A prototyping environment for real-time graphics

by NOLA DONATO
Four Phase
Cupertino, California
and
ROBERT ROCCHETTI and JANET TOM
Mattel Electronics
Chicago, Illinois

ABSTRACT

As technology advances, graphics displays are becoming more powerful and less expensive, making interactive graphics increasingly popular as a method of man–machine communication. Often, nonprogrammers play a principal role in the design and implementation of applications involving graphics. Because interactive graphics require such a high level of feedback with both human and hardware, traditional programming languages are not well suited for the graphics environment.

This paper describes CGRASS, a portable, general-purpose programming language, and how it is used for prototyping videogames. The design rationale for a game-prototyping system is given, followed by an overview of the CGRASS language with emphasis placed on features particularly helpful for user interface design and modeling. We show examples of tools implemented for different hardware architectures and targeted for users of varying backgrounds.
INTRODUCTION

Programming a production videogame requires a lot of time and an experienced assembly language programmer. Hardware for both arcade and home videogames is very inexpensive compared to other types of graphics hardware. Correspondingly, resources such as CPU time and memory are quite limited, making the use of high-level languages seldom feasible. A significant part of game development involves modifying a program to fit in the small amount of memory allocated for the game. Time is also spent optimizing certain areas of code to make the game-play fast enough. A programmer frequently has to take advantage of quirks in the game hardware to produce graphic effects.

Usually, software tools to assist in the creation of a game run almost entirely on the game unit. They are programmed in assembly language (usually by game programmers) and are subject to the same speed and memory constraints as the game programs. As a result, tools are limited in functionality and do not tend to be very user-friendly; nonprogrammers often find them difficult to use. Typically, tools exist to create data structures, such as pictures and sounds, and to manipulate them in simple ways. However, to do anything more complex, one must resort to assembly language.

As the relative costs of human professional time and computer time shift, the interface between man and machine becomes increasingly important. Many researchers are investigating methods of improving user interfaces for a variety of applications. Powerful, user-friendly software tools are especially important in the videogame environment where nonprogrammers, such as artists, educators, game designers, and marketing experts make major contributions to the application.

One way to improve the game design process is to provide a way to make a working model of a game in a short amount of time. The decision about whether or not to manufacture a game can be made much earlier in the game's development cycle. Shortening the development loop allows more ideas to be tried, improving the quality of the game. Parts of the prototype (such as graphics, sound, and algorithms) can be applied to the final product. Finally, a good user interface can take input from novices as well as experts.

DESIGN APPROACH

We had several objectives for the design of our graphics-prototyping environment.

1. The system should be easy to learn and usable both by nonprogrammers and programmers
2. It should be interactive and provide immediate feedback
3. The system should have the ability to interface with vastly different hardware architectures
4. The user interface should be consistent across satellite and host systems wherever possible.

We divided the prototyping task into two parts. User interaction is delegated to a host processor, allowing applications to be written in a high-level language. Machine-specific functions are programmed on the game unit (or, more generally, the satellite processor). By using a reasonably powerful computer as our host, we are freed from many of the limitations imposed by the smaller game units. Therefore, game development tools can be more comprehensive, user-friendly, and so on. Satellite graphics systems have been used successfully in both commercial and research environments.

The type of host and satellite vary, as does the method of communication. For example, one configuration uses a large host, the VAX/11-780, connected to various types of satellites via serial ports. We have also been successful with a 16-bit microprocessor-based host linked through a parallel port to a graphics display.

From the host, one can invoke a variety of tools to create pictures and sounds, animate objects, and so forth. One can also write game prototypes and new applications directly. In addition, tools for translating from one target system format to another are available. For example, videocamera input from a bit-mapped display can be converted into a form usable with some character-mapped devices.

The Host System

As the basis of our system we chose CGRASS, an interpretive programming language written by the authors. The CGRASS language is implemented in C and has been successfully ported to many different machines and operating systems. This choice allows the environment for both program development and production applications to be consistent across all hosts.

In addition to being portable, the CGRASS language is very extensible. New commands may be implemented in either CGRASS or C and are easily added to the system. Programmers may create their own data types and define how the system operators and commands will interact with them. These capabilities make the language good for communicating with different satellite processors and for designing human interfaces.

Because it is interactive and dynamic, CGRASS is easy to learn and use. Control structures and data types are high-level and the system does much housekeeping automatically. Auto-
matic type conversion, dynamic allocation, succinct expressive constructs, and data abstraction mechanisms make CGRASS ideal for implementing user interfaces as well as for general programming tasks. The system has on-line helps and source level debugging facilities, which shorten program development time.

The Target Systems

Because videogame hardware is by nature idiosyncratic, we decided to implement most of the real-time graphics capabilities on the satellites. Each satellite processor has its own small, special-purpose, real-time executive that handles the coordination of graphic and audio events. This approach allows us to isolate the machine dependencies and standardize the user interface as much as possible. It also makes efficient use of the limited memory and speed resources.

