Gambit: A Prototyping Approach to Video Game Design

Tracy Larrabee and Chad Leland Mitchell
Stanford University

Video game designers don't have to be assembly language programmers. Their creative talents can be expressed in a prototyping language.

Designing video games is an extremely complex process. Like many designs involving computers, video games require sophisticated algorithms and data structures and must run on real-world hardware that is limited in capability and demanding of programmers skills. However, because video games depend on complex user interaction to function correctly, programs that merely work are not good enough: video games must also be intriguing and entertaining.

Background

In the fall of 1982, we began work on a programming language to address problems encountered in user interaction, graphical representation, simulation, and real-time control. Video game design seemed an interesting application that would benefit from a coherent solution to each of these problems. Our interest in effective game design aids peaked when *IEEE Spectrum*, in its December 1982 cover story,[1] identified game design as possibly the most difficult problem in the video game industry.

To us, it seemed inappropriate that video game designers, whose job it is to create dazzling visual effects and intriguing action, should be concerned with low-level technical details such as graphics display frame buffers, color registers, instruction sets, and instruction timing. When a designer first conceptualizes a game, he should be able to experiment with his idea—to prototype it and test it—before committing to the lengthy, difficult, and costly implementation process. Thus, we decided to create a special-purpose language for prototyping video games to see if we could in fact make the designer's work easier and, therefore, more creative.

The result of our efforts, called Gambit, is a high-level, machine-independent language and an accompanying runtime environment. Gambit provides a simple but elegant solution to the problem of video game design while addressing the larger issues of interactive graphical simulation and real-time control. The simple notions of type, type extension, asynchronous control, and object interactions allow a Gambit programmer to express a complex game design without focusing energy on implementation details peripheral to the central idea of the game.

We and our colleagues, Kim McCall and Benjamin Pierce, are still refining the language specification, but the basic language structure was solidified by the end of 1982.[2] During the first half of 1983, McCall implemented a compiler for a significant subset of Gambit, which ran in the Smalltalk programming environment at Xerox PARC. We now have a partial runtime environment in which several games have been written and run successfully. (An example of a running Gambit game appears on pp. 32-34.)

Approach

Perhaps the most important decision made early in the design of Gambit was that the language and its execution environment should closely parallel the player's perception of the video game universe. This meant that the designer's model of a game in Gambit need not differ greatly from the player's model, and it required that certain notions be built into the

0740-7459/84/0010/0028$01.00

language. For example, the language models independent objects in the game universe and provides a flexible means for generating their graphical representations on the screen. A notion of location or space, including position both on the screen and relative to other objects, allows for the fact that objects occupy space and may collide or interact with each other. There is also a uniform notion of time within the game universe, yet the system has a mechanism for handling asynchronous events such as user input.

Objects, classes, messages, and control

A game in Gambit is different from games in most other languages. In most languages, the programmer thinks of the objects on the screen as part of the game's global state, possibly maintained as entries in a large table, with a game controller that updates the table and performs any special processing. In Gambit, however, programs are constructed with an object-oriented processing model. The objects on the screen are not described as tables, but as objects in their own right with both local state and intelligence to manage that state and interact with other objects.

Most object-oriented languages define a fairly rigid chain of control. One object controls the game and sends messages to seconds-in-command, who send messages to their subordinates, and so on. In Gambit, this level of control is hidden, both in the language and in the runtime support system. All programmer-defined objects are essentially equal. They interact with the system, receive alarms and input events, display themselves on the screen, and possibly interact with other objects. No one object is necessarily in charge of the game; rather, each object is individually responsible for a small part of the game.

All objects in Gambit are instances of user-defined classes that are described in a class definition module. Each class includes a specification of the local state maintained by objects of that class and of the messages that objects of that class are willing to accept. For each message that can be received from another object or from the system, a message handler specifying the operations to be performed upon receiving that message is included in the class definition module. Global variables, along with global constants and types, are declared in a globals definition module, that can be imported by any class definition module and by any other globals definition module, provided circularity is not introduced. (See Figure 1.)

