Responsive Real-time Searching

In this article we will discuss a simple technique you can use to implement real-time data searches that produce responsive feedback and updates in your apps.

The Challenge

If you have ever tried to implement a real-time search, you will be aware that it can be a difficult to maintain application responsiveness for large data sets and/or dynamically changing search criteria.

For example, your application may have these requirements:

- The app has a massive data set.
- The data set needs to be search-able.
- Searches need to execute as soon as a user starts entering the search criteria.
- When the search criteria change, the search should automatically adjust in real-time.
- Matching entries are returned as they are found, updating the app interface.
- The app should remain responsive.

The last requirement is critical. If your app hangs or has temporary hiccups while a search executes, you may as well not distribute it.

So, how to do this?

The Sample App

To demonstrate a solution to the general problem above, let me specify an exact application and then provide code solving the problem.

This application will have the following features:

- **Large Data Set** - A simple word list containing over 100,000 words.
- **FPS Counter** - A simple FPS counter will be shown at all times to give concrete proof of responsiveness.
- **Search Field** - A single text entry field. (Works on devices and in both simulators.)
- **Progress Counters** – Meters to show total words, words found, and current search index.
- **Results List** – A basic (non-scrollable list) of words as they are found.

The App Modules

The sample code that comes with this post has several modules (found in same named LUA files):

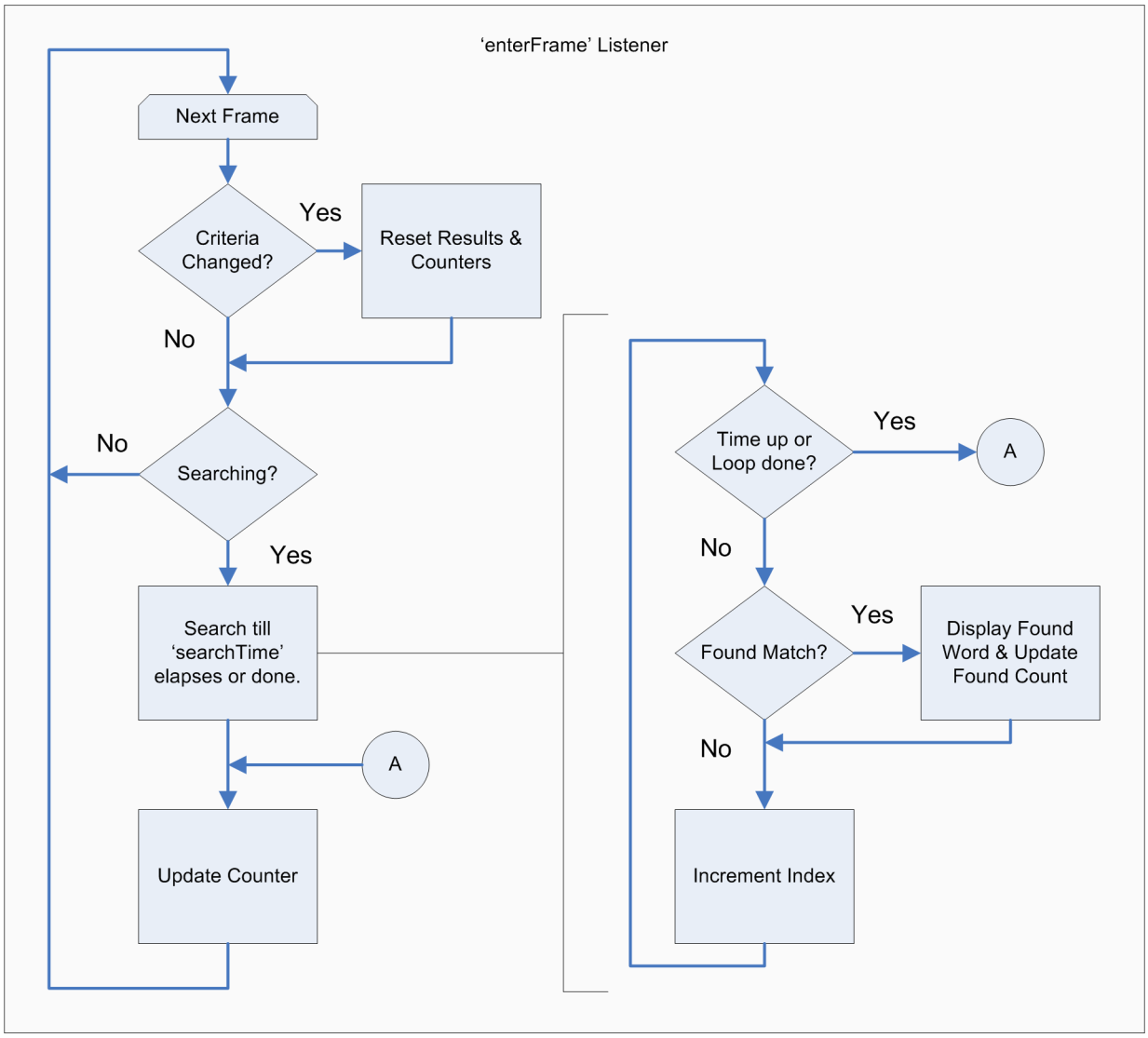
- **common** - Calculates and discovers useful variables and flags (`left`, `right`, `centerX`, `onSimulator`, etc.).
- **wordList –** Generates the data set.
- **meter** - Creates an frame-rate meter.
- **searchField** - Creates a 'text input field' for our search that will work on devices and both simulators. (Also creates count and index counters.)
- **example** - The solution to the problem posed at the start of this blog.

