4. High Score System

This chapter explains how to use the high score system provided by the GameKit. The high score system includes objects for manipulating high scores and a few example .nib files which may be used in your application, with or without modification. Also included are some programs, a network high score server (in Apps/HighScoreServer), a high score editor which allows you to ªcheatº with high scores (in Apps/HighScoreEditor), and an application which allows you to test and experiment with the high score system (in Apps/HighScoreTester).

The GameKit classes used to implement a high score system are the following: `HighScoreController`, `HighScoreSlot`, and `HighScoreTable`. High score servers also make use of two additional classes: `HighScoreDistributor` and `HighScoreServer`. Both sets of objects know about the protocols described in `HighScoreProtocols.h`. Under most circumstances you should not need to alter or subclass the table, server, and distributor classes. Most customizations will occur with the slot and controller classes. Figure 3-1 below shows how these objects interact. The rest of the information in this chapter explains the purpose of the various connections in the diagram. One final object, which is not shown, is the `GameInfo` object. A `GameInfo` object is used to tell a server object about a particular game; its function is described below in the section on network high scores.

paste.eps ¬

The rest of this document describes the high score system from different perspectives. First is a list of what you will need to do to add high score capacity to your game. Next, the procedure for setting up a network high score server is described. That section includes a conceptual explanation of how the server and client (game or editor) interact. An explanation of how to configure the high score system is next; it describes changes which may be made by simply subclassing the GameInfo object±adequate for most games. Finally, many of the changes which would require subclassing high score objects are discussed.

High Scores in a game

The HighScoreTable is the central depository of high score information in a game. It is basically a list of HighScoreSlots, each of which contain information about a single game play. All objects which wish to change or look at the information in the HighScoreTable must do so via the HighScoreController. This includes other GameKit objects, the interface objects, and any objects which you add to the game. In order to use high scores in your game, you must do the following:

- In the main .nib of your game, instantiate a HighScoreController.
- Control-drag a connection from the existing GameBrain instance to the HighScoreController instance. Set the connection to be the *highScore* outlet of the GameBrain.
- Copy one of the GameKit high score .nib files to your project's ªEnglish.lprojº directory, rename it ªHighScore.nibº, and add it to your ProjectBuilder project. The .nib file you choose will depend upon which `HighScoreSlot` information you wish to display.
- Add any menu items you'd like to have for the high scores. Examples might be ªHigh Scores¼º or ªClear High Scoresº. Menu items should be connected to the HighScoreController's action method of choice. The two examples given here would

connect to the ±displayHighScores: and ±clearHighScores: methods, respectively.

Everything else which is necessary is handled automatically by the GameKit. The HighScoreController, for example, creates it's HighScoreTable and loads the .nib with it's associated interface objects when needed. As they are needed, `HighScoreSlot` objects are created and deployed automatically by the `GameBrain` and the `HighScoreController`. Note that the GameBrain object can tell other objects the *id* of the HighScoreController. Therefore, if you have an object which needs to connect to the HighScoreController, it should obtain the *id* via the GameBrain. The following message is an example of this±remember that the GameBrain is the Application's delegate:

```
id hsControl = [[NXApp delegate] highScoreController];
```

Once the above items have been completed, your game will support high scores. The GameKit will automatically load and save high scores, the preferences system will allow the user to choose between local and network high scores, high scores will be updated as the game is played, and new high scores will be inserted into the list when a game ends. The following information is stored about a game: score, starting time, ending time, elapsed time (with paused periods not counted), starting level, ending level, player's user name, player's name as entered by the player, and the machine the game was played on. The default system will limit the number of high scores to 10 high scores in a given game and will limit a particular user@machine to only three entries in the network table or 10 entries in the local table. (***** number of entries limiting is not yet implemented! *****) If any of the default behaviors are inadequate or inappropriate for your game, then you will want to begin subclassing things in order to get the behavior you want. Later sections of this chapter give some ideas for and examples of modifying the behavior of the high score system, but by no means do they list all the possibilities.

Warning: in order for all the information stored in high scores to be generated automatically, you must be sure to use the standard GameKit methods; many of the GameBrain's methods for handling level changes and the starting and stopping of games are used to generate this information. If something isn't working right, be sure that your subclasses of GameBrain message *super* and that you aren't bypassing the GameBrain somewhere. If you intend to use the high score system without using the rest of the GameKit, then you should look at the

implementation of the GameBrain to see how it cooperates with the HighScoreController and implement similar code in your system.

