3. Scoring system

The GameKit provides a complex mechanism for handling the scoring of a game. One major assumption is made: that you are using a score-based system. Note, however, that if this is not appropriate for your game (say, scoring is by elapsed time, for example) there is no requirement that you make use of the provided scoring system. Since the ScoreKeeper object is the pathway through which GameKit objects affect a player's score, all you have to do is override the GameBrain's default ScoreKeeper object, set up during the ±**appDidInit:** method. Note that simply connecting your custom ScoreKeeper to the GameBrain in InterfaceBuilder™ is sufficient to do this.

The flexibility comes in when you begin to use bonus tracking objects in conjunction with a ScoreKeeper. This allows you to make complex changes in a player's score with little or no programming effort. By using delegates, you can have the scoring system notify other sections of your game when certain special events occur. An obvious example of this would be notification that a score has passed a certain value so that an ªextra guyº may be awarded.

Introduction to the Scoring System

The diagram below shows how a simple GameKit-based scoring subsystem might look: (note that the layout of delegates and bonus trackers is completely dependent upon your game's requirements)

315848_paste.eps ¬

The score system revolves around a few simple ideas. The center of action is the ScoreKeeper object. The ScoreKeeper holds a score for a particular player in a game. In order to change the score, you ask the ScoreKeeper to add or subtract points. The ScoreKeeper also has several bonus ªtrackers.º All a bonus tracker does is provide a point value, based upon some criteria, to the ScoreKeeper. For example, the most common uses are to provide a random-valued bonus to a game or a series of bonus values. (The saucer in NX_Invaders is a random-valued bonus while the fruit and eaten ghost bonuses are both an obvious series, one following a set mathematical formula and the other somewhat arbitrary.) When you wish to add a bonus to a score, you simply ask the ScoreKeeper to do so.

Because the most common and obvious use of delegation is to award extra lives to a player who scores well, the GameKit will automatically set up most of the necessary connections for this function. For a discussion of how to finish this initialization, see the discussion of the PlayerUpView found in Chapter 10 of this tutorial. (Basically, all your need to do is provide an appropriate BonusTracker instance to the PlayerUpView.)

Bonus trackers have one function: supply a series of numbers. *(Note: I may move these to the daymisckit and call them DAYSeries, since really that's what they do¼provide an arbitrary series of numbers.)* Several types of bonus trackers are already supplied with the GameKit. This means that it is very likely that one of the supplied objects will work fine to solve your needs. If not, the BonusTracker is a very easy object to subclass. In fact, the objects supplied with the GameKit are in themselves excellent examples of how to accomplish this.

Delegates are used extensively to notify other objects, external to the score system, about what is happening. Whenever a score is about to change, all the ScoreKeeper's delegates are asked to approve the change. Delegates are notified after a change takes place, as well. This system allows other objects to carry out specific

functions when certain scores are achieved.

The final job of the ScoreKeeper is to keep the ªScoreº TextField in the ªStatisticsº panel up to date. *(Note: Currently, it also updates the ªHighScoresº field, but this functionality will most likely move to the High Score system since that would make a whole lot more sense.)* Whenever the score changes, this field is updated.

The ScoreKeeper object

The ScoreKeeper, as the core of the scoring system, is the object which does most of the work. A GameKit application's GameBrain will automatically create a ScoreKeeper object if one is not already connected to it via InterfaceBuilder. The only time you really need to set up a ScoreKeeper is when you need a custom version of the ScoreKeeper to be used. The GameBrain will forward the ±**appDidInit:** message to the ScoreKeeper, which will cause the ScoreKeeper to connect itself to other GameKit objects as needed. Again, you can override these default connections by connecting them up a priori in InterfaceBuilder.

It is very easy to use a ScoreKeeper object. The simplest use is to simply send ±**addToScore:** and ±**subtractFrom Score:** messages as necessary. Each method takes an integer value as an argument. The ±**resetScore** method will clear the score. This message is automatically sent by the GameKit when a new game begins, so usually you won't need to send it yourself. If you need to know the current score for any reason, the ±**currentScore** method will return an integer: the current score.

