06: Active Objects

Exercise Instructions

Goal

To make the subject of active objects more interesting, this exercise is based on a small console-based action game. You will have to write two active objects that provide the game engine with input.

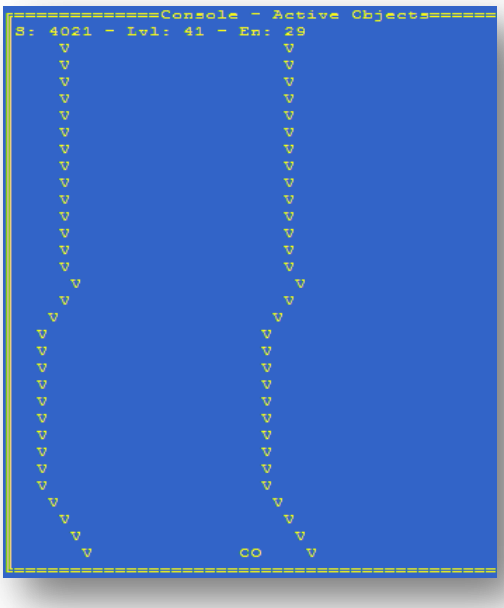
Introduction

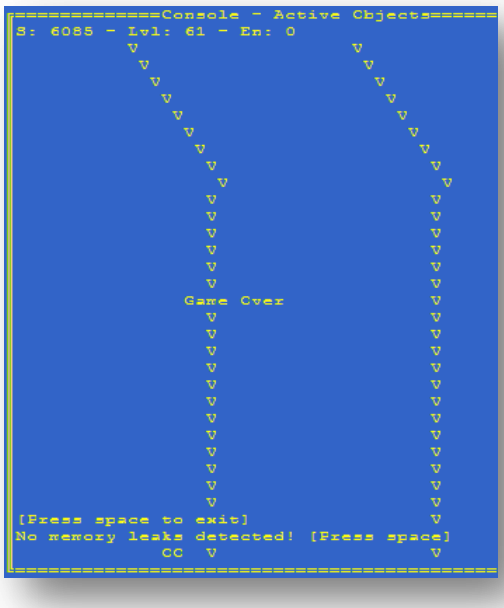
In the game of this exercise, you're driving on a moving street and have to make sure that you keep on it. The longer you make it, the more points you get – but the faster the game will be. If you leave the street, you'll lose energy. When you've got no energy left, the game is over.

Through using the text mode of the Symbian OS console, this exercise is a good introduction to Active objects. You will have to write two of them to make the game work:

- The update active object for regular updates & redraws of the game screen.
- The key listener active object that asynchronously waits for key presses.

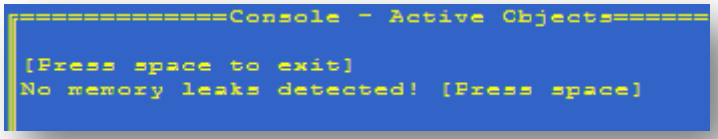
Two screenshots of the game are presented below. In the top line you can see the status bar, which displays the current score, level and energy. The street boundary is marked by the "V" signs; the player is the "OO" symbol at the bottom of the screen. Control the game by using the left/right keys of your keyboard / the joystick in the emulator.



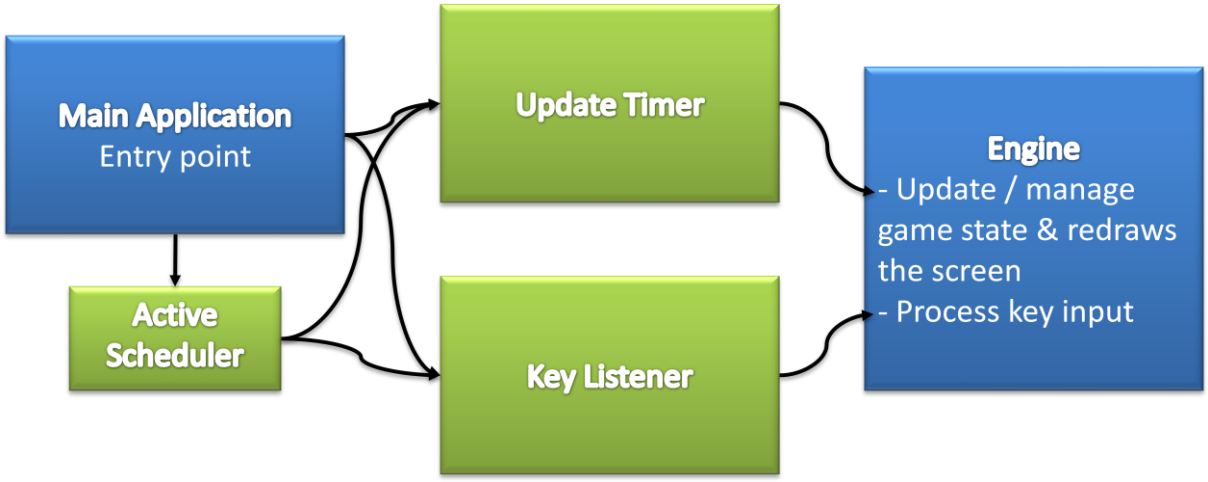


Structure of this Exercise

When you start this exercise, you won't see anything of the game – it's intended to be this way, as you will have to write the active object for the update timer first:



