UDC 621.397: 681.3

High Speed CG and Simulation Application Development Environment "Firstsight"

●Fumio Nagashima ●Kaori Suzuki ●Tsugito Maruyama

(Manuscript received June 5,1997)

Recently, the efficient development of software is becoming more and more important as multimedia technology progresses. In particular, software for creating and simulating images based on three-dimensional computer graphics requires an increasing number of man-hours to develop. This development process must be reconsidered from the viewpoint of reusing software components. No one has, however, succeeded in developing reusable software components that offer both general-purpose applicability and high-speed processing. The authors have developed a computer graphics and simulation application development environment which allows the application designer to reuse software components with high-speed processing and time control capabilities. This environment is named "Firstsight". The basic concept of the Firstsight system is software LSI. It is an improved concept of constructing componentware based on the concepts of object-oriented and data flow programming. Software LSIs bear many similarities to hardware LSIs. This is a totally new method of constructing realtime applications.

1. Introduction

We have developed a fundamental programming technology which offers high efficiency in both the development and execution of software. Based on this technology, we constructed a program development environment called "Firstsight". The basic concept of the Firstsight system is software LSI.

The technique of creating 3-D graphics extends over a broad variety of fields. It is thus difficult for a small number of people to develop a system for creating contents. Such a system should be constructed through the cooperation of experts in many fields. The authors of this article represent groups that specialize in two different fields. We have discussed the problem of cooperation and found a way to solve it.

The two groups mentioned include a group specializing in the creation of CG-based walkthrough simulation systems[1] and a group that specializes in robot dynamics.[2] Both groups use computer graphics. A program created by one group, however, could not be used by the other group as is. In most cases, creating a new program was faster than modifying a program created by the other group. Although both groups used object-oriented programs coded in C++, the programs were not exchangeable because of critical differences in their methods of selecting objects.

Consequently, we developed a fundamental programming technology to enable the use or reuse of these programs, and constructed an application development system based on this technology. This is based on general-purpose componentware technology. General-purpose componentware offers very high development efficiency because each software component can easily be reused. On the other hand, componentware specific to a particular data format cannot connect two systems developed on different concepts, so that component reusability is extremely low.[3] With general-purpose componentware, however, efficiency in program execution is generally low.[4), 5]

We have determined that we must solve the problem of "improving execution efficiency using general-purpose componentware."

2. Problems with conventional methods

We will first describe problems with conventional methods. This chapter outlines the two major concepts for componentware, object-oriented programming[6] and data flow programming, [7] and describes the problems inherent to each.

2.1 Problems with object-oriented programming

An object-oriented program consists of objects and the messages passed between them. **Figure 1** illustrates their relationships.

Message passing creates a bottleneck in program execution. The efficiency within each object can be improved to any degree determined by the designer of the object, because the object need not expose its internal structure, including its data and processes. It is difficult, however, to improve the efficiency of message passing. **Figure 2** is a schematic diagram of an object handled in message passing. Low efficiency in message passing is caused by:

1) Large messages
2) Processing required for eliminating algorithm-dependent data

Each message contains not only the type of processing to be performed but also the parameters required to execute that processing. Some messages are, therefore, very large in size. Programs which handle three-dimensional computer graphics contain particularly extensive data in the parameter section. Moreover, the data must be converted to a more generic structure to hide the internal structure of each object. The process used to hide algorithm-dependent data structures degrades the performance of the entire object-oriented program.

2.2 Problems with data flow programming

Along with object-oriented programming, data flow programming is another concept on which the creation of componentware is based. Data flow programs are created by describing the flow of data. Implementing data flow programming in current computer architecture will, however, give rise to the following two large problems:

1) Low execution efficiency caused by the monitoring of data changes
2) Low component independence and reusability caused by data sharing

Each component of a data flow program is triggered by a change made to any data related to itself. Each component must, therefore, constantly monitor data changes. This monitoring operation is automatically performed by the data flow programming system, so it remains invisible to the application creator. Thus, the application program can be coded very simply to improve its clarity. The system, however, monitors data changes regardless of whether they affect each component. The processing time required to monitor data changes, therefore, degrades the overall execution efficiency.