Delegation

If your objects need to know about changes to the score, then you should set them up as delegates of the ScoreKeeper. Sending a delegate-to-be to the ScoreKeeper as the argument to ±**addDelegate:** will do this. The ScoreKeeper object on one of the GameKit palettes uses a custom connection inspector in InterfaceBuilder to allow you to connect multiple delegates to the ScoreKeeper from within InterfaceBuilder. *(This palette is not yet implemented -- sorry!)* See the class specification sheet for other methods which allow you to maipulate delegates to the ScoreKeeper. Each delegate of a ScoreKeeper is sent two messages:

±**scoreWillChangeFrom:to:** and ±**scoreChangedFrom:to:**. The first method allows your delegate to keep the score from changing for whatever reason. If any delegate returns a NO, then the score will not be changed. The first delegate to return NO will abort the notification process since it is known that the score in fact will not be changing any more (rendering such a message meaningless). The second method lets you know whenever a score has actually changed. This is the message that should cause events to occur when scores change (extra lives, etc.). You do not have to implement either method in your delegate objects; you only need to implement the methods that you wish to receive.

Bonuses

In order to handle bonuses, you can use BonusTracker objects, or a subclass of BonusTracker. The section below describes how to create and manage a BonusTracker and configure it to fit your needs. In the case of the ScoreKeeper, all you need to do is create the BonusTrackers that you will need. Once this is done, pass them to the ScoreKeeper by sending a ±**addBonusTracker:** message. This message returns an ID for the bonus tracker which you will later use to identify it. *(It might be easier for you if I allow string-valued keys to access these objects¼if you would like this, let me know.)* Once you have installed the BonusTrackers, simply ask the ScoreKeeper to add, subtract, or reset a bonus whenever that would be appropriate. When you add or subtract a bonus from a player's score, you also send a flag to the ScoreKeeper telling it whether or not it should move the bonus tracker to the next number in the series. In most cases, you will want to do this, so that the series advances to the new value each time it is used. The methods needed to add and subtract bonuses are ±**addBonusToScore:advance:** and ± **subtractBonusFromScore:retreat:**. The ±**resetBonus:** method will reset the requested bonus tracker. Requesting a negative bonus tracker will cause *all* the bonus trackers to be reset.

As a concrete example of using a bonus tracker, take the ghosts in PacMan. When a power-dot is eaten, the ghosts' bonus tracker is reset. This starts us out at 200, which is the value of the first ghost. Then, each time the player eats a ghost, a message is sent to the ScoreKeeper to add the ghosts' bonus and then advance it. That's *all* you have to do. (Note that when the power pill runs out, we don't need to reset the bonus tracker since that will happen anyway when the next power pill is eaten.) In a game with a complex scoring system, use of bonus trackers will simplify your code immensely. Note that advanced programmers might want to consider each bonus tracker to be a simple state machine,
which is exactly what it is.

The BonusTracker objects

Here are some examples of where a bonus tracker would be used (taken from the GameKit examples):

- Points for eaten ghosts in PacMan, powers of two multiplied by 100 starting with 200: 200, 400, 800, 1600
- Points for the fruits in PacMan (arbitrary array of values)
- Points for the saucer in NX_Invaders (random number from 1 to 10 multiplied by 100)

Each of the three methods above just happen to require the use of a different BonusTracker class. Each of these classes is included as a part of the GameKit because each might be useful in any type of game. The first type of BonusTracker is the BonusTracker class itself. It has two subclasses, ArrayBonusTracker and RandomBonusTracker, which are also useful. Below, each type of bonus tracker is discussed briefly.

Basic BonusTracker

The BonusTracker class keeps track of bonuses with the following paramters:

- Minimum value: the value of the BonusTracker will never be below this number.
- Maximum value: the value of the BonusTracker will never be above this number.
- Base value: the first value in the BonusTracker's series is always this number.
- Increment: to advance a BonusTracker's value, this number is added to the value.
- Multiplier: to advance a BonusTracker's value, this number is multiplied by the value.

