2. The Core of a GameKit Application

When you build a game from the GameKit, you can consider it to be a game ªcoreº surrounded by various subsystems, such as score keeping and preferences management.   Sometimes the subsystems will interact, but only when absolutely necessary.   Whenever the core and subsystems need to communicate, it is done via the GameBrain object, which is intended to be the Application object's delegate.   The core functionality itself resides in a GameView subclass, in its code and in the objects it manages.   Most of the subclassing and customizations you will make are to the GameView class and various associated objects; most of the other subsystems can be easily configured from InterfaceBuilder™.

This document attempts to describe the architecture of a GameKit based game and provide details about the two main objects in the GameKit, the GameBrain and GameView.   Once these two objects' functions are understood, the other subsystems become quite simple.

Architectural Overview

The diagram below maps out a typical GameKit based game, showing how the objects are connected together.   It is *not* an inheritance tree.

paste.eps ¬

Note that the subsystems need to communicate with each other for various reasons. The GameBrain has methods for locating each of the subsystems, but it is more efficient for a subsystem to cache a pointer to the subsystems with which it communicates. This is done as the application launches, initiated by the ±**appDidInit:** method, which is sent from the Application object to the GameBrain. If you have not explicity connected the subsystems in your main .nib file, the GameBrain will automatically create all the necessary connections. If you have made explicit connections, they will not be broken by the GameKit. This provides an easy way for you to override the structure of a GameKit game, allowing you to use different subsystems from what the GameBrain might expect. In some cases, such as the BonusTrackers, the responsibility to connect objects together is left to the programmer, since the GameKit cannot predict how a game will make use of BounsTracker, Sprite, and StateMachine objects; it is highly game dependent.

The GameKit assumes that the Application object is of the class ExtendedApp, which is found in the DAYMiscKit object library, which should be available from where you obtained the GameKit. The reason for this is that the ExtendedApp provides methods that give more information about an application, information that the GameKit needs. Mostly, this extra information is use by the high score subsystem.

Under most normal circumstances, you will not need to change the above architecture. If you do try to change it, expect to break things that you have been taking for granted; the GameKit attempts to cover many tiny details that make a game's interface simply ªnicer.º A lot of these details will be lost if you make drastic changes¼and there are actually very few cases that this generic architecture will not cover. (Note that the architecture underneath the GameView, to be discussed later, is very game-dependent and will change dramatically from game to game. That's where 90% of your work will end up being focused.)

GameBrain

The GameBrain acts like a distributor cap; it knows where all the subsystems of the GameKit are and is able to identify them to other GameKit objects. It also functions as the delegate for the Application object and the GameView's Window object. In addition to these uses, the GameBrain handles the top-level logic of a game: starting, stopping, changing level of play, and pausing. It also keeps a detailed record about the current game, stored in a HighScoreSlot instance, which is passed to the high score subsystem upon completion of a game.

Application delegate

As the delegate of the Application object, the GameBrain receives messages about the Application object's state. The first such message is the ±**appWillInit** message. The GameBrain, when this message is sent, will attempt to place a ªloadingº panel on the screen to inform the user that the game is initializing itself. Since initialization may take a while, it is a good idea to display this panel. The panel itself is connected to the GameBrain's **alert** outlet. If nothing is connected to the outlet, then ±**buildAlert** is invoked to build an alert panel programmatically. The application then proceeds with it's initialization. Once this is complete, the ±**appDidInit:** method is invoked.

Any special initialization twhich you need to do in your game should be placed in a GameBrain subclass which extends the ±**appDidInit:** method by overriding it (and calling **super**). The GameKit implementation puts up a new panel, pointed to by the **loadingPanel** outlet, and then steps through each of the subsystems one at a time and initializes them. If the **loadingPanel** outlet is the same as the **alert** outlet, then the **alert** panel is left up. As the various subsystems are being initialized, the GameBrain places text in the **loadingText** text field to tell the user what is being initialized. All the strings for this come from the GameBrains StringTable instance, pointed to by **strings**. Finally, the ªloadingº panel is ordered out and an initialization event is added to the Application's event queue. (An application defined event, of type GK_STARTUP.)

