

2

Symbian OS Game Basics

Jo Stichbury

2.1 Introduction

This chapter describes some of the basic issues to consider when creating a game on Symbian OS v9 in C++. We discuss how to write a standard game loop using an active object timer, how to handle user input from the keypad and screen, and discuss how to pause the game when interrupted by a change in application focus. Some of the resource limitations associated with the mobile platform (such as memory, disk space and floating point math support) are also discussed.

A sample game skeleton, imaginatively called ***Skeleton***, which illustrates the points discussed, is available for download from ***developer.symbian.com/gamesbook*** and runs on both UIQ 3 and S60 3rd Edition emulators and hardware (it was tested on Sony Ericsson M600i, Motorola Z8, Nokia N73 and E61). The code uses the basic application framework for each UI platform generated using the New Project wizard available with Carbide.c++ v1.2. Very little additional UI-specific code has been added, because the basics described here use generic Symbian APIs. However, where differences exist between the platforms, they are described below.

2.2 The Game Loop

Most applications on Symbian OS are driven by user input, directly or indirectly. For example, the Calendar application displays text as a user types it, or responds to commands the user submits through the UI (such as formatting text, or creating and deleting appointments). If the

user stops interacting with the application, the thread is suspended and only continues executing on receipt of, for example:

- an event generated when the user next does something, for example, by pressing a key or tapping the screen
- a system event such as a timer completion
- a notification of a change of focus, such as when the user switches to use another application, or the application is sent into the background by an incoming call or other event. For instance, the Calendar application may have an outstanding timer that expires when the application needs to display a notification dialog to the user to remind them of their next appointment.

The Symbian OS application model is said to be event-driven, which means that it does not run on a tight polling loop, constantly checking for input or changes of system state, but instead waits to be notified to run when it needs to respond to an event. This makes the threads that do need actually need to run more responsive, because they don't have to compete for a time slice. And when no threads need to run, Symbian OS can enter a power-saving state where all threads are suspended, thus optimizing the battery life.

Games, and other applications with constantly changing graphics, have different requirements to the event-driven applications described above, because they must continue to execute regularly in order to update their graphics and perform other calculations (such as detecting collisions or displaying countdown timers on their screens). For a game to execute regularly, regardless of the user input received, it is driven by what is known as a *game loop*, which is a loop that runs regularly for the duration of the game.

Typically, each time the game loop runs:

- The time elapsed since the last time the game loop executed is calculated, and any user input that has occurred since then is retrieved.
- The game engine updates the game internals, taking into account factors such as the time elapsed and the user input received.

For example, in a game like *Asteroids*,¹ the user's key presses are used to calculate the new position of the player's ship, based on the actual input received and the time that has passed since the last graphics update. Besides making calculations to correspond to the player's input, the game engine also computes the new positions of all other moving graphics elements relative to the time of the last redraw. The

¹ See developer.symbian.com/roidsgame for a paper which describes writing an *Asteroids* clone for Symbian OS v9.

game engine must determine if there have been any collisions between the graphics elements and calculate their resulting status, such as their visibility (Have they moved off screen? Is one asteroid overlapping the others? Has a collision broken an asteroid into fragments?), and their changes in shape or color. The game engine also calculates status values, such as the player's score or the ammunition available.

- The screen is updated according to the new state calculated by the game engine.
Having calculated the internal game state to reflect the new positions and status of each game object, the game engine prepares the next image to display on the screen to reflect those changes. Once the new frame is prepared, it is drawn to the screen. Each time the screen is updated, the new image displayed is known as a frame. The number of frames written to the screen per second is known as the frame rate. The higher the number of frames per second (fps), the smoother the graphics transitions appear, so a desirable frame rate is a high one in games with rapidly changing graphics. However, the exact figure depends on the game in question, affected by factors such as the amount of processing required to generate the next frame and the genre of game, since some games, such as turn-based games, do not require a high frame rate.
- The game's audio (background music and sound effects) are synchronized with the game state.
For example, the sound of an explosion is played if the latest game update calculated that a collision occurred.

