: ffff
ES	3129	Base: 31290, Limit: ffff
SS	0	Base: 0, Limit: ffff

2.2.1 CPU

The simulation of the computer CPU is crucial to understanding the operation of the computer and associated software. The execution of instructions directly manipulates the registers, status flags, and current addresses being read and written. A typical depiction involves symbolic representation of the registers, lines, and flags, sometimes with animation of trace activation. Often, the CPU representation is paired with some type of instruction trace listing and a memory view.

For single stepping through instructions, the representation in 2.16 provides insight into what the current state is. These are aspects that could be improved:

1. The space taken is not used efficiently or effectively. About 30 items are using significant screen real estate while the labels are difficult to read.

2. Once an instruction has executed, while the current state is visible, it is difficult to notice all transitions involved such as the state that the flags were in a moment

ago. A better solution would visibly indicate what the last transition was, and it would support running backwards and forwards at will to clarify.

3. What does a sequence of instructions accomplish? What is the pattern of those instructions?

4. What algorithms are executing?

5. When is the operating system / application boundary crossed? When are device drivers running?

6. What instructions and instruction types are used frequently and which are seldom used?

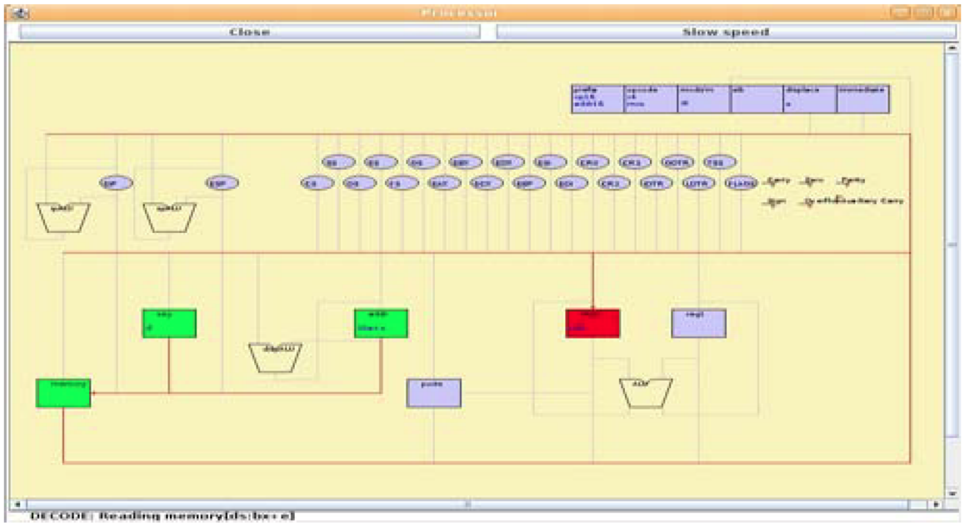
7. How much time is each instruction taking? What are the statistics for processor / memory efficiency for a segment of code?

8. What are the data dependencies and data flows through a system?

9. What is the type and use of data being processed? Are they scalars, strings, or pointers?

10. What was learned from a previous run of the same program that is running now? What does it know about what the user is interested in?

Figure 2.16. Emumaker 86(Black and Komala, 2011)

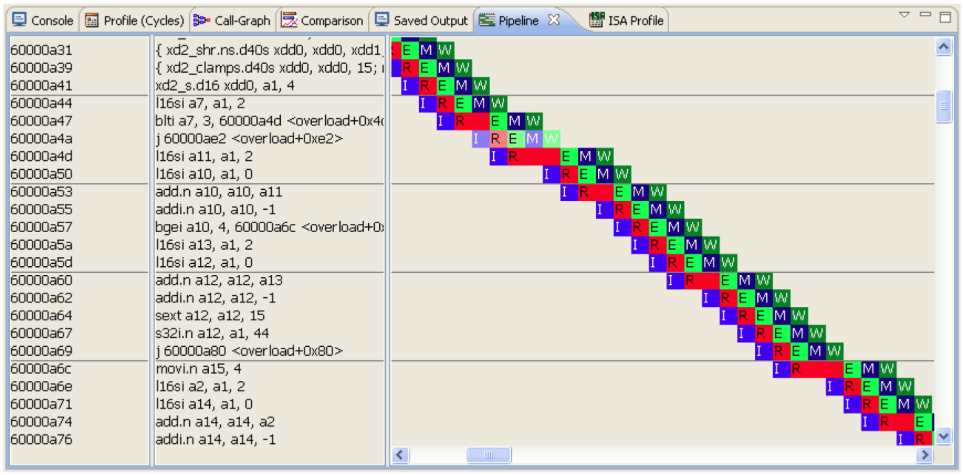


DECODE: Reading memory[ds:bx+ e]

2.2.2 Pipeline

Pipeline representation for a processor is key to understanding instruction and CPU performance and the full mechanics of instruction execution. If pipelining information is shown, it is usually shown using the cascading stage table as in 2.17. This is useful for a small number of instructions on a single core, but doesn't provide a scalable, multi-core mechanism that gives aggregate or as efficient as possible summary of execution.

Figure 2.17. Arcis Instruction Pipeline Representation(Black and Komala, 2011)



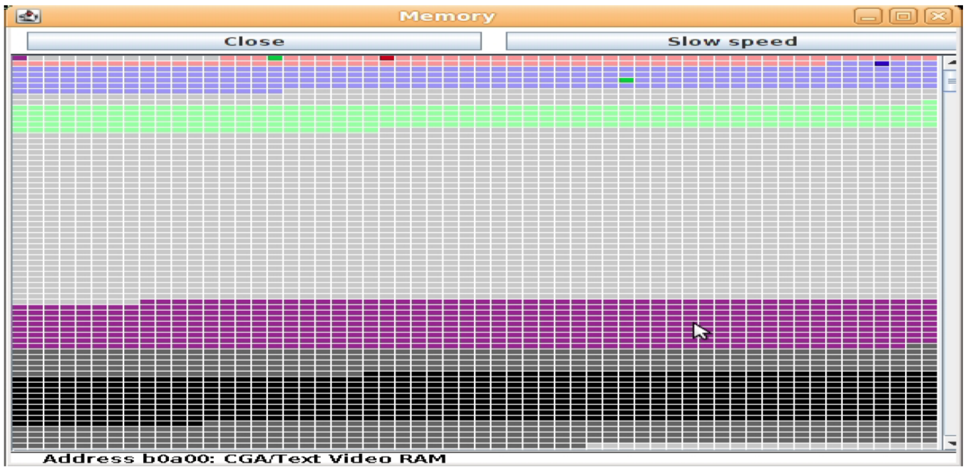
60000a31	{ xd2_shr.ns.d40s xdd0, xdd0, xdd1
60000a39	{ xd2_clamps.d40s xdd0, xdd0, 15;
60000a41	xd2_s.d16 xdd0, a1, 4
60000a44	l16si a7, a1, 2
60000a47	blti a7, 3, 60000a4d <overload+0x4
60000a4a	j 60000ae2 <overload+0xe2>
60000a4d	l16si a11, a1, 2
60000a50	l16si a10, a1, 0
60000a53	add.n a10, a10, a11
60000a55	addi.n a10, a10, -1
60000a57	bgei a10, 4, 60000a6c <overload+0
60000a5a	l16si a13, a1, 2
60000a5d	l16si a12, a1, 0
60000a60	add.n a12, a12, a13
60000a62	addi.n a12, a12, -1
60000a64	sext a12, a12, 15
60000a67	s32i.n a12, a1, 44
60000a69	j 60000a80 <overload+0x80>
60000a6c	movi.n a15, 4
60000a6e	l16si a2, a1, 2
60000a71	l16si a14, a1, 0
60000a74	add.n a14, a14, a2
60000a76	addi.n a14, a14, -1

2.2.3 Memory

Memory is perhaps the most important visualization element: It contains inputs, current state, outputs, and all evidence of useful computation. There are an unlimited number of possible ways to view and interpret memory; handling all of them in a tailored fashion is infeasible. However, a large set of commonly needed interpretations are known and many data visualization and organization techniques can be applied directly or indirectly through preprocessing. A few of these include: representing many scalars visually (described below), zooming, lensing, or filtering data to highlight active or interesting data among very large sets, and temporal processing for statistics, trends, vector fields, and multidimensional analysis for hotspots and correlation.

In a typical operating system and application environment, data structures and convention determine the high-level use of each area of memory. This can be used to provide a first-pass segmentation of use and type, which also bounds further interpretation. In CAS systems, this is seldom exploited and not yet to the level possible. A good guide here is the capabilities inside and outside a Linux kernel to determine what type memory is and what it is being used for. The Linux kernel, for instance, uses a type tag as a first field in every kernel data structure to prevent crashes due to corruption and for general debugging. In an emulator, memory usage could also be tracked and typed by an advanced system.

Figure 2.18. Graphical PC Simulator Memory(Black and Komala, 2011)

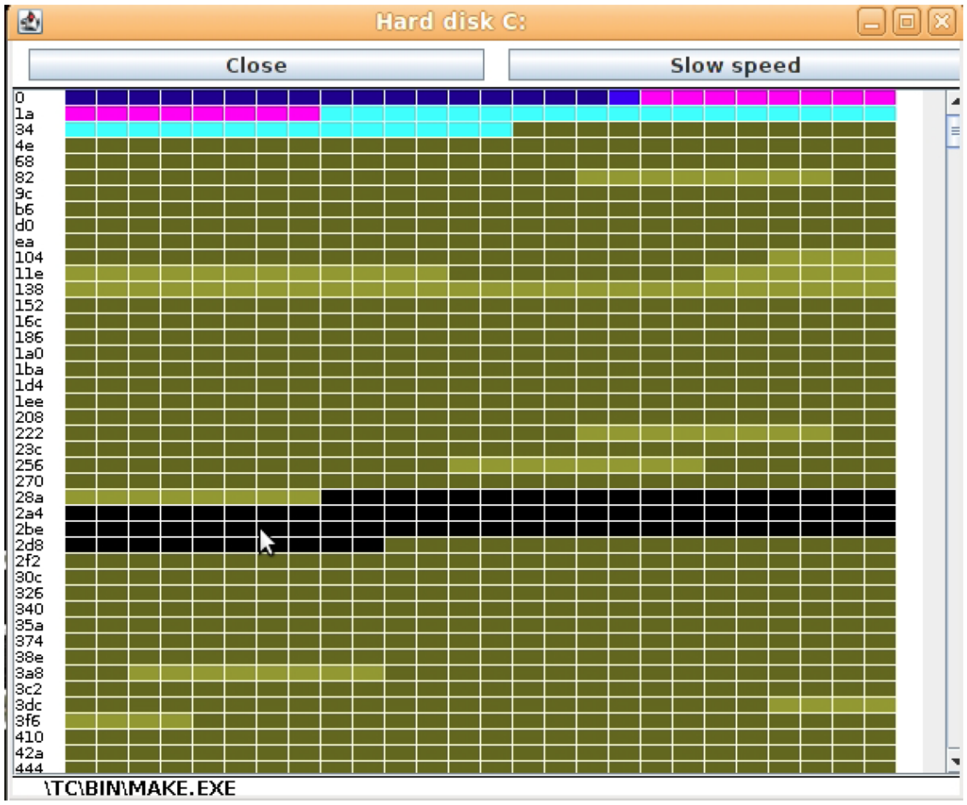


Address b0a00: CGA/Text Video RAM

2.2.4 Storage

External storage, such as floppy or hard drives, represents a memory-like interface with expensive and important latency, sequencing, and reliability concerns. Memory visualization methods apply to this data, along with knowledge of specific filesystem formats which further constrain the overall structure.

Figure 2.19. Graphical PC Simulator Hard Drive(Black and Komala, 2011)

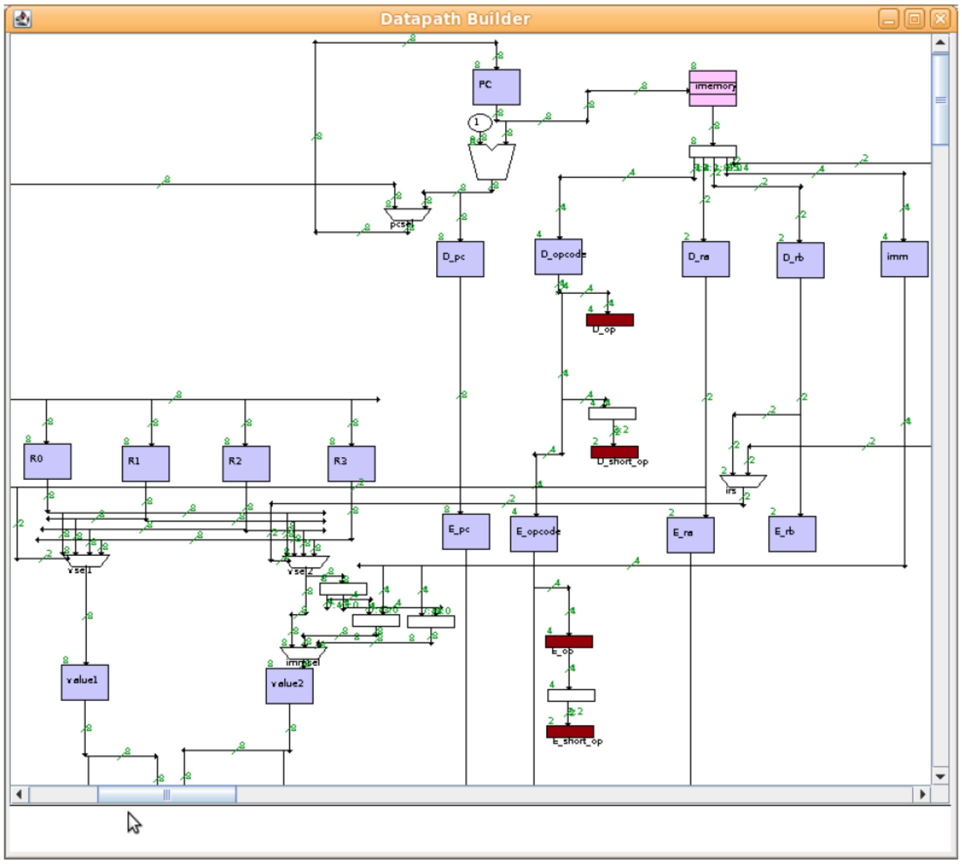


2.2.5 Data Path / Control Path

In a CPU architecture simulator, the data path and control path definitions of a CPU or CPU-like device often take the form of a node-link graph and a table of activation lists, respectively. The former has all of the problems of a node-link graph, namely difficulty of understanding and navigation as soon as a graph reaches even moderate size. Similarly, a large control path definition set will have the problems of a large spreadsheet: seemingly endless scrolling and difficulty in seeing patterns.

Both of these can be improved by a number of graph, lens, and auto-arranging relational and grouping mechanisms. Probably several methods should be usable simultaneously to support different views and purposes.

Figure 2.20. Graphical PC Simulator Data Path Definition(Black and Komala, 2011)



2.3 Interactive Visualization Design

Every emulator examined had hard coded, directly wired visualization implementations. None had alternate or pluggable visualizations and none could be easily modified or upgraded without invasive development. Most emulators other than EmuMaker 86 simulator were severely lacking support for many common and important peripherals and key subsystems.

