111

Developing Programming Environments for Programmable Bricks

by

Kuo Ching Hsu

Submitted to the Department of Electrical Engineering and Computer Science

in Partial Fulfillment of the Requirement for the Degree of

Master of Engineering in Electrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 28, 1996

Author_______________________________

Department of Electrical Engineering and Computer Science
May 17. 1992

Certified by__________________________

Professor Mitchel Resnick

Accepted by______________________

Chair

Developing Programming Environments for Programmable Brick
by
KuoChing Hsu

Submitted to the Department of Electrical Engineering and Computer Science

May 28, 1996

In Partial Fulfillment of the Requirements for the Degree of
Master of Engineering in Electrical Engineering and Computer Science

ABSTRACT

Programmable Brick is a small hand-held computer for educational use. There are two programming environments to support programming the Brick. The first programming environment is a text-based, full-featured environment called BrickLogo. On the other end of ease of use is a simple programming environment called LogoBlocks, a drag and drop environment that manipulates graphical language blocks instead of text. These environments were initially implemented on the Mac. I re-implemented and redesigned these programming environments on the PC platform. In the process, I also added a useful software component to make it easier for designers to create new user interfaces for the Programmable Brick. The thesis gives a description of the protocol used between the host computer and the Brick, as well as the reasoning behind the major design decisions.

Table of Contents

1. Introduction

The Programmable Brick [1] is a small hand-held battery-powered computer developed at MIT's Media Lab. It was specifically designed for educational uses and has been successfully deployed in schools around Boston. Teachers and volunteers use it as a tool to teach school children about the fundamentals of robotics, computer programming, and physical modeling. Its small size affords easy mobility, so it is suitable for mounting on a robot as the control unit. The current Brick model, model 120, features 6 input ports and 4 output ports. The ports can be connected to sensor attachments and motor attachments using standard LEGO blocks. A wide variety of attachments are available, including light sensor giving readouts between 0-255, to trigger sensors such as push buttons. Instructions have been published on how to construct custom sensor and motor attachments [2].

Similar to a traditional general purpose computer, the Brick has memory chips to store its various programs. However, unlike a traditional computer, whose memories consists of ROM and RAM, the Brick's memory is actually EEPROM, so the Brick remembers user programs even after a power cycle. The Brick's memory can be roughly divided into an operating system memory area and a user program memory area. The OS memory area stores the operating system of the brick, which is a multitasked byte code interpreter. This OS memory area is initialized during a bootstrap process through the Brick's serial port. A new revision of OS can be downloaded to the Brick in minutes. However, in normal operation the OS is downloaded quite infrequently. The user memory area stores the more transient (although still persistent through power cycle) user programs. These programs are byte codes that are interpreted by the OS. They can also be downloaded to the Brick through the onboard serial port.

Users typically program the Brick in a language called "Brick Logo". Brick Logo is a variant of the Logo language, with additional primitives for motor and sensor controls. After a program is written in Brick Logo, it must be compiled into byte codes before it can be downloaded and executed by the Brick. A graphical user interface (GUI) program running on a host computer is typically the means by which a user indirectly interacts with the Brick. The GUI program accepts Brick Logo source code, compiles into byte code, and downloads the byte code to the Brick through a serial link between the host and the Brick.

One of the designing guidelines of the Brick project is that "Simple task should be simple, and complex task should be possible." Since the GUI program is what the users interact with the most, it is important to make the interface both intuitive for novice and functionally complete for experts. To that end, there are two different flavors of GUI programs available. For the novice users, a simple drag-and-drop environment called LogoBlocks is available (Figure 1). For experienced users, another GUI program called BrickLogo can accomplish anything that can be done with textual source code (see Figure 2). Both GUIs have been previously implemented on the Mac. For my thesis, I re-implemented and re-designed these GUIs for the PC platform. In the process, some new features were added. One such feature is an binary component called an OCX. The OCX is most useful to GUI designers building and customizing programming environments. The OCX has all the intelligence to compile Brick Logo source code and to communicate with the Brick, so it can be embedded easily in a controller program. This paper is a discussion of the design and implementation of these different programming environments for the Brick.