Figure 2.1 shows this graphically, while in pseudocode, a basic game loop can be expressed as follows:

```
while ( game is running )
{
    calculate time elapsed and retrieve user input
    update game internals
    update graphics (prepare frame, draw frame)
    synchronize sound
}
```

Note that a constantly running game loop is not necessary in every game. While it is needed for a game with frequently changing graphics in order to update the display, some games are more like typical event-driven applications. A turn-based game, like chess, may not need a game loop to run all the time, but can be driven by user input. The loop starts running when a turn is taken, to update the graphics to reflect the move, and calculate the AI's response, then returns to waiting on an event to indicate the user's next move or other input, such as deciding to quit the game.

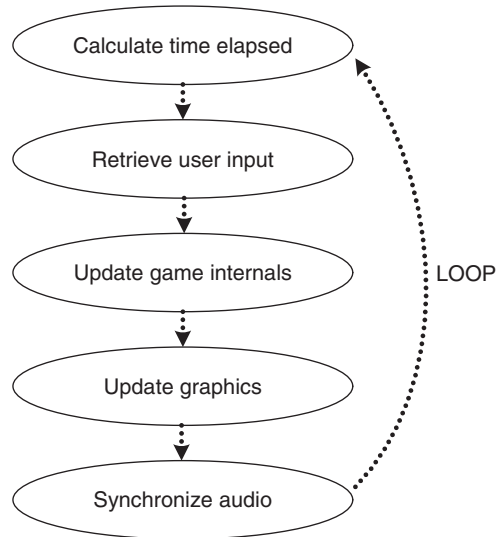


Figure 2.1 A graphical representation of a game loop

2.3 The Heartbeat Timer

On Symbian OS, it is important not to hog the CPU so you should not run a tight synchronous loop like the one shown in pseudocode above. The game loop must return control to the thread's active scheduler to allow other system events to run as necessary, allowing other threads in the system the opportunity to run. If the game loop prevents the active scheduler from running a particular active object associated with the view server regularly, between 10 and 20 seconds, the infamous `EVwsView-EventTimeout` panic occurs. This terminates the game thread with panic category `ViewSrv` and error code 11.

So instead of trying to work around this, on Symbian OS, a game loop should be run using a timer active object, such as `CTimer` or `CPeriodic`, to trigger the game loop to run regularly. It is better to use a single timer to run one iteration of the game loop than to use multiple timers to control different time-based events in the game. The latter approach can lead to very complex code, since there may be many elements in a game that respond after different timeouts have expired, which would be difficult to synchronize and to debug. Active objects cannot be pre-empted, so the accuracy of a timer active object is also dependent on other active objects in the system. If multiple timers are used for different events, the combination could lead to significant timer drift for some lower-priority active objects. Additionally, each timer consumes kernel resources – if a large number are required, this could add up to an undesirable system overhead.

The solution is to use a single timer object to provide a game ‘heartbeat.’ All elements of the game that require a particular time-based activity use the heartbeat to synchronize themselves. The timer fires as many times per second as necessary to achieve the desired frame rate. Each time the timer fires, it runs the game loop once, and increments a stored value known as the tick count. The tick count is used to provide the time elapsed to the rest of the game.

The resulting pseudocode for the game loop looks as follows, where `TimerEventHandler()` is the method called once every heartbeat to handle the timer event:

```
TimerEventHandler()
{
    calculate the time elapsed and retrieve user input
    update game internals
    update graphics (prepare frame, draw frame)
    synchronize sound
}
```

One of the benefits of using a single heartbeat timer is that the beat rate can easily be slowed down for debugging purposes, stopped entirely when the game needs to pause, or tweaked to modify the frame rate when the graphics performance differs across a range of devices with different capabilities.

The `CPeriodic` timer is used most commonly to drive a game loop. It uses a callback function to execute the game loop on each heartbeat. `CPeriodic` is the simplest class to use to provide a timer without coming into direct contact with active objects because it hides the details of the active object framework (it derives from `CTimer` – an active object-based timer class – and implements the active object methods `RunL()` and `DoCancel()` for its callers). The *Skeleton* example uses this class, as shown below. However, if you already use a number of active objects in your game, you may simply prefer to use the `CTimer` class directly.

```
void CSkeletonAppView::ConstructL( const TRect& aRect )
{
    ...
    // Create the heartbeat timer for the game loop.
    iPeriodicTimer = CPeriodic::NewL(CActive::EPriorityStandard);
    // Activate the window (calls StartHeartbeat())
    ActivateL();
}

void CSkeletonAppView::StartHeartbeat()
{
    // Start the CPeriodic timer, passing the Tick() method for callback
    if (!iPeriodicTimer->IsActive())
        iPeriodicTimer->Start(KFramesPerSecond, KFramesPerSecond,
                              TCallback(CSkeletonAppView::Tick, this));
}
```

```

// Tick() callback method
TInt CSkeletonAppView::Tick(TAny* aCallback)
{
    ASSERT(aCallback);
    static_cast<CSkeletonAppView*>(aCallback)->GameLoop();
    // Return a value of ETrue to continue looping
    return ETrue;
}

void CSkeletonAppView::GameLoop()
{
    // Update the game internals - details omitted here
}

