Report Number: WUCS-91-10

1990-11-01

SwarmView Animation Vocabulary and Interpretation

Kenneth C. Cox

SwarmView Animation Vocabulary and Interpretation

Kenneth C. Cox

WUCS-91-10

November 1990

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

1 Introduction

This document specifies the syntax and interpretation of the language which is used to transmit descriptions of animations from SwarmExec to SwarmView. SwarmExec is a Prolog-based execution engine running on a Macintosh® IIfx. SwarmView is a C-based graphical engine running on a Silicon Graphics Personal Iris®. The major design elements of SwarmExec and SwarmView are discussed in the referenced papers.

2 Basic Concepts

The language described in this paper specifies collections of graphical objects that change with time. The language design is centered around *primitive graphical objects* or *PGOs*. PGOs are those simple graphical elements (such as lines, circles, rectangles, and spheres) which are provided by the SwarmView graphical engine. Each PGO has a *type* and a number of typed *attributes*. The type is the graphical form of the PGO, for example *line* or *sphere*. The attributes are those parameters of the PGO which are required to generate the object; in the case of a sphere, this would include the center, radius, and color. Attribute types are numbers, lists of numbers, and lists of lists of numbers – for example, coordinates are represented by lists of three numbers, and sets of coordinates are represented by lists of lists of three numbers. (Lists are enclosed in square brackets with elements separated by commas, a notation selected for its resemblance to Prolog.) Appendix 2 contains a list of the primitive graphical objects currently provided, along with the attributes of each object and the type of each attribute.

The basic unit of animation is a *primitive graphical event* or *PGE*. A PGE specifies an animation involving a single PGO. A PGE is represented in the language by a tuple

$$type\ (\ attribute_1 = expr_1,\ attribute_2 = expr_2,\ \ldots,\ attribute_n = expr_n\ )$$

where the *type* is the PGO type, each $attribute_i$ is one of the PGO's attributes, and each $expr_i$ is an appropriately-typed expression, either a constant or a time-dependent function. The tuples are variadic; that is, the number of tuple components (attribute/expression pairs) which appear is permitted to vary. Attributes for which an expression is unspecified are assigned a default value as indicated in Appendix 2. Animation is achieved by using time-dependent *functions* provided by the SwarmExec graphical engine. For example, to make a sphere move from point A to point B, we assign to the sphere's *center* attribute a function which changes smoothly from A to B.

The language is used to specify a series of *graphical transitions*, each of which represents a change from one image to another, the two images corresponding to two consecutive states of the underlying computation. Between each such pair of *endpoint images* are additional images or *frames*, generated by the animation, which produce the desired smooth transition from the initial image to the final image. Each graphical transition can be perceived in two ways:

- as a collection of PGEs which occur simultaneously;
- as a collection of frames which occur consecutively.

The first view is the one expressed by the language; the second is that which SwarmView generates.

Generation of frames involves the concept of time. Time within a transition is measured in terms of frame counts, also called *ticks*. The first frame of a transition occurs at tick 0, the second at tick 1, and so forth; the final frame occurs at a time *t_max* which is determined from examination of the tuples making up the graphical transition. The times used in the animation functions are specified as numbers in terms of ticks. One use of time is in the *lifetime* attribute; this attribute, which is possessed by all objects, is of type "list of two numbers" and represents the (closed) range of frames between which the object will be produced. An object with *lifetime* = [3,5] will be produced only in frames number 3, 4, and 5; one with *lifetime* = [0, *t_max*] will be produced in all frames.

3 Language Interpretation

A "program" in the language consists of an arbitrarily-long sequence of graphical transitions. Each transition consists of a sequence of PGE tuples separated by semicolons; the sequence is terminated by the special token `end` followed by a semicolon. A complete BNF grammar for the language appears in Appendix 1.

