Radiation Therapy Planning: an Uncommon Application of Lisp *

Ira J. Kalet, Robert S. Giansiracusa, Craig Wilcox, and Matthew Lease

Department of Radiation Oncology and
Department of Computer Science and Engineering
University of Washington, Seattle, WA

Abstract

We used Common Lisp to build a complex and powerful interactive graphics simulation system called "Prism", for planning radiation therapy. Special features of Common Lisp that we used to advantage include: lexical closures, the Common Lisp Object System (CLOS), and the Common Lisp binding to the X window system (CLX). We use events, indirect invocation and mediators to achieve modularity. Some of the components of Prism are: a contoured volume editor, computed medical images, a rule based function to generate target volumes, and a radiation dose computation function. To achieve fast floating point computation in the latter, we applied both generic and vendor specific optimizations. The result is a system that is routinely used in the University of Washington Cancer Center, by people with no programming expertise. Our experience shows that Lisp is practical, powerful and efficient for interactive graphics, complex modeling and intensive floating point computations such as radiation dose modeling. Additional work in progress includes a medical image server and an interface to an on-line anatomy atlas.

Keywords: CLOS, events, modeling, graphics

1 Introduction

We designed, tested and are using in the University of Washington Cancer Center a complex medical application of Common Lisp, the Prism radiation treatment planning system. Radiation treatment planning (RTP) systems provide modeling of the human body and radiation treatment machines, analogous to computer-aided design systems. Historically RTP programs been written for mainframe computers, time-sharing systems, mini-computers with interactive graphics, and most recently window-based desktop workstations. Radiation therapy planning projects have contributed to developments in computer systems, in addition to having an enormous impact on the level of sophistication of radiation treatment itself. The Prism project demonstrates the effectiveness of using Lisp in a demanding production software environment.

Radiation therapy directed at malignant tumors involves aiming and collimating a radiation beam at the tumor in such a way as to deposit a large amount of energy in the tumor and as little energy as possible to surrounding (healthy) tissue. Figure 1 shows a typical radiation therapy machine. Radiation, such as X-rays (high energy photons), electrons, neutrons and other particle beams, can kill tumor cells by causing ionization of atoms in or around the cell, and in turn, this can result in molecular bond breakage, i.e., damage to the DNA of the cell, or it can produce free radicals, active chemical species, which then attack the DNA. In either case, the goal of radiation therapy is mainly one of solving an energy delivery problem, to direct as much radiation into the tumor as possible consistent with avoiding the healthy tissue.

The remainder of this section describes the radiotherapy planning problem, the role of computer simulation, and a brief review of related work. Section 2 describes some of the design problems in the Prism system, with attention to how we use various characteristic features of Lisp. Section 3 discusses how effective these solutions are in light of four years of experience using the system, fixing problems discovered after deployment, and adding new features. In section 4 we present briefly our work in progress toward future capabilities.

* Communications to: Ira J. Kalet, Radiation Oncology Department, University of Washington, Box 356043, Seattle, Washington 98195-6043 Phone: (206) 548-4107, FAX: (206) 548-6218, E-mail: `ira@radonc.washington.edu`

Figure 1: A radiation treatment machine, with a member of the UW staff positioned as a patient would be, and a therapist to operate the machine. Once the patient and treatment machine are properly positioned, the therapist leaves the treatment room, then turns on the radiation beam, while monitoring the patient via closed-circuit television. This procedure is repeated for each treatment beam direction.

1.1 The use of computers in radiotherapy planning

A modern radiation therapy machine consists of a high energy linear accelerator, producing X-ray (photon) beams or electron beams in the range of 4 to 25 million electron volts (MeV). The machine has a lot of flexibility to aim the beam and shape it in arbitrary ways. The radiation oncologist would like to take advantage of this flexibility by designing a plan that achieves a curative dose in the target region while minimizing the dose to surrounding tissues.

The process of designing good radiation plans requires the use of RTP systems. The basic steps are:

1. Gather clinical and physical data
2. Decide general approach
3. Select radiation type(s)
4. Use computer simulation to configure radiation beams
5. Verify feasibility

Two factors make radiation therapy effective and practical. Understanding these makes clear the role of computer simulation as a step in the design of radiation treatment.

The first factor concerns the physics of radiation beams. For high energy photons, the maximum dose, or energy deposition, is not at the surface, but can be as much as several centimeters below the skin surface. This is illustrated by the graph of dose vs. depth in figure 2.

The second factor concerns the geometric capabilities of the radiation treatment machine. It is possible to deliver the radiation by aiming the machine in each of several directions, turning it on for a short time from each, so that the aggregate dose to the tumor is high, but the dose deposited in the surrounding tissues is spread out, and therefore lower.

Figure 3 illustrates a two-dimensional cross section of a plan showing two radiation “beams” overlapping to give this effect.

Thus the problem of deciding how to treat the patient is one of choosing directions, apertures, and relative amounts of radiation from some num-

Figure 3: A simple radiation treatment plan cross sectional view

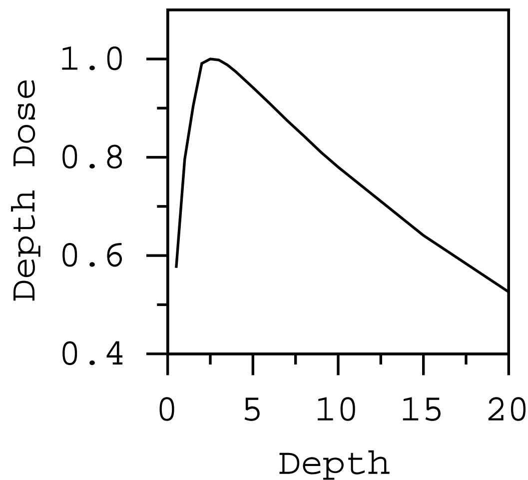

Figure 2: Dose from a single radiation beam vs. depth in tissue

ber of radiation beams. A collection of radiation beams thus specified constitutes a plan.

In figure 3 there are lines showing isolevel contours for various radiation dose levels. These come from interpolation of a grid or array of computed values of radiation dose that would result from the specified arrangement of radiation beams. The formulas used to predict the dose at any point within a patient from any specified radiation beam are partly based on principles of radiation physics and partly empirical, interpolating measured data in standard test conditions. The refinement and testing of such formulas is a large and active area of research in medical physics. A survey of the principles can be found in textbooks, e.g., Khan [1], and the particular formulas used in our RTP system are described in exhaustive detail in a technical report [2]. We describe aspects of the implementation of these formulas in Common Lisp here in section 2.4.

1.2 A short history of RTP system development

As mentioned, methods are well known for computing the physical dose received anywhere in the body from a given beam configuration. However, there is no established method for solving the inverse problem of computing a set of beam parameters, given a desired dose distribution. In typical clinical practice a dosimetrist (a person with special training and experience) uses an RTP system to display the geometry and dose distribution. The dosimetrist looks for regions of inade-

quate dose to the target and excessive doses to sensitive structures, and applies modifications that will correct the problem and thereby improve the plan. Therefore the "configure radiation beams" step above is an interactive generate-test loop.

The importance of having powerful three dimensional radiation treatment planning software tools has been evident for some time [3, 4]. Much recent work has focused on enhancing the delineation of anatomy [5, 6], and providing visualization tools [7].

Early systems ran as teletype applications on dialup time sharing systems, then on dedicated minicomputers with frame buffer displays. As the computer and graphics hardware became faster and cheaper, the medical physicists writing RTP programs became more ambitious about the kind of interactive graphic displays they implemented. Through most of this period, from the late 1970's through early 1990's, most emphasis was on arcane and clever features, rather than on software architecture.

At the University of Washington Radiation Oncology Department our focus has been on software design, both to make the development process efficient and to provide flexibility in use and software evolution. We previously developed two generations of systems that experiment with the organization of treatment planning steps such as delineation of anatomy, manipulation of radiation beams, display of dose distributions, and production of output. The first system [8], which provided two dimensional treatment planning, used a new modular design [9] that made the system very flexible. Its menu system was easily modified or supplemented without changing the programs themselves. It allowed the user to add or change any data items in the treatment plan at any time rather than enforcing a sequence of operations on the user. The second system [10, 11] provided high resolution display and allowed the user to put on the screen multiple plots displaying plan and image data simultaneously. Integration with computer controlled therapy machines [12] has been part of the project from the beginning.

These first two systems were written in Pascal, for DEC VAX computers running VMS. The graphic display was a Ramtek 9465 frame buffer. It was difficult because there are no facilities in Pascal for defining abstract objects, i.e., classes, and for defining generic functions for those objects. Although we created a workaround for our RTP system, it was inflexible (it was difficult to add new object classes), and hard to understand, because the implementation had much code devoted to object management and function dispatch. This obscured the actual application, the radiotherapy objects and their operations. Even in our earliest publication [8] we recognized that Lisp would be a promising language to use for our design ideas. At that time, though, there was no well supported Lisp system that could be used for our application.

