1. Basic Architecture

The purpose of this document is to provide an overview of the GameKit. It introduces the major parts of the GameKit along with their respective functionality. It does not attempt to provide any intimate details, however. That is left to the class specifications and other documentation supplied with the GameKit. There are still many features that are not completely fleshed out or complete; most of these ideas are italicized to set them apart. *(Italics often also represent interesting parenthetical remarks which will probably be removed from the final version of the document, so enjoy them now while you can.)*

The GameKit itself mostly resides in the /LocalDeveloper directory. Various parts of it are to be found in the following directories:

- /LocalDeveloper/Headers/gamekit
- /LocalDeveloper/Examples/GameKit
- /LocalDeveloper/Apps
- /LocalDeveloper/Makefiles
- /LocalDeveloper/Palettes
- /LocalLibrary/Documentation/GameKit
- /LocalLibrary/Source/gamekit_proj
- /usr/local/lib
- /usr/local/lib/GameKit

What is the GameKit?

The GameKit is a collection of tools intended to simplify the process of creating entertainment software which runs under NeXTSTEP. It makes extensive use of the software kits provided by NeXT. The GameKit is not exactly a game engine, since it requires additional programming. What it does provide is a flexible structure within which a game may be rapidly built. Many games can be built by simply subclassing a few GameKit objects and dropping them into the existing framework.

Provided with the GameKit are several components. First, and perhaps most useful, are several Objective-C classes. Many of these classes may be used directly to implement a game. In fact, many of these classes can be useful in other settings besides games! Some of the classes are abstract superclasses, intended to be subclassed in order to provide actual functionality. In these cases, usually several useful subclasses are provided to serve both as examples and as a way to further reduce the amount of programming that might otherwise be necessary.

The GameKit also includes several ªstandardº .nib files which may be used to reduce development time. The .nib files are used by several controlling objects in the GameKit. The developer is free to replace these .nibs with other .nib files of the same name, however, if they wish. There are also some Interface Builder palettes which provide an easy way to incorporate the objects from the GameKit into an application.

In addition, the GameKit includes several demo applications which show how to use the GameKit to create an actual game. Also included is a generic application project which may be copied an used as a starting point for a new game. This generic project already contains several .nib files and many of the connections that would be required to build a working game.

Finally, the GameKit includes a few simple applications which may be used as tools to speed the development process.

As a kind of ªauxiliaryº inclusion, there is also full documentation and source code provided. This ensures that the GameKit may be easily customized to fit an individual developer's needs.

The following sections attempt to provide a deeper look into the classes and other tools provided in the GameKit.

GameKit classes

The GameKit's Objective-C classes can be divided into several sub-systems. In theory, one sub-system may be successfully

used without involving the other sub-systems. In many cases, however, this goal is difficult to achieve. There are cases where interactions between the various objects provide extra, worthwhile, functionality. In these cases, a developer may still use a GameKit object without the objects it depends upon, but may lose some of the desired functionality, too. The documentation for the various classes explains these cases and how to use the class by itself without losing the functionality. As a rule, these cases are kept to a minimum wherever possible.

Scoring system

The scoring system allows for very complex scorekeeping within a game. The ScoreKeeper object is the center of activity, but uses several auxiliary objects to do much of the work. Messages may be sent directly to a ScoreKeeper to add or subtract from the score (namely ±**addToScore:** and ±**subtractFromScore:**). A ScoreKeeper also keeps an internal list of BonusTracker objects. A BonusTracker simply keeps track of an arbitrary sequence of scores. This provides a way that a message may be sent to the ScoreKeeper requesting it to add the value of a particular bonus ± obtained from a BonusTracker ± to the score. An example of this sort of thing is the way PacMan gives bonus points for fruit that the Pac eats. By setting up the sequence of point values in a BonusTracker, all the developer has to do is tell the ScoreKeeper to add a fruit bonus to the score when the player eats a fruit.