Data flow programs sometimes share rather than transfer data to improve the processing speed. Systems rarely transfer data on images or three-dimensional figures. Such data sharing, however, involves a great danger. Sharing in a casual manner may cause algorithm-dependent data to leak out of its component, thus degrading the independence of each component (see **Fig.3**). That is, any component which references and uses algorithm-dependent data from another component will become dependent on that same algorithm. Such components, which can cooperate with only those components using a particular algorithm, cannot cooperate with components which execute the same function but use a different algorithm. This greatly reduces the reusability of components.

3. Development concepts

Firstsight is meant to use highly independent componentware to construct applications requiring high-speed processing, such as realtime CG and simulations. For this purpose, we must address the issues that lead to degraded performance and

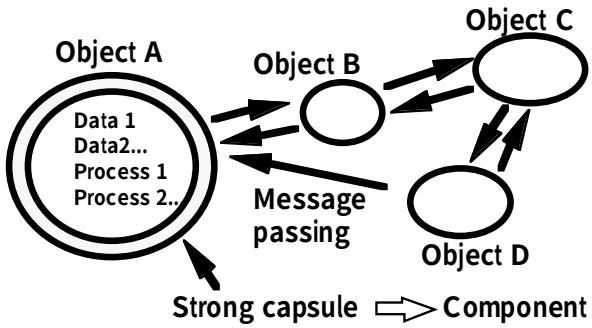


Fig.1– Object-oriented program.

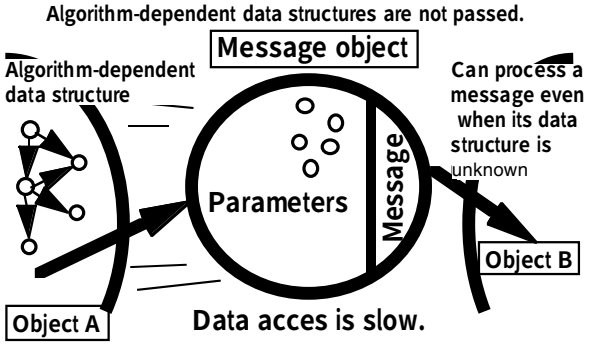


Fig.2– Object-oriented message.

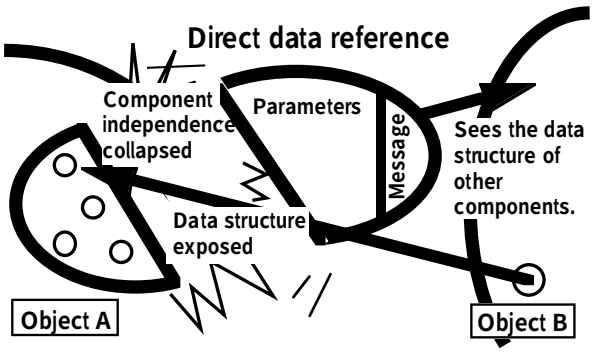


Fig.3– Danger in direct data reference.

independence as described in Chapter 2. This chapter first describes the basic strategies for solving these problems. We have established these strategies after comparing several candidates. A method to solve the problems based on these strategies is then described. Finally, the concepts are summarized. The basic strategies are as follows:

1) In terms of object-oriented programming, messages and methods should be divided into instructions and data.
2) Data should be distinguished as primitive data when independent of the algorithm.
3) Primitive data should be automatically expanded to allow for direct reference.
4) The above processes should be dynamically executed at program run time.

The following sections detail these strategies sequentially.

3.1 Eliminating large messages

One factor which degrades the efficiency of executing object-oriented programs is large messages. Firstsight separates the parameter section from the message, thus reducing the message size. We refer to the separated parameter section as data and the remaining part of the message as an instruction. As described later, the data will be passed between components separately from the instruction. From the viewpoint of data flow programming, this strategy enables describing the flow of control as well as the flow of data.

