Portable Software Tools for 3-D Radiation Therapy Planning

Jonathan Jacky, Ph.D.[1] Ira Kalet, Ph.D.[1] Jun Chen, M.S. [2]

James Coggins, Ph.D.[2] Steve Cousins, M.S. [3]

Robert Drzymala, Ph.D.[3] William Harms, B.S.[3]

Michael Kahn, M.D., Ph.D.[3] Sharon Kromhout-Schiro, Ph.D.[1]

George Sherouse, Ph.D.[2] Gregg Tracton, B.S.E.[2]

Jonathan Unger, M.S.[1] Martin Weinhous, Ph.D.[3]

Di Yan, Ph.D.[3]

[1]University of Washington, Seattle, Washington. NCI Contract number N01-CM97566

[2]University of North Carolina, Chapel Hill, North Carolina. NCI Contract number N01-CM97565

[3]Washington University, St. Louis, Missouri. NCI Contract number N01-CM97564

[0](Degrees and affiliations at the time when the work was done.)

[0]Reprint requests to: Jonathan Jacky, Department of Radiation Oncology RC-08, University of Washington, Seattle WA 98195. Phone: 206-548-4117. Fax: 206-548-6218. email: `jon@radonc.washington.edu`

Abstract

Purpose: Produce a collection of software tools (computer programs) that support three-dimensional (3D) radiation therapy planning. The tools are not a complete 3D planning sytem. Instead, they work with any 3D planning system that meets certain minimal specifications. The tools assist in deriving anatomic data from images, generating target volume contours, evaluating treatment plans and verifying accurate treatment delivery. The tools are *portable*: they can run without source code changes in any computing environment that provides a library of functions and data definitions called the Foundation. The Foundation couples the portable tools to the (usually non-portable) file system and dose calculation associated with a particular 3D planning system.

Methods and Materials: Tools were written at three different (geographically separated) institutions. Software developers from all three sites specified the Foundation. The programmers' interface to the Foundation is portable, but a Foundation implementation need not be portable. Each group implemented a Foundation adapted to the (different) 3D planning system used at their site.

Results: All tools run at all three sites without source code changes. Each Foundation was implemented in a few person-months of programming effort. The program text and documentation for the tools have been placed in the public domain.

Conclusions: It is practical and economical to produce portable radiotherapy treatment planning tools. Providers of 3D planning programs should offer Foundations for their systems, so they can be used with tools. Researchers considering new computer programs should write them as tools, so they can work with any 3D planning system.

Key Words: Three-dimensional treatment planning, Software engineering, Software tools, Portable software, Collaborative working group.

1 Introduction

Planning radiotherapy treatment usually involves the use of large, complex computer program systems called radiation therapy planning (RTP) systems. 3D RTP systems have been produced at several research centers and by commercial vendors. Developing a 3D RTP system is a major undertaking. Software development costs dominate 3D RTP system costs and form a major impediment to the widespread adoption of 3D treatment planning.

The time and cost to create these systems motivates interest in *software exchange*, in which software developed at one institution (research center or commercial vendor) could be used at another with little additional effort or cost, rather than each institution replicating the system building effort of every other one.

In the past, such software exchange for RTP systems has usually required that the recipients must expend several months of effort on modification of the software from the donors, and on their own software as well. Sometimes, software is so dependent on the originating institution's entire RTP system, that it simply cannot be adapted for use elsewhere.

Producing shareable software was an important goal of the Radiotherapy Treatment Planning Tools project, a five-year collaborative effort of the National Cancer Institute, University of Washington, Seattle and Washington University, St. Louis [10]. We developed a powerful, practical, cost-effective method for creating shareable software and for adapting RTP systems to be able to use the software. The remainder of this paper describes the details of our method, and reports our experience applying it.

The documentation and source program text for the software that we exchanged is available for incorporation into any RTP system whose designers are prepared to make the effort described here.

2 Tools

The difficulties of exchange arise from the very substantial differences between RTP systems. We do not believe it is desirable or necessary to mandate the universal use of a single standard planning system. There are many valid approaches to how to build a system and how it should work from the user's point of view. Even in a community that supports a variety of quite different planning systems, the effort of exchanging RTP software can be greatly reduced. This is made possible by organizing RTP systems around independent components that we call *tools*.