The BonusTracker is basically an abstract superclass. The only type of sequence it can generate is a sequence which counts, in equal-sized steps, from some base value up to some maximum value. Also provided in the GameKit are two subclasses which generate a series of bonus values in different ways. These subclasses are ArrayBonusTracker, which uses an array to generate any arbitrary sequence, and RandomBonusTracker, which generates a random number between minimum and maximum values. The RandomBonusTracker is sophisticated enough that it can generate the numbers in steps. (For example, in NX_Invaders, flying saucers can be worth 50, 100, 150, 200, 250, or 300 points. By setting the minimum value to 50, the maximum to 300, and the step size to 50, only these numbers are generated.)

The ScoreKeeper also maintains a list of delegates which are informed of any change to the score. This way, for example, it is possible for an object (the delegate) to award the player when the score crosses certain point values. The most obvious use is to award extra one-ups every so often. (Note that the delegate may want to make use of a BonusTracker itself in order to determine which scores are the critical points¼)

The final function of the ScoreKeeper is to maintain the contents of two TextFields, one of which displays the current score and the other which displays the current high score. In PacMan, these fields are on the statistics panel. *(***** In the future, the HighScoreController will probably update the current high score field, since that makes more sense. Since both the score system and high score system were originally part of the GameBrain object, the splitting up has been a rocky process, and it's not yet completely to my satisfaction. *****)*

In a multiple-player game, several ScoreKeepers would exist, one for each player.

High score system

The high score system is composed of several objects which maintain a database of high scores. Inside a game, most messages are sent to a HighScoreController object. The HighScoreController owns one or more instances of the HighScoreTable class, which contain the actual data, in the form of HighScoreSlot objects. The information may be stored locally, in a file, by the HighScoreController's local servers, or sent across a network to a server which is capable of tracking high scores for several different games. The server makes use of HighScoreDistributor and HighScoreServer objects to perform it's functions. The basic server, although very flexible, may be extended through subclassing of the HighScoreSlot class. (The HighScoreServer and HighScoreDistributor objects use HighScoreTable and HighScoreSlot objects just like the client does. Since the HighScoreSlot stores all the relevant information, the other objects will rarely need to be subclassed.)

Note that a different style of server could be used by simply creating a subclass of the HighScoreController class that knows how to talk to the new server. The provided server uses NeXT distributed objects, but that doesn't mean that you couldn't create your own server using RPC, NetInfo, or something even more strange.

HighScoreSlot objects store the information about a single play of a game. They hold the player's name, score, starting level, ending level, time the game began, time it finished, amount to time actually spent playing, name of the machine the player ran the game on, and the player's login name. Other information could be stored as well by simply subclassing HighScoreSlot, with no other changes to the high score system being necessary.

HighScoreTable objects are a simple subclass of List which know how to keep their list of HighScoreSlots sorted. Adding a HighScoreSlot to a HighScoreTable adds it in at the appropriate place in the table, keying off the value of the score stored in the slot. The HighScoreSlot object actually provides the precedence through the ±**isAbove:** method, however, so to change the sort order, the HighScoreSlot class should be subclassed, and *not* the HighScoreTable object!

Animation system

Since the animation system is far from complete at this point in time, much of this is subject to change. Classes may be

added, their names changed, and so on. Contact Don_Yacktman@byu.edu *if you have any requests, suggestions, gripes, or whatever concerning this subsystem.*

Animation is done through coperation of the GameView, GameActor, Animator, and DirtPile classes. Both the GameView and the GameActor classes are abstract superclasses which provide a framework for animation.

The Animator class is technically not a GameKit class, per se, but rather a very useful class for managing Display Postscript timed entries, taken from NeXT's developer examples. It is compiled into the GameKit library, however, since the GameKit relies upon it. Because of this, the Animator class falls under licensing restrictions as given in the source code which are different from the license provided for the rest of the GameKit.