Few developers are going to have the time and effort available to spend on modifying a poorly architected emulator interface, especially if there is a probability that a new version will obsolete their changes. A properly architected emulator engine delegates visualization to an implementation of an interface. This interface is provided with emulation events such as reading and writing memory, register changes, and I/O port events. There could be a null implementation to allow for maximum speed or a complex implementation that maps into a 3D OpenGL environment.

Each major emulator engine module, for each device type for instance, has an independent event interface. These could be satisfied with many independent emulation class instances, or just one class that implements all interfaces.

2.4 Visualization Integration Architecture

When emulators did include visualization mechanisms, they were generally found to be integrated into the engine in a monolithic, non-replaceable fashion. This greatly limits the flexibility of these emulators and places an almost insurmountable burdon on an academic researcher that may only have a short timeframe to focus on a key problem. The pluggable Processing interactive visualization approach solves this and supports open, lightweight development of CAS visualizations.

CHAPTER 3

METHODS

The methods used in this research are grounded in concrete development of a pluggable emulation framework by modification of a computer architecture simulator, using that emulator and examining other existing emulators to develop a critique, considering related problems and well-known human abilities and suitable methods, and synthesizing solutions. These solutions answer the requirements and critiques while leveraging proven solutions to efficiently enable and empower users with effective interaction and visual design models.

3.1 Curating Constraints and Concerns

Visuospatial thinking research has provided some valuable insight into human capabilities, identifying both a commonly shared baseline and characterizing variability. While often pointing to areas where further research is needed, current knowledge includes some immediately usable principles that help to constrain solutions in certain ways. This helps avoid exploration that conflicts with these known limitations and abilities.

Users have a number of learned mental models that can be applied metaphorically, and they may have a preconceived notion of how these should combine in a particular system. Conversely, they may have no substantial mental models for a particular situation. A system needs to indicate some degree of implicit and explicit information that helps a user determine what mental models to use when interacting with system. When possible or necessary, a system should educate the user about useful mental models. Ideally, this includes externally representing a construct that can readily map to an internal mental model through the use of stable, attractive, tunably dense representations that have logical, consistent, and powerful interaction models.

3.2 Cataloging Relevant Design Elements

Through emulation and simulation product, project, and literature search, and analysis and investigation of analogous fields, a range of successful interactive visualization methods were identified. These were filtered for estimated likelihood of applicability and rationalized into a progressive thread.

3.3 Identifying Key Literature

Key literature was identified and filtered through several methods: frequent citations, original source of widely used methods, exceptional explanations of key points and/or comprehensive compendium of alternative views, methods, or aspects of an area.

3.4 Determining Requirements & Missing Features

Through use of available emulators or consideration of published but not available emulators, required and desired actions were noted, visible data was observed, and experience was gained about desired but unavailable information. Interviews with some emulator users was also used. By considering both the visually available information and interfaces, and working back from a range of expected user goals, gaps in information and missing features were identified. Examples of information gaps and features to resolve them include lack of different granularities, temporal statistics, or interpretation type of data.

3.5 Critiquing Computer Emulation Systems

The EmuMaker 86 simulator emulator was used to boot an operating system, run programs, and access the virtual floppy and hard drive. Memory can be examined, instructions run interactively, and the virtual monitor and keyboard allow interaction with the running system. New logic, including defining processor architecture, can be defined by the use of datapath definitions and rulesets. Other emulators, including JSLinux, JavaPC, and QEmu, were booted, run, and examined at the source code level. Based on this experience, user interfaces for a broader range of emulators was interpreted based on published documentation and screen shots to determine behavior and visualization capabilities.

3.6 Modifying Emulator for Pluggable Visualization

The Java applet-based EmuMaker 86 simulator was modified to support pluggable visualization classes written for the Java-based Processing(Reas and Fry, 2007) "sketch" environment. This was a difficult process as the visualization integration architecture did not anticipate pluggable visualizations or multiple visualizations switching or running in parallel. After all emulator classes were modified to use an injection-pattern interface class for all visualization-relevant events, the engine and UI classes could be separated. Next, support for multiple simultaneous Processing sketches, with one using OpenGL and 3D potentially, was integrated and wired into the emulator module structure.

3.6.1 Designing Reference Emulator Integration

Debugging the emulator itself without a reference emulator is very difficult. A number of techniques were designed to facilitate the use of an existing reference emulator to facilitate debugging and evolution. These techniques require significant effort to implement but they allow efficient iterative development of the emulator where solving each problem can be exceedingly time consuming.

3.7 Developing New Design Elements

Based on analysis of the problem, existing practice, and considered methods, the requirements and components of potential designs are detailed. This provides a rich source of research and development opportunities within a broad conceptual framework. By considering this background and the identified problems,

3.8 Implementing Interactive Visualization Elements

An existing Java sort algorithm visualization system(Gosling et al., 2002), originally targeting Java AWT environments, was used as the basis for a Processing-based visualization development testbed. Little effort was required to modify the sorting test harness code to use Processing graphics and text output and keyboard and mouse input. The sort algorithm's integer array is analogous to memory while instrumenting each instruction is similar to tracing each machine instruction in an emulator. The resulting visualization routines are suitable for use as initial implementations of emulator visualization plugins, although optimization for selectivity, speed, and memory usage may be required. The code for some of these visualization explorations can be found in Appendix B and the resulting output can be seen in 4.6.

3.9 Developing Emulator Scenarios

Developing emulator usage scenarios is a process of considering the user goals, anticipating their mental state and active mental models, and envisioning interactive visualization solutions that are feasible and meet the user need at each point.

3.10 Evaluating Usability

The evaluation of usability of existing systems was performed through an examination of the utility and range of features compared to the gathered and synthesized requirements and based on estimation of cognitive load and support of a particular interface. First, systems had to have an interface to show or allow interaction with an area. Second, these interfaces were evaluated based on efficient support for the expected range of user actions. For instance, for many tasks, a direct manipulation interface with clear meaning and feedback scores better than a form-based text entry approach. The resulting work product, consisting of many pedantic, repetitive, and mostly obvious observations, was mostly an intermediate result that is represented here by the chosen design solutions. These design solutions are described partially in terms of the problem they are addressing.

The improvement in usability of new design solutions was estimated through a combination of a similar evaluation process and through likely benefit analysis of the application of previously documented successful design strategies and elements. Scenarios were examined for sequences of likely intent, apparent information given certain mechanisms, hypothesized efficiency of certain methods, and new capabilities not present in most or any existing systems. A number of visualization experiments were isolated and tested with a small number of users to confirm novelty and improvements in usability and usefulness.

3.11 Determining Research Opportunities

Considering user requirements, gaps in information and missing features, and the many possible avenues for exploration, recommendations were made for the most interesting and the most enabling for future use.

CHAPTER 4

RESULTS

Each of these results feeds later research activities. Understanding the requirements of the system and the user perspectives involved, considering cognitive insights and successful design solutions, both implementation and solution design can be performed.

4.0.1 Constraints, Concerns, & Design Elements

Review of the literature for constraints, concerns, and relevant design elements is embodied by Section 1.4 Concepts and Background. This includes consideration of human cognition and perception, solutions for analogous problems, and related topics.

4.1 Requirements & Missing Features

Existing computer emulators have been used for computer architecture instruction and, in certain narrow cases, for software development. The user interfaces for these emulators are mostly directly functional, screen emulation and a basic representation of a running CPU, registers, memory, and peripherals. A critique of these existing technologies allows contextual description of shortcomings and opportunities. In support of that, a description of personas provides a framework for reasoning about features based on user role and experience.

4.1.1 User Requirements & Motivations

Users of computer architecture simulators typically have certain direct goals. They may also be aware of certain indirect goals, although some of these may not be apparent or understood initially. The following describes requirements from a top-down perspective:

1. A professor wants to impart computer architecture knowledge to a user. The most important computer architecture (Blanchet and Dupouy, 2012)(Comer, 2004)(Parhami, 2005)(Hennessy and Patterson, 1990) topics vary somewhat based on time and author emphasis, but include many shared core concepts.

> "Computer architecture is an area of study dealing with digital computers at the interface between hardware and software. It is more hardware-oriented than "computer systems," an area typicall covered in courses by the same name in computer science or engineering, and more concerned with software than fields known as "computer design" and "computer organization." The subject matter, nevertheless, is quite fluid and varies greatly from one textbook or course to another in its orientation and coverage."

(Parhami, 2011)

   (a) Basic Architecture: modules, representation of information.

   (b) Programming model and operation: instructions and microinstructions, processor, I/O.

   (c) Memory hierarchy: memory, caches, virtual memory.

   (d) Performance and parallelism: pipeline, multi-processor caches, superscalar.

   (e) Programming models.

   (f) Storage and peripherals

   (g) Interconnection networks

2. Users want to understand a computer system.

   (a) Computers involve massive spatial (memory) and temporal (instruction sequence) detail.

   (b) Obvious representations are not understandable beyond very small windows.

   (c) Views are usually very high level or very low level, usually without the ability to smoothly adjust granularity.

   (d) There are few available, successful, and usable implementations of complex information display, navigation, and interaction.

   (e) Existing implementations are custom, narrow, and usually not flexible.

3. Users want to understand software. Visualization of systems and algorithm visualizations (AVs) is highly desirable by both educators and students for this purpose.““An important conclusion from the literature is that to make AVs pedagogically useful, they must support student interaction and active learning.””(Shaffer et al., 2010, 2)

   (a) How does software work?

      i. Applications, of all types

      ii. Operating systems, in various aspects

      iii. Device drivers, different types and patterns of usage

   (b) What does it look like at different scales?

   (c) What are the loops and sequences of instructions?

   (d) What does an algorithm look like when executing from various viewpoints?

   (e) How is data flowing through a system?

   (f) Is software as efficient as it can be?

4. Users want to debug a problem.

   (a) Where is it happening? What memory and code is involved?

   (b) When is it happening? After how many instructions? Repeatable? Related to I/O?

   (c) Need to isolate what is involved and important.

   (d) Need to understand what algorithms are involved.

5. Users want a system that is easy and efficient to use.(Norman, 2002)(Alexander, 1979)(Lidwell et al., 2003)

   (a) A system should remember their preferences and their place.

   (b) A system should have easy to understand and operate controls.

   (c) Displayed information should be concise and as complete as possible without being overwhelming.

   (d) Sequences of operations should be able to be recorded and replayed easily.

   (e) A system should remember what a user needs help remembering.

   (f) A system should be easy to extend and should evolve easily so that it continually improves without undue effort.

4.1.2 Mental Models

Mental models describe how someone conceptualizes some aspect or aspects of the system in a way that allows them to understand, reason about, and effectively interact with the system. Some example mental models include:

1. Understanding processor architecture in the form of datapath & control state machine rules or as instructions made up of microcode supports an intuitive understanding of the range of possible implementations and their implications.

2. Memory as a 2D spatial field, changing through individual instructions or via DMA, with tiered cache memory. This is the basic memory model.

3. Memory as typed data, changing over time, with dependency on prior operations and other memory locations and having some cost to access. This is the somewhat higher level view.

4.2 Pluggable Visualization

The Java-based EmuMaker 86 simulator emulator was first built with a Java Swing user interface. Each major subsystem, such as memory, the CPU, display, keyboard, hard drive, timers, etc. was implemented as a Java class that directly collaborated with a GUI implementation class. Each visualization could be enabled by the user, arranged in a scroll view. This provided a fairly traditional, although unusually comprehensive user interface. The problem with this approach is that the visualization and interaction was tied directly to the emulation engine component that it was modeling, making evolution and alternate visualizations difficult.

4.2.1 Adapted Application architecture

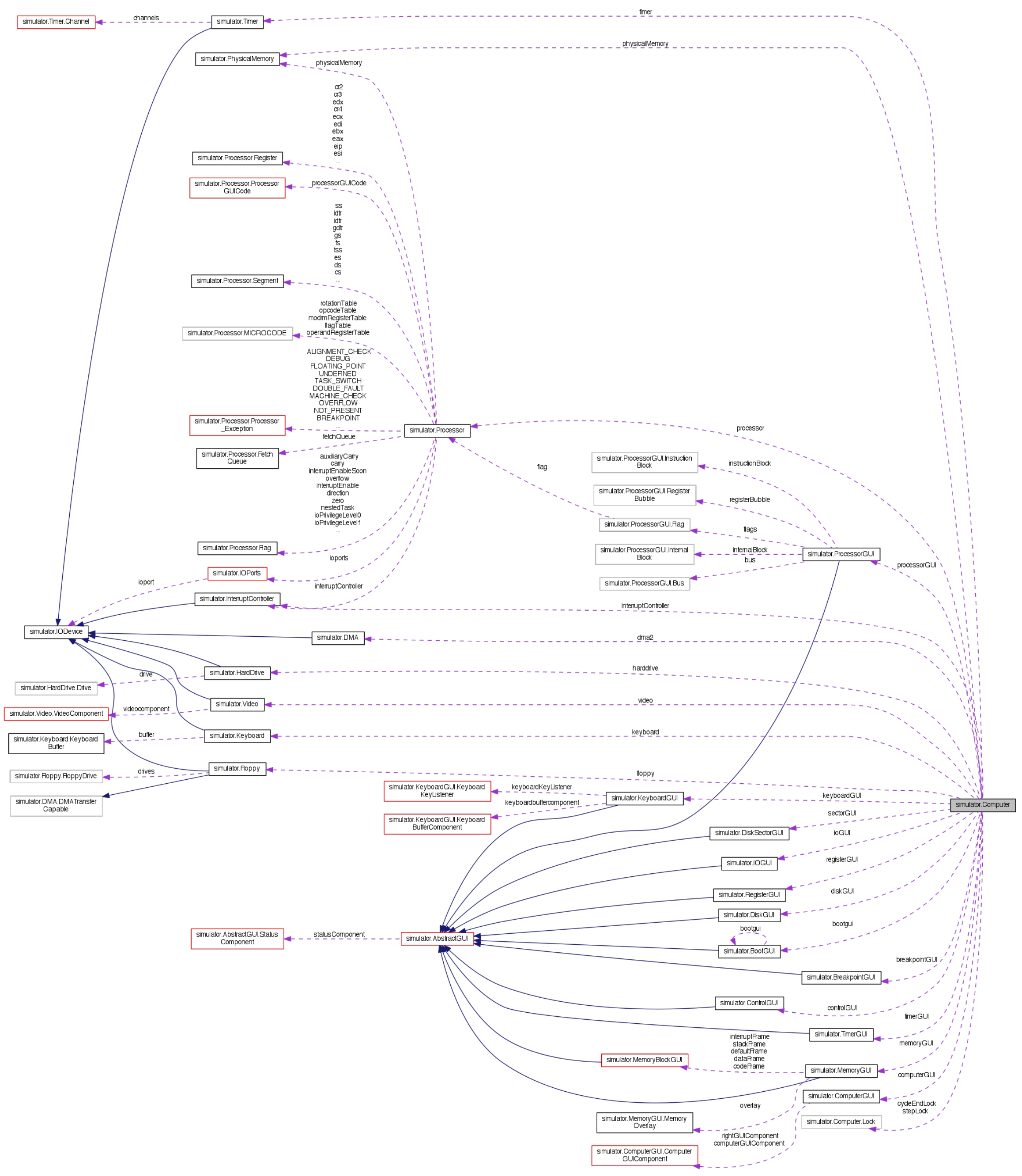
This architecture was modified to first separate all user interface and interaction mechanisms from the emulation engine components. This was done by implementing an injection pattern callback which implements methods to receive all of the key events each component can produce. This allows the engine to have no UI dependency, allowing it to support zero or more visualization and interaction implementations, even including a web-based front end or a scripting interface.

