1 The Problem

A huge amount of software-development effort in industry is wasted creating multiple implementations of the same functionality. Often, common data structures, algorithms, and protocols are reimplemented as part of different systems developed by different groups within the same company. The reason for such duplication is that the components of existing systems cannot easily be reused in new systems. Companies often have large legacy code bases that have grown by accretion without global modular design. The monolithic nature of such code bases prevents code reuse. Industry needs a way to migrate functionality from existing systems to new systems. Without such migration, duplicate software-development effort significantly reduces the overall efficiency of the existing manpower pool and limits the ultimate level of functionality of the code base produced by that limited manpower resource.

There is a need for software reuse outside of industry as well. As part of the scientific process, researchers often write software. Advancement of science requires that new research build upon prior results. The scientific community has long had publication mechanisms for archiving and disseminating research results in the form of text. The advent of the Internet and the Web has added the ability to disseminate software. Researchers often make their code available on Web pages and on-line appendices. This allows researchers to easily download and run each other’s code much as they read each other’s papers. Unlike papers, which are primarily read as static documents to foster new research that builds on prior results, new research rarely involves simply downloading and running existing code. Instead, it involves retrieving, modifying, and combining existing code for new purposes. Simply making code available on Web pages does not facilitate this process.

The need and desire to share software is not limited to industry and the scientific community. Recent years have seen an explosion in the availability of free software. More than a dozen new packages are announced every day. Yet, while the free-software community has perfected methods that support a distributed team of developers when working on the same project, there is little support for people wishing to share software between different projects. Suppose one is writing a new system that needs an implementation of B-trees. In all likelihood, one or more implementations of B-trees lie buried deep inside some free-software package, such as Red Hat LINUX. But it is difficult to reuse such a B-tree–implementation, that was developed as part of one system, in a new system.

Software reuse requires effort on the part of both the producer of code that is intended to be reused as well as the consumer of such code. A potential consumer of reused code faces a collection of difficulties. It is difficult to find a small piece of code of desired functionality in a large mass of existing code. That code may be scattered in different files, in different directories, on different machines, on different networks. It may be in different languages, may be stored in different archive formats, and may use a plethora of different tools for managing the build process. It takes extra effort to retrieve such code. Once you find a desired piece of code, it is difficult to separate a small piece of code (and all of the dependent code) from one large monolithic system for use in a different system. Dependent function definitions, variable declarations, structure definitions, class definitions, etc. may be scattered in different files in different directories. You may need to reengineer the process used to build the separated code. Once you separate a desired piece of code, it is difficult to understand that code and adapt it to a new use. Once you adapt a desired piece of code, it is difficult to combine that code with other pieces of code culled from other sources. You may need to rename identifiers to avoid name clash. Version class is a severe form of name clash. A new program p may wish to call two functions f and g, culled from different sources, that use two different but incompatible versions, h1 and h2, of the same function h, respectively. You may need to systematically rename all identifiers in a package to avoid version clash. Finally, all the effort expended to find, retrieve, separate, build, understand, adapt, and combine existing code in a new system must be redone as that existing code changes (perhaps incompatibly).

A potential producer of reusable code faces a corresponding collection of difficulties. Developers must expend additional effort to provide their code to others in a way that facilitates reuse. It takes extra effort to package a piece of code as a library or module, to document a piece of code, to release a piece of code so that other users can access it, to advertise a piece of code so that other users are aware of its existence, to support a piece of code that has other users, and to track users of a piece of code as that code changes. The existence of other users acts as a deterrent to code improvement that requires incompatible changes.
The current methods for sharing source code cannot solve this problem. Existing tools, such as CVS, support a
distributed team of developers working on the same project. There is little support for people wishing to share software
between different projects. Tools like CVS support coarse-grained sharing of source code at the level of monolithic
packages. Coarse-grained source-code sharing, however, does not support code reuse. We need fine-grained source-
code sharing, at the level of individual function definitions. This will allow many people to share code between many
different packages.

2 The Proposed Solution and its Impact

We propose a novel infrastructure, called OCTOBER, to address these issues. The overall goal of OCTOBER is to
increase software-development productivity by supporting both the producer and consumer of reusable code. From the
perspective of the consumer, the goal is to make it as simple as possible to find, retrieve, separate, build, understand,
adapt, and combine existing code instead of writing it from scratch. From the perspective of the producer, the goal is
to make it as simple as possible to provide software in a way that facilitates reuse and eliminate the extra effort needed
to package, document, release, advertise, support, and track reusable code. The same OCTOBER infrastructure can
be used to support code reuse in industry, the scientific community, and the free-software community.

