SLAC–PUB–6135
April 1993
(M)

The Gismo Project*

William Atwood
Stanford Linear Accelerator Center, Stanford, CA 94309
Alan Breakstone
University of Hawaii, Honolulu, HI 96822
David Britton
McGill University, Montreal, PQ, H3A 2T8 Canada
Toby Burnett
University of Washington, Seattle, WA 98195
David Myers
CERN, CH-1211, Geneva, 23 Switzerland
Gary Word
Rutgers University, Piscataway, NJ 08855

Introduction

In the quest to understand the fundamental structure of the universe, high-energy particle physicists smash together subatomic particles at a handful of accelerator complexes around the world. The enormous energy and the grand size of these machines, such as the 27 km circumference Large Electron-Positron (LEP) collider in Geneva, or the 83 km circumference Superconducting Super Collider (SSC) under construction in Texas, contrasts with the tiny scales on which matter is being probed. These machines are designed to examine distances as small as 10^{-16} centimeters and to investigate particles that live for only a picosecond or less before decaying. These scales are far beyond the perception of the unaided senses so the particle physicist must rely heavily on computers to design, control, simulate, and interpret such experiments.

Submitted to C++ Report

* **Work supported in part by Department of Energy contracts DE–AC03–76SF00515 and DE-AC03-83ER-40103; Natural Sciences and Engineering research Council Canada; National Science Foundation; and National Science Foundation grant PHY-92-24 196.**

The detectors used to capture the signatures of the particles that fly out from these subatomic collisions are of colossal scale and complexity. For example, the particle detector ZEUS at the recently commissioned HERA electron-proton collider in Germany is the height of a four story building, weighs 3600 tons and records 257,000 channels of data up to several times per second. The trigger decision, the decision on whether to record an event for future analysis offline, requires a tremendous amount of computing power to assimilate the large quantity of data in a fraction of a second.

The ZEUS experiment has a three-level trigger, each level performing an increasingly sophisticated analysis culminating in a third-level trigger composed of a 1000-MIP array of Silicon Graphics processors. For experiments now being planned, data rates may exceed 10 Mbytes/second, and the amount of archival storage needed may exceed 50 Tbytes/year (yes, 5×10^{13} bytes/year).

These huge detectors are typically a composite of many different detection elements each of which returns different information about the particles such as their position, momentum, energy, and particle type. The process of associating and interpreting the raw data taken by the various detector elements and extracting the quantities which allow an understanding of elementary particle physics is referred to as reconstruction. This is a very complex process and relatively little of it can be done as the data are taken. Instead, the data from each collision (usually called an event) are written to some permanent storage medium, usually magnetic tape or disk, for later analysis. The data analysis thus begins with the event reconstruction in which the raw data from all detector elements are combined to make hypotheses as to what types of particles were produced, and to estimate their energies, momenta, and angular distributions. This analysis relies heavily on the results of simulations which predict how the various detector elements respond to different types of particles passing through them. Individual events may be studied, or else subsequent analysis may group together events of the same or similar type, and probabilities of that type of event occuring may be calculated. These probabilities, expressed in the form of cross sections (see the sidebar), are

then compared to theoretical predictions based on the present understanding of the physical processes involved in the collision. In this way, the basic physics is inferred from studies of colliding particles. Consequently, the simulation of each detector element and of the detector as a whole is not only crucial in optimising the detector performance at the design stage, but is equally important in the understanding of the underlying physics once the data has been recorded.

The simulation sequence starts with a computer generated collision between two particles which produces an event containing up to several thousand additional particles. This part of the simulation code is called an event generator and is independent of the properties of a specific detector. A typical event is shown in Fig. 1. As the particles produced in the initial collision are propagated outwards by the simulation program, they interact with the materials in the detector and eventually stop, annihilate, decay into other particles, or else escape completely from the detector. All these processes must be simulated, and in addition, the interactions of the particle with the various measuring devices must be calculated. The signals produced by the measuring devices in response to the interaction of the particles are what the physicist analyses.