Network High Scores

If you followed the steps outlined above, your game will already support network high scores and will be able to use any available general server which already exists on the network. You may wish, however, to set up such a server yourself. Here is what you have to do to create your own server:

1. Compile the HighScoreServer example in the GameKit. Strip the executables which are created (high and serverd). (To compile and strip, just type ªmakeº.) The `high` program is what is run to start the server; `serverd` is the program it runs. If `serverd` crashes, `high` will restart it automatically.
2. Move `high` and `serverd` to /usr/local/bin. You may wish to create a user name like ªhighscº and install `high` and `serverd` to run setuid to that user. This will allow only root and the server to access the high score tables on disk, which should enhance security.
3. Change /etc/rc.local to start up the server at boot time. (Add the line ª/usr/local/bin/high &º somewhere in the file to accomplish this.)
4. Create the directory ª/usr/local/games/highscoresº. You should set it's owner to the username from step two above if you are using that scheme. This is where high scores (and dynamically loaded HighScoreSlot subclasses) will be stored. The average game will typically not require more than about 1 kbyte to store high scores, so this directory won't take up very much space. Note that if you want the scores stored in a different directory, you can re-make the GameKit with `PATH_TO_TABLES` set to the appropriate path in HighScoreServer.m in the GameKit source.
5. Reboot or run `high` manually and you'll have a high score server running on your machine.

When a game connects to the server for the first time, the server will ask it for it's GameInfo object, which will be used to configure a new server for the game. The server will attempt to configure itself to your specifications as much as possible, even up to dynamically loading code for custom HighScoreSlot classes. (***** In the future, if the slot code isn't on the server's system, the server will ask the client to send the code over the network, making it totally dynamic. This is one of the last feature I'll be putting in, however, since right now it's a low priority. *****)

Network score system protocol

This section describes the communication between a game and the high score server. Games don't make use of all the methods available, but a high score editor would do so. (Note that the special methods provided for editors require clients to validate themselves by sending passwords before they will work.)

When a game connects to the high score server, it is given a connection to a `HighScoreDistributor` object. The `HighScoreDistributor` object's sole purpose is to keep track of the servers which is currently has running. Each game has it's own server; for instance, PacMan, NX_Invaders, etc., might be the servers which are available. The game asks the HighScoreDistributor for a connection to the necessary server. (NX_Invaders would ask for the NX_Invaders server, for example.) If the server is running, then a connection to the server itself is created and the game will then check in as described below. If the server is not running, then a new server instance is created and the appropriate connection is again formed. If there hasn't been a server for this particular game running before, then the server will ask the game to send it information about itself, in the form of a `GameInfo` object. This information is used to configure the server and is stored with the high scores in /usr/local/games/highscores. (Technically, the exchange of configuration information actually happens during the first check-in of a client, which is described below.)

If a game requires a special type of `HighScoreSlot`, then the server will attempt to dynamically load the code for the slot from the highscore directory, if it isn't already loaded. If the server

can't find the slot code, then the server will inform the client that a proper table cannot be stored. Thus, any game with a custom slot type needs to be distributed with a .o file for the slot which may be used in conjunction with the server; people maintaining servers need to be given copies of the compiled slot so that they can provide services. (***** As noted above, in the future, leaving the appropriate .o file in the .app wrapper will be sufficient, as a facility to transfer the .o file over the net to the server will eventually be provided. *****)

Once a network connection has been established, a game ªchecks inº to tell the server that it exists. This allows the server to dynamically update the game's high score table. The complete high score table is sent to a client when it checks in. At this point, all connections and initializations are complete. Now, whenever a gameplay completes, a client will submit an appropriate `HighScoreSlot` object to the server for inclusion in the network table. If the server accepts the slot, then it will forward the change to all the other clients which are currently connected. A slot is only sent if it is better than the lowest score currently in the table. However, there is the slight possibility of a score submission and a table update message to cross each other in transit. If this happens, the slot may end up being thrown out. Since the server is currently single-threaded this is no problem. In the future, the server will run multi-threaded and this will be taken care of via locks on the high score table.