The DirtPile class is a class which manages flushing of image buffers to the screen in an efficient manner. A buffered window in NeXTSTEP will flush the smallest rectangle which contains all changes in the window to the screen. When animating many small objects, this is often very inefficient. For example, in PacMan, when there are ghosts at opposite corners of the screen, the entire screen will be flushed! Since only about 5% of the flushed rectangle was actually changes, this is very inefficient. On the other hand, if a game draws directly in a retained window, there is a lot of flicker. As a solution to this, the DirtPile is used to track changes within a window and flush them to the screen in as efficient a manner as possible. It allows a retained window to look like a buffered window as far as the user is concerned, while keeping much of the speed of the retained window. It simply flushes the dirty area of the window to the screen one at a time. If two objects on the screen slightly overlap, their dirty areas are coalesced to form a single, larger, rectangle. This reduces the number of messages to the window server, further improving efficiency. A developer can tailor the coalescing of dirty rectangle for efficiency so that rectangles are not coalesced if the action would increase the area flushed by more than a threshold percentage. (So a 0% threshold disables coalescing and 100% always coalesces.) This is useful, for example, if two skinny objects, one horizontally oriented and the other vertically oriented overlap. In this case, coalescing may actually be less efficient than two separate flushes. For a better understanding of how this works, try running PacMan with the -NXShowAllWindows option.

The GameView object is the workhorse of the GameKit. It's subclasses define the basic behavior of a game. The GameView itself provides a mechanism for having a static background image, complete with drag and drop facilities of .tiff, .eps, and colors. It also traps `n' and `p' keypresses and uses them to start a new game or pause/unpase the current game, respectively. It also uses an Animator object to run a state machine which controls the game itself. The framework for the state machine is also provided. *Several enhancements will appear in future releases which will include dynamic backgrounds (i.e. scrolling scenery, etc.) and other thins, as requested by GameKit users. Also, I am working on a generic state machine object which will be used by the GameView for control purposes. It will be able to send messages in a periodic manner when in certain states or whenever states change, allowing you to write methods which deal with certain key events in the game.*

Finally, the GameActor object is an abstract superclass which describes a sprite. *In the future, the name will probably*

change to Sprite *to reflect this.* A GameActor knows how to draw itself and steps through several frames of an animated sequence. The individual frames are stored in a single .tiff or .eps image; the GameActor simply draws itself on the screen by compositing from it's associated NXImage. It is capable of displaying one of several possible animation sequences at any given time. It also keeps track of it's position inside of the GameView. A GameActor is capable of limiting itself to be only on certain parts of the GameView, as well, when used with an instance of the Maze class. Without a Maze instance, it can still restrict itself to fall on a particular grid, if desired . Some subclasses of GameActor are supplied with the GameKit to help simplify things.

- The Player class is used to represent the player, and controls it's motion according to the keypresses that the user types into the GameView. (GameView objects forward keypresses to associated Player objects.) Subclasses of the Player classes can also generate their own actions intelligently when the GameView is in demo mode so that an intelligent demo may be presented to a user.

- TrackFollower objects follow a pre-programmed course, defined by an array of coordinates. They use extrapolation to determine their exact location between the pre-defined points on the track. (A utility to draw tracks with the mouse is provided with the GameKit.)

- Newtonian objects attempt to follow physical laws of motion such as momentum and gravity, and have a mass associated with themselves. They are controlled by use of acceleration, for example, which is partially determined by gravity as calculated towards a list of other Newtonian objects. Since some feautres may cause performance to vary dramatically, Newtonian objects can turn off some or all of their simulation capabilities.

- Chaser objects, a subclass of the Newtonian class, will chase another GameActor subclass around the screen. Their intelligence may be adjusted, and they can be set to attempt to guess where an object may move to next and accelerate appropriately. As with Newtonian objects, the amount of realism may be adjusted.

The astute reader will notice that as of yet, a collision detection system is lacking. The final version of the GameKit will obviously need to have this rectified. One approach is to simply compare the rectangles occupied by various Sprite objects. I am not satisfied with this, since most objects will contain alpha and thus the pure insersection of rectangles is really only a poor approximation. As an example, in Xox, near misses with planets, etc., are possible¼if anyone has any recommendations with regard to a collision detection system, I'd be happy to hear them; they could help augment some of the ideas that I have been thinking about already. The GameView will probably act as a cental controller which initiates collision detection, but I expect the code to be located in the GameActor subclasses. Currently, the system I'm planning on is to check for overlapping rects; if they do overlap, then check for intersections of custom Postscript paths, which you would have to provide for a given object. Another possibility is to use an extra image as a "mask" and check for masks which collide¼but I'd rather not have to do that.