OCTOBER will be a world-wide fine-grained shared source-code repository. Users will be able to search the
repository to find existing code of desired functionality; retrieve, modify, and enhance that code; and release new
code into the repository. The goal of OCTOBER is simple: to enable and foster software reuse on a world-wide scale
and allow individual programmers to leverage the efforts of a huge world-wide pool of programmers. If OCTOBER
is successful, it will create a code base of unprecedented functionality, yet do so by efficiently utilizing the existing
software-development manpower resources.

OCTOBER will be organized like the Web. Just as the Web is a world-wide distributed repository of hyperlinked
documents (i.e. Web pages), OCTOBER will be a world-wide distributed repository of function definitions, hyperlinked
at call sites. Just as Web pages are named with URLs, function definitions will be named with version identifiers in
OCTOBER. Just as there are Web browsers and Web servers that communicate via HTTP, there will be OCTOBER
browsers and OCTOBER servers that communicate via the OCTOBER protocol. Just as Web servers can cache Web
pages to improve performance, OCTOBER will incorporate caching and mirroring to provide high performance. Just
as there are search engines and other programs that crawl the Web, OCTOBER will have the ability to search the
repository to find functionality and crawl the repository to assemble all the requisite functionality into a compiled
program. Just as Web pages are written in HTML, function definitions will be stored in OCTOBER as abstract syntax
trees (ASTs). And just as the Web accommodates new document formats with DTDs, OCTOBER will accommodate
new programming languages with programming language definitions (PLDs).

The same OCTOBER infrastructure can be used to support code reuse in industry, the scientific community, and
the free-software community. Just as one can build both an open, world wide, public Internet and a secure, corporate
wide, private Intranet using the same Web infrastructure, one can build both an open, world wide, public repository
and a secure, corporate wide, private repository using the same OCTOBER infrastructure.

3 The Need for a New Infrastructure

The existing Web infrastructure, however, is inadequate as a basis for OCTOBER. First, the Web is inherently
read-only. The tools and protocols for retrieval and publication on the Web are asymmetric:

- One cannot publish Web pages using the same browser that retrieves Web pages.
- One retrieves Web pages using a client-server architecture but does not publish Web pages using such an archi-
tecture.
- One retrieves Web pages using a standard API but does not publish Web pages with such an API.
- One can create portable automatic tools for retrieving Web pages but cannot create such tools for publishing
Web pages.

This suffices for the Web because most users retrieve, but do not publish, Web pages. In contrast, OCTOBER users will
be programmers who need to retrieve as well as publish source code. Thus the OCTOBER protocols and browser will
support symmetric retrieval and publication. Second, publication on the Web is transient. Web pages can be deleted,
modified, or moved. What you retrieved yesterday may no longer be there today. While this is annoying for Web
users, it would be intolerable for OCTOBER users. An OCTOBER user would not be willing to use someone else’s code
if it may disappear tomorrow. Or worse, change tomorrow in a way that breaks the program or erroneously changes
the result that it produces. OCTOBER users will be willing to hyperlink to each other’s code only if there is a guarantee
that what you built yesterday can be built today.

These fundamental differences between document and software repositories motivate the design of OCTOBER as
distinct from the Web. The OCTOBER browser will operate as an editor and programming environment, not just as
a previewer. The OCTOBER protocol will support publication of new function definitions, not just retrieval of existing
ones. And the semantics of the OCTOBER repository will be write-once, with no modify and no delete. Just as
OCTOBER will support fine-grained source-code sharing, it also will support fine-grained versioning. Editing oper-
ations will create new versions of function definitions rather than modify existing versions. Fine-grained versioning
unifies the concept of versioning with unbounded undo and redo. OCTOBER will contain an inherent version-control
mechanism that will allow retrieval of any version of any function that ever existed as well as building any version of
any software package that ever existed. Nonetheless, the repository will efficiently store the many versions created by
fine-grained versioning as difference lists.

