Contents

Structured Canvas Language

All SCL scripts begin with the *Start* routine. *Start* is the only upper-case keyword in SCL, and SCL is case-sensitive. *Start* is very important and it is the first thing your animation does. Usually it will **launch** your first sprites.

```
create routine as Start
  launch MyAirplaneSprite
end
```

That is just the beginning of SCL. We must also define sprites, and animations. These tasks are simply pieces that need to be assembled.

Basic SCL code consists of:

Sprites

That define the images that are animated

Routines

That control how your animation plays

Alterations

That define the animations to be applied to sprites

Events

That control interactions between sprites and users

```
create sprite from fan.png as spinner
    where x=100 y=100 center=90,90
    having alt=ThisWay
end
create routine as Start
    launch spinner
end
create rotation as ThisWay
    where speed=130 distance=135 completion=ThatWay
end
create rotation  as ThatWay
    where speed=-130 distance=135 completion=ThisWay
end
```

In the example above, *Start* launches a sprite and that sprite has a rotation alteration assigned to it that is called *ThisWay*. When that rotation is complete, it assigns the next alteration to the sprite, called *ThatWay*. And when *ThatWay* completes, it assigns *ThisWay* to the sprite… and the circle of animation goes on.

This is the basic process of all SCL animations. Launch sprites, and assign alterations to them.

To remove a sprite from the canvas, call **drop** inside a routine.

```
drop MySprite
```

Changing Alterations

The primary purpose of routines is to change sprite alterations. You can do this 2 ways:

Adding alterations to a sprite:

```
insert into {spritename} where alt={altname}
```

Removing alterations from a sprite

```
remove from {spritename} where alt={altname}
```

For efficiency you can also use a "sub" to define an alteration inside a routine instruction:

```
insert into {spritename}
  where alt=(sub
    create rotation where speed=700 distance=530
  end)
```

Alterations defined in routines can be anonymous - they don't need to be named. However it is your responsibility to include an end-point as part of the alteration otherwise the alteration will run forever and can only be removed if all alterations are removed from the sprite. An end-point, for example, is setting a destination or distance to how far a sprite should rotate. When the end-point is reached, the rotation is done and is automatically removed from the sprite.

Chaining Alterations

Sprite effects in SCL are called Alterations because they alter the sprite in some way. When an alteration completes, an event can be fired for which you can instruct SCL to do something else. That something else can be either to run a routine or to apply one or more alterations. Being able to

apply a new alteration when an alteration completes allows for making a chain of alterations.

Anonymous alterations are alterations that may not be given a name but are defined within the completion event.

```
create sprite from 2.png as whatever
  having alt=(sub create slide
      where direction=0 distance=50
    end)
end
```

In the example below a sprite is moved around using a chain of alts. The first alteration requires a name, FirstSlide, so that the last alteration can refer back it to restart the sequence.

```
create sprite from sclicon160x160.png as whatever
  having alt=(sub
    create slide as FirstSlide
      where direction=0 distance=50 completion=(sub
    create slide
      where direction=270 distance=50 completion=(sub
    create rotation
      where speed=600 distance=720 completion=(sub
    create translation
      where x=0 y=0 completion=FirstSlide
    end) end) end) end)
end
create routine as Start
  launch whatever
end
```

Event Handlers

When an alteration completes, the **completion** event handler can specify another alteration or a routine to run. Routines run this way are called event handlers. If the value assigned to **completion** is the name of a routine, then that routine is run.

```
create sprite from fan.png as spinner
  where x=100 y=100 center=90,90
end
create routine as Start
  launch spinner
  run routine ThisWay
end
create routine as ThisWay
  insert into spinner where
    alt=(sub create rotation as rot1 where speed=130
      distance=135 completion=ThatWay end)
end
create routine as ThatWay
  insert into spinner where
    alt=(sub create rotation as rot2 where speed=-130
      distance=135 completion=ThisWay end)
end
```

Changing Sprite Properties

You can change just about any sprite value that you can set when a sprite is created. Do this with an 'update' command:

```
update sprite {spritename} where {property}={value} and
{property2}={value2} [etc]
```

A special feature of routines run this way is the special variable _**sprite**. _**sprite** contains a reference to the sprite that triggered the event handler. That way you can use the same routine for any sprite, without knowing the sprite's name. Simply use _**sprite** in place of the sprite name anywhere within the event handling routine.

Calling Routines with Parameters

You can call a routine from inside another routine two ways. The first is a simple call using parenthesis, with or without values:

```
MyRoutineName()
MyRoutineName(11,22,33)
run routine MyRoutineName
```

The () indicates that that is a routine name. You can list values between the parenthesis that can be used by the routine. To reference parameters use arg.1 arg.2 arg.3 etc

```
create sprite from fan.png as spinner
  where x=100 y=100 center=90,90
end
```

```
create routine as Start
  launch spinner
  spinit(100)
end
create routine as spinit
   insert into spinner where alt=(sub create rotation where
speed=arg.1 distance=60 completion=spinagain end)
end
create routine as spinagain
   var rnd=(random(100)-50)*20
   spinit(rnd)
end
```

Calling Routines with Repeat

You can call a routine multiple times with 'repeat'

```
run routine MyRoutine where repeat=7
```

That would run the routine 7 times.

You can also repeat with a data object. This has the effect of running the routine once for each value in the list, with a small caveat. It is up to you to advance the data pointer so that the routine does not run forever on one value.

```
create data from list as L
set
   25 25 .25
   50 50 .5
   75 75 .75
end
```

```
create sprite from fan.png as spinner
  where center=90,90
  having alt=(sub create rotation where speed=360 end)
end
create routine as Start
  run routine X where repeat=data.L
end
create routine as X
   clone from spinner using original
   update sprite _clone where x=data.L.next
   update sprite _clone where y=data.L.next
   update sprite _clone where size=data.L.next
end
```

Notice how **data.L.next** is called. This enables the end of the list to be reached and for the routine to stop being called.

Collision Detection

You can detect when two sprites collide using collision detection. For performance reasons, SCL uses a very specific method to do this. First, you must specify which sprites to monitor for collisions. Secondly, collisions are detected based on the center point of the colliding sprite.
For example, to determine when an arrow hits a ball, you specify that the ball is the **target** of the arrow. When the center point of the arrow overlaps a non-transparent point on the ball sprite, the collision events are fired.

```
create sprite from arrow.png as Arrow
  where center=50,12 x=50 y=100 hashit=HasHit
  having target=Ball
```

```
  alt=(sub create vector where speed=100 end)
end
create sprite from ball.png as Ball
  where center=25,25 x=250 y=100 beenhit=BeenHit
end
create routine as Start
   launch Arrow
   launch Ball
end
create routine as HasHit drop Arrow end
create routine as BeenHit drop Ball end
```

You may also want to target many sprites at once. In this case you can target a group of sprites using the "**into"** keyword in your sprite definition and the **targets** keyword instead of **target**.

```
create sprite from arrow.png as Arrow
  where center=50,12 x=50 y=100 hashit=HasHit
  having targets=Balls
  alt=(sub create vector where speed=100 end)
end
create sprite from ball.png as Ball1 into Balls
  where center=25,25 x=250 y=100 beenhit=BeenHit
end
create sprite from ball.png as Ball2 into Balls
  where center=25,25 x=350 y=100 beenhit=BeenHit
end
create routine as Start
  launch Arrow
  launch Ball1
  launch Ball2
end
```

```
create routine as HasHit
     update sprite Arrow where alpha=Arrow.alpha-.35
end
create routine as BeenHit drop _sprite end
```

In the example above, both ball sprites are put **into** the same group, *Balls*. The arrow uses **targets** to detect collisions with any sprite in the *Balls* group.

