



Tr3 Overview

Version 10.02.d

August 19, 2010

Tr3 (pronounced “tree”) is a realtime media performance ontology that connects Human gestures to device interfaces. Tr3 was designed for the novice, programmer, designer, and tester. A novice will use Tr3 to orchestrate interactive media on tablets. A programmer will use Tr3 to wrap around their preferred language and APIs. A designer will use Tr3 to iterate through many user interfaces. And finally, a tester will verify the results.

Tr3 was started in 1998, as a patch bay for a visual music synthesizer that used a variety of input devices to create visual music. The favorite device is a tablet, which was put in the hands of all age groups with mesmerizing results. Tr3 has been re-written to focus on multi-touch surfaces, such as the Apple iPad.



Tr3 consists of a namespace generator, structure decorator, auto-scaled ranges, event observers, a code wrapper, ontology viewer, and plug-ins for recording and potentially selling performances.

Names decorated with data structures

The **namespace** generator borrows from Lisp and Xpath. For example, a two handed interface could be generated like so:

```
hand (left right).finger[5].(x y)
```

The same structure would require 46 lines of XML.

The expanded Tr3 data structure looks like a cross between C and Python using {} blocks and implicit types.

```
hand { left { finger { 0 {x y} 1 {x y} 2 {x y} 3 {x y} 4 {x y} }}  
      right { finger { 0 {x y} 1 {x y} 2 {x y} 3 {x y} 4 {x y} }}}
```

Each attribute can have a number, string, or a ranged type value

following this format: <name>:<value>;

for example: `hand.left.finger.0.x: 100;`

Tr3 has both inclusive (a..b) and exclusive (a:b) ranges. For example, the iPad has a range of 1024 by 768 pixels. Meanwhile, a ticTacToe board may have a range of 3 by 3 squares. Here are two different ways of representing the respective ranges:

```
touches { x: 0:1024; y: 0:768; } // 0:1024 is a python like slice from 0 thru 1023  
board   { x: 1..3; y: 1..3; }    // 1..3 is a minmax from 1 thru and including 3
```

A generated namespace can be **decorated** with a data structure:

```
hand (left right).finger[5] { x::1024; y::768; }
```

The `^` operator searches an existing namespace for descendants, somewhat like an Xpath `/**` search. This allows a whole sub-tree to be decorated with the same data structure, like so:

```
hand^{x::1024; y::768;} // landscape orientation
```

And if you want to change orientation, redecorate:

```
hand^{x::768; y::1024;} // portrait orientation
```

Observers with Auto-Scaled Ranges

The `@` symbol is an **observer** that connects to events, like a callback. For example:

```
board.x@touches.x;
```

means that a change in `touches.x` will trigger `board.x`. Recall that `touches` and `board` have two different ranges:

```
touches { x: :1024; y: :768; }
board   { x: 1..3; y: 1..3; }
```

Because both `touches` and `board` have a ranged type, a value in `touches.x` will rescale to `board.x`'s range, like so:

```
touches.x: 0;      rescales to board.x: 1;
touches.x: 512;   rescales to board.x: 2;
touches.x: 1023; rescales to board.x: 3;
```

The `@+` and `@-` symbols do not rescale ranges; instead, they increment or decrement the receiver. This is useful in creating a switch, like so:

```
switch: 0..1; {
    @+ button.one.on;
    @- button.one.off;
}
```

Can also be used with a modulo range to create a toggle or multi-state switch:

```
toggle: %2; @+ button.two; // toggles modulo on off
3way: %3; @+ button.three; // three state switch
```

An `@ { }` observer can trigger its own data structure. A common example is a button triggering a new connection:

```
whatever @ button.on { newReceiver @ slider.x; }
```

But, what happens to the previous connection? If there is one, it still exists. To get around that problem, there is a manual and an automatic solution.

The `!@` observer operator disconnects the previous connection. To manually switch `oldReceiver` to `newReceiver`, use the `disconnect` and then `reconnect`:

```
whatever @ button.on {  
  
    oldReceiver !@ slider.x;  
    newReceiver @ slider.x;  
}
```

The `@:` operator adds references to nodes to the receiver, but not the sender. This is useful for marshaling events later on. For example an application may want to record changes to state for a portion of inputs. But, when not recording, there is no need to be receiving events.

This is useful for recording a session. For example, a recorder object may accept input except `gps`, like so:

```
recorder.event @:input^; !@:input.gps^;
```

Since the source outputs are not connected, the input events are not automatically sent to `recorder.event`. Instead, the application programmatically connects the outputs (hidden from the Tr3 script) when the recorder is turned on.