The central idea is to distinguish between the core of essential RTP functions — data storage, dose calculations, and a few basic 3D displays — and other optional functions that can be separated out. Instead of making the optional functions inextricably bound up with the core system, we build them as separable, largely independent programs, or *tools*.

Each tool performs just one or a few special tasks that the basic RTP system does not already have, or they may perform certain tasks faster or better than the corresponding components in the basic RTP software.

At this writing, tools that the working group has completed (or nearly completed) perform the following tasks:

1. Semi-automatically segment a computed tomography (CT) image data set, gen-

erating data identifying normal tissues (organs), much reducing the need for time-consuming hand drawing on CT images [17].

2. Calculate a digitally-reconstructed radiograph (DRR), resembling a (projected) simulator film image, from volume CT data [2].

3. Generate *tiles* (data structures used to produce 3D surface displays), from series of planar contours or 3D dose distributions [18].

4. Compute and display dose-volume histograms, tumor control probabilities (TCP), normal tissue complication probabilities (NCTP), and other dose statistics from dose distributions computed by the RTP system (not the tool's) dose computation code [4].

5. Rank treatment plans, incorporating treatment preferences of radiation oncologists, applying decision-analytic and heuristic techniques [8, 9].

6. Assist in verifying treatment delivery by comparing prescription images such as a simulator film with multiple portal images such as port films [1].

7. Automatically generate a planning target volume from a gross tumor volume (delineated perhaps by hand drawing on CT images) [12, 13].

We suggest that the best way to obtain these functions is not to expend programming effort modifying one's own core system, but to acquire and install these tools.

The various tools are described in greater detail elsewhere. The rest of this paper concerns the development method common to all tools.

3 Design model

For a tool to be useful to others, it must integrate easily with whatever RTP system happens to be in use at each recipient site. How can a tool developed to work with one particular core RTP system be quickly adapted to work with all the others? This is the technical problem that we solved.

Our method is based on a *design model* (introduced in reference [10] and elaborated in reference [7]). According to our design model, a complete RTP system comprises several components. The *Virtual Machine Platform* (VMP) includes the computer and display hardware itself and its system software, including its operating system and programming languages. The *Local Radiotherapy Planning Program* (RTP) is software that provides the basic treatment planning dose calculations, displays, and data storage. The *Tools* are separate programs that perform optional therapy planning functions. The last component is software that adapts all of the tools to the local planning program. We call this adapter the *Foundation.*

At any one installation, the VMP may be any sufficiently capable computer system. The RTP program may be one of the commerical products or a locally developed research system. Tools may be produced locally or obtained from some other site.

An essential feature of the design model is that every tool is *portable.* This means that *the program text for each tool is exactly the same at every installation*, regardless of the kind of computer system or local RTP program used there. Any tool may be used at any installation *without requiring any changes to the tool's program text.* This is made possible by providing a suitable adapter, which must be customized to the local RTP program.

Our solution requires some investment by the developers of each core RTP system for which one would like to incorporate software tools, but *this needs to be done only once for each type of RTP system.*

This represents a significant saving of effort over the way things have been done in the past, where the recipient of a potentially useful program usually needs to modify the program, and the local RTP system as well, every time they wish to incorporate a new tool.

4 Computer system requirements

The tools can run on any sufficiently capable computer system, which we call a *Virtual Machine Platform* (VMP). Our definition of a "sufficiently capable" VMP appears in reference [10]. Almost any modern high-performance graphic workstation meets the VMP hardware requirements. It appears that some of today's more powerful personal computers meet our VMP specification, although we have not attempted to run any tools on them.

A VMP must provide the X Window system [14], standard software that provides a *graphical user interface* (GUI) featuring overlapping display windows controlled by a pointing device called a "mouse." The X Window system is supported by every major workstation vendor and is usually included ("bundled") in purchases.

The collaborative working group does not limit itself to a single programming language, so a VMP must support several. At this writing, tools have been written in ANSI C [11], Common Lisp [15], and C++ [16].