Tips, Tricks, and Gotchas

- Use 'and' in routine property lists but it is optional in declarations. 'and' is not required here:

  ```
  create rotation where speed=50 distance=100 end
  ```

  but 'and' is required here:

  ```
   update sprite _sprite where x=40 and y=30
  ```

- No spaces before or after periods or equal signs
- A list of children with only one item must not be in parenthesis.

  ```
  having children=ball
  ```

- A list with more than one child requires parenthesis.

  ```
  Having children=(eyes,mouth,nose)
  ```

- Don't quote the file name in sprites declarations
- Don't let sprites linger off-screen. Remove them from the animation using **drop**
- Comment your code using two slashes **//** for code on a single line or within **/* comment */** for code across multiple lines.

  ```
  // This is a comment on one line
  /* This is a comment
     across two lines */
  ```

SCL Reference Guide

Angles

Rotation is based on the unit circle - just like high school math.

Things to know:

- 0 zero degrees points towards the right
- Degrees increase by moving **counter**-clockwise
- There are 360 degrees in the circle
- 0=360
- SCL assumes all sprites are angled at 0 degrees when drawn on screen

Bezier Curves

The Bezier alteration will move a sprite along a curve that is specified with a path.

Curves are implemented using cubic Bezier curves. To work, they require a start point, an end point, and 2 guide points between them.

- Curve coordinates are mapped relative to the sprite's location when the alteration is applied
- **start** is a point separate from the main coordinate list (**path**.) **start** acts are a guidepost. It has nothing to do with where your sprite is or

will be. Let me explain: If you use a drawing program to design a path, you'll draw the curve on the canvas of the drawing program. But you don't know how your sprite will map to that canvas. So the **start** point lets you tell SCL that when designing the curve you started at that point so SCL can map all the later coordinates relative to that point, then apply those points to where the sprite actually is.

- Each coordinate is mapped by subtracting the **start** point position from its value. This gives a *relative* position to the current position of the sprite.
- A simple way to design your curves is in GIMP (gimp.org). Use the curve drawing tool then export the curve as a text file and edit the curve description.
- Since coordinates are relatively mapped, when designing your curve you don't need to guess where your sprite might be when you actually want to use the curve.
- You can specify **orientation** as being **none**, **normal** or **tangent**. This describes the angle that the sprite will hold as it follows the curve
- If you need the sprite to loop the curve infinitely, then set **laps** to -1

```
create routine as Start
  launch curver
end
create sprite from attacker1.png as curver
  where x=640 and y=80
  having alt=bez1
end
create bezier as bez1
  where start=775,81 orientation=tangent time=3000 laps=2
```

```
path="775,152 670,191 611,157 534,112 538,65 478,93 291,244
287,68 223,85 -31,297 24,50 39,48 104,8 120,213 403,54 537,-2
537,149 718,37 748,28 776,18 775,81"
end
```

Parameters

completion={name}

a routine or alteration that begins after all the laps of the curve have been traversed by the sprite.

laps={number}

how many times sprite should follow the path. For unlimited, use -1

orientation={none|normal|tangent}

indicates how to angle the sprite on the path

path={"x,y "...}

a series of points separated by spaces. The first two points are guides leading from **start**. Do not put a linefeed or unnecessary spaces inside the path value.

```
path="775,152 670,191 611,157”
```

start={number,number}

x,y position from which to compare path coordinates. See the notes above

time={number}

a rough guide to how long it should take the sprite to traverse the path once.

Child Sprites

SCL allows you to create unlimited parent/child sprite relationships. A child sprite is displayed in a parent sprite and animates the same way as its parent. But beyond this, a child sprite can have its own alterations that do not affect its parent sprite.

Child sprites are normally specified in the definition of the parent sprite.

```
create sprite as face having children=(eyes,nose,mouth) end
```

All child sprites must be defined in the normal way, however the x/y position of the child sprite is relative to the center point of the parent sprite. **IMPORTANT:** more than one child sprite must be listed inside parenthesis, but a single child MUST NOT be in parenthesis.

Another useful way to specify a parent sprite is in routines using "**set parent={spriteName}**". That instruction forces all sprites launched after the command to be children of the sprite given. To cancel this command use "**set parent=_none"**. Animations that want to transition in an entire scene may find this very useful.

You can also add and remove child sprites in routines using the **remove** and **insert** instructions. See how in the reference section.

Cloning Sprites

Cloning is one of the most powerful features of SCL. When you clone a sprite, the new sprite gets all the properties and alterations of the original sprite. You can clone either a **current** sprite or an **original** sprite as it was originally defined. A clone can be assigned a name using **"as *cloneName*"**, but this is optional. Immediately after you clone a sprite you can refer to the

clone to make updates by using **_clone** instead of a sprite name, as in **"update sprite _clone where x=7"**.

```
create sprite as Empty
  where x=30 y=30
end
create routine as Start
  clone from Empty using original
  update sprite _clone where image="ducky.png"
  update sprite _clone where x=50 and y=80
end
```

Note that you cannot update an alteration. If you want your clone to have a different alteration, you must remove or insert alterations into your .

You can assign a name to a clone using **"as"**: `clone from Empty as Ducky using original`. If you name a clone then you can reference it like any other sprite. You should not assign a name if the same clone statement will be used repeatedly.

Please note the **_clone** keyword. **_clone** always refers back to the most recently created clone. The value of **_clone** can be used until a new clone is created - even in other routines.

Data Structures in SCL

There are two types of data structures in SCL: **Lists** and **key/value pairs**.

Data Lists

Lists let you keep a set of values that you can access in loops. There are many special commands that help you go through a list.

```
create data from list as MyList
set
  “yellow” “red” “blue”
End
```

Data Pairs

Key/value pairs let you access values using a 'key' name that you give the value. This is a convenient way to store and retrieve data that is used through-out your program.

```
create data from pairs as MyPairs
set
   x=100 y=222 z="hi there"
end

create routine as OnClick
   update sprite _sprite set x=data.MyPairs.x
   update sprite _sprite set y=data.MyPairs.y
   var data.MyPairs.y=333
end
```

List Processing

List processing is done inside routines with these commands.

current retrieves the current value

```
update sprite Dog where x=data.MyList.current
```

next retrieves the next value in the list

```
var nextVal=data.MyList.next
```

prev retrieves the previous value in the list

```
var prevVal=data.MyList.prev
```

select retrieves a random value from the list

```
update var curId set=data.Ids.select
```

pluck retrieves a random value from the list, and will not retrieve that value again. Use "**update data MyList renew**" to reset the list and pluck it again.

```
var nextCard=data.CardDeck.pluck
```

tofirst move the pointer back to the first item in the list. When you move through a list using **data..next**, a pointer tracks where you are. To return to the start of the list, use **tofirst**

```
update data MyList tofirst
```

Changing Lists

push will add a value at the end of the list.

```
update data MyList push dogSprite.x
```

pop will remove the value at the end of the list and return it. (ex: update var NAME set data.LISTNAME.pop) fush will add a value to the beginning of the list.

```
update sprite _sprite where angle=data.MyAngles.pop
```

fop will remove the value from the beginning of the list and return it.

```
update sprite _clone where angle=data.MyAngles.fop
```

remove will remove the first instance of the given value from the list
remove _all will remove all items from the list

```
update data MyList remove "red"
update data YourList remove _all
```

reset will reset the list to its original state (as given by its 'create' statement).

```
update data MyList reset
```

renew will re-randomize the list. This is useful when using *pluck* or *select*

```
update data MyList renew
```

set will set a particular list item value. The value to change is specified by the position (1-based) that it appears in the list. The first item is number 1.

```
update data MyList set 3="new value"
```

Getting List Information

Data also supports some information requests in the form **data.LISTNAME.*request***

- **contains({test value})** returns true if the test value exists in the list.

```
var result=data.MyList.contains("happy dog")
if (result==true)
  launch Bone
```

```
endif
```