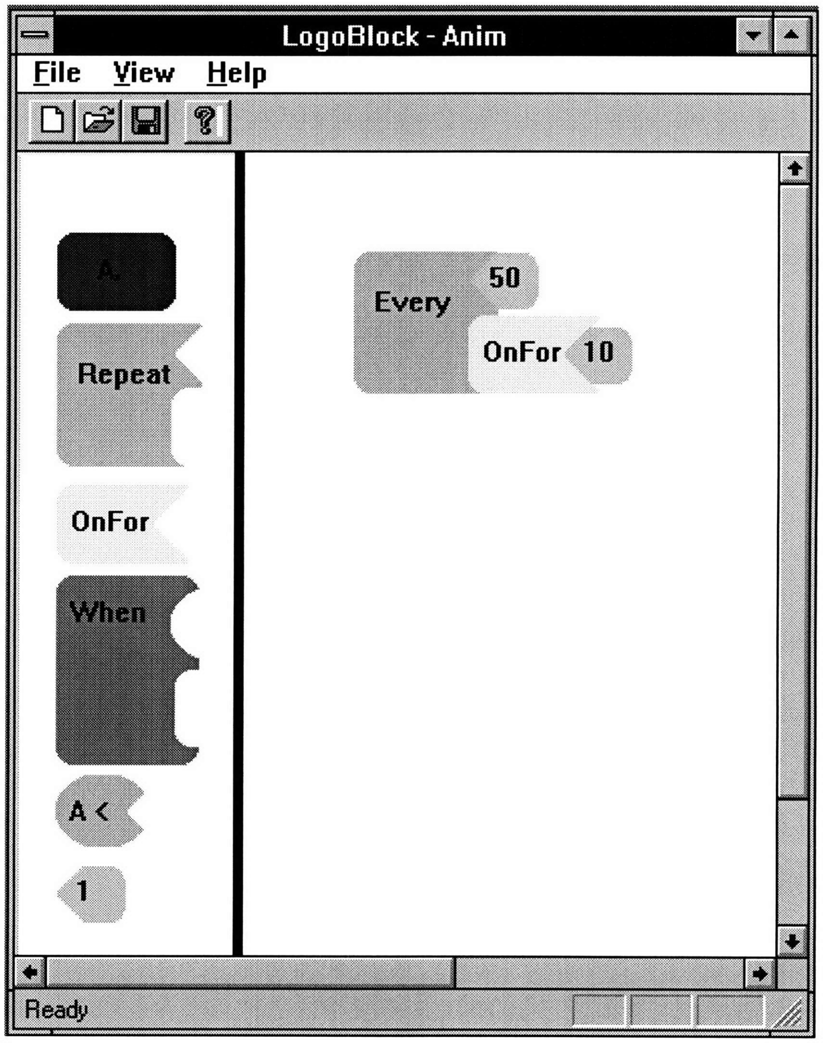
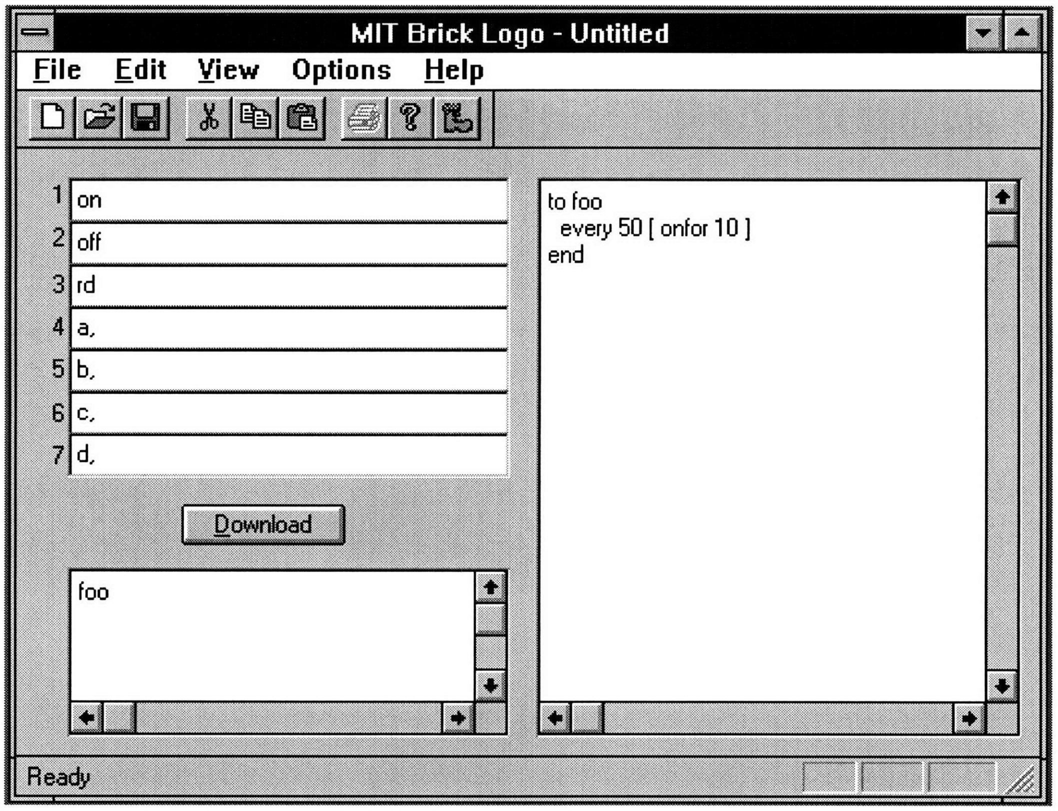