The GK_STARTUP event, when sent, invokes the GameBrain's ±**startUp** method which will attempt to either bring up the file README.rtfd in the NEXTSTEP Help Panel or start a new game, if the user has turned on the ªNew Game Upon Launchº preference. If you are using the registration key system and the game is unregistered, a warning panel will also be brought up.

The GameBrain also is sent messages whenever the Application is hidden, unhidden, activated, or deactivated. The GameBrain will attempt to pause and unpause as would be appropriate. Note that when unhiding an Application, it is often important to assure that the various satellite windows to the main game window be layered properly. In order to do this, the ±**layerWindows** method is invoked. (Note that the WinDel object is useful when implementing this method; see the later chapter describing the WinDel class to better understand this method.)

Finally, the GameBrain is sent messages when the Application object is about to terminate the application; this allows saving of preferences and other vital information before shutting down completely. Note, however, that the Quit menu item in the main menu should be connected directly to the GameBrain's ±**quit:** method and *not* to the Application object. This allows the GameBrain to intercept the quit button's message and abort if the user decides not to quit. Anytime the user attempts to abort the game, the ±**askAbortGame:** method is (or should be) sent to the GameBrain. The sender should only continue with the abort if this method returns a YES.

Window delegate

The GameBrain should also be the delegate for the main game window.   This allows it to pause the game whenever a window is closed or miniaturized and unpause when a window is opened up on the screen (if the ªAuto Unpauseº preference allows it).

If the main game window moves, then the GameBrain will attempt to reposition the sattelite windows so that they ªfollowº then main window around.   You will most likely want to override the ±**windowDidMove:** method to alter this behavior for your game.

Other functions

The GameBrain can print the game's main window via the ±**printGame:**.   The main reason for sending this message to the GameBrain instead of the window itself is so that the title of the window may be adjusted appropriately.

Information about a game is maintained by the GameBrain.   It uses a HighScoreSlot (or subclass) instance to store this information; the ±**buildNewSlot** method may be overridden to use a specific class and stuff different initial information into the HighScoreSlot.   The ±**currentHighScoreSlot** method returns the slot, completely up to date.   This may be used to obtain information about the current game.   A basic HighScoreSlot knows the login name of the user playing the game, the time the game started, how long the player has been playing, the starting level, the current score, and so on.   When the game ends, the current high score slot is updated one last time and then sent off to the high score subsystem which will then handle the slot appropriately.

You can use the methods ±**startNewGame:**, ±**abortGame:**, and ±**pauseGame:** to control the game itself.   Note that the ±**pauseGame:** method is a toggle between pause and unpause, whereas the ±**unpauseGame:**, ±**pause**, and ±**unpause** methods are not.   If you want to initialize anything at the beginning of a game only, then it should be placed in the ±**startNewGame:** method of a GameBrain subclass.   Be sure to call **super**, though!

Whenever a level is completed, you should send the GameBrain the ±**nextLevel** message.   There is also a ±**nextLevel:** message, but since it can be sent as an InterfaceBuilder™ target message, it causes an internal ªplayer is cheatingº flag.   As a result, you should never call the ±**nextLevel:** message from within your code unless you want this flag set.   In particular, this flag will affect whether or not the player is allowed to get his or her name into the high score table.   Currently, there is no warning to the user when this flag is about to be set; in the future there may be, as well as a mechanism to allow the player to back out.   If you need to do any special initialization at the start of a level, then it should be placed in the

±**nextLevel** method in a GameBrain subclass, along with a call to **super**.

Every game based upon the GameKit has a GameInfo object which is used to configure the GameKit dynamically.   By changing the parameters in the global GameInfo object, you can alter many of the GameKit's features.   Since this object is available on an InterfaceBuilder™ palette, it allows you to make many customizations without the need for subclassing of GameKit objects.   If you don't connect a GameInfo instance to the GameBrain's **gameInfo** outlet, however, the GameBrain will invoke the ±**makeGameInfo** method to create a default GameInfo instance.