- **count** returns the number of items in the list

```
var listCount=data.MyList.count
```

- **current** returns the current item in the list without changing where you are in the list.

```
var currentSprite=data.SpriteList.current
```

- **first** returns the value of the first item in the list

```
var firstItem=data.MyList.first
```

- **fopped** returns the value that was most recently returned using fop. IE: fopped items are items just removed from the front of the list using **fop**
- **get({position in list})** get the value at the given position in the list (1-based indexing).

```
var item7=data.MyList.get(7)
```

- **last** returns the value of the last item in the list

```
var last=data.MyList.last
```

- **popped** returns the value that was most recently returned using pop

```
var popped=data.MyList.popped
```

- **position** returns a number from 1 to {n} indicating at what position you are in the list as you move through it.

```
var pos=data.MyList.position
```

- **remaining** returns the number of items left as you move through it

```
var firstItem=data.MyList.last
```

Calling Routines with Data

You can pass data to routines in two ways: using **repeat** or by passing the data object as a parameter.

```
run routine FinalizeMonkeys where repeat=data.MonkeyList
create routine as FinalizeMonkeys
  var nextMonkey=data.MonkeyList.next
// … do stuff …
End
```

repeat will cause *FinalizeMonkeys* to be run repeatedly until the end of the data list is reached. Inside the routine, we assign the next value in the list to a local var, in this case *nextMonkey.*

IMPORTANT: It is vital that you call **data.MonkeyList.next** inside the routine else your animation will call the same routine forever using the same value from the data object.

The second method to pass data to routines is to pass the data object as a parameter. For example:

```
create data from list as MyData set 1 2 3 end
create routine as Start
  MyR(data.MyData)
End
create routine as MyR
  var dataObj=arg.1
```

```
  update sprite MySprite where x=dataObj.next
end
```

In the above example, the data is passed as a parameter and the parameter is assigned to a local var. This makes is available to the routine as a normal(ish) object.

```
var PList=arg.1
update sprite MySprite where x=PList.next
```

Setting up levels in SCL

Levels are supported in SCL by assigning data or variables *into* levels. You then select a level by setting the global *level* variable. Names for data and variables assigned to different levels have the same name, but different level names. Then when a value is requested by a given name, the value for that level is used. Levels can be names or numbers.

Example

```
create data from list as charNames into Level1
set
  "Sue" "Mary" "Johnny" "Casper"
end
create data from list as charNames into Level2
set
  "Larry" "Steward" "Ann" "Nadine"
end
create var as EastGuy into Level1 where value="ED" end
create var as WestGuy into Level2 where value="STEVE" end

create routine as Start
  set level=Level1
  log(data.charNames.next) // prints "Sue"
  log(EastGuy) // prints "Ed"
```

```
  set level=Level2
  log(data.charNames.next) // prints "Larry"
  log(WestGuy) // prints "STEVE"
end
```

Drawing

You can use basic drawing calls directly in SCL. You can draw onto sprites, onto the bottom of the canvas, or on top of the canvas overtop of all sprites.

You draw inside a routine. For example:

```
create routine as DrawBackground
  draw onto _bottom
    fillstyle "rgba(255,0,255,0.65)"
    fillrect 430,140,40,40
    filltext "Hello",10,10,100
  enddraw
end
```

Draw commands are all small-case, followed by a space, then parameters separated by commas. Strings for text or CSS styles must be double-quoted. The above example demonstrates this.

Warning: expressions (like math) are have only experimentally supported in draw commands because of performance considerations.

Drawing onto **_bottom** will draw on the canvas before any sprites are drawn.
Drawing onto **_top** will draw on top of everything after everything else is drawn.
Drawing onto a sprite (specified with its name) will draw onto the sprite with all the sprites alterations applied to the drawing. That means, by drawing on sprites, you create mini-canvases for drawing.

Drawing is sticky. Once you draw on a sprite or onto _bottom or _top, that drawing stays until it is cleared with "clear".

You are strongly advised to read the W3 2D Context drawing specification for how some of these command work.

Supported calls are:

- **arc** -
- **arcto** - extends the current path as an arc
- **beziercurveto** - draws a curve using the bezier method
- **beginpath**
- **circle** - creates a circle at x,y with radius: circle 100,230,50
- **clear** - clears the instruction list (see below)
- **clearrect** - clears a rectangle of the given size
- **clip** - creates a clipping region based on current path
- **closepath**
- **composition**
- **fill** - fills the current path with the current fillstyle
- **fillrect** - fills a rectangle with the current fillstyle
- **fillstyle** - sets a fill color which is used to fill an area created by paths

- **filltext** - draws given text using the current fillstyle
- **floodfill**
- **font** - defines font to use. See docs reference below for settings. ex ("italic 400 12px/2 Unknown Font, sans-serif"). That format is "[style] [variant] [weight] [size/line height] [font]"
- **image** - draws an image at the specified x/y: **image dog.png,0,0**
- **lineto** - draws a line from the current position to the given coordinate
- **linewidth** - set the line width for strokes made after
- **moveto** - moves your current drawing tip to the given coordinate
- **quadraticcurveto** - draws a curve using the quadratic method
- **rect** - creates a rectangular subpath
- **shadowblur** - set shadow under text blur value (1 is slight blur, 20 is very blurry, 20+ is possible
- **shadowcolor** - set shadow color
- **shadowoffsetx** - set shadow under text x-offset
- **shadowoffsety** - set shadow under text y-offset
- **stroke** - outlines the current path with the current strokestyle
- **strokerect** - draws a rectangle outline with the current stroke style
- **strokestyle** - sets a stroke color which is applied to lines draw with paths or text
- **stroketext** - draws the given text using the current strokestyle
- "top","middle","hanging","alphabetic","ideographic","bottom"
- **textalign** - set to "center", "left", or "right"
- **textbaseline** - set to

Read the W3 drawing specification for details but be aware of some basics with SCL:

- All the names of draw commands are completely small case

- After the command name, leave a space, then a list of parameters separated by commas. Do not use an equal sign or paranthesis.
- Using moveto and lineto doesn't draw anything unless you follow the list of commands with "stroke" or "fill".
- Once you draw something, it stays until you call "clear". That clears all the drawn items from the target (the target being "_top", "_bottom" or a sprite name)
- Multiple lines can be separated with \n In this case, vertical spacing of lines will be based on the most recent call to 'font' where a px value was specified.

Fade

The Fade alteration will cause the sprite to incrementally change its transparency.

This effects the alpha value of the sprite. Therefore, to fade, you must specify a negative value for the rate and to unfade, use a positive value.

```
create routine as Start
  launch horse
end
create sprite from horse12.png as horse
  where x=100 y=100
  having alt=(sub
    create fade as f where rate=-0.5 and until=0.1
  end)
end
```

Parameters

rate={number}

a fade rate per second. Use a negative value to fade (increase transparency) and a positive value to make the sprite more opaque.

until={number}

the value at which the fade should stop. Also, the final alpha value for the sprite. Must be a value between 0.0 and 1.0. Be sure this number makes sense with respect to the initial alpha of the sprite and the rate being negative or positive.

completion={routine}

specify a routine to run when the fade is complete.

Flipbooks

The Flipbook alteration will change the image of your sprite repeatedly to create a simple flipbook animation.

```
create routine as Start
  launch rider
end
create sprite from horse12.png as rider
  where x=100 y=100
  having alt=flipHorse
end
create flipbook as flipHorse
  where loop=true and
```

```
images=("horse1.png","horse2.png","horse3.png","horse4.png","hor
se5.png","horse6.png","horse7.png","horse8.png","horse9.png","ho
rse10.png","horse11.png")
end
```

Parameters

completion={routine}

specify a routine to run when the last frame is complete (non-looping)

frames=({number},{number},...})

a list of frame numbers in the order in which the frames should display. The first frame is number 1. You can also use **random** as a value to randomize frames.

images=({string},{string},...})

list of images to be shown for each frame. Image names must be quotes, with no spaces around the commas.

keeplast={true|false}

specify if the sprite should keep the last frame as its image after the flipbook is done running

loop={true|false}

specify if the flipbook should repeat continuously

times=({number},{number},...})

specify how long each frame should be displayed. The default is 50ms per frame.