The organization of OCTOBER is designed to greatly ease the task of finding, retrieving, separating, building,
understanding, adapting, and combining existing source-code fragments. The ability to search the repository will
allow users to find existing implementations of desired functionality. Organizing the repository as a hyperlinked col-
lection of individual function definitions will make it easy to retrieve and understand such existing code. Hyperlinks,
combined with the fine-grained nature of the repository, will make the build process easy by allowing crawlers to sep-
arate the required dependent components that are needed to support the desired functionality from those components
that are not needed to support that functionality. Storing function definitions as ASTs will allow crawlers to rename
identifiers to avoid clash and make it easy to combine existing components. Fine-grained versioning, along with the
write-once, no-modify, and no-delete architecture of the repository, will make it easy to adapt existing code to new
uses without impacting existing systems. Designing the OCTOBER browser as an editor and programming environ-
ment, instead of a read-only viewer, will allow users to easily provide the code that they write and eliminate the extra
effort needed to package and release reusable code. The ability to search the repository will eliminate the extra effort
needed to advertise reusable code. The hyperlinked organization of the repository will make it easier to document
reusable code. The ability to invert hyperlinks will eliminate the extra effort needed to track new users of existing
code and make it easier to support reusable code.

4 Design Principles

A key design principle of OCTOBER is preference neutrality: the tool should adapt to the preferences of the user
and not vice versa. Unlike Sun’s JAVA or Microsoft’s C# and CLR, OCTOBER will be neither a new programming
language nor a new compiler. It will support existing programming languages and compilers. It will be possible to
extend OCTOBER to support a new programming language simply by writing a PLD for that language. It will not
force any programming style on the user. The objective for OCTOBER is to allow programmers to retrieve, view,
and enhance each other’s code. Different programmers have different styles and preferences. They would be hesitant
to use each other’s code if they dislike the style in which it was formatted. Thus source code will not be stored as text in
the OCTOBER repository. Rather, it will be stored as abstract syntax and rendered on-the-fly by the browser according
to the user’s preferences. PLDs will separate the definition of the abstract and concrete syntaxes of a programming
language and provide bidirectional mapping between the two. Users will be able to augment the PLDs with their own
abstract-to-concrete syntax mappings. This will allow users to change the specification of such things as colorization,
indentation, line breaks, and even the concrete syntax itself. For example, one user will be able to write and view code
in SCHEME-like syntax while another user will be able to view and modify the same code in C-like syntax. Similarly,
one user will be able to write and view code with the comments and identifiers appearing in English while another
user will be able to view and modify the same code with the comments and identifiers translated to Chinese.

Storing source code as ASTs rather than as text will enable more powerful approaches to code indexing and
search. Current approaches to code indexing and search use string-matching on program identifiers and comments.
New approaches are envisioned that perform large-scale incremental distributed persistent static analysis, such as type
inference and flow analysis, on the entire repository and use the results of this analysis to support code indexing and
search based on the semantics of the code in the repository, rather than its surface syntax. Static analysis of the entire
code base has the potential not only of providing semantic-based retrieval but also of enhancing the reliability and
maintainability of the code base.

One lesson learned from the Internet is that protocols survive, particular implementations of clients and servers that use those protocols do not. Few people still use the PDP-10 versions of TELNET and FTP yet everyone still uses TCP/IP. Likewise, it is expected that the protocols developed by this research, the OCTOBER protocol and the format of version identifiers, ASTs, and PLDs, will survive, not the prototype implementations of the OCTOBER browser and server. It is envisioned that, over time, others will build browsers and servers that will be compatible with the OCTOBER protocol. Nonetheless, prototype browsers and servers must be built to evaluate the approach and facilitate its adoption.

5 Current Status

A prototype of OCTOBER has been built. An initial specification of the OCTOBER protocol and the format of version identifiers, ASTs, and PLDs has been written. A prototype browser and server have been built that support searching for, retrieving, editing, and compiling source code. The browser dynamically renders source code to the colorization, indentation, line break, and concrete-syntax preferences of the user. It can crawl the repository to collect all of the function definitions needed to compile a program. PLDs have been constructed for SCHEME and C. A PLD for JAVA is almost complete and a PLD for C++ is under construction. The PLDs include the ability to import existing text-based code into the abstract-syntax–based repository. This has been used to import several thousand lines of SCHEME and C source code into the repository for testing and evaluation. Section 9 gives a brief demo that illustrates this prototype in operation.

6 Future Work

Future work will address the following issues, inter alia:

1. PLD support for C, C++, and JAVA is important for wide-spread adoption of OCTOBER. Complete PLDs exist for SCHEME and C while PLD support for JAVA is almost complete. One major focus of research over the coming year will be completion of the JAVA PLD and construction of a PLD for C++. PLDs for additional languages will be constructed in the future.

