

Vecset Editor

Design Specs

By Mike Hahn

Copyright © 2007 Vecset.net
Date Last Edited: 15-Sep-2008

Table of Contents

Overview	1
<u>Introduction</u>	<u>1</u>
<u>Screen Layout</u>	<u>2</u>
Board Layout Editor	4
<u>Layout Editor</u>	<u>4</u>
Code Editor	5
<u>Editor Window</u>	<u>5</u>
<u>Editing Modes</u>	<u>5</u>
<u>Code Menu Mode</u>	<u>6</u>
<u>Miscellaneous Features</u>	<u>20</u>

Overview

Introduction

[Home]

The Vecset Editor features a code editor and a board layout editor. The code editor is where the user enters/tests/debugs Vecscript code. This editor has basically 2 modes: Free-Form Mode and Structure Mode. Pressing the Esc key toggles between these 2 modes. Pressing the question mark (?) enters a 3rd mode: Code Menu Mode, in which a menu of available context-sensitive options is displayed. Selecting an option may bring up another, lower-level menu, which may lead to yet another lower-level menu, and so on.

The code editor also has advanced features such as syntax highlighting and debugging (breakpoints, single-stepping, and stepping over, or single-stepping without stepping into lower-level subroutines). Another feature of the code editor is the ability to toggle between infix and prefix modes (prefix is the default). With infix mode, binary operators come in-between their operands, while in prefix mode, all operators come before their operands. When infix mode is selected, Structure Mode is (possibly) unavailable.

The board layout editor allows the user to select reusable software components (objects) from multiple object palettes and place them on the (design-time) game window. When the game under development is executed, the user sees and interacts with the runtime (not the design-time) version of the game window. Objects placed on the game window can be modified with the Object Inspector, which displays a list of modifiable properties, as well as a list of events and methods linked to by those events.

Screen Layout

Object Tree/Inspector	
1st	2nd
my-button: Button	
Properties	Events
cancel	false
caption	Click Me!
cursor	cr-default
default	false
enabled	true
font	Font
height	25
hint	
left	388
mod-res	mr-none
name	my-button
show-hin	false
tab-ord	2
tag	0
top	72
visible	true
width	79

Vecset – Sample_Project															
File Edit ... Help															
Standard				Extra				System				Board			
■	■	□	■	■	□	■	■	□	■	■	□	■	■	□	■

sample.vs				
shell	Sample	samp2	samp3	
<pre> import (Hidden-form) (class (public) Scrab-form extends Form var (auto) (Game-server game-server; Button my-button; Button next-button; Board-grid board-grid; Rack-grid rack-grid; Card-stack bag; Editable-grid score-grid;) (proc (public static) main (String-list args) var (Scrab-form form1 (new Scrab-form); Hidden-form form2 (new Hidden-form);)) (proc (public) Scrab-form do (: game-server init-game;)) </pre>				

Main Window

The caption bar contains the project name, preceded by "Vecset." Below that is the menu bar, and below that is a button bar. If there are enough available buttons, and the main window is resized tall enough, there may be more than one row of button bars. If the main window is not resized tall enough to display all buttons, then the rightmost button contains a ">>" graphic, and the hint of that button reads: "More buttons." Clicking on it displays a 2D grid of additional buttons.

Below the button bar(s) is a row of tabs. Each tab contains the name of a collection of reusable software components (objects). Below that row of tabs is a row of objects (an object palette). At the right end of the row of tabs are 2 small buttons: a left arrow and a right arrow. Clicking on these buttons scrolls the row of tabs horizontally (if all tabs can be displayed, these buttons are hidden). The object palette includes a similar pair of buttons, which are hidden if all available objects can be displayed.

Code Editor

The caption bar contains the file name of the current Vecscript source file. If this file is not a member of the current project, the full path name is displayed. Below the caption bar is a row of tabs. Each tab contains a file name (minus the extension, which is usually ".vs"). At the right end of the row of tabs are 2 small buttons: a left arrow and a right arrow. Clicking on these buttons scrolls the row of tabs horizontally (if all tabs can be displayed, these buttons are hidden). Below the row of tabs is the contents of the file (usually a source code file) corresponding to the currently selected tab.