Functions

Use functions in place of values for some automatic calculations. Example:

```
update sprite MySprite where x=random(0,50)
if (covered(MySprite))
  update sprite MySprite
    where
angle=getpolard(MySprite.x,MySprite.y,AnotherSprite.x,AnotherSprite.y)
endif
```

- **covered(spriteName)** - returns true if the named sprite is covered by another sprite's bounding box
- **arcdistance(x1,y1,x2,y2)** - calculates the distance between two points on an arc
- **distance(x1,y1,x2,y2)** - calculates the pixel distance between two points
- **getdata(dataName,#)** - returns value in the # position of the data structure (1-based)
- **getglobalx(spriteName)** - returns the global x-position of the sprite (useful if the sprite has a parent)
- **getglobaly(spriteName)** - returns the global y-position of the sprite (useful if the sprite has a parent)
- **getpolard(x1,y1,x2,y2)** - get angle from point 1 to point 2
- **getpolarx(x,angle,distance)** - given an x coordinate, return a new x value the given angle and distance away
- **getpolary(y,angle,distance)** - given an y coordinate, return a new y value the given angle and distance away
- **iskeydown("n")** - returns true if the letter (in quotes) is currently pushed down
- **islive(spriteName)** - returns true if the sprite has been launched and is still on the canvas

- **Math**.*(…) – if you are familiar with JavaScript, you can also use the built in Math object directly. This gives you access to all of its static methods like Math.floor() or Math.cos().
- **random(n)** - returns a random number between 0 and n (not including n)
- **timems()** - returns the time, in milliseconds, that the animation has been running (not including pauses)

Joystick

A joystick object creates a virtual touch area on the screen which automatically tracks a user's touch movements. When you create a joystick you specify a center location for the stick. You may also provide a sprite that follows the user's touch movements.

Joysticks in absolute mode have a radius around their centers inside which touches are registered. Once a touch is detected within a joystick's radius, an event is fired for which a routine can read various joystick parameters.

In **absolute** mode, joystick position values are relative to the center positions of the joystick. In the other mode, **relative**, the position values are relative to the last position of the joystick.

Joystick event values give position and angles of the joysticks position via a **_joy** variable.

```
create joystick as joy1
  where
    center=150,100
```

```
    sprite=joysprite
    radius=100
    move=OnMove
    mode=absolute
end
```

Parameters

active=[true|false]

whether or not joystick is currently accepting user input.

center={number,number}

the location of the joystick on the canvas.

mode=[relative|absolute]

relative (default) or absolute. Indicates if _joy expression values are relative to the last stick position or to the center of the joystick.

move={routine name}

the routine to run when the joystick is moved.

```
create routine as OnJoyStick
  if (_joy.direction=="up")
    update var NewManDir set=1
  elsif (_joy.direction=="right")
    update var NewManDir set=2
  elsif (_joy.direction=="down")
    update var NewManDir set=4
  elsif (_joy.direction=="left")
    update var NewManDir set=8
  endif
  if (_joy.compass=="nw")
    Whatever()
```

```
  Endif
end
```

radius={number}

radius for detecting a touch on the joystick when mode is absolute.

sprite={sprite name}

a sprite to move with the drag of the joystick.

The routine called by 'move' will have access to the values below.

Joystick Values

_joy.angle

Angles are based on the unit circle where up is 90° and towards the right is 0°

- **Absolute Mode**: the angle of this position compared to joystick's **center** position
- **Relative Mode**: the angle of this position compared to the last joystick position

_joy.distance

the distance of the joystick's current position to the joystick's **center** position

_joy.x

the canvas x position of the touch

_joy.y

the canvas y position of the touch

_joy.deltax

the change in x position since last touch movement

_joy.deltay

the change in y position since last touch movement

_joy.percent

the percentage of the distance from the center of the joystick to the radius for the joystick

_joy.direction

doubled-quoted: up, down, left or right

```
if (_joy.direction=="up") exit endif
```

_joy.compass

doubled-quoted: n,ne,e,se,s,sw,w, or nw

_joy.moved

the distance the stick moved since its last position

Mouse Events

For desktop games created with SCL, most likely you will need to support mouse events. SCL supports two levels of mouse support for 5 different mouse events. The levels are **canvas level** and **sprite level**. Canvas level events apply to mouse events anywhere on the canvas. Sprite level events apply to individual sprites. The events supported at both levels are:

- **click** - fired when mouse is depressed and released on the canvas or sprite
- **mousedown** - fired when the mouse is depressed on the canvas or sprite
- **mouseup** - fired when a depressed mouse is released on the canvas or sprite

- **mousemove** - fired when a mouse is moved within the canvas or sprite
- **mousein** - fired when the mouse enters the canvas or the sprite
- **mouseout** - fired with the mouse exits the canvas or sprite

```
create routine as Start
  set mousedown=OnCanvasMD
  set mouseup=OnCanvasMU
end
create routine as OnCanvasMD
  log("mouse down at "+mouse.x+","+mouse.y)
end
create routine as OnCanvasMU
  log("mouse up at "+mouse.x+","+mouse.y)
end
```

As above, to set a mouse event for the canvas, use the **set** command inside a routine. You can access the location of the mouse click using the build-in variables, **mouse.x** and **mouse.y**

To set mouse events for individual sprites, use the same event names in the **where** clause of the sprite. Each sprite tracks its own mouse events. These events only fire if they occur within the sprite boundary and over an area of the sprite that is not completely transparent (alpha>0).

```
create sprite from image.png as S1
  where x=300 y=300 click=OnSClick
end
create routine as Start
  launch S1
end
```

```
create routine as OnSClick
  log("mouse up at "+mouse.x+","+mouse.y)
  log("Sprite loc of mouse up: "+smouse.x+","+smouse.y)
end
```

As above, you can access the x/y location of the click relative to the center of the sprite by using the built-in variables smouse.x smouse.y

You can change an event handler or cancel them altogether by specifying **_none** as the event handler.

```
create routine as Example
  set mousemove=_none
  update sprite MySprite where click=OnHappyClick
  update sprite YourSprite where click=_none
end
```

Normally, the top-most sprite will receive the click event. If you would like a sprite to ignore mouse events, thus allowing sprites under it to receive the event, use **_ignore** for the **click** event handler. This setting also effects touch events the same way.

```
create sprite from image.png as S1
  where x=300 y=300 click=_ignore
end
```

Pursuits

The Pursuit alteration will cause a sprite to follow another sprite. If no speed is given then the pursuing sprite will not move but only point at the sprite that it is following.

```
create routine as Start
  launch ball
  launch arrow
end
create sprite from arrow1.png as arrow
  where x=150 y=170 center=50,26 angle=90 size=0.25
  having alt=pursuit1
end
create pursuit as pursuit1
  where prey=ball speed=230 follow=300
end
create sprite from ball1.png as ball
  where x=150 y=30 center=25,25
  having alt=move1
end
create translation as move1
  where x=random(0,300) y=random(0,300) speed=100
completion=DoneTrans
end
create routine as DoneTrans
  insert into ball where alt=move1
end
```

Parameters

follow={number}

speed of rotation toward prey in degrees per second

prey={sprite}

the name of the sprite to follow

speed={number}

speed of pursuit in pixels per seconds

Range

The Range alteration triggers events based on one sprite's proximity to another.