2. Web users tolerate delays of many seconds when following hyperlinks. To be used as an editor and programming environment, OCTOBER must support instantaneous retrieval from the repository. This requires some form of local caching and mirroring of the remote distributed repository. The initial prototype OCTOBER server did not support caching and mirroring. A new prototype server is under construction that adds caching and mirroring support. This new prototype implements caching and mirroring with a totally decentralized peer-to-peer architecture, without a central server, that will allow new repository servers to come on-line by contacting any existing server, tolerate existing servers going off-line, and bring servers back into sync as they come on line again or for the first time.

3. While OCTOBER is envisioned primarily as a public mechanism for sharing source code among a world-wide community of users, a use is also seen for private source-code repositories within smaller organizations such as companies, schools, and government agencies. Hierarchal repositories are planned to support projects with a mixture of public and private code.

4. Many security issues arise when contemplating a world-wide writable repository such as OCTOBER. Even a write-once repository with no delete and no modify can be vulnerable to denial-of-service attacks and malicious code inserted into the repository. A full discussion of these security issues is beyond the scope of this proposal. Future work will attempt to address these issues.

7 Achieving Impact

We expect that OCTOBER will become as pervasive as the Web among software developers and researchers. Moreover, we expect that it will become the single most widely used software-development environment. In doing so, it will dramatically change the way software is designed and disseminated. This will lead to a significant increase in software-development productivity. A four-stage roll-out is planned to achieve these objectives. In stage one, OCTOBER will
be used extensively within our own research group to test its efficacy as a programming environment. In stage two, OCTOBER will be released to a small group of approximately a half dozen colleagues at remote locations to test the caching and mirroring protocols of the server. In stage three, OCTOBER will be released to a larger group of approximately one hundred researchers to test scalability. In stage four, a general release is planned. It is anticipated that feedback from each stage will guide a redesign of the protocols and reimplementations of the browser and server. As part of such redesign and reimplementations, tools will be developed for migrating the repository across such changes.

We anticipate that the early adopters of this infrastructure will be the scientific and free-software communities. Stage three will introduce OCTOBER to a small select subset of the scientific community, while stage four will make OCTOBER available to the scientific and free-software communities at large. We expect that, after use of OCTOBER becomes widespread in the scientific and free-software communities, there will be compelling pressure for industry to adopt OCTOBER as well.

8 Why Fundamental Research is Needed to Build this Infrastructure

On the surface, it may appear that the construction of the OCTOBER infrastructure is a development project that requires little novel research. This is not true for several reasons. The first is the sheer size and scope of the intended infrastructure. New protocols, algorithms, and system architectures will be needed to allow OCTOBER to scale to support millions of users and allow them to navigate a code base containing billions (and perhaps trillions) of lines of source code. Especially when users will expect instantaneous retrieval and effective methods for quickly finding a precise desired piece of software in such a large code base. Second, since we expect OCTOBER to become a pervasive long-lived infrastructure, we wish to plan ahead and think through the impact of each design decision so that the resulting infrastructure has as few fundamental flaws as possible. The Internet suffers from the fact that it was not designed from the ground up with security in mind. The Web suffers from the fact that it was not designed from the ground up to prevent stale links. It would have been easy to remedy such flaws in the Internet and the Web during their initial design yet it is nearly impossible to remedy such flaws now that they have attained widespread use. In both cases, we are be forced to live with, and work around the limitations of, prematurely adopted standards that must be retained for compatibility. We wish to avoid such mistakes, as much as possible, when designing OCTOBER. Third, we are making at least three radical choices in the design of OCTOBER:

fine-grained storage and sharing: We are making the unit of sharing as small as possible, organizing programs as collections of individual function definitions stored in the repository. With the exception of the MIT LISP machine, the Xerox INTERLISP, SMALLTALK, and CEDAR/MESA environments and perhaps a few others, programs have been almost universally organized around coarser-grained units such as files, directories, libraries, and modules. There is little experience adopting a more fine-grained approach to software organization with more mainstream languages, tools, and users that OCTOBER intends to support.

storing and editing structure: We intend to store code in the repository as ASTs and have users edit code with a structure editor. With the exception of Xerox INTERLISP, the Cornell/GrammaTech Program Synthesizer, the Apple DYLAN environment, and perhaps a few others, programs have been almost universally stored and edited as text. To our knowledge, except for the GrammaTech Synthesizer Generator, there has been no attempt to build a programming environment that stores and edits code as ASTs that supports multiple programming languages in an extensible fashion.

fine-grained versioning: We intend to make the unit of version control as small as possible, implicitly creating a new version with each atomic change to the individual function being edited. Existing version-control systems archive entire systems, not individual functions, and do so only when asked.