At first glance, the simulation of this complex system of detectors may seem to be both difficult and of limited application, but in essence the problem is simply one of propagating particles though matter and simulating their fundamental interactions. This generalization leads to a wider application for the project described in this article than is apparent at first sight. Medical physicists use particles in treatments such as cancer therapy and in diagnostic work such as Positron Emission Tomography (PET) scans. In both of these applications, the results are interpreted through the simulation of particles traversing the material of the human tissue. Many other groups of physicists also need to simulate the tracking of particles through material, where the particles are either generated from accelerators or come from outer space as cosmic rays. Thus there are many groups that need tools to propagate particles through matter and to simulate the same basic processes.

The processes of detector simulation and event reconstruction lend themselves quite naturally to an object-oriented approach. There are many general properties of detectors, such as their geometrical shape and their response to particles, which can be abstracted. Likewise event reconstruction can be abstracted as a collection of reconstructed particles, each of which has certain general properties, such as a trajectory through space. In the remainder of the article we will discuss how an object-oriented approach is used in the Gismo project and comment on the strengths and weaknesses of C++ as an object-oriented language within this context.

History

The Gismo (Graphical Interface for Simulation and MonteCarlo with Objects) project began in the summer of 1990 as a prototype to test the applicability of object-oriented techniques to detector simulation and event reconstruction. It was written using the Objective-C language with a graphical user interface (GUI) designed using the NeXT Interface Builder. Several of us now engaged in the project were impressed by this effort and wished to expand the project so that it could be a more complete physics tool, usable by a larger segment of the high-energy physics community. Others saw Gismo as being applicable to a wide range of simulation tasks in more diverse fields. About twenty physicists and programmers from eleven institutions, from Hawaii to Geneva, Switzerland, formed a collaboration in 1991. We have held two workshops, one at the Stanford Linear Accelerator Center (SLAC) in July, 1991, and the other at the University of Florida in January, 1992. At these workshops we defined the goals of the project and discussed details of their implementation.

Gismo Goals and Decisions

Gismo is designed specifically to speed up the process of detector design, simulation, and event reconstruction. For the design process, it is of paramount importance that changes in the geometry of the detector may be made quickly and easily and in a manner which is not prone to introducing errors. For the

simulation of events it is important that Gismo be flexible enough that the user may choose the level of detail to which the simulation is performed. For example, in some studies it may be sufficient to parameterize the response of many detector elements in a way that allows the rapid simulation of many events. For other studies it will be important to simulate the interactions in great detail and pay the price in the required CPU time. The reconstruction of events by Gismo must also be implemented by a flexible scheme that allows the user to investigate the effects of different detector elements independently and to try out new methods of interpreting the information provided by the detectors. By fulfilling these needs, Gismo will serve both as a tool for designing new generations of high-energy physics detectors and as a tool with which present day detectors may be more easily understood. Thus, Gismo will have users who only deal with it interactively through the GUI or via batch processing; customizers who need to know more about Gismo "hooks" to allow them to customize the simulation for their particular detector; and finally developers who write the more abstract classes defining the program's overall architecture.

At the workshops, we chose Unix to define the generic environment in which Gismo would be developed. The project is organized into a platform independent kernel, with interfaces to detector and event I/O, a graphical display, and a GUI (see Fig. 2). The kernel and I/O should be suitable for batch applications. Both detector simulation and event reconstruction should be available in one program. Gismo will also allow an easy hookup to existing and future tools, such as event generation packages written by other high-energy physicists. Most of the existing tools are written in FORTRAN, while some of the tools to be used in the future are written in C++, for example the event generation program MC++ [Leif Lonnblad and Anders Nilsson. The MC++ Event Generator Toolkit–version 0. Computer Physics Communications, 71 (August 1992) 1.].

The source code for the kernel is further divided into a number of subprojects which separate different aspects of the simulation problem and which provide an easy way to assign tasks among the collaborators with minimal interference among

the widely-scattered programmers. The major subprojects are general tool classes: mathematics classes, geometry classes, classes for the propagation of particles, classes for the interactions of particles, classes related to the GUI and graphics, and miscellaneous support routines.

Most of the Gismo collaborators came from a FORTRAN background with little experience with other languages and essentially no previous experience with object-oriented programming. Those of us who participated in the prototype project did have experience with object-oriented programming using Objective-C, and one of us had additional experience with Eiffel. However, we all had a great deal of experience with large software projects typical in the simulation and analysis of high-energy physics experiments.