At runtime, instantiated objects interact with system facilities (such as the display manager) by calling primitive system procedures and functions

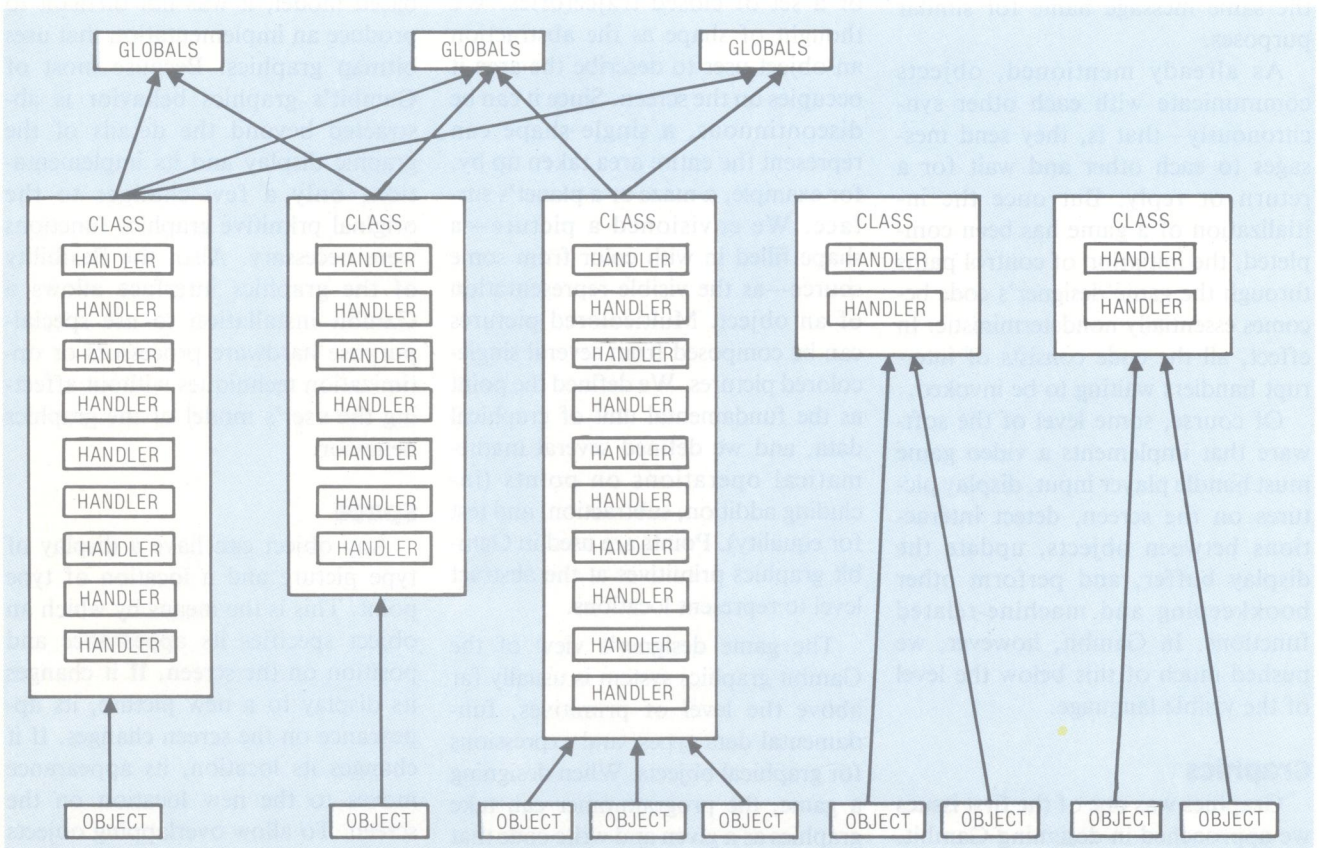


Figure 1. The structure of a simple game in Gambit.

and by receiving messages from the system. They interact with other objects by sending and receiving messages. We have found that the object-oriented programming style as exemplified by Simula,[3] Smalltalk,[4] and Clu[5] is a natural way to think about a running video game. The message mechanism is used uniformly in providing the other modeling mechanisms included in Gambit.

A message in Gambit is not a communication between concurrently executing parallel processes. Sending a Gambit message is very similar to calling a Pascal procedure; the basic difference is in the manner of determining exactly what code gets executed. In Pascal, a procedure name uniquely specifies the code to be executed on calling that procedure; in Gambit, a message name alone will not do that. To determine which code will be executed, it is necessary to know both the message name and the type of the object receiving the message. This permits reusing message names, so several classes can use the same message name for similar purposes.

As already mentioned, objects communicate with each other synchronously—that is, they send messages to each other and wait for a return or reply. But once the initialization of a game has been completed, the initiation of control paths through the game designer's code becomes essentially nondeterministic. In effect, all the code consists of interrupt handlers waiting to be invoked.

Of course, some level of the software that implements a video game must handle player input, display pictures on the screen, detect interactions between objects, update the display buffer, and perform other bookkeeping and machine-related functions. In Gambit, however, we pushed much of this below the level of the visible language.