The above parameters allow for fairly flexible construction of series based upon fixed exponents and differences. For example, the PacMan series for the ghosts (200, 400, 800, 1600) is implemented by min=200, max=1600, base=200, increment=0, and multiplier=2. The extra lives awarded in PacMan occur every 15,000 points, with the first at 10,000. This is produced easily with min=10000, max=MAXINT, base=10000, increment=15000, multiplier=1. Note that setting multiplier=0 will make the function non-invertible. Rather than having the retreat attempt a divide by zero, only the subtraction of the increment is performed.

Note that it might be nice to create a subclass which can handle higher order differences and factors or floating point values and multipliers. Currently such BonusTrackers are not supplied with the GameKit. If there is enough demand, such objects will be included. (That would be better than forcing everyone to write their own, regardless of how trivial the task actually is¼)

RandomBonusTracker

A RandomBonusTracker is sensitive to the above parameters, but does not advance or retreat. Instead, each time its value is requested, a random number between min and max is returned. The number returned will be equal to min plus some multiple of increment. This allows the random number to be ªgranular.º For example, flying saucers in NX_Invaders are worth a number of points which is a multiple of 100 and between (and including) 100 and 1000. (That is, 100, 200, 300, ¼, 1000.) Simply, min=100, max=1000, and increment=100. Note that interesting effects can be obtained by making increment and min different numbers.

ArrayBonusTracker

In some cases, the bonuses needed will follow no obvious mathematical model. In such cases, the above bonus trackers are not very useful. The ArrayBonusTracker is designed to handle such cases, where you just throw your hands in the air and compile a list of values into an array. All you do is pass an array of values and its length to the ±**initForBonuses:count:** method and then let the ArrayBonusTracker do its thing. Any arbitrary series can be represented this way quite easily. In the case of the fruit bonuses in PacMan, this object becomes necessary. Here is the array used to define the series:

```
static long fruitval[NUMFRUITVALS] = { 100, 200, 300, 300, 500, 700, 700,
                      1000, 1000, 2000, 2000, 3000, 3000, 3000, 3000, 5000 };
```

You'll note that there is no particular formula which would apply to this series. Now, simply initialize the ArrayBonusTracker as so, and you're done:

```
                          id fruitTracker = [[ArrayBonusTracker alloc] initForBonuses:fruitval
count:NUMFRUITVALS];
```

Two notes are important to remember: (1) it is OK to free the array that you pass to the ArrayBonusTracker. It creates a copy for its own internal use. (2) You can advance and retreat the tracker all you want. Rather than

go past the end of the array, it will ªstickº at the highest or lowest index, whichever is appropriate. *(A future version might allow the pointer to ªwrap aroundº as an alternate behavior, depending upon the setting of an appropriate flag. Would this be useful to anyone?)*

Subclassing

If none of the above BonusTracker classes will produce the series that you need, then you should create your own subclass which does. The easiest way to create a new BonusTracker is to override the ±**bonusValue** method. This is the approach taken by the RandomBonusTracker. It uses the **max**, **min**, **increment**, and **baseValue** parameters of the BonusTracker to determine the constraints upon the random numbers that it generates.

You could alternatively override the ±**advanceBonus** and ±**retreatBonus** methods. In this case, you exit the method of choice after setting the instance variable **value** to the new value of the BonusTracker. This is the way the the ArrayBonusTracker works. For most of the subclasses that you will create, the latter method will be more appropriate. Note that the existence of the advancing and retreating methods assumes that the series can be generated by an invertible function. If this is not the case, then the ±**retreatBonus** method should be overridden to do nothing.

No matter how you make your subclass, you will probably find that some of the standard BonusTracker parameters do not apply, whereas there will be a need for new parameters which are not provided by the BonusTracker superclass. All you need to do is provide the new methods and instance variables as necessary and ignore the parameters that are not needed by your function.

It is highly recommended that you peruse the source code for the ArrayBonusTracker and RandomBonusTracker classes to see examples of how you might change the function implemented by a BonusTracker. Note that currently BonusTrackers do not do any form of delegation. If you feel that this might be useful, make a suggestion to the author. Of course, it would help if you could describe a situation where this might be useful.