The emergence of Common Lisp as a standard and widely supported commercial product was a radical change. Common Lisp implementations became available that had very efficient compilers, support for the X window system (CLX, the de facto standard Common Lisp binding to the X protocol), facilities for application deployment, and an excellent standard object-oriented programming system, the Common Lisp Object System (CLOS).

The Prism system design emphasizes a previously unimaginable flexibility in the user interface and in the ability to incorporate arbitrary numbers and kinds of radiotherapy related objects in the simulation.

A sample Prism screen is shown in figure 4.

2 Design ideas used in Prism

Prism consists of a number of modular components, including a graphical user interface building kit, an implementation of abstract behavioral types and relationships, a graphic rendering subsystem, a scheme for storing simulation data in files for later retrieval and display, control panels for user manipulation of radiotherapy objects, special tools such as a rule based reasoning system, and a radiation dose computation component. We describe the implementation issues of some of these in the following subsections.

2.1 SLIK: a lightweight GUI kit

SLIK (Simple Lisp Interface Kit) is a graphical user interface tool kit written in Common Lisp, using CLOS (the Common Lisp Object System) and CLX (the Common Lisp interface to the X window system protocol). The purpose of SLIK is to provide a facility for handling user interaction in an X window environment. It provides a minimal set of facilities for building real applications, including the usual set of user interface devices or

Figure 4: A screen display from the Prism RTP system, showing control panels, medical images and other objects derived from them, as well as graphic representations of radiation beams.

"widgets". It is not intended to completely hide all the details of the X window system, but rather to encapsulate X event processing and provide some basic user interface widgets.

This limited goal is in contrast to more comprehensive systems such as CLIM [13] and Garnet [14]. CLIM provides an abstract drawing model that can be realized on several window systems, not just X. However, CLIM does not provide the completely general desktop model of an application that can be made of multiple active windows. Garnet provides a comprehensive system for building applications. It includes its own object system, an alternative to CLOS. At the time Garnet was developed, CLOS was not highly developed or efficient, but these considerations are moot at present, with efficient and complete CLOS implementations coming standard in Common Lisp systems from most sources. Garnet is large, a consequence of providing a lot of capabilities. One interesting aspect of Garnet is the provision for constraints. These are important in interactive applications that have dynamically interacting components. In SLIK such constraints can be implemented by using abstract behavioral types, events, and mediators [15]. In Garnet, only "one-way" constraints are supported, but the event/mediator strategy used in SLIK is more general as it supports bidirectional constraints and can also represent other kinds of behavioral relationships.

The event scheme used in SLIK is *not* related to X window system event dispatching. In the

SLIK code X window system events are handled independently (and invisibly to the application).

SLIK is not targeted for a particular window manager. SLIK widgets are very spartan, having no fancy shading or style such as that typical of *Motif* [16]. Future implementations of SLIK may change the appearance of the devices, so they have a more polished appearance, but the programmer's interface will not change.

SLIK provides three facilities for the interface builder: a collection of user-interface objects (dials, sliders, control panels, graphical pictures of data, etc.), a function for dispatching X window events to the objects that need to act on them, and a protocol for the objects to interact with each other and with user application code.

The objects in SLIK that are available for use by the interface builder (programmer) are the typical Graphical User Interface (GUI) objects found in most interface tool kits. They are implemented as instances of a class hierarchy. This provides a straightforward way to add new kinds of objects.

The `frame` is the base class that encapsulates a lot of X details, and is able to handle an X event in its window. The kinds of X events that frames handle include pointer entry and exit, pointer motion within the frame's window, button press and release, keystrokes, and window exposure.

A frame may be a simple control such as a dial with a pointer, a compound control such as a dialbox (which combines a dial and a textline), a dialog box which waits for input, or a *control panel* of your own design, a frame with other frames arranged as you wish in the control panel window, possibly including graphic illustrations. A control panel may have smaller control panels as its components, as well as individual controls.

A *picture* is a frame that contains graphical and/or text information that is part of the application, e.g., a graph of some data or an image, or a 3-dimensional rendering of some physical object. A picture can also respond to X events – for example, the picture might include "control points" that can be grabbed so the object may be pulled, stretched or rotated. The SLIK package includes support for several types of "control point" objects.

2.1.1 Events and mediators: functions as first class objects

Within the SLIK tool kit, components may need to notify other components when things change or events occur. For example, when a dial pointer in a dialbox moves, a text representation of the dial setting should be updated, and vice versa. Also, in the application itself, there will be interacting components. A dial may be attached to some physical object in a simulation, for example, and when the dial changes, the simulation pictures must update. One way to handle this is to code explicitly this interdependence of behavior in the objects themselves. This explicit invocation leads to large tangled systems. Object oriented programming languages do not avoid this problem, as explicit mention of particular objects by other objects, as well as generic function names, is still required. Even implicit invocation is not sufficient as this just reverses the dependencies. Instead we use *abstract behavioral types* and *mediators* [17], where the behavioral relationships between objects are external to the description of those objects.

An abstract behavioral type (ABT) defines a class of objects in terms of the operations that can be applied to the objects and in terms of the activities or events the object can announce. One ABT instance can observe and respond to the activities of another by registering one of its own operations with an activity (event) in the interface of the other ABT. This provides a mechanism for one or more objects to be notified when a source object announces an event. The announcement or event interface is part of the object's interface to the surrounding, and *not* an external device or global variable.

Abstract behavioral types are implemented in SLIK by providing events, announcement of events, and mechanisms for registering interest in events. SLIK objects use this mechanism for interaction with each other in addition to providing an event interface to the applications that use them. The attribute accessors and other functions of a SLIK object provide the usual object-oriented way in which external agents act on the object. Events provide a way for other objects to act in response to the announcement of an event associated with an object.

In SLIK, an event slot of an object is just an association list of *(target, action)* pairs. The target is the object that registered interest in the event, and the action is a function (symbol, function object or lexical closure) to be called when the event is announced. As entities in the running system register, they simply add a pair to the list, and as they unregister, the pair is removed. An an-

nouncement simply is an iteration over the list, calling the action function of each pair, passing to it the announcer and the target, and any other useful information. The complete code is shown in figure 5.

```
(deftype event () 'list)

(defun make-event () nil)

(defmacro add-notify (party event action)

  "ADD-NOTIFY party event action
Adds the party, action pair to
the specified event."

  `(setf ,event
      (cons (list ,party ,action)
            (remove ,party ,event
                  :test #'eq :key #'car))))

(defmacro remove-notify (party event)

  "REMOVE-NOTIFY party event
removes the entry for party in event."

  `(setf ,event
      (remove ,party ,event
              :test #'eq :key #'car)))

(defun announce (object event &rest args)

  "ANNOUNCE object event &rest args
applies the action part of each entry
to the party part of each entry."

  (dolist (entry event)
    (apply (second entry)
           (first entry) object args)))
```

Figure 5: The event interface implementation

An example of an ABT is illustrated in implementing objects that include variable numbers of elements. The mathematical notion of a set is the natural starting point. The idea of a set can be supplemented with events that announce when an element is inserted or deleted, thus making the interaction of the set with other objects straightforward and consistent with the rest of the tool kit. SLIK includes a small package, the `collections` package, that implements the `collection`, an ABT that provides this extension of the idea of a set. An excerpt of the implementation is shown in figure 6.

The basic type, `event`, provides a simple one-

```
(defclass collection ()
  ((elements :accessor elements
             :initarg :elements
             :initform nil)
   (inserted :accessor inserted
             :initform (make-event))
             :documentation
  "Announced when an element is inserted.")
   (deleted :accessor deleted
            :initform (make-event)
            :documentation
  "Announced when an element is deleted.")
   ))

(defun insert-element (el coll
                       &key (test #'equal))

  "INSERT-ELEMENT el coll &key test
inserts el into collection coll if not
already present.  The new element is added
at the end, not the front of the list."

  (unless (member el (elements coll)
                  :test test)
    (setf (elements coll)
       (append (elements coll) (list el)))
    (announce coll (inserted coll) el)))

and more...
```

Figure 6: An excerpt from the implementation of collections, mathematical sets with behavior added.

way interface for implicit invocation. In SLIK, as well as in other applications, more complex relationships are sometimes required. An example of such a relationship is the maintenance of a one-to-one relationship between two sets, e.g., a set of objects in a simulation and the set of control panels by which the user can manipulate them. Another example is the case where an attribute of one object must be kept consistent with an attribute of another object, a constraint relationship. Implementing this with events does not avoid the possibility of a circularity or infinite loop.