Graphics

Graphics was one of the first issues we approached in designing Gambit. At first, it seemed important for Gambit to define an elegant and powerful way of describing graphic entities on the display screen. But after reviewing several types of graphics systems, we realized that no single graphics system would be adequate for prototyping all the game designs we might want to support. If a prototype was to have the same appearance as the final product, they needed similar graphics systems. (Obviously, the visual effects of the same game on a raster display and on a vector display differ significantly.) The solution was to interface Gambit for a variety of graphics packages. Only those types and operations directly involving graphics need be changed to use a different package, and as far as we can see, any graphics package with the general characteristics of the one we chose (a polygon and spline-based package described by Warnock and Wyatt[6]) would work as well.

Originally, we envisioned the fundamental graphics data types to be shape, picture, source, point, and trajectory.[6] A trajectory is a line or curve. A shape is a closed trajectory or a set of closed trajectories. We thought of shape as the abstraction an object uses to describe the area it occupies on the screen. Since it can be discontinuous, a single shape can represent the entire area taken up by, for example, a maze or a planet's surface. We envisioned a picture—a shape filled in with color from some source—as the visible representation of an object. Multicolored pictures can be composed from several single-colored pictures. We defined the point as the fundamental unit of graphical data, and we defined several mathematical operations on points (including addition, subtraction, and test for equality). Points are used in Gambit graphics primitives at the abstract level to represent locations.

The game designer's view of the Gambit graphics system is usually far above the level of primitives, fundamental data types, and expressions for graphical objects. When designing a game, the programmmer can take graphics as a given and write code that assumes graphical objects are provided somewhere. To display objects and move them about the screen, the programmer uses messages such as "Put me on the screen at this location," "Rotate me by this many degrees," "Change my color to green," or "Move me this far in that direction." The actual pictures are generated by a programming tool called the graphics preprocessor. When using the graphics preprocessor, the designer draws and edits pictures on the screen and specifies some of the object's behavior. The preprocessor then generates the Gambit source code to build these pictures and inserts them into the file system where the programmer can simply refer to them by name. The programmer need not worry about constructing these entities in Gambit source code; he can simply draw them on the screen, name them, and reference them as global constants. The experimental Gambit environment uses the standard Smalltalk-80 graphics editor for this purpose.

Even though we originally defined Gambit to use the polygon-spline-based model, it was not difficult to produce an implementation that uses bitmap graphics. Because most of Gambit's graphics behavior is abstracted beyond the details of the graphic display and its implementation, only a few changes to the original primitive graphics functions were necessary. Also, the flexibility of the graphics interface allows a Gambit installation to use special-purpose hardware processors or optimization techniques without affecting the user's model of the graphics behavior.

Space

Any object can have a display of type picture and a location of type point. This is the means by which an object specifies its appearance and position on the screen. If it changes its display to a new picture, its appearance on the screen changes. If it changes its location, its appearance moves to the new location on the screen. To allow overlapping objects to block out things "behind" them, we included a system primitive that

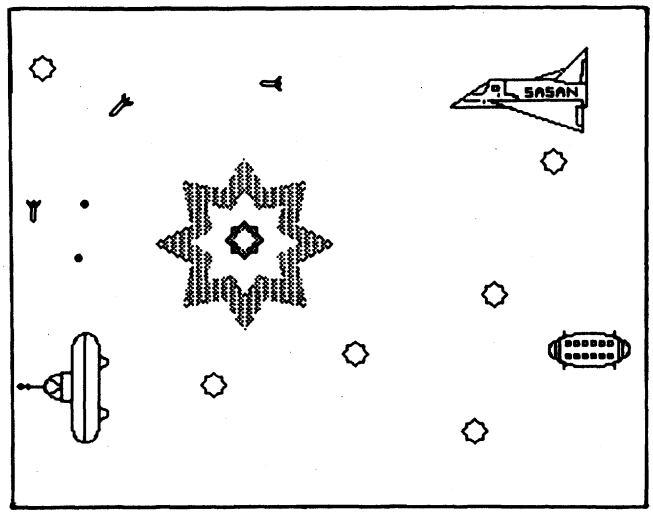


Figure 2. Game display on screen.

