Eco 2 rm 500

Laura Bow II

Programming Style Manual

Coding Preferences Of Brian K. Hughes

This is an attempt at a programming style manual. It is intended to provide the reader with a list of my coding styles, and the DOs and DONTs of coding on Laura Bow II. *It is not intended to be an authorized representation of Sierra coding policies.* While many of the rules listed in this document are no more than the personal preferences of a demented lead programmer (yours truly), I hope that the majority of this material will become ingrained in your programmer-self and carried on to your next project. Obviously, I'd like to see everyone adopt my coding style, but I am not the only lead programmer here. Just as I like things my way, so another lead may like it differently. The best we can hope for, therefore, is a union of styles; if we all code alike, it really doesn't matter if it's my way or someone else's.

Please keep in mind that the information in this document is not carved in stone (but only because my printer has trouble feeding slabs of granite). If you have a serious objection to a concept herein, or feel you have a more efficient way to achieve the desired result, feel free to contact me. I firmly believe that one who is in a teaching capacity must spend roughly half his time learning as well.

Contents

1.0 Module Organization

Arranging the elements of your module in a particular order makes the code much easier to read and maintain by others. Generally, grouping object by type is the easiest, although sometimes it is more logical to group objects by subject. For example, an actor could be grouped with all the other actors, or with his mover, sound, and script. Either way is acceptable.

Clearly mark your groups so that they can be found easily. A distinguishable comment before a group of items is the easiest way to do this. I have a macro, called MakeBox, which will create a double-lined box around a piece of text. I will be happy to provide this macro to anyone who wants to adopt its style.

1.1 The Order of Things

The following chart represents how I like to organize a file. You need not adhere to it strictly, but I strongly recommend it. Items marked with a § are required for every module.

header comments
script #
includes
procedure list
public entry list
defines
locals
instance of LBRoom
procedures
Actors & Props
Talkers & related objects
Views & PicViews
Features
Sound, Code, & Misc

1.2 Procedures

In the above list, you'll notice that I recommend procedures do not come before the instance of room. The rational behind this is that the main object in the module (the room instance in room modules) should be the first code encountered. Some modules may contain several long procedures. It is inconvenient to one perusing the file to have to page-up and page-down through these procedures to see what the main object looks like.

Be aware of your coworkers' procedures. If there is a procedure already written that may be used again with little or no modification, it may be beneficial to break the procedure out of the module in which it resides, put it into room 0 or other public module, and create an external reference define for it in GAME.SH.

---

[1]Header comments should include the name and purpose of the module, the author, the date last updated, and a list of any classes defined in the module.

1.3 Public Entries & Externals

Any object that will be referenced outside the module in which it resides must be assigned a public entry number in that module. As an example, an instance of room in a room module must be assigned public entry number 0. This is how *Game startRoom* knows about the room[2]. Public objects are referenced in other modules via the *ScriptID* kernel call. *ScriptID* takes as parameters the number of the module in which our public object resides and the public entry number assigned to it, and returns an address to that object. For example, given:

```
(script# 100)

(public
        myActor 0
)

(instance myActor of Actor)
```

we can reference *myActor* from another module with *(ScriptID 100 0)*

If you will be referencing an object more than once, you should create a define for the *ScriptID* call. In the above example, we could save the object ID of *myActor* for use in several places, by creating the following define:

```
(define xMyActor[3] (ScriptID 100 0))
```

Procedures may be referenced externally as well as objects. The process is the same for objects and procedures, but our procedures will be referenced through *extern* statements in **GAME.SH**. These work the same as the *ScriptID* statement above, but generate a different PMachine op code.