LANGUAGE OVERVIEW

CGRASS borrows heavily from its predecessors GRASS3 (which ran on the PDP-11) and ZGRASS (a subset of GRASS3 for the Bally Arcade). Many of the ideas for graphics tools in the prototyping system are derived from work done with GRASS3. Similarly, ZGRASS provided a model for development of machine specific capabilities for low resolution bitmap displays.

CGRASS is a higher level language than GRASS3, with more powerful data abstraction capabilities. It has the run-time flexibility of languages such as SNOBOL4, while maintaining a structured nature similar to C. Data types in the language include variable-length strings and lists as well as traditional numbers and arrays. Like C, CGRASS is operator-rich and expression-oriented. The global, local, and static identifier scoping found in C is also present in CGRASS. In addition to the C-like control structures, CGRASS provides backtracking and goal-directed evaluation similar to ICON.

CGRASS has no storage declarations, explicit allocation, or deallocation. Variables may be assigned any value during execution. Storage management and type conversion are handled automatically by the system. Strings and lists may be arbitrarily long, limited only by physical resources. Like GRASS3, the number and type of arguments to procedures is determined at run time.

Execution Environment

CGRASS is a conversational system; any statement that may be included in a program can also be typed at the terminal. In this way, CGRASS functions as a command language as well as a programming language (similar to the UNIX shell). LISP-based programming environments have shown this approach to be successful for numerous applications, including interactive graphics.

The CGRASS program development environment allows code to be written entirely from within the system or imported from the outside. The system has a resident editor and the ability to invoke any other editor on the host operating system. Programs are debugged interactively at the source level, with assistance from the system in the form of on-line helps and descriptive error messages.

To simplify the user interface with the language, many data types are identical at the source level; the differences between types are embedded in the implementation. For example, files, strings, arrays, and lists may all be printed using the same syntax. This holds true for other operations in the language such as comparison, subscripting, etc. For the most part, disk files and strings behave identically; the use of the disk is hidden in the implementation.

Built-in DataTypes

CGRASS contains numbers, strings, and arrays, and provides ways to compare, subscript and do arithmetic operations. Strings are scalars in CGRASS, not arrays of characters as in conventional languages. They may be compared, concatenated, indexed, or executed. Another built-in data type is the variable-length list. Like strings, lists can be concatenated, indexed, or extended. CGRASS uses the same syntax for list manipulation as it does for the corresponding string or array operations.

Files are a special data type in CGRASS because they act like strings but may also be treated as programs. Any operation allowed on strings also works for files. In addition, they may be interactively debugged with the source level debugger. Files also have their own set of low-level input-output directives, making it possible to access individual lines or characters in an operating-system-dependent fashion.

Control Structures

Parameter passing in CGRASS is derived from the method used by its predecessor, GRASS3. A function does not have explicit parameters, argument input is done at run time, and the language provides a mechanism for automatically prompting the user when a required argument is omitted.

prompt 'What is your name'
input name NAME
prompt 'How old are you'
input integer AGE

In the example above, the prompt command will only be executed if there are no more arguments left to be parsed. The input command will fetch the next argument from the list passed to the function. If there are no more arguments left in the list, the program prompts the user at the terminal. This feature is especially useful for writing user interfaces.

In addition to the traditional if, while, and switch constructs, CGRASS supports goal-directed evaluation and generators. Generators allow a single expression to produce different values until a computationally useful one is found. Other languages like CLU, database systems, and command languages have similar constructs, but in a more limited setting.
Basic Primitives

Numerous built-in commands are provided to assist with input, output, type conversion, calculation, and debugging. Almost any data type can be printed on the terminal with the print command or input from the terminal with the input command. Output from a command or function may be redirected into a string or file using the > operator. Similarly, input can be redirected with < (like the UNIX shell). Functions exist to open and close files, and to read and write lines or characters. These functions provide a low-level communication path to serial ports as well as disk files.

CGRASS has a set of list- and string-processing functions that assist in scanning the aggregate types. Each indexable data type keeps track of the last element accessed. In the case of strings, an element is considered to be a line (not a character). At any time, one can refer to the first, last, current, previous, or next element of an aggregate.

function bubble {  
    input value V  
    for V[count(count(size(V) -1, 2, -1))]< next(V)  
        this(V) = > prev(V)  
    return V  
}

The function above performs a bubble sort on its argument, which must be indexable. Two nested count generators are used; the inner count generates subscripts starting at the back of the vector toward the first element and the outer count iterates from the first item to the inner index. Consecutive elements are compared and exchanged, with the operator, if they are out of order. Note that combined use of the scanning functions and goal-directed evaluation allows the body of the sort to be written in a single CGRASS statement without the use of temporary variables. This example illustrates how CGRASS can make a programming job easier by reducing the amount of information the programmer must handle.