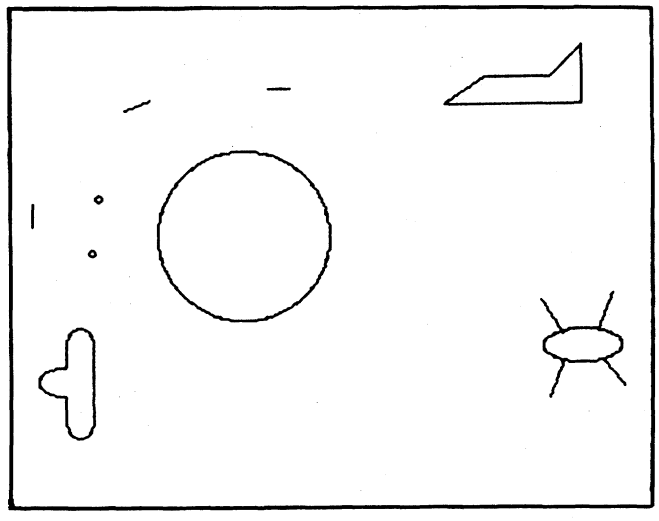


Figure 3. Game interaction boundaries.

sets object layers so that lower numbered layers block the displays of higher numbered layers. A Gambit implementation could use a graphics package that includes full three-dimensional functionality so that type point would have three coordinates. But a two-dimensional graphics package plus layers is adequate for most games and, for performance reasons, is the choice on today's hardware.

An object can also have an interaction boundary of type shape. This boundary does not appear on the screen, but it does move around with the display as the object changes location. (See Figures 2 and 3.) It may or may not have the same shape as the picture assigned to the object's display, and its shape may be complicated or even disjoint. When the boundaries of two objects overlap, the system detects this as an interaction. Each of the interacting objects may be sent a message that will include, as a parameter, a reference to the other object.

Not all objects will care about interactions with all other objects. Often, only one of the objects will care about an interaction. As an example, if something hits an indestructible wall, the wall will probably not need to be notified. We have, therefore, included system primitives that objects can use to indicate which interactions they are interested in and which messages should be sent when those interactions are detected. To add determinacy, an object can also indicate the priority of its interest in the interaction. Objects indicating a higher priority interest will be notified first.

Time

In Gambit, time is represented by a very simple mechanism. The game universe as modeled has a master clock that ticks or cycles at preset intervals (for example, 60 times per second). By calling an alarm-setting primitive, an object can sign up to receive a message at some time in the future—that is, after a specified number of ticks. These primitives employ only a relative notion of time; Gambit has no notion of absolute time. As a convenience to the programmmer, both repeating and one-shot alarms are provided, each set by a different primitive. There are also primitives to disable alarms and primitives and to set alarm priority. The higher priority alarms go off first in a given tick.

This mechanism—partitioning time into minimal discrete units, combined with a background process for counting those units and sending messages after a certain number of units have gone by—supports our games remarkably well and seems to be as powerful as it is simple. Since this is Gambit's only mechanism for representing the passage of time, the most natural way for an object to move—that is change its location as a function of time—is to set a repeating alarm for itself and respond to that alarm's associated message by changing its location on the screen. We considered including a velocity vector, along with display, location and boundary, as part of an object's system-defined state. Since the movement implied by a velocity vector can be programmed explicitly, we did not include it in the initial version of Gambit. We continue to consider it a candidate for later enhancements, along with such things as rotational velocity and inertia.

Events

The Gambit implementation represents an asynchronous event in a game—pushing a button, for example—by a constant of the form *EVENT*n where "n" is a positive integer (not limited to a single digit). The instances of these events, taken collectively, specify all possible values of the scalar type *EventName*. Any object can use a system primitive to indicate interest in an event. As long as its interest continues, the object will be sent a message during the first tick after the event occurs. Priorities can be attached to event notification requests.