The problem of large messages has thus been completely resolved.

3.2 Protecting algorithm-dependent data

The independence of each component collapses when its internal algorithm-dependent data becomes exposed. To maintain the independence of each component, therefore, algorithm-dependent data must be protected. Attempting to prohibit the exposure of the data itself, as is done in object-oriented programming, does not have much effect. Firstsight has employed the following strategy to maintain the independence of each component. Because the algorithm mainly affects the structure of the data (such as the tree structure and network structure), preventing the exposure of the data structure will preserve the independence of each component. Therefore, the data must be disassembled into primitive data at an early stage in the program's execution. This strategy maintains the independence of each component, resulting in high component reusability.

3.3 Direct referencing of algorithm-independent data

Primitive data, when disassembled as described in Section 3.2, is independent of the algorithm. Such data can be directly referenced without affecting the independence of the component, thus enabling high-speed data transfer. **Figure 4** shows the concept of data reference.

This strategy enables high-speed data processing separate from messages.

3.4 Sharing data at an early stage of program execution

A program usually does not include data before being executed. Therefore, the expansion of data for direct referencing, as described in Section 3.3, must be performed immediately after the program receives data.

As a result, we have enabled the direct referencing of primitive data to be performed dynamically during the execution of the program.

3.5 Birth of software LSIs

Strategies for improving the execution speed while maintaining component independence render components having two clearly discriminated interfaces: those of instruction and data. In traditional software, functions and messages are always called together with parameters, and separating the parameters from functions and messages was unthinkable.

Although such discrimination is rare in the software field, it is often encountered in the hardware. It is similar to the discrimination of LSI pins. Firstsight components have a larger number of "pins," compared with those of object-oriented and data flow programs. Thus, we have decided to refer to these components as "software LSIs" (see **Fig.5**). We also use other LSI terms to represent several Firstsight elements. A data sharing symbol is referred to as a data bus, an instruction symbol as an instruction bus, data sharing specifications as data bus connections, and instruction specification as instruction bus connections. **Figure 6** shows example software LSIs, displayed with a Firstsight tool.

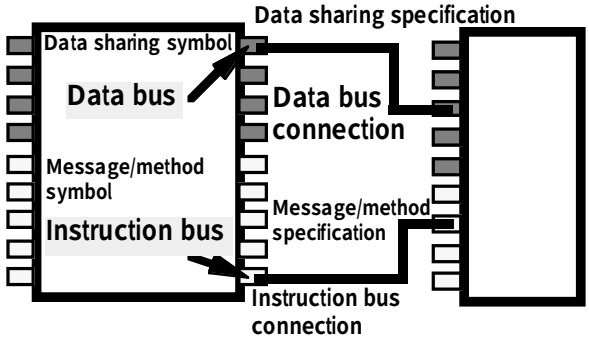


Fig.5– Software LSIs.

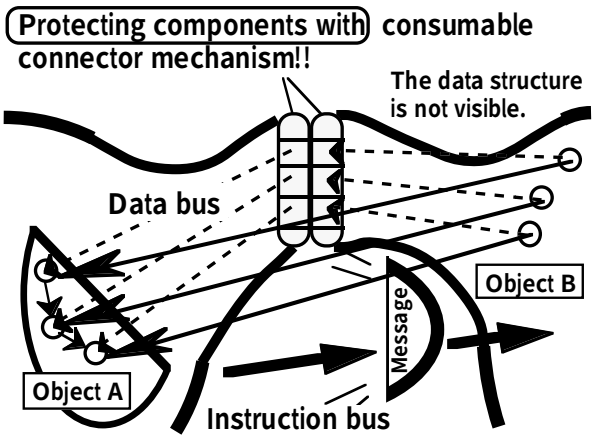


Fig.4– Data referencing in Firstsight.