1.4 Overlay Modules

Overlay modules are modules that contain objects related to, but not always necessary for, a room module. You should review the code situation very carefully before splitting code out into an overlay module. Overlays are often difficult to manage, and can be very tricky as far as memory management. By contrast, however, they are not as difficult to create as they seem. Overlays should be considered if:

- The total size of the source module is approaching 64K
- There is a group of objects or large amount of code that is used only in a specific situation
- The amount of hunk available when running the room is less than 5K
- There is a group of objects or large amount of code that can be shared by other modules

Creating an overlay module is a relatively simple process. Refining it, however, may take some time and will most certainly break the original module temporarily. To create an overlay module, do the following:

---

[2]If you are receiving a **NOT AN OBJECT: $0** error when *Game startRoom* tries to add your room to the regions list, check to see if you've forgotten the public entry for your instance of LBRoom.

[3]Refer to section 5.4 Defines, Globals, Flags, & Vars for more information on this syntax.

- § Create a new file with an appropriate name
- § Move the code to be split out into the new module
- § Create public entries in the original module for objects that must be referenced in the overlay
- § Create external reference defines in the overlay to point to the public entries in the original module
- § Create public entries in the overlay for objects that must be referenced in the original module
- § Create external reference defines in the original module to point to the public entries in the overlay
- § Be sure that either the original module or the overlay itself removes the overlay from memory when finished with it

The hardest part about creating overlay modules is getting the external referencing correct. **NOT AN OBJECT** will generally mean that one of the two modules is trying to send a message to an object that is no longer in that module. **DISPATCH NUMBER TOO LARGE** generally is caused by trying to reference a public entry that is larger than the last public entry for that module. For example, the error will occur if *(ScriptID 100 3)* is used in one module and script# 100 only contains public entries 0, 1, and 2.

2.0 Object Organization

How you lay out your objects' code is nearly as important as how you lay out the module, in terms of style. For most objects, this section is not important. But for a few objects such as rooms, these guidelines are recommended.

2.1 Room *Init*

The *room* is one of the most common objects in the game, and the room's *init* method is one of the most frequently referenced. It is fairly important, therefore, to keep the room's *init* method clean and easily readable. One way in which this can be accomplished is to order your room's *init* code is the following way:

1. Pre-loading Resources
2. Setting the region, if any
3. Setting up ego[4]
4. Initial code based on previous room
5. *Super init:*
6. Setting Up Polygons
7. Setting up the cast
8. Setting up features
9. Miscellaneous initial code
10. Setting script(s)

[4]Ego must generally be inited before the *(super init:)*, if the room is an instance of a subclass of Room that handles walking ego in and out of rooms.

2.1.1 Pre-loading Resources

All views, pics, sounds, cursors, and fonts used in the module should be loaded here. This serves two purposes: 1) to annotate which resources are being used (better than comments at the top of the file), and 2) to prevent "disk hits" during the execution of a room, the theory being that if all the resources are loaded the game should never require disk access until another room change.[5]

2.1.2 Setting the Region

Setting the room to a region (or vice versa, depending upon how you choose to look at it) is accomplished with the *(self setRegions: regionModuleNum)* statement, where *regionModuleNum* will usually be a define from GAME.SH.

2.1.3 Setting Up Ego

Setting up ego will normally consist of two messages, an *init* and a *normalize*. Additionally, ego may positioned here or have alternate cyclers and movers set. Note that the game's *startRoom* method will put a StopWalk cycler on ego by default. Except in extreme cases, you should not alter ego's *edgeHit* property here.

2.1.4 Initial Code Based on Previous Room

Any initial code based on the previous room, such as positioning ego or setting vars, is accomplished by the *(switch prevRoomNum)* statement, as follows:

```
(switch prevRoomNum
        (north
                [code if coming from the north]
        )
        (south
                [code if coming from the south]
        )
)
```

It is helpful to include an *else* clause that sets debugging information when teleporting.

2.1.5 Super Init:

The *(super init:)* statement causes the super class, Room or a subclass thereof, to perform its initialization code. Among other things, this code draws the picture and sets ego to walk into the room (if appropriate).

2.1.6 Setting Up Polygons

This is where polygons should be added to the room's obstacles list. They may be created dynamically (the normal output from the polygon editor) or be static instances that are simply added to the list.