The VMP specification does not require any particular operating system because this would have limited distribution of the tools to sites that used it. In fact, the VMP specification prohibits writing tools that depend on any particular operating system. Standard programming languages and the X Window library provide access to all necessary functions and are supported by many different operating systems.[1]

5 Object model

Software exchange has been difficult because different RTP systems represent radiation treatments very differently. The most serious differences concern radiotherapy concepts and their representation, not the trivial details of programming languages or file formats. The various RTP systems differ in such basic matters as what items of information describe a radiation beam or treatment field. For example, some systems locate the beam with respect to the patient by specifying the isocenter coordinates, while others specify the patient support apparatus (couch) settings.

To overcome these differences we had to reach agreement about such radiotherapy matters as what constitutes an organ, a radiation beam, a dose distribution. Together, these agreements constitute our *object model*. Our object model is described in a 60 page technical report [7]. (In some of our technical reports, the object model is called the *Foundation Library Specification*.)

We defined our object model as rigorously as possible in English, using a particular format and a special vocabulary of terms which we defined carefully. Figure 1 shows a sample page from the report that describes an object we call *polyline*, which is simply a curve that lies in some plane. More complex objects such as organs and tumors are built up systematically from this and other simple objects.

[1]Our tools have run under several vendors' variants of the Unix operating system, as well as Digital Equipment Corporation's VMS operating system

The object model is essentially a standard vocabulary of radiotherapy concepts expressed in terms that are meaningful at a level that is closer to physical reality than computer language terms. Since the model is defined at this level in English, not a programming language, it is not restricted to be implemented in any one particular programming language. This is essential to our effort, because RTP programs and tools may all be written in different programming languages.

Having defined the objects that are found in radiation therapy, we also defined (again in English) three operations that act on these objects. The *Fetch* operation retrieves some collection of objects — typically the input needed by some tool — from the local RTP program's storage (usually, its disk file system). The *Store* operation saves some collection of objects — typically the output produced by some tool — in the local RTP program's storage. The definition of Fetch and Store are sufficiently general that each operation can handle any collection of objects that might be needed. The *Compute* operation invokes the local RTP program's dose computation, providing it with a collection of objects that define the patient anatomy and a radiation beam, and obtaining from it a 3D dose distribution.

The three operations Fetch, Store and Compute are all that are needed to connect any tool to any RTP program. It is important to understand that a single object model applies to all RTP programs and all computer systems. The work of creating and documenting the object model [7] need not be repeated.

6 Programming language bindings

Once we had agreed on the ingredients of RTP systems at an abstract level, we could provide specific realizations of these agreements in the terms of each programming language that we use. We call the representation of the object model in a particular programming language a *binding.* There is just one object model, but there are potentially many language bindings — one for each programming language that tool developers might wish to use.

We produced two language bindings. The binding for ANSI C is described in a 35-page report [6]; this binding serves C++ as well. The binding for Common Lisp is described in a 12-page report [5]; it is shorter than the C binding because Lisp provides a higher level of abstraction and is, therefore, a closer match to the object model.

Figure 2 shows a sample from the C binding that represents the same object defined in English in Figure 1. Figure 3 shows the corresponding sample from the Lisp binding. Every object and the three operations defined in the object model [7] are defined in this way. *Objects* and *operations* in our model are represented by *data structures* and *functions* in the programming language, respectively.

The languages ANSI C and Common Lisp are standardized languages and are available on most computer systems. Therefore, our language bindings apply to all of these computer systems and every RTP program that runs on any of them. *It is not necessary that the RTP program itself be written in one of these languages.*

The work of creating and documenting the two language bindings [5, 6] need not be repeated. However, creating a binding for another language (e.g., FORTRAN) would require additional work (which would then be useable for any tool written in FORTRAN).

7 Writing portable tools

With language bindings available, it is possible to write portable treatment planning tools. A tool is portable if its program text (or "code") uses only the standard features of the programming languages, libraries provided by the VMP (for example, the X Window library), and our language bindings.

From the programmer's point of view, our language bindings play much the same role as any other library of data structures and routines that might be provided by a computer vendor or a software company (e.g. the X Window library itself).