We implement these relationships by constructing additional objects we call "mediators". The purpose of a mediator is to explicitly and externally express these complex relationships rather than embed them in the design of the related objects. This makes the objects themselves more modular and makes it easy to understand how the relationships work. In some cases, it becomes possible to describe a family of relationships, and thus

reuse the mediator code as well as the code for the object. Behavior abstraction separates the *behavior* of an object from its *use* in more complex structures. Mediators explicitly provide the connections between interacting objects. An example use of mediators, that maintains consistency between sets of objects and the set of views that display them, is shown in figure 7.

```
(defclass object-view-mediator ()
  ((object :reader object :initarg :object)
   (view :reader view :initarg :view)))

(defmethod initialize-instance :after
           ((ovm object-view-mediator) ...)
  (add-notify ovm (refresh-fg (view ovm))
      #'(lambda (med vw)
              (draw (object med) vw))) ...

(defclass object-view-manager ()
  ((obj-set :accessor obj-set
            :initform (make-collection))
   (view-set :accessor view-set
             :initform (make-collection))
   (mediator-set :accessor mediator-set
            :initform (make-collection) ...

(defmethod initialize-instance :after
           ((ovm object-view-manager)
           &key mediator-fn ...)
  ...
  (add-notify ovm (inserted (obj-set ovm))
      #'(lambda (md oset obj)
          ...
          (dolist (vw (elements
                         (view-set md)))
            (insert-element
               (funcall mediator-fn obj vw)
               (mediator-set md)))))
  (add-notify ovm (inserted (view-set ovm))
      #'(lambda (md oset vw)
              ...like above...
```

Figure 7: Excerpt of code that implements the object-view mediator and the object-view manager.

The *object-view mediator* in figure 7 simply connects an object with a view in which it appears. When the view announces its `refresh-fg` event, the object is redrawn in the view. There is nothing special here about using a lexical closure.

Since the number and kind of objects can change during the course of simulation, as well as the number and kind of views, we need a mediator (the *object-view manager*) that creates and destroys object-view mediators as necessary, i.e., when an object is inserted or deleted in the set of objects or a view is inserted or deleted in the set of views. Since the kind of mediator needed might be more specialized than the general object-view mediator here, i.e., it may vary with the kind of object set or view set, the object-view manager *must* use a lexical closure to capture the mediator constructor function from the lexical environment, when registering an action with either of the respective sets.

Because the code is perfectly general, it works anywhere you need such a relationship, and all that is needed in each place is to pass the right object-view mediator constructor function as a parameter to the object-view manager constructor function. Got that?

This is particularly cute, because the local context being captured is a function, `mediator-fn`, not just a variable. It's compact, works reliably, and is not nearly as hard to trace as it looks.

2.2 The graphics pipeline: the use of CLOS multimethods

Prism uses a pipeline design to implement 2-d and 3-d graphics in the X window system environment. In order to achieve the effect of gray scale image display and color graphic overlay planes like the old fashioned frame buffers, while being able to run on a minimal 8-bit display, Prism does 3-d to 2-d projection in software, and also computes a 128 gray level image from the original 16 bit image data, then adds the color graphics to the pixmap containing the image. Although this design does not give the highest performance, and does not leverage any 3-d graphics accelerator hardware that might be present, it achieves modularity, ease of adding new objects, and especially ease of adding hard copy outputs to various devices (currently HP-GL and PostScript are supported).

Graphic objects and images are handled differently in order to deal effectively with a limitation of the X window system and the use of an 8-bit display. Graphic objects include any object in the system which has some graphical representation and may be depicted in a Prism *view*. These include beams, tumors, anatomy, anatomical landmarks, seeds, textual annotation, and locator bars. Images are typically selected or derived from a set of cross sectional images of the patient's body, for example, a Computed Tomography (CT) study.

(CT images are computed from X-ray projections through a cross section of a patient's body.) A view contains a SLIK picture, with a CLX pixmap and a CLX window. The window appears on the screen and the pixmap (referred to here as the *picture pixmap*) is set to be the CLX "background" of the window.

2.2.1 Pipeline components

Prism uses *caching* of intermediate graphical data structures to enhance the efficiency of drawing, and employs *double buffering* to ensure that a series of redrawing operations appears smooth and flicker free. The purpose of this "graphics pipeline" is to be able to update the window on the screen efficiently and without the flicker that might appear if it were erased and redrawn with substantial delay in between those two operations. The pipeline also serves to separate image data and graphic overlay data, while providing the option to perform grey-scale transformations on images before displaying them. Figure 8 illustrates this scheme.

So a view contains a `foreground`, which is a list of *graphic primitives*, and a `background`, which is a pixmap separate from the SLIK picture pixmap. A graphic primitive is a representation of the drawable data corresponding to an object, in a form suitable for input to CLX primitive drawing routines such as `clx:draw-lines`. Graphic primitives are elemental graphic types such as text, polygons, disconnected line segments, represented in screen coordinates. The background pixmap contains an image to be displayed as the background of the view.

The sequence of operations that lead to display of data in a view consists of:

1. From the image data, compute the background pixmap,
2. Transform the object data from real space into graphic primitives, which are then stored in the foreground list.
3. Copy the *background* pixmap containing image data to the picture pixmap associated with the view (or set the picture pixmap to all black pixels if no image is to be displayed).
4. Draw the graphic primitives into the picture pixmap.
5. Copy the picture pixmap into the picture window (or alternatively erase the window — since the pixmap is the window background its contents will appear in the window on erasure).

The generic function *draw* implements some of these operations. The *draw* function takes two required arguments, an object to draw and a view in which to draw. Three sets of draw methods are used to display graphical information — one that updates the graphics cache (graphic primitives) derived from the graphic objects, one that generates or processes images (or sets of images) to fill the background pixmap, and one that writes the contents of the cache into the part of the view that gets displayed. The *draw* methods for images update the view's background pixmap with a CLX image derived from the Prism image data and the view specifications.

It appears in this design that the pipeline is memory intensive, and there is a lot of overhead because pixmaps are copied wholesale on each view update, even though in many situations only a single object's graphic rendition needs to be changed. In practice, copying pixmaps is an operation that typical graphics hardware performs extremely well, and it is optimized in many X server operations. If the X server caches pixmaps in the server or the hardware, this provides very impressive update rates when running Prism on a host that is remote from the display. In a demonstration using a display at the University of Chicago, running Prism on a system at the University of Washington, we were able to get Prism to do several updates per second, even though the Prism program was running on a host computer almost 2,000 miles from the display computer.

2.3 Storing objects in files: Lisp, CLOS and MOP make things generic

Prism includes functions that read and write ASCII files, with text representations of objects such as patient cases. The text representation for all objects follows a general template, based on the idea of keyword-value pairs. Printed representations of these class instances and associated data are written to and read from files by two functions, `put-object` and `get-object` respectively.

A design goal for this file storage scheme was to have files that could be manipulated with an ordinary text editor if necessary, and therefore easily human readable. We also wanted to avoid writ-

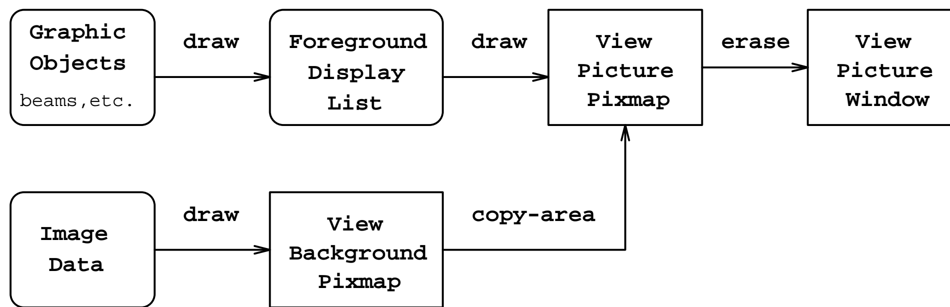

Figure 8: The graphics pipeline and double buffering

ing special formatting and parsing code each time a new kind of object was added to the system. The scheme we decided on uses Lisp's ability to parse the printed form of an object and determine its type, and similarly to produce a printed form dependent on its type. The fact that Lisp can determine types at run-time is essential in this design. We also use a little of the Meta-Object Protocol (MOP) to let the class definitions provide slot names.

A Prism data file can be understood as a text stream that can be processed by the Common Lisp `read` function, and therefore consists of a series of printed representations of Lisp objects, mostly symbols, numbers, strings, and possibly lists and arrays. Symbols must be qualified by package names. This is automatically done when the system writes data to a file, if the package in which each symbol is interned is *not* the current package and the symbol is *not* accessible in the current package (i.e., it may be external or internal in another package, but not imported into the current package). This is the normal way to run Prism.

The text representation of an object begins with a symbol that is the class name of the object. Following that are pairs of items consisting of a symbol naming one of the object's attributes, and a value for that attribute. The value may be a readable Lisp object (a number, a symbol, a list or a modest sized array), a large binary array, for which a special rendition is provided, or a construct that should be interpreted as a list of objects (rather than a list of simple values). Figure 9 shows an excerpt from one such data file.

Since the Lisp reader is insensitive to the amount of "whitespace" between items, the file may include whitespace and "newlines" as convenient to make it more easily readable by humans. The Lisp reader generally converts lowercase input to uppercase, so except for quoted string data, we assume case insensitivity. The system must be configured so this works, i.e., the readtable is the default as described in Steele [18, page 540].

Strings are delimited by the double quote (`"`) character. Unquoted strings in files are read as symbols.

The end of the data for the object is demarcated by the keyword `:end` with no value following it.