IMPORTANT: The target sprite must be on screen before this alteration is created!

```
create routine as Start
  launch targetArrow
  launch arrow
end
create sprite from uparrow.png as targetArrow
  where x=200 y=200 center=103,27 angle=45
end
create sprite from uparrow.png as arrow
  where x=50 y=10 center=103,27 angle=340
  having alt=moveArrow alt=TrackRange
end
create vector as moveArrow
  where speed=120
end
create range as TrackRange
  where target=targetArrow distance=200 enter=near exit=leave
arc=90
end
```

```
create routine as near
  update sprite targetArrow where alpha=0.5
end
create routine as leave
  update sprite targetArrow where alpha=1.0
end
```

Parameters

arc={number}

You can set an arc "in front" of your sprite such that the target sprite must be within that arc for events to fire. Imagine a cone extending from the front of your sprite. The arc sets this cone. The front is considered to be the angle at which the sprite is pointing. For example, if your sprite is currently at 45 degrees and the arc=20, then the target must be between 35 and 55 degrees for your sprite in order to register a range event.

distance={number}

the number of pixels distance from the center points of the sprites to be considered in range.

enter={routine}

specify a routine to run when the target sprite enters range. This can fire multiple times if no completion event is set

exit={routine}

specify a routine to run when the target sprite leaves range. This can fire multiple times if no completion event is set

target={sprite}

the name of the sprite to watch

completion={routine}

specify a routine to run when the target sprite enters range. If this is set then this alteration will close upon completion. This means that no exit event will fire and no more enter events will fire if the target sprite leaves then re-enters the range.

Regions

Regions are rectangular areas of the canvas that can detect mouse events.

```
create region as R4
  where x=600 y=0 w=200 h=300
    mousein=OnR4MouseIn mouseout=OnR4MouseOut
end
```

Regions must be launched in order to be active. Once launched, region mouse events will fire until the region is dropped.

```
create routine as Start
  launch R4
end
create routine as OnR4MouseIn
  drop R4
end
```

Parameters

x={number}

The left edge coordinate of the regions

y={number}

The left edge coordinate of the regions

w={number}

The left edge coordinate of the regions

h={number}

The left edge coordinate of the regions

click={routine}

The routine to call when the mouse is called in the region

mousein={ routine }

The routine to call when the mouse enters the region

mouseout={ routine }

The routine to call when the mouse exits the region

mousedown={ routine }

The routine to call when the mouse button is depressed in the region

mouseup={ routine }

The routine to call when the mouse button is released in the region

mousemove={ routine }

The routine to call when the mouse is moved in the region

Reveal

The Reveal alteration hides or shows the images as if appearing from behind something.

```
create routine as Start
  launch ballShow
  launch ballHide
```

```
end
create sprite from ball1.png as ballShow
  where center=25,25 x=150 y=50
  having alt=show
end
create sprite from ball1.png as ballHide
  where center=25,25 x=150 y=150
  having alt=hide
end
create reveal as show
  where direction="left" speed=20 type="show"
end
create reveal as hide
  where direction="right" speed=20 type="hide"
end
```

Parameters

type="{show|hide}"

should the sprite enter or leave the view with this alteration.

direction="{top|bottom|right|left}"

from which direction to enter or leave.

speed={number}

the speed with which to move the sprite

completion={routine name}

the name of a routine to run with the operation is complete.

Rotation

A Rotation is an alteration that will rotate a sprite around its center point. It has a number of interesting options in order to get your sprite angled how you want it.

When applying a rotation to a child sprite, the angles are relative to the sprite's direct parent sprite, not the screen.

```
create routine as Start
  launch spinner
end
create sprite from windmill-180x180.png as spinner
  where center=90,90 x=350 y=160
  having alt=rot1
end
create rotation as rot1
  where speed=-100 distance=720 easein20 easeout=40
end
```

Parameters

speed={number}

speed of rotation in degrees per second. This can be a negative value to rotate clockwise.

distance={number}

the distance to rotate - **positive values only - speed sets direction. OR,**use ***distance=short*** to travel the shortest distance to the given destination. In this case, the sign of **speed** is ignored.

destination={number}

rotate until this angle is reached

toward={number},{number}

rotate until pointing at this x,y coordinate

easein={number}

gradually increase speed over this distance (in degrees).

easeout={number}

before reaching distance, gradually decrease speed over this distance (in degrees).

completion={routine}

specify a routine or another alteration to run when the rotation stops

Routines

Routines make things happen. The first routine run is called **Start**. It is case-sensitive.

The structure of a routine is basically a list of things to do.

Routines are run in response to events that you attach to user actions, sprites, or alterations. Routines can also be called by other routines.

4 special variables exist in a routine:

1. **_sprite** represents the sprite that called this routine through an event. Consider it to be the default sprite for future instructions.
2. **_clone** refers to the clone most recently created in the routine
3. **_repeat** contains the which iteration (number) of a "repeat" is currently running
4. **_none** will deactivate an event, ie: click=_none

```
create routine as MyProc
  var vSprite=arg.1
  if (vSprite.name="cliff")
```

```
    exit
  elsif (vSprite.name="dog")
    return "happy dog"
  else
    update sprite s1 where xscale=2.5
  endif
  open url "http://example.com/" into "exampleWindow"
  run routine OtherProc where repeat=5
end
```

Routine commands

as {string}

name assigned to routine - **required**

set click={routine}

set a routine to run when the mouse is clicked on the canvas.

set enterframe={routine}

set a routine to run when every animation frame is entered before any sprites are processed

set exitframe={routine}

set a routine to run when every animation frame is exited after processing all sprites

set keydown={routine}

set a routine to run when a key is pushed down. Deactivate with **_none**.

set keypress={routine}

set a routine to run when a key is pressed and released. Deactivate with **_none**.

set keyup={routine}

set a routine to run when a key is released. Deactivate with **_none**.

set mousein={routine}

set a routine to run when the mouse enters the canvas.

set mousedown={routine}

set a routine to run when the mouse is depressed anywhere on the canvas.

set mouseup={routine}

set a routine to run when the mouse is released anywhere on the canvas.

set mousemove={routine}

set a routine to run when the mouse moves on the canvas.

set mouseout={routine}

set a routine to run when the mouse exits the canvas.

set offset={x,y}

offsets the coordinates of the canvas origin but the given amount.

set parent={sprite}

set a sprite to be the parent of all sprites launched hereafter. Set this value to **_none** to remove the default parent.

set pinch={routine}

set a routine to run when the canvas is pinched.

set pivot={routine}

set a routine to run when the canvas is rotated with a 2 finger pinch.

set _sprite={sprite}

set _sprite to a different default sprite to be used in subsequent run calls.

set swipeup={routine}

set a routine to run when the canvas is swiped upwards.

set swiperight={routine}

set a routine to run when the canvas is swiped rightwards.

set swipedown={routine}

set a routine to run when the canvas is swiped downwards.

set swipeleft={routine}

set a routine to run when the canvas is swiped leftwards.

set touchstart={routine}

set a routine to run when a touch event on the canvas occurs.

set touchend={routine}

set a routine to run when a touch event on the canvas ends.

set touchmove={routine}

set a routine to run when a touch is moved across the canvas.

launch {sprite}

add the given sprite to the canvas

insert into {sprite} **where alt**={alteration}

applies a named alteration to the sprite. Sprite can be a sprite name, _sprite, _clone, or a data value. **alt** can also be an anonymous routine defined inline using the "**sub**" method described in the section on chaining alterations

insert into {sprite} **where children**={(sprite names)}

makes the listed sprites children of the given "into" sprite. If there is more than one child then list the children inside parentheses, else list a single child without parentheses.

insert into {sprite} **where attach**={(sprite names)}

Like 'children' except the child position will remain at some screen location after attachment.

remove from {sprite} **where alt**={alteration}

remove the named alteration from the named sprite. Use **_all** to remove all alterations from a sprite.