At the workshops there was a great deal of discussion on the choice of a programming language. Clearly we were only interested in an object-oriented language, yet we were very concerned by the lack of standards compared to FORTRAN. We felt it was unrealistic to continue to use Objective-C from the Gismo prototype since the language is not widely supported. C++, on the other hand, has compilers available on all the platforms of interest to the Gismo group, although some felt that the language appeared too complex and had a very awkward syntax. In addition, the lack of a current ISO or ANSI standard means that there are platform dependent differences which would have to be avoided. Since we had essentially no experience with C++, there was some trepidation in deciding to use this to rewrite Gismo, but we felt that there were no viable alternatives.

Having made the decision to use C++, we discussed the issue of base class libraries. Unfortunately, physicists are not accustomed to paying for software, so in order to gain wide acceptance in our field we could not base Gismo on a commercial product. We therefore investigated the use of the NIHCL and GNU C++ class libraries, but felt that both carried too much excess baggage in the form of unneeded classes and deep class hierarchies. We were also concerned that the lack of standards might mean that class libraries could change with future releases

such that we would have to rewrite major parts of our code in order to remain compatible. Finally, we worried that, again due partly to the lack of standards, there would be class name conflicts if we needed to include more than one base class library. Thus, since our needs were modest and the C++ language encourages a shallow class hierarchy, we opted to write our own very limited base class library and so reduce one of our portability concerns.

Experience with Object-Oriented Design and C++

We were quite concerned that it might take a long time to learn how to implement object-oriented design and to program efficiently and effectively in C++. In practice, it typically took a few months to learn enough C++ to write meaningful classes for the Gismo project. The major time-consuming step was, and remains, object-oriented design. We found that the best attitude to have when writing Gismo code was to consider it as prototype code which may need frequent major revisions as shortcomings are uncovered and, indeed, most of the subprojects have undergone several iterations. Now, about a year after this effort began, we are still rewriting large sections of code, but the process does seem to be converging. Undoubtedly the use of a design tool with skeleton code generation would help.

We decided to write the code in a very conservative fashion, trying to avoid using features that may not be universally available. For example, although C++ compilers which handle templates are widely available, we do not yet have such a compiler on one of our major development platforms (the NeXT). We have therefore chosen to not take advantage of this very nice feature of the language. We have compiled the Gismo kernel using the NeXT C/Objective-C/C++ compiler which is based on GNU version 1.37, the Borland version 3.1 compiler, and GNU's gcc 2.0 compiler on Sun, DEC and RS/6000 workstations. So far we have not encountered any major portability problems.

We have been very pleased with the debugging environment provided by our development platforms for use with C++. Our main experience is with the GNU

gdb debugger on the NeXT. We find this debugger is easy to use, allowing us quickly to find and fix many problems. Unfortunately, though, for "memory scribbling" problems, the inability to set debugger conditions to test for the change of one or more particular memory locations (i.e., to set "watch points"), as opposed to when a particular source code line is reached, is a major drawback. One of us tried PURIFY, from Pure Software, on part of the code, and considers it worthwhile to use such a product to test the integrity of the complete program.

Some of the non-object-oriented features of C++ have also been useful. First, it is strongly typed: misspelling the name of a variable or function is always caught by the compiler. The *const* attribute of structures provides a way to prevent modification of an object when it is not appropriate. Function name overloading and default parameters give convenient flexibility to function names, allowing the same name to be used with different numbers and types of arguments. We have used operator overloading to define operations on 3- and 4-vectors. We like being able to declare and initialize variables as they are needed.

However, there are some features we would like to see incorporated in C++ in the future. In particular, an exponentiation operator, such as exists in FORTRAN, would be very useful to us. Also we could use some means of handling exceptions, such as is envisioned for future versions of C++ and which has been described in some of the textbooks.