Interpretation begins with the input and storage of the PGE tuples making up a transition. The value of *t_max* for the transition is then determined by examining all the tuples and selecting the maximum of all times present (or 1, if no time is present). Times can appear in two contexts: as particular arguments of functions (for example, the first and third arguments of a *ramp* function), and as values assigned to the *lifetime* attribute of objects. One the value of *t_max* is determined, the interpreter generates the sequence of frames. For each tick, the values of all attributes of all objects are determined and the resulting collection of objects is rendered (although a particular object is rendered only if the current tick falls within the object's *lifetime*). The attribute may have been assigned a constant (a number, list of numbers, or list of list of numbers); in this case, the value calculated is simply the constant. In the case of a function, the value is obtained by evaluating the function at the appropriate tick and using the resulting value.

3.1 Function Specification and Evaluation

A function is either a simple function or a composition of simple functions. The notation for a simple function is standard:

$$function\_name\ (\ argument_1, argument_2, \ldots, argument_n\ )$$

A composite function is represented by a list of simple functions. A list of the available simple functions, with explanations of their behavior, is in Appendix 3.

The arguments of a simple function are of two types, times and constants. A time is an integer or the special token `t_max` which represents the value *t_max*. The constants are numbers, lists of numbers, or lists of lists of numbers; all constants provided to a particular function must be of the same type, and the value produced by the function will be of the same type.

We first consider the case of a simple function calculating a value of type number. Each function has two associated times (two of the function arguments) called the *start time* and the *end time* of the function. These divide the time from 0 to *t_max* into three periods, those *before*, *during*, and *after* the function's time range. The *before* period consists of those ticks t such that $0 \leq t <$ *start time*; the *during* period covers *start time* $\leq t <$ *end time*; and the *after* period is *end time* $\leq t <$ *t_max*. Any of these periods can be of length 0. For ticks within the *during* period, the value produced by the function is determined through application of the function and generally varies in some way. For ticks within the *before* and *after* periods, the value produced is a constant also determined by the function – most typically, the value *before* is the value of the function at the *start time* and the value *after* is the value of the function at the *end time*.

The *start time*, *end time*, and behavior of each function are explained in Appendix 2. As an example, consider the function *ramp(4,3.0,8,5.5)*. The *start time* of this function is 4, the *end time* is 8; assume *t_max* is 20. Then the function behavior is as shown in Figure 1: A ramp extending from time 4 to time 8 with value varying from 3.0 to 5.5, preceded by a constant value 3.0 and followed by a constant value 5.5.

When a simple function applied to more complex arguments (such as a list) the result is of the same type as the arguments and is obtained by applying the simple function to each of the list components individually. If arguments are lists of lists, the function is applied recursively to each list component. For example, the value resulting from the function *ramp(0,[0,0,0],2,[2,4,8])* at time 1 is *[1,2,4]*. This result can be considered as the value obtained by forming the list *[ramp(0,0,2,2),ramp(0,0,2,4),ramp(0,0,2,8)]*. (The language currently does not support the list of functions construct ; it is under consideration as a convenient expansion.)