The leftmost tab may or may not consist of the text: "shell." (Under the View menu is a command to show/hide the shell.) If this tab is selected, the portion of the code editor below the row of tabs corresponds to the Vecscript version of the shell window in Python. The shell allows the user to enter Vecscript expressions and display the result of evaluating those expressions.

Object/Tree Inspector

The upper portion of this window displays up to 3 different tree views of the current project. It is separated from the lower portion by a splitter bar. The user can drag this splitter bar up and down. The lower portion of this window contains the Object Inspector. Below the splitter bar is a combo box containing the name and type (class name) of the current object. Below that is a pair of tabs: Properties and Events. Below that is a list of name/value pairs of available properties/events. Some property values include an ellipsis button (...), which when clicked displays a dialog box corresponding to that property.

Docking Behavior

To dock the code editor with the main window, drag its caption bar until it slightly overlaps with the bottom of the main window, keep holding down the mouse button for about one second, and then both windows will become docked together. To undock the code editor, drag what was once its caption bar (which is now part of the main window) away from the main window.

To dock the object/tree inspector with the main window (which is only possible if the main window is already docked with the code editor), drag its caption bar until it slightly overlaps with the left-hand edge of the main window, keep holding down the mouse button for about one second, and then both windows will become docked together. To undock the object/tree inspector, drag one of the tree tabs away from the main window.

Using menu commands to dock/undock (under the Window menu):

- **Dock All:** when checked, all 3 windows are docked together
- **Undock All:** when checked, none of the 3 windows are docked together
- **Partial Dock:** when checked, the code editor is docked to the main window

After docking 2 windows together, the user can drag a splitter bar separating the 2 windows up and down (or left and right).

Board Layout Editor

Layout Editor

The board layout editor allows the user to select reusable software components (objects) from multiple object palettes and place them on the game window. When the game under development is executed, the user sees and interacts with the runtime (not the design-time) version of the game window. Objects placed on the game window can be modified with the Object Inspector, which displays a list of modifiable properties, as well as a list of events and methods linked to by those events.

Object/Tree Inspector

The upper portion of this window displays up to 3 different tree views of the current project. It is separated from the lower portion by a splitter bar. The user can drag this splitter bar up and down. The lower portion of this window contains the Object Inspector. Below the splitter bar is a combo box containing the name and type (class name) of the current object. Below that is a pair of tabs: Properties and Events. Below that is a list of name/value pairs of available properties/events. Some property values include an ellipsis button (...), which when clicked displays a dialog box corresponding to that property. Under the Events tab, double-clicking on an empty event will add a new event handler (with an empty body) corresponding to the appropriate event, and belonging to the current object.

Object Sizing

The currently selected object on the game window is surrounded by 8 little black squares (one at each corner and one at the center of each edge). The object can be resized by dragging one of these black squares (dragging the object itself, or pressing Ctrl+Shift+Arrow Key to nudge it one grid unit, is how you move it). If more than one object is selected, the color of the black squares at each corner changes to gray, the other 4 black squares disappear, and those gray squares are no longer draggable.

Right-Click Menu

Right-clicking an object on the game window gives the user access to the following commands (which may also be accessed from the menu bar):

- Align to Grid
- Send to Front/Back
- Align with other selected objects, or space equally
- Change size to match other selected objects
- Change tab order

Code Editor

Editor Window

The code editor window appears below the main Vecset window. Its caption bar contains the file name of the current Vecscript source file. If this file is not a member of the current project, the full path name is displayed. Below the caption bar is a row of tabs. Each tab contains a file name (minus the extension, which is usually ".vs"). At the right end of the row of tabs are 2 small buttons: a left arrow and a right arrow. Clicking on these buttons scrolls the row of tabs horizontally (if all tabs can be displayed, these buttons are hidden). Below the row of tabs is the contents of the file (usually a source code file) corresponding to the currently selected tab.