The functions `get-object` and `put-object` call a generic function, `slot-type`, to determine how to read object slots from and write them to the file system. The `slot-type` function takes two arguments, the object and the slot name. For each class in Prism, that will be read from or written to the file system, we provide a method, as necessary. If the class inherits slots from any superclass, then that method calls `call-next-method`, since the slot asked about may be inherited from an ancestor class. The default *slot-type* method returns type `:simple`, so slot names of that type do not need to be explicitly mentioned in the method. If all the slots are `:simple`, no special *slot-type* method is needed.

When an object is being read from or written to the file system, the slot type of the slot being read will cause the following behavior on the part of `get-object` and `put-object`:

`:simple` Use the standard Common Lisp functions `read` and `write` to read and write the slot's data.

`:bin-array` In this case the information in the file consists of a list of three or four items, as follows: a string representing a filename, and two or three numbers, representing the dimensions of the 2 or 3 dimensional binary

```
PRISM:PATIENT
  PRISM:COMMENTS  ("")
  PRISM:DATE-ENTERED  "6-Jun-1994 12:10:25"
  PRISM:IMMOB-DEVICE  PRISM::NONE
  PRISM:ANATOMY
    PRISM:ORGAN
      PRISM:DENSITY  1.0
      PRISM:CONTOURS
        PRISM:CONTOUR
          PRISM:DISPLAY-COLOR  SLIK:MAGENTA
          PRISM:VERTICES ((-11.389 -4.717) (-11.322 -1.146)
                          (8.761 -0.067) (7.278 0.202) (5.863 0.0)
                          (-10.378 -6.672) (-11.254 -5.054))
          PRISM:Z  5.0
        :END
        PRISM:CONTOUR
          PRISM:DISPLAY-COLOR  SLIK:MAGENTA
          PRISM:VERTICES ((9.569 2.628) (6.806 4.717) (4.38 5.189)
                          (-2.359 6.132) (-5.256 6.267) (-7.346 4.919)
                          (9.03 2.763))
          PRISM:Z  9.0
        :END
        etc...
      :END
      PRISM:DISPLAY-COLOR  SLIK:BLUE
      PRISM:NAME  "Liver"
    :END
    PRISM:ORGAN
      PRISM:DENSITY  0.3
      PRISM:CONTOURS etc...
```

Figure 9: A portion of a patient data file

array of data. The binary array data is read from a separate file by a function for that purpose.

`:object` Make a recursive call to `get-object` or `put-object` to read or write the slot's data, which is itself an object,

`:object-list` If the attribute value is a list of objects, each object is read in turn by a (recursive) iterated call to the `get-object` function, and each of those objects follows the same format. The end of a list of objects for an attribute is demarcated by an additional `:end` keyword following the last object in the list.

`:collection` If the slot's data is a collection, the format in the file is the same as for a list of objects, but as each is read, it is inserted into the collection, instead of added to a list.

Not all attributes for an object need to have values in the file, but if there is no value, the corresponding attribute name must *not* be present, i.e., it is an error for the file to contain two attribute names in succession with no intervening value. Note that `nil` is an allowable value for some attributes (not all) and is different from an unbound (or missing) attribute.

All this is independent of anything about radiotherapy. It is parametrized by the class definitions themselves (we use the MOP functions `slot-definition-name` and `class-slots` to get slot names when writing an object to a file), and by providing methods as necessary for `slot-type`.

The resulting file store is not an object oriented database, but suffices for our purposes. Use of an object oriented database might offer some advantages and useful functionality but it would also introduce other problems. This is under consider-

ation for the future.

2.4 Dose modeling: efficient floating point computation in Lisp

The implementation of the Prism dose calculation module is one area that exposed us to both the best and the worst that Lisp has to offer. On the one hand, we have no doubt that the superb programming environment offered by Lisp assists immeasurably in the construction of large and complex software. On the other hand, tweaking floating-point-intensive Lisp code for optimal performance is not easy.

The Prism dose calculation module comprises approximately 3000 lines (about half code and half comments). It implements an algorithm that is fairly straightforward but somewhat elaborate. Essentially, we compute the dose to each point (either in a prespecified set or on a regular 3-dimensional grid) by multiplying together factors accounting for the geometry of the generating equipment, the inverse-square-law divergence of photon flux, and the near-exponential absorption in tissue. The calculation modifies these factors to approximate the effects of tissue nonhomogeneities. If blocks (Cerrobend shielding material) are present in the radiated field, we calculate a component that accounts for radiation scattered to the dose point as if the blocked area were the only area illuminated by the beam and then subtract this component (weighted by the block's absorption) from the dose at that point as if the blocks were not present. If the radiation source has a complex geometry (for instance, as produced by a multileaf collimator which can approximate arbitrary portal field shapes) we use numerical spatial integration to calculate the equivalent rectangular geometry which would yield the same dose to the test point.

Our dose calculation model is not very sophisticated in a theoretical sense – it is basically a complicated table lookup. However, the several numerical integrations and the correction for tissue absorption inhomogeneities (which uses a ray-tracing algorithm) result in code of considerable complexity. This complex code is iterated over each dose point in the point-set or dose-grid. Typical grids may consist of up to a million points. And all this is for a single beam. A typical calculation sums the computation of dose to each point from up to 20 or 30 beams, each calculated independently. Thus a not-uncommon case may involve several dozen table-lookups and tri-linear interpolations to integrate scatter dose to a single point, iterated over 30 million points, resulting in hundreds of billions of floating-point calculations. A dosimetrist might repeat this entire process many times during the formulation of a plan. The computation time varies with the complexity of the plan and may range from a few seconds to several hours.

Since speed of an interactive application is very important, optimization of this code was on the critical path. However, far more important to us initially than getting it fast, was getting it correct. Using Lisp (rather than a more traditional imperative language) helped in two ways. First, the elegance of the language itself (its syntactic simplicity and functional style) greatly eased the initial design and verification of the system. Second, the highly-interactive nature of our vendor's implementation (Allegro Common Lisp from Franz, Inc.) was of tremendous assistance in debugging and testing.

Regarding the language itself, we were repeatedly impressed by the similarity of Lisp code to mathematical notation. While in a formal sense all programs specify Turing machines (or equivalents like Lambda calculus), that specification is especially perspicuous in the case of Lisp. We were able to write initial Lisp implementations of several components of our dose computation module almost as direct transliterations from mathematical notation (flowgraphs and equations) to prototype working code. Verifying the correctness of the implementation at this stage was also easy due to the close correspondence between the mathematical specification and the functional style of our initial Lisp code.

Our previous dose computation module was written in Pascal and ran as a separate process, using Unix interprocess communication (via Lisp streams) to ferry data back and forth. The Pascal code ran fast but was not completely correct. There were certain configurations that produced anomalous results, primarily unexplained points of zero dose in regions where the physics (i.e., the formula) clearly made that impossible. In ten years of use and debugging we were never able to account for (or fix) these anomalies. Our very first Lisp implementation fixed the problem, and these anomalies have never recurred since. (Other anomalies have, but we have always managed to trace them to easily fixed oversights and plain stupid mistakes.)

Of course, the initial implementation ran rather slowly. The current version, now in clinical use for about six months, runs 100 times faster. It is, as far as we can tell, still correct. In fact, it is even more correct, in that we later discovered several points on which our implementation differed from the specification or in which the specification was ambiguous. Fixing them resulted in code which now better represents our original intent than did the first implementation.

The close correpondence between Lisp code and other formal or mathematical styles of description was of great value during the optimization phase of the implementation. We used relatively straightforward source-to-source transformations to speed up the code. Some were simple and language-specific, like converting `mapcar` to `do` (transforming functional application over repeatedly consed intermediate data structures to iterative mutation of persistent state). Others were at the algorithmic level, like replacing linear and binary search with constant-time lookup. Our table-lookups, at the innermost levels of inner loops and therefore the most time-critical, had to use search because our represention of machine characteristics involved non-uniform sampling (for example, dense sampling at beam edge and sparse sampling over flat regions). We converted search to constant-time lookup by precomputing (at table-load time) an access-vector that maps an input value directly to the correct table entry.

While these transformations were easy to analyze mathematically (and we could therefore verify their correctness), there was always the chance that subtle coding errors would go unnoticed. That is where the interactive nature of the Lisp environment (and our vendor's extensions) really shined. It was trivially easy to trace execution paths, insert debugging printouts of critical values, gather timing and function-calling statistics, and do myriad other test probe insertions into the code to verify that optimization was not compromising correctness. We built several test jigs that gathered large quantities of data from test runs of different versions (code with varying levels of optimization). These jigs enabled the automatic analysis of output from test runs so that we could be sure that changes did not introduce errors. Some of the jigs also did graphical display so that we could verify that various algorithms were working correctly – this was especially valuable in studying two components: a polygon-clipping routine (which clipped block outlines to the radiation portal) and the spatial integration used for calculating scatter dose under the blocks. The interactive nature of Lisp (full-featured interpreter combined with fast optimizing compiler, trace features, data inspector, etc) made the rapid-prototyping of these test jigs very easy.

After optimizing at the algorithmic level we did code tuning via declarations. The ANSI standard allows and our vendor's product provides inlining of arithmetic functions and array accessors. Careful and liberal sprinkling about of floating point declarations and the use of type-specific arrays to hold floating point numbers resulted in considerable speedup. On occasion this strategy required passing extra variables as control flags – where a function might return data of more than one type, we had it return them via separate variables and used a flag to indicate which was the "real" value.