Our design model ensures that programmers can write tools without any knowledge of the internal details of any RTP program. To read or write data into any RTP program's storage, the programmer need only invoke our Fetch or Store functions. The tool programmer need not know anything about the internal organization of the RTP program's file system, and we did not need to define any (putatively) standard file formats. Likewise, a tool programmer can use the RTP program's dose computation by invoking our Compute function, without knowing any details of the RTP program's dose calculation algorithm, or even knowing what dosimetric data it requires.

8 Writing the adapters

To make it possible for a particular kind of RTP program to use the tools, it is necessary to write some additional software that couples or adapts that RTP program's storage and dose computation to the language bindings we defined. We call this adapter the *Foundation* (in some of our technical reports, this adapter is called the

Foundation Implementation to distinguish it from the *Foundation Library Specification.*)

Writing this adapter software is the essential task that must be done to make the work of all tool developers everywhere accessible to users of a particular kind of RTP program (e.g., some particular research system or commercial product). The adapter need only be written once, and the same adapter will work for all copies of a particular research system or product, and with all present and future tools.

To write an adapter, a programmer must be familiar with the internal organization of the RTP program, as well as our object model and language binding. The adapter code must deal with all the low-level internal details of the RTP program that our design model hides from tool programmers.

A different adapter is required for each language binding. It is only necessary to write an adapter for the languages used in the tools one wishes to acquire. At this writing, there is only one tool that requires the Lisp binding; all the others use the C binding.

The adapter code itself need not be portable. It need not be written in the same language as the RTP program. In fact, most of an adapter can be written in a different programming language than the binding it supports. At two of our three sites, the adapter for the Lisp binding is largely written in C.

The effort of producing the adapters is of interest because it is the entry cost of using the tools. At each of our three sites, we produced adapters with a few person-months of effort; they comprise only a few thousand lines of code (Table 1). This is only a modest programming effort. (The adapter for each language can be undertaken as an independent project, so C and Lisp are separated in the table.)

9 Installation experience

Each of the three working group sites has installed and run tools developed at the other two sites (as well as its own tools).

To make this possible, we had to agree on numerous issues, for example: What materials (computer files, documents) must be provided by developers? What degree of completion is deemed necessary before a tool may be distributed? Who is responsible for fixing any errors that might be discovered?

We recorded our decisions in a 35-page manual [3] that documents a *method* that we followed in a repeatable way (for each tool) and which could be adopted later by other institutions outside the original working group.

A key idea in our manual holds that a completed tool is not just files of programming language text (the *source code*). It also includes all the supporting documents needed to install, use, and maintain the program, and to assure that it works. The material distributed with each tool includes a user's manual, design documents, certain supporting data, test data and instructions for running the tests.

Authors may release a tool for distribution to other sites when the program and all the associated documents required in reference [3] are complete, and the program has passed its acceptance test at the originating site.

Other sites obtain the tool by acquiring the collection of files that constitute its distribution kit (source code, test data, installation instructions, user's guide, etc.). Files were distributed over the Internet computer network, using the *FTP* (file transfer program) network utility. We have never resorted to distributing computer files on physical media (tapes or disks). In a few cases, it has been necessary to supplement the electronic distribution by sending printed documents in hardcopy form (where these included figures and formatting that could not be printed easily at all sites).

We do not distribute ready-to-run software. Even after a site has completed its adapter, some additional work is required to install each tool. Each program is distributed as several (or many) files of human-readable program text. Staff at the recipient site must use the utilities of their own computer system to *compile* these files into executable form and *link* them with each other and with the local adapter. However, *they do not make any changes to the files obtained from the tool's authors.* When they have built an executable program, the recipients run the acceptance tests provided by the tool's authors, making sure that the promised test results are produced.

No installation has been trouble-free. Each installation has required a part-time effort extending over days or weeks. Typically, recipients find at first that compilation fails. This happens because computer systems at the three sites are very diverse, and programmers inadvertantly write non-portable code that makes unwarranted assumptions about the computing environment at the other sites. After these are corrected and an executable program is produced, a tool typically fails to pass all of its acceptance tests. This reveals programming errors in the tools and/or the adapters, that usually result from ambiguity in or misinterpretation of our various specifications [5, 6, 7].