This pluggable interactive visualization capability was implemented through the use of the Processing(Reas and Fry, 2007) visualization system by integrating the Processing library into the applet-based EmuMaker 86 simulator. To support multiple simultaneous sketches, the giCentreUtils(Wood et al., 2013) library is used to support multiple simultaneous visualization sketches. A notable current limitation of Processing and current systems is that only one sketch at a time can make use of OpenGL for 3D modeling.

"Pluggable" means that the application could enable/disable, change, or chain visualizations easily at any time while the emulator is running. This supports direct control by the user of the current mode while remaining as efficient as possible. This method of pluggable visualization injection was not found in the computer emulators examined. A substantially similar mechanism would need to be present to meet the requirements of this problem, namely: flexible compile or runtime switching or simultaneous use of a visualization event interface by an emulation engine.

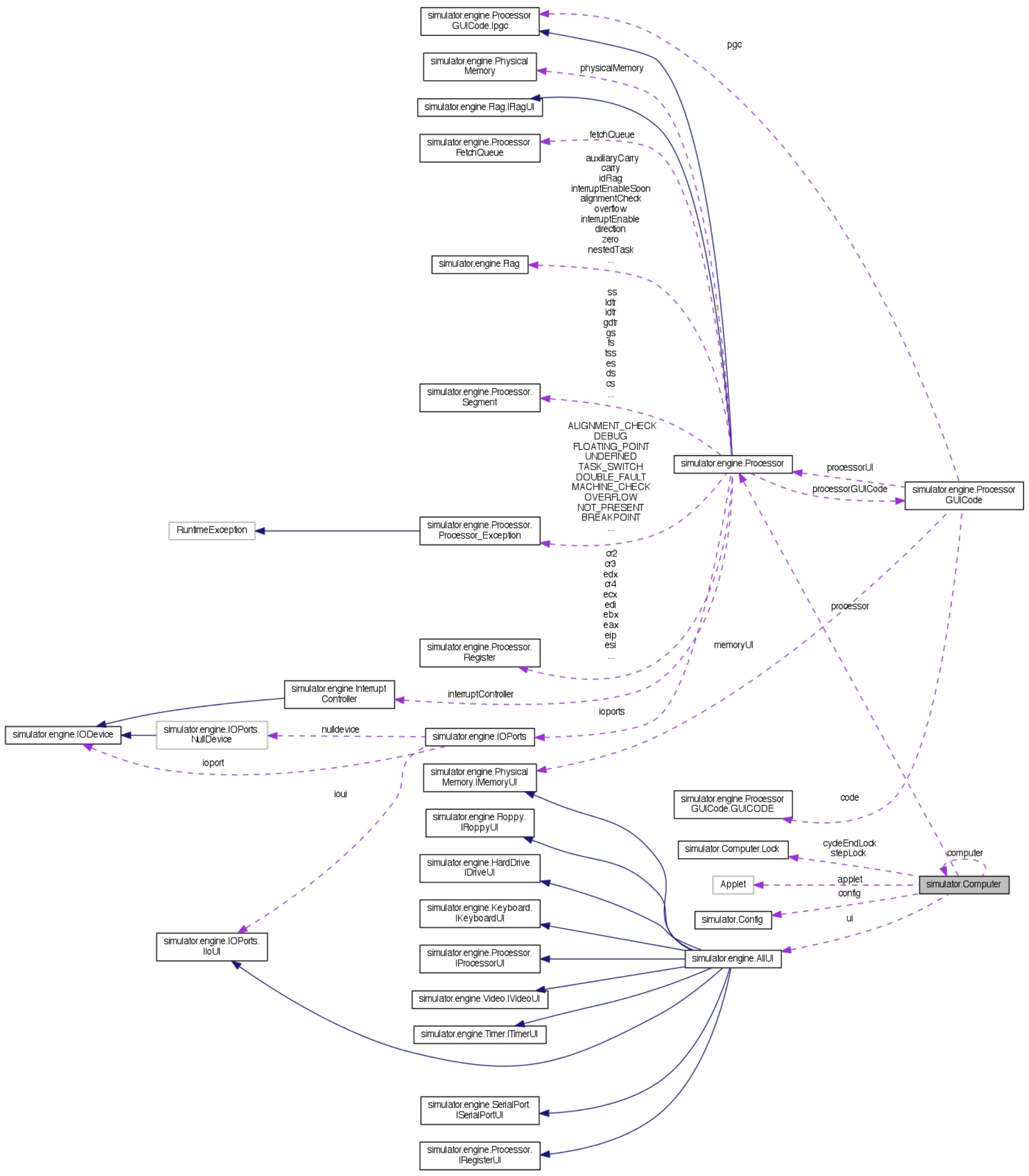
Figures 4.1 and 4.2 illustrate the before and after for simulator.Computer, the top emulation controller. Implementing the engine module interface and a matching visualization implementation of that interface is relatively straightforward, as illustrated in Appendix A. Modifying an existing emulator with intertwined visualization using Java Swing to a model that strictly separates engine from visualization, and supporting multiple Processing visualizations simultaneously is more involved. The resulting system, which is an adaptation of EmuMaker 86, is available as AmeriSim as noted in the appendices.

Figure 4.1. Collaboration Diagram for simulator.Computer, Before Adapting to Pluggable Visualization.



Note that in 4.2 the class structure is much less complex and engine and visualization classes have minimized interconnections.

Figure 4.2. Collaboration Diagram for simulator.Computer, After Adapting to Pluggable Visualization.


simulator.engine.Processor GUICode.Ipgc
pgc
simulator.engine.Physical Memory
physicalMemory
simulator.engine.Flag.IFlagUI
simulator.engine.Processor. FetchQueue
fetchQueue
auxiliaryCarry
carry
idFlag
interruptEnableSoon
alignmentCheck
overflow
interruptEnable
direction
zero
nestedTask
...
simulator.engine.Flag
ss
ldtr
idtr
gdtr
gs
fs
tss
es
ds
cs
...
simulator.engine.Processor. Segment
ALIGNMENT_CHECK
DEBUG
FLOATING_POINT
UNDEFINED
TASK_SWITCH
DOUBLE_FAULT
MACHINE_CHECK
OVERFLOW
NOT_PRESENT
BREAKPOINT
...
simulator.engine.Processor
processorUI
processorGUICode
simulator.engine.Processor GUICode
RuntimeException
simulator.engine.Processor. Processor_Exception
cr2
cr3
edx
cr4
ecx
edi
ebx
eax
eip
esi
...
processor
memoryUI
simulator.engine.Processor. Register
simulator.engine.Interrupt Controller
interruptController
ioports
simulator.engine.IODevice
simulator.engine.IOPorts. NullDevice
nulldevice
simulator.engine.IOPorts
ioport
simulator.engine.Physical Memory.IMemoryUI
ioui
simulator.engine.Processor GUICode.GUICODE
code
simulator.engine.Floppy. IFloppyUI
simulator.Computer.Lock
cycleEndLock
stepLock
computer
simulator.engine.HardDrive. IDriveUI
Applet
applet
simulator.Computer
config
simulator.Config
ui
simulator.engine.Keyboard. IKeyboardUI
simulator.engine.IOPorts. IIoUI
simulator.engine.Processor. IProcessorUI
simulator.engine.AllUI
simulator.engine.Video.IVideoUI
simulator.engine.Timer.ITimerUI
simulator.engine.SerialPort. ISerialPortUI
simulator.engine.Processor. IRegisterUI

4.2.2 Reference Emulator Integration Design

These are the main design elements that would be created to provide a high-quality, effective capability for rapidly evolving an emulator to match the emulation engine capabilities of an existing emulator. Absent this integration and these features, such evolution is slow and painstaking.

1. *Integration of a high-functioning emulator.* The chosen candidate was JSLinux(Bellard, 2013), which is a Javascript PC emulator, normally running in a web browser, which can boot Linux by retrieving device blocks over HTTP. Also provided is an ANSI C compiler, a full terminal emulator in Javascript, and a custom mini-Emacs clone. While the source is readily available in public, the author has not agreed to distribution by others, so beyond debugging and observation, utility is limited. Other options include QEMU(Bellard et al., 2013) and BOCHS(Assorted, 2013a). These are advanced enough to boot and run Linux, and they are open source. JSLinux may be an interesting choice as it is compact, efficient, runs in web browsers, and is paired with a minimized Linux, C compiler, etc.

2. *Fast, lightweight tracking and summarization of memory*, register, and state content through the use of hierarchical hashes, such as a Merkle tree(community, 2013b) and delta compression.(Sayood, 2002)(Suel and Memon, 2002)

3. *Visualization of memory values by summaries*, including identicons(Wikipedia, 2013d), and indication of change over time and change since a certain point through the use of areas of color, animated glow, and similar.

4. *Visualizing memory type and change age* based on whether memory was written recently, whether it was stored as a byte, short, int, float, or double.

5. *Implementing an efficient multi-checkpoint mechanism* for both emulators, allowing fast switching to a previous state. Since some types of instructions and situations may not occur for perhaps millions of instructions, a checkpoint is a crucial mechanism for thorough debugging. Checkpoints are widely used in supercomputing simulations for a similar reason: allowing restart while avoiding long waits to get to a certain point.

6. *Automatic execution of the EmuMaker 86 simulator emulator and the comparison emulator*, using an increasing resolution slow-start like sequence: If the first N instructions cause matching state, then run N*2 instructions in fast, unmonitored mode before causing a state comparison. When a mismatch is found, start from the last checkpoint and run until the midpoint of the gap, then repeat on the upper side if successful and the lower if not. This is a combination of Newton's Method and the slow start strategy, a widely known design element, notably used in the design of TCP/IP.

7. *Implement the emulator equivalent to the Unix 'make -k'*, which continues compiling remaining files even after receiving a compiler error. One way to apply that here is to generate a patch for that instruction number that brings the state of the emulator under debugging into alignment with the reference emulator. This allows finding numerous deviations in a single pass, potentially improving the fix/compile/debug cycle.

8. *Determining what differences are notable and which are not.* Emulators are well known not to be totally accurate with respect to real world processors most of the time. Security and other software can nearly always create a fingerprint of a particular processor or environment, detecting differences present but not normally relied upon by published operating systems and applications. Even on actual hardware,

crucial results can significantly vary due to differences in operating system, compiler, and hardware implementation of round-off error and similar.(Hong et al., 2013)

It is also possible to create a self-monitoring bootstrap instance that, once at least a small set of instructions run sufficiently, could provide similar state checking to an external monitor such as the emulator under debugging. This bootstrap state monitor could run in a virtual machine meant for production use or on actual hardware, either directly or in an operating system container.(Brokmeier, 2010)

4.3 Solution Design Elements

For a CAS context, there are a number of fundamental functions, such as representing scalar values, and some that are both fundamental and subject to many possible features and permutations, such as memory and instruction interactive visualization. These functions and features can be addressed by combinations of design elements. A design element could consist of any aspect of the implementation, function, and look of a visualization implementation. The fundamental aspects of design elements have been addressed in various ways with mixed results.(Saraiya et al., 2004)

4.3.1 Representing Scalars

Scalar values, bits and bytes and sequences in memory, represent the fundamental data building blocks of a system. Everything that happens in a system consists of data and action. Understanding the representation, value, meaning, and effect that data has on other data and running software is a key part of helping users to understand systems and software.

Typically, a scalar value such as an integer is represented as a hexadecimal or decimal number string of characters. A float or double is represented as a scientificnotational number string. While this is precise and useful as one mode, it fails to provide a visual representation that can clearly represent a rapidly changing value, history, correlation with other values, or scalable compactness. This is a problem when there is a need to rapidly understand approximate or exact values with limited screen real estate, rapidly changing values, or large numbers of similar values. There is presently a gap between character based representation and aggregate pixel-based images. The following techniques, alone or in combination, can provide subsets of these characteristics. The effectiveness of this approach will generally be apparent and it is highly measurable by testing speed and accuracy of interpretation of values. For instance, in the visualization detailed below, the angled line can be perceived to have many more distinct values than an 8-pixel high bar graph.

Numbers can be represented as:

1. Color values on a pseudo-color spectrum.

2. Bar chart shape.

3. Line and dot pattern, indicating high/low bytes or similar.

4. Line at an angle, perhaps also with width or shape of endpoint. The human visual system has high sensitivity to discontinuities in certain types of repeating patterns.

5. Motion according to value: higher value could be faster vibration. Angle, phase, and oscillation could encode bits.

6. Value or history as points on a 3D surface.

7. Hashed to an image such as an Identicon. (Wikipedia, 2013d)

8. Sparklines(Tufte, 2013) of past values, and whether the value is cyclical or apparently random. The latter can be done by alpha-blended value traces or animation cycles.

Many of these methods can be applied to fields of numbers are various levels of area summary to support useful information when zoomed out. Key information in this case includes distribution of values, average, mean, max, how many values have changed recently, etc.

Animation is a key technique. There are many variations which can highlight key information. Some of these are applicable when another algorithm has highlighted some values as noteworthy. Other methods allow the human visual system to be the primary pattern detector. Some of these methods include:

1. Cycling through recent or total historical values to illustrate information visually. This should normally be in sync with actual relative value changes of other memory cells or areas, allowing relationships to be seen.

2. A/B onion skin animation. This and cycling are commonly used to augment still photographs in a cinemagraph, while toggling between two stereo views provides a 3D effect with a 2D display system.

3. Vibration according to value.

4. Jiggling, vibrating values according to how recently changed.

5. Duty cycle based representation of number values.

6. Coordinated movement of multiple values associates them, even if they have different size, shape, and color.

Relationships between numbers can be very important. When these are deduced directly, by tracking dataflow, or to show how values or change in values is related, this can be shown graphically through similar representation, similar movement, or lines or background shading connecting values.

The user interface mechanism for switching between modes can take several forms. One of the more powerfully interactive is the use of a pseudo-realistic lens object. One or more lenses can be grabbed, stacked, and tuned, and then these can be swept over values to instantly and temporarily reveal alternate values.

4.3.2 Representing Memory

Memory needs to be viewable by a user in a number of ways, sometimes simultaneously. The main problems are: memory is too large to see anything significant in detail, most memory is uninteresting, important information may be widely scattered. Past solutions have mostly involved text, character, and sometimes image display of memory regions, usually only at the lowest level of granularity. A variety of solutions can be made available to the user to provide appropriate filtering, highlight, tracing, and type or source tracking, as detailed below. The effectiveness of these techniques can be measured by testing for comprehension of the state of memory and recognition of known and new patterns in a variety of circumstances and problems. These would include cases where a single value is watched, a series of values which are changing and the user is able to discern linkages that cause change patterns, and where whole fields of numbers are updated.

Key parts of the solution include:

1. Zoomable UI (ZUI) with appropriate level of detail summary

2. Filtering unused memory by default, with an easy toggle to view used memory in context.

3. Indication, probably by color saturation, of how recently memory was read or written.

4. Typed memory based on how memory is used. This requires multiple runs with the same program and inputs.

5. Pre-arranged used memory layout. This requires multiple runs with the same program and inputs.

6. Clear separation, coloring, and filtering of memory from the system and other processes.

7. Automatic and semi-auto (via dragging) typed memory display and rearrangement appropriate to a particular granularity level and known algorithmic process, probably based on an earlier run.