Sound system

Two important objects allow games to make noise. The SoundPlayer plays back digitized sounds and the ScorePlayer uses the MusicKit to play back .score, .playscore, and .midi files. Both objects are quite independent of each other and the GameKit, and may therefore be used in a variety of settings.

The SoundPlayer object keeps a list of lists of sounds. At first this seems a bit odd. Each list of sounds is a list of the sounds for an entire game. By switching the list, the game can instantly play different sounds for each noisy event. An example of this is the game Columns, which has two different sets of sounds from which the user may select, one for the falling blocks and the other for the falling gems. In a SoundPlayer object, the sounds are played back using the NeXT SoundKit and do not use the Sound object's ±**play** method because it is incompatible with the MusicKit. A SoundPlayer may be set up such that more than one sound will play back at any given moment in time or such that all sounds are queued up so that a sound doesn't start playing until the previous sound finishes.

The ScorePlayer object allows for playing back musical scores via the MusicKit. Currently, in order to use this object, you must have the 3.1 CCRMA MusicKit installed. Using the music kit will cause a game's binary to nearly double in size because it is no longer a shared library. (Thanks a lot, NeXT¼**NOT**!) On the bright side, it works with the SoundPlayer object allowing creation of games which have music and sound simultaneously, which is pretty cool.

Miscellaneous objects

The GameKit also includes several classes that simplify other parts of the user interface and so on. These objects include the InfoController, GameBrain, GameInfo, PreferencesBrain, ExtendedApp, WinDel, PlayerUpView, Maze, and RandomNumber. *In the future, other objects may be added, or the functionality of these objects may be split into new objects, depending upon feedback that is received from users.*

The InfoController controls the Info¼ menu. It provides methods for animating an Info panel, loads .nibs for various panels such as the Info panel, Registration panel, and Order Form panel. It acts as the intelligence behind the Registration panel and the Order Form panel. (It also provides hooks to allow use of the Simson Garfinkel and Associates, Inc., registration objects.) Finally, an InfoController can bring up a "README" file in the NeXTSTEP Help Panel.

The GameBrain handles such things as popping up alert panels at various times and providing a repository for various "global" variables that other GameKit objects might want to know about. It is expected to be the Application class'

delegate, so it may be accessed via **[NXApp delegate]** from anywhere in the program. This allows it to trap important things like the user wanting to quit the game and initiate initialization of various objects. For example, all subclasses of GameView are sent a ±**loadPix** message to initialize them while the Loading¼ panel is up (put up by the GameBrain, in fact). This allows all the offscreen buffers to be created while the user waits. This is useful, since NXImages don't create the buffers until required for drawing; for example, in the current version of Xox, the first time an explosion occurs, there is a pause while the offscreen image buffer for the explosions is created. The GameView and GameActors create all these buffers at the start so that the animation is always as smooth as possible±at the expense of lengthening launch time. Many of the subtleties of user interface that are in games like Columns, PillBottle, and PacMan are provided by the GameBrain. (Like most of the alert panels, auto pause/unpause, and so on.)

The GameBrain also knows the **id**s of most of the important GameKit classes, so objects that need to find the GameView subclass, PreferencesBrain, ScoreKeeper, or whatever, can ask the GameBrain where they are. The GameBrain can also start and end a game and well as control pausing and unpausing, along with requisite interface details such as enabling/disabling menu items or changing names of menu items.

GameInfo objects allow you to adjust many of the GameKit parameters without the need for recompiling or subclassing. Many GameKit behaviors may be altered by changing these parameters, such as size and number of high score tables, the number of sound effect6s used by your game, and so on.