2.1.7 Setting Up the Cast

This is where all the Actors, Props, Views, and PicViews are initialized. From a size standpoint, it is cheaper to send complex messages here than to override the object's *init* method. For example:

```
(object init:, approachVerbs: verbList)
```

---

[5] If all resources have been pre-loaded and disk access is still occurring, this may indicate that the room is "hunk heavy" and prone to thrashing. See also sections 1.4 Overlay Modules and 3.2 Pre-loading Resources.

2.1.8 Setting Up Features

Features should all be initialized in one place to make locating them later easier. Features should be initialized individually, just like other objects[6]. As with objects in the cast, it is better to have complex messages here than in the Feature's *init* method.

2.1.9 Miscellaneous Initial Code

This is where other initialization code should go. I strongly suggest annotating this code well to make it as clear as possible.

2.1.10 Setting Script(s)

Finally, the *init* method of a room should set any scripts that need to be executed upon entering a room. Sometimes another *switch* is required to set different scripts based on the previous room.

2.2 Dynamic Objects

A dynamic object is a clone of an object, an exact duplicate. They are useful when you need *an* object, but don't care *which* object. For example, you might need a temporary list to hold numbers for a sort routine. A dynamic list can be used just like any instance of List, but does not require an instance in the module. In the debugger's object list, a dynamic object will be preceded by an asterisk and will bear the same name as the object from which it was cloned. There are two ways to create a dynamic object: *(theObj new:)* and *(Clone theObj)*. Whenever possible, the *new* method should be used instead of the kernel call.

Because dynamic objects allocate space off the heap instead of hunk, it is important to set a variable to the object's address so that it may be disposed of later. Orphaned dynamic objects (those left "hanging" with no handle to them) can cause frags or heap problems. By the same token, the number of dynamic objects in memory at one time should be monitored carefully. Too many objects can lead to low heap problems.

2.3 A Word About Features

The notion that it is more efficient to add all the features to the features list and then init them all is incorrect. Since the *init* method of Feature adds the object to the features list, it is redundant to add it manually before the init. For example:

Replace

```
(features
   add:
      object1
      object2
      object3,
   eachElementDo: #init
)
```

with

```
(object1 init:)
(object2 init:)
(object3 init:)
```

Features created with the on-line feature writer will have simple nsRects that define the area for which the *onMe* method will return TRUE, thus claiming an event. This "claim area" can be redefined by:

---

[6]For more information on features see also section 2.3 A Word About Features.

- redefining the *onMe* method
- setting the *SKIPCHECK* signal bit
- setting the *onMeCheck* property to a control color
- setting the *onMeCheck* property to an instance of polygon

Redefining the *onMe* method is by far the most versatile, though often not the most convenient. Setting the *SKIPCHECK* bit in the signal property is very easy and requires no changes to the view or the code. Setting the *onMeCheck* property to a control color is also very easy, but requires a change to the picture. Of the three, I recommend using *SKIPCHECK* wherever possible.

This brings us to dynamic polygons in features. Creating a dynamic polygon in the feature's *init* method, then putting its object ID into the *onMeCheck* property is inefficient at best and dangerous at worst. Not only does this method require redefining the *init* and *dispose* methods to create and dispose the polygon, but each dynamic polygon created will allocate approximately 300 bytes off the heap. In a room with 25 such features, up to 7,500 bytes of heap could be allocated by dynamic polygons. In addition, should you forget to dispose of the polygon in the feature's dispose method, the resulting frag can be very difficult and time-consuming to trace. I strongly advise against this method of defining features.

A word about addToPics: Whenever the *addToPic* method of a View, Prop, or Actor is invoked, two things happen. First, a PicView clone is created and added to the addToPics list. It will have all the properties of the original object, but will be named "PicView". Second, the original object will be removed from the cast and added to the features list. This is NOT a dynamic object, but the original object itself.

