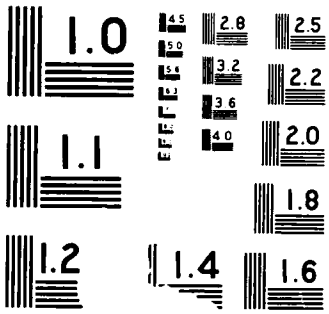
AD-A192 584 INTEGRATED CLASS STRUCTURES FOR IMAGE PATTERN RECOGNITION AND COMPUTER GRAPHICS(U) NORTH CAROLINA UNIV AT CHAPEL HILL DEPT OF COMPUTER SCIENCE 1/1

UNCLASSIFIED J M COGGINS NOV 87 N00014-86-K-0680 F/G 12/5 NL


1.0
1.1
1.2
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0

Proceedings of the First USENIX C++ Workshop, Santa Fe, NM, November 8-10, 1987, pages 240-245.

Integrated Class Structures for Image Pattern Recognition and Computer Graphics

James M. Coggins
Computer Science Department
University of North Carolina
Chapel Hill, NC 27599-3175

N00014-86-K-0680

Introduction

Research efforts in image pattern recognition and computer graphics face two kinds of software problems. First is the intrinsic complexity of the algorithms developed in the course of the research. Second is the design and construction of the researcher's software toolbox including fundamental operations, standards for data storage and communication, and interfaces to rapidly changing sets of display and interaction devices. The complexity problem is intrinsic to the subject matter and objectives, and it is the proper domain of the researcher. The second kind of problem is incidental to the research objectives and can be addressed by adoption of modern software development tools and disciplines, including object-oriented design for code and hypertext structures for documentation.

This paper describes several techniques we have developed while designing an integrated object-oriented software toolbox for image pattern recognition and interactive computer graphics research. Design criteria for the system include (1) pervasive integration of constructs, (2) maximum flexibility for researchers using the system, (3) minimum user effort to invoke the facilities, and (4) purity of the object-oriented design. We will describe in this paper techniques we have developed for managing massive data structures, providing type-independence at the user level, encapsulating device dependencies, processes, and class interfaces, and decomposing the required software system while maintaining integration of concepts.

Managing Massive Data Structures

The kinds of structures we manipulate (images, pattern matrices, graphical models) often have large or *very* large memory requirements. We do not want to reallocate, copy, and destroy these large structures in each function call and function value return. Instead, we implement large structures using a **header object** that stores descriptive data about the object along with a pointer to a **storage object** that contains the data. The header class `image`, for example, contains the image size and shape, its

disk file name, history, and other data, plus a pointer to an object of class `buffer` which contains the pixel data for the image. Class `image` understands messages that will be forwarded to the `buffer` object for processing. For example, `image image::operator+=(image&)` must be defined, but its action is simply to invoke the `buffer` method `void buffer::operator+=(buffer*)`. Objects of class `image` are small, so they may be copied, allocated, and destroyed as needed with negligible performance penalty. We must be careful, however, in the constructors and destructor of `image` to avoid repeated allocation, copying and deallocation of large `buffer` objects. This means that the `image::image(image&)` and `image::operator=(image&)` messages must *copy* the buffer pointer of the source image and not allocate a new buffer. In order to prevent the destructor `~image()` from deallocating the buffer while it is still being used, we place a reference count in the `buffer` object and delete the `buffer` only if the reference count is zero after decrementing. This mechanism also prevents the large `buffer` objects from being left in the heap as garbage, which would soon result in exhaustion of virtual memory. We are planning to use the header class/storage class implementation for all of our large structures such as pattern matrices and some graphical models.

Type independence

The separation of header and storage classes also makes possible type independence at the user level. Storage types for images include byte, color (4 bytes for RGBS), integer (short), real (float), and complex (a pair of floats). Objects of class `image` are manipulated as desired, independent of the storage type. Messages involving the pixel data are passed on to the image's `buffer` object. The `buffer` class contains virtual function declarations for the suite of operations affecting the pixel data. The real work is performed in subclasses of `buffer` that are type-specific: `byte_buffer`, `color_buffer`, `int_buffer`, `real_buffer`, and `complex_buffer`. Necessary coercions are provided between the buffer subclasses. When more than one coercion is possible (complex-to-real can be performed by real part, imaginary part, magnitude, or phase), a default is assigned and an optional parameter can be used to override the default.