The leftmost tab may or may not consist of the text: "shell." (Under the View menu is a command to show/hide the shell.) If this tab is selected, the portion of the code editor below the row of tabs corresponds to the Vecscript version of the shell window in Python. The shell allows the user to enter Vecscript expressions and display the result of evaluating those expressions. When the user scrolls the row of tabs horizontally, the shell tab always remains onscreen as the leftmost tab.

Keyboard Shortcuts

Most commonly-used editor commands have keyboard shortcuts, which are entered by holding down the Ctrl key, and while doing that, pressing one or more letter keys. More shortcuts can be entered by holding down both the Shift and Ctrl keys, and while doing that, pressing a single letter key. When typing the last letter key in a series of one or more letters (while holding down the Ctrl key, or both the Shift and Ctrl keys), it does not matter if that letter key is released before or after releasing the Ctrl and/or Shift keys.

Editing Modes

The code editor has basically 2 modes: Free-Form Mode and Structure Mode. Pressing the Esc key toggles between these 2 modes. Pressing the question mark (?) enters a 3rd mode: Code Menu Mode, in which a menu of available context-sensitive options is displayed. Selecting an option may bring up another, lower-level menu, which may lead to yet another lower-level menu, and so on.

Another feature of the code editor is the ability to toggle between infix and prefix modes (prefix is the default). With infix mode, binary operators come in-between their operands, while in prefix mode, all operators come before their operands. When infix mode is selected, Structure Mode is (possibly) unavailable.

Structure Mode Commands

A bottom-level token (e.g. a keyword, identifier, operator, or constant) or an entire list is often highlighted. Using the Shift key in conjunction with the Up/Down Arrow keys, it is possible to select more than one token/list at a time.

- **Esc** – toggle between Free Form and Structure Editor modes
- **Up Arrow** - go to previous list element

- **Down Arrow** - go to next list element
- **Left Arrow** - go to parent list
- **Right Arrow** - go to first child element (if none, display text cursor following current bottom-level token)
- **Shift+Up/Down Arrow** - select a range of tokens/lists
- **Printable Char.** - incrementally select valid matching menu item (if any)
- **Backspace** - undo insertion of previous printable char.
- **Delete** - delete current token/list
- **Enter** - display text cursor, insert space after cursor (or insert result of incremental menu selection)
- **Space** - display text cursor, insert space before cursor (or insert result of incremental menu selection)
- **Ctrl+Enter** – if at end of line, append blank line (otherwise break line into 2 lines)

Code Menu Mode

Pressing the question mark (?) enters Code Menu Mode, in which a menu of available context-sensitive options is displayed. Selecting an option may bring up another, lower-level menu, which may lead to yet another lower-level menu, and so on.

Code Menu Commands

A popup menu above or below text cursor (and including text cursor) is displayed. The contents of this menu include all valid code elements in the context of the text cursor (ignoring anything after the text cursor). If the current menu item refers to a list, the entire list is highlighted (defaults to light gray if background color of whitespace is white).

- **Question Mark** - toggle between Code Menu and Free Form/Structure Editor modes
- **Esc** - show/hide code menu
- **Up Arrow** - move selection up (scroll up after pressing Esc)
- **Down Arrow** - move selection down (scroll down after pressing Esc)
- **Left Arrow** - go to parent code menu
- **Right Arrow** - go to lower-level code menu, if any
- **Enter** - go to lower-level code menu (if none, insert current menu item, go to next menu item, or if none, go to parent code menu)
- **Space** - go to lower-level code menu (if none, insert current menu item, go to next menu item, or if none, go to parent code menu; exit Code Menu mode)
- **Printable Char.** - incrementally select matching menu item
- **Backspace** - undo operation of previous printable char.
- **Page Up** - page up after pressing Esc
- **Page Down** - page down after pressing Esc
- **Shift Arrow** – only used if current menu item is repeated, such as a statement in a block, a declaration, or a parameter in a parameter list
 - **Shift Up Arrow** – select previous instance of current menu item
 - **Shift Down Arrow** - select next instance of current menu item
 - **Shift Left Arrow** – insert above current menu item
 - **Shift Right Arrow** - insert below current menu item
 - **Semicolon** – toggle parent list: multi-line/single-line

Code Menus

Each menu is preceded by a menu name, followed by a colon (:). A menu line enclosed in square brackets ([]) is optional. If all menu lines are preceded by a vertical bar (|), then the user must select exactly one of those menu lines. A menu line followed by an asterisk (*) may be repeated one or more times (or zero or more times if also enclosed in square brackets).