Figure 1. Screen shot of LogoBlocks, a drag-and-drop environment.



```
to foo
  every 50 [ onfor 10 ]
end
```

Figure 2. Screen shot of Logo program, a Brick Logo text-based environment.

2. BrickLogo and LogoBlocks

In this section we give a description of the two GUI environments from a user's perspective. We will describe the visible elements of the GUIs, and then go through a sample project in each environment, and show how a user would create a procedure, download it to the Brick, then executes it from the Brick.

BrickLogo

The BrickLogo program (Figure 1) is a text-based program that is the most functionally complete of all the environments. The screen of application is divided into three sections. On the left upper half of the screen are 7 menu items. These menu items correspond to the items shown by the scrollable LCD display of the Brick. A user can type into these menu items, and press the "download" button beneath the menu area to have the menu items compiled and downloaded to the Brick. No execution will be effected though. The menu items can only be executed by a user scrolling the Brick's display and press the "start" button on the Brick to start the currently selected menu item.

On the lower left half of the form is a command center area. Brick Logo source code typed into this area is immediately downloaded and executed by the Brick. Programs typed into this area are transient in that each new command overwrites the previous command.

On the right half of the form is the procedure area. Named procedures can be typed into the procedure area and be compiled and downloaded to the Brick by pressing the "download" button. The procedures downloaded are persistent, as are menu items.

As an example, after the user has typed in the code shown in figure 1 in the procedure area, the user can type in the command center "foo" to start running the procedure foo. The procedure "foo" then execute the code "Every 50 [onfor 10]", which turns on the currently selected motor for 1 second every 5 seconds.

The other visible element of the screen is the application menu bar across the top of the application. The application menu bar has the standard project manipulation functions. A new project in this environment can be started with either the blank form displayed when the application is just started, or with the FILE/NEW menu item. With a new project, the menu items are loaded with the default 7 menu entries: "a,", "b,", "c,", "d,", "on", "off", and "rd". These can be changed by typing directly into the menu items. Global variables and named procedures go in the procedure area. The user can experiment with Brick Logo statements in the Command Center area as well. After the menu area and the procedure area are modified, the "download" button can be pressed to compile and download the menus and procedures to the Brick. The procedures can be executed by issuing commands in the Command Center area. When the user is done with the current project, the FILE/SAVE menu item provides a way to save the current project in a file so it can retrieved the next time.

One other useful menu item is FILE/BOOT. Selecting this item will bring up another dialog for "booting" the Brick. Booting refers to the process of downloading the Brick OS, which is sometimes necessary after the Brick has crashed because of incorrect user program. The dialog shows a progress bar while it's booting. The booting process takes roughly a minute to complete. Follow the instruction on the dialog to prepare the Brick to boot.

LogoBlocks

The LogoBlocks is a graphical drag-and-drop environment (see Figure 2). The screen of LogoBlocks is much simpler than BrickLogo. On the left is a block palette, and the rest of the screen is a canvas that a user can drop the blocks on. To construct a new program, a user would drag from the block palette a selected block, and drop it on the canvas. Depending on where the block is dropped, it would either attach to an already dropped block, or would stand by itself at where it is dropped. Each type of block has its own behavior, and only compatible blocks that can be connected would attach themselves to each other. In this way, the user gets an instant feedback on what is permissible.

The labels on the blocks can be changed by pressing the right button and choosing an option from the pop-up menu. Different types of blocks have different pop-up menus. For number blocks, the number can also be entered directly while the cursor is on the number block.

After the blocks are constructed, the user can compile and download the program to the Brick by pressing the right button on the background of the canvas. A popup menu will let the user choose whether to display the source code or to compile and download to the Brick. If there is any problem with the constructed source code, a message box would pop up and explain the error. Most of the errors occur because the some required connections are not fully satisfied. Since the blocks perform some checks themselves, the chance of syntax errors is much reduced.