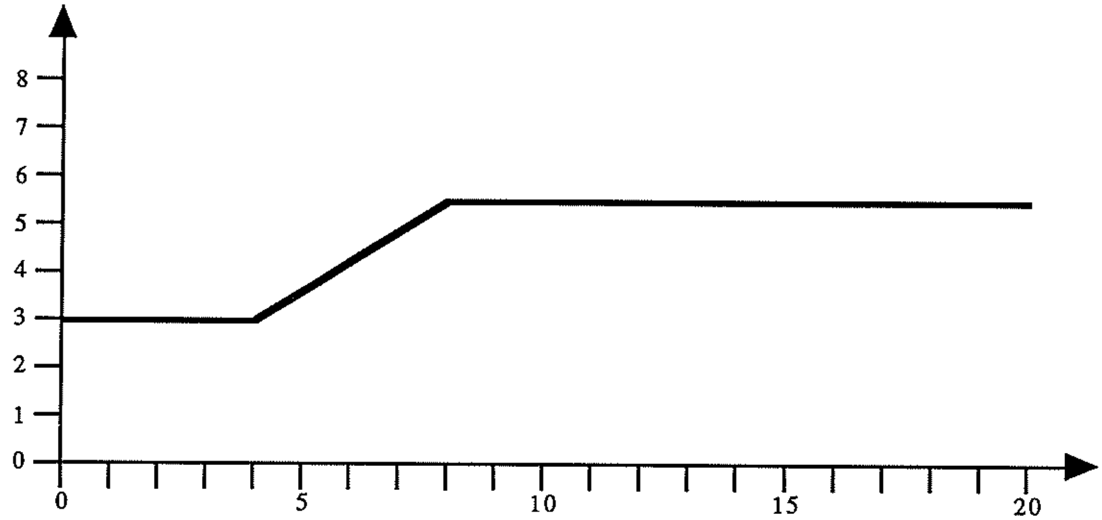


Figure 1. Behavior of *ramp(4, 3.0, 8, 5.5).*

In the case of a composite function the results are somewhat more complex. Assume for now that two functions are involved. The most important rule is the following: *the* during *periods of the two functions must not overlap.* That is, the intersection of the two *during* periods must be empty; no tick can be common to both. If the *during* periods do overlap, the behavior of the function is undefined (in the sense that any value produced is considered legitimate).

With this restriction, it must be the case that one function's *during* period occurs before the other's. This divides time into five periods:

- The *before* period of the first function. The value of the composite function in this period is the value produced by the first function in its *before* period.

- The *during* period of the first function. The value of the composite function is the value produced by the first function in its *during* period.

- The intersection of the *after* period of the first function with the *before* period of the second function. The value of the composite function is a smooth interpolation between the value produced by the first function for its *after* period and the value produced by the second function for its *before* period. That is, if *end time* of the first function is tick t_1 and that function produces value v_1 for its *after period,* and *start time* of the second function is tick $t_2$and that function produces value v_2 for its *before* period, the value of the composite function between times t_1 andt_2 is that produced by a *ramp* (t_1, v_1, t_2, t_2).

- The *during* period of the second function. The value of the composite function is the value produced by the second function in its *during* period.

- The *after* period of the second function. The value of the composite function in this period is the value produced by the second function in its *after* period.

Figure 2 illustrates the results of composing several ramp functions. The dashed lines indicate the periods of time as discussed above. The second graph in the Figure illustrates the behavior when the third time period is of length 0; in this case the ramp degenerates to a step where the value at the step is the value produced by the *second* function.