A key requirement of this process is that *all corrections to the tool must be made by the originating site.* Recipients only report errors to the tool's original authors. It is sometimes difficult for recipients to resist the temptation to fix seemingly trivial problems themselves. However, doing so would defeat the purpose of the project, since it would result in different (incompatible) versions of every tool, each customized to the incidental peculiarities of the recipient site.

We have always been able to correct every problem (eventually) and produce a single version of each tool that can be successfully built and run at every site. The effort seems large but is not unlike what commercial vendors experience when they first release a new product for "beta-testing" outside the original group of developers. We expect that the process will become easier as we gain experience.

There is also the problem of integrating the tools *seamlessly* into the local system. This is quite distinct from, and harder than, just getting them to run, involving issues of convenience and aesthetics. Work on this problem continues at each site.

10 Discussion

Does the tools approach work? Can a research center or commercial vendor benefit by acquiring our tools, instead of writing their own?

Our collaborative working group expended considerable effort in developing this approach. By far the larger portion of that effort was devoted to reaching consensus on goals and procedures and then creating and documenting the design model, the computer system requirements, the object model, and the programming language bindings. The products of this effort apply to any RTP system and are now available in our reports [3, 5, 6, 7, 10]. Therefore, this effort need not be repeated. Compared to the large initial design effort, relatively little additional effort is needed to produce the adapter that couples all tools to a particular RTP system. Only this lesser portion of the work needs to be repeated in order to adapt our tools to additional RTP systems.

The data presented in Table 1 shows that an individual or small team of programmers working from the design information provided in our reports can produce an adapter in a few months, or less. This is considerably smaller than the year or more usually needed to produce a single tool. This suggests that producing an adapter and acquiring portable tools is an economical alternative to modifying the local RTP program to duplicate tool functions.

We do not distribute ready-to-run software. To use our work, an institution must be prepared to devote several person-months to build an adapter and some additional time to install and integrate each tool. These tasks require program development tools such as compilers and linkers and a thorough understanding of the internals of the local RTP program. Vendors of commercial RTP products are in the best position to do this work for their own systems, although customers or third parties could also do the work if vendors make the necessary information and permissions available to them.

To take best advantage of our work requires a quite different approach to RTP system development than has been customary in the past. The two biggest differences are conceiving RTP systems as separable tools rather than monolithic systems, and a willingness to understand and comply with specifications written by others. Programmers who wish to incorporate our work must master the contents of our reports, about 150 pages technical material. Important distinctions, such as tool versus core RTP program, and portable tool code versus site-dependent adapter code, are quite difficult to infer from the tool code itself.

Our documents and program text (code) are available, and some recipients may choose to bypass our method and simply extract interesting fragments to incorporate into their own systems, in the traditional ad-hoc way. We do not recommend this. Although the amount of detail and strictness of our method may seem burdensome, it achieves overall a large saving in effort and makes the software developments of one institution rapidly available for use by others. The greatest advantages will accrue if a large community, including commercial vendors, adopts this approach. If this occurs, innovations from each center can be made available to the entire community.

References

[1] W. R. Bosch, D. A. Low, R. L. Gerber, J. M. Michalski, M. V. Graham, C. A. Perez, W. B. Harms, and J. A. Purdy. An electronic viewbox tool for radiation therapy treatment verification. *International Journal of Radiation Oncology Biology and Physics*, 27(Supplement 1):179, 1993. (Abstract).

[2] E. L. Chaney, J. S. Thorn, G. Tracton, T. Cullip, J. G. Rosenman, and J. E. Tepper. A portable software tool for computing digitally reconstructed radiographs. *International Journal of Radiation Oncology Biology and Physics*, 27(Supplement 1):180 – 181, 1993. (Abstract).