The following image is a representation of the logical structure of the game. The green components are of special importance to this exercise:



According to the logical structure, the game is split up into four parts – the Active Scheduler is a system component:

- ***ActiveObjects.cpp:*** The main file, rather similar to the framework of the exercises up to now. Responsible for starting the game – and for creating the Active Scheduler.
- ***Engine.cpp:*** The game engine, responsible for the game logic, handling key input and drawing the screen. Note that the console does not support any gaming-related techniques like double buffering, therefore the engine makes sure that flickering because of redraws is minimal.
- ***UpdateTimer.cpp:*** The first active object. It will regularly cause the engine to update the world and to refresh the screen.
- ***ConsoleKeys.cpp:*** An asynchronous way of accessing the console, which waits for key presses.

Detailed Description & Tutorial

The game engine works like a black box that just needs the two inputs for regular updates as well as for key presses. To complete the game, you have to implement those two tasks – the first one is to complete the update timer class, the second one is to write the key listener class. The integration into the existing framework is very easy. Essentially, you have to follow the edit steps. The sections below will give more detailed information about what to do in which order and where to find the next edits.

Completing the UpdateTimer-class

Part 1: Edits 1 – 4 → UpdateTimer.h

Most of this class has already been written and defined – you have to add the relevant lines to transform the class from a normal class to an active object.

This includes deriving the class from the active object base class (`CActive`) as well as to define the relevant methods you have to override when implementing an active object: `RunL()` and `DoCancel()`.

Also, the class will communicate with an asynchronous service provider. In this case, it's a timer. The class, which provides the communication to the server, is called `RTimer`. An instance variable has to keep this variable.

Part 2: Edits 5 – 17 → UpdateTimer.cpp

In these edits, you have to implement the relevant functionality for the asynchronous functions. This includes assigning the priority to the active object, adding it to the active scheduler and creating the thread-relative timer.

The `StartProcess()` method can be called from the outside to issue the first request and to start the timer – the interval is specified with a parameter. When the time has passed, the `RunL()`-function calls the update function of the game engine (`UpdateWorldL()`) and immediately schedules the next call-back in the same interval.

This interval can be modified by the game engine to increase the speed of the game in later levels.

As always in Symbian OS, it's very important to do proper cleanup. The cancel-method (`DoCancel()`) has to make sure that a possibly outstanding request to the active scheduler is cancelled. This method has to be called indirectly (through `Cancel()`) by the destructor of the active object, which also has to close the connection to the server side object, to make sure that any allocated resources are freed.

Part 3: Edits 18 – 22 → ActiveObjects.cpp

This part contains creating an instance of the update active object as well as activating it. It's important that at least one request is issued to the active scheduler before the scheduler is actually started – the start-method does not return until the active scheduler is stopped; if the application doesn't wait for a single request before the scheduler is started, it'll wait forever.

Part 4: Edit 23 → Engine.cpp

As mentioned above, the active scheduler has to be stopped to make the application continue after the source code line that started the scheduler. This is done by the engine when the game is over – which can either happen automatically when the time is over, or through keyboard input.

Implementing the ConsoleKeys-class

Part 5: Edits 24 – 31 → ConsoleKeys.cpp

In previous exercises, a synchronous version of getting key input from the console has been used. This is of course not possible for a game like this, which has to do regular screen updates and can't wait until the user presses another key. The console was mainly made for testing and isn't fully featured, so it's not possible to query the current key state through it – which is not a limitation of Symbian OS, just of the current `CConsoleBase` implementation.

However, as this class does provide an additional, asynchronous version for getting key input, it's a perfect example for an active object – even though `CConsoleBase` is a C type class, not an R class as would usually be the case.

With the experience from the update timer active object, this time your task is different. While the class has already been defined, you have to implement the methods yourself. This also includes the two phased construction from a previous module.

The `StartL()` method should be called from the outside to issue the first request to get a key value. As soon as the user presses a key, the `RunL()`-method of our active object is executed. Through `iConsole->KeyCode()` you can query which key has been pressed and send this key code to the engine. Again, you should immediately start another read to get the next key.

Note that this method is not optimal for a game like this, as you will get the key events with the key repetition rate set in the system. This means that when the player wants to move a longer distance to one side and therefore holds down the key, he'll move only one step in this direction at first, and start a faster movement only after a short delay.

Better alternatives would be either to set a flag on key down and release it on the key up event, or to query the key state in every run of the game engine. As said before, this is not possible through `CConsoleBase`, though.

Note that you do not have to delete the `CConsoleBase`-instance in the destructor of the object, as this variable is not owned by the active object.

Part 6: Edits 32 – 34 → ActiveObjects.cpp

The final edits are related to starting the key listener, which you just implemented. Like for the update timer, you have to issue the first request before starting the active scheduler. Do not forget to increase the number of objects to delete through the cleanup stack at the end of the function, so that the instance of the `ConsoleKeys` active object is deleted from the heap memory again.