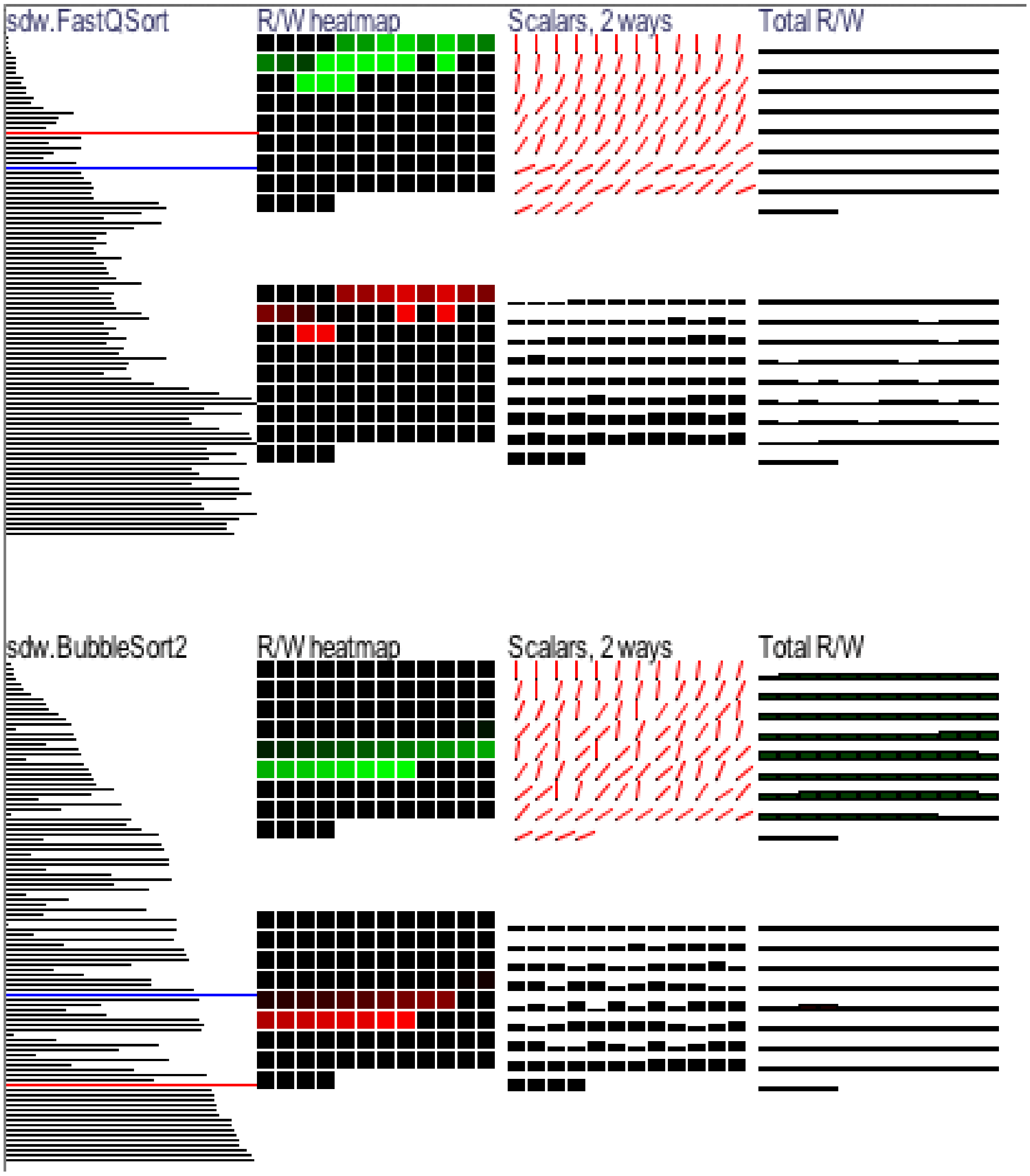
8. Dependency tracking can be added for particular memory, showing upstream or downstream effects. This can be visualized in a number of ways: either temporal heatmap changes or Sankey Diagrams are appropriate here.

Memory Visualization Experiments

A processing sketch was created based initially on a Java sort algorithm visualization system(Gosling et al., 2002). This was rewritten in minor ways to run in the Processing environment. The existing visualization, illustrating the actions of the sort algorithms over a series of integers, was augmented to display a heat map for recent read/write, to display the integers visually in two different ways, and to show graphically the number of times each memory location is read or written. 4.3 is a screen shot of the running simulation. This was done while two different sort routines run at the same time. This clearly illustrates the different memory allocation patterns, total speed of each algorithm, and vastly different number of reads and writes required. Clicking anywhere will cause either or both finished sort routines to be restarted. The up arrow key will add 1ms to the simulation delay while the down arrow key will do the opposite, allowing the user to control the speed of simulation. This simulation code can be moved to and integrated in the full emulator, with optimization.

Both novice and experience computer programmers and computer science students and professors provided positive feedback. They indicated that the heat map and the angle-based scalar representation were beneficial in clarifying operation of the example algorithms.

Figure 4.3. Memory Visualization Experiments



4.3.3 Representing Instructions

In addition to understanding data and memory at all scales, a user needs to observe instructions that actually cause changes in the system at a similar range of scales to fully understand a system.

There is a need for alternate representations of instructions for machine code and microcode to support faster, more efficient understanding of instruction meaning, effect, and overall flow of system and application code. Existing methods generally consist of displaying assembly code mnemonics to the user who may not have the long periods of training needed to immediately parse and grasp the meaning of particular instructions. Although the number of basic operations is small, the permutations for addressing modes and other variance tends to make instruction interpretation difficult. There are a number of promising opportunities for experimentation around improving this understanding.

KanjiCode

In an instruction set as complex as the Intel 386 and above, representing instructions only by their mnemonic, a typical existing practice, can be improved upon for certain purposes, such as rapid scanning of running code. There are a variety of alternatives which ought to be available for a user. One of the most promising is a design that incorporates a specially arranged logical CPU "stage" where when an instruction or set of instructions is plugged in, it is immediately clear what will happen at a certain level of detail. This approach can be tested for effectiveness by running code sequences by users of various experience and at various speeds and lengths and measuring recognition and understanding. This can be done at both the general instruction type level and for specific instructions and arguments.

At the lowest level of detail, instructions are represented by a Kanji(Tuttle, 1988) like set of radicals, where each radical may be a microcode instruction. By carefully designing the radicals and their placement, and matching components of the stage, a representation of the activation paths of the CPU is apparent.

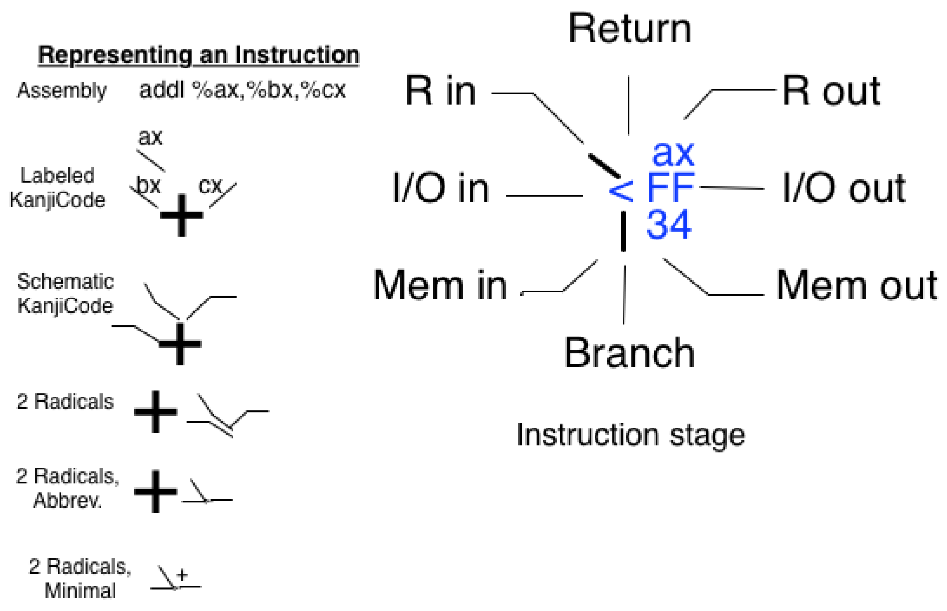
For instance, if the instruction Kanji has a load area, a save area, an operations area, and operands, then a load instruction would be blank for the operations area. Well known symbols could be used for +, -, *, etc. In 4.4, six different options for display of instructions appear on the left. These range from the traditional instruction style to increasingly stylized schematic representations. The assembly instruction in that diagram might mean: "Add long from ax and bx, storing the result in cx." Each of the following representations have the same meaning but increasingly simplified and stylized form. In the last four examples, the angle of the line segment tail indications which register is involved.

At higher levels of granularity, this instruction Kanji can be summarized in a variety of ways. The simplest is a weighted combination of the opcode microcode radicals. Another would be a 3D representation of the temporal depth-stack of instructions something like a key in the CPU ALU stage. A third would be a representation of the loops and sequences and memory interaction which could be symbolically summarized in a way that could be drilled into.

A key visualization method for understanding how a system operates and what it accomplishes is to create an animated sequence that shows data flow and transformation. These would be either steady state or storyboard sequences depending on the code in question. The operating system would tend to be a steady state, cyclical system.

In natural languages such as Japanese and Chinese, the Kanji system is somewhat of an anti-pattern as those languages tend to require far more effort to master than alphabet-based languages. However, it is apparent that much of this is due to the fact that natural language Kanji have been formed through various types of historical evolution rather than a coherent design. In fact, native speakers of Japanese are not regularly taught the etymology of Kanji which in some cases provides logical support, but are rather taught mostly through rote memorization. Considering the proliferation and successful use of cross-cultural icons used for user interfaces, signs, and other purposes, it seems likely that a well-designed Kanji-like system might be efficient.

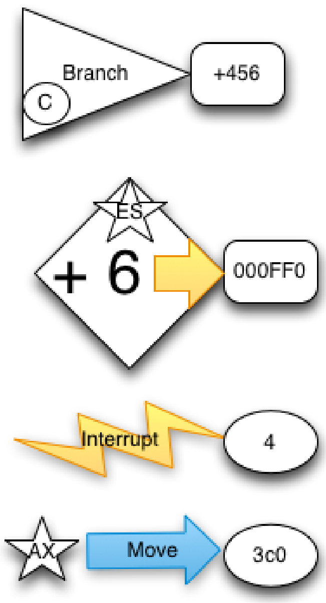
Figure 4.4. KanjiCode Example



ShapeCode

Another solution to instruction representation involves variations on 2D and 3D shapes to represent instructions, possibly including vibration or shape change or other features. 4.5 illustrates examples of graphical relative branch, register add immediate with store to memory, interrupt, and register to memory move instructions. Instruction types, addressing modes, and any other important attributes could be translated into a shape component language. Each instruction would be represented by a combination of shapes in a particular, distinct way. This makes use of the human object recognition capability and has the advantage of supporting abstract, instruction class at low resolution and detailed, exact instruction and argument at higher resolution. Using color or other themes, the structure of code could be visible at a glance. This approach can be tested for effectiveness of code and structural understanding at different resolutions and different speed of presentation.

Figure 4.5. ShapeCode Example



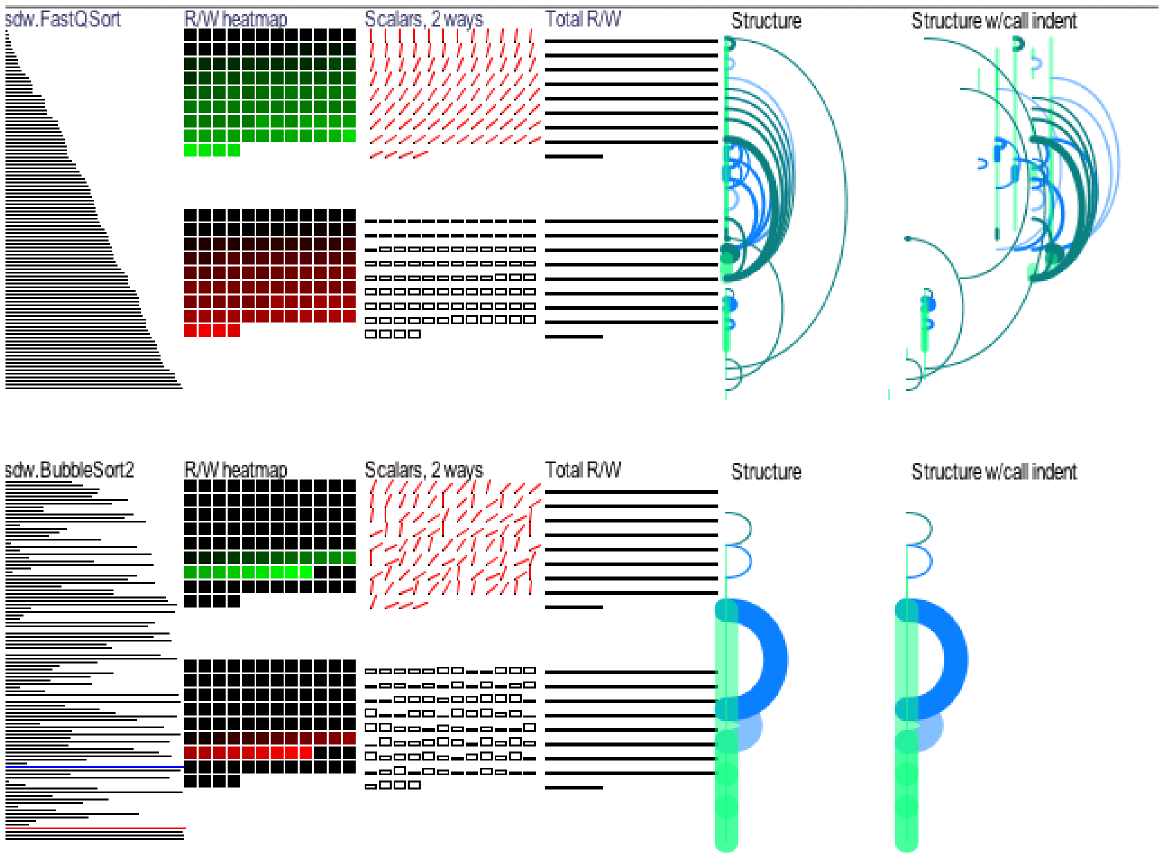
Code Structure Visualization

The structure of code can be inferred statically or dynamically through instruction or data flow trace. This is usually done manually by users although there exists some degree of automation, not widely available for the CAS context. Once the structure of code has been determined, a variety of visualization techniques can illustrate layering, functional relationships, commonality, actual and possible flow of control, instruction type ranges (IO, privileged, etc.), and processing time hotspots. Testing implementations of code structure visualization would involve measuring how effective and complete the code representation is and how well users are able to understand what is being conveyed. Ideally, this would work and be testable at different levels of granularity.

Key visualization methods for this problem would include both 2D and 3D methods. A promising solution is to contrast linear sequences of instructions with branches, subroutine calls, and system calls or interrupts. These could be drawn as linear lines, arcs, lines to and from parallel instruction graph tiers, and call/return 'up', respectively. An important feature would be to draw these traces with a partial alpha channel so that repeated traces would tend to reinforce repeated traces while showing alternate paths for unusual branches. This historical processing could fade based on age to allow tunable ongoing representation of the current execution path along with a certain degree of past history. In 4.6, the rightmost two columns show these techniques, with and without subroutine call indenting. This clearly shows the structure of branches, loops, function calls, and sequences of instructions, along with color saturation and the width of lines indicating relative percentage of execution time used. Being able to 'scrub' this history in real-time by dragging the mouse or touch pointer over the visual trace history is a very powerful technique, as in the Layer-Time visualization technique, as shown in 1.13.