Our next effort was to improve the efficiency of memory allocation for floats. In this phase we directly confronted limitations of the design philosophy of Lisp. There is no doubt that many features of the language (like dynamic memory management, typed data objects, generic functions, automatic method dispatch, and presence of source terms such as symbols naming variables in the object code) greatly facilitate the construction of reliable and robust programs. Once the program is debugged, some of them can be compiled away for production versions. But others cannot. The single biggest obstacle we encountered was the boxing of floating point data (that is, the representation of floats as pointers to an allocated chunk holding the actual datum bit pattern).

Our application computes many billions of floating point values during a run of the dose calculation. Only a relative few of them survive as final outputs – the vast majority become garbage soon after creation. This is the ideal situation for generational garbage collection (which is what our vendor's implementation uses). However, even though generational collection is currently the best method of managing storage, we still encountered a huge cost in allocating those billions of floats and in chasing pointers to pass their values to arithmetic primitives. In our attempts at speedup we studied both vendor-specific and generic Lisp-level techniques.

The vendor-specific technique we tried was the use of argument and return-value type declarations enabling the Franz Allegro compiler to produce code that passes floats directly to user-defined functions rather than by allocating a boxed value. It worked well (almost 10

times speedup on certain numerically-intensive routines), but ultimately we decided to abandon this technique as too system-specific. It did not even work in the Linux (as opposed to HP-UX) version of the Lisp from the same vendor – and we hope to run our application with Lisp products from other vendors.

Also, even if we could pass unboxed floats directly to certain user-defined functions, the semantics of the language will not allow this technique for Lisp primitives themselves. For example, suppose we want to pass a list of a few floats from one function to another. The Lisp primitive does not know where its inputs came from or are going; it must generate a list of `cons` cells each pointing to a fully-boxed float. There is no way we can tell Lisp that we don't care about full run-time type tagging because we are just going to hold on to this list for a few nanoseconds before destructuring it. Of course, we can obtain that effect by passing the float values in a specialized float vector, but that still requires all the overhead of array construction and the associated memory management.

We obtained almost equivalent speedup (a factor of about 8 rather than 10) and accomplished the same goal (elimination of float boxing) with a portable generic Lisp solution. We pass float arguments and return values via a specialized float vector. The Lisp standard includes arrays of specialized types (like `single-float`). While not required by the standard, all implementations of which we are aware can implement those specialized arrays by storing the acual bit patterns in the array slots rather than by indirecting through pointers. Thus the boxing problem goes away.

Or at least, it almost goes away. Now we have to stuff all the float arguments for a function into a vector before calling the function, and the function must dereference them. When it starts up, either it must copy these values into local variables or it must indirect through the vector every time it needs to access one of them. This overhead is unavoidable, given the semantics of the language (or so it seems). If one is to get the benefits of generational garbage collection, one must be willing to let the garbage collector move data around during a computation. That means that the compiler cannot optimize memory references to data in arrays by generating memory accesses directly to the cell holding the data. That cell is part of an array that the garbage collector may move at any time; therefore, it must be accessed at least by indexing from the pointer to the array itself. This seemingly minimal overhead of one memory indirection per data access, multiplied by billions of accesses, becomes a significant cost.

2.5 Image projection: another optimization challenge

Prism recently added the ability to view a *projection* of a set of CT images. A CT image represents the X-ray density of a patient in a two dimensional transverse cross-sectional plane. An example CT image of an abdomen cross section is shown in figure 3. An example projected image, through a collection of cross sections of a head as a volume data set, can be seen in figure 10. Thus, a set of these CT images can be used to represent a 3-D grid of density information. For treatment planning, it is quite useful to be able to visualize this data from arbitrary viewpoints. For example, by projecting the CT data to the viewpoint of a radiation beam source, dosimetrists are able to use this data to guide adjustments to the beam geometry. In Radiation Oncology, this generated image is commonly called a Digitally Recomputed Radiograph (DRR) because it simulates an X-Ray (radiograph) of the patient.

2.5.1 The DRR Algorithm

The implementation of a DRR calculation is similar to a ray tracing algorithm. The algorithm is as follows:

For each pixel in the resulting DRR image,

1. Calculate the ray from the virtual camera point to the current pixel.

2. Trace the ray through the 3-D grid to find the CT image voxels which intersect the ray.

3. Calculate the total density for the current pixel by summing the values of the intersected voxels weighted by the length of the ray within each voxel.

Our implementation is very close to the algorithm presented by Siddon [19] except that we do not require a constant distance between the CT images in an image set. This flexibility is useful when a high degree of precision is required around the treatment area and resolution requirements are much lower further from the treatment area.

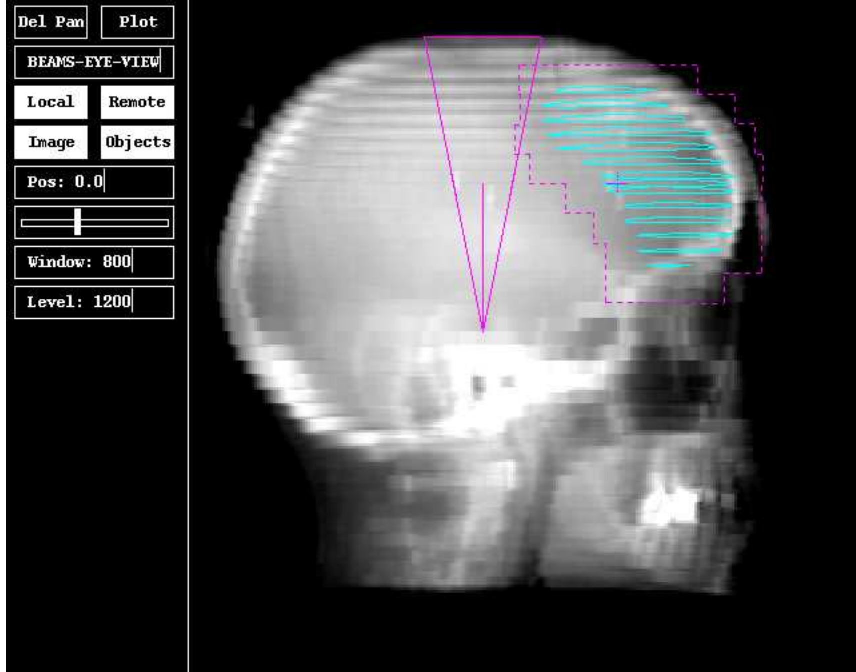

Figure 10: A digital reconstructed radiograph generated from a collection of cross sectional X-ray CT images

2.5.2 Performance tuning

It is important to note that this involves a significant number of accesses to the 3-D image grid. Thus, efficient array access is an important factor for performance. To gain efficient array access in Lisp, the compiler must know both the type of data in the array and the size of the array at compile time. Of course the compiler must also have its optimization settings enabled. This simple change alone made the code about 10 times faster by inlining array references.

The code optimization for this module is not yet complete and there are still a few options to pursue. For example, there is still potential optimization to be done with respect to cache coherency because the 3-D dataset is rather large (tens of megabytes).

2.6 Planning target volumes: a rule based component

As part of the advance toward true three dimensional radiation therapy treatments, the International Commission on Radiation Units recently defined an entity known as the planning target volume (PTV) [20]. While a physician should ideally be able to outline a tumor volume on a patient's treatment planning CT (cross sectional image set) and then irradiate only that volume, certain realistic constraints make this impossible. During a radiation treament, the tumor may move relative to its location on the CT images due to physiologic causes such as respiration, swallowing, stomach motion, etc. Additionally, since treatments are given daily over a period of weeks, further error is introduced from variations in daily treatment set-up. Consequently, in order to ensure treating the entire tumor volume to full dose, an expansion of the tumor volume is defined which takes into account these intra- and inter-treatment uncertainties. This larger volume, illustrated by the dashed line in Figure 11, is the planning target volume.

2.6.1 The PTV model

The Planning Target Volume Tool [21] included in the Prism system generates a PTV using a knowledge base, inference engine, and volume expansion algorithm. It relies on the provided tumor volume as a basis for expansion, and uses symbolic (categorical) information about the tumor location, histology, stage, and extent of patient immobilization to derive the parameters of the expansion.

Planning target volumes are generated by en-

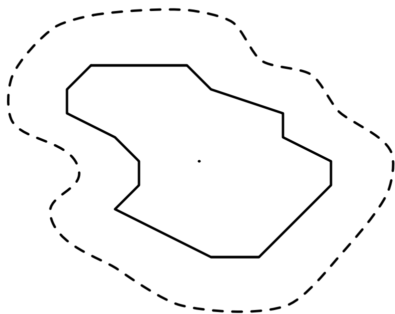

Figure 11: Tumor volume outline in one plane (solid line), together with surrounding PTV (dashed line).

larging the tumor volume cylindrically by a margin Δ in three dimensions. Δ is the root-mean-square combination of the values of several components. Component types are: patient motion (τ_{pm}), tumor motion (τ_{tm}), and day to day setup variation (τ_{su}).

$$\Delta = 1.614 \times \sqrt{\tau_{pm}^2 + \tau_{tm}^2 + \tau_{su}^2}$$

The values of these components are determined by information about the tumor, patient organs and other aspects of the patient's condition. This information is mostly symbolic rather than numeric or geometrical. The types of information used to describe the tumor and patient follow:

T stage: This label, one of the four levels, `T1, T2, T3, T4`, indicates the extent of the primary tumor. The categories are related to size but have qualitative significance towards prognosis and treatment strategy.