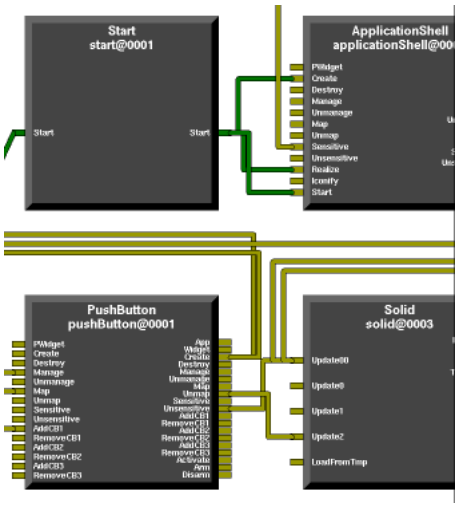


Fig.6– Software LSI displayed with a tool.

3.6 Co-LSIs

It is not necessary to strictly maintain the independence of all software LSIs. Conventionally, many ordinary applications have been created by placing several processes around a some data structure. The Firstsight system also allows applications to be created in such a way and refers to such processes as co-LSIs, as they constitute LSI groups dependent on particular data. Note, however, that the data of multiple software LSIs having different algorithms can be shared using the software LSI base class function, even if they have been created in the above way. That is, co-LSIs offer substantially higher reusability, as compared with conventional programs.

4. Implementing software LSIs

Implementing those processes which all components must have in common, such as the software LSI functions, can be performed by inheritance, a mechanism of object-oriented programming. Software LSIs have been implemented using inheritance supported by C++. We first created the software LSI base class. The main tasks of this class are as follows:

1) Processing an instruction bus
2) Processing a data bus
3) Maintaining a software LSI class name
4) Maintaining a software LSI object name

Note the following for data bus processing. The data must be disassembled into primitive data by the component creator who alone knows which data is primitive data and which is algorithm-dependent. The software LSI base class only shares the disassembled primitive data automatically.

5. Performance measurements

Figures 7 and **8** show the measured differences in the execution speed between data transfer with conventional object-oriented programming and that with Firstsight. The figures show the times required to reference and set a single data item in an object. As shown in the figures, Firstsight processes data two or three times as

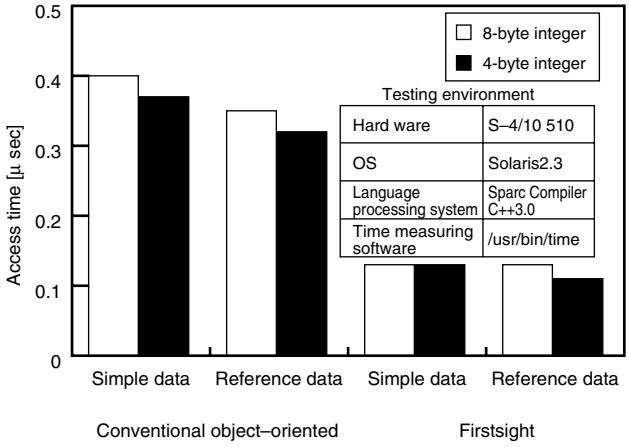


Fig.7– Comparison of access time for a single data item (reference).

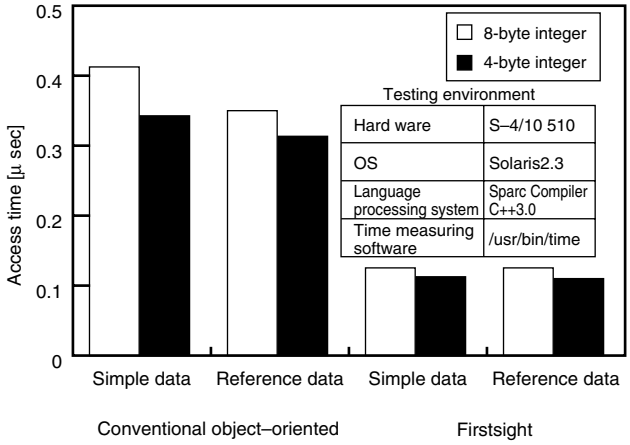


Fig.8– Comparison of access time for a single data item (set).