The code structure visualization in 4.6 grows gradually as the algorithm runs. Here, the location of each branch, loop, or function call endpoint is fixed, allowing the sequence of actions to be related to specific code areas as the algorithm progresses. Many specializations of this can be made to highlight various aspects of an algorithm or system, such as function call overhead, or code complexity. The implementation, shown in Appendix B, shows how the visualization generically models statements, branches, loops, and function calls. This module could be used for instrumented Java or C/C++ code, or machine instructions could be mapped to it through the emulator engine for showing similar software architecture: The visualization will look the same for the same algorithm regardless of the language or level of instrumentation. In this specific example, the simple nested loop nature of bubble sort is clear, along with saturation and line width illustrating how many more instructions are needed compared to quicksort. The FastQSort implementation of quicksort shows additional code complexity while illustrating the minimal number of instructions needed to complete the sort much faster than bubble sort. Too-simple is not better when it comes to sort algorithms.

Figure 4.6. Code Structure Visualization



4.4 Usability

The usability improvements of these methods are hypothesized to be significant based on new features enabled, rational application of principles known to work for analogous problems, scenario walk-through to validate need and utility, and visualization experiment implementation and evaluation. These hypotheses are supported by design principles, rational estimation given visuospatial background knowledge, and a small amount of experimentation. Coupled with the generative and constraining knowledge gathered here, a wide range of experimentation and experience is supported which should lead to jumps in usability. Additionally, the enablement of a highly modular, multi-author multi-sketch visualization environment in an emulation context can be expected to provide the same general open evolution benefits experienced by other highly open systems such as Android, Java, Linux, and the Internet in general.

CHAPTER 5

CONCLUSIONS

5.1 Summary of Findings

The overall conclusion is that many opportunities exist to greatly improve the features, usability, and effectiveness of computer architecture simulators for pedagogical and software development needs. Successful application of these findings may lead to better educational and problem-solving efficiency. Both in relying on existing experience that has shown successful application and through hypothesizing design solutions to specific problems, implementing the best of these, and measuring early results, progress has been shown toward better usability.

One of the best ways to quickly evolve software is to foster many independent developers working to solve separate and overlapping problems. By moving from a monolithic to a pluggable visualization architecture using a popular and capable visualization environment, this open and parallel evolution becomes much more accessible and manageable. The updated EmuMaker 86 simulator is freely available, providing a base for open exploration among independent developers and students.(Black and Williams, 2013)

5.2 Conclusions Drawn

These scenarios represent the conclusions of analysis and design, providing contextual views of possible features and characteristics of an effective system.

5.2.1 Solution Scenarios

The use of these solution design elements is described in the context of a particular persona attempting to achieve their goals with respect to the CAS. This illustrates how they are used by a system to enhance the user experience in particular ways.

Persona: Professor

A computer science or architecture professor needs to impart knowledge and an intuitive understanding of concepts at all levels of abstraction. Listing facts is easy while imparting an internalized, intuitive, and useful understanding can be gradual and difficult. Employing a visually appealing and revealing computer architecture emulator can drastically accelerate this process by making the mechanics accessible and transparent.

1. CAS setup, run, and observation - Distribution, parameterization, and use should be simple and efficient.

2. Machine Instructions - microcode, parallelism, performance should be easily understood and accessible.

3. Memory - architecture, patterns of access, caching, alternate architectures, performance should have effective visualization.

4. Application programming & debugging - should be supported through emulator features such as single step, reverse, and snapshots.

5. Operating system programming & debugging

6. Device driver development & debugging

7. Forensics - Requires features such as change monitoring of memory ranges and stack and data segment corruption detection.

Persona: CS, CA &, LSWD

A computer science student or a learning software developer would come to a CAS with goals of gaining a deeper understanding of computer architecture, including hardware and software components, function, and how they work together. Such a student would be expected to have some knowledge of programming, probably mostly in a high level language. Knowledge of computer architecture, assembly language, and computer internals would vary greatly. Each assignment supports progressive proficiency with the CAS and concept understanding.

1. CAS is easy to access, run, and use. – Fast Java Applet, graphical window with subwindows for each major component.

2. Input/Output devices, including virtual screen, keyboard, mouse, floppy/hard-drive, and network devices, are obvious and have both physically inspired and symbolically modeled modes.

3. When possible, the operating system, drivers, processes, and threads should be identifiable in memory and with respect to what code is executing.

4. Instruction processing should be viewable from the instruction level down including microcode, pipeline stages, hazards, and buffering.

5. Instruction processing should be viewable from the instruction level up, including sequences, loops, jumps, functions, algorithms & idioms, and processes.

6. Instruction processing over time should be supported, showing progress through a sequence of instructions, variability should be apparent and scrubbable, and the user should be able to zoom in and out, have a lens effect view into the current focus in context, and have multiple focus areas to watch related function on different sides.

7. The system should support running forward and backward, using checkpointing plus bisection search rerun to a particular point, adaptive buffering, and trace summaries that can be played forward and backward.

8. The current session should be able to be saved in an efficient and shareable way so that it can be reloaded later or by others.

9. A sequence of actions should be able to be recorded, such as running with full visualization for a certain sequence of instructions with certain traps and event processing enabled.

5.3 Recommendations for Further Research

Many interesting problems, constraints, and solution ideas have been presented. Each of these can be explored within the context of a working computer architecture simulator with a pluggable, easily adaptable interactive visualization environment.

In addition to completing the reference emulator integration to support quickly evolving the emulator's capabilities, many visualization methods can be implemented and run quickly. With proper integration architecture, this can be accomplished with easily by modification of an existing visualization component. Some specific areas for promising research include:

5.3.1 Memory Visualization

A variety of visualization methods should be available to a user to support different application scenarios. These should include various scalar, image, temporal, and relationship methods, often graphically represented to the user.

5.3.2 Instruction, Microcode, and Pipeline

Instructions, microcode, and pipeline and cache state are all important but difficult to represent and understand in complex, high-volume situations. There are a number of different problem areas that will likely require independent innovation.

5.3.3 Code Understanding & Machine Learning

Significant progress has been made in analyzing source code and machine code in running systems to understand the organization and, in some cases, the underlying algorithms. Combining this capability with a CAS system allows deeper insight at various levels of granularity. This understanding can even be used to deemphasize or allow collapse of uninteresting areas to enhance focus on the rest.

Detecting Patterns

A variety of classification, pattern recognition, and machine learning methods could be applied to a CAS to provide insight into instruction and memory usage, algorithm understanding, user action intent (especially repetitive action sequences), data type inference, and identification of inefficient processing.

Gesture Sensing & Input Methods

In addition to mice, touch pads, touch screens, pens, and 6 degree of freedom input devices, several ways of tracking precise hand movement are becoming available. Face recognition and now eye tracking are becoming standard, which is about to provide simulated 3D views, based on your viewing angle. Additionally, head-tracking virtual reality displays such as the Occulus Rift provide an opportunity for 360 degree interactive and navigable space.

APPENDIX A

PLUGIN CODE

The following source code illustrates example visualization plugins for the computer architecture simulator. The emulator engine module interface determines what emulation events can be delivered while avoiding any knowledge of visualization in the engine. The visualization plugin implements this interface to store, display, or otherwise react to these emulation events.

This code will be available at https://github.com/sdwlig/AmeriSim (Black and Williams, 2013) and http://sdw.st/amerisim .

First, the Video.IVideoUI interface is defined:

```java
public class Video extends IODevice {
  public interface IVideoUI {
    public void rect(int c, int x, int y, int width, int height);
    public void drawText(int c, String text, int x, int y);
    public void repaint();
  }
```

src/Video.java

Then, that interface is implemented by the Video visualization, in Processing Java calls here.

```java
package simulator.gui;
```

```

import java.awt.*;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

import org.gicentre.utils.multisketch.EmbeddedSketch;

import processing.core.PFont;
import controlP5.ControlP5;

import simulator.Computer;
import simulator.engine.Video;
import simulator.engine.Video.IVideoUI;

public class VideoGui extends EmbeddedSketch implements IVideoUI {
  static final long serialVersionUID = 557;
  // ------------------- Sketch-wide variables ----------------------
  Computer computer = null;
  float textScale;
  static int XSIZE = 640, YSIZE = 480;
  static int MAXCOMPONENTS = 20, COMPONENTHEIGHT = YSIZE / 2 - 20;
  ControlP5 cp5;

  // ---------------------- Initialization -------------------------
  public void setup() {
      size(XSIZE, YSIZE);
      frameRate(10);
  }
  public void draw() {
    computer.video.paintScreen();
  }
  public VideoGui(Computer computer) {
    this.computer = computer;
  }
  // ------------------------ Processing draw ---------------------------

    public int width() {
        return Video.VWIDTH;
    }

    public int height() {
        return Video.VHEIGHT;
    }

    @Override
    public void rect(int c, int x, int y, int width, int height) {
      stroke(c); // c.getRed(), c.getGreen(), c.getBlue());
      fill(c);
      rect(x, y, width, height);
    }
    PFont mono = null;
    public void drawText(int c, String text, int x, int y) {
      stroke(c); // .getRed(), c.getGreen(), c.getBlue());
      fill(c);
      if (mono == null) {
        mono = loadFont("AndaleMono-12.vlw");
      }
      textFont(mono);
      text(text, x, y);
    }

}
```

src/VideoGui.java

APPENDIX B

VISUALIZATION MODELS

Some explored visualization models are shown in the following source code. The resulting visualization is shown and described in 4.3.3. Included here are bubble sort and fast quick sort. Additional sort algorithm visualizations can be found in (Gosling et al., 2002). These could be added to this visualization with minimal effort by a similar transformation.

This code will be available at https://github.com/sdwlig/AmeriSim (Black and Williams, 2013) and http://sdw.st/amerisim .

This is the Gosling BubbleSort2 algorithm, modified to fit in a Processing, multi-algorithm visualization environment. Key lines have been augmented with calls to sw.statement(), sw.branch(), etc. to support structure visualization.

```
package sdw;

/*
 * @(#)BubbleSortAlgorithm.java 1.6 95/01/31 James Gosling Copyright
 * (c) 1994 Sun Microsystems, Inc.  All Rights Reserved. Permission to
 * use, copy, modify, and distribute this software and its
 * documentation for NON-COMMERCIAL purposes and without fee is hereby
 * granted provided that this copyright notice appears in all
 * copies. Please refer to the file "copyright.html" for further
 * important copyright and licensing information. SUN MAKES NO
 * REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
 * DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
 */
/*
 * Changes Copyright 2013 Stephen D. Williams sdw@lig.net
 */

/**
 * A bubble sort demonstration algorithm SortAlgorithm.java, Thu Oct 27
    10:32:35 1994
 *
 * @author James Gosling
 * @version 1.6, 31 Jan 1995
 *
 *          Modified 23 Jun 1995 by Jason Harrison@cs.ubc.ca: Algorithm
    completes early when no
 *          items have been swapped in the last pass.
 */
class BubbleSort2Algorithm extends SortAlgorithm {
  void sort(SortItemP5.MemInts a, Sw sw) throws Exception {
    sw.called(1);
    for (int i = a.length; --i >= 0 && sw.loop(2);) {
      boolean flipped = false;
      sw.statement(3);
      for (int j = 0; j < i && sw.loop(4); j++) {
        if (stopRequested && sw.branch(5)) { sw.retrn(6); return; }
        if (a.get(j) > a.get(j + 1) && sw.branch(7)) {
          int T = a.get(j);
          sw.statement(8);
          a.set(j, a.get(j + 1));
          sw.statement(9);
          a.set(j + 1, T);
          sw.statement(10);
          flipped = true;
          sw.statement(11);
        }
        pause(i, j);
      }
      if (!flipped && sw.branch(12)) { sw.retrn(13); return; }
    }
  }
}
```

src/p5/sdw2/src/sdw/BubbleSort2Algorithm.java

Color bridges the differences between the AWT style of color type and the Processing method.

```
package sdw;
/*
 * Copyright 2013 Stephen D. Williams sdw@lig.net
 */

public class Color {
    public static int scolor(int red, int green, int blue) {
      return (255 << 24) | (red << 16) | (green << 8) | blue;
    }

    public static int scolor(int red, int green, int blue, int alpha) {
      return (alpha << 24) | (red << 16) | (green << 8) | blue;
    }

    public static final int black = scolor(0, 0, 0, 255);
    public static final int white = scolor(255, 255, 255, 255);
    public static final int red = scolor(255, 0, 0, 255);
    public static final int green = scolor(0, 255, 0, 255);
    public static final int blue = scolor(0, 0, 255, 255);
    public static final int yellow = scolor(0, 255, 255, 255);
    public static final int lightGray = scolor(20, 20, 20, 255);
}
```