N stage: Similary the extent of apparent involvement of regional lymph nodes is graded, `N0, N1, N2, N3`.

Cell type: The cell type is related to the tissue of origin of the tumor cells, and therefore the allowable values may depend on the tumor site. Some typical cell types are: squamous cell, large cell, small cell, adenocarcinoma, lymphoepithelioma.

Immobilization device: This refers to any mechanical device that will be used to restrict the patient's movement during treatment, and to aid in reproducing the precise positioning of the radiation beams each day of treatment. Devices include: mask, alpha cradle, plaster shell, none.

Region: This gives some guidance, for lung tumors, about how much internal motion to expect. Regions of the lung include: hilum, lower lobe, mediastinum, upper lobe.

Much of this information is site specific. The current version of the tool supports only two anatomic sites, lung and nasopharynx, but it is only a matter of gathering more clinical and physical data to add support for other sites.

2.6.2 Implementation of the PTV tool

The PTV tool is implemented as a Common Lisp function within the Prism system, that produces radiotherapy planning target volumes given information about a patient and the patient's tumor.

The relations between the patient and tumor information and the magnitudes of the margin components are represented by *rules*. A *rule interpreter* uses the rules and input data to determine the appropriate margin component values.

```
(define-rule LUNG-ALPHA
  (AND (type-p ?tumor-instance tumor)
       (within-p ?tumor-instance lung)
       (type-p ?pat-immob immob-dev)
       (immob-p ?pat-immob alpha-cradle))
  ->
  (AND (margin setup-error (0.6 0.6 0.6))
       (margin pt-movement (0.2 0.2 0.2))))
```

Figure 12: A sample rule relating patient and tumor information to a value for an expansion parameter

The PTV is generated by expanding the convex hull of the contoured tumor volume by the amount Δ computed after all the applicable rules have been used to determine the τ components. This model and implementation were tested by comparing the generated target volumes with PTV's drawn by experienced radiation oncologists, using standard expert system evaluation methods [22]. The model is sufficiently effective to be useful as an aid in routine treatment planning.

3 Results and discussion

Having four years of real world clinical use of Prism, as well as experience with maintenance and enhancement, we can report that in general the choice of Lisp as the development language has

worked out very well. We were able to build a richly featured application and tune it for high performance, with a relatively small amount of effort. While we have not compiled detailed statistics on the discovery and repair of programming errors (bugs), in general they have not been more frequent than in our previous work, and they have been much easier to find and fix than in the Pascal programming environment.

3.1 Adequacy of features of Prism

In addition to basic 3-D radiation treatment planning capabilities, the Prism system includes unique features, and has some characteristics that are not found in other RTP systems:

1. dynamic random access - there is no enforced policy for the order of treatment planning steps,
2. built-in capability for artificial intelligence tools, including a rule interpreter [23] providing support for experimental development of automated radiation treatment planning algorithms [24],
3. a software tool for automatically generating a planning target volume from a physician drawn tumor volume [21],
4. more explicit attention to quality engineering, better documentation of system behavior and internals,
5. more adaptability, for integration of new modules and outside software, for example, integration of an on-line digital anatomy atlas [25].
6. more adaptability, with respect to diversity of radiation treatment machine types, and integration with computer controlled radiation treatment machines.

3.2 Performance

The two critical areas where run-time execution speed is important are the updating of the graphic displays and the dose calculation. In both areas we achieved acceptable performance relative to our experience with a more conventional programming language and implementation.

On a typical RISC workstation, Prism provides adequately responsive performance in most circumstances. With a few cross sectional views displayed the user can operate the beam control dials and rotate the radiation beams, getting update rates of 5-10 updates per second, even with all the pixmap copying, and with image display in the background. The projected beam's eye views update slightly slower, but still fast enough for interactive use. The one area where performance lags is the gray scale mapping of images and the calculation of projected (DRR) images. We know from examining the code that there is still room for performance improvements there, and we intend to pursue this.

Prism does not have surface-rendered displays of organs, etc. but this is important and we are pursuing this now. Typically applications achieve high performance in 3-D surface rendering by using specialized libraries such as OpenGL. This is under investigation but it is not obvious whether this will really be better than writing reasonably efficient implementations of standard tiling and rendering algorithms in Lisp.

Much as CLX provides a reasonably efficient interface to 2-D graphics in the X environment, it would be a tremendous contribution for someone to provide a highly tuned implementation of 3-D graphics for Common Lisp, especially one that can flexibly take advantage of graphics acceleration hardware and still be compatible with X. This involves to some extent the current debate over the future of extensions to X such as PEX for 3-D graphics.

It must be said that higher performance could be obtained by leaving the X environment and using a system that leverages local hardware. The importance of network operation of Prism is sufficient that we will not pursue this path. This is not a language issue.

Our current Lisp implementation of the dose calculation runs about 20 percent faster than did the old Pascal version, and it is 100 percent more correct.

3.3 Project management

We have gained experience with software engineering methods that can greatly reduce the effort involved in building a treatment planning system. The use of abstract behavioral types and mediators, with Entity-Relationship modeling, helped keep the design of Prism at a high level and helped

untangle many possible design traps. We used both a waterfall approach and some amount of bottom-up design (building small generic tools that seem useful). All this and the use of Lisp appear to have saved a lot of time and effort.

This is important, since the cost of software development can be enormous, and for commercial products the cost of an RTP system product is dominated by the cost of the software. Commercial RTP products that include software and a single Unix workstation are typically in the $200,000 to $300,000 price range. Informal conversations with other developers indicate that these products represent source code on the order of 300,000 to 500,000 lines of C code, and can involve as much as 50 person years of effort.

Prism was built with relatively little effort and costs very little to maintain and enhance. The initial release of Prism in July 1994 was about 45,000 lines of code with about 7 person years of effort, *including* the time it took to develop specifications, do testing, and write documentation. Since then maintenance and further development represent about another 4 person years. Prism today has grown only to about 52,000 lines of code but has many more features than the 1994 version. Of this, the SLIK user interface toolkit, the part that is completely generic, is about 9,000 lines of code. The SLIK programmer's manual is 3,000 lines of LaTeX source (60 pages when printed).

The documentation for Prism includes a functional specification, a programmer's guide to internals, the dose computation specification mentioned above, and a User Manual, totalling about 23,000 lines of LaTeX source in addition to the SLIK manual.

3.4 Lisp as an industrial strength programming language

Lisp has features and characteristics that make it radically different from all the more familiar programming languages. These features are important in writing RTP software; they are not just exotica that artificial intelligence researchers play with. In Prism the following features proved to be especially useful:

- Run-time types: Lisp can determine the type of a piece of data at run-time, and act accordingly. We took advantage of this to make the storage and retrieval of RTP plan data very simple.
- Multi-methods: In the Common Lisp Object System, methods for generic functions can be selected at run-time based on the type of any number of arguments. This is used in many places in Prism.
- Lexical closures: Lisp code can create new functions while the program is running. This accounts for much of the modularity and compactness of the Prism code.

What would really be nice is a world in which one could reprogram the semantics of the language with the same flexibility with which we can reprogram the syntax of Lisp. That way, during program development, debugging, and testing we could use all the safety features that Lisp includes (data typing, garbage collection, error checking). Once convinced that the code works correctly, we could tell the compiler (by changing a few local declarations) to assume the correctness of certain invariants and to generate code which thereby runs blazingly fast.

Production Lisp compilers partially achieve this goal. By declaring types of local variables, most compilers will dispense with type checking and will inline arithmetic operations. It would be nice if such declarations could enable the Lisp system to dispense with type checking and boxing of non-immediate data (such as floats) when passed-to/returned-from not only user-defined functions but also system primitives. If the user passes a fixnum to a function whose argument is declared to be a float, that user must expect to pay the price of a bus error or segmentation violation. But if the code is correct, that code might run ten times faster (ours did).

On the old Lisp Machines one could partition memory into areas, and thereby control where data were allocated. A common efficiency hack was to allocate temporary data in an area that could be reset periodically (when safe, reset the free pointer to the beginning of the area). The result was intantaneous garbage collection (faster even than generational). Of course, this technique sacrificed safety – programming errors easily could crash the system. But this is the situation C programmers are in *all the time*!

It would be nice to incorporate such programmable semantics into the standard. That would give Lisp programmers the best of both worlds – the safety, elegance, and purity of Lisp combined with the raw speed of assembler or C. Of course programmers would have to remain vig-

ilant, but it is always easier to be vigilant and to program correctly when there is a safety net available (by turning optimizations off). Users of most traditional languages have no safety net, whether they want it or not.