Other uses for `@:` are for saving and recalling a placemark for a performance and marshaling events between devices, when connections may be turned on or off.

Breaking loops

Each observer breaks loops. So, what happens with the following patch?

```
board.x @ touches.x;  
touches.x @ board.x;
```

The touch events triggers `board`, but stops there. The `touches` observer notices that it had already handled the event so it stops. This allows a novice to connect controls in an observation loop without crashing.

Language Wrapper

Tr3 currently wraps C++ as a virtual controller, within the MVC (model view controller) pattern. Here's an example of implementing a simple callback objective C:

```
// FILE: test.tr3

main {yo: 1;}

// FILE: Something.mm **make sure to use mm for objc++ to recognize C++

#import "Tr3.h"

@interface Something : NSObject {
    Tr3* _yo;
}
@end

@implementation Something

- (id) init {
    ...
    // payload for callback
    CallIdSel cis = new CallIdSel;
    cis->_id = self;
    cis->_sel = @selector(objcCall);
    // bind to test.tr3 value, with optional callback
    Tr3 *main = Tr3::bind("main");
    _yo = Tr3::bind(main, "yo", (Tr3CallTo)(&callback), (void*)cis);
    ...
}

void callback(Tr3*from, CallClassData*data) {
    id target = (id)(data->_instance);
    SEL sel = (SEL)(data->_data);
    [target performSelector:sel];
}

-(void) objcCall {
    int status = *_yo; // converts to integer
    NSLog(@"status: %i", status);
}
```

Ontology Viewer

A previous version of Tr3 produced XML that was fed to a utility to visualize the ontology. The ontology consists of a directed graph three types of arcs:

red: namespace hierarchy
green: observer inputs
blue: caller outputs

Here is an early prototype of the [ontology viewer](#) written in Java.

Wish list:

C++. Ideally, the viewer would be ported to C++, which would enable to work on a wider variety of tablets while consuming the least amount of power.

Clustering. Clustering of nodes through Neuman's algorithm helps discover tightly bound relationships, but is often too fine grained, which obfuscates a higher level process. Weighted clustering around the namespace would be ideal. For example:

```
draw.dot @^ touch
```

might cluster around each x and y, with 10 arcs:

```
draw.dot.x @ touch.0.x  
draw.dot.y @ touch.0.y  
...  
draw.dot.x @ touch.4.x  
draw.dot.y @ touch.4.y
```

A preferred clustering would be around dot and touch, with a single arc between the two.

```
draw.dot @^ touch
```

Interactivity. Currently, the viewer is read only; the only way to create or delete patches is through a script. Ideally, the user could draw new arcs with a finger - bypassing the script completely.

Language Definition

tr3 (whos what? when* wheres? why?)

whos (who+ why*)

```
who ( dot | circ | path | array | list)
  path (r'^([A-Za-z0-9_]|.[^(){}])+')
  array (r'^[ ]*\[\]' num r'^\[\]' )
  list (r'^\[\]' (who why*)+ r'^\[\]' )
  dot (r'^[.]')
  circ (r'^[\^]' )
```

what ((colon whatis)? ';' why*)

```
colon (r'^[ \t\r\n]*[:][ \t]*')
whatis (scalar | num | quote )
```

scalar ((minmax | cycle) span? dfault? timing?)

```
minmax (minNum? '..' maxNum?)
  minNum (num)
  maxNum (num)
cycle ('%' num)
span (':' num)
dfault ('=' num)
timing ('~' num)
```

```
num (r'^([+-]*[0-9]+[.][0-9]+|[+-]*[0-9]*[.][0-9]+|[+-]*[0-9]+)')
quote (r'^\"([^\"]*)\"')
```

wheres ('{' why* where+ '}')

where (tr3)

when (observe obWhos? (hows | ';'))

```
observe (nada? '@' (byname | blind | group | solo | mute | incr | decr)*)
  nada ('!')
  byname ('^')
  blind (':')
  group ('&')
  solo ('$')
  mute ('~')
  incr ('+')
  decr ('-')
obWhos (who+ why*)
```

hows ('{' how* '}')

```
how (patch | call | why )
  patch (path observe whom+)
  whom (who)
  call (path what)
```

why (r'^[/][][]*(.?)[\r\n]+|^[\r\n\t]+')