Coming from a FORTRAN world without encapsulation, inheritance, polymorphism, or dynamic binding, it took a while to utilize these features of C++ to our advantage. Now it would be difficult to go back to a language without them. The advantages that the textbooks advertise are manifest. We have enforced encapsulation by never making data members of Gismo classes public, even though C++ allows this. We use inheritance, polymorphism and dynamic binding extensively. For example, in the geometry subproject, we use dynamic binding in writing the code to calculate the intersection point of a straight line or a helix with any one of several shapes of surfaces without knowing until run time

exactly what kind of surface is used. Another example occurs in the propagation of particles from the collision point through the detector for which the trajectory may be either a straight line or a helix depending on whether the particle is electrically charged and whether there is a magnetic field present (see the sidebar on particle propagation for more details). Furthermore, the type of medium varies among the different detector elements. Dynamic binding allows us to write the code without the case statements which would be required in FORTRAN. This makes the code easier to write (and to understand later) than has been the case with FORTRAN. Dynamic memory allocation in the language does much of the tedious bookkeeping for the programmer, so that he or she can concentrate on more fundamental issues. This greatly speeds up the debugging process. Unfortunately, lack of garbage collection still leaves plenty of room for memory leaks.

Gismo Project Status and Future Plans

So far, most of our development of the GUI and graphics has been on NeXT workstations, where Display PostScript has been used for drawing, although the graphics package has also been ported to an X-window environment. We have written code in the Gismo kernel which interfaces between geometrical objects and the drawing package. The GUI is written using the NeXTstep Interface Builder, which we find to be a very powerful tool. The GUI allows a user easily to input detector components, define materials, initiate particle propagation, and display the results. A similar effort is planned for the X-window environment.

Another part of the project is to write an I/O package to support object persistence. This is particularly important to allow the user to store and retrieve those objects created interactively with Gismo, such as detector descriptions and the individual simulated events, including reconstructed quantities. It might be possible to use an object-oriented database system to do this, but the functionality we require is far too modest to justify the high cost of such systems. Also, as noted earlier, we do not want to tie the Gismo kernel to any commercial package.

For the more distant future, we are contemplating connecting Gismo to a CAD package to allow better and more detailed communication between the engineers and physicists in an experiment. For example, one could input a detailed detector design from a CAD system and simulate its response to the processes of interest to a physicist. In addition, we want to incorporate various event generators into Gismo which will allow Gismo to simulate a wider variety of physics processes. Finally, we plan to connect Gismo event output to various analysis tools that others in our field are developing to allow a seamless transition from detector design, through event generation, detector simulation and event reconstruction, to detailed physics analysis, and even publication!

Conclusion

In summary, the Gismo project is an ambitious effort to modernize the process of high-energy physics detector design, simulation, and event reconstruction using the increased computational power available with Unix workstations. We hope that Gismo will gain wide acceptance both in the field of high-energy physics as well as in other disciplines of science. This effort capitalizes on the advantages of object-oriented programming using the C++ language. Undoubtedly C++ successfully achieves what Bjarne Stroustrup set out to do: namely to incorporate support for object-oriented programming into C. We have found the language has strengths, such as being widely available and C compatible, but it also has weaknesses. The major problem is the complexity of the language which makes it difficult to learn; witness the text books with 600 pages. Currently, there is a lack of standards for both the language and for the basic class libraries. We look forward to improvements in the language and particularly to standardization and the inclusion of exception handling.

Sidebar on High Energy Physics

Particle physicists study the most basic constituents of matter and their fundamental interactions by colliding beams of particles with other beams of particles or with stationary targets. Knowledge about the fundamental

constituents of our universe is extracted by studying how often various types of collisions occur, what particles are produced in these collisions, and what subsequently happens to the particles created. Typically the particles produced in the initial collision decay rapidly into other particles which in turn may also decay until eventually only long-lived particles remain. The physicist observes these final state particles in massive detectors, and by measuring their energies, angles, and momenta, tries to piece together the complex series of decays.

There are known to be four basic interactions among elementary particles: the strong interaction which is responsible for the binding of protons and neutrons in nuclei; the electromagnetic interaction which keeps electrons bound to nuclei in atoms; the weak interaction which governs most radioactive decays; and the gravitational interaction which is so weak in comparison to the other three that its effects cannot be seen in particle collisions. The division between these different interactions is somewhat arbitrary in that they may be just different aspects of a single unified interaction. For example, the electromagnetic and weak interactions are now known to be just different aspects of a single electroweak interaction.