Data Abstraction

In modern languages, abstract data types provide an important means by which the programmer may extend the language to include new data types not present in the base language.21, 22 CGRASS is no exception to this. Users may create their own data types and define how existing operators and functions apply to them. Operators may also be defined for built-in data types.

To illustrate how one goes about defining a new data type in CGRASS, let's define a table along the lines of SNOBOL4. For our purposes, a table will be a heterogeneous vector indexed by strings rather than integers. To keep the example simple, a linear search is used to look up each element; in reality, one would use a more efficient hashed-access method. The example below illustrates a class capable of instantiating and indexing a table.

```cpp
class table {  
    if _class = c_MAKE return table(listO)  
    input list TABLE  
    if _class = c_INDEX  
        {}  
        input string S  
        for each(V){1} = = $ return this(V){2}  
        V = $ list(list(S, null))  
        return last(V){2}  
    }
```

The class declaration defines a function that will be invoked automatically whenever an operation is performed on an object of the class. The system sets the global variable _class to indicate which operation to perform. CGRASS then invokes the user-defined class function; this function uses _class to dispatch to the appropriate section and returns the result of the operation.

Internally, our table is maintained as a list of index and value pairs. Each individual table element is a list whose first element is the index string and whose second element is the value of the element. To make a table we would do the following:

```cpp
abc = table() : instantiate table  
abc['first'] = 1 : give it elements  
abc['second'] = 2
```

When the first statement above is executed, the code associated with the class table is invoked with _class equal to c_MAKE indicating that we are instantiating a table. When the table is indexed, as in the last two statements, the class code is again called, this time with _class set to c_INDEX. In this case, the arguments to the class function are the table object and the index value. The code then searches the existing table elements, comparing their index strings to the one passed. If a match is found, this element is returned. If not, a new table entry is made and appended to the end of the list. Similarly, we could define other built-in operations such as print, each, this, etc., for our new data type.
APPLICATIONS

CGRASS was used in the development of many graphics applications. Some of these were prototypes which were later recoded in assembler and became part of the satellite processor repertoire. We developed tools to create and modify pictures, to define moving objects, and to animate them in various ways. Mechanisms were also provided for color animation and audio processing. For some satellites, one is given direct control over machine-dependent hardware features.

There are two kinds of display hardware in the game environment—vector (analog) and raster (digital). Digital systems can be further subdivided into character-mapped and bit-mapped architectures. CGRASS has been used to design tools for several different digital video displays of both types.

The remainder of this section demonstrates how CGRASS was applied in the case of a character-mapped architecture. We discuss the distribution of work between satellite and host and give examples of specific data abstractions.

Character-Mapped Architecture

Character mapping is widely used in CRT terminals and consumer electronics. It is simple, inexpensive, and supports dynamic motion in a somewhat limited framework. The screen is broken up into M-by-N pixel rectangles, each of which is assigned a pointer. The pointer for a given rectangle (cell) refers to the particular member of the character set that will be displayed in that position. In addition to a pointer to a character, an individual screen cell may have other attributes such as color, orientation, and so on. Some systems have programmable character generators with which users can define their own characters.

For the purposes of this example, we now describe a hypothetical, simplified character-mapped display. Each screen cell can have two attributes—a character number and a color. The background color of each character is fixed across the entire screen; the foreground color is variable. User-defined characters are not considered in this discussion.

The following class permits the programmer to view the screen as a two-dimensional array. Each element of the array has two attributes: the number of the character that occupies the cell, and the color of the foreground.

```c
#include <stdio.h>

class Cell {
    int charNo, color;

    Cell(int c, int col) { charNo = c; color = col; }
    Cell() : charNo(0), color(0) {}  // default constructor

    int get_char() const { return charNo; }
    int get_color() const { return color; }

    void set_char(int c) { charNo = c; }
    void set_color(int col) { color = col; }

    friend void put(Card* c, int col);  // put function
    friend void get(Card* c);           // get function
};

class screen {
    Cell** board;
    int screen_width, screen_height;

    screen() : board(new Cell*[screen_width*screen_height]),
              screen_width(0), screen_height(0) {}  // default constructor
    ~screen() { delete[] board; }

    void init(int w, int h) { screen_width = w, screen_height = h;
                            board = new Cell*[screen_width*screen_height];
                            for (int i = 0; i < screen_width*screen_height; i++)
                                board[i] = new Cell();
                        }

    void clear() {
        for (int i = 0; i < screen_width*screen_height; i++)
            board[i]->set_char(0);
    }

    void clear(int col) {
        for (int i = 0; i < screen_width*screen_height; i++)
            board[i]->set_color(col);
    }

    void put(int x, int y, int c, int col) {
        int index = x + y * screen_width;
        board[index]->set_char(c);
        board[index]->set_color(col);
    }

    friend void put(Card*, int, int);
    friend void get(Card*);

    friend void show(screen*, int, int);
};
```

There are only two operations defined for the class `screen` in the declaration above. No data are associated with members of this class because they are all maintained by the satellite processor.