PreferencesBrain subclasses allow the user to alter the various parameters of the game that the developer is willing to let them change. It provides hooks to turn sound and music on and off, alter game speed, adjust starting level, change keys used to play the game, and change the method of storing high scores. Other features may easily be added by a subclass. The PreferencesBrain class also handles switching views on a multi-pane Preferences panel, if necessary.

The ExtendedApp class adds a few functions to the NXApp class that can be handy to the developer. It provides wrappers around several UNIX functions and returns values for the group and user id's of the process, as well as the user's login name *and* full name as supplied by netinfo. It also returns the path to the app wrapper, which is needed by the HighScoreController class, for example, to store local high scores. (Note that the main NXBundle now can supply this as of 3.0, so this method isn't as useful as it used to be.) This object is actually a part of the daymisckit library of miscellaneous useful objects, but the GameKit relies upon it.

The WinDel class has been used to keep track of auxiliary panels for the GameBrain class. In most cases, these aren't very useful; they are typically used only in cases like PacMan where windows have to be kept in certain orders.

The PlayerUpView displays how many lives the player has left. It is given an NXImage of what the player looks like on the screen and uses it to draw remaining lives. It also can be used as a delegate to the ScoreKeeper, with an appropriate BonusTracker, to award extra lives. When the player dies, the Player and GameView objects can query this object to see if it is allowed a new life, and if so take one of the lives.

An instance of the Maze class is capable of limiting where a GameActor is allow to move to on the GameView. It also know the initial position of a given GameActor at the start of any level. It controls reading the appropriate levels from the app wrapper and also renders the maze on the GameScreen. GameActors query it to find out what restrictions, if any, should be placed on their movement. The mazes themselves can be specified by a simple text file, and several behaviors such a walls are one-way doors are predefined; a subclass could add more "special" types of maze locations.

The RandomNumber class simply provides random numbers to the GameKit. It wraps around the random() C function but is really provided only as a way to interface the GameKit to other random number generators provided either through public domain or commercially. By creating a subclass of RandomNumber, the developer will be able to change the way the GameKit gets it's random numbers internally. Thus, the low budget project just uses the standard C functions whereas the high powered software companies can use whatever SuperAmazingMostExcellent™ random number generator they prefer to use. *(I intend to at least provide interfaces for the CDS RandomSystem™ as well as the public domain Random class available from the Internet archives, for those who may care use either of these systems. My supporting them has nothing to do with product endoresement, etc. I'm just trying to be helpful. ***** This hasn't yet been implemented. *****)*

Interfaces provided with the GameKit (.nib files)

Several .nib files are provided with the GameKit to simplify the building of games using the GameKit. Currently, .nibs are provided for the following (they all have internal StringTable objects, as well):

- Registration panel
- Order Form
- Info panel (displays version number, etc.)
- High Scores panel
- Preferences panel
- Skeleton main .nib with menu, etc.

As of this moment, it has not yet been decided what is worth putting on palettes. Any input as to this might be helpful. Probably most of the controlling objects and the GameView will be available for this, as well as a few menu items for starting games and pausing games.

Demos provided with the GameKit

There are currently two games which demonstrate how to use the GameKit to build a game:

- NX_Invaders ±€the classic game of Space Invaders, which no computer is complete without.
- PacMan ± another arcade classic we can't live without

A generic game framework is also provided which has the PB.project set up to link in the GameKit and also use the appropriate headers files when compiling. It also has the "standard" .nibs mentioned above as well as a head start on the Help files.

These are, of course, not the only times the GameKit has been used. An even more extensive use of the GameKit is made by Columns and PillBottle, shareware applications currently available. (Source code for these applications is available to registered users upon request.) There are also several other up and coming games from Don Yacktman which will use the GameKit, some of which might be available in source code form eventually.