Gambit: a simple example

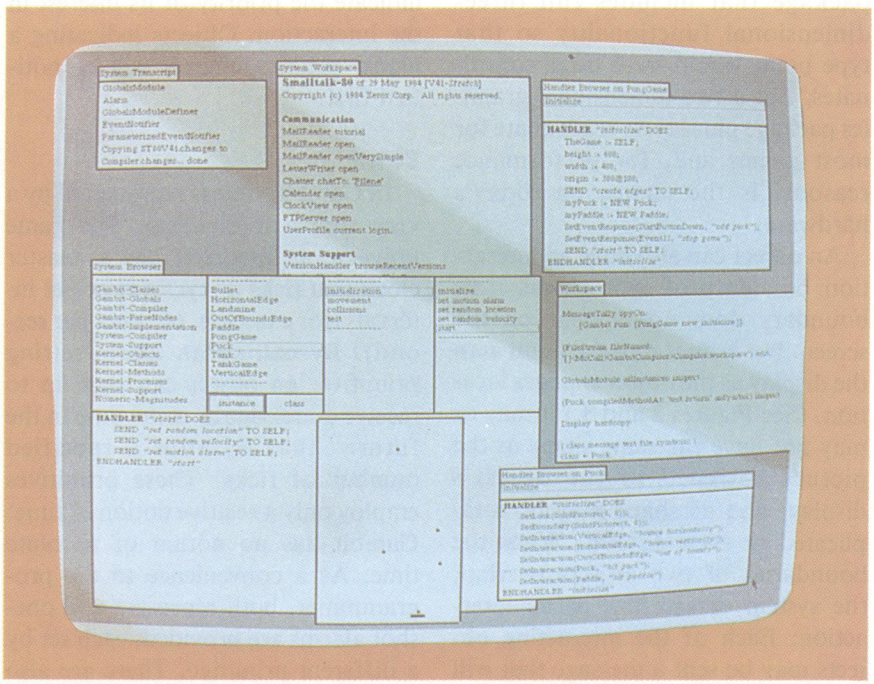
The game programmed below is similar to the first video game, Pong. the player manipulates a paddle to hit a puck that bounces off the playing court walls defined on the screen. the game contains three globals definition modules and five class modules, all of which appear below. Further understanding of this simple game can be gleaned from reading the code. Although not all the primitives have been explained, they should be clear from the context and accompanying comments (comments start with double dashes in Gambit and continue to the end-of-line).

```
GLOBALS PingConstants
  --Global constants
      CONST
            StartXVelocityRatio = 8.0;
            StartYVelocity = 4.0;
            PaddleMovement Distance = 0.5;
      ENDGLOBALS PingConstants

GLOBALS PingEventNames
  --Event definitions
      CONST
            UserMovedLeft = Event3;
            UserMovedRight = Event1;
            StartButtonDown = Event11;
            StopButtonDown = Event15;
      ENDGLOBALS PingEventNames

GLOBALS PingVariables
  --The only global variable in the game.
      VAR
```

Top: Display of the example game.

Above: Kim McCall, one of the original developers of Gambit, did a prototype of the system on a Dorado computer under the Smalltalk environment at Xerox PARC, where he worked part time in 1983. Currently a PhD candidate at Stanford University, McCall now works full time at Xerox PARC and is continuing some of the research on Gambit.

Right: A view of the interactive Gambit programming environment running on top of the Xerox Smalltalk-80 system. The "system browser" (lower left) allows display/edit/compile access to all the code and class definitions in the system. The user can also open individual "handler browsers" for similar access to multiple handlers. The handlers displayed here show most of the initialization and start-up code of the simple example game. Other windows allow textual feedback, store useful top-level commands for later interactive execution, and display the game in progress.



Gambit allows constant definitions such as *FireButtonPushed = EVENT27*. A programmer could place all such definitions in a globals definition module and use the more descriptive names for the events everywhere else. Moving a game to a different implementation with different events would simply require changing the event name mapping as needed. Gambit also allows hypothetical or not yet implemented events to be modeled by available events.

An implementation could easily handle events with associated data, such as the movement vector of a track ball or other input device. For example, a globals definition module could define global variables that reflect the device's current position. This module would then be included in the standard library for that Gambit implementation.