remove from {sprite} **where type**=_all

remove all alterations from a sprite

remove from {sprite} **where children**={(sprite names)}

remove the named children from the named sprite. **_all** is also supported. If there is more than one child then list the children inside parentheses, else list a single child without parentheses.

remove from {sprite} **where detach**={(sprite names)}

removed children remain at same screen location after detachment. If there is more than one child then list the children inside parentheses, else list a single child without parentheses.

var {name}={value}

assign a variable that will only exist inside this routines. Parameters passed into a routine should be saved to a local variable with this instruction. IMPORTANT: this "var" is unrelated to the "var" created outside of a routine. Also, **var** must be used every time you want to assign a value to the variable even if you have used it before. **var** is an instruction, not a declaration.

update var {name} **add**={number}

adds any number to the named global var

update var {name} **subtract**={number}

subtracts any number from the named global var

update var {name} **set**={number}

set any number to the named global var

update var {name} **reset**

restores a variable to its creation settings

update sprite {name} **detach**

detaches a child sprite from its parent, but it stays at same screen location.

update sprite {name} **where image**={image file}

changed the image of the sprite to the file given. This image may not be preloaded by the browser if you have not used the image previously.

update sprite {name} **where x**={number}

change the given value of the name sprite as indicated

update sprite {name} **where y**={number}

change the given value of the name sprite as indicated

update sprite {name} **where size**={number}

change the given value of the name sprite as indicated

update sprite {name} **where xscale**={number}

change the given value of the name sprite as indicated. **FireFox may fail** to launch completion events if sprite size reaches 0.0. Therefore, also set a minimum size of 0.01 or something smaller if necessary.

update sprite {name} **where yscale**={number}

change the given value of the name sprite as indicated **FireFox may fail** to launch completion events if sprite size reaches 0.0. Therefore, also set a minimum size of 0.01 or something smaller if necessary.

update sprite {name} **where alpha**={number}

change the given value of the name sprite as indicated

update sprite {name} **bringtotop**

brings the sprite to appear above all others on the canvas

update sprite {name} **where angle**={number}

change the given value of the name sprite as indicated

update sprite {name} **where center**={number}

change the given value of the name sprite as indicated

update sprite {name} **where click**={routine name}

change the given value of the name sprite as indicated

update sprite {name} **where composition**={value}

change the given value of the name sprite as indicated

update sprite {name} **where hflip**={number}

change the given value of the name sprite as indicated

update sprite {name} **where mouseup**={routine name}

change the given value of the name sprite as indicated

update sprite {name} **where mousedown**={routine name}

change the given value of the name sprite as indicated

update sprite {name} **where size**={number}

change the given value of the name sprite as indicated

update sprite {name} **where swipeup**={routine name}

change the routine when the sprite is swiped on upwards

update sprite {name} **where swiperight**={routine name}

change the routine when the sprite is swiped on rightwards

update sprite {name} **where swipeleft**={routine name}

change the routine when the sprite is swiped on leftwards

update sprite {name} **where swipedown**={routine name}

change the routine when the sprite is swiped on downwards

update sprite {name} **where target**={number}

change the given value of the name sprite as indicated

update sprite {name} **where targets**={number}

change the given value of the name sprite as indicated

update sprite {name} **where touchdrag**={sprite name}

make the sprite draggable by touch

update sprite {name} **where vflip**={number}

change the given value of the name sprite as indicated

update sprite {name} **set** {variable}={value}

change the given sprite variable to the value indicated

update textsprite {name} **where text**={"string"}

change the text of the textsprite as indicated

pause sound {name}

pauses a sound that is playing.

pause wait {name}

pauses a wait alteration.

unpause sound {name}

unpauses a sound that is playing.

unpause wait {name}

unpauses a wait alteration.

if ({expression}) {new lines of code } **endif**

if the expression evaluates to true, do the lines of code below the current line but above endif, elsif, or else

elsif ({expression}) {new lines of code }

if the expression evaluates to true, do the lines of code below the current line but above endif, elsif, or else

else {new lines of code }

if no conditions are met above, do this block of code above endif.

endif

close an 'if' block

drop {sprite}

remove the named sprite or region from the canvas

open url "{url}" **into** {"targetBrowserWindow"}

open the given url in the named browser window

run routine {name} **where repeat**={number}

runs another routine (called a child routine) the number of times given by "repeat". Two **special variables** are associated with routines. **_repeat** is defined in the child routine and indicates which instance of the repeat is currently runnin. _repeat=1 means this is the first time through. **However** if it is set to the name of a list (such as "repeat=data.MyList") then _repeat will represent which item in the list is being used (ie: _repeat=0 if no list items were used, or _repeat=5 if 5 list items have been used). **_sprite** exists in all routines spawned by a sprite event and indicates which sprite spawned this routine. You can use **_sprite** in place of a sprite name.

reset all

remove all sprites from canvas, resets all variables, and sets the animation back to its original state. This is often followed immediately by **Start()** which would cause the animation or game to play again.

clone from {sprite} **as** {name} [**using** {current|original}]

makes a clone of the given sprite using either the original definition of the sprite, or its current properties if it is already onscreen. You can change the sprite after cloning it by using the special variable **_clone**. The _clone variable always refers to the most recently created clone in the current routine.

trace into {"id"} **where value**={value}

displays the given value in an HTML div section specified by "id". This is a valuable debugging tool. You can also use the log() command but this requires a div called 'log' to exist on the page.

log({value}**)**

Will send the value to be displayed in a div called "log", if it exists.

exit

causes the current routine to stop processing any more commands and exit.

return {value}

causes the current routine to stop processing and return the value given.

and

Commands like 'update' normally only change one value per command. However you can change more than one if you separate each value with "and". If you omit 'and' then you must have a separate "update" for each value that you want to change.

Scale

The Scale alteration will cause the sprite to incrementally change size. You can scale on the x or y axis or both.

FireFox may fail to launch completion events if sprite size reaches 0.0. Therefore, also set a minimum size of 0.01 or something smaller if necessary.

```
create routine as Start
  launch horse
end
create sprite from horse12.png as horse
  where x=100 y=100
  having alt= MyScale
end
create scale as MyScale
  where rate=-0.5 until=0.1 type=y
end
```

Parameters

rate={number}

a scale rate per second. Use a negative value to shrink and a positive value to grow the sprite.

type={x|y}

specify in which direction to scale the sprite. If type is not specified the scaling is performed both ways.

until={number}

the value at which the scaling should end. Be sure this number makes sense with respect to the initial size of the sprite and the rate being negative or positive.

completion={routine}

specify a routine to run when the scaling is complete.

Shear

Shearing is best understood by example.

WARNING: FireFox cannot always correctly run a shearing effect.
This bug in FireFox has been submitted to Bugzilla (661452)
See the tutorial for more details.

```
create shear as MyShear
  where rate=3 until=1 type=x completion=Done
end
```

Parameters

completion={routine}

specify a routine to run when the shearing is complete.

rate={number}

a shear rate per second. Use a negative value as appropriate.

type={x|y}

specify in which direction to shear the sprite. If type is not specified the shearing is performed both ways.

until={number}

the value at which the shearing should end. Be sure this number makes sense with respect to the initial size of the sprite and the rate being negative or positive.

Slides

The Slide alteration will cause the sprite to move in the direction given for the given distance.

```
create routine as Start
  launch horse
end
create sprite from horse12.png as horse
  where x=100 y=100
  having alt=slider
end
create slide as slider
  where speed=100 direction=95 easein=20 easeout=20 distance=100
end
```

Parameters

completion={routines}

a routine to run when the slide is complete.

direction={number}

the direction to travel without regard to the sprite's current angle.

distance={number}

how many pixels to travel (optional).

easein={number}

gradually increase speed over this distance.

easeout={number}

before reaching distance, gradually decrease speed over this distance.

speed={number}

move in pixels per second.