The example in figure 2 shows the same code as in figure 1, namely "every 50 [onfor 10]". After the user selects the right mouse button to start running the constructed program, the Brick should run the same program as in Figure 1.

3. Description of the Brick Memory and Communication Protocol

A description of the memory structure and communication protocol of the Brick is necessary before we dive in to the depth of the compiler and communication modules. This information describes the model 120 Brick, and future versions of the Brick may have different architectures.

The Brick memory is divided into several areas:

Purpose	Starting Address
Procedures	0xC400
7 Menu items	0xC001
Global variables	0xB800
Data Memory	0xD004
Command Center	0xFD00

The Procedure area stores the compiled procedures byte code. The byte code in this area is stored with no formatting. The compiler is responsible for keeping track of where a particular function is stored in this area.

The menu items area stores the 7 scrollable menu shown on the LCD display of the Brick. The text of each menu entry is followed by the byte codes of that entry. A two-byte byte count prefixes each text and byte code string so the runtime can quickly skip through the entries.

The global variable area stores global variables, with two bytes allocated for each global. Again, the compiler is given the task of maintaining the globals assignments. The data memory area is not currently used. The command center area stores the compiled byte code of the command center that is to be run immediately. The byte code stored here should be no more than 256 bytes long.

The low level serial communication between the host computer and the Brick follows a protocol dictated by the Brick runtime. The protocol is fairly straight forward, with the host computer issuing one byte command and the Brick responding to the command and sending back response bytes if appropriate. The Brick runtime recognizes the following commands:

Command	Arguments	Description
0x0	addr-hi addr-low	Set pointer
0x1		Read byte; inc pointer
0x2	byte to store	Store byte; inc pointer
0x3		"Do-it", run C.C. code
0x4		Stop program
0x5		"I-am-here", Sanity check

The byte 0x0 is sent to the Brick to set the current Brick pointer. The Brick maintains a current pointer pointing to it memory, and any read and write operation operates on the memory address points to by the current pointer. The 0x0 command resets the current pointer to the two byte argument following the 0x0 command. The two bytes should be sent with the high byte first.

The 0x1 command reads the value at the address pointed by the current pointer, increments the pointer by 1, and returns the byte read to the host computer. The command 0x2 sends a byte to be stored at the current address, and increments the pointer as well. A block read/write can be accomplished by first setting the pointer using 0x0, then issuing 0x1/0x2 repeatedly to read/write.

The command 0x3 instructs the Brick to begin executing the byte code in the Command Center area. The byte code must have been downloaded previously using 0x0 and 0x1 command and specifying the Command Center area as the starting address. Prior to downloading the Command Center code, the Brick should be stopped using the 0x4 command, or else the Brick runtime might continue to execute code in the same area while it is being overwritten (resulting in an unstable Brick runtime). The final command 0x5 performs a sanity check on the Brick runtime. If the runtime is still able to respond in its listen loop, then the runtime sends back a 0x55 respond byte. This command should be used to check if the Brick is actually connected, before any downloading.

4. Design of the Brick Logo Compiler

A prior implementation exists on the Mac. However, it was not possible to port directly from the Mac implementation since it was written using a proprietary system available only on the Mac. The Mac version was written in MicroWorlds Logo, a flavor of the Logo language. It was decided early on that the PC implementation would be done from scratch in C++. There were also some compiler works done using traditional compiler construction tools (lex/yacc). Although the works were done on PCs, the grammar specified and the implementation existed were not complete. An attempt was made to salvage the existing source code. However, after a careful evaluation, it was decided that it would be easier to rewrite the compiler component instead of updating the existing code, since maintenance of existing code was difficult.

Once we decided to rewrite the compiler component, we began looking at the Mac version and evaluate the feasibility of translating the Logo code to C++. There were some features of Logo that are not present in C++. Dynamic scoping of Logo, for example, does not have a direct counterpart in C++. After some discussion, however, it was concluded that the easiest approach would be to implement the missing features of the language, rather than coming up with a different scheme of parsing and compiling. Dynamic scope was mimicked in C++ by carrying an environment structure through call stacks, with all variables used in dynamic scope represented in the environment structure. Other missing features, such as list manipulation, were easily implemented as separate classes based on the C++ Frame work (MFC) used. Reusing the Mac architecture allows us to easily make changes to both the Mac and PC version of the software and keep them in sync.