It is very unusual to write large programming projects in radiation oncology or radiology in programming languages other than the mainstream (FORTRAN, C, perhaps C++). The Lisp programming language in particular has had a reputation as a specialized tool for artificial intelligence research, unsuitable for serious medical computing. Modern Lisp compilers and systems are, however, highly developed, and experience with Prism shows that Lisp is a powerful and useable general purpose programming language. Prism's interactive performance is acceptable for clinical use, and for speed critical operations it is comparable to programs written in C. The Prism dose computation code achieves performance fully comparable with other more conventional languages. The use of advanced design concepts that saved us years of development effort did not adversely affect the performance of the resulting system. Lisp is no longer part of the exotica of the past, but a well-supported environment for building powerful radiation oncology software.

4 Future directions/work in progress

Many enhancements to Prism, and research projects using Prism, are under way. We describe here only a few, the implementation of a medical image server using TCP/IP, a network client interface to an Internet medical anatomy knowledge resource, and the development of a macro language for radiation therapy planning systems.

4.1 The Prism DICOM-3 medical image server

The application and presentation layer network protocol developed by the American College of Radiology (ACR) and NEMA for interchange of medical image data between imaging computer systems is called DICOM-3 [26]. It is an object oriented model for medical images and related entities as well as an encoding scheme and communications protocol that is designed to work in several wide area network environments, including TCP/IP. DICOM-3 is now supported by most manufacturers of computerized medical imaging equipment.

The Prism DICOM-3 server is a facility for receiving images and image sets from imaging devices such as CT (Computed Tomography) or MRI (Magnetic Resonance Imaging) scanners, and storing them in the image database of the Prism RTP system for subsequent use in treatment planning with Prism as described above. We are implementing it also in Common Lisp, to take advantage of the fact that the image objects and image sets that the server receives are already well modeled and supported in existing Prism code. We are confident that the performance and robustness will be adequate. Our experience and that of others with the Common Lisp Hypermedia Server (CL-HTTP) from the MIT AI lab [27] already has demonstrated the effectiveness of writing socket based TCP/IP network applications in Lisp.

The Prism DICOM-3 server design follows the standard model for a TCP/IP connection-oriented server [28]. The overall flow is shown in figure 13. The names in the figure correspond to the standard BSD socket library calls.

Other DICOM implementations have been made available including full source code and documentation, for example the CTN project at Mallincrodt Institute of Radiology [29]. These are very large programs and suites of programs and libraries written in `C`. They are difficult to integrate into other systems like Prism, and are not very modular. We decided it would be more efficient simply to write our own directly in Common Lisp.

The code in figure 14, together with an interface to the BSD socket library (using a Lisp vendor specific foreign function facility), is a sketch of an implementation of the server structure in figure 13. Since some of the socket function names are already defined in Common Lisp, we used slight variations in those cases.

Although this project is not completed, it appears that our server will be very small and modular. The interfacing of Lisp code to the BSD socket library was a simple application of the foreign function interface of the particular Common Lisp system we are using. Although this part of the code is not ANSI standard, it would be easy to adapt to a different vendor's foreign function interface. The core, the `DICOM-STATE-MACHINE` function, is vendor and system independent.

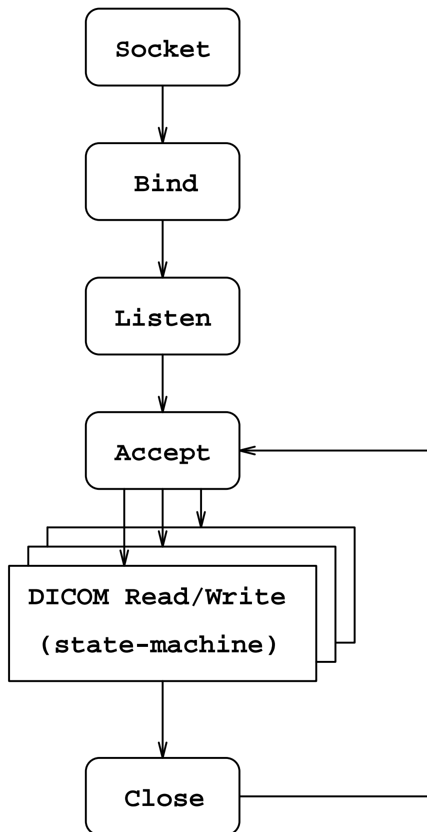

Figure 13: Model for a TCP/IP based DICOM-3 server

```
(defun DICOM-SERVER (port qlen)
  (let ((local-addr (make-sockaddr-in))
        (sock (SOCKET *pf-inet*
                      *sock-stream* 0))
        (remote (make-sockaddr-in))
        descriptor)
    (if (< sock 0)
        (error "Cannot create socket"))
    (setf ;; set up local socket params
       (sockaddr-in-addr local-addr)
              *inaddr-any*
       (sockaddr-in-port local-addr)
              (tcp-port-number port)
       (sockaddr-in-family local-addr)
              *af-inet*)
    (if (< (BIND sock local-addr
                 *sockaddr-in-len*) 0)
        (error "Bind failed for socket"))
    (if (< (tcp-LISTEN sock qlen) 0)
        (error "Listen failed for socket")
    (loop
      (setf descriptor
         (tcp-ACCEPT sock remote))
      (if (< descriptor 0)
          (error "Accept failed...")
        (progn
          (DICOM-STATE-MACHINE descriptor)
          (tcp-CLOSE descriptor))))))
```

Figure 14: A sketch for a DICOM server

4.2 The Digital Anatomist

This section describes work in progress on an on-line anatomy reference system for radiation treatment planning.

4.2.1 Use of anatomy knowledge in RTP

Delivering enough radiation to kill a tumor while minimizing damage to surrounding tissue requires knowing the precise locations of all anatomical structures nearby the tumor site, and although there are generally plenty of CT and MRI patient scans of the site availiable, certain soft structures often do not show up well using either imaging technique, forcing a dosimetrist to infer their locations from the structures that are visible and from his own knowledge of anatomy. Because of the difficulty involved with making such inferences with any degree of accuracy, the dosimetrist will often refer to an anatomy atlas book. Unfortunately, this book will have all of the usual drawbacks of printed media: static content with a fixed organization and presentation style. As an alternative, we are currently exploring how the use of an on-line anatomy reference tool could help to improve the process.

4.2.2 On-line anatomy information and knowledge resources

At the University of Washington, the Structural Informatics Group has an ongoing project known as the Digital Anatomist in which repositories of anatomical information are accessible to multiple clients via a distributed framework. Although all clients to date have been educationally oriented, the use of a distributed framework implicitly enforces a strict separation between content and presentation, meaning that how a client chooses to interact with its user is completely independent of that client's ability to retrieve information from the Digital Anatomist's databases. Add to this a rich semantic network of relationships between anatomical terms which allows the potential for intelligent searching, and the availability of a wide

variety of segmented anatomical data, and we have an excellent foundation on which an online anatomy clinical reference tool could be built.

Our ultimate goal, should the tool prove to be useful, is to integrate it with Prism, making it a natural choice to build it in Common Lisp using SLIK and CLX. Another incentive for using Lisp was that most Digital Anatomist components communicate with each other using a Lisp-like command syntax [30, page 478].

4.2.3 Implementation experience

We implemented a prototype clinical reference interface to the Digital Anatomist resources (the image repository and semantic network of anatomic terms). The prototype provides support for the kinds of queries we believe would be useful in radiation treatment planning. Radiation oncologists in the department examined the prototype and provided comments on the user interface, the usefulness of the functions, and the completeness of the database. A fuller report of this project from a clinical perspective is in preparation. Here we only comment on technical implementation issues.

While things went smoothly for the most part, one interesting problem did arise along the way: how to display the GIF images stored in the Digital Anatomist's image database when CLX provided support only for the XBM file format. Our first idea was to look for source on the web that could handle the decoding of GIF images, but the source for every decoder we found was tightly coupled with its respective program. Our next approach was to decode the images ourselves, but this proved to be very difficult and inefficient due to the GIF format's use of LZW compression, which required manual unpacking of bits since our file system does not directly support opening files using non-standard bit-length bytes. Our eventual solution was to use an image processing program called "ImageMagick" via Allegro's `run-shell-command` operating system extension to Lisp. The "ImageMagick" program converts the image into PPM format, which was trivial to then decode using Lisp.

Our initial implementation of the PPM decoder was correct but inefficient (a medium sized GIF required about ten seconds to load) and required refining. Our first refinement was to extract the raster data from the file in one lump sum using `read-sequence` instead of extracting one byte at a time using `read-byte`. This reduced the load time to about two seconds. Next we inlined references to the array in which the raster data had been deposited, and this reduced load time to about seven-tenths of a second, which was satisfactory for our purposes.

As with Prism, using Lisp helped us to write abstract, succinct, correct, readable code which could be tuned for efficiency when necessary. In addition, when we begin exploring intelligent search, Lisp will be the perfect tool, both for communicating with the Digital Anatomist and for the implementation and availability of search algorithms.