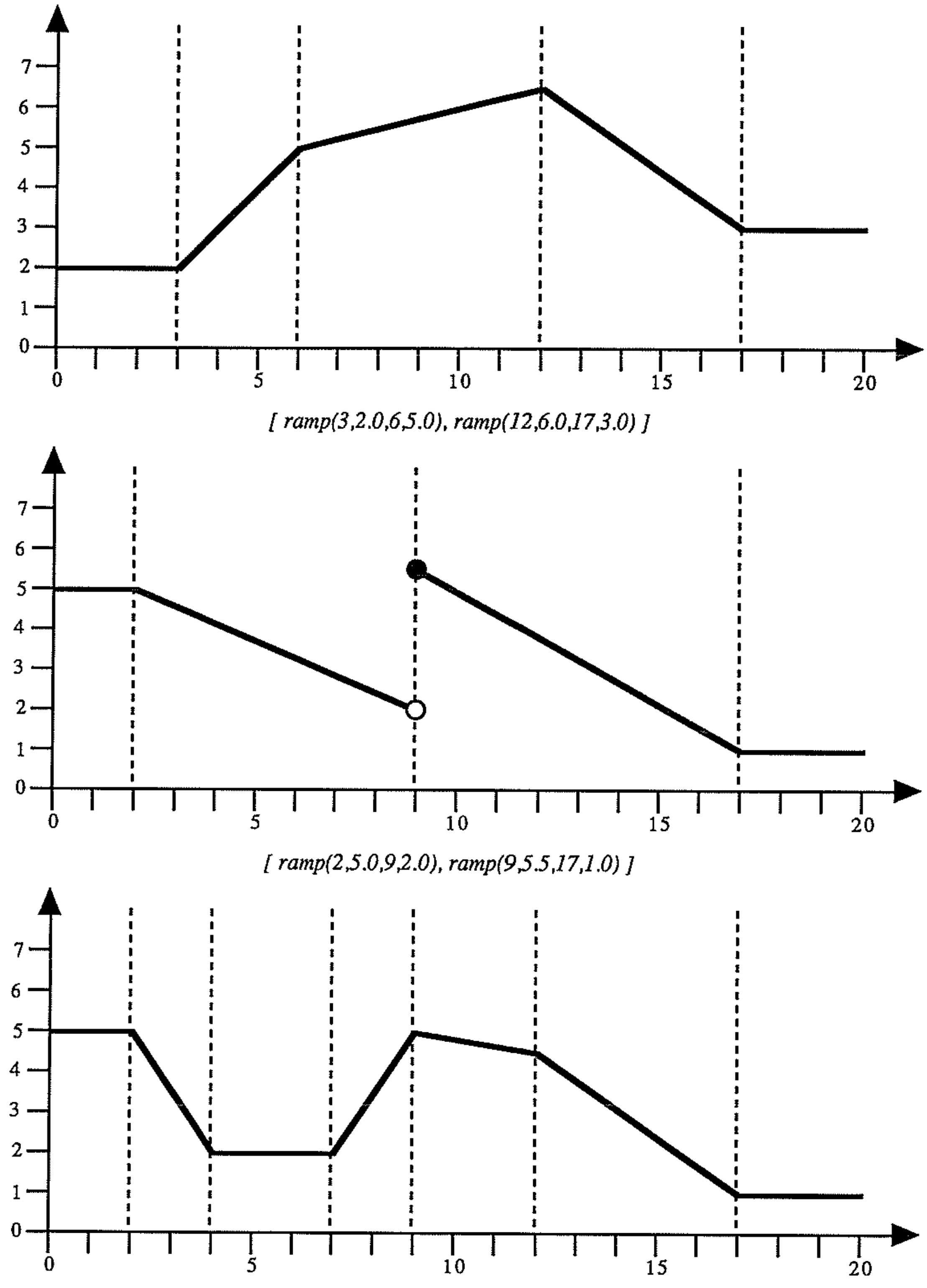


Figure 3. Compositions of various functions.

The third graph in Figure 2 illustrates the composition of three simple functions. The method is a logical extension of the composition of two functions. We first compose any two consecutive functions, i.e., any two functions whose *during* periods are such that no other function has a *during* period intervening. We consider the result as a single function whose *during* period includes the *during* periods of both subfunctions as well as the period between the two. We then compose this function with the third. Composition of any number of functions can be performed in the same manner, and a recursive specification for the result of composing a set of n functions can be easily expressed:

> If $n = 1$, the result of the composition is the single function in the set.
> If $n > 1$, select any two consecutive functions and remove them from the set. Compose the two functions using two-function composition, and add the resulting function to the set. Apply this algorithm to the result.

It should be noted that the above, although a concise means of expressing the result of a composition, is not the algorithm used by the SwarmExec interpreter for function composition – the interpreter does not need to have an expression for the composition, merely a means of calculating the value of the composite function at any particular tick.

4 Interpretation of Special Tuples

The SwarmView implementation defines a number of special tuple types. These tuples do not correspond to PGOs; instead, the tuples are used to transmit certain types of control information. Two special tuple types are currently defined: *view* and *define.*

The *view* tuple is used to specify the desired viewpoint for the graphical transition. The SwarmView viewing model is polar, where the user is considered to be looking at a particular point in the graphical space (usually the origin) from a viewpoint defined relative to the observed point by a distance, azimuth angle, and incidence angle. The parameters of the *view* tuple specify the observed point, distance, and angles. If a *view* tuple is encountered in a graphical transition, the tuple is stored; during the transition the *view* tuple is used to determine the viewpoint information. Normal viewer control of the viewpoint is overridden; the user can pause the animation and change the viewpoint, but when progress is resumed the viewpoint will return to that specified by the *view* tuple. Note that the attributes of the *view* tuple can vary with time, allowing the visualization to (for example) "zoom in" on areas of particular interest.