```

As I mentioned above, because Symbian OS active objects cannot be pre-empted, if another active object event is being handled when the timer expires, the timer callback is blocked until the other event handler completes. The timer's event processing is delayed, and the timer request is not rescheduled until it runs. This means that periodic timers are susceptible to drift, and will give a frame rate slightly lower than expected. Additionally on Symbian OS v9, the standard timer resolution is 1/64 seconds (15.625 milliseconds) on both hardware and the emulator. So, for example, if you request a period of 20 milliseconds, you actually receive an event approximately once every 30 milliseconds (that is, 2×15.625 milliseconds). What's more, if you've just missed a 1/64th 'tick' when you submitted the timer, then it'll only be submitted on the next tick, which means that the first period will be almost 3×15.625 (=48.875) milliseconds. This can add up to a significant jitter.

As an example, in the *Skeleton* example, I set the timer period to be 1/30 second (33.3 milliseconds), which at first sight could be expected to deliver 30 fps. The frame rate actually observed was relatively constant at 22 fps occasionally dropping to 21 fps. To deliver 30 fps, I would need to increase the period and add in logic to see how much time has passed between each tick, and then compensate for it when rescheduling the timer.

Symbian OS does actually provide a heartbeat timer class, called `CHeartBeat` to do this. This class is similar to the periodic timer, except that it provides a function to restore timer accuracy if it gets out of synchronization with the system clock. The `CHeartBeat` class accommodates the delay by calling separate event-handling methods depending on whether it ran accurately, that is, on time, or whether it was delayed, to allow it to re-synchronize. However, this additional functionality is not usually necessary. Extremely accurate timing isn't generally required because a game engine only needs to know accurately how much time has elapsed since the previous frame, so, for example, the total amount of movement of a graphics object can be calculated.

There is also a high resolution timer available through the `CTimer` class, using the `HighRes()` method and specifying the interval required

in microseconds. The resolution of this timer is 1 ms on phone hardware, but defaults to 5 ms on the Windows emulator, although it can be changed by modifying the `TimerResolution` variable in the `epoc.ini` emulator file. However, on Windows, there is no real-time guarantee as there is on the smartphone hardware, because the timings are subject to whatever else is running on Windows.

2.4 Handling Input from the Keypad

Handling user input made by pressing keys on the phone's keypad is quite straightforward in the Symbian OS application framework. The application UI class should call `CCoeAppUI::AddToStackL()` when it creates an application view to ensure that keypad input is passed automatically to the current view. The key events can be inspected by overriding `CCoeControl::OfferKeyEventL()`.

Note that only input from the numerical keypad and multi-way controller is passed into `CCoeControl::OfferKeyEventL()`. Input from the user's interaction with the menu bar, toolbar, or softkeys, is passed separately to the `HandleCommandL()` method of the application UI class for S60. In UIQ, the application view handles in-command input. This is because a command can be located on a softkey, in a toolbar, or in a menu, depending on the interaction style of the phone. To allow for this flexibility, commands are defined in a more abstract way, rather than being explicitly coded as particular GUI elements, such as menu items.

For either UI platform, the input event is handled according to the associated command, which is specified in the application's resource file. The ***Skeleton*** example demonstrates basic menu input handling in `CSkeletonAppUi::HandleCommandL()` for S60 and `CSkeletonUIQView::HandleCommandL()` for UIQ.