Syntax

We decided that Gambit's syntax and semantics should, where possible, be similar to languages with which most programmers are already familiar. We chose Pascal as a representa-

```
        TheGame : PingGame
    ENDGLOBALS PingVariables

CLASS PingGame
  --This class takes care of the initialization of Ping game.
    SEES PingConstants, PingEventNames, PingVariables
    USES Paddle, Puck, Horizontal Edge, Vertical Edge
    VAR
        myPaddle: Paddle;
        myPuck: Puck;
        leftEdge, rightEdge: VerticalEdge;
        topEdge, bottomEdge; HorizontalEdge;
        height, width; Integer;
        origin: Point
    HANDLER "initialize" DOES
      --Intialize the game
        TheGame := SELF;
        height := 600;
        width := 400;
        origin := 300 @ 100;
        SEND "create edges" TO SELF;
        myPuck := NEW Puck;
        myPaddle := NEW Paddle;
        SetEventResponse(StopButtonDown, "stop
          game");
        SEND "start" TO SELF;
        ENDHANDLER "initialize"
    HANDLER "create edges" DOES
      --Set up the playing area
        leftEdge := NEW VerticalEdge WITH
          (4, height);
        rightEdge := NEW VerticalEdge WITH
          (4, height);
        topEdge := NEW HorizontalEdge WITH
          (width, 4);
        bottomEdge := NEW Horizontal Edge WITH
          (width, 4);
        SEND "move to" TO leftEdge WITH (origin);
        SEND "move to" TO topEdge WITH (origin);
        SEND "move to" TO rightEdge WITH (origin
          + (width @ 0));
        SEND "move to" TO bottomEdge WITH (origin
          + (0 @ height));
        ENDHANDLER "create edges"

    HANDLER "start" DOES
      --Start the game
        SEND "start" TO myPaddle;
        SEND "start" TO myPuck;
        ENDHANDLER "start"

    HANDLER "starting paddle location" RETURNS Point
      DOES
      --Compute and return the initial paddle location
        RETURN origin + (width / 2 @ height - 24);
        ENDHANDLER "starting paddle location"

    HANDLER "stop game" DOES
        EndGame
        ENDHANDLER "stop game"

    HANDLER "height" RETURNS Integer DOES
        RETURN height
        ENDHANDLER "height"

    HANDLER "origin" RETURNS Point DOES
        RETURN origin
        ENDHANDLER "origin"

    HANDLER "width" RETURNS Integer DOES
        RETURN width
        ENDHANDLER "width"

    HANDLER "add puck" DOES
        SEND "start" TO NEW Puck
        ENDHANDLER "add puck"

    ENDCLASS PingGame

CLASS Paddle
  --variable and behavioral definitions for any player paddles
        SEES PingConstants, PingEventNames,
          PingVariables
        USES VerticalEdge.
        VAR oldLocation: Point

    HANDLER "initialize" DOES
      --set boundary and picture interaction
        SetLook(SolidPicture(40,8));
        SetBoundary(SolidPicture(40, 8));
        SetEventResponse(UserMovedLeft,
          "move left");
        SetEventResponse(UserMovedRight,
          "move right");
        SetInteraction(VerticalEdge, "hit edge");
        ENDHANDLER "initialize"

    HANDLER "start" DOES
      --Put the paddle at the starting location
        MoveMeTo(ASK "starting paddle location" OF
          TheGame);
        ENDHANDLER "start"

    HANDLER "hit edge" WITH (anEdge: VerticalEdge)
      DOES
      --When a vertical edge of the playing area is hit
        MoveMeTo(oldLocation);
        ENDHANDLER "hit edge"

    HANDLER "move left" WITH (dist: Integer)
      --move the paddle to the left
        DOES
        oldLocation := GetLocation;
        MoveMeBy(0.0 - PaddleMovementDistance *
          dist @ 0);
        ENDHANDLER "move left"

    HANDLER "move right" WITH (dist: Integer)
      --move the paddle to the right
        DOES
        oldLocation := GetLocation;
```

tive language. Where possible, we used Pascal syntax[7] for constructs that have the same semantics as Pascal; where our semantics differ from Pascal, we used a different syntax. Notable differences between Pascal and Gambit include user-defined types and inter-object communication. Several of the non-Pascal concepts were borrowed from Smalltalk[4] and Clu.[5]

Gambit is a strongly typed language. It contains types defined by the language, and a programmer or an installation can define new ones to extend the type system. While our experimental Gambit compiler does not fully implement type safety, a complete Gambit environment would benefit, both in terms of efficiency and programmer peace of mind, from the type safety specified in the original design.