The *define* tuple is used to manipulate SwarmView's symbol table, either to create new object types based on existing object types or to change the default attributes of an existing type. The notation used is:

> *define (typename = primitive graphical event tuple)*

The interpretation of the *define* tuple is to create the new object type *typename* (or modify the existing *typename*), where *typename* will be a name for objects of the type given by the primitive graphical event tuple with default attributes as given by the PGE. The object type of the PGE must be one of the "primitive" types as listed in Appendix 2. For example, if the tuple

> *define (smallsphere = sphere (radius = 0.2))*

is processed, the result will be a symbol table entry for objects of type *smallsphere* which are identical to a sphere except the default radius is 0.2. The new type *smallsphere* may then be used as a typename in PGE tuples. If the *typename* and the type of the PGE are the same, the *define* tuple re-defines the defaults for the object type. For example, the following tuple causes all spheres to have radius 0.2 by default:

> *define (sphere = sphere (radius = 0.2))*

The *define* tuple is the only case in which the order in which the tuple set is read by SwarmView is significant. The SwarmExec execution engine is set up so all *define* tuples are transmitted before any tuples of other types, but not in any particular order. This ensures that all definitions appear before they are used.

5 Acknowledgments

The author would like to thank Dr. Jerome R. Cox of the Department of Computer Science at Washington University for his support. This research was supported in part by the National Fellowship Program in Parallel Processing, supported by DARPA/NASA and administered by the University of Maryland Institute for Advanced Computer Studies (UMIACS).

Appendix 1. Grammar

The following is a BNF grammar of the language. Terminals are in **bold**. The symbol λ represents the null string. The "terminal" **identifier** represents any legal identifier (anything described by the LEX-style regular expression [a-z][a-zA-Z_0-9]*); **number** represent any legal numeral, either integer or real.

```
language                  ::=  transition_list

transition_list           ::=  transition transition_list
                           |   λ

transition                ::=  pge ; transition
                           |   end;

pge                       ::=  type_name ( attribute_list )
                           |   type_name ()

type_name                 ::=  identifier

attribute_list            ::=  attribute , attribute_list
                           |   attribute

attribute                 ::=  attribute_name = expression

attribute_name            ::=  identifier

expression                ::=  function
                           |   constant

function                  ::=  primitive_function
                           |   [ primitive_function_list ]

primitive_function_list   ::=  primitive_function , primitive_function_list
                           |   primitive_function

primitive_function        ::=  identifier ( constant_list )

constant_list             ::=  constant , constant_list
                           |   constant

constant                  ::=  number
                           |   t_max
                           |   [ constant_list ]
```

Appendix 2. Graphical Objects

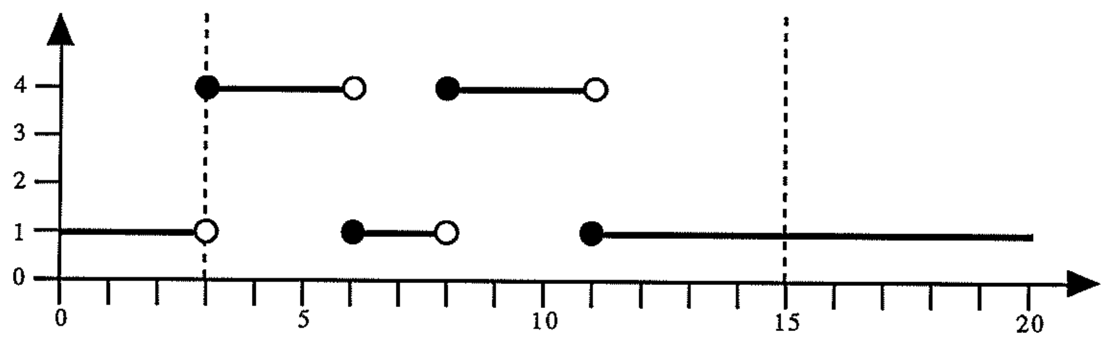
The following graphical objects are provided by the interpreter. All objects have a *lifetime* attribute, which is of type list of two numbers. The type *coordinate* is a shorthand for "list of three numbers" and specifies the X/Y/Z coordinates of the point. The type *color* is "list of three numbers" and specifies the red/green/blue color values, each in the range 0 to 255. The default color is *white*, or [255, 255, 255].