In the ***Skeleton*** example, the handling of the user input is decoupled from the actual event, by storing it in a bitmask (`iKeyState`) when it is received, and handling it when the game loop next runs by inspecting `iKeyState`. This approach is shown in the simplified example code, for S60, below.

```
TKeyResponse CSkeletonAppView::OfferKeyEventL(const TKeyEvent&
                                               aKeyEvent, TEventCode aType)
{
    TKeyResponse response = EKeyWasConsumed;
    TUint input = 0x00000000;

    switch (aKeyEvent.iScanCode)
    {
        case EStdKeyUpArrow: // 4-way controller up
            input = KControllerUp; // 0x00000001;
            break;
        case EStdKeyDownArrow: // 4-way controller down
```

```

        input = KControllerDown; // 0x00000002;
        break;
    case EStdKeyLeftArrow: // 4-way controller left
        input = KControllerLeft; // 0x00000004;
        break;
    case EStdKeyRightArrow: // 4-way controller right
        input = KControllerRight; // 0x00000008;
        break;
    case EStdKeyDevice3: // 4-way controller center
        input = KControllerCentre;
        // This is the "fire ammo" key
        // Store the event, the loop will handle and clear it
        if (EEventKey == aType)
        {
            iFired = ETrue;
        }
        ...
    default:
        response = EKeyWasNotConsumed; // Keypress was not handled
        break;
    }

    if (EEventKeyDown == aType)
    { // Store input to handle in the next update
        iKeyState |= input; // set bitmask
    }
    else if (EEventKeyUp == aType)
    { // Clear the input
        iKeyState &= ~input; // clear bitmask
    }

    return (response);
}

```

`TKeyEvent` and `TEventCode` are passed in as parameters to `OfferKeyEventL()`, and contain three important values for the key press – the event type, scan code, and character code for the key. Let's discuss these in some further detail.

2.4.1 Key Events

Key events are delivered to the application by the Symbian OS window server (WSERV). WSERV uses a typical GUI paradigm which works well for controls such as text editors and list boxes. That is, when a key is held down, it generates an initial key event, followed by a delay and then a number of repeat key events, depending on the keypad repeat rate setting, until it is released.

There are three possible key event types. Each key press generates the three event types in sequence:

1. `EEventKeyDown` – generated when the key is pressed down

2. `EEventKey` – generated after the key down event, and then regularly as long as the key is kept down²
3. `EEventKeyUp` – generated when the key is released.

The `EEventKeyDown` and `EEventKeyUp` pair of events can be used to set a flag in the game to indicate that a particular key is being held, and then clear it when it is released, as shown in the previous example. Using this technique, the state of the relevant input keys is stored, and key repeat information can be ignored (the repeated input notification often does not work well for games, because it results in jerky motion, for example, when a user holds down a key to drive a car forward in one direction). However, it can be useful in other contexts. Since the `EEventKey` event is passed repeatedly if a key is held down, it can be used for repeating events, such as firing ammunition every quarter second while a particular key is held down. This is much easier and more comfortable for the player than needing to repeatedly press the key. The ***Skeleton*** example code, shown above, illustrates this each time a repeat event is received from holding down the centre key of the 4-way controller.

2.4.2 Key Codes

Either the scan code or the character code can be used by the game to determine which key has been pressed. The scan code (`TKeyEvent::iScanCode`) simply provides information about the physical key pressed, which is usually sufficient for processing events in games, since all that is required is information about the direction or the keypad number pressed. The character code may change when, for example, a single key is pressed multiple times, depending on how the mappings and any front end processors are set up. This level of detail isn't often needed by a game, so the character code is generally ignored.

A game must also be able to handle the case where two or more keys are pressed simultaneously, for example, to simulate 8-way directional control or allow for more complicated key sequences and controls. The ***Skeleton*** example does this by storing input in a bitmask, so multiple simultaneous key presses can be handled together by the loop.

By default, pressing multiple keys simultaneously is not possible; the event from the first key to be pressed is received and others are discarded while it is held down. This is called key blocking. However, the application UI can call `CAknAppUi::SetKeyBlockMode()` on

² `RWsSession::GetKeyboardRepeatRate()` can be used to determine the repeat interval and it can be modified by calling `RWsSession::SetKeyboardRepeatRate()`.

S60 to disable key blocking for the application and accept multiple key press events. However, this is not possible on UIQ.