fast as conventional object-oriented programming. This difference is mainly derived from the time required to manipulate the stack, which is used by an object-oriented program to transfer data. **Figure 9** shows the time required to reference multiple uniform data items in an object. Object-oriented programs do not directly expose the internal data of an object. Object-oriented programming, therefore, requires the conversion of the internal data to a more generic format. When an object contains multiple data items, an object-oriented program requires a process for searching the data group for the target data. Thus, data access becomes longer time as the number of data

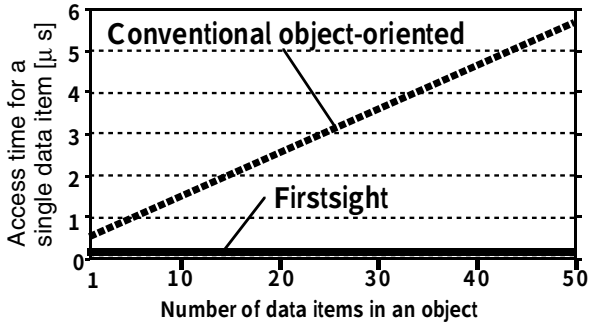


Fig.9– Comparison of access time for multiple data items (reference).

items increases. Firstsight directly references algorithm-independent primitive data, thus eliminating the need to hide the internal data or search for the target data, so that processing time does not increase with the number of data items.

6. System overview

This chapter gives an overview of the current Firstsight system. The basic unit of the Firstsight system is software LSIs. Both system expansion and application enhancement are performed by adding new software LSIs.

Figures 10 and **11** show the work assigned to the software LSI creator, the software LSI user, and Firstsight. The software LSI creator is in charge of:

1) Creating a primitive data collection process
2) Creating methods and messages

The software LSI user is in charge of:

1) Connecting data buses
2) Connecting instruction buses

Firstsight is in charge of:

1) Creating connectors, sharing primitive data, and deleting connectors
2) Executing the method and message processing flow

Figure 12 is a conceptual diagram of the system operation. A Firstsight user first uses an LSI builder to create a software LSI based on existing software resources. The created LSI is stored in the local archive. From this archive, the application programmer selects the appropriate LSIs and then creates the application by connecting the

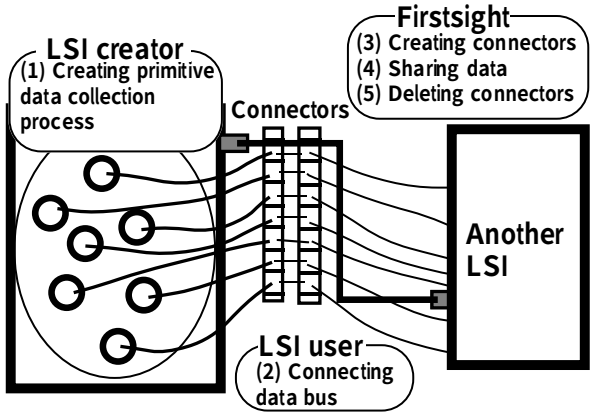


Fig.10– Work assignment (data bus).

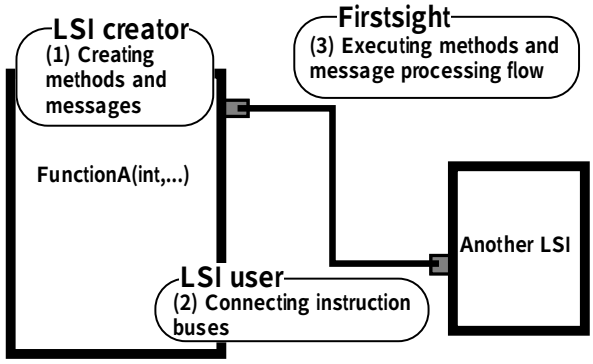


Fig.11– Work assignment (instruction bus).