Many of the language-defined types are identical to those in Pascal. We adopted the Integer, Real, Boolean, and Char types, as well as enumerated and subrange types, without modification. Gambit does not have sets, records, pointers, or file variables, nor does it presently provide any kind of

```
            MoveMeBy(PaddleMovementDistance *
              dist @ 0);
            ENDHANDLER "move right"

      HANDLER "location" RETURNS Point DOES
        --what is the current location of the paddle?
            RETURN GetLocation
            ENDHANDLER "location"

      ENDCLASS Paddle

CLASS Puck
  --variable and behavioral definitions for pucks
      SEES PingConstants, PingVariables
      USES Paddle, HorizontalEdge, VerticalEdge
      VAR location, velocity: Point;

      HANDLER "initialize" DOES
        --set boundary and picture interaction
            SetLook(SolidPicture(8, 8));
            SetBoundary(SolidPicture)8, 8));
            SetInteraction(VerticalEdge,
              "bounce horizontally");
            SetInteraction(HorizontalEdge,
              "bounce vertically");
            SetInteraction (Paddle, "hit paddle")
            ENDHANDLER "initialize"

      HANDLER "set motion alarm" DOES
        --arrange to have the puck move every tick
            SetRepeating Alarm(1, "move");
            ENDHANDLER "set motion alarm"

      HANDLER "set random location" DOES
        --move myself to a random location
            location := ((ASK "width" OF TheGame) *
              Random @ (Ask "height" OF TheGame) / 4)
              + (ASK "origin" OF TheGame);
            MoveMeTo(location);
            ENDHANDLER "set random location"

      HANDLER "set random velocity" DOES
        --set myself to a semi-random velocity
            velocity := (Random - 0.5) *
              StartXVelocityRatio @ StartYVelocity
            ENDHANDLER "set random velocity"

      HANDLER "start" DOES
        --put myself in the game
            SEND "set random location" TO SELF;
            SEND "set random velocity" TO SELF;
            SEND "set motion alarm" TO SELF:
            ENDHANDLER "start"

      HANDLER "move" DOES
        --move myself (according to my state)
            MoveMeBy(velocity);
            ENDHANDLER "move"

      HANDLER "new velocity" WITH (aPoint: Point) DOES
        --set my velocity
            velocity := aPoint;
            ENDHANDLER "new velocity"

      handler "bounce horizontally" WITH (anEdge: Vertical
        Edge)
            DOES
        --I've hit something vertical, now bounce
            SEND "new velocity" TO SELF WITH
              (0 - XOf(velocity) @ YOf(velocity));
            ENDHANDLER "bounce horizontally"

      HANDLER "bounce vertically" WITH (anEdge:
        HorizontalEdge)
            DOES
        --I've hit something horizontal, now bounce
            SEND "new velocity" TO SELF WITH
              (XOf(velocity) @ 0 - YOf(velocity));
            ENDHANDLER "bounce vertically"

      HANDLER "hit paddle" WITH (aPaddle: Paddle)
            DOES
        --I've hit the paddle, now bounce
            SEND "new velocity" TO SELF WITH
              (XOf(velocity) @ 0 - YOf(velocity));
            ENDHANDLER "hit paddle"

      ENDCLASS Puck

CLASS HorizontalEdge
  --the horizontal boundaries of the playing area

      HANDLER "initialize" WITH (width, height: Integer)
            DOES
        --set boundary and picture interaction
        SetLook(SolidPicture(width, height));
        SetBoundary(SolidPicture(width, height));
        ENDHANDLER "initialize"

      HANDLER "move to" WITH (aPoint: Point) DOES
        --move me to a new location
            MoveMeTo(aPoint);
            ENDHANDLER "move to"

      ENDCLASS HorizontalEdge

CLASS VerticalEdge
  --the Vertical boundaries of the playing area

      HANDLER "initialize" WITH (width, height: Integer)
        DOES
        --set boundary and picture interaction
            SetLook(SolidPicture(width, height));
            SetBoundary(SolidPicture(width, height));
            ENDHANDLER "initialize"

      HANDLER "move to" WITH (aPoint: Point) DOEs
        --move me to a new location
            MoveMe To(aPoint);
            ENDHANDLER "move to"

      ENDCLASS VerticalEdge
```

I/O other than player actions and screen displays. Language-defined types not present in Pascal include those that support the graphics interface (such as point) and inter-object communications (such as message).

Fundamental cycle

At each tick, the underlying system goes through a fixed cycle of activities. These background activities can initiate an interrupt-like message to an object, thereby beginning a chain of control through the game designer's code. Upon receiving the message, the object can send more messages, and arbitrary computation can result. Only when the initial handler terminates can the system initiate another chain of control. Thus, from the time that the system generates an interrupt-like message until control returns from the handler activated by that message, the flow of control is strictly sequential, just as in any standard procedure-based language.

The order of the background activities is strictly fixed. The fundamental tick cycle order is as follows.

(1) *Transmit user input.* If there has

been any user since the last tick, send the appropriate messages to all objects requesting notification of that input.

(2) *Detonate alarms.* Send messages to any objects that have waited the requested number of ticks.

(3) *Detect collisions.* Either of the two previous steps may have resulted in objects appearing or moving on the screen. Now detect whether the boundaries of any pair of objects overlap or have overlapped since the last tick. This need not be an isolated step in the underlying system but, depending on hardware and efficiency considerations, might be performed in conjunction with primitives encountered in the previous two steps. In any case, by the time we get to the next step, we must have a complete list of all colliding pairs of objects.

(4) *Send collision messages.* For any pair of objects identified in the previous step, if either of the objects has declared that it cares about collisions with instances of the other object's class, then send it the appropriate message. New collisions resulting from activity during this step do not generate their own messages until the next tick.

(5) *Compose display buffer.* All processing involving programmer code is finished for this tick, so all objects have assumed their end-of-tick state. Now we can create the next display frame, based on the current state of the objects, and show it on the screen.