Naming Convention

Throughout this project, we try to follow Microsoft's naming convention of naming classes starting with a prefix of capital C, as in CCompiler. Member variables start with the prefix m_, as in m_SymbolTable. Pointers are usually prefixes with p, as in pszLine. Although not strictly Hungarian, these simple naming conventions do help to make the source code much more readable.

Project Dependency.

The bulk of the compiler component is contained in a compiler dynamic link library (DLL). This DLL is used by both the Logo program and the Logo Block program. Much of the Logo OCX also borrows from the source code of the compiler DLL. The Logo OCX, however, can be built standalone and installed without the compiler DLL.

Class Hierarchy, Design, and Algorithms

The compiler DLL exports a top level class CCompiler to handle the task of compiling Logo Source code to byte code. A second class CBrick encapsulates all features of the Brick. The member functions of CBrick take the byte code generated by CCompiler and download them to the appropriate memory address in the Brick. There are also functions to directly communicate with the Brick through the low level serial link.

A number of supporting classes are used in the compiler project. All are used as structures with no inheritance. The data hierarchies are constructed by containment of support classes as member variable, rather than inheritance.

As in the Mac version, the compiling of Logo source code was broken down to 3 passes. Each pass processes the output of the prior pass and cooperatively the distinct passes convert the Logo source to byte codes. The first pass is the tokenizing pass, where the source code is tokenized and global symbols are gathered. Global symbols include function names and global variable names. These names are registered in a global name space and are used in subsequent passes. Functions are separated and the number of arguments to each recorded. Global variables are allocated space in the Brick memory implicitly by their order of appearance in the source code.

The second pass converts each function from the source code to some intermediate representations (p-codes in traditional compiler terms). Program logic is parsed and each statement examined for one of special constructs, function calls, or Logo primitives. Special constructs and primitives are translated to p-codes directly. Function calls are verified to have the correct number of arguments and that call semantics are followed correctly. Some constructs require recursive calls, and an environment block must be created to support dynamic scoping correctly.

The third pass of compiling converts p-codes to byte codes. This mostly involves a table lookup indexed by p-codes. The list of available Brick Logo primitives and their byte code is read in at compiler startup time, so it is relatively easy to extend the compiler to accept new Brick Logo primitives.

The complete header files and implementation of the compiler module can be found in Appendix A.

5. Design of Logo OCX.

Why OCX.

OCX is a term used to describe a specific class of binary software components, known also as OLE Custom Controls. OLE is a technology from Microsoft that evolved from a need to specify the interfaces between different Linked and Embedded documents, to a complete specification and implementation of how desktop applications should communicate with each other. Software that follows the OLE model can be ensured that they will inter-operate with each other. For example, an OCX can be hosted in Microsoft's Visual Basic, Borland's Delphi, Powersoft's Power Builder, or a number of other OLE controllers available.

To achieve interoperability, OCXs have to implement certain well-known interfaces so that containers of OCXs can at least have some way to treat all OCXs polymorphically. Fortunately, many of the interfaces needed are already implemented by the MFC C++ frame work that comes with the Microsoft Visual C++ compiler. Wizards in the MSVC IDE make the development of OCXs virtually trivial. A few clicks in the IDE is all that is needed to start a template project that will compile into a template OCX. The bulk of the work in designing an OCX is to define the interface that the users of the OCX will see, and in implementing functionalities that are not already provided by the Framework.

For the Logo Compiler OCX, we have the following properties and methods available:

```
                [id(1)] BSTR CommPort;
                [id(2)] brick_model_type Model;
                [id(3)] short SensorA;
                [id(4)] short SensorB;
                [id(5)] long DownLoadMenus(BSTR pszMenu1, BSTR pszMenu2, BSTR
pszMenu3, BSTR pszMenu4, BSTR pszMenu5, BSTR pszMenu6, BSTR pszMenu7);
                [id(6)] long RunCommandCenter(BSTR pszCC);
                [id(8), propget] short Sensors(short whichSensor);
                [id(7)] short DownLoadProcs(BSTR pszProcs);
```

The property CommPort directs the Brick's communication module to open the specified communication port. Valid values for CommPort are COM1 through COM4.

The property Model controls the hardware echo feature of the communication module. Current model of the Brick (model 120) does a hardware echo for each byte sent. Future model of the Brick will not have this hardware echo enabled. This property, if set to 120, enables the Brick to expect the hardware echo.

The properties SensorA and SensorB returns the current reading of on board sensor A and sensor B. These values are read-only.