Can be operated as a stand-alone or with a LAN or WAN.

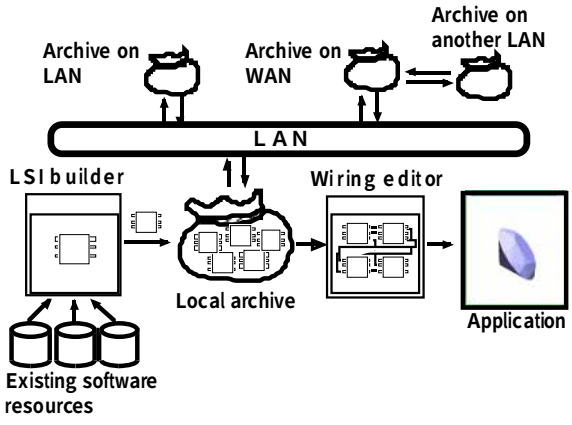


Fig.12– System operation conceptual diagram.

instruction and data buses using a wiring editor.

If such a system is connected to a LAN, the local archive allows the LSIs to be shared on the LAN. Thus, the user can obtain any LSI available on the same LAN or use a WAN to download an LSI from another LAN.

Figure 13 shows the main tools used in this system and the major software LSIs for which development is nearly completed. Several important tools and basic software LSIs are described below.

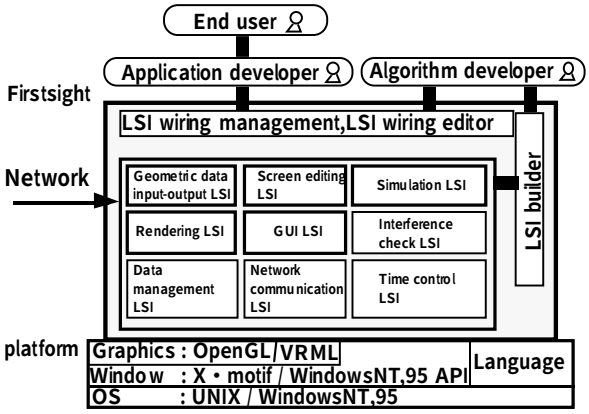


Fig.13– Current system configuration.

6.1 LSI builder

To create a software LSI, messages and methods must be disassembled into instructions and data. The bulk of this disassembly process is, however, mechanical work. To improve efficiency, therefore, the system provides a tool that supports the disassembly process. **Figure 14** shows the LSI builder screens. The screens include the appearance of the software LSI being created and a dialog box used for bus definition. This LSI builder has been designed as a Firstsight application using 15 software LSIs.

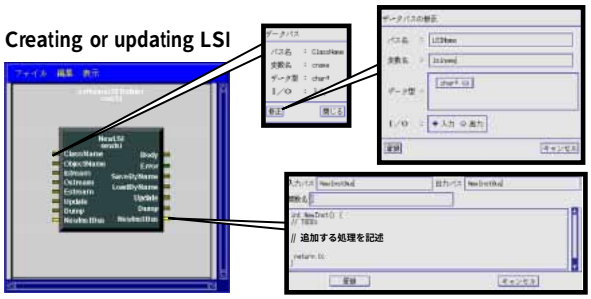


Fig.14– LSI builder.

6.2 Wiring editor

The application creator must connect wires between software LSIs. The wiring operation is very laborious work if performed on a character-based system. The Firstsight system, therefore, provides a tool that visually displays the wiring and allows the application creator to edit them. **Figure 15** shows the wiring editor screens. In this example, the system is performing a realtime dynamic simulation of the motion of a chain with power applied. This wiring editor has been designed as a Firstsight application using 19 software LSIs.