Sound

Use "create" to create a sound object and in a routine specify which sound to play and other manipulations.

To play a sound, it must be in mp3 format for maximum portability. Some browsers may have issues with sound fade effects.

_all rather than a sound name can be used for: play, mute, unmute, pause, and unpause.

```
create sound from "music1.mp3" as sndMusic1
  where repeat=3 volume=0.4 fadein=500
end
create sound from "music2.mp3" as sndMusic2
  where repeat=2 volume=0.8 fadeout=1000
end
create routine as ChangeMusic
```

```
  stop sound sndMusic1 where fadeout=1200
  play sound sndMusic2 // no parameters supported
  pause sound sndMusic2 where fadeout=500
  unpause sound sndMusic2 where fadein=500
  mute sound sndMusic2 where fadeout=500
  unmute sound sndMusic2 where fadein=500
end
```

Parameters

from {mp files}

a double-quoted " sound file name.

as {name}

the name of the new sound object.

completion={routine name}

the name of a routine to run when the sound finishes playing.

volume={0-1}

set between 0.0 and 1.0 inclusive

fadein={milliseconds}

the time for which the fadein should last.

fadeout={milliseconds}

the time for which the fadeout should last. This may be limited by some platforms to require a value greater than 300 for best-sounding effect.

repeat={number}

number of times the sound repeats. Use a really high 9999999 number for continuous play.

Sprite

Each moving entity is called a sprite. Normally sprites are defined with an image and a name. You can create a sprite without an image then use it as a container for other things. Sprites support many parameters.
Sprites are moved using **alterations**. Alterations include **rotation**, **slide** and many more.
The general animation structure of SCL is based on this relationship.

After a sprite is on the screen it can be changed in a routine using **update sprite**

If you want to change a sprite that you just created with a **clone** command, then refer to the clone by **_clone** instead of a sprite name.

If you want to refer to a sprite that signalled an event (such as click or completion event), then use **_sprite** to refer to the current sprite.

```
create sprite from windmill-180x180.png as windmill
into windfarm
  where x=100 alpha=0.5 angle=60
    beenhit=OnBeenHit
    center=20,25 click=clickWM composition={see below}
    enterframe=clickWM exitframe=exitFrame
    hashit=OnHasHit hflip=true
    mousein=mouseInWM mousemove=mouseMoveWM
    mouseout=mouseOutWM
    parent=PlayArea penultimate=penultimateExit
```

```
    vflip=true xscale=2.5
    y=200 yscale=5.6
  having target=bumbleBee children=(fin1,fin2)
    alt=spinal,moveUp
  set memvar1="dog" memvar2=4
end
```

Parameters

into

sprites can be grouped and the groups can be used as subjects or targets of different operations (ex: collision detection).

where clause

```
create sprite as MySprite where x=7 y=334 end
```

alpha={range 0.0-1.0}

0.0 makes sprite completely transparent. 1.0 makes the sprite fully opaque.

angle={number}

at what angle to place sprite relative to the unit circle (0 degrees is default and is pointing towards the right)

beenhit={routine}

specify a routine to run when it has been hit by another sprite

center={number},{number} **OR** auto (example: center=auto)

specify the sprite's 0,0 origin relative to the 0,0 position of its image. The center given is the point around which the sprite will rotate, placed on the screen, or used as a collider.

click={routine}

specify a routine to run when the mouse click while on an opaque portion of the sprite. Set to **_ignore** to disable all user interactions on the sprite.

composition={See below}

This parameter determines how a sprite is drawn with respect to the background. It can be one of: "source-atop", "source-in", "source-out", "source-over", "destination-atop", "destination-in", "destination-out", "destination-over", "lighter", "copy", "xor". It is best to experiment to see what you need

enterframe={routine}

specify a routine to run when the frame is entered

exitframe={routine}

specify a routine to run after the **penultimate** event and after all variable conditions have been checked and all the completion routines have finished running.

hashit={routine}

specify a routine to run when this sprite has hit its target (see target below)

hflip={true|false}

flip the sprite horizontally about the center point.

mouseup={routine}

specify a routine to run when the mouse button is released over an opaque portion of the sprite

mousedown={routine}

specify a routine to run when the mouse depressed over an opaque portion of the sprite

mousein={routine}

specify a routine to run when the mouse moves onto an opaque portion of the sprite

mouseout={routine}

specify a routine to run when the mouse moves out of an opaque portion of the sprite

mousemove={routine}

specify a routine to run when the mouse moves within an opaque portion of the sprite

parent={sprite}

specify a parent sprite onto which to attach this sprite as a child

penultimate={routine}

specify a routine to run after all sprites are processed for the frame, but before the var conditions are tested

set {name}={value}

creates a variable specific only to the sprite. Do not use the same name as one of the built in properties

size={number}

scale the sprite relative to its natural size. Size=1 is normal size

touchdrag={routine}

indicates that the sprite can be dragged via touch on a touch screen device

touchend={routine}

specify a routine to run when a sprite touch is lifted off a touch screen device

touchstart={routine}

specify a routine to run when the sprite is first touched via a touch screen device

vflip={true|false}

flip the sprite vertically about the center point.

x={number}

horizontal position of sprite based on the sprite's center

xscale={number}

scale the sprite horizontally

y={number}

vertical position of sprite based on the sprite's center

yscale={number}

scale the sprite vertically

having clause

Sprite have some parameters that can have more than one value. These parameters are preceded by a "having" clause:

```
create sprite as MySprite having children=(dog,cat) end
```

alt={alteration} alt={alteration}...

apply an alteration to the sprite. A sprite can have any number of alterations applied to it by assigning **alt** repeatedly.

children=({sprite},{sprite},...)

specify a list of sprites that move and orient themselves relative to this sprite. Children sprites do not need to be launched separately but are automatically launched when its parent sprite is launched. Note, child sprites absorb events and do not relay them to the parent. Set touch events on child sprites, not the parent sprite. This will change in upcoming releases.

target={sprite}

specify a single sprite for which to trigger a collision event

targets={spritegroup}

specify a group of sprites which can all trigger collision events. Sprites are assigned to groups using "into" (see above)

Textsprites

A text sprite is like a regular sprite but instead of loading an image, it can display text using CSS notation.

```
create routine as Start
  launch example
end
create textsprite as example
  where x=100 y=100 text="Example Text"
    color="rgb(242,129,142)" blur=5
    shadow="blue" font="bold 48px sans-serif" style="fill"
    xshadow=5 yshadow=3
end
```

Parameters

NOTE: regular sprite parameters apply, plus these text specific parameters.

align="{string}"

"center", "left", or "right" in double-quotations marks.

blur={number}

the width of the blur applied to the shadow of the text - it must be in quotes to work with iOS.

color="{string}"

a color value represented in any CSS format.

ex: "#efa81c"

font="{string}"

specify font parameters using the CSS shorthand format for fonts:

ex: "normal bold 18px sans-serif"

shadow="{string}"

specify a text shadow using a color value represented in any CSS format.

style="{fill|stroke|border}"

specify how the text is drawn, as filled or just stroked (default=fill).

borderwidth={number}

when style="border", this parameter specifies the width of the border

bordercolor="{string}"

when style="border", this parameter specifies the color of the border represented in any CSS format.

ex: "#efa81c"

text="{string}"

the text to display in double-quotations marks.

xshadow={number}

the horizontal offset of the text shadow.

yshadow={number}

the vertical offset of the text shadow.

Touch Events

SCL supports many kinds of touch events across all touch-enabled devices. Touch events can be relative to the whole canvas or to a particular sprite, depending on the type of event. The table below illustrates.