Particles which make up matter are divided into two general classes based on these interactions: leptons, such as the electron, which do not have strong interactions; and hadrons, such as the proton and neutron, which do. Hadrons are believed to be composed of more elementary constituents called quarks. Hundreds of different kinds of hadrons, made up of different combinations and types of quarks, have been observed. There is clear experimental evidence for five quark types, and it is strongly believed that a sixth will soon be found. Similarly, five types of leptons have been observed, and there is strong evidence for the sixth. The interactions between constituent particles are explained by the exchange of other particles called gauge bosons. For example, the electromagnetic interaction, giving rise to light, is due to an exchange of photons (in common terms, particles of light) which are a type of gauge boson. Similarly, the neutral Z and charged W gauge bosons are responsible for the weak interaction, and yet other gauge bosons called gluons produce the strong interaction.

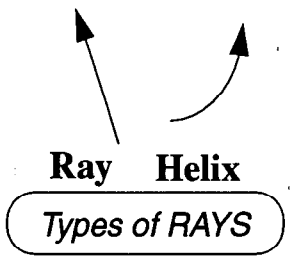
Knowledge about the different interactions and constituents of matter is gained by studying particle collisions. The probability that a reaction will end up with a particular final state of particles from an initial state defined by the particle beams (or beam and target material) is typically measured. These initial and final states are characterized by the types of particles, their energies and trajectories, and perhaps other properties, such as polarization. The reaction probabilities are expressed as effective cross sectional areas (cross sections) of the particle collisions. The sizes of these cross sections are tiny compared to everyday scales. They range from typically 10^{-42} cm^2, for the weak interaction, up to 10^{-25} cm^2 for the strong interaction. These small cross sections set the parameter scales necessary for the design and construction of particle accelerators so that useful event rates may be observed. They also set the scale for the size of the detectors that are built to do the experiments. The more massive the particle, the higher the energy needed to create it, and the bigger the accelerators and the detectors must be.

How Particles Propagate

The propagation of particles through 3-dimensional geometries is illustrated by the following C++ code and pictures. Polymorphism is used to allow generalized messaging of both different types of Rays as well as different types of geometrical objects. The recursive nature of C is used to follow, while keeping track of the location of all daughter (grand daughter, great grand ..., etc.) particles resulting from interactions or decays.

```
void Particle::propagate(const Medium* stuff)
{
    const Medium *newStuff = stuff;
while (status==STATUS_ALIVE)
{
   stuff = newStuff;
// return a Ray that estimates the trajectory
   Ray *segment = trajectory(stuff);
```

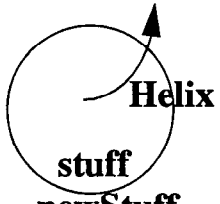
Private member function of Particle
Choose which type of Ray.



Types of RAYS

```
// Ask the Medium for a distance along ray
   step = stuff->distanceToLeave
                  (*segment, detector, newStuff);
```

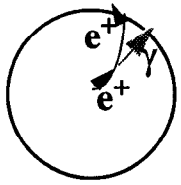
Returned arguments



Geometry Only!

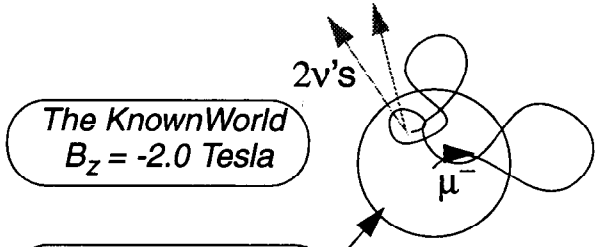
```
// Perform the step, taking all physics into account
      stepBy(step, *segment, stuff);
   if (detector != 0) detector->score(*this);
```

Detector records energy loss, trajectory information, etc.