Since the `image` objects are, in effect, annotated pointers to buffers, `image` objects can be manipulated naturally in expressions such as `result=im1+im2*2.0;` without the distractions of creating and dereferencing pointers.

The overhead of this extra level of indirectness is negligible because by assumption we are manipulating large objects. Since we get both type-independent manipulations and a reasonable method for managing the large memory requirements, we do not begrudge the overhead cost.

Encapsulating device dependencies

A serious problem in a high-technology lab such as ours is keeping the software base current and consistent with the available hardware capabilities. We have experienced a phenomenon we call "hardware indigestion" in which we have difficulty incorporating new hardware into existing projects because of incompatibilities between the various devices and the device-specific nature of the controlling software supplied by the vendors. Of these issues, the software incompatibility is the more serious problem. Advances in graphics and imaging devices have usually involved speed and resolution enhancements, not entirely new functionality. Accessing that functionality is difficult because the vendor's software involves intricate code that is optimized in some sense for the device's capabilities and that is not amenable to incorporation into a uniform interface.

We have worked on three kinds of device encapsulations. Disk file manipulations are supported by a class `diskfile` that handles basic disk operations such as open, close, read, write, and seek. These operations are then invoked by derived classes that "know" the structures of particular kinds of disk files such as `imagefile`, `polyfile`, and `patternfile`. New kinds of file structures can be added as derived classes of `diskfile` without changing any of the existing code.

Another kind of device encapsulation we have developed involves analog input devices such as knobs, joysticks, and sliders. The key to the design of these classes was recognizing that the only differences between them from the system's viewpoint are the name of the device handler and the number of bytes expected from the device in a single read operation. Operations provided by the abstract superclass include `poll` to force a read of the device and `int rawdata(int)` to obtain one of the values provided by the A/D converter. A uniform user interface is provided by adopting the convention that the device-specific classes convert the integer raw data value to a double between 0.0 and 1.0. Now the roles of the devices can be interchanged by simply declaring the device object to be of a different device subclass. Device-specific interfaces are also available; a 2-D joystick can return a point, and a 3-D velocity joystick can return a vector.

The third kind of device encapsulation involves display devices. We decided to make the unit of encapsulation be the viewing surface, so on window-oriented systems, the display object created is a window. Thus, several display objects may be active at once on a device. Several basic capabilities are defined in the abstract superclass including clearing the display, drawing lines, writing text, rendering polygons, and displaying

images. We are still debating how the enhanced capabilities of some devices may be made available within this framework, especially when the architecture of the device requires the data to be prepared differently for display processing.

Process encapsulations

Since our research involves development of new algorithms for imaging and graphics problems, encapsulation of these frequently-changing processes is essential to our research software environment. We have used a technique called *process encapsulation* to simplify the use and invocation of the processes based on the dictum "Encapsulate most deeply that which is most likely to change." Conceptually, a process encapsulation creates an object that we call an *enzyme* or a *catalytic object*, whose purpose is to mediate interactions among other objects. In a process encapsulation, a class structure is defined for the process type, a renderer or a classifier, for example, that specifies the minimum functionality and parameters of such a process. Then specific algorithms are defined as derived classes with their own parameters as required. To use the process, we create an object of the desired subclass, connect it to other objects and supply the parameters it needs, and send it a "begin" message. The input objects and parameter values can be supplied in three ways: by arguments to the constructor, by assignment in separate messages to the object, and by arguments to the "begin" message. This design allows the user to customize the process in a separate code segment from that where the process is invoked, leading to very clean code for the basic algorithm that invokes the process. The inheritance of fundamental operations and structures from the process' base class contributes to rapid development and evaluation of algorithm modifications and parameters.