[3] James Coggins, George Sherouse, Sharon Hummel, Jonathan Jacky, Robert Drzymala, and Martin Weinhous. *Standards and Practices of the Collaborative Working Group on Radiotherapy Treatment Planning Tools.* Technical Report 90-2, National Cancer Institute, Radiation Research Program, 6120 Executive Boulevard, Executive Plaza North, Rockville, MD 20852, 1990. Report of the Task Group on Documentation of the Collaborative Working Group on Radiotherapy Treatment Planning Tools.

[4] R. E. Drzymala, M. Holman, Di Yan, W. B. Harms, , N. Jain, M. Kahn, B. Emami, and J. A. Purdy. Integrated software tools for the evaluation of radiotherapy. *International Journal of Radiation Oncology Biology and Physics*, 24(Supplement 1):157, 1992. (Abstract).

[5] Jonathan Jacky, Ira Kalet, Michael Kahn, and Steve Cousins. *Common Lisp Language Bindings to the Foundation and Virtual Machine Platform (VMP).* Technical Report 92–2, National Cancer Institute, Radiation Research Program, 6120 Executive Boulevard, Executive Plaza North, Rockville MD 20852, 1992.

Report of the Radiotherapy Treatment Planning Tools Implementation Task Group.

[6] Jonathan Jacky, Gregg Tracton, Di Yan, and William Harms. *ANSI C Language Bindings to the Foundation and Virtual Machine Platform (VMP).* Technical Report 92–1, National Cancer Institute, Radiation Research Program, 6120 Executive Boulevard, Executive Plaza North, Rockville MD 20852, 1992. Report of the Radiotherapy Treatment Planning Tools Implementation Task Group.

[7] Jonathan Jacky, Martin Weinhous, James Coggins, Robert Drzymala, William Harms, Sharon Kromhout-Schiro, George Sherouse, Gregg Tracton, and Jonathan Unger. *Foundation Library Specification and Virtual Machine Platform (VMP) Specification.* Technical Report 91–1, National Cancer Institute, Radiation Research Program, 6120 Executive Boulevard, Executive Plaza North, Rockville MD 20852, 1991. Report of the Radiotherapy Treatment Planning Tools Specification Task Group.

[8] N. L. Jain and M. G. Kahn. Ranking radiotherapy treatment plans using decision-analytic and heuristic techniques. *Computers in Biomedical Research*, 25:374 – 383, 1992.

[9] N. L. Jain, M. G. Kahn, R. E. Drzymala, B. Emami, and J. A. Purdy. Objective evaluation of 3-D radiation treatment plans: a decision-analytic tool incorporating treatment preferences of radiation oncologists. *International Journal of Radiation Oncology Biology and Physics*, 26:321 – 333, 1993.

[10] Ira J. Kalet, Edward Chaney, James Purdy, and Sandra Zink. *Radiotherapy Treatment Planning Tools First Year Progress Report.* Technical Report 90–1, National Cancer Institute, Radiation Research Program, 6120 Executive Boulevard, Executive Plaza North, Rockville MD 20852, 1990.

[11] Brian W. Kernighan and D. M. Ritchie. *The C Programming Language.* Prentice-Hall, second edition, 1988.

[12] Sharon Kromhout-Schiro. *Development and Evaluation of a Model of Radiotherapy Planning Target Volumes.* PhD thesis, University of Washington, 1993.

[13] Sharon Kromhout-Schiro, Mary Austin-Seymour, and Ira Kalet. A computer model for derivation of the planning target volume from the gross or clinical tumor volume. In A. R. Hounsell, J. M. Wilkinson, and P. C. Williams, editors, *Proceedings of the Eleventh International Conference on Computers in Radiotherapy*, Christie Hospital, Manchester, UK, 1994.

[14] R. W. Scheifler and J. Gettys. The X window system. *ACM Transactions on Graphics*, 5(2):79–109, 1986.

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

[16] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley, Reading, Massachusetts, 1986.

[17] 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*, SPIE, 1994. (in press).

[18] Gregg Tracton, Jun Chen, and Edward Chaney. CTI: automatic construction of complex 3D surfaces from contours using the Delaunay triangulation. In *Mathematical Methods in Medical Imaging III: Proceedings of 1994 International Symposium on Optics, Imaging, and Instrumentation*, SPIE, 1994. (in press).