The method DownLoadMenus( ) accepts 7 strings as input, and compiles and downloads the byte code to the menu area of the Brick. This method encapsulates the intelligence to run the compiler and package up the resulting byte code in a list format required for the menu area. Not all 7 menus need to be specified. NULL can be passed in some menu items and the corresponding menu item will be empty when downloaded.

The method RunCommandCenter ( ) accepts a string as input. It compiles, downloads, and starts running the instruction in the command center.

The method Sensor( ) is a more generic way to get the readings of onboard sensors. This method takes an integer input ranging from 1 to 6, and returns the sensor value corresponding to Sensor A through F.

The method DownLoadProcs ( ) accepts a string as input. It compiles the string as procedures and download them to the procedure area in the Brick. The program downloaded can be invoked via either the command center or the 7 menu items.

Sample Logo program using the Logo OCX

A sample program has been written in Visual Basic that demonstrates the power of the Logo OCX. The Visual Basic program was constructed in less than a day, and yet it has the same look and feel, and 80% of the features of the Brick Logo program written in C++. The reason it can be done so quickly is because all functionality is encapsulated in the OCX, so the GUI designer has to concern only with the GUI layout, and not with the details like serial communication to the Brick, or compiling Brick Logo source code.

Features Missing.

A number of features are missing from the OCX due to time constraint. First, there should have been better error handling. Currently the OCX will throw up a dialog if there is a parsing error (syntax error, for example). A better interface to the OCX user would be to throw an exception from the OCX, and have the OCX user catch the exception and display the error, if appropriate. One can imagine in certain situations that popping up a dialog box would be an inappropriate response. For example, when the OCX is used remotely, with no operator attending. A dialog box in this situation would block the program until the dialog box is dismissed by some human operator.

Another desirable feature not present is the ability to monitor the Brick dynamically. For example, if a user starts multiple threads, the user might like to know whether any of those threads of execution has completed, or perhaps whether any of condition was satisfied. Currently there is no runtime support for such dynamic monitoring.

6. LogoBlocks.

For novice users, the need to learn a programming language, even a simple one, might be too intimidating. A simple drag and drop programming environment was developed to help novice users quickly see the result of their "programs" without typing in the programs. In this environment, language constructs are presented to the users as "blocks" on the screen that can be assembled together to form a program. Blocks of compatible types would automatically snap together when one is dropped near the vicinity of the other. This instant feedback is very useful in educating the user on how to use the various programming constructs. The edges of the blocks also suggest the types of blocks that would fit together; round concave edge naturally suggests a complementary round convex edge from another block.

One of the design goal in the Logo Block project is to build a supporting infrastructure in which we can extend the functionality easily in the future. It was decided that one such supporting infrastructure is the underlying sprite system. A sprite is an animated image, usually with transparent background. A sprite system would handle the creation and destruction of sprites, manage the image (bitmaps) of the sprites, manage the color palette the sprites use. The sprite system should also be implemented in a way that avoids flickering, most commonly using a scheme known as double buffering.

In choosing a sprite system, we evaluated many options:

Our first option is the off-the-shelf component market. We evaluated the Gurewich controls, which has a sprite OCX from TegoSoft [4]. As with other OCXs, the TegoSoft sprite OCX is easy to use. A demo program shipped with the control shows that with just a few lines of BASIC code, a program can add sprite animation to its feature list. However, there were some problems with the OCX approach. One of the most serious one is that the OCX cannot be easily extended. In the system we desire, we need to label each block with some text string. Although we could add the text to the bitmap, there was no easy way of dynamically changing the text. Another problem was performance. The OCX performs well when we had just a few sprites, but seemed a bit sluggish when more sprites were added. It was probably because Visual Basic is an interpreted environment and cannot be scale well.

Next we considered writing the sprite system ourselves. Our target platform is Windows 95. On Windows 95, there are several choices of APIs available for writing graphic applications, depending on the sophistication and the performance requirement of the applications. The first one we investigated was the WinG technology. The WinG is targeted for game developers who need very fast and efficient graphic operations. The problem with WinG was that it does not work with GDI operations easily. For text manipulation, and to incorporate non-image blocks (such as a "gauge" block), we would like to use GDI operations so we can take advantage of the text rendering and some control rendering (drawing a gauge, for example) GDI offers. WinG does not seem to have this feature.