A menu line followed by a semicolon (;) is similar to an asterisk, but when the parent menu is expanded into a list, all items of that list are delimited with semicolons, and the list defaults to multi-line mode. A menu line followed by 2 semicolons (;;) means that when the parent menu is expanded into a list, that list defaults to single-line mode, delimited with semicolons (currently a parameter list is the only example of the double-semicolon in the following grammar).

A menu line not enclosed in angle brackets (<>) is a terminal symbol (no further submenus exist). Terminal symbols are often in **bold**.

```
<unit>:
  [<pkg line>]
  [<import line>]*
  <class interf def>*
```

```
<pkg line>:
  package
  <pkg list>
```

```
<pkg list>:
| <pkg name>
| <pkg name list>
```

```
<pkg name list>:
(
  <pkg name>*
)
```

```
<import line>:
  import
  <import list>
```

```
<import list>:
| <import class>
| <import interf>
| <import all>
```

```
<import class>:
(
  <pkg name>*
  <class name>
)
```

```
<import interf>:
(
  <pkg name>*
  <interf name>
)
```

```

<import all>:
  (
    <pkg name>*
    *
  )

<class interf def>:
| <class def>
| <interf def>

<class def>:
  (
    class
    [<class modif list>]
    <class name>
    [<extends clause>]
    [<implements clause>]
    <class body>
  )

<interf def>:
  (
    interface
    [( public )]
    <interf name>
    [<extends interf clause>]
    <interf body>
  )

<class modif list>:
  (
    <class modif>
  )

<class modif>:
| public
| public abstract
| public final
| abstract
| final

<class name>:
| <cls name>
| <cls list>

<cls list>:
  (
    <pkg name>*
    <cls name>
  )

<interf name>:
| <int name>
| <int list>

<int list>:

```

```

(
  <pkg name>*
  <int name>
)

<extends clause>:
  extends
  <class name>

<extends interf clause>:
  extends
  (
    <interf name>*
  )

<implements clause>:
  implements
  (
    <interf name>*
  )

<class body>:
  [<field var list>]*
  [<stat obj init>]
  <method def>*

<interf body>:
  [<field var list>]*
  <method hdr>*

<method def>:
| <proc def>
| <func def>
| <cons def>
| <event handler>

<proc def>:
(
  proc
  [<modifier list>]
  <method name>
  [<parm list>]
  [<throws clause>]
  [<loc var list>]*
  <do with>
  <block>
)

<func def>:
(
  func
  [<modifier list>]
  <type>
  <method name>
  [<parm list>]
  [<throws clause>]
  [<loc var list>]*
)

```

```

    <do with>
    <block>
  )

<cons def>:
  (
    cons
    [<modifier list>]
    <class name>
    [<parm list>]
    [<throws clause>]
    [<loc var list>]*
    <do with>
    <block>
  )

<event handler>:
  (
    proc ( auto )
    <method name>
    (
      <event class name>
      <id>
    )
    [<throws clause>]
    [<loc var list>]*
    <do with>
    <block>
  )

<method hdr>:
| <proc hdr>
| <func hdr>

<proc hdr>:
  (
    proc
    [<modifier list>]
    <method name>
    [<parm list>]
    [<throws clause>]
  )

<func hdr>:
  (
    func
    [<modifier list>]
    <type>
    <method name>
    [<parm list>]
    [<throws clause>]
  )

<throws clause>:
  throws
  <exception list>