Class Interface Encapsulations

The definition of standard process interfaces is facilitated by classes that store intermediate results in a standardized form (or a set of agreed-upon forms). Some details of this design innovation are still under development, but a typical example occurs in a graphics pipeline where various kinds of object models must be converted into "rendering primitives" that the display devices understand and can process. By adopting a standard set of rendering primitives, developers of process encapsulations for renderers and developers of display device encapsulations are insulated from each other's internal data structures and processing requirements. Development of renderers can proceed in a device-independent fashion and augmentations to the set of rendering primitives are explicitly noted and handled by all device encapsulations.
By standardizing interface classes, most of our graphics research efforts can begin to share code. We find that research that is advancing the state

of the art sometimes still must diverge from the standards either in order to optimize performance or in order to explore new paradigms that are beyond the state of the art. An example of the former case is real-time interactive graphics using customized parallel architectures requiring a different structure in the graphics pipeline. An example of the latter case is research in texture mapping in which the graphics pipeline is modified to accomodate an entirely new kind of rendering.

In order to accomodate these research efforts, our graphics class structures are designed for ease of use by *system developers* and has been criticized for being less than optimal for users. This is an explicit design tradeoff that we have accepted in order to provide flexibility and control at the expense of some ease of use and fidelity to user-level conceptual structures.

Separation of Concerns

The interface classes described above are used to implement a separation of concerns that has guided the design of our basic class structures. We will illustrate its effect with an example from our image processing library.

In our first implementation of class `image`, we included messages such as `load`, `save`, and `display`. The resulting structure had several problems. First, putting everything into `image` made the code too large. Second, the code for `image` had an unpleasing asymmetry. The code for the load, save, and display messages overwhelmed the code for the numerous image processing messages, most of which were less than ten lines each. Third, the intricate code we worked out for handling disk I/O was unusable by any other classes, and the display operation was useless on any but the device we defined it for.

The next incarnation of `image` separated the concerns of storage, processing, and interaction devices into different classes. Storage was handled by a class, `diskfile`, that encapsulates all low-level disk operations but without any knowledge of the semantics of the file being manipulated. A subclass `imagefile` directs the decoding and interpretation of the disk data. The `image` class retains the processing operations. Display of images was moved out to a `display_device` class with subclasses for the various devices available in our lab. Thus, an "image" became a "rendering primitive" that all display devices are expected to process in some reasonable manner.

The principle of separation of concerns is primarily an implementation principle that helps to provide the control and flexibility that we need in our research environment, but it sometimes works against the kind of

user-level ease-of-use and fidelity to conceptual structures that is a hallmark of Smalltalk. We are still investigating whether and how a user-level class structure might be imposed atop our implementation structures without redesign for each alternative implementation of a graphics or imaging pipeline.

Conclusion: Is C++ Really the Right Tool?

Object-Oriented Programming is a code packaging discipline that imposes a reasonable structure on large bodies of code, with additional benefits of code sharing within each class hierarchies and effective conceptual metaphors for talking and thinking about programs. Object-oriented programming provides just the kind of discipline and structure that we need as the size and complexity of our software base increases beyond a level where a single person can maintain, control, and understand it. Since we are an established UNIX environment, a C derivative makes sense in view of our large installed base of C code and our need for implementation control in order to support real-time operations and efficient handling of large storage structures. These properties of our environment and objectives make C++ a reasonable language for our software development efforts.

We eagerly await the development of a symbolic debugger for C++ and some relief to the problem of proliferating header files and the compile-time overhead they require. A precompilation of header files into an "environment file" similar to those available for DEC's VMS Pascal would be, for us, an ideal solution to the problem.

Our software development efforts are proceeding on two levels. Low-level encapsulations of basic data structures are being debated and sometimes shared among implementers. High-level architectures for large structures such as graphics and imaging pipelines are also being designed and debated. We believe that the largest benefit will be obtained from the high-level architectures and the standards decided at that level, but the low-level encapsulations are more immediately useful to implementers who already have their own versions of the major processes in the laboratory. Whether anticipated revisions to C++ such as parameterized types impact the effectiveness or generality of these design activities remains to be seen.

Acknowledgements

This research was supported in part by ONR contract N00014-86-K-0680. Students Muru Palaniappan, John Rohlf, and Brice Tebbs have made significant contributions to the design and implementation of the integrated class structures. About a dozen other faculty and student researchers in our department's Graphics and Imaging Cluster have provided valuable advice and criticism.

END

DATE

FILMED

6-88

DTIC