STATUS = INTERACTED

```
// Add (Ray *)segment to track (List of Rays)
      track->addSegment(segment);
}
```

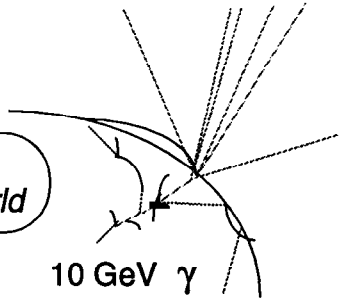


The KnownWorld
B_z = -2.0 Tesla

A Tube of Aluminum
B_z = +2.0 Tesla

```
// Now recurse: make sure that all of the children
// of a particle's decays or interactions propagate
// out of the medium in which they were created
if (status==STATUS_DECAYED||status==STATUS_INTERACTED)
{    for(int i=0; i<numChildren(); i++)
        child(i)->propagate(stuff);
}
```

A magnified view of the above KnownWorld



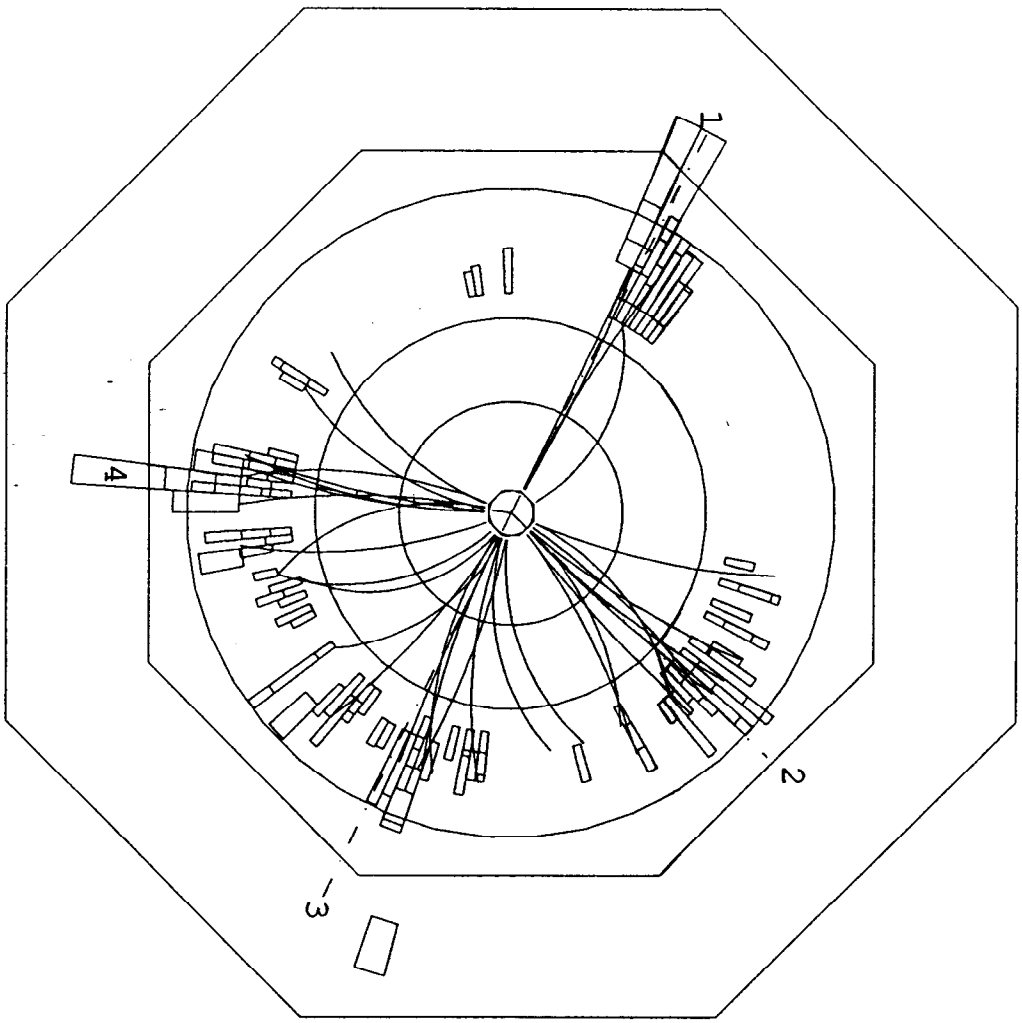


Figure 1. An event display, looking along the beam direction, of a neutral Z particle decaying into four "jets" of particles using the SLD detector at the Stanford Linear Collider. The curved lines are trajectories of charged particles reconstructed from the measurements in the inner cylindrical detectors, while the trapezoidally shaped boxes represent energy deposited in the outer detectors.