While each of the above modules may be useful and interesting, we will focus only on 'example'.

The Solution

After all this build up you may be disappointed to find that the solution is basically a self-regulating enterFrame listener.

In a nutshell, the listener starts a new search whenever the search criteria change and searches in a tight loop till a set amount of time passes. It then stops searching and exits. On the next frame the entire sequence starts again. The listener has this logical structure:



Now let's look at the actual code.

Initialize Search Settings

Before we start the enterFrame listener, we need to initialize the module:

1. Create and position initial results display group.
2. Initialize flags and variables to starting values.
3. Set how many comparisons we're allowed to do per frame.

```
function example.init( maxTime )
   -- 1.
   foundGroup = display.newGroup()
   foundGroup.y = com.top + 60

   -- 2.
   searching = false -- Not currently searching
   lastTerm = ""     -- No search term yet.
   curIndex = 1      -- On first word in word list.
   foundCount = 0    -- No words found yet.

   -- 3.
   searchTime = maxTime or (1000/display.fps/2)
end
```

Notice, that (in step #3) when we initialize the search code, if we don't specify a specific time, the code automatically detects the FPS (as set in config.lua) and then calculates a time equal to half of one frame.

The enterFrame Listener

Once the module is initialized, we can define the 'enterFrame' listener and start it running. The definition has five parts.

Part 1 – Get the current search term and see if it has changed since the last frame.

```
local searchTerm = searchField.getSearchTerm()
if( lastTerm ~= searchTerm ) then
   example.resetResults()
   lastTerm = searchTerm
   searching = ( string.len( searchTerm ) > 0 )
end
```

If the the search results have changed, we reset the search results (similar to initialization of module), take note of the new search term, and and set a flags saying that we are 'searching'. If they have not, we simply ignore this bit of code and continue on.

Part 2 – Abort if not 'searching'.

```
if( not searching ) then return end
```

If the 'searching' flag is set to false, we abort early and wait for the next frame to start again.

Part 3 – Search till we are out of time, or at the end of the word list.

```
local getTimer       = system.getTimer -- localize for speedup
local strLower       = string.lower -- localize for speedup
local startTime      = getTimer()
local elapsedTime    = 0

while( elapsedTime < searchTime and curIndex <= #wordList ) do
   local curWord = wordList[curIndex]
   if( string.match( strLower(curWord), strLower( searchTerm ) ) ~= nil ) then
      example.drawResult( curWord )
   end
   elapsedTime = getTimer() - startTime
   curIndex = curIndex + 1
end
```

This code:

- Localizes some useful functions for an execution speedup.
- Takes note of the 'startTime'
- Sets 'elapsedTime' to zero.
- Enters a search loop and does not exit till it get to the end of the list or runs out of time.
    - Upon finding match, the code displays it, and continues.

This is the meat of the solution and you should understand that by measuring 'elapsed time' each time we search and (possibly) display results, we ensure that:
A. The search can give us as soon as it needs to and not block the completion of this frame.
B. The code that stops and resumes searching is dynamic and takes into account the cost of the search and displaying the results.

Part 4 – Update the search index label

```
searchField.setSearchIndex( curIndex )
```

Note: This part is purely for feedback in the example.

Part 5 – Check to see if we reached the end of list and quit if we did.

```
searching = curIndex < #wordList
```

As a final step in the listener, we check to see if the end of the word list was reached. If it was, we set 'searching' to false. In either case, we drop out of the function. It will execute again at the beginning of the next frame.

Example Code

As I mentioned above, this blog post comes with sample code. It can be found here:

https://github.com/roaminggamer/RG_FreeStuff/raw/master/AskEd/2015/05/responsiveSearches/code.zip