Game development environment

Gambit's syntax and semantics were designed to accommodate a highly interactive programming environment. A Gambit game consists of several modules, each of which is either a globals definition module or a class definition module. A class definition module consists only of the definitions of Gambit classes, and a globals definition module consists of variable, type, and constant declarations that the programmer would like to make visible within other modules.

Globals definition modules and class specification modules are Gambit's fundamental units of compila-

tion, as well as the units of source and translated code. Since each class is defined in a separately compiled module, classes can be shared easily by a number of different games. When we designed this feature into Gambit, we envisioned a diverse and growing library at each site, and this is occurring in our experimental Gambit environment. To use an example from the game presented on pp. 32-34, we found that class definitions for Puck and Paddle were used in several different games.

By using the graphics preprocessor, described earlier, an artist's designs can be pulled from the library in combination with previously defined objects to quickly create complex, polished graphics for new games. This feature not only increases design flexibility, it also simplifies debugging. The objects can be tested individually or in combination, because the development environment allows the designer to intercept messages and modify them or send new ones.

Future work

The success of our prototype implementation demonstrates the validity of the Gambit concept and its basic design. As we experiment further, we expect to extend the language somewhat. For example, support for special game hardware, such as sound generators and video disks, could easily be incorporated into the Gambit game development environment.

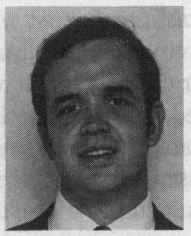
In addition to continuing our work with the Smalltalk implementation of Gambit, we are experimenting with a derivative language for the Apple Macintosh. We also are investigating the possibility of a custon architecture to support a high-performance Gambit workstation. Hardware support could significantly speed up many aspects of the language—for example, the detection of interaction boundary collisions.

We are also considering several enhancements to the language itself—for example, explicit I/O to implement games that access large databases, subclasses or some other kind of class inheritance, primitives to globally disable specific events, new types or mechanisms for creating complex types, and a velocity vector to augment the implicit object state. First, we intend to gain more experience with Gambit as it is so that we can better distinguish actual from perceived needs. ■

References

1. T. Perry, C. Truxal, and P. Wallich, "Video Games: The Electronic Big Bang," *IEEE Spectrum,* Vol. 19, No. 12, Dec. 1982, pp. 20-33.
2. T. Larrabee et al., "Gambit—A Programming Language for Designing Video Games," Dec. 1982.
3. O. J. Dahl, B. Myrhaug, and K. Nygaard, "SIMULA-67, Common Base Language," tech. report, Norwegian Computing Center, Oslo, 1970.
4. A. Goldberg and D. Robson, *Smalltalk 80: The Language and its Implementation,* Addison-Wesley, Reading, Mass., 1983.
5. B. Liskov et al., "Abstraction Mechanisms in CLU," *Comm. ACM,* Vol. 20, No. 8, Aug. 1977, pp. 564-576.
6. J. Warnock and D. Wyatt, "A Device Independent Graphics Imaging Model for Use with Raster Devices," *Computer Graphics (Proc. Siggraph 82),* Vol. 16, No. 3, July 1982, pp. 313-319.
7. K. Jensen and N. Wirth, *Pascal User Manual and Report,* Springer-Verlag, New York, 1974.

Tracy Larrabee is a PhD student in computer science at Stanford University, where she plans to complete a dissertation in the area of computer systems. This summer she is working at Digital Equipment Corporation's System Research Center. Last summer she wrote a graphics editor at Xerox Corporation's Palo Alto Research Center. Larrabee worked for two years at Hewlett-Packard Laboratories on Lisp environment implementations. After receiving her BS from the California Institute of Technology in 1979, she worked for one year at Silicon Systems on a graphics interface for an integrated circuit design system. During her final year at Caltech she worked on the CADAM graphics project at Lockheed Corporation.



Chad Leland Mitchell is a third-year PhD student at Stanford University, where his interests include computer architecture, programming languages, and compilers. He worked for three years as a member of the technical staff in the Administrative Data Systems and Communications Department at the University of Utah and has been a member of the technical staff at Bell Laboratories (currently on educational leave of absence) since 1981. Mitchell received his BA in mathematics and BS in computer science, (both magna cum laude) from the University of Utah in 1979 and 1980, respectively, and his MS in computer science from Stanford in 1982. He is president of the IEEE Computer Society Chapter of Stanford University and a member of the ACM.

Questions about this article can be addressed to Chad Mitchell at 69-B Escondido Village, Stanford, CA 94305.