	UW		*UNC*		*WU*	
	C	*Lisp*	*C*	*Lisp*	*C*	*Lisp*
Size, lines	4000	250	9500	2000	5000	2000
Effort, person-months	3	< 1	2	1	2.5	1

UW: University of Washington, Seattle
UNC: University of North Carolina, Chapel Hill
WU: Washington University, St. Louis

Table 1: Approximate size and effort in adapters ("Foundation")

List of Figures

Polyline

Role Represents an unconstrained curve in a plane, e.g. a clipped isodose contour or a physician's signature.

Also used to derive other classes, e.g. Contour

Ancestry

Attributes

vertices:	sequence of real x, y pairs
transform:	4×4 matrix of reals

Conventions *Transform*, when applied to any vertex, yields the 3D coordinates of the vertex in a reference coordinate system. This reference coordinate system is determined by context (e.g., the object of which the curve is part).

Usually the curve is constructed by connecting consecutive elements or *vertices* with straight line segments but something fancier (e.g. splines) is also permissible.

The z coordinate of each vertex is implicitly 0 in the untransformed coordinate system.

Constraints *Vertices* must include two or more x, y pairs.

Transform is a rigid transformation that preserves the shape of the curve and the length of each segment.

Comments *Transform* allows a curve defined only with x, y pairs to lie in any plane in the enclosing reference coordinate system (usually the patient coordinate system): transverse, coronal, sagittal, oblique ...

Figure 1: Sample object description

All classes in the Foundation [7] are represented by C structs. . . . The first member of each of these typedefs must be named `obj_id` and be of type `rtpt_id_t`. The value of this member must one of the constants named in section 6.1.5.
In transformation matrices, the first dimension is the column and the second dimension is the row. For example, the t_z element in a transformation matrix `trans` (as defined in equation 2.1 in [7]) is `trans[2][3]`. . . .
Contours are declared employing `rtpt_vertex_t` and the usual C idiom for variable-length arrays: a member in the structure is a pointer to the first element of the array. The number of elements is stored in another member:

```
typedef struct rtpt_vertex
{
   small_float  x;
   small_float  y;
} rtpt_vertex_t;

typedef struct rtpt_transform
{
   large_float transform[4][4]; /* transformation matrix */
}  rtpt_transform_t;            /* [column:0..3][row:0..3] */

/* Polyline */
typedef struct rtpt_polyline
{
    rtpt_id_t         obj_id;    /* has value RTPT_POLYLINE */
    rtpt_vertex_t    *vertices_p;/* vertices -- array of rtpt_vertex_t */
    medium_uint       nvertices; /* n of elements in vertices */
    rtpt_transform_t  trans;     /* transform */
} rtpt_polyline_t;
```

Figure 2: Sample ANSI C language binding

For each class in [7], the RTPT Support must provide a `defclass` form. Tool authors must use `make-instance` to create an instance of a Foundation class, as in

```
        (make-instance 'rtpt:polyline ... )
```

For each attribute in each class, the RTPT Support must provide an accessor function. The name of the accessor function is the attribute name given in [7].
A tool author uses this function to read the value of of an attribute, as in:

```
        (rtpt:vertices  ... )
```

To write a new value for an attribute, the programmer uses `setf` with the same accessor function:

```
        (setf (rtpt:vertices ... ) ... )
```

This document does not specify the actual CL code in the class definitions; that is not necessary. . . .
Matrices in [7] (4×4 transformation matrices, dose matrices, images) are represented by CL arrays. Transformation are arrays of floats. . . .
For transformation matrices, the first dimension is the column and the second dimension is the row. This means that in `aref` forms, the index that occurs first is the column index and the index that occurs second is the row index. For example, the t_z element in a transformation matrix `transform` (as defined in equation 2.1 in [7]) is obtained by

```
        (aref transform 2 3)
```

The x, y pairs used to represent vertices of polylines in [7] are represented by 2-element lists, where each element is a CL `float`. The first element of the list is the x-coordinate and the second element is the y-coordinate.
The sequence of vertices is represented by a CL list of such pairs.

Figure 3: Sample Common Lisp language binding