Development tools

Currently, there are two tools available. The first is TrackDraw which allows you to create tracks for TrackFollower instances. It creates both line segments(fast animation), line segments approximations to cubic splines (still fast), or actual cubic splines (slow) to define the path to be followed by the TrackFollower. The second is a simple utility that uses a GameView and a Player instance to test out various animation sequences to see how they look. This allows a developer or artist to see what a character looks like without having to actually compile a game. It's interface allows control of most GameActor parameters and keyboard control to move the GameActor around.

Source code, documentation, and support

The full source code is distributed with the GameKit. If it wasn't in the distribution you got, then you didn't get the real GameKit! The source code attempts to provide liberal comments so that it is obvious what is going on inside. Of course, the class spec sheets and other documentation should make poking around the source code rather unnecessary. Providing source does allow users to enhance the GameKit, however. If you do make any enhancements, you are required by license to share them with the original author so that other folks can take advantage of them as well. Note that this applies only to actual changes in the GameKit objects and associated files themselves, and not to subclasses that you create. The object is to not keep commercial developers from using the GameKit, but rather provide a central source for support and enhancements to the GameKit and prevent things from dipping into utter chaos as everyone tears up the code!

If you find bugs in the GameKit, or errors in the documentation, please notify Don_Yacktman@byu.edu so that things can be fixed quickly. This is obviously to everyone's benefit. Also, for discussion regarding the GameKit as well as a source of infromation about it, you may wish to subscribe to the GameKit mailing list. Simply send mail to Don Yacktman at the above address, or send a message to gamekit-request@byu.edu (use the request address to get off the list, too). To send a message to the list, send mail to gamekit@byu.edu. If you have a specific problem, it should probably be sent directly to Don Yacktman. If you have a question, remark, or suggestion that would be of public interest and might spark discussion, it's probably better to send it to the gamekit mailing list. (Don Yacktman *does* get the mailing list, too, by the way, so he'll see what you post there just like everybody else.)

Since this GameKit project is largely a for-fun thing and does generate any real cash, don't expect the world's best support. On the other hand, if you want to pay me as a consultant, I'll give you the best support I can. As awful as it sounds, I listen when money talks. If you don't pay me, well, you'll still get a great deal (as if the GameKit itself isn't wonderful :-)) since I try to courteously reply to e-mail inquiries within 24 hours whenever possible. Obviously, though, things that put food on my table take precedence. Since many of the projects that I expect will generate revenue for me in the future rely upon the GameKit, though, bug fixes and updates will certainly happen! If you have any suggestions for things which should be added to the GameKit, please do let me know. I want the GameKit to be as flexible and powerful as possible. Nothing's set in stone yet, so don't hesitate to harass me.

± *Don Yacktman*

Philosophy and licensing (a personal note)

This is my space to ramble on, and it gives the curious reader a feel for where I'm coming from. Hopefully I won't bore anybody too badly. :-)

The GameKit came about because of all the games I've put together. I've placed two shareware and one freeware game on the Internet archives, and have several other games at various stages of development. In the process of building these games, I've built and refined a framework which I feel is useful for building games under NeXTSTEP. It seems silly to me to keep this all to myself. There are thousands of lines of code that are pretty much common to any game; why make everyone else re-invent the wheel? Not to mention the hours spent in deep thought trying to come up with useful abstract classes and object hierarchies¼ I don't claim to have come up with necessarily the *best* way to do things, but what I have here *works*.

The GameKit is intended to spark development of games under NeXTSTEP. It's object is to make the process as easy as possible. I'm sure several other creative uses for the GameKit and it's objects will show up. That's great, and I hope it happens! Not only should the GameKit be easy to use, it should be as flexible as possible, and provide as many re-useable objects as might be necessary. An attempt has been made to provide a framework which is comfortable, unrestrictive, and yet useful. Obviously, much is missing; users requesting enhancements will help accelerate addition of more useful features.