```

```

<exception list>:
| <exception>
| <except list>

<except list>:
(
  <exception>*
)

<exception>:
  <exception class name>

<modifier list>:
(
  <modifier>*
)

<modifier>:
| public
| protected
| private
| final
| static
| abstract
| synchronized
| transient
| volatile

<stat obj init>:
| <st obj init>
| <obj st init>

<st obj init>:
  <stat init>
  [<obj init>]

<obj st init>:
  <obj init>
  [<stat init>]

<stat init>:
  static
  <block>

<obj init>:
  do
  <block>

<parm list>:
(
  <parm>;
)

<parm>:
  <type>
  <id list>

```

```

<id list>:
| <id>
| <id lst>

<id lst>:
(
  <id>*
)

<field var list>:
  var
  [<modifier list>]
  (
    <field decl>;
  )

<loc var list>:
  var
  [<modifier list>]
  (
    <loc decl>;
  )

<field decl>:
| <fld decl>
| <typedef>

<local decl>:
| <loc decl>
| <typedef>

<fld decl>:
  [<modifier list>]
  [<property decl>]
  <type>
  <id list>
  [<expr>]

<loc decl>:
  [<modifier list>]
  <type>
  <id list>
  [<expr>]

<typedef>:
  [<modifier list>]
  typedef
  <type>
  <type name>

<property decl>:
| ( property read )
| ( property write )
| ( property read write )

<type>:
| <simple type>

```

```

| <class name>
| <list type>
| <set type>
| <array type>
| <enum type>

<simple type>:
| byte
| short
| int
| long
| float
| double
| char
| boolean

<list type>:
  list
  [<type>]

<set type>:
  set
  <set typ>

<set typ>:
| <enum type name>
| <subrange>

<subrange>:
  (
  ..
  <ord expr>
  <ord expr>
  )

<array type>
  array
  [<dim size>]
  <type>

<enum type>
  enum
  <id lst>

<do with>:
| do
| <with do>

<with do>:
  with
  <id list>
  do

<block>:
  (
  [<stmt>];
  )

```

```

<stmt>:
| <if stmt>
| <switch stmt>
| <while stmt>
| <for stmt>
| <with stmt>
| <asst stmt>
| <inc dec stmt>
| <ptr stmt>
| <call stmt>
| <jump stmt>
| <try stmt>
| <throw stmt>
| <synch stmt>

<if stmt>:
  if
  <bool expr>
  then
  <block>
  [<elseif clause>]*
  [<else clause>]

<elseif clause>:
  elseif
  <bool expr>
  then
  <block>

<else clause>:
  else
  <block>

<switch stmt>:
  switch
  <expr>
  (
  <case clause>*
  [<default clause>]
  )

<case clause>:
  case
  <expr list>
  do
  <block>

<default clause>:
  default
  <block>

<while stmt>:
| <while do stmt>
| <do while stmt>

<while do stmt>:

```

```

while
  <bool expr>
do
  <block>

<do while stmt>:
  do
  <block>
  while
  <bool expr>

<for stmt>:
  for
  <id>
  <for range>
  [<by clause>]
  do
  <block>

<for range>:
| <for int var>
| <for list var>

<for int var>:
  (
  <init expr>
  <limit expr>
  )

<for list var>:
  <id>

<by clause>:
  by
  <step expr>

<init expr>:
<limit expr>:
<step expr>:
  <int expr>

<with stmt>:
  <with do>
  <block>

<asst stmt>:
  <asst op>
  <var expr>
  <expr>

<inc dec stmt>:
  <inc dec op>
  <var expr>

<ptr stmt>:
  <left right op>
  <var expr>

```

```

    <expr>

<call stmt>:
| <proc call>
| <colon call>

<proc call>:
    <proc name>
    [<expr>]*

<colon call>:
    :
    <colon head>
    [<colon expr>]*
    <colon tail>

<colon tail>:
| <proc name>
| <proc call expr>

<proc call expr>:
    (
    <proc name>
    <expr>*
    )

<jump stmt>:
| break
| continue
| <return stmt>

<return stmt>:
    return
    [<expr>]

<try stmt>:
    try
    <block>
    [<catch clause>]*
    [<finally clause>]

<catch clause>:
    catch
    (
    <exception id>
    <except var name>
    )
    do
    <block>

<finally clause>:
    finally
    <block>

<throw stmt>:
    throw
    <throw obj>