We expect these radical design choices to lead to radical new ways of thinking about software development. Finally, there is room for substantial collaboration with researchers in at least two other areas on longer-term facets of OCTOBER:

compilers: The compiler community has long investigated static-analysis techniques such as type inference and flow analysis. In the past, these have been applied to program verification and optimization. OCTOBER introduces a new application of these techniques: guiding search engines to find code with desired functionality. It also introduces new challenges: scaling these techniques to a huge distributed persistent but dynamically changing repository.
natural-language processing: The presence of documentation along with the source code in the repository offers several possibilities for innovative research in natural-language processing. This includes grounding the semantics of such documentation in the semantics of the program for purposes of guiding the retrieval of desired source code based on the combined meaning of the code and documentation as well as automatically translating the identifiers of the source code or the documentation into other natural languages.

The fact that static analysis and natural-language processing can both be applied to both retrieval and translation may lead to fruitful collaboration between these otherwise disjoint research communities.

9 A Brief Demo

Figures 1 through 12 contain a brief demo of the operation of the prototype implementation of OCTOBER. This demo assumes that two existing libraries, the KHOROS image-processing library and the V4L library that encapsulates statements, in addition to calling functions from the repository, as shown in figure 10. Like all editing operations in the VIDEO4LINUX API, have been preloaded into the repository using the ability to import existing text-based code. This demo illustrates how it is possible to reuse the code from these libraries to write a new program that uses the V4L library to grab an image and then uses the KHOROS library to perform edge detection on that image.

The user can search the repository by typing a query, such as video c-s, which gives three ‘hits,’ as shown in figure 1. The user can then click right on one of these hits, such as v4l_open_video, and retrieve the source code for that function, as shown in figure 2. The code is dynamically rendered on the fly as to the user’s colorization, indentation, line-break, and concrete-syntax preferences. Blue identifiers indicate hyperlinks to other definitions in the repository. The user can click right on any of these hyperlinks, such as v4l_get_width, and retrieve the source code for the function referenced at the call site, as shown in figure 3.

The user can create a new function definition by clicking on the (){} keyword to create the skeletal definition, clicking on the int keyword to specify the return type, typing c-a c-k main RET to change the name of the function from the default, clicking left on the space between the specifier place holder and the declarator place holder to select the unfilled argument, clicking on the Delete button to delete that unused argument, and clicking on the declaration-or-statement place holder to select that place holder for further editing. This yields the result shown in figure 4.

The user can create a local-variable declaration by clicking on the <declaration> keyword to create a skeletal declaration, typing xvim c-s to search the repository for structure definitions, clicking c-left on the xvim-age hit to fill the specifier position of the declaration with a reference to that structure declaration, clicking on the * keyword to create an indirect declarator, clicking on the Delete button to delete the place holder in that declarator for the unused specifiers, clicking left on the declarator place holder to select that place holder, clicking on the identifier keyword to fill the declarator place holder with an identifier, and typing c-a c-k image RET to change the name of the identifier from the default. This yields the result shown in figure 5. A similar sequence of keystrokes and mouse clicks can create a second local-variable declaration, as shown in figure 5.

The user can call a functions in the repository by clicking left on the ; keyword to create a skeletal statement, and clicking c-right on the v4l_open_video hit to insert a call in the skeletal statement to the function indicated in the hit. This sequence of operations is illustrated in figures 6 through 8. Similar sequences of keystrokes and mouse clicks can create additional calls to functions in the repository as illustrated in figure 9. Note that when a function in the repository takes arguments, the skeletal call is created with place holders for those arguments as illustrated in figure 9. Simple sequences of keystrokes and mouse clicks can fill in the arguments to those calls. The user can create arbitrary C code, such as for statements, in addition to calling functions from the repository, as shown in figure 10. Like all editing operations in OCTOBER, skeletal for statements are created with empty place holders for subexpressions that can be filled in later.