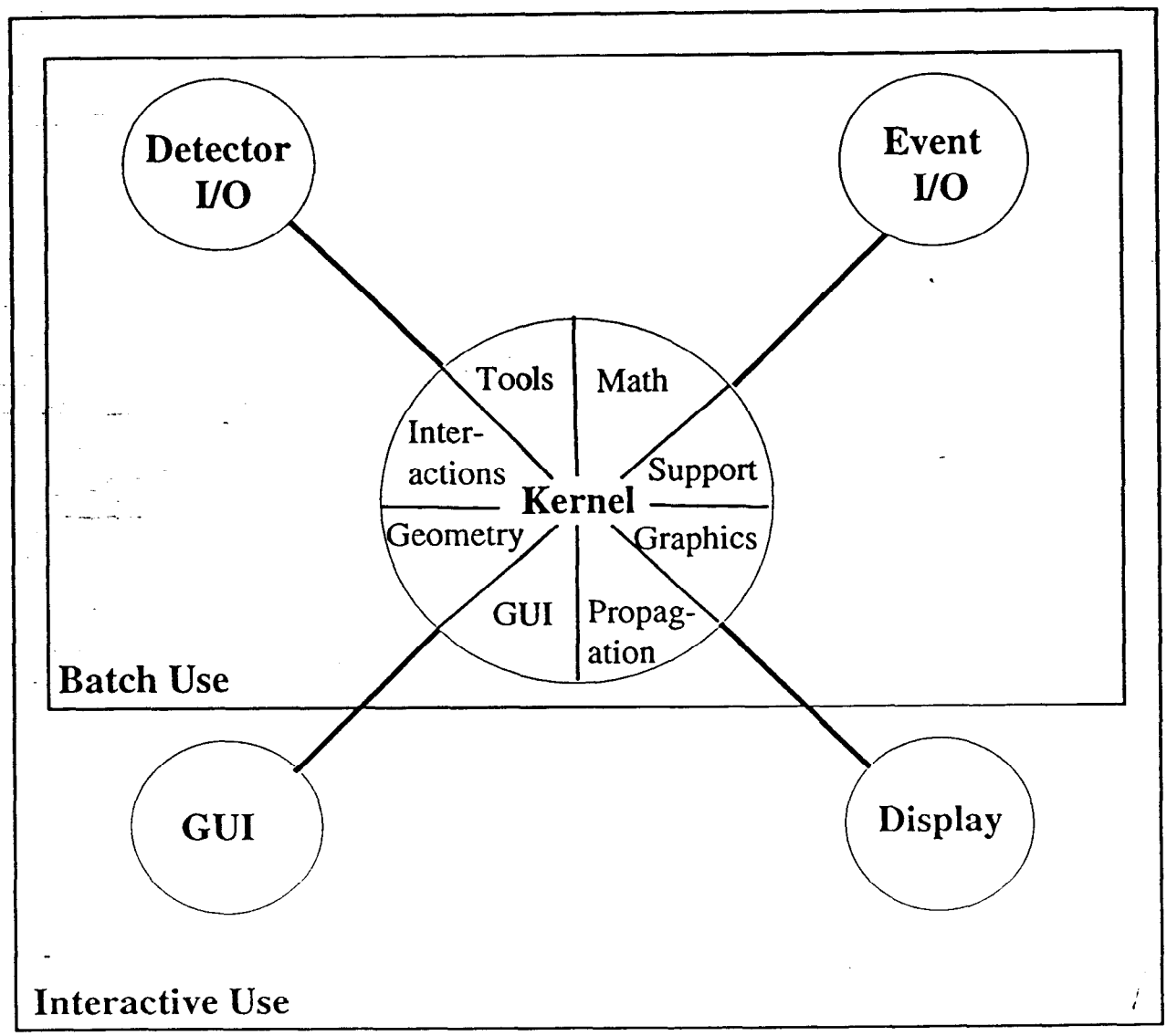


Figure 2. Illustration of the relationship of the Gismo kernel to detector and event I/O, the graphical display, and the graphical user interface.