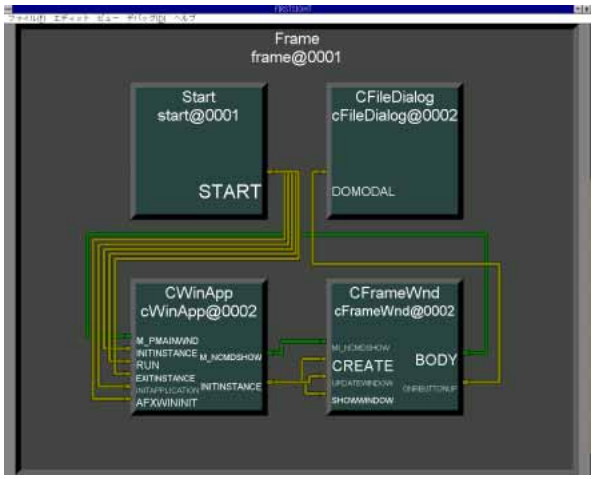


Fig.15– Wiring editor.

6.3 Frame LSI

An LSI used to group several LSIs is called a frame LSI. Frame LSIs are classified into several types. Basic frame LSIs are described below.

6.3.1 Circuit frame LSI

This is the simplest frame. Its functions include wiring its internal LSIs and enabling specified LSI pins to be shared with outside components.

6.3.2 Dynamic circuit frame LSI

This frame contains internal LSIs together with their wiring prototypes and generates or deletes specified LSIs and wires at any timing during program execution. The dynamic circuit frame supports the description of an object which is generated or deleted during program execution.

6.3.3 Parallel processing frame LSI

This frame operates its internal LSIs as separate threads or processes. This parallel processing originally includes process synchronization and data lock mechanisms.

6.3.4 Distributed processing frame LSI

This frame operates its internal LSIs using different CPUs. Its basic operation is the same as that of a parallel processing frame.

7. Discussion

Firstsight has more object interfaces than conventional object-oriented programming systems. In addition, data is transferred through multiple LSI pins simultaneously. This is similar to parallel interfacing in computer hardware. Conversely, conventional object-oriented programs transfer messages and their parameters through a single logical channel. This is similar to serial interfacing in computer hardware. Computer hardware uses a parallel interface, such as a SCSI bus, for high-speed processing, while it uses a serial interface, such as Ethernet, for connections between individual computers. Software also requires such natural interfaces to select from according to its purposes. Conventional systems relying exclusively on object-oriented programming result in low execution speeds. The system proposed in this article will enable the creation of optimum software.

7.1 Comparison with an object-oriented programming system

Object-oriented programming systems, such as IDL, exchange information between objects by adding parameters to messages. Firstsight divides the information to be exchanged into a message section and parameter section, each of which is transferred between objects independently. This mechanism can be regarded as serial and parallel interfaces between computers. Object-oriented programming corresponds to the serial interface while Firstsight corresponds to the parallel interface.

7.2 Comparison with data flow programming

The execution of a data flow program proceeds as the data changes, while information indicating these changes is transferred between components. Usually, this information transfer is not significant because the user does not notice it.

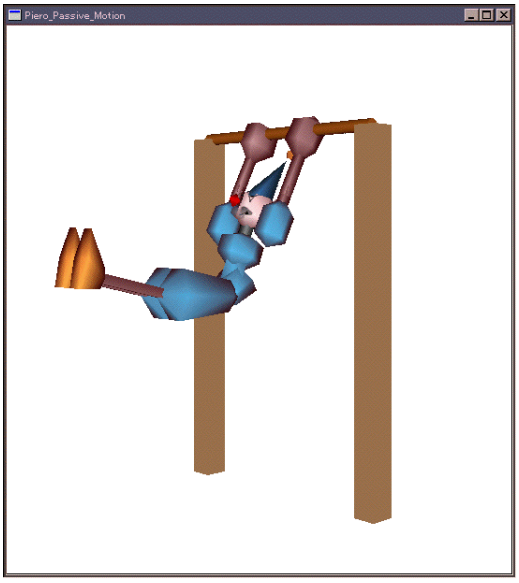
Firstsight performs this information transfer by connecting instruction buses, so that the user can control the transfer. If instruction bus connections are always automatically performed together with data connection, the result is equivalent to data flow programming. The most significant difference is whether the user can control the information transfer.