By performing of a small sequence of editing operations, all of the flavor described above, the user completes the definition of the desired function as illustrated in figure 11. Note that this code is automatically stored in the repository in a fully hyperlinked fashion. It hyperlinks to existing code and future code can hyperlink to it. Furthermore, every intermediate stage of the editing process is automatically stored as well and can be retrieved if desired. The user can compile this code simply by clicking on the Compile button. This crawls the repository to retrieve all code referenced both directly and indirectly by the code the user just created and builds an executable. While not shown in this example, cross-language function calls are supported. Calling code in a different language is done in the exact same fashion as calling code in the same language. OCTOBER automatically generates the necessary cross-language declarations and linker instructions. Finally, running the above executable produces the images shown in figure 12.
ftp://ftp.ecn.purdue.edu/qobi/october-edge-demo.ps contains a more elaborate version of this demo while ftp://ftp.ecn.purdue.edu/qobi/october-demo.ps contains demos of other functionality of OCTOBER.

10 Summary

We plan on building a world-wide distributed source-code repository that supports fine-grained sharing of source code. It will be organized like the Web with a browser-server architecture that supports retrieving and editing hyper-linked code. From the perspective of the consumer, it will make it as simple as possible to find, retrieve, separate, build, understand, adapt, and combine existing code instead of writing it from scratch. From the perspective of the producer, it will make it as simple as possible to provide software in a way that facilitates reuse and eliminate the extra effort needed to package, document, release, advertise, support, and track reusable code. This will reduce the duplicate effort to re-implement existing functionality and amplify the productivity of the existing software-development manpower pool. We anticipate that this infrastructure will become as pervasive and widely used as the Web in the scientific community, the free-software world, and ultimately industry. We expect that when deployed, it will dramatically influence the software-development process.
Figure 1: A demo of the prototype of OCTOBER in operation—part I
Figure 2: A demo of the prototype of OCTOBER in operation—part II

```c
/* Function Definitions */

#include <IdiographicFrame.h>

bool OCTOBER::OCTOBER()
{
    return true;
}

void OCTOBER::OCTOBER()
{
    int O_RDONLY = 0;
    v4l_fd = open("/dev/video0", O_RDONLY);
    if (v4l_fd < 0)
        perror("v4l open failed");
        exit(-1);
    v4l_width = v4l_get_width();
    v4l_height = v4l_get_height();
}
```

*Function Definition* v4l_open_video

*Function Definition* v4l_close_video

*Structure Declaration* video_window
Figure 3: A demo of the prototype of OCTOBER in operation—part III

```c
/* Function Definitions */
void v41_open_video(void)
int Q_RDONLY = 0;
v41_fd = open("/dev/video0", Q_RDONLY);
if (v41_fd < 0) { perror("v41 open failed"); exit(-1); }
v41_width = v41_get_width();
v41_height = v41_get_height();

int v41_get_width()
{struct video_window video_window;
 int VIDIOCGWINS;
 VIDIOCGWINS = -1435692279;
 memset(&video_window, 0, sizeof(struct video_window));
 if (ioctl(v41_fd, VIDIOCGWINS, &video_window) == 0)
 { perror("v41 ioctl VIDIOCGWINS failed"); exit(-1); }
 return video_window.width;

```
Figure 4: A demo of the prototype of OCTOBER in operation—part IV

```c
int main(void) {
    // declaration or statement
```

Figure 5: A demo of the prototype of OCTOBER in operation—part V

```c
int main(void) {struct xImage *image;}
```

```
C Structure Declaration xImage
C Function Definition v41_close_video
C Function Definition v41_open_video
C Structure Declaration video_window
```
Figure 6: A demo of the prototype of OCTOBER in operation—part VI
int main(void) { struct xvimage *image; int x, y; expression; }

C Structure Declaration xvimage

C Function Definition v4l_close_video
C Function Definition v4l_open_video
C Structure Declaration video_window

Figure 7: A demo of the prototype of OCTOBER in operation—part VII
Figure 8: A demo of the prototype of OCTOBER in operation—part VIII
Figure 9: A demo of the prototype of OCTOBER in operation—part IX

```
int main(void)
{
    struct xvimage *image;
    int x, y;
    v4l_open_video();
    v4l_restore_factory_settings();
    v4l_allocate_frame();
    v4l_get_frame();
    v4l_write_pgm("original.pgm");
    image = create_image(height, width);
}
```
Figure 10: A demo of the prototype of OCTOBER in operation—part X
Figure 11: A demo of the prototype of OCTOBER in operation—part XI
Figure 12: A demo of the prototype of OCTOBER in operation—part XII