One other thing to note about input is that phone keypads have a limited number of keys, which are not always ergonomic for game play. The variety of layouts and the size limitations need to be considered when designing a game, and ideally should be configurable, through a menu option, so the user can choose their own, according to their preference and the smartphone model in question. It is also a good idea to design games to always allow for one-handed play, since some devices are too small for game playing to be comfortable using both hands. However, other devices are suited to two-handed game playing, particularly when they can be held in landscape orientation, such as the Nokia N95 or the Nokia N81. The instance where phones can be used in both portrait and landscape orientations, and the game supports each mode, is another good reason for allowing the game keys to be configurable. The user can set different key configurations for gameplay depending on the orientation of the phone.

2.5 Handling Input from the Screen

Some UIQ smartphones, such as the Sony Ericsson P1i, support touch-screen input, which can be used to add another mode of user interaction to a game. Screen events can be detected and handled in much the same way as keypad input, by overriding `CCoeControl::HandlePointerEventL()` as shown below. In the example given, to capture events that occur in the close vicinity of a particular point on the screen, an area of the screen is constructed, ten pixels square, and centered on the point in question. When a pointer-down screen event occurs, if the point at which the tap occurred lies within the square, the user input is stored in a bitmask in the same way as for a key press.

```
void CSkeletonUIQView::HandlePointerEventL(const TPointerEvent&
                                           aPointerEvent)
{
    if (TPointerEvent::EButton1Up == aPointerEvent.iType)
        { // Pointer up events clear the screen
            iKeyState = 0x00000000; // clear all previous selections
        }
    else if (TPointerEvent::EButton1Down == aPointerEvent.iType)
        {
            TRect drawRect( Rect());
            TInt width = drawRect.Width();
            TInt height = drawRect.Height();

            TPoint offset(10,10); // 10x10 square around the screen position

            TPoint k1(width/4, height/2);
```

```

TRect rect1(TPoint(k1-offset), TPoint(k1+offset));
if (rect1.Contains(aPointerEvent.iPosition))
{
    iKeyState |= KKey1;
    return; // stored the event, so return
}

TPoint k2(width/2, height/2);
TRect rect2(TPoint(k2-offset), TPoint(k2+offset));
if (rect2.Contains(aPointerEvent.iPosition))
{
    iKeyState |= KKey2;
    return; // stored the event, so return
}
... // Other numbers similarly inspected
}

// Pointer events for other areas of the screen are ignored
// Pass them to the base class
CControl::HandlePointerEventL(aPointerEvent);
}

```

Figure 2.2 illustrates the *Skeleton* example in the UIQ emulator. The mouse is clicked and held on, or near, the '8' digit to simulate a tap and hold on the screen. The event is detected through `CSkeletonUIQView::HandlePointerEventL()` as shown above for digits '1' and '2.' The `iKeyState` member variable is updated to reflect the screen event, and next time the game loop runs and the screen is updated, the display for that digit highlights that the stylus is held to the screen. When the stylus (or mouse click, in the case of the emulator) is released, a pointer-up event is received and the highlight is removed.



Figure 2.2 Handling a pointer event received from a screen tap

2.6 System Events

As we've discussed, the game loop is driven by a timer active object and runs regularly to update the game engine internals, resulting in a given number of frames per second. For any length of time, on a mobile operating system, this kind of regular looping can lead to a drain in battery power. The requirement to regularly run the game thread prevents the OS from powering down all but the most essential resources to make efficient use of the battery.

2.6.1 Loss of Focus

As a developer, you need to be aware that the game is consuming battery power whenever it is running the loop, and ensure that the heartbeat timer runs only when required. For example, the game loop should be paused when the game is interrupted by a system event, such as an incoming call, or because the user has tasked to another application. The game loop should also pause if the user stops interacting with the game for any length of time; for example, if she is playing the game on the bus and has to get off at her stop without quitting the game before doing so.