2.4 Annotation

When faced with the decision of throwing something away, my father used this rule of thumb: If he hadn't used it in the last year, he figured he'd never use it again and it got tossed. The same rule of thumb applies to commented code. Lines of code are usually commented because:

- they aren't needed any more
- they have been temporarily replaced by new code
- they have been permanently replaced by new code
- they have been temporarily removed for testing

If your commented code falls into any of the first three categories and you have not re-instated it within two to three days, remove it. Old commented code is a waste of space and makes reading the uncommented code more difficult.

When dealing with objects, there are two areas in particular where I like to see comments. The first is immediately following a method declaration, unless the function is obvious, and the second is following each new method and property declared in a class. For example:

```
(instance myObj of Prop
        (method (doit)
                ;I like to see comments here unless the code in this
                ; method is really obvious
        )
)
```

```
(class myClass kindof Prop
        (properties
                fooBar          ; Explain what this property does!
        )
        (methods
                setFooBar       ; Explain what this method does!
        )
)
```

Additionally, I have found that it is helpful to comment the closing paren of a method or object, especially if the code is more than 30 or 40 lines. When scrolling through a file it helps to be able to identify the object that immediately precedes the code I'm reading. For example:

```
(instance myObj of Prop
        (method (init)

        ) ;Don't need a comment here
        (method (doit)
                ... 50 lines ...
        ) ;end doit    < - Nice to have this!
) ;end myObj           < - Nice to have this, too!
```

The general rule is: If you can see both the method declaration and closing paren without touching the arrow keys or Page Up or Page Down, a comment is unnecessary.

Most people have customized their comment macros to insert special characters after the semicolon. For instance, your comments might look like this: ;§§. Some people use their initials, as in: ;SRC or ;BH. These are helpful in determining who has changed code.

3.0 Optimization

We are pushing toward single-platform development, which means that once the game is shipping on the IBM platform, we would like to be able to roll it over to the Macintosh and Amiga with no significant changes to code or other resources. Granted, this will require many changes to our existing system and tools. But even when the all else has been done to that effect, the single largest responsibility lies with us, the lowly application programmers: Optimization. It won't do us any good to be able to roll a game over to the 4.77 MHz Amiga in 2.6 hours flat, if it runs like a slug when it's there.

We have to take the responsibility at the time the code is designed and written to ensure that it will run as efficiently as possible on even the slowest machines. That does not mean, however, that we have to sacrifice animation to the point where the game becomes a slide show on a fire-breathing 386 or 486 machine. There is a happy medium, which is well-designed and well-written code that respects machine performance variations.

The worst offenders, in terms of speed and efficiency, are:

- *doit* methods with code that need not be done every cycle
- sending messages each time information is needed instead of setting a variable the first time and using it in subsequent code
- not making certain animating objects are stop-updated or even added to pic when possible

- § not implementing detail level
- § not pre-loading resources
- § bad overlay module management
- § inefficient code layout and lack of encapsulation
- § "Band-Aids", "quick fixes", "patches", "prophylactic code"

If you have any questions about the best way to accomplish a task, ask me. If I don't know I'll be happy to find out.

3.1 The Dreaded Doit

Doit methods are really neat. Every game cycle each object that has a *doit* method gets a shot at stardom. Imagine that each object with a *doit* method is a lead programmer and that one game cycle is one project status meeting. The more material each one has to report and the more inefficiently he presents his report, the more time will be required to get around to everyone and the longer the lead meeting will take. Now imagine that Ken Williams has mandated that the project status meeting will take no more than 30 minutes. To fit within this time constraint each lead has to report only the material necessary in a clear, concise manner. Likewise, each *doit* method needs to contain only the code necessary and in a clear, concise syntax.

Every message sent takes time to execute and 16 bytes of compiled object code. Therefore, every message you can avoid in a doit is that much less overhead. If you will be checking an object's property or the return value from an object's method repeatedly, set a variable the first time, then use it for all subsequent checks.