A newer API, available only on the Windows 95 platform, supersedes the WinG API. The new API is called the Direct Draw API, and is part of the DirectX technology designed to entice game developers to Windows 95. Direct Draw offers the same or better performance as WinG, with a more consistent interface. The DirectX API represents all objects using the Component Object Model, which is standard across the DirectX API set. Although there are better supports for sprite systems in Direct Draw, it is not clear how Direct Draw objects would coexist with GDI objects.

The traditional way to writing graphics applications on Windows is with GDI operations and manipulates GDI objects (e.g., bitmaps) directly. There are two types of GDI bitmaps available, the old Device Dependent Bitmap (DDB), and the newer Device Independent Bitmap (DIB). DIBs are available on the newer operating systems, such as Windows 95 and Windows NT 3.51. DDB was the only bitmap format in Windows up to Windows 3.1. The problem with DDB is that it's hardwired to a particular device, so when the image is displayed on a different display device, there is no guarantee what it will look like. DIB was introduced to solve the problem of device dependency. DIB bitmaps include extra information such as image resolution parameters and color palettes (instruction for color rendering/matching), so consistent rendering can be achieved. The newer operating systems also have new APIs to support DIBs, which make DIB operations as efficient as WinG. In fact, WinG is now layered on top of DIBs. Overall, DIB is the most flexible option since we have to manage the bitmap memories ourselves and we can use GDI operations on these bitmap memories.

After evaluating the different options available for a sprite system, we decided to go with the traditional approach of using DIB bitmaps. Fortunately, we did not have to write the sprite system from scratch. There are source code available that implement a sprite system completely[5]. What we had to do was to tweak the existing system so it has the features we need. We also reused the color palette management code, since it is very much an essential part of the graphics system.

One design criteria of the Logo Block project is that the sprite system be extensible. It is likely that we would add new primitives and new block types as we extend the capability and vocabulary of the Brick. Subsequently, it is important to keep the design flexible so that such extension can be done easily. To this end, we grouped the common functionality of block and sprite management in a base class, and derive from that base class distinct types of block that have specific behaviors. Each type of blocks has its own bitmap, as well as its own rules of connection and code generation. To add a new type of block, one derives from the base class a new subclass, overrides a few absolute virtual member functions, adds an instance of the new class to the block palette, and the rest is taken care of by the sprite system. The sprite system manipulates all blocks polymorphically using only the virtual functions of the base class.

The design of the Logo Blocks specifies that blocks should have some physically intelligent behavior when grouped together. One such behavior is that the aggregate should represent a valid Logo program. Our algorithm in doing this is to follow some simple conventions in deciding how an aggregate maps to the Logo program. It is decided that blocks should read from left to right and top to bottom. Therefore, for each block we designate either the top or left edges as the leader edge, and we follow the leader edges from block to block to arrive at the top left most block, which is also the beginning of the program. From that block, we traverse downward to expand the Logo source code.

Each type of blocks has certain available connection points when created. Each connection is of a particular type, and only compatible connections can cause the adjacent blocks to snap together. For example, Action blocks can be stacked together vertically, and the connections between the blocks represent sequential executions of Logo statements. To represent such an ACTION connection, the Action block has a ACTION_BOTTOM connection at the top edge and an ACTION_TOP at the bottom edge. The ACTION_BOTTOM connection can only be matched with a ACTION_TOP connection. Other blocks that can be executed sequentially also have ACTION_BOTTOM and ACTION_TOP connections at top and bottom edges, respectively.

Each block also has a label that identifies what Brick Logo construct it represents. This label can be changed using the right mouse button. Primitives and constructs with similar connection points are grouped under one type of block. For example, the Repeat block can be either "repeat" or "every". The "repeat" construct takes a number and an action, and executes the action the specified number of times. The "every" construct also takes a number and an action block, and execute the action every specified number of seconds.

Both constructs have a NUMBER_LEFT and a ACTION_LEFT input, and so are grouped under the "repeat" block.

To facilitate code generation, each type of block has its own code generation rules. Most rules simply use the current label as the code and insert the code from the right and bottom connections into the appropriate place, format the resulting code fragment with white spaces and brackets if necessary, and return the resulting code fragment to its leader block.

As an example of the process of generating Brick Logo source from blocks, consider the blocks in figure 3.

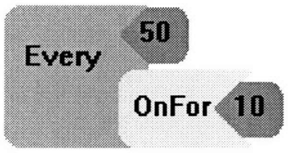


Figure 3. An example Logo Blocks program that represents "Every 50 [onfor 10]".