You should implement `CCoeControl::FocusChanged()` to be notified of a change in focus of the game – that is, when another application or system dialog sends the game to the background. As the example shows, the timer is halted when the focus is lost, and started when it is regained. The benefit of having a single heartbeat time to drive the game loop is that it is very straightforward to stop and start it when necessary.

```
void CSkeletonAppView::FocusChanged(TDrawNow aDrawNow)
{
    if (IsFocused())
        { // Focus gained
        StartHeartbeat(); // Starts the gameloop
        }
    else
        { // Focus lost
        if (iPeriodicTimer->IsActive())
            {
                StopHeartbeat(); // Pauses the game loop
            }
        // Save game data here if necessary
        }

    if(aDrawNow)
        DrawNow();
}
```

2.6.2 User Inactivity

If no user input is received, after a certain length of time, to minimize the phone's power usage, the system will gradually dim the backlight, and eventually turn it off completely. The length of inactivity time before each

step occurs is often customizable by the phone's owner, and usually, a screensaver is displayed, such as the time and date, which can also be customized.

Regardless of the fact that the phone appears to be in power-saving mode, the game loop will continue running, even when the screensaver is showing and the backlight is off, unless the developer adds code to explicitly stop it. To save power, it is usual practice for the game to move into 'attract' mode, or to display a pause menu and stop the game loop after user inactivity of a certain duration.

Attract mode is so called because it displays a graphics sequence which looks something like the game running in a demo mode, showing off the game to attract the user to play it. Attract mode also consumes power, because it is still displaying regularly updated graphics and running a game loop, even if the frequency of the timer is much reduced from that of normal gameplay. Unless the phone is running on main power (i.e., connected to its charger), it is usual to run the attract mode for a pre-determined amount of time only, perhaps two minutes, and then display the pause menu and halt the game loop by completely stopping the heartbeat timer.

User inactivity can be detected by creating a class that inherits from the `CTimer` active object and calling the `Inactivity()` method. The object receives an event if the interval specified, passed as a parameter to the method, elapses without user activity.

```
class CInactivityTimer : public CTimer
{
public:
    static CInactivityTimer* NewL(CSkeletonAppView& aAppView);
    void StartInactivityMonitor(TTimeIntervalSeconds aSeconds);
protected:
    CInactivityTimer(CSkeletonAppView& aAppView);
    virtual void RunL();
private:
    CSkeletonAppView& iAppView;
    TTimeIntervalSeconds iTimeout;
};

CInactivityTimer::CInactivityTimer(CSkeletonAppView& aAppView)
: CTimer(EPriorityLow), iAppView(aAppView)
{
    CActiveScheduler::Add(this);
}

CInactivityTimer* CInactivityTimer::NewL(CSkeletonAppView& aAppView)
{
    CInactivityTimer* me = new (ELeave) CInactivityTimer(aAppView);
    CleanupStack::PushL(me);
    me->ConstructL();
    CleanupStack::Pop(me);
    return (me);
}
```

```
void CInactivityTimer::StartInactivityMonitor(TTimeIntervalSeconds
                                             aSeconds)
{
    if (!IsActive())
    {
        iTimeout = aSeconds;
        Inactivity(iTimeout);
    }
}

void CInactivityTimer::RunL()
{
    // Game has been inactive - no user input for iTimeout seconds
    iAppView.SetPausedDisplay(ETTrue); // Displays the pause screen
    iAppView.StopHeartbeat(); // Stop the game loop
}
```

The `CInactivityTimer::StartInactivityMonitor()` method must be called each time the game loop starts, to monitor for user inactivity while it is running.

Following any kind of pause, when the user returns to the game and it regains focus, the game should remain in the paused state until the user makes an explicit choice to proceed, quit, or perform another action, depending on what the pause menu offers.

The usability of the pause menu is a hot topic – it's been said that you can tell the quality of a mobile game from how easy it is to resume from the pause menu. This aspect of game design is outside the scope of this book, but you can find an interesting paper called *At The Core Of Mobile Game Usability: The Pause Menu* in the Usability section of the Forum Nokia website (www.forum.nokia.com/main/resources/documentation/usability).³ This section has a number of other useful documents for mobile game developers and designers, and I recommend that you browse them regularly, even if you are developing a mobile game for the other Symbian OS UI platforms.

2.6.3 Simulating User Activity

There are occasions when the game may want to simulate activity. For example, while attract mode is playing, it is desirable to prevent the system from dimming the backlight, the screensaver taking over, and eventually the backlight turning off completely. This may seem contradictory, but often games will detect user inactivity, move to attract mode, and then actually simulate user activity while attract mode is running, so the graphics can be seen. User activity is simulated by calling

³ The exact location of the paper has a URL which is far too long to type in from a book, and is subject to change anyway, if the content is updated in future. We will keep a set of links to papers and resources that are useful to game developers on the Symbian Developer Network Wiki (developer.symbian.com/wiki/display/academy/Games+on+Symbian+OS).