Most of all, remember that objects have minds of their own; they have properties to remember data and methods to facilitate actions and reactions. They don't need to be babysat by a *doit* method. For instance, in the following *doit* code:

```
(if (== (theMusic prevSignal?) -1)
        (theScript cue:)
)
```

a message is sent to *theMusic* and a comparison is done each and every game cycle. This sort of code should be replaced by the following lines, where the music is first played:

```
(theMusic play: theScript)
```

3.2 Pre-loading Resources

Every time the interpreter needs a view, picture, sound, cursor, font, or script that is not in memory, it has to go searching on the disk. Even on a fast hard drive this can cause pauses in the animation that can ruin the continuity of a scene. These "disk hits" can be avoided by simply pre-loading every resource the room will use, before it tries to use them. As explained in section 2.1, these pre-loads should be done at the very beginning of a room's *init* method, using the following statement:

```
(Load resourceType resourceNumber resourceNumber ...)
```

where *resourceType* is one of the resource type definitions in **SYSTEM.SH**. Following the resource type is a list of all the resources of that type you wish to load, as in:

```
(Load VIEW
        vEgoDying
        vEgoLaughing
)
(Load CURSOR
        exitCursor
        walkCursor
)
(Load SCRIPT ABOUTCODE)
```

You can use the -c option when running the interpreter to help you find disk hits in your room. This option will cause the cursor to change to a disk drive icon whenever a resource is loaded. No disk cursor means no disk hits.

3.3 Messaging

A message can either invoke a method of an object, set a property, or query a property. The syntax rules are as follows:

invoking a method:	*(object method: parameters)*
setting a property:	*(object property: expression)*
querying a property:	*(object property?)*

While the punctuation following the selector (: and ?) are optional to the compiler, they are required by me.

As mentioned in section 3.1, every message sent to an object requires 16 bytes of compiled object code. When you consider that often times there are many messages sent that you can't see in your code, the total number of messages passed during the execution of one little room is astounding.

For this reason it is important to optimize your messaging as much as possible. Store the result of a message if you will be using it frequently. Often this can be done in-line with other code, as in:

```
(if (= theObj (theScript client?))
        (theObj posn: 100 100)
)
```

By storing the result of *(theScript client?)* in the variable *theObj*, we have saved ourselves from having to query *theScript*'s client every time we want to use it.

4.0 Indention

More than any other factor, indention style can make code very readable or impossible to read. Different college instructors teach different methods, depending on their background and the language they teach, but here is the way I want to see code indented:

- § Set your tabs in BRIEF to 4, 7. This means the first tab is at column four and every tab thereafter is three columns apart.
- § As a general rule, all statements in method, procedure, *if* or *cond* block, *while* or *for* loop, or *switch* statement should be indented to the nearest common column

- § Values in property lists should be indented to the nearest common column (see Example 1 below)
- § Parameters in complex messages should be indented to the nearest common column (see Example 2 below)
- § Closing parens should be outdented to the same column as their match
- § Expressions in an *and* or *or* statement should be indented to the nearest common column (see Example 3 below)
- § The first expressing following an *and*, *or*, or *not* statement should be on the same line as the statement (see Example 3 below)
- § Values in consecutive *define* statements should be indented to the nearest common column
- § Initial values of local variables should be indented to the nearest common column
- § Array elements should be indented to the nearest common column following either the array name or the open bracket (see Example 4 below)

4.1 Tabs vs. Spaces

BRIEF will allow you to fill columns to the left of text with either tabs or spaces. I prefer tabs because they are easier to manage when moving code around. Get into the habit of using the TAB and SHIFT-TAB keys to indent and outdent blocks of code. You'll save lots of time over manually inserting and deleting spaces.

4.2 In *If* Blocks

All statements that belong to a specific condition should be indented to the same column. Likewise, all expressions should be so aligned, with the first expression following an *and*, *or*, or *not* statement falling on the same line as the statement. (See Example 3 for details or talk to me).