4.3 A macro language for RTP

In 1977, when most medical physicists were struggling to master simple graphic output devices like the Calcomp pen plotter, Ted Sterling, then a computer scientist at Simon Frazier University, was already looking into visionary ideas for radiation therapy planning software. Sterling proposed [31] that powerful systems for RTP could be built by first designing a macro language for RTP, with primitives for body parts, radiation machines and their functions. Prism is a step in this direction. A future enhancement we are exploring is to carry this out and provide a higher level macro language that the user may create convenience functions and other enhancements to Prism without having to modify any code. There is no better environment than Lisp for this kind of work.

5 Conclusion

Our experience points to some straightforward conclusions. We believe that the expressive power of Lisp gives Prism a lot of functionality with a small amount of source code. Using good software engineering practices helped considerably to reduce the amount of time to a deliverable product. Performance of Prism in an interactive environment is satisfactory for use in a busy clinic. If Prism can be used as an example, there is no reason Lisp cannot be a mainstream application development language.

6 Acknowledgements

This work was supported in part by National Institutes of Health grants R01 LM04174 and R01 LM06316 from the National Library of

Medicine. Its contents are solely the responsibility of the authors and do not necessarily represent the official views of the sponsoring agencies.

It is a pleasure to acknowledge contributions to the Prism system code from Jonathan Unger, Jon Jacky, Mark Phillips and Dat Nguyen in addition to work cited. It is also a pleasure to acknowledge contributions to the definition of the Prism system user interface and functional requirements from Sharon Hummel, Beth Miller, Paul Cho, Mary Austin-Seymour, and many other members of the University of Washington Radiation Oncology Department. Thanks go to Jim Brinkley, Cornelius Rosse and Jonn Wu, who contributed to the Digital Anatomist interface project. The excellent support of our primary Lisp vendor, Franz, Inc., made it possible for Prism to be a reliable and robust clinical software tool.

References

[1] Faiz M. Khan. *The Physics of Radiation Therapy*. Williams and Wilkins, Baltimore, MD, 1984.

[2] Ira J. Kalet, Gavin Young, Robert Giansiracusa, Jonathan Jacky, and Paul Cho. Prism dose computation methods, version 1.2. Technical Report 97-12-01, Radiation Oncology Department, University of Washington, Seattle, Washington, 1997.

[3] Michael Goitein and M. Abrams. Multi-dimensional treatment planning: I. delineation of anatomy. *International Journal of Radiation Oncology Biology and Physics*, 9:777–787, 1983.

[4] Michael Goitein, M. Abrams, D. Rowell, H. Pollari, and J. Wiles. Multi-dimensional treatment planning: II. beam's eye-view, back projection, and projection through CT sections. *International Journal of Radiation Oncology Biology and Physics*, 9:789–797, 1983.

[5] James F. Brinkley. A flexible, generic model for anatomic shape: Application to interactive two-dimensional medical image segmentation and matching. *Computers and Biomedical Research*, 26(2):121–142, 1993.

[6] Gregg Tracton, Edward Chaney, Julian Rosenman, and Stephen Pizer. MASK: combining 2D and 3D segmentation methods to enhance functionality. In *Mathematical Methods in Medical Imaging III: Proceedings of 1994 International Symposium on Optics, Imaging, and Instrumentation*, volume 2299. SPIE, July 1994.

[7] Photon Treatment Planning Collaborative Working Group. Three-dimensional display in planning radiation therapy: A clinical perspective. *International Journal of Radiation Oncology, Biology and Physics*, 21:79–89, 1991.

[8] Ira J. Kalet and Jonathan P. Jacky. A research-oriented treatment planning program system. *Computer Programs in Biomedicine*, 14:85–98, 1982.

[9] Jonathan Jacky and Ira Kalet. A general purpose data entry program. *Communications of the ACM*, 26(6):409–417, 1983.

[10] Jonathan Jacky and Ira Kalet. An object-oriented approach to a large scientific application. In Norman Meyrowitz, editor, *OOPSLA '86 Object Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 368–376. Association for Computing Machinery, 1986. (also *SIGPLAN Notices*, 21(11), Nov. 1986).

[11] Jonathan Jacky and Ira Kalet. An object-oriented programming discipline for standard Pascal. *Communications of the ACM*, 30(9):772–776, September 1987.

[12] Ira J. Kalet, Jonathan P. Jacky, Ruedi Risler, Solveig Rohlin, and Peter Wootton. Integration of radiotherapy planning systems and radiotherapy treatment equipment: 11 years experience. *International Journal of Radiation Oncology, Biology and Physics*, 38(1):213–221, 1997.

[13] Scott McKay and William York. Common Lisp Interface Manager, release 2.0 specification. Technical report, Symbolics, Inc. and International Lisp Associates, Inc., May 1992.

[14] Brad A. Myers, Dario A. Giuse, Roger B. Dannenberg, Brad Vander Zanden, David S. Kosbie, Edward Pervin, Andrew Mickish, and Philippe Marchal. Garnet: Comprehensive support for graphical, highly interactive user interfaces. *Computer*, 23(11):71–85, November 1990.

[15] Kevin Sullivan and David Notkin. Reconciling environment integration and component independence. In *Proceedings of SIGSOFT90: Fourth Symposium on Software Development Environments*, pages 22–33, Irvine, California, 1990.

[16] T. Berlage. *OSF/Motif: Concepts and Programming*. Addison-Wesley, Wokingham, United Kingdom, 1991.

[17] Kevin Sullivan and David Notkin. Reconciling environment integration and software evolution. *ACM Transactions on Software Engineering and Methods*, 1(3):229–268, July 1992.

[18] Guy Steele, Jr. *COMMON LISP, the Language*. Digital Press, Burlington, Massachusetts, second edition, 1990.

[19] Robert L. Siddon. Fast calculation of the exact radiological path for a three-dimensional ct array. *Medical Physics*, 12(2):252–255, 1985.

[20] International Commission on Radiation Units and Measurements. *Prescribing, Recording and Reporting Photon Beam Therapy*. International Commission on Radiation Units and Measurements, Bethesda, MD, 1993. Report 50.

[21] Mary Austin-Seymour, Ira Kalet, John McDonald, Sharon Kromhout-Schiro, Jon Jacky, Sharon Hummel, and Jonathan Unger. Three-dimensional planning target volumes: A model and a software tool. *International Journal of Radiation Oncology Biology Physics*, 33(5):1073–1080, 1995.

[22] Case H. Ketting, Mary M. Austin-Seymour, Ira J. Kalet, Jonathan P. Jacky, Sharon E. Kromhout-Schiro, Sharon M. Hummel, Jonathan M. Unger, and Lawrence M. Fagan. Evaluation of an expert system producing geometric solids as output. In Reed M. Gardner, editor, *Proceedings of the Nineteenth Annual Symposium on Computer Applications in Medical Care*, pages 683–687, Philadelphia, PA, 1995. Hanley and Belfus, Inc.

[23] Christine Sweeney. RULER user manual. Technical Report 91-10-01, Radiation Oncology Department, University of Washington, Seattle, Washington, 1991. A rule interpreter based on discrimination nets.

[24] Ira J. Kalet and Witold Paluszyński. Knowledge-based computer systems for radiotherapy planning. *American Journal of Clinical Oncology*, 13(4):344–351, 1990.

[25] James F. Brinkley, Kraig Eno, and John W. Sundsten. Knowledge-based client-server approach to structural information retrieval: the digital anatomist browser. *Computer Methods and Programs in Biomedicine*, 40(2):131–145, 1993.

[26] W. Dean Bidgood, Jr., Steven C. Horii, Fred W. Prior, and Donald E. Van Syckle. Understanding and using DICOM, the data interchange standard for biomedical imaging. *Journal of the American Medical Informatics Association*, 4(3):199–212, 1997.

[27] John C. Mallery. A Common LISP hypermedia server. In *Proceedings of The First International Conference on The World-Wide Web*, 1994. conference held at CERN, Geneva, Switzerland.

[28] Douglas E. Comer and David L. Stevens. *Internetworking with TCP/IP: Client-Server Programming and Applications*, volume III. Prentice-Hall, Englewood Cliffs, NJ, 1993.

[29] S. M. Moore, D. E. Beecher, and S. A. Hoffman. Dicom shareware: a public implementation of the dicom standard. In R. Gilbert Jost, editor, *Proceedings of the SPIE Conference on Medical Imaging 1994 – PACS: Design and Evaluation*, volume 2165, pages 772–781, Bellingham, Washington, 1994. Society of Photo-optical Instrumentation Engineers (SPIE).

[30] James F. Brinkley, Scott W. Bradley, John W. Sundsten, and Cornelius Rosse. The digital anatomist information system and its use in the generation and delivery of web-based anatomy atlases. *Computers and Biomedical Research*, 30:472–503, 1997.

[31] Theodor D. Sterling. Natural language compilers and interpreters in radiation treatment planning or nous parlons anglais better than jede andere sprache. In Ulf Rosenow, editor, *Proceedings of the Sixth International Conference on the Use of Computers in Radiation Therapy*, pages 8–24, Göttingen, 1977.