`User::ResetInactivityTime()` regularly, in the game loop. This is unnecessary while the game is being played, because the user is generating input. It should also cease when the game changes modes to display the pause menu, to then allow the system to switch off the backlight and display the screensaver as it normally would do.

Since simulating user activity during attract mode prevents the system from saving power by dimming its backlight, the length of time the game displays its attract mode should be limited, as section 2.6.2 described, unless the phone is powered externally by the main charger. The `CTelephony` class can be used to retrieve information about whether the phone is connected to a main charger, by calling `CTelephony::GetIndicator()`. To use `CTelephony` methods, you must link against `Etel3rdParty.lib`, and include the `ETel3rdParty.h` header file.

2.6.4 Saving Game Data

When the game is paused, it should save any important game state in case it is needed to continue the game in the state it was in before pausing. For example, the player may re-boot the phone before the game is next played, but it should still be able to continue the game from the point before it was paused. A player who has got to a particular level in a game will not be happy if their progress is lost because they had to interrupt their session.

A game should always save its data at regular intervals so it can be restored to the same state, should the game suddenly be stopped, for example if the battery power fails or the battery is removed completely. In the event of a 'battery pull,' the game's data manager code should be prepared for the possibility that the data has been corrupted – such as if an update was under way when the power down occurred. The game should be able to recover from finding itself with corrupt data, for example, by having a default set available in replacement, or performing data updates using rollback.

When saving game data, it is important to consider the amount of information that should be written to file. If there is a large amount of data, for example over 2 KB, it is advisable not to use the blocking `RFile::Write()` method, since this could take some time to complete and would block the game thread. It is advisable to use an active object to schedule the update, using the asynchronous overload of `RFile::Write()`, or to use one of the higher-level stream store APIs.

2.7 Memory Management and Disk Space

Mobile devices have limitations on the amount of memory (RAM) they can support for a number of reasons, including size (the limitations of the

physical space available), cost, and current draw (the amount of power required to access the memory). Symbian OS is a multitasking operating system, meaning that other applications are running alongside the game. So, because the memory resources available are finite, the available memory may suddenly be very limited.

It would make a bad user experience if, halfway through shooting an enemy, the game crashed or closed because there was insufficient memory to continue. It is important, therefore, for a game to reserve all the memory it will need at start up, so it can guarantee to satisfy all memory allocation requests. If the memory cannot be reserved, the game startup should fail and ask the user to shut down other applications before trying to play.

It is quite straightforward to allocate a pool of memory to a game, by setting the *minimum* size of its heap using the `EPOCHEAPSIZE` specifier in its MMP file.⁴ If the game process succeeds in launching, then this amount of memory is available to the heap – for all memory allocations in the main thread, or any other thread within the process, as long as it uses the same heap. This approach makes writing code that allocates heap-based objects within the game quite straightforward – you don't have to check that the allocation has succeeded in release builds, as long as you profile and test every code path to ensure that, internally, the game does not use more memory than was pre-allocated to it.

However, if your game launches any separate processes (e.g., Symbian OS servers) or places heavy demands on those already running (for example, by creating multiple sessions with servers, each of which uses resources in the kernel) then this approach will not be sufficient to guarantee the game will run if the phone's free memory runs low. Again, the game should attempt to allocate the resources it needs, including those in other processes, at startup – or have a means of graceful failure if this is not practical and a later resource allocation fails.

When pre-allocating memory for the game, it is important to make an educated guess at what is required, rather than simply reserve as much of the memory as is available on the device. Games for the N-Gage platform, for example, which are typically the highest quality rich content games available on Symbian OS v9, limit themselves to a maximum allocatable heap size of 10 MB.⁵ So if you are writing a basic casual game, you should certainly try to stay well under this limit! Your game will not be popular if it prevents normal use of the phone, such that it

⁴ The minimum size of the heap specifies the RAM that is initially mapped for the heap's use. By default it is set at 4 KB. The process can then obtain more heap memory on demand until the maximum value, also set through use of the `EPOCHEAPSIZE` specifier, is reached. By default, the maximum heap size is set at 1 MB.