src/p5/sdw2/src/sdw/Color.java

This is the Gosling FastQSort algorithm, modified to fit in a Processing, multi-algorithm visualization environment. Key lines have been augmented with calls to sw.statement(), sw.branch(), etc. to support structure visualization.

```
package sdw;

/*
 * @(#)QSortAlgorithm.java 1.3 29 Feb 1996 James Gosling Copyright (c)
 * 1994-1996 Sun Microsystems, Inc. All Rights Reserved. Permission to
 * use, copy, modify, and distribute this software and its
 * documentation for NON-COMMERCIAL or COMMERCIAL purposes and without
 * fee is hereby granted. Please refer to the file
 * http://www.javasoft.com/copy_trademarks.html for further important
 * copyright and trademark information and to
 * http://www.javasoft.com/licensing.html for further important
 * licensing information for the Java (tm) Technology. SUN MAKES NO
 * REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
 * DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. THIS SOFTWARE IS NOT
 * DESIGNED OR INTENDED FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT
 * IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS
 * IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR
 * COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT
 * MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF THE SOFTWARE
 * COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL
 * OR ENVIRONMENTAL DAMAGE ("HIGH RISK ACTIVITIES"). SUN SPECIFICALLY
 * DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR HIGH RISK
 * ACTIVITIES.
 */
/*
 * Changes Copyright 2013 Stephen D. Williams sdw@lig.net
 */

/**
 * A quick sort demonstration algorithm SortAlgorithm.java
 *
 * @author James Gosling
 * @author Kevin A. Smith
 * @version @(#)QSortAlgorithm.java 1.3, 29 Feb 1996 extended with
 *          TriMedian and InsertionSort by Denis Ahrens with all the
 *          tips from Robert Sedgewick (Algorithms in C++). It uses
 *          TriMedian and InsertionSort for lists shorts than
 *          4. <fuhrmann@cs.tu-berlin.de>
 */
public class FastQSortAlgorithm extends SortAlgorithm {
  /**
   * This is a generic version of C.A.R Hoare's Quick Sort
   * algorithm. This will handle arrays that are already sorted, and
   * arrays with duplicate keys.<BR>
   *
   * If you think of a one dimensional array as going from the lowest
   * index on the left to the highest index on the right then the
   * parameters to this function are lowest index or left and highest
   * index or right. The first time you call this function it will be
   * with the parameters 0, a.length - 1.
   *
   * @param a an integer array
   * @param lo0 left boundary of array partition
   * @param hi0 right boundary of array partition
   */
  private void QuickSort(SortItemP5.MemInts a, int l, int r, Sw sw)
    throws Exception {
    sw.called(1);
    int M = 4;
    int i;
    int j;
    int v;
    sw.statement(2);
    if ((r - l) > M && sw.branch(3)) {
      sw.statement(4);
      i = (r + l) / 2;
      if (a.get(l) > a.get(i) && sw.branch(5)) swap(a, l, i, sw); //
    Tri-Median Methode!
      if (a.get(l) > a.get(r) && sw.branch(6)) swap(a, l, r, sw);
      if (a.get(i) > a.get(r) && sw.branch(7)) swap(a, i, r, sw);

      sw.statement(8);
      j = r - 1;
      swap(a, i, j, sw);
```

```
        sw.statement(9);
        i = l;
        sw.statement(10);
        v = a.get(j);
        for (; sw.loop(11);) {
          while (a.get(++i) < v) {
            sw.statement(12);
            sw.loop(13);
          }
          while (a.get(--j) > v) {
            sw.statement(14);
            sw.loop(15);
          }
          if (j < i && sw.branch(16)) break;
          swap(a, i, j, sw);
          pause(i, j);
          if (stopRequested) {
            sw.retrn(17);
            return;
          }
        }
        sw.statement(18);
        swap(a, i, r - 1, sw);
        pause(i);
        sw.statement(19);
        QuickSort(a, l, j, sw);
        sw.statement(20);
        QuickSort(a, i + 1, r, sw);
      }
      sw.retrn(21);
    }
```

```
private void swap(SortItemP5.MemInts a, int i, int j, Sw sw) {
  int T;
  sw.called(22);
  T = a.get(i);
  sw.statement(23);
  a.set(i, a.get(j));
  sw.statement(24);
  a.set(j, T);
  sw.retrn(25);
}

private void InsertionSort(SortItemP5.MemInts a, int lo0, int hi0, Sw
  sw) throws Exception {
  sw.called(26);
  int i;
  int j;
  int v;

  for (i = lo0 + 1; i <= hi0 && sw.loop(27); i++) {
    sw.statement(28);
    v = a.get(i);
    j = i;
    while ((j > lo0) && (a.get(j - 1) > v) && sw.loop(29)) {
      sw.statement(30);
      a.set(j, a.get(j - 1));
      sw.statement(31);
      pause(i, j);
      j--;
    }
    sw.statement(32);
```

```
        a.set(j, v);
      }
      sw.retrn(33);
    }

    public void sort(SortItemP5.MemInts a, Sw sw) throws Exception {
      sw.called(34);
      QuickSort(a, 0, a.length - 1, sw);
      sw.statement(35);
      InsertionSort(a, 0, a.length - 1, sw);
      sw.statement(36);
      pause(-1, -1);
      sw.retrn(37);
    }
}
```

src/p5/sdw2/src/sdw/FastQSortAlgorithm.java

This is the applet main which initializes the two sort modules and starts the threads and visualization. It also handles keypresses and mouse clicks to control simulation speed and restart of any sort routine that finishes.

```
package sdw;
/*
 * Copyright 2013 Stephen D. Williams sdw@lig.net
 */

import processing.core.*;

public class Sdw2 extends PApplet {
  public Sdw2() {}
  int DSIZE = 100;
  int data[] = new int[DSIZE];
```

```
SortItemP5 item = null;
SortItemP5 item2 = null;
// BubbleSort2Algorithm bs = null;

public void setup() {
  size(640, 480);
  smooth();
  // noStroke();
  for (int x = 0; x < DSIZE; x++)
    data[x] = (int) random(100);
  item = new SortItemP5(this, 0, 0, 100, 200, .05, .05);
  item2 = new SortItemP5(this, 0, 250, 100, 200, .05, .05);
  item.init("sdw.FastQSort");
  item.startSort();
  item2.init("sdw.BubbleSort2");
  item2.startSort();

  // bs = new BubbleSort2Algorithm();
  // FastQSortAlgorithm fqs = new FastQSortAlgorithm();
}

public void draw() {
  background(255);
  fill(0xFF333366);
  // ellipseMode(RADIUS);
  // ellipse(200, 200, 20, 20);
  item.paint();
  item2.paint();
}

public static void main(String args[]) {
    PApplet.main(new String[] { "sdw.Sdw2" });
    // PApplet.main(new String[] { "--present", "sdw.Sdw2" });
  }

  public void mouseClicked() {
    item.startSort();
    item2.startSort();
  }
  public void keyPressed() {
    if (key == CODED)
      switch(keyCode) {
        case UP:
          item.slower();
          break;
        case DOWN:
          item.faster();
          break;
      }
  }
}
```

src/p5/sdw2/src/sdw/Sdw2.java

This is the base class for sort algorithms, especially the sort() method.

```
package sdw;

/*
 * @(#)SortAlgorithm.java 1.6f 95/01/31 James Gosling Copyright (c)
 * 1994-1995 Sun Microsystems, Inc.  All Rights Reserved. Permission
 * to use, copy, modify, and distribute this software and its
 * documentation for NON-COMMERCIAL or COMMERCIAL purposes and without
 * fee is hereby granted. Please refer to the file
 * http://java.sun.com/copy_trademarks.html for further important
 * copyright and trademark information and to
 * http://java.sun.com/licensing.html for further important licensing
 * information for the Java (tm) Technology. SUN MAKES NO
 * REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
 * DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. THIS SOFTWARE IS NOT
 * DESIGNED OR INTENDED FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT
 * IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS
 * IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR
 * COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT
 * MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF THE SOFTWARE
 * COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL
 * OR ENVIRONMENTAL DAMAGE ("HIGH RISK ACTIVITIES"). SUN SPECIFICALLY
 * DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR HIGH RISK
 * ACTIVITIES.
 */
// import SortItemP5;
/**
 * A generic sort demonstration algorithm SortAlgorithm.java, Thu Oct
 * 27 10:32:35 1994
 *
 * @author James Gosling
 * @version 1.6f, 31 Jan 1995
 */
/*
 * Changes Copyright 2013 Stephen D. Williams sdw@lig.net
 */
```

```

class SortAlgorithm {
  /**
   * The sort item.
   */
  private SortItemP5 parent;

  /**
   * When true stop sorting.
   */
  protected boolean stopRequested = false;

  /**
   * Set the parent.
   */
  public void setParent(SortItemP5 p) {
    parent = p;
  }

  /**
   * Pause for a while.
   */
  protected void pause() throws Exception {
    if (stopRequested) { throw new Exception("Sort Algorithm"); }
    parent.pause(parent.h1, parent.h2);
  }

  /**
   * Pause for a while and mark item 1.
   */
  protected void pause(int H1) throws Exception {
    if (stopRequested) { throw new Exception("Sort Algorithm"); }
    parent.pause(H1, parent.h2);
  }

  /**
   * Pause for a while and mark item 1 & 2.
   */
  protected void pause(int H1, int H2) throws Exception {
    if (stopRequested) { throw new Exception("Sort Algorithm"); }
    parent.pause(H1, H2);
  }

  /**
   * Stop sorting.
   */
  public void stop() {
    stopRequested = true;
  }

  /**
   * Initialize
   */
  public void init() {
    stopRequested = false;
  }

  /**
   * This method will be called to sort an array of integers.
   */
  void sort(SortItemP5.MemInts a, Sw sw) throws Exception {}
}
```

src/p5/sdw2/src/sdw/SortAlgorithm.java

Sets up the data to be sorted and directly manages the sort objects.

```
package sdw;

/*
 * @(#)SortItem.java 1.17f 95/04/10 James Gosling Copyright (c)
 * 1994-1995 Sun Microsystems, Inc. All Rights Reserved. Permission to
 * use, copy, modify, and distribute this software and its
 * documentation for NON-COMMERCIAL or COMMERCIAL purposes and without
 * fee is hereby granted. Please refer to the file
 * http://java.sun.com/copy_trademarks.html for further important
 * copyright and trademark information and to
 * http://java.sun.com/licensing.html for further important licensing
 * information for the Java (tm) Technology. SUN MAKES NO
 * REPRESENTATIONS OR WARRANTIES ABOUT THE SUITABILITY OF THE
 * SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO
 * THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
 * PURPOSE, OR NON-INFRINGEMENT. SUN SHALL NOT BE LIABLE FOR ANY
 * DAMAGES SUFFERED BY LICENSEE AS A RESULT OF USING, MODIFYING OR
 * DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES. THIS SOFTWARE IS NOT
 * DESIGNED OR INTENDED FOR USE OR RESALE AS ON-LINE CONTROL EQUIPMENT
 * IN HAZARDOUS ENVIRONMENTS REQUIRING FAIL-SAFE PERFORMANCE, SUCH AS
 * IN THE OPERATION OF NUCLEAR FACILITIES, AIRCRAFT NAVIGATION OR
 * COMMUNICATION SYSTEMS, AIR TRAFFIC CONTROL, DIRECT LIFE SUPPORT
 * MACHINES, OR WEAPONS SYSTEMS, IN WHICH THE FAILURE OF THE SOFTWARE
 * COULD LEAD DIRECTLY TO DEATH, PERSONAL INJURY, OR SEVERE PHYSICAL
 * OR ENVIRONMENTAL DAMAGE ("HIGH RISK ACTIVITIES"). SUN SPECIFICALLY
 * DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY OF FITNESS FOR HIGH RISK
 * ACTIVITIES.
 */
/*
 * Changes Copyright 2013 Stephen D. Williams sdw@lig.net
 */
import java.awt.*;
import java.io.InputStream;
import java.util.Hashtable;
import java.net.*;
import processing.core.*;

/**
 * A simple applet class to demonstrate a sort algorithm. You can
 * specify a sorting algorithm using the "alg" attribute. When you
 * click on the applet, a thread is forked which animates the sorting
 * algorithm.
 *
 * @author James Gosling
 * @version 1.17f, 10 Apr 1995
 */
public class SortItemP5 implements Runnable {
  PApplet p = null;
  int naplen = 40; // ms to sleep for redraw / animation speed.
  int fontHeight = 10;

  public void faster() {
    naplen++;
    if (naplen > 100) naplen = 100;
  }

  public void slower() {
    naplen--;
    if (naplen < 0) naplen = 0;
  }

  int realx, realy, wx, wy, ww, wh;
  double fader, fadew;
  public SortItemP5(PApplet p, int ax, int ay, int aw, int ah, double
   faderp, double fadewp) {
    this.p = p;
    realx = ax;
    realy = ay;
    wx = ax;
    wy = ay+fontHeight+1;
    ww = aw;
    wh = ah;
    fader = faderp;
    fadew = fadewp;
  }
  void line(int ax, int ay, int x2, int y2) {
    p.line(wx+ax, wy+ay, wx+x2, wy+y2);
  }

  void rect(int ax, int ay, int x2, int y2) {
    p.noStroke();
    p.rect(wx+ax, wy+ay, x2, y2);
  }

  void rectOffset(int offx, int offy, int ax, int ay, int x2, int y2) {
    p.noStroke();
    p.rect(offx+wx+ax, offy+wy+ay, x2, y2);
  }
```

```
/**
 * The thread that is sorting (or null).
 */
private Thread kicker;

/**
 * The array that is being sorted.
 */
// int arr[];
MemInts arr;

/**
 * The high water mark.
 */
int h1 = -1;

/**
 * The low water mark.
 */
int h2 = -1;

/**
 * The name of the algorithm.
 */
String algName, minAlgName;

/**
 * The sorting algorithm (or null).
 */
SortAlgorithm algorithm;