```

```

<synch stmt>:
  synchronized
  <obj name>
  do
  <block>

<expr>:
| <const expr>
| <var expr>
| <op expr>
| <func call>
| <array ref>
| <obj expr>
| <lst expr>

<bool expr>:
<int expr>:
<ord expr>:
  <expr>

<var expr>:
| <var name>
| <prop name>
| <array ref>
| <colon var expr>

<colon var expr>:
(
  :
  <colon head>
  [<colon expr>]*
  <var end expr>
)

<var end expr>:
| <prop name>
| <array ref>

<obj expr>:
| <obj name>
| <colon obj expr>

<colon obj expr>:
(
  :
  <colon head>
  <colon expr>*
)

<op expr>:
(
  <op>
  <expr>*
)

<func call>:
| <func name>

```

| <func call expr>

<func call expr>:

```
(  
  <func name>  
  <expr>*  
)
```

<array ref>:

```
| <array name>  
| <array expr>
```

<array expr>:

```
(  
  <array name>  
  <int expr>*  
)
```

<colon head>:

```
| <obj name>  
| <func call>  
| <array expr>
```

<colon expr>:

```
| <func call>  
| <prop name>  
| <class name>  
| <array ref>
```

<expr list>:

```
| <expr>  
| <list expr>
```

<list expr>:

```
(  
  <expr>*  
)
```

<lst expr>:

```
(  
  list  
  <expr>*  
)
```

<const expr>:

```
| <int const>  
| <float const>  
| <string const>  
| <ord const>  
| <bool const>  
| nil
```

<asst op>:

```
| =  
| =+  
| =-  
| =*
```

```
| =/  
| =%  
| =&  
| =^  
| =<<  
| =>>  
| =>>>  
| =|
```

<op>:

```
| +  
| -  
| *  
| /  
| %  
| &  
| ^  
| <<  
| >>  
| >>>  
| |  
| ==  
| <  
| <>  
| <=  
| >  
| >=  
| not  
| and  
| or  
| xor  
| in  
| left  
| right  
| new  
| resize  
| instanceof  
| quest
```

<inc dec op>:

```
| ++  
| --
```

<left right op>:

```
| pchild  
| pnext
```

<bool const>:

```
| true  
| false
```

Miscellaneous Features

Syntax Highlighting

The foreground color, background color, and text attributes (bold, italics, and underline) of each class of code elements can be modified using this feature. The classes of code elements include the following:

- White space
- Identifiers
- Keywords
- Operators
- Numbers
- String Literals
- Comments
- Non-Terminal Symbols

The user can select from various pre-defined color schemes, including monochrome, and modify them if desired.

Variable-Width Hyphens

This feature is only available when the code font is mono-spaced (it is allowable to use variable-width fonts as the code font).

- **Hyphen Width:** the percentage value of the width of the hyphen (-) character divided by the width of a normal character.
- **Space Width:** the percentage value of the width of the space character divided by the width of a normal character.
- **Tab Width:** the percentage value of the width of the leading white-space space character divided by the width of a normal character.
- **Tabs to Spaces:** the no. of spaces to insert when the user presses the space bar as leading white space.

Keyboard Aid

This feature eases code entry by enabling the user to enter commonly used characters which are relatively hard to type with more easily-typed characters.

- **Hyphen:** press apostrophe ('). Use the double-quote (") for string literals.
- **Code Menu Mode:** press slash (/). Use the question mark (?) to enter divide-by (/).
- **Parentheses:** press comma (,) for the open parenthesis, and period (.) for the close parenthesis. Use the close parenthesis to enter a decimal point.
- **Hyphen (alternate):** when entering an identifier, hold down the Shift key and while doing that, press a letter key. This will enter a hyphen (-) followed by a lowercase letter.

Keyboard Aid is always disabled inside comments and string literals.

Debugging

Set breakpoints, single-step through your code, and step over your code (single-step without stepping into lower-level subroutines).