4.3 In Properties Lists

Values in properties lists should be indented to the nearest common tab, meaning that they will be aligned vertically in the tab position closest to the longest property name. I have a BRIEF macro that automatically aligns property values which I will be happy to give you. (See Example 1 for details).

4.4 In Complex Messages

Similar to values in a properties list, parameters in a complex message should be indented to the nearest common tab. The should be aligned vertically in the tab position closest to the longest selector name. I have a BRIEF macro that automatically aligns complex message parameters which I will be happy to give you. (See Example 2 for details).

4.5 In Arrays

Initial values in arrays should be aligned with the nearest common tab following either the array name or the open bracket. While I prefer moving the open bracket to the line following the array name and indenting it one tab, this is against standard C coding conventions and so I won't enforce it. The indention of the array elements, however, is required. (See Example 4 for details).

4.6 Blank Lines

Blank lines can be very helpful in setting groups of related statements apart from each other. I don't have any hard fast rules about using blank lines; use your best judgement. If a blank line makes the code easier to read, put one in.

4.7 Examples

Example 1: Indention of property values

```
(properties
        x       5,
        y       5,
        view    100
)
```

Example 2: Indention of complex message parameters

```
(theObj
        setLoop:        0,
        setCel:         0,
        setCycle:       Forward
)
```

Example 3: Indention of expressions

```
(if     (and    (== theView 100)
                (or     (theObj isKindOf: Thing)
                        (theObj respondsTo: thingValue)
                )
        )
        (Print "Foo to all")
)
```

Example 4: Indention of array elements

```
(local                          or      (local
   myArray =                                myArray = [
      [                                        value1
         value1                                value2
         value2                             ]
      ]                                  )
)
```

5.0 Naming Conventions

There are a few Sierra standards concerning the naming of variables and objects that we will be using. Note that capitalization is essential. The standard conventions are:

globals, locals, temps	*globalName*
flags	*fFlagName*
point flags	*ptFlagName* – defines, points ptsDefineName
defines, text substitution	*defineName*
defines, numeric constant	*DEFINE_NAME*
defines, external reference.	*xDefineName*
classes	*ClassName*
procedures	*ProcedureName*
inventory enums	*iInvItemName*
views	*vViewName*
inventory views	*ivViewName*
pictures	*pPictureName*
actors	*aActorName*
talkers	*tTalkerName*
scripts	*sScriptName*
music	*mMusicName*
sounds	*nSoundName*

If you can think of other defines that should be in this list, let me know.

The Part at the End

In general, keep your code as clean and efficient as possible. Again, the rules in this text are not set in cement. Any comments or suggestions will certainly be ~~laughed at~~ considered.

The first time any of you break the above rules, you will be told nicely. The second time you will be told firmly. The third time I will personally dull all your pencils so they won't stick in the ceiling anymore, change all occurrences of your name in the source code to Arvin Slatherlord Loudermilk III, and force you to take lunch at 8:35 am for a week. So there.

From: Lucy
To: Brian

1/3/92

Comments on LBZ Programming Manual

1. I like it.

2. Page 13 — Blank lines.
I like blank lines between all major blocks objects, methods, procedures, etc. Also after large blocks within methods (e.g. cond's, etc.)

3. Page 13 — example 4 (indention of array elements).
Don't worry about "c" standards — your first (preferred) approach is more readable, should be the standard.

4. Add to indention standards: When sending messages or assigning values to several objects or variables in a row, indent to common Tab.
Examples:

```
(= x      17)
(= foobar 29)
(x      doit:   y z)
(foobar rescind: TRUE)
```

5. Page 14 — I like your different naming conventions. How about dDefineName for text sub. defines so they don't look like variables (I've been using DefineName, but could switch).

6. Page 11 — Messaging — I use "?" when invoking a method which solely returns a value (i.e. a "property-like" method) — "?" is for questions, ":" for imper