⁵ N-Gage games are tested and certified prior to release (as we'll describe in Chapter 8) and the 10 MB limit is enforced throughout the launch, execution and termination of the game.

cannot run in parallel with other applications. Since many applications for Symbian OS v9 require Symbian Signing, which verifies the general behaviour and ‘good citizenship’ of the application, you need to stay within self imposed limits. You should also consider advice and best practice for memory usage on Symbian OS, including regular testing for memory leaks, for example, by using debug heap checking macros like `__UHEAP_MARK`. Wherever possible, de-allocate and reuse memory that is no longer required to minimize the total amount your game needs.

Some developers create memory pools for managing their memory allocations in order to minimize fragmentation of the heap, use memory more efficiently, and often improve performance. A common approach is to create a fixed-size pool for persistent data and then use a separate pool for transient data, which is recycled when it is no longer needed in order to minimize fragmentation.

You can find more information about working efficiently with limited memory in most Symbian Press C++ programming books, and general advice in *Small Memory Software: Patterns for Systems with Limited Memory* by Noble and Weir, listed in the References chapter at the end of this book. There is also a set of tools available for checking memory usage and profiling code on Symbian OS. You can find more information about these from the Symbian Developer Network (developer.symbian.com).

Disk space refers to the non-volatile shared memory used to store an installed game, rather than the main memory used to store the game’s modifiable data as it executes. Some Symbian smartphones, like the Nokia N81 8 GB and N95 8 GB have large amounts of space (8 GB) for internal data storage, but the typical internal storage size for other Symbian OS v9 smartphones can be as limited as 10 MB. However, users can extend their storage with removable drives such as mini SD or micro SD cards or Memory Sticks. The drive letter for the removable media may vary on different handsets and UI platforms (for example, the removable media is sometimes the D: drive and sometimes the E: drive). You can use the `RFs::Drive()` method to determine the type of drive, and `RFs::Volume()` to discover the free space available on it.

When installing a game, there must be sufficient disk space to contain the installation package for the game, and subsequently decompress and install the game and its asset files. It is desirable to keep a game footprint – the storage space it occupies – as small as possible if it is to be downloaded over-the-air (OTA), since a very large game will be more costly and slower to download over a phone network.

2.8 Maths and Floating Point Support

Games that require a number of physics-heavy calculations (for example, those modeling acceleration, rotations, and scaling) typically use floating

point numbers. Most current ARM architectures do not support floating point instructions in hardware, and these have to be emulated in software.⁶ This makes the use of floating point numbers expensive in terms of processing speed when thousands of operations are required. Where possible, calculations should instead be implemented using integers or fixed-point arithmetic, or they should be performed in advance on a PC and stored in static lookup tables to be used at run time in the game. It is rare that high precision is required for geometric operations such as sin, cos, and square root, so there is room for optimization, for example, for sin and cos, the Taylor series can be used to approximate sin and cos to a number of decimal places.

The use of 32-bit integers should be preferred where possible; 64-bit integers have run-time performance and memory consumption overheads, and are rarely needed in practice.

2.9 Tools and Further Reading

This chapter has discussed some of the basic aspects of creating a well-behaved game loop, handling input, managing interruptions and user inactivity, and coping with resource limitations.

Where the performance of the game is critical, we advise you to analyze the code for each device it is to run on, since devices and platforms can have very different characteristics. A good tool for performance analysis is the Performance Investigator, delivered with Carbide.c++ Professional and OEM Editions. A profiling tool, it runs on an S60 3rd Edition device and collects information about processes, memory, CPU usage, and battery usage. Within Carbide.c++, the trace data collected can be reviewed and analyzed. More information, and a white paper that describes the Performance Investigator tool, giving an example of how to use it, can be found at www.forum.nokia.com/carbide.

For more information about general C++ development on Symbian OS, we recommend the Symbian Press programming series, described in the References and Resources section of this book. There are also a number of papers available on the developer websites listed in that section, including a discussion forum specifically for C++ games development for S60 on Forum Nokia at discussion.forum.nokia.com/forum.

⁶ Some of the high-end phones based on ARM11 processors do have floating point support, for example, the Nokia N95. To use it, code must be compiled with a particular version of the RVCT compiler, with floating point hardware support options enabled. The resultant binaries do not execute on hardware without floating point support.