8. Application examples

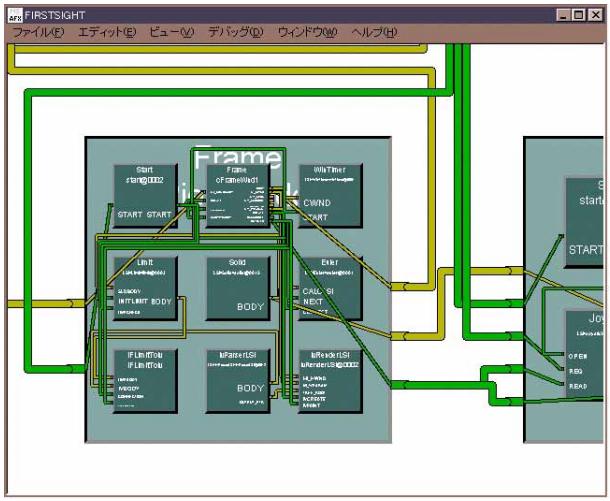
Figure 16 show the application developed by Firstsight and its circuit. This circuit consists of many LSIs and work together using high speed data exchange. Usually, the application can be easily modified by connecting several buses.

9. Conclusion

Because the CG and simulation application has a complex data structures, it was difficult to reuse the software component inside such applications. And these applications require high speed execution. We propose a new method to reuse soft-



a) Screen Image



b) Firstsight LSI circuit

Fig.16– The application developed by Firstsight.

ware component for such CG and simulation applications. The basic concept is software LSI. The new method offers high efficiency in both the development and execution of software. We constructed a program development environment based on this technology and named it Firstsight.

We are now developing some easier development tools for end users. For example, we are developing a program launcher on which end users can develop a CG and simulation application easily.

References

1) Kamada, H. et al. : Time-realistic 3D CG simulator 'Sight'. : SPIE 2409, pp. 255-266 (February 1995).
2) Nagashima, F. and Nakamura, Y. : Efficient Computation scheme for the Kinematics and Inverse Dynamics of a Satellite-Based Manipulator. Proc. of the IEEE Int. Conf. on Robotics and Automation, May 1992, pp. 905-913.
3) Bethel, W. : Modular Virtual Reality Visualization Tools. LBL Report Number 36693, UC 405.
4) Arnold, K. and Gosling, J. : The Java Programming Language. Addison- Wesley, 1996.
5) Brockshmidt, K. : Inside OLE, Microsoft, ISBN4-7561-3, 1996.
6) Goldberg, A. and Robson, D. : Smalltalk-80 The Language and its Implementation. Addison-Wesley Publishing Center, 1983.
7) Adams, D. : A Computation Model with Data Flow Sequencing. Stanford University, 1968.

Fumio Nagashima received the Dr. degree in mechanical engineering from Keio University, Tokyo, Japan, in 1989. He joined Fujitsu Laboratories Ltd., Kawasaki in 1989 and has been engaged in research and development of software simulation tools. He is a member of the Japan Society of Mechanical Engineers(JSME).

Tsugito Maruyama received the Dr. degree in electrical engineering from Tohoku University, Japan in 1970. He joined Fujitsu Laboratories Ltd., Kawasaki in 1989 and has been engaged in research and development of computer vision. He is a member of the Society of Instrument and Control Engineerings.

Kaori Suzuki received the Master degree in earth and planetary physics from Hokkaido University, Japan in 1986. She joined Fujitsu Laboratories Ltd., Kawasaki in 1986 and has been engaged in research and development of computer graphics.