Finally, the GameBrain has methods which return pointers to the main controlling objects of the various GameKit subsystems.   These methods include: ±**highScoreController**, ±**oneUpView**, ±**scoreKeeper**, ±**scorePlayer**, ±**soundPlayer**, ±**mainStrings**, ±**gameScreen**, ±**gameWindow**, ±**preferencesBrain**, ±**infoController**, and ±**gameInfo**.   As long as you remember to connect the GameKit subsystems to the GameBrain's outlets and connect the GameBrain as the delegate to the Application object, the rest of the GameKit will be able to use these methods to find the other subsystems and the game will automatically build up most of the necessary connections for you.

GameView

The GameView is a very big class and attempts to provide a framework within which user interaction and complex animation are both possible.   The GameView itself accepts user events±mouse and keyboard events±and attempts to forward them to the appropriate place.   When you override View methods like ±**mouseDown:** and ±**keyDown:**, it is very important that you still send messages to the **super** implementation of the method, since the GameKit uses many of these methods to automatically unpause the game.   Be sure to look at the appropriate examples to see how to best accomplish this.

Aside from user events, the GameView also provides a framework for animation and game logic.   Chapter 5 on the animation subsystem of the GameKit specifically addresses the issues involved in creating the visible part of a GameView. The GameView itself provides a framework for the animation, including buffers and ways to use various animation support objects, but it relies upon the GameActor and other objects to do actual rendering.

A background image is maintained in a buffer by the GameView and may be changed via messages (±**setBackgroundFile:andRemember:**, ±**revertBackGround:**), drag and drop, or the Open¼ panel (by sending a ±**changeBackground:** message).   You can also drag and drop a solid color to fill the GameView's background.   *Currently, the drag and drop of images doesn't support 3.0 image filters very well, even though it is using the 3.0 drag and drop* *protocol. This will be rectified before the first non-beta release of the GameKit, of course. For now, stick to using .eps and .tiff images and things will work just fine.*

If you want to have an appearing/disappearing bezel in the GameView, as is seen in Columns and PillBottle, the GameView supports this. It is recommended that you avoid this, however. It exists mainly because of poor original design in the original Columns and PillBottle code. The only reason it still exists is that there are some people who like to play the games with BackSpace running behind the game, and a bezel-less window looks really cool against the predominantly black background of the Space module, expecially when you first start Columns and the screen is empty and is set to a solid black background image. I prefer the bezel, myself. The ±**changeBorder:** method handles this, and since the preferences subsystem already supports this feature, there is no need for you to worry about it in your code; just make the preference available on the Preferences¼ panel.

At the start of a new game, the ±**restartGame** method is invoked, so you can place initialization code for the start of a game in here. Whether you place this code here or in the GameBrain really depends upon your whims at the moment. The advantage of this method is that it is called from the middle of the GameBrain method, which may or may not be useful. Look at the source code of the GameBrain and decide for yourself. All images that your animation will use should be loaded into appropriate instance variables that you have added to your subclass of GameView from the ±**loadPix** method, which is called during the ±**appDidInit:** phase in the GameBrain. The current implementation prepares the buffers and loads the background image. You will probably want to load the sprites' images here as well.

The rest of the GameView is described in the chapter on animation. There are two significant parts. First is the actual rendering code, which fits within a specific framework of GameView methods. *The GameView itself does practically nothing alone, and as it currently stands needs to be subclassed to get anything useful onto the screen. As the animation and state machine facilities of the GameKit are expanded, this will probably change, since display lists will be used. The current state of the source for PacMan, for example, is a good measure of this. You'll see a major overhaul when the state machine facilities are finally completed.* The second part is a state machine which defines the behaviors of the game and the various phases of play and animation. Building the state machine is complex enough that it gets it's own chapter (Chapter 6) and the two programming examples (NX_Invaders and PacMan) to help you understand the process. *The current implementation puts the state machine in the GameView by simpling using a big method, ±**autoUpdate:** to contain the relevant code. Even for a simple game, this method can end up being huge, and this is why the state machine objects are being developed¼not to mention that it will make the code much easier to follow!*

There remain a few methods of the GameView which will not be discussed here; this is because they are mostly for the GameBrain and other parts of the GameKit to use; they are intended to provide information about the current status of the game and allow the subsystems to alter dynamic parameters as appropriate. (Most of the methods are very simple and their function is obvious from looking at the header file and class specification, hence there is no need to further discuss them here other than to note that they exist.)