Object Type	Attribute	Type	Default	Notes
point	position	coordinate	[0, 0, 0]	location of point
	color	color	white	color of point
line	from	coordinate	[0, 0, 0]	one endpoint
	to	coordinate	[0, 0, 0]	the other endpoint
	color	color	white	
	width	number	1	width of line (screen pixels)
rectangle	corner	coordinate	[0, 0, 0]	lower left corner
	xsize	number	1	dimensions
	ysize	number	1	
	color	color	white	
	fill	number	0	if non-zero, rectangle is filled
	xrot	number	0	rotation about X-axis, degrees
	yrot	number	0	rotation about Y-axis
	zrot	number	0	rotation about Z-axis
crect	center	coordinate	[0, 0, 0]	centroid
	xsize	number	1	dimensions
	ysize	number	1	
	color	color	white	
	fill	number	0	if non-zero, rectangle is filled
	xrot	number	0	rotation about X-axis, degrees
	yrot	number	0	rotation about Y-axis
	zrot	number	0	rotation about Z-axis
polygon	vertices	list of coordinates	[[0, 0, 0]]	in the order to be connected
	color	color	white	
	fill	number	0	
circle	center	coordinate	[0, 0, 0]	
	radius	number	1	
	color	color	white	
	fill	number	0	
	xrot	number	0	as in **crect**
	yrot	number	0	
	zrot	number	0	
sphere	center	coordinate	[0, 0, 0]	
	radius	number	1	
	color	color	white	

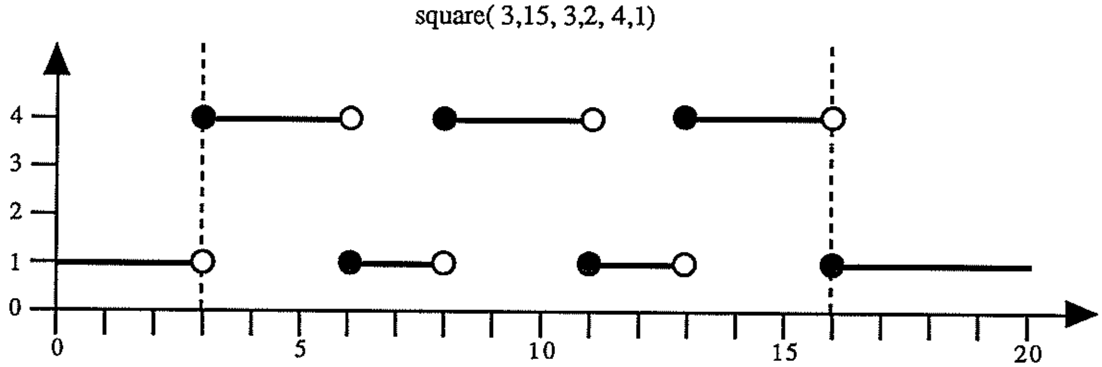
Appendix 3. Functions

Function	Start time	End time	Value before	Value during	Value after
step (t, v_0, v_1)	t	t	v_0	N/A	v_1
ramp(t_0, v_0, t_1, v_1)	t_0	t_1	v_0	linear interpolation from v_0 at t_0 to v_1 at t_1	v_1
constant(t_0, v, t_1)	t_0	t_1	v	v	v
square(t_0, t_1, p_{on}, p_{off}, v_{on}, v_{off})	t_0	t_1	v_{off}	square wave: v_{on} for p_{on} ticks, v_{off} for p_{off} ticks	v_{off}

The *square* function takes value v_{on} at time t_0, then alternates between v_{on} and v_{off} for the rest of the *during* period. Ass v_{on} periods last a complete time p_{on}; if the interval remaining in the *during* period is insufficient for a complete v_{on}, the value will be held at v_{off} until the expiration of the *during* period. The following diagram gives some examples of this for clarification. Both graphs show a square wave with $p_{on} = 3$ and $p_{off} = 2$. In the upper graph the last v_{on} period ends at tick 11; if another period were started, it would begin at time 13 and end at time 16, after the expiration of the during. In the lower graph there is sufficient time for an addition v_{on} period to be included.



square(3,15, 3,2, 4,1)



square(3,16, 3,2, 4,1)