In figure 3, we have a "Repeat" block that has been changed to "every", a number block attached to the number argument to the "Every" block (50), an "OnFor" block that is the action part of the "Every" block, and finally another number argument to "OnFor" (10). Between the "Every" block and the number 50 block, and between the "OnFor" block and the number 10 block, are NUMBER connections that consist of NUMBER_LEFT and NUMBER_RIGHT from the joining blocks. Similarly, between the "Every" block and the

"OnFor" block is an ACTION connection that consists of an ACTION_LEFT from the "Every" block and an ACTION_RIGHT from the "OnFor" block. To generate Brick Logo source code from figure 3, we would randomly select any block from the above 4 blocks, and follow the leader edges of the blocks until we reach the top-left most block of the lot. For example, say we select the "OnFor" block as the starting point. The "OnFor" block has as its leader edges the top and the left edges. Since there is no connection to the top, we follow the left edge and arrive at the "Every" block. The "Every" block is a top level block and does not have any leader edges designated, so the search ends there and we begin constructing the source code from this point. The "Every" block takes the numeric argument from its NUMBER_LEFT connection (which is 50), appends the code from its ACTION_LEFT connection (which is "OnFor 10", derived similarly), and strings them together to arrive at the final source code: "Every 50 [OnFor 10]". This source code is then either displayed to the user or compiled and downloaded to the Brick.

Types of Blocks Available

CIfLessBlock.

This block represents the "If <predicate> then <action>" construct, where <predicate> is a comparison of some sensor value and a number. The available connections for this block are: ACTION_BOTTOM at the top edge, NUMBER_LEFT at the right edge, and ACTION_TOP at the bottom edge. The label for this block is of the form "if <x> <op>", where <x> is one of the sensors (A through F), and <op> is a binary comparison operator (>,<,=).

CNumberBlock.

This block represents an integer number. There is only one available NUMBER_RIGHT connection at the left edge, and the label is simply the current integer value displayed. The value can be either chosen with the right mouse button, or type in through the keyboard while the mouse pointer is over the number block.

CActionBlock.

This block represents a generic action primitive. Possible labels are: motor selection (A through D, and combinations of), motor actions (on, off), direction selection (thisway, thatway, rd). Available connections are: ACTION_RIGHT, ACTION_LEFT, ACTION_BOTTOM, ACTION_TOP at left, right, top, and bottom edges, respectively.

CRepeatBlock.

This block represents either a "repeat" or "every" construct. A number specifies either the repeat count or the wait interval between actions. Available connections are: ACTION_BOTTOM at top edge, ACTION_TOP at bottom edge, NUMBER_LEFT, at the right edge, and ACTION_LEFT at the right edge.

COnForBlock.

This block represents either a "onfor" or "wait" construct. Both constructs take a number input. "Onfor x" turns the currently selected motor on for x numbers of seconds, "wait x" waits for x numbers of seconds before continuing with the next action. Available connections are: ACTION_BOTTOM at top edge, ACTION_TOP at bottom edge, ACTION_RIGHT at left edge, and NUMBER_LEFT at right edge.

CWhenBlock.

This block represents either a "when" or "if" construct. An expression (CBoolBlock) is a required connection. Available connections are: BOOL_LEFT at right edge and ACTION_LEFT at right edge.

CBoolBlock.

This block represents a logical boolean expression of the form "<sensor> <operation> <number>", where sensor is one of A through F, operation is one of (=, <, >), and number is a NUMBER connection. Available connections are: BOOL_RIGHT at left edge and NUMBER_LEFT at right edge.

7. Summary.

In this paper we discussed the design and implementation issues of the various programming environments for the Programmable Brick. The first is a GUI program capable of interpreting Brick Logo source code directly, thus is the most functionally complete environment. For novice user we also have a GUI program that manipulates blocks representing programming constructs. Users program the Brick by drag and drop blocks to form a pictorial representation of the underlying Brick Logo code. Finally we wrap the functionality of interfacing with the Brick in a reusable software component (OCX). This component can be inserted in a variety of GUI environments, thus making possible rapid application prototype based on the Brick technology.

<u>References</u>

[1] The Programmable Brick Handbook. Epistemology and Learning Group, MIT Media Laboratory, April 1995.
[2] Building Sensors for the Programmable Brick, by Fred Martin, MIT Media Laboratory, April 1995.
[3] MicroWorlds Reference. LCSI Logo Computer Systems Inc. 1993.
[4] Gurewich OLE Controls for Visual Basic 4, by Ori and Nathan Gurewich. SAMS publishing, 1995.
[5] Animation Techniques in Win32 by Nigel Thompson. Microsoft Press, 1995.