Assignment into an object of type `screen` clears the entire screen to the given color. In order to produce any visible change, we must tell the game unit to clear the screen by calling the `gput` and `ggo` functions, which send the appropriate information to the satellite. In this case, we send a predefined opcode `(O_CLR)` to clear the screen, followed by the color we wish to clear it to. The `gput` function stores each of its arguments in an output buffer. Invoking `ggo` causes the accumulated contents of the output buffer to be sent to the satellite processor.

Although references to individual screen cells are trapped in class `screen`, the actual work is done by class `Cell`, described below.

```c
class Cell {
    int charNo, color;

    Cell(int c, int col) { charNo = c; color = col; }
    Cell() : charNo(0), color(0) {}  // default constructor

    int get_char() const { return charNo; }
    int get_color() const { return color; }

    void set_char(int c) { charNo = c; }
    void set_color(int col) { color = col; }

    friend void put(Card*, int, int);
    friend void get(Card*);

    friend void show(screen*, int, int);
};
```

The `Cell` class allows one to read or write the contents of an individual screen cell—that is, the character number and the color. The satellite has opcodes `O_CGET` and `O_CPUT` defined to read and write attributes of a particular cell. The `gget` function fetches the next input byte from the satellite processor, in this case the character number and then the color.

A scheme such as the one above buries a lot of the machine-dependent details inside the satellite processor. For example, the dimensions of the screen need not be known to the host; limit checks are made on the satellite. The communication mechanism used by `gput`, `ggo`, and `gget` is also transparent to the application.

One of our satellite processors has two serial ports and connects the terminal to the host. A single serial line transmitting ASCII hex format data handles all communication. On another system we use two serial lines. One line is used for host–satellite communications and uses a binary protocol; the other handles a terminal. Still another system uses a parallel port. The same low-level set of functions is used in all three cases. Whenever possible, we have tried to make the same opcodes accepted by different satellite processors.

Given a general view of a character-mapped architecture machine, we can go on to implement an outer layer of software tools using the abstract data type `screen`. The following
is an excerpt from a picture creation utility. It uses the numeric keypad on a standard terminal to move a cursor on the screen. The space bar controls whether or not the cursor leaves a trail as it moves.

```
OLD = screen[x,y]  : old screen cell
screen[x,y] = list(0, _col)  : draw cursor
if DRAW == 0  : do we draw?
  screen[x,y] = OLD  : no, restore cell
C = getch(0)  : get keypress
switch C  : dispatch
  {  :
case 0C8--y  : up
  case 0C2 + +_y  : down
  case 0C4--x  : right
  case 0C6 + +_x  : left
  case 0C DRAW = xor(DRAW, 1)  : toggle draw
}
```

Three global variables are maintained: _x and _y contain the coordinates of the current screen cell and _col is the current foreground color. OLD and DRAW are local variables that contain the displaced contents of the screen cell and a flag determining whether or not the cursor should leave a trail.

The paint program from which these lines were taken has many other features. The color (_l_col) can be chosen from a palette. An area of the screen can be reduced and made into a character. The cursor is selectable from the list of possible characters.

Using what we learned by implementing the character manipulation tools on our prototyping system, we were able to determine quickly what capabilities were needed and what view we wished to present to the user. Once the graphics interface is defined, performance enhancements that do not affect functionality can be made without rewriting applications.

For example, on one system we recoded part of the paint program, embedding cursor movement in the satellite. For the most part, the change was transparent to the rest of the software; the body of the cursor movement function is replaced with a small sequence of code, which asks the satellite for the cursor position. Thus, response is still quite good for simple functions even when the host is heavily loaded.

**FUTURE WORK**

The next step in designing a game-prototyping system is to completely remove the restrictions placed by the target system hardware. Making the number and size of moving objects variable, for example, would allow a game designer to concentrate more on the game and less on the limitations of the hardware.

The communications port is also somewhat of a bottleneck. Our prototyping efforts to date indicate that the host and satellite systems must be tightly coupled for efficient simulation.

We are currently working on a system that uses a high-speed, microcodeable frame buffer as a satellite. By defining very powerful real-time graphics primitives we hope to have

the satellite processor handle the bulk of the simulation with directions given by the host. We will be able to plug in various analog devices, such as tablets, joysticks, dials, etc., and use them to manipulate aspects of a simulation in real time. For example, one could control the position of a moving object with a joystick and its size or color with a dial. It is our belief that capabilities such as these will elevate the level of game design, making it possible to produce a playable game prototype in a very short amount of time.

**REFERENCES**