```

```
  MemInts mi = null; // Tracking memory
  Sw sw = null; // Tracking code flow

  int size;
  /**
   * Fill the array with random numbers from 0..n-1.
   */
  void scramble() {
    size = wh/2;
    sw = new Sw();
    mi = new MemInts(size);
    double f = ww / (double) mi.size();
    for (int i = size; --i >= 0;) {
      mi.set(i, (int) (size * f * Math.random()));
/*
 * jh: fill the array with numbers from 0..n-1, not a scrambled set of
   0..n-1 unique numbers. So
 * duplicates will occur in in most cases
 */
/* a[i] = (int)(i * f); */
    }

/* jh: we don't shuffle the array anymore */
/*
 * for (int i = a.length; --i >= 0;) { int j = (int)(i * Math.random())
   ; int t = a[i]; a[i] = a[j];
 * a[j] = t; }
 */
    arr = mi;
    mi.resetAge();
  }
```

```

    /**
     * Pause a while.
     *
     * @see SortAlgorithm
     */
    void pause() {
      pause(-1, -1);
    }

    /**
     * Pause a while, and draw the high water mark.
     *
     * @see SortAlgorithm
     */
    void pause(int H1) {
      pause(H1, -1);
    }

    /**
     * Pause a while, and draw the low&high water marks.
     *
     * @see SortAlgorithm
     */
    void pause(int H1, int H2) {
      h1 = H1;
      h2 = H2;
      if (kicker != null) {
        p.redraw();
      }
      try {
```

```
      Thread.sleep(naplen);
    } catch (InterruptedException e) {}
  }

  /**
   * Initialize the applet.
   */
  public void init() {
    String at = p.getParameter("alg");
    init(at);
  }
  PFont font;
  public void init(String at) {
    font = p.loadFont("ArialNarrow-12.vlw");
    if (at == null) {
      at = "BubbleSort";
    }

    algName = at + "Algorithm";
    minAlgName = at;
    scramble();

    // resize(100, 100);
  }

  int limit(int val, int age, int range) {
    int r = age-val;
    if (r > range) return 0;
    return r;
  }
```

```
/**
 * Paint the array of numbers as a list of horizontal lines of
 varying lengths.
 */
public void paint() {
  int y = wh - 1;

  // Draw new lines
  y = wh - 1;
  p.stroke(Color.black);
  for (int i = size; --i >= 0; y -= 2) {
    line(0, y, arr.getNoTrack(i), y);
  }

  if (h1 >= 0) {
    y = h1 * 2 + 1;
    p.stroke(Color.red);
    line(0, y, ww, y);
  }
  if (h2 >= 0) {
    y = h2 * 2 + 1;
    p.stroke(Color.blue);
    line(0, y, ww, y);
  }

  // Labels
  p.textFont(font);
  p.text(minAlgName, realx, realy+fontHeight);
  p.text("R/W heatmap", realx+ww, realy+fontHeight);
  p.text("Scalars, 2 ways", realx+ww*2, realy+fontHeight);
  p.text("Total R/W", realx+ww*3, realy+fontHeight);
```

```

    // Viz2
    int scale = 7;
    int row = 0;
    int over = ww/(scale+1);
    for (int i=0; i < size; i++) {
      int li = i-(row*over);
      if (li >= over) { // Wrap
        row++;
        li = 0;
      }
      // Read/write recency / pattern: This fades too quickly.
      // int red = (128-limit(mi.ager[i], mi.age, 128))*2;
      // int red = (int)((1.0/(mi.rage-mi.ager[i]+1))*255);
      // int green = (128-limit(mi.agew[i], mi.age, 128))*2;
      // int green = (int)((1.0/(mi.wage-mi.agew[i]+1))*255);

      // This works well and is tunable:
      int bright = (int)mi.agerm[i];
      p.fill(Color.scolor(0, bright, 0));
      rect(ww+li*(scale+1),row*(scale+1),scale,scale);
      bright = (int)mi.agewm[i];
      p.fill(Color.scolor(bright, 0, 0));
      rect(ww+li*(scale+1),wh/2+row*(scale+1),scale,scale);
      if (!done) mi.fade(fader, fadew);

      // scalar angles = angle == magnitude
      int dx = wx+2*ww+li*(scale+1);
      int dy = wy+row*(scale+1);
      float halfScale = (float)(scale/2.0);
      p.pushMatrix();
      p.translate(dx+halfScale, dy+scale);
      p.rotate(PApplet.radians(p.map(mi.a[i], 0, 255, 0, 180)));
      p.stroke(Color.red);
      p.line(0,0,0,-scale);
      p.stroke(Color.black);
      p.line(0, 0, 0, 0);
      p.popMatrix();

      // scalar blocks: size == magnitude
      dx = wx+2*ww+li*(scale+1);
      dy = wy+wh/2+row*(scale+1);
      float fscaled = p.map(mi.a[i], 0, 255, 0, scale+2);
      p.fill(255,255,255);
      p.rect(dx, dy+scale-fscaled, scale-1, fscaled);

      // scalar blocks = size == number of times read
      dx = wx+3*ww+li*(scale+1);
      dy = wy+row*(scale+1);
      int val = p.constrain(mi.reads[i], 0, 255);
      fscaled = p.map(val, 0, 255, 0, scale+2);
      p.fill(Color.scolor(0, val, 0));
      p.rect(dx, dy+scale-fscaled, scale, fscaled);

      // scalar blocks = size == number of times written
      dx = wx+3*ww+li*(scale+1);
      dy = wy+wh/2+row*(scale+1);
      val = p.constrain(mi.writes[i], 0, 255);
      fscaled = p.map(val, 0, 255, 0, scale+2);
      p.fill(Color.scolor(val, 0, 0));
      p.rect(dx, dy+scale-fscaled, scale, fscaled);
```

```
    }
    int dx = wx+4*ww;
    sw.draw(p, dx, wy, ww, wh, false);
    sw.draw(p, dx+ww, wy, ww, wh, true);
  }

  /**
   * Update without erasing the background.
   */
  public void update() {
    p.redraw();
  }
  boolean done = false;
  /**
   * Run the sorting algorithm. This method is called by class Thread
    once the sorting algorithm is
   * started.
   *
   * @see java.lang.Thread#run
   * @see SortItem#mouseUp
   */
  public void run() {
    done = false;
    try {
      if (algorithm == null) {
        algorithm = (SortAlgorithm) Class.forName(algName).newInstance
   ();
        algorithm.setParent(this);
      }
      algorithm.init();
      algorithm.sort(arr, sw);
      done = true;
    } catch (Exception e) {
      System.out.println(e.toString());
      e.printStackTrace();
    }
  }

  /**
   * Stop the applet. Kill any sorting algorithm that is still sorting.
   */
  public synchronized void stop() {
    if (kicker != null) {
      try {
        kicker.stop();
      } catch (IllegalThreadStateException e) {
        // ignore this exception
      }
      kicker = null;
    }
    if (algorithm != null) {
      try {
        algorithm.stop();
      } catch (IllegalThreadStateException e) {
        // ignore this exception
      }
    }
  }



  /**
   * For a Thread to actually do the sorting. This routine makes sure
   we do not simultaneously start
   * several sorts if the user repeatedly clicks on the sort item. It
   needs to be synchronized with
   * the stop() method because they both manipulate the common kicker
   variable.
   */
  public synchronized void startSort() {
    if (kicker == null || !kicker.isAlive()) {
      scramble();
      p.repaint();
      kicker = new Thread(this);
      kicker.start();
    }
  }


  /**
   * The user clicked in the applet. Start the clock!
   */
  public boolean mouseUp(java.awt.Event evt, int x, int y) {
    startSort();
    return true;
  }

  /**
   * This tracks memory accesses.
   * It should be rewritten to do aging without updating each element
   at each step.
   * @author sdw
   *
   */
```

```
public class MemInts {
  public int wage, rage, length;
  public int a[];
  public int ager[];
  public int agew[];
  public int reads[];
  public int writes[];
  public float agerm[];
  public float agewm[];
  public MemInts(int n) {
    length = n;
    wage = rage = 0;
    a = new int[n];
    ager = new int[n];
    agew = new int[n];
    reads = new int[n];
    writes = new int[n];
    agerm = new float[n];
    agewm = new float[n];
  }
  public void resetAge() {
    for (int x = 0; x < length; x++) {
      ager[x] = 0;
      agew[x] = 0;
      reads[x] = 0;
      writes[x] = 0;
      agerm[x] = 0;
      agewm[x] = 0;
    }
  }
  public int get(int n) {
        reads[n]++;
        ager[n] = rage++;
        agerm[n] = 255;
        return a[n];
    }
    public int getNoTrack(int n) {
        return a[n];
    }
    public void set(int n, int i) {
        writes[n]++;
        agew[n] = wage++;
        agewm[n] = 255;
        a[n] = i;
    }
    public void fade(double r, double w) {
        for (int i=0; i < length; i++) {
            agerm[i] = agerm[i]>r ? agerm[i]-(float)r : 0;
            agewm[i] = agewm[i]>w ? agewm[i]-(float)w : 0;
        }
    }
    public int size() { return length; }
  }
}
```

src/p5/sdw2/src/sdw/SortItemP5.java

Sw.java contains the software structure, control flow visualization module. The algorithm used maintains a data structure that only has as many entries as there are distinct linenumber calls, plus any alternative paths from each statement. Additional calls increment the count for statement.

```
package sdw;
```

```
/*
 * Copyright 2013 Stephen D. Williams sdw@lig.net
 */

import java.util.ArrayList;
import java.util.Map;
import java.util.TreeMap;

import processing.core.PApplet;

/**
 * This class allows tracking of software statement types and graphing
    of the resulting structure.
 * The previous statement is remembered for each new trace so that
    lines can be drawn back to it.
 *
 * @author sdw
 *
 */
public class Sw {
  int stepi = 0;
  int max = 0;
  int gdepth = 0;

  class Trace {
    public int step;
    public int next;
    public char type;
    public int freq;
    public int depth;
    public Trace(int s, int n, char t) {
      freq = 1;
      step = s;
      next = n;
      type = t;
      if (s > max) max = s;
      step();
    }
    public void updateDepth(char t) {
      if (t == 'c') gdepth++;
      if (t == 'r') gdepth--;
      this.depth = gdepth;
    }
    public void bump() {
      freq++;
    }
  }

  int lastStep = 0;
  char lastType = ' ';
  Trace lastTrace = null;
  TreeMap<Integer, TreeMap<Integer, Trace>> code = new TreeMap<Integer,
    TreeMap<Integer, Trace>>();

  int step() {
    return stepi++;
  }

  public int size() {
    return code.size();
  }
```

```
public void add(int step, char c) {
  synchronized (this) {
    if (stepi == 0) {
      lastStep = step;
      lastType = c;
      step();
      return;
    }
    // We have current and last. Process last.
    TreeMap<Integer, Trace> traces = null;
    try {
      traces = code.get(lastStep);
    } catch (Exception ex) {}
    if (traces == null) {
      traces = new TreeMap<Integer, Trace>();
      code.put(lastStep, traces);
    }
    Trace trace = traces.get(step);
    if (trace == null) {
      trace = new Trace(lastStep, step, lastType);
      traces.put(step, trace);
    } else trace.bump();
    trace.updateDepth(c);
    lastStep = step;
    lastType = c;
    step();
  }
}

public interface TraceApply {
  public void perTrace(Trace tr);
```

```
}

public void apply(TraceApply ta) {
  synchronized (this) {
    int codes = code.size();
    for (Map.Entry<Integer, TreeMap<Integer, Trace>> codeEntry : code
  .entrySet()) {
      TreeMap<Integer, Trace> traces = codeEntry.getValue();
      if (traces != null) {
        for (Map.Entry<Integer, Trace> traceEntry : traces.entrySet()
  ) {
          ta.perTrace(traceEntry.getValue());
        }
      }
    }
  }
}

public Sw() {}

public void called(int line) {
  add(line, 'c');
}

public void retrn(int line) {
  add(line, 'r');
}

public boolean branch(int line) {
  add(line, 'b');
  return true;
```

```
}

public void statement(int line) {
  add(line, 's');
}

public boolean loop(int line) {
  add(line, 'l');
  return true;
}

public void draw(final PApplet p, final int ax, final int ay, final
 int aw, final int ah, final boolean levels) {
  final int cs = stepi/4;
  final float scale = (1.0f * ah) / max;
  p.pushStyle();
  p.text("Structure"+(levels?" w/call indent":""), ax+3, ay);
  // p.fill(Color.scolor(0, 0, 0));
  // p.noStroke();
  apply(new TraceApply() {
    public void perTrace(Trace trace) {
      int maxStroke = 20;
      float x = ax + (levels ? trace.depth*(1.0f * ah)/20 : 0);
      float y = ay + trace.step * scale;
      float x2 = ax + (levels ? trace.depth*(1.0f * ah)/20 : 0);
      float y2 = ay + trace.next * scale;
      float wq = Math.abs(trace.next - trace.step * 1.0f+1) / max *
 aw;
      int freqi = trace.freq;
      switch (trace.type) {
        case 'b':
          p.noFill();
          p.stroke(10, 128, 255, 128);
          p.strokeWeight(p.map(freqi, 0, cs, 1, maxStroke));
          p.bezier(x, y, x + wq, y, x2 + wq, y2, x2, y2);
          break;
        case 'c':
          p.noFill();
          p.stroke(10, 128, 128, 255);
          p.strokeWeight(p.map(freqi, 0, cs, 1, maxStroke));
          p.bezier(x, y, x + wq, y, x2 + wq, y2, x2, y2);
          break;
        case 'r':
          p.noFill();
          p.stroke(10, 128, 128, 255);
          p.strokeWeight(p.map(freqi, 0, cs, 1, maxStroke));
          p.bezier(x, y, x + wq, y, x2 + wq, y2, x2, y2);
          break;
        case 'l':
          p.noFill();
          p.stroke(10, 128, 255, 255);
          p.strokeWeight(p.map(freqi, 0, cs, 1, maxStroke));
          p.bezier(x, y, x + wq, y, x2 + wq, y2, x2, y2);
          break;
        default:
        case 's':
          p.stroke(10, 255, 128, 128);
          p.strokeWeight(p.map(freqi, 0, cs, 1, maxStroke));
          p.line(x, y, x2, y2);
          break;
      }
    }
```

```
        });
        p.popStyle();
    }
}
```

src/p5/sdw2/src/sdw/Sw.java

REFERENCES

Aisch, Gregor (2011, June). "Radial bubble tree." URL `http://driven-by-data.net/about/interactive-bubbletree`; last checked 2013-08-08.

Alexander, Christopher (1979). *The Timeless Way of Building.* Oxford University Press, Inc.

Allen, James F. (1983, Nov). "Maintaining knowledge about temporal intervals." *Communications of the ACM* 26(11), 832–843.

Alspaugh, Thomas A. (2013). "Allen's interval algebra." URL: `https://www.ics.uci.edu/~alspaugh/cls/shr/allen.html`; last checked 2013-07-28.

Assorted, ed. (2011). *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology.* The Association for Computing Machinery.

Assorted (2013a, July). "bochs - the cross platform ia-32 emulator." URL: `http://bochs.sourceforge.net/`; last checked 2013-0718.

Assorted (2013b, July). "Freemind - free mind mapping software." URL: `http://freemind.sourceforge.net/wiki`; last checked 2013-07-18.

Assorted (2013c, July). "Processing tiddlyspot." URL: `http://processing.tiddlyspot.com/`; last checked 2013-07-18.

Assorted (2013d, July). "Tiddlyprocessing." URL: `http://whatfettle.com/2008/05/TiddlyProcessing/`; last checked 2013-07-18.

Atkin, Albert (2013). "Peirceś theory of signs." In Edward N. Zalta, ed., *The Stanford Encyclopedia of Philosophy* (Summer 2013 ed.). Stanford University.

Balzer, Michael, Oliver Deussen, and Claus Lewerentz (2005). "Voronoi treemaps for the visualization of software metrics." In *Proceedings of the 2005 ACM symposium on Software visualization*, SoftVis '05, New York, NY, USA, pp. 165–172. ACM.

Battista, Giuseppe Di (1999). *Graph Draing: algorithms for the visualization of graphs.* Prentice-Hall, Inc.

Bederson, Benjamin B. and Ben Schneiderman (2003). *The Craft of Information Visualization, Readings and Reflections* (2007 ed.). Morgan Kaufman.

Bellard, Fabrice (2013, July). "Jslinux - javascript pc emulator running linux." URL: `http://jslinux.org/`; last checked 2013-07-18.

Bellard, Fabrice et al. (2013). "Qemu open source processor emulator." URL `http://wiki.qemu.org`; last checked 2013-07-18.

Bereciatua, Martin Ignacio (2005, Sep). "Letter-pairs analysis application." URL `http://www.m-i-b.com.ar/mib/letter_pairs/eng/index.htm`; last checked 2013-08-04.

Black, Michael (2011). "NSF proposal for teaching computer simulator." Project proposal.

Black, Michael and Manoj Franklin (2011). "Teaching computer architecture with a graphical pc simulator." In *ACM ITiCSE 2011*. ACM.

Black, Michael and Priyadarshini Komala (2011). "A full system x86 simulator for teaching computer organization." In *The 42nd ACM Technical Symposium on Computer Science Education*, pp. 271. ACM: ACM.

Black, Michael and Nathaniel Waggoner (2013). "Emumaker86: A hardware simulator for teaching cpu design." In *SIGCSE'13*, pp. 24. SIGCSE: ACM.

Black, Michael and Stephen Williams (2013, July). "Amerisim home page at github." URL: `https://github.com/sdwlig/AmeriSim`; last checked 2013-07-18.

Blanchet, Gérard and Bertrand Dupouy (2012). *Computer Architecture*. Wiley.

Bradley, Steven (2011, February). "3 design layouts: Gutenberg diagram, z-pattern, and f-pattern." URL `http://www.vanseodesign.com/web-design/3-design-layouts/`; last checked 2013-08-08.

Branovic, Irina, Roberto Giorgi, and Enrico Martinelli (2003). "Webmips: A new web-based mips simulation environment for computer architecture education.

Brokmeier, Jon 'Zonker' (2010, April). "Containers vs. hypervisors: Choosing the best virtualization technology." URL: `https://www.linux.com/component/content/article/186-virtualization/300057-containers-vs-hypervisors-choosing-the-best-virtualization-technology-`; last checked 2013-07-28.

Card, Stuart K., Jock D. MacKinlay, and Ben Schneiderman (1999). *Readings in Information Visualization - Using Vision to Think*. Academic Press.

Catto, Erin (2013, July). "Box2d, a 2d physics engine for games." URL: `http://box2d.org/`; last checked 2013-07-18.

Cockton, Gilbert (2013). *The Encyclopedia of Human-Computer Interaction* (2nd ed.)., Chapter Usability Evaluation. Aarhus, Denmark: The Interaction Design Foundation. URL `http://www.interaction-design.org/encyclopedia/usability_evaluation.html`; last checked 2013-08-05.

Comer, Douglas E. (2004). *Essentials of Computer Architecture*. Addison-Wesley.

community, Wikipedia (2013a, May). "Fitts's law." URL `https://en.wikipedia.org/wiki/Fitts%27s_law`; last checked 2013-08-08.

community, Wikipedia (2013b, July). "Merkle tree." URL: `https://en.wikipedia.org/wiki/Merkle_tree`; last checked 2013-07-28.

community, Wikipedia (2013c). "Skeuomorph." URL: `https://en.wikipedia.org/wiki/Skeuomorph`; last checked 2013-07-18.