If a client attempts to send a message reserved for an editing program, then the server will check to see if that client is validated. If not, it will ask the client for validation password before evaluating the method. If incorrect, then the message is thrown out. If proper validation is sent, then this fact is remembered until the client disconnects. Each game may have a unique password; an encrypted form of the password is found in the `GameInfo` object. (This allows a game maker to distribute source for their game and still keep the password a secret. Standard UNIX password facilities are used to perform the encryption of passwords.)

When a game or editor is about to quit, or wishes to disconnect, it informs the server that it is leaving by ªchecking outº. After that point, messages will not be sent to the client when the high scores change. If the client crashes, the server is smart enough to check the client out automatically.

Note that wherever possible, messages between client and server are `oneway`. This design gives

faster perceived performance for the user but also makes the protocol more complex than it might otherwise be, including making it asynchronous.

Warning: Before you modify any of the objects which interact over the client/server connection, make sure that you really understand what is going on! A seemingly harmless change could induce crashes and/or deadlocks quite easily and render the high score system inoperative.

Simple customizations

The high score system is really quite flexible as it currently stands. This means that you may only need to adjust the `GameInfo` object to set up the high score system to function the way you require. The `GameInfo` object is provided on an InterfaceBuilder palette. If you main .nib doesn't have an instance of the `GameInfo` object connected to the `gameInfo` outlet of the GameBrain, then drag one off the palette and into the main .nib and connect it to the GameBrain's `gameInfo` outlet. Then, select the `GameInfo` object and inspect it in InterfaceBuilder. Changing the various values that appear will alter the behavior of the high score system (as well as other GameKit parameters discussed in other chapters). Double clicking on the object will open up an editor window which provides extra options that aren't in the inspector. (***** The palette has not yet been built. You'll need to override the ±`makeGameInfo` method of GameBrain to return a programmatically built GameInfo for now. *****)

Here are the parameters that affect the high score system:

- `maxHighScores` ± The maximum number of slots in a given table. (***** in the future, this may be on a per-table basis. Right now, it's a single number for all tables. *****)
- `numHighScoreTables` ± The number of different tables used in a game.
- `slotType` ± The class used by a game for HighScoreSlots.
- `encryptedPassword` ± The password used to enable table editing for your game. The

default password is ªNONEº.

- (***** In the future there will be values for the max number of slots for a given user on the local/server tables. Right now, it is unlimited. *****)

If you do not wish to use InterfaceBuilder to instantiate a GameInfo object and set it's parameters, then you should override GameBrain's ±`makeGameInfo` method in a subclass of GameBrain.

Another way to customize the high score system is to alter the HighScore.nib file you are using. You can change the cosmetics of the panel which is displayed and you can add or leave out information to taste. Most of the .nibs provided with the GameKit will be adequate; for more information about the required connections in the .nib files, see the spec sheet for the GKHighScorePanel object in the reference manual.

Advanced customizations

There are many customizations which require subclassing other classes besides the GameInfo class. Typically, you will only need to subclass the GK`HighScorePanel` and `HighScoreSlot` objects. This usually occurs when you need to add a piece of information to the high score tables which isn't already stored by the high score slot or if you wish to have the table sorted by some method other than by score.

XXX not finished XXX

Changing sort order of the table

Subclassing HighScoreSlot objects ± XXX not finished XXX

Adding information to the table

Subclassing HighScoreSlot and GKHighScorePanel objects ± XXX not finished XXX

Further information

Many of the finer details of the high score system have been omitted from this chapter. For example, many of the methods used by the various objects are not discussed, nor are the interactions between the interface objects and the `HighScoreController`. For this information, see the class spec sheets in the reference section. Most of the details which have been omitted were left out because they aren't of general use and are better left in the reference section so as to avoid confusing the reader. In general, the information above ought to be enough for most cases and will provide a point of reference for the reader when, if ever, the need for digging into the guts of the objects arises. Most of the actual methods used by the various objects are meant for use by other GameKit objects, and you should never need to worry about them. The only special case is if you are attempting to use the high score system in a custom application which is not using the full GameKit. If this is the case, you should note how the GameBrain, ScoreKeeper, PreferencesBrain, and HighScoreController interact and duplicate it in your application.