Now the question arises. I've put a lot of time into this, as the expense of health, social life, and opportunities to make mounds of money doing other things. So why do I bother? I like NeXTSTEP, and want to promote it. If I end up programming the rest of my life, I want it to be on NeXTSTEP, or a system which is superior. Since I don't believe a superior environment exists, here I am. (By the way, Solaris, Windows NT, and X-Windows are quite definitely inferior, and if someday NeXTSTEP dies, well, I might as well find another profession. NeXTSTEP is the only environment that I've really had *fun* programming, and since my baser nature is hedonistic¼ :-)) On the other hand, though, by making a public release of the GameKit I stand to get screwed over by certain people who would take all my work and use it in a commercial product with paying me for my time. If that happened, I'd be irked.

This brings me to the part I hate to have to write about, but I feel this is a necessary evil. Therefore, I have decided that the GameKit is freely available, but¼ if you plan to charge money for a product which makes use of *anything* in the GameKit, you owe me a license fee. Talk to me and we'll come to some sort of agreement as to what is fair. This includes both shareware and commercial products. If you distribute your project for free, then you don't owe me anything. This will probably spark a lot of free games. I hope so. But, I'm not trying to discourage shareware or commercial authors from making use of the GameKit: I'm really quite a reasonable person when you talk to me, so get in touch! (Actually, though, if you're considering doing shareware, don't bother. You'll be lucky to make even $1/hr for the time invested, if your experience is anything like mine.) The license fee I'd prefer would be a small percentage of the gross on your application, since that way, if your app doesn't sell well, you're not out lots of money, but if it does sell well, then I can pay my grocery

bill¼

By the way, I don't feel that it's fair for someone to pore over the source and then use all my ideas to write their own stuff and then sell it, either. I've spent a lot of time organizing and designing the GameKit, and having others effectively steal my ideas just seems obnoxious. The object hierarchies and much of the animation stuff took a lot of thinking. In fact, back in my days of assembly on the Apple][, I actually put together the beginnings of much of this, including several nifty tools, so it's not like I just came up with this stuff last night. :-) Of course, there's probably not much I can do if somebody does use all my ideas, and it's a risk I take by releasing source code. (Also, I feel that software patents are evil things, so I'm not trying to be all that hard nosed about this. A developer knows when they've stolen someone else's ideas, and so I think that in many cases, the conscience will have to decide. And I certainly don't claim that all the ideas in the GameKit are completely original; in the comments in my code I try to give credit where it's due.) Anyway, I leave this up to you; if you plan to make your own GameKit that duplicates my stuff just so you can avoid a license fee, you're probably out of bounds and you're wasting your time, since I don't plan to ask for a large fee at all. Anyway, I think the it is worth taking the risk to distribute source for the benefit of those who are honest, and besides, if someone does play the scumbag, well, they have to live with themself, which would be bad enough punishment in most cases!

Now, one other question remains, about which I have not yet made any decisions. (That's an invitation for you to add your own $0.02¼) Should I allow others to help me out with the GameKit and if I do, how should the licensing be handled? I'm currently of the mind to allow others to pitch in if they want to, and they'd get a percentage of whatever license fees come in. (And I don't actually expect to make much money off this, so beware to those who think jumping into this project will be lucrative. I really suspect that it won't be.) However, I don't expect any help with this; if you feel you really could offer something useful to the project, however, please do contact me.

Finally, I have been apprised of at least two other GameKit types of projects. Even so, I am planning on continuing with this one, because I feel that it has it's merits. From what I gathered about these two projects, one is more in the intellectual stage, and does not include as much of a framework as the GameKit, but rather emphasizes animation ± which could provide good input as to how to improve the GameKit±and the other is more of an engine, sort of like BackSpace, which uses loadable modules to create games. It seems like a good idea, and Xox makes a great proof of concept: it works. However, I feel that the GameKit is a more flexible and "industrial strength" solution. It requires the developer to do a little bit more work, but also has the potential to be faster and more flexible. Both approaches seem valid and worth pursuing, from what I've seen to this point.

If you have any comments, questions, suggestions, worries, or gripes, please feel free to contact me. My e-mail address is Don_Yacktman@byu.edu and if it's something really important I'll even give you my phone number. If all else fails, you can write to me at the address below.

± *Don Yacktman*
4279 N. Ivy Lane
Provo, UT, 84604