Csikszentmihalyi, Mihaly (2013, July). "Mihaly csikszentmihalyi: Flow, the secret to happiness." URL: `http://www.ted.com/talks/mihaly_csikszentmihalyi_on_flow.html`; last checked 2013-07-18.

Csikszentmihalyi, Mihaly and Isabella Selega Csikszentmihalyi, eds. (1988). *Optimal experience - Psychological Studies of Flow in Consciousness*. Cambridge University Press.

de Vega, Manuel, Margaret Jean Intons-Peterson, Philip N. Johnson-Laird, Michel Denis, and Marc Marschark (1996). *Models of Visuospatial Cognition*. Oxford University Press, Inc.

Despain, Wendy, ed. (2013). *100 Principles of Game Design*. New Riders is an imprint of Peachpit, a division of Pearson Education.

Dike, Jeff (2006, April). *User Mode Linux*. Prentice Hall.

Dike, Jeff (2013). "User mode linux." URL `http://user-mode-linux.sourceforge.net/`; last checked 2013-07-18.

Eagle, Chris (2011). *The IDA Pro Book* (Second Edition ed.). No Starch Press.

Ernst, Michael D., Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao (2007, Dec). "The Daikon system for dynamic detection of likely invariants." *Science of Computer Programming* 69(1–3), 35–45. URL `http://www.cs.washington.edu/homes/mernst/pubs/daikon-tool-scp2007.pdf`; last checked 2013-08-05.

Fagin, Barry and Dale Skrien (2013, July). "Iassim - a programmable emulator for the princeton ias/von neumann machine." URL: `http://www.cs.colby.edu/djskrien/IASSim/`; last checked 2013-07-18.

Force, ACM/IEEE-CS Joint Task, ed. (2013). *Computer Science Curricula 2013*. ACM. URL `http://ai.stanford.edu/users/sahami/CS2013/`; last checked 2013-07-18.

Ford, Neal, Matthew Mccullough, and Nathaniel Schutta (2013). *Presentation Patterns - Techniques for Crafting Better Presentations*. Addison-Wesley.

Fouh, Eric, Monika Akbar, and Clifford A. Shaffer (2012, April). "The role of visualiztion in computer science education." *Computers in the Schools* 29, 1–2, 95–117.

Foundation, Linux (2013a). "The xen project." URL `http://www.xenproject.org/`; last checked 2013-07-18.

Foundation, The Linux (2013b). "Kvm: The kernel based virtual machine." URL `http://www.linux-kvm.org/page/Main_Page`; last checked 2013-07-18.

Garcia, M.I., S. Rodriguez, A. Perez, and A. Garcia (2009, May). "p88110: A graphical simulator for computer architecture and organization courses." *Education, IEEE Transactions on* 52(2), 248–256.

GmbH, Parallels IP Holdins (2013). "Parallels." URL `http://www.parallels.com/`; last checked 2013-07-18.

Göktürk, Mehmet (2010, May). "Fitts's law." URL `http://www.interaction-design.org/encyclopedia/fitts_law.html`; last checked 2013-08-08.

Goodall, John R., Conti Gregory, and Kwan-Liu Ma, eds. (2007). *VizSEC 2007: Proceedings of the Workshop on Visualization for Computer Security*. Springer-Verlag Berlin Heidelberg.

Gosling, James, Jason Harrison, and Jim Boritz (2002, Dec). "Sorting algorithms." URL `http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html`; last checked 2013-08-04.

Göttling, Klaus (2013, June). "File: Redstair gearcompressor.png." URL `https://en.wikipedia.org/wiki/File:Redstair_GEARcompressor.png`; last checked 2013-08-08.

Graphviz (2013). "Graph visualization software." URL: `http://www.graphviz.org`; last checked 2013-07-18.

Group, MIT Program Analysis et al. (2013, 7). "The daikon invariant detector." URL `http://groups.csail.mit.edu/pag/daikon/`; last checked 2013-08-04.

Heer, Jeffrey and Ben Schneiderman (2012). "Interactive dynamics for visual analysis." *acmqueue - Microprocessors* Vol. 10 No. 2 - February 2012, 1–26.

Hennessy, John L. and David A. Patternson (1996). “Windlx.” URL `http://electro.fisica.unlp.edu.ar/arq/downloads/Software/WinDLX/windlx.html`; last checked 2013-08-04.

Hennessy, John L. and David A. Patterson (1990). *Computer Architecture - A Quantitative Approach* (Third Edition 2003 ed.). Morgan Kaufman Publishers.

Hong, Song-You, Myung-Seo Koo, Jihyeon Jang, Jung-Eun Esther Kim, Hoon Park, Min-Su Joh, Ji-Hoon Kang, and Tae-Jin Oh (2013, July). “An evaluation of the software system dependency of a global atmospheric model.” *Monthly Weather Review*.

Johnson, B and Ben Schneiderman (1991). “Tree-maps: a space-filling approach to the visualization of hierarchical information structures.” In GM Nielson and L Rosenblum, eds., *Second Conference on Visualization ’91*, Los Alamitos, CA, pp. 284–291. IEEE Visualization: IEEE COmputer Society Press.

Johnson-Laird, P. N. (1983). *Mental Models*. Harvard University Press.

Joyce, Daniel T., Deborah Knox, Wanda Dann, and Thomas L. Naps, eds. (2004). *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2004, Norfolk, Virginia, USA, March 3-7, 2004*. ACM.

Lamping, John, Ramana Rao, and Peter Pirolli (2013). “Hyperbolic trees.” URL: `http://www.infovis-wiki.net/index.php?title=Hyperbolic_trees`; last checked 2013-07-18.

Larus, James (1990-2011). “Spim: A mips32 simulator.” URL `http://spimsimulator.sourceforge.net/`; last checked 2013-08-04.

Lidwell, William, Kritina Holden, and Jill Butler (2003). *Universal Principles of Design* (2010 ed.). Rockport.

Lopez-Hernandez, Roberto, David Guilmaine, Michael J. McGuffin, and Lee Barford (2010). “A layer-oriented interface for visualizing time-series data from oscilloscopes.” In *Proceedings of IEEE Pacific Visualization (PacificVis) 2010*, pp. 41–48. PacificVis: IEEE.

Maeda, John (2009, August). “When a good idea works - purity, openness, and simplicity are engines of design.” URL `http://www.technologyreview.com/review/414828/when-a-good-idea-works/`; last checked 2013-08-04.

Magnusson, Thor (2009, August). “Of epistemic tools: musical instruments as cognitive extensions.” *Organized Sound* 14(02), 168–176.

McGuffin, Michael J., Gord Davison, and Ravin Balakrishnan (2009). “Expand-ahead: A space-filling strategy for browsing trees.” URL: `https://www.youtube.com/watch?v=aB5kitbivEM`; last checked 2013-07-18. Video.

Milligan, Colin and Ruth Thomas (2009, May). "JeLSIM java elearning SIMulations." URL http://www.jelsim.org/; last checked 2013-08-04.

Moure, J. C., Dolores I. Rexachs, and Emilio Luque (2002, March). "The kscalar simulator." *J. Educ. Resour. Comput.* 2(1), 73–116.

Munzner, Tamara (2006a, May). "15 views of a node-link graph: An information visualization portfolio." URL http://www.cs.ubc.ca/~tmm/talks/nih06/; last checked 2013-08-04.

Munzner, Tamara (2006b). "15 views of a node-link graph: an infovis portfoloio." URL: http://www.cs.ubc.ca/~tmm/talks.html; last checked 2013-07-18.

Narayanasamy, Viknashvaran, Kok Wai Wong, Chun Che Fung, and Shri Rai (2006, apr). "Distinguishing games and simulation games from simulators." *Comput. Entertain.* 4(2), 1–18.

Newman, Mark (2013a, July). "Images of the social and economic world." URL: http://www-personal.umich.edu/~mejn/cartograms/; last checked 2013-07-18.

Newman, Mark (2013b, July). "Maps of the 2012 us presidential election results." URL: http://www-personal.umich.edu/~mejn/election/2012/; last checked 2013-07-18.

Norman, Donald A. (2002). *The Design of Everyday Things.* Basic Books.

of Engineering University of Dulsburg-Essen, Faculty (2013). "FreeStyler." URL http://www.collide.info/en/freestyler; last checked 2013-08-04.

Olli, Leino, Wirman Hanna, and Fernandez Amyris, eds. (2008). *Extending Experiences. Structure, analysis and design of computer game player experience.* Lapland University Press.

Oracle (2013). "VirtualBox." URL https://www.virtualbox.org/; last checked 2013-07-18.

Parhami, Behrooz (2005). *Computer Architecture: From Microprocessors to Supercomputers.* Oxford University Press, Inc.

Parhami, Behrooz (2011, November). "Behrooz parhami's textbook on computer architecture." URL http://http://www.ece.ucsb.edu/~parhami/text_comp_arch.htm; last checked 2013-08-08.

Parnin, Chris (2013, January). "Programmer interrupted." URL http://blog.ninlabs.com/2013/01/programmer-interrupted/; last checked 2013-08-04.

Pereira, Maicon Carlos (2013). "Webmips - mips cpu pipelined simulation on line." URL `http://www.maiconsoft.com.br/webmips/riconoscimento.asp`; last checked 2013-08-04.

Plaisant, Catherine, J. Grosjean, and B.B. Bederson (2002). "Spacetree: supporting exploration in large node link tree, design evolution and empirical evaluation." In *IEEE Symposium on Information Visualization, 2002*, pp. 57–64. INFOVIS 2002: IEEE.

Preim, Bernhard and Dirk Bartz (2007). *Visualization in Medicine, Theory, Algorithms, and Applications*. Morgan Kaufman.

Preston, Ian, Rhys Newman, Jeff Tseng, Chris Dennis, Guillaume Kirsch, and Mike Moleschi (2013, July). "Jpc - the pure java x86 pc emulator." URL: `http://jpc.sourceforge.net/home_home.html`; last checked 2013-07-18.

Project, Community (2013). "Openvz linux containers." URL `http://openvz.org`; last checked 2013-07-18.

Random House, Inc., ed. (2005). *Random House Kemerman Webster's College Dictionary*. K Dictionaries Ltd. and Random House, Inc.

Reas, Casey and Ben Fry (2007). *Prrocessing, A Programming Handbook for Visual Designers and Artists*. The MIT Press.

Riehmann, Patrick, Manfred Hanfler, and Bernd Froehlich (2005). "Interactive sankey diagrams." In *Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*, INFOVIS '05, Washington, DC, USA, pp. 31–. IEEE Computer Society.

Saraiya, Purvi, Clifford A. Shaffer, D. Scott McCrickard, and Chris North (2004). "Effective features of algorithm visualizations." See Joyce et al. (2004), pp. 382–386.

Sayood, K. (2002). *Lossless Compression Handbook*. Communications, Networking and Multimedia. Elsevier Science. URL: `http://books.google.com/books?id=LjQiGwyabVwC`; last checked 2013-07-28.

Schneiderman, Ben (1998). *Designing the User Interface* (Third ed.). Addison-Wesley.

Scott, Mike (2012, April). "Winmips64." URL `http://indigo.ie/~mscott/`; last checked 2013-08-04.

Shaffer, Cliff et al. (2013, February). "Research questions." URL `http://algoviz.org/algoviz-wiki/index.php/Research_Questions`; last checked 2013-08-08.

Shaffer, Clifford A., Matthew L. Cooper, Alexander Joel D. Alon, Monika Akbar, Michael Stewart, Sean Ponce, and Stephen H. Edwards (2010, August). “Algorithm visualization: The state of the field.” *ACM Transactions on Computing Education* 10(3), 1–22.

Shah, Priti and Akira Miyaka, eds. (2005). *The Cambridge Handbook of Visuospatial Thinking*. Cambridge University Press.

Solutions, Insignia (1996-). “Softpc.” URL `https://en.wikipedia.org/wiki/SoftPC`; last checked 2013-07-18.

Suel, Torsten and Nasir Memon (2002). “Algorithms for delta compression and remote file synchronization.” URL: `cis.poly.edu/suel/papers/delta.pdf`; last checked 2013-07-28.

Taherkhani, Ahmad, Ari Korhonen, and Lauri Malmi (2012, 4). “Classifying and recognizing students’ sorting algorithm implementations in a data structures and algorithms course.” Research reposrt, Aalto University.

Tufte, Edward (2013). “Sparkline theory and practice.” URL: `http://www.edwardtufte.com/bboard/q-and-a-fetch-msg?msg_id=0001OR`; last checked 2013-07-18.

Tuttle, Charles E. (1988). *A Guide to Remembering Japanese Characters* (Twelfth Printing, 2000 ed.). Tuttle Publishing.

Unknown (2012). “Flow chart.” URL `https://farm2.static.flickr.com/1228/1385806571_c7a6bf2006.jpg`; last checked 2013-08-04.

VMWare (2013). “VMWare.” URL `https://www.vmware.com/`; last checked 2013-07-18.

Ware, Colin (2004). *Information Visualization* (Second ed.). Morgan Kaufman.

Ware, Colin (2008). *Visual Thinking for Design*. Morgan Kaufman Publishers is an imprint of Elsevier.

Whitaker, Harold, John Halas, and Tom Sito (1981a). *Timing for Animation* (2009 ed.). Focal Press is an imprint of Elsevier.

Whitaker, Harold, John Halas, and Tom Sito (1981b). *Timing for Animation* (Second Edition 2009 ed.). Focal Press is an imprint of Elsevier.

Wickham, Hadley (2008). *Practical tools for exploring data and models*. Self.

Wikipedia (2008). “Choropleth — wikipedia, the free encyclopedia.” URL: `http://en.wikipedia.org/w/index.php?title=Choropleth&oldid=243885069`; accessed 19-July-2013.

Wikipedia (2013a). “Activity recognition — wikipedia, the free encyclopedia.” URL: `http://en.wikipedia.org/w/index.php?title=Activity_recognition&oldid=553352507`; last checked 2013-07-18.

Wikipedia (2013b). “Emulator — wikipedia, the free encyclopedia.” URL: `http://en.wikipedia.org/w/index.php?title=Emulator&oldid=563955226`; last checked 2013-07-18.

Wikipedia (2013c). “Flow (psychology) — wikipedia, the free encyclopedia.” URL: `http://en.wikipedia.org/w/index.php?title=Flow_(psychology)&oldid=563904712`; accessed 2013-07-18.

Wikipedia (2013d). “Identicon.” URL: `https://en.wikipedia.org/wiki/Identicon`; last checked 2013-0718.

Wood, Jo, Aidan Slingsby, and Jason Dykes (2013, April). “gicentre utilities.” URL `http://gicentre.org/utils/`; last checked 2013-08-04.

Wurman, Richard Saul (1997). *Information Architects.* Graphis Inc.

Yau, Nathan (2011). *Visualize This, The FlowingData Guide to Design, Visualization, and Statistics.* Wiley Publishing, Inc.

Yee, Ka-Ping, Danyel Fisher, Rachna Dhamija, and Marti Hearst (2001). “Animated exploration of dynamic graphs with radial layout.” In *InfoVis 2001.* IEEE Visualization: IEEE. URL `http://bailando.sims.berkeley.edu/papers/infovis01.htm`; last checked 2013-08-05.