Canvas Touch	Sprite Touch
• **touchstart** • **touchend** • **touchmove** • **swipeup** • **swiperight** • **swipedown** • **swipeleft** • **pinch** • **pivot**	• **touchstart** • **touchend** • **touchdrag** • **swipeup - experimental** • **swiperight - experimental** • **swipedown - experimental** • **swipeleft - experimental**

Canvas Touch Events

Touch events that apply to the whole canvas are set inside a routine using a "set" command.

```
create routine as Start
  set touchstart=OnTouchStart
  set touchend=OnTouchEnd
end
```

Sprite Touch Events

Touch events for a particular sprite are set inside a sprite definition in the **where** clause.

```
create routine as Start
  launch MySprite
end
create sprite from 2.png as MySprite
  where x=400 y=300 center=50,50 size=3
    touchstart=OnSpriteTouchStart
    touchend=OnSpriteTouchEnd
end
create routine as OnSpriteTouchStart
  insert into _sprite where alt=(sub create rotation end)
end
create routine as OnSpriteTouchEnd
  remove from _sprite where type=_all
end
```

Some touch events on sprites can be changed in a routine using **update**. For example:

```
update sprite MySprite where touchdrag=true
```

Notice how "_none" removes an event handler.

Dragging a sprite

touchdrag is a sprite property that allows a finger on the screen to drag a sprite. This property takes a *true* or *false* value.

```
create sprite from 2.png as s1
  where center=50,50 x=260 y=300 touchdrag=true
end
create sprite from 4.png as s2
  where center=50,50 x=250 y=320 touchdrag=true
end
create sprite from 8.png as s3
  where center=90,90 x=230 y=290 touchdrag=true
end
create routine as Start
  launch s1
  launch s2
  launch s3
end
```

On a touch-enabled device, you can drag sprites by touching their visible portion.

_touch variable

_touch is a special variable that gives you values related to the most recent touch event. What the values mean vary depending on the most recent type of touch event.

_touch.x

The x position of the touch on the canvas. This is synonymous with _touch.x2

_touch.y

The y position of the touch on the canvas. This is synonymous with _touch.y2

_touch.x1

The x position of the previous touch on the canvas.

_touch.y1

The y position of the previous touch on the canvas.

_touch.x2

The x position of the touch on the canvas. This is synonymous with _touch.x

_touch.y2

The y position of the touch on the canvas. This is synonymous with _touch.y

_touch.distance

The distance between the last two touch positions if a touch was dragged.

_touch.distancex

The horizontal distance between the last two touch positions if a touch was dragged.

_touch.distancey

The vertical distance between the last two touch positions if a touch was dragged.

_touch.change

(Experimental) For pivot and pinch events, _touch.change returns angle and pixel change respectively.

_touch.percent

(Experimental) For pinch events, returns percent change relative to start position of pinch fingers.

_touch.angle

(Experimental) For pivot events, returns the angle between 2 touch positions.

_touch.size

(Experimental) For pinch events, returns the new size of the current sprite being updated.

Translation

A Translation is an alteration that will move a sprite from its current location to a new location in a sliding motion.

```
create routine as Start
  launch checker
end
create sprite from checker-1.png as checker
  having alt=trans1
end
create translation as trans1
  where x=400 y=200 speed=200 completion=Done
end
```

Parameters

x={number}

horizontal destination of the translation [optional]

y={number}

vertical destination of the translation [optional]

easein={number}

distance to use to accelerate up to speed

easeout={number}

distance to use to slow down to 0

speed={number}

speed of movement in pixels per second

completion={routine}

specify a routine to run when the sprite reaches its destination

Vars

Variables are not an alteration but a way of tracking values and using them later.

First you define your variable, then you add or subtract from it in a routine. You may also set the variable value explicitly.

Variables can be set to run a routine when a particular value is set or reached.

You can also create a **var** using shorthand. Shorthand allows you to simply create a variable and assign it value like: `var MyVar="pineapple"` The above is equivalent to: `create var as MyVar where value="pineapple" end` The shorthand does not allow you to add conditions or completion events but the variable created with shorthand can be changed and manipulated in all the same ways as a full declaration.

Note that this use of 'var' is done outside of routines. Using 'var' inside a routine is a local variable and behaves differently. Read about them in routines documentation.

```
create var as countHits
  where value=3 condition=0 completion=countDone
end
create routine as spriteHit
  update var countHits subtract=1
end
```

Parameters

value={number}

the starting value of the variable.

The condition of variables are checked just once per frame, after all sprites have been processed. The order in which different variable conditions are met is indeterminate. Therefore you cannot know the order that routines will be called as different conditions are met in that frame.

There are 2 frame events which can help in processing your variable conditions. Conditions are checked after the "penultimate" frame event, and before "frameexit".

condition={number}

if the value held by the variable reaches this "condition" value, then the completion event will fire.

completion={routine}

specify a routine to run when the value of the variable reaches the condition.

Vector

The vector alteration will move your sprite in the direction of its angle value.

```
create routine as Start
  launch mover
end
create sprite from uparrow.png as mover
  where center=103,27 x=200 y=50 angle=225
  having alt=vmove
end
create vector as vmove
  where speed=100 distance=50 completion=Done
end
```

Parameters

distance={number}

how far to move the sprite.

speed={number}

speed of movement in pixels per seconds.

completion={routine name}

a routine to run when the movement is complete.

Wait

The wait alteration isn't an alteration so much as a timing device. When applied to a sprite it will trigger a completion event after the given number of milliseconds. One convenient method of timing a whole animation sequence is to have a single sprite contain several wait alterations. This is actually easier to read and work with than having one event trigger the next.

```
create routine as Start
  launch bkgrd
  launch ship
end
create sprite from background.jpg as bkgrd
  having alt=w1 alt=w2 alt=w3
end
create wait as w1 where delay=1000 completion=w1Done end
create wait as w2 where delay=3000 completion=w2Done end
create wait as w3 where delay=5000 completion=w3Done end

create sprite from attacker5.png as ship
  where center=40,20 x=50 y=50 angle=300
end
create vector as v1 where speed=100 end
create routine as w1Done
  insert into ship where alt=v1
end
create rotation as r1 where speed=400 destination=70 end
create routine as w2Done
  insert into ship where alt=r1
end
create routine as w3Done
  remove from ship where alt=v1
end
```

Parameters

delay={number}

number of milliseconds to pass before triggering the completion event. The timer starts when the alteration is added to the sprite or the sprite is launched.

repeat={number|forever}

specify the number of iterations that the wait should occur, or mark it to repeat it forever. The **completion** routine will be called after each iteration and the **_repeat** value in the routine will be set to which iteration this is, starting at 1.

completion={routine}

specify a routine to run when the timer delay is reached.

Pausing a wait

A wait can be paused and unpaused in a routine:

```
pause wait WaitName
unpause wait WaitName
```

To use pause/unpause the 'wait' must have a name. Therefore it cannot be Panonymous, and it cannot be an unnamed clone.

Troubleshooting

Animations don’t run but show a message instead

If you haven’t registered your plugin purchase through the Settings page, you will see an error message when you try to run your animation. To resolve, go to the “Settings” menu and select “SCL Settings”. From there, enter your purchase code, check your license, and then save your changes.

Purchase code doesn’t work

The purchase codes confirms that you have purchased the plugin appropriately. Verify that your code is correct and try again. If it continues you fail, please contact us via Support and provide the relevant details.

I can’t get my animations to run properly or just get errors

For more complex animations, it takes time to learn SCL and get familiar with it. Specific error messages are given where possible. If you are an experienced programmer, you might find some clues in the JavaScript console. For more help, check the section in this documentation called “Tips, Tricks, and Gotchas”.

Animations don’t run, no images appear

This can occur if your WordPress site was initially installed on a non-SSL server, then the server is updated to SSL. In this case, the WordPress setting for ‘siteurl’ may still reference the non-SSL site. To change this setting, in administration, go into Settings>General, and change the web addresses given for ‘WordPress Address (URL)’ and ‘Site Address (URL)’ from http:// to https:// and save your changes.