The language has 44 symbols. There are no reserved words. A python utility called Tr3.py reads in a definition file tr3.def and produces a header file tr3.def.h. A similar python header file tr3.def.py is produced and verified on a rudimentary level.

```
// generated by Tr3Print.py
Par_(root      , And( Any(why) , Mny(tr3) ) )
Par_(tr3      , And( One(whos) , Opt(what) , Any(when) , Opt(wheres) , Opt(why) ) )
Par_(whos     , And( Mny(who) , Any(why) ) )
Par_(who      , Or( One(dot) , One(circ) , One(path) , One(array) , One(list) ) )
Par_(path     , Regx("^[([A-Za-z0-9_]|.[^()])+") )
Par_(array    , And( Regx("[ ]*[\[\]]") , One(num) , Regx("[\[\]]") ) )
Par_(list     , And( Regx("[\[\]]") , Mny( And( One(who) , Any(why) ) ) , Regx("[\[\]]") ) )
Par_(dot      , Regx("[.]") )
Par_(circ     , Regx("[\[\]]") )
Par_(what     , And( Opt( And( One(colon) , One(whatis) ) ) , Quo(";") , Any(why) ) )
Par_(colon    , Regx("[ \t\r\n]*[:][ \t]*") )
Par_(whatis   , Or( One(scalar) , One(num) , One(quote) ) )
Par_(scalar   , And( Or( One(range) , One(cycle) ) , Opt(span) , Opt(dfault) , Opt(timing) ) )
Par_(range    , Or( One(minmax) , One(slice) ) )
Par_(minmax   , And( Opt(minNum) , Quo("..") , Opt(maxNum) ) )
Par_(slice    , And( Opt(minNum) , Quo(":") , Opt(maxNum) ) )
Par_(minNum   , One(num) )
Par_(maxNum   , One(num) )
Par_(cycle    , And( Quo("%") , One(num) ) )
Par_(span     , And( Quo(",") , One(num) ) )
Par_(dfault   , And( Quo("=") , One(num) ) )
Par_(timing    , And( Quo("~") , One(num) ) )
Par_(num      , Regx("[+-]*[0-9]+[.][0-9]+|[-+]*[0-9]*[.][0-9]+|[-+]*[0-9]+") )
Par_(quote    , Regx("[^\"([^\"]*)\"") )
Par_(wheres   , And( Quo("{") , Any(why) , Mny(where) , Quo("}") ) )
Par_(where    , One(tr3) )
Par_(when     , And( One(observe) , Opt(obWhos) , Or( One(hows) , Quo(";") ) ) )
Par_(observe   , And( Opt(nada) , Quo("@") , Any( Or( One(byname) , One(blind) ,
    One(group) , One(solo) , One(mute) , One(incr) , One(decr) ) ) ) )
Par_(nada     , Quo("!") )
Par_(byname   , Quo("^") )
Par_(blind    , Quo(":") )
Par_(group    , Quo("&") )
Par_(solo     , Quo("$") )
Par_(mute     , Quo("~") )
Par_(incr     , Quo("+") )
Par_(decr     , Quo("-") )
Par_(obWhos   , And( Mny(who) , Any(why) ) )
Par_(hows     , And( Quo("{") , Any(how) , Quo("}") ) )
Par_(how      , Or( One(patch) , One(call) , One(why) ) )
Par_(patch    , And( One(path) , One(observe) , Mny(whom) , Quo(";") ) )
Par_(whom     , One(who) )
Par_(call     , And( One(path) , One(what) ) )
Par_(why      , Regx("[/]/[/] [ ]*(.?[ \r\n]+|^ [ \r\n\t]+") )
```

Future development

Named Connections

Included in the definition and parser, but not implemented is `@^` symbol, which will connect a hierarchy of attributes by the name of their sub-attributes. So

```
board @^ touches;
```

is equivalent to:

```
board.x @ touches.x;  
board.y @ touches.y;
```

resulting in the following data structure:

```
board {  
  x: 1..3; @ touches.x;  
  y: 1..3; @ touches.y;  
}
```

Solo Observers

Included in the definition and parser, but not implemented is the `@$` symbol, which will exclusively connect to a single receiver; previous receivers are automatically disconnected. To automatically switch from old Receiver to newReceiver, no prior knowledge needed:

```
whatever @ button.on { newReceiver @$ slider; }
```

Symbol elements can be combined, so to connect solo attributes by name:

```
whatever @ button.on { newReceiver @$^ touches; }
```

Combined Observers

Included in the definition and parser and implemented, but not tested, is the `@&` symbol, which requires all senders to be active, before a receiver becomes active. Very useful for key combinations, like so:

```
menuAKey @& input.key(menu a);
```

Cascading Observers

Some background: an early version of Tr3 supported a visual synthesizer that had user inputs via: tablet, joystick, midi keyboard, computer keyboard, mouse, virtual puppet, and writing recognition. All inputs shared the same namespace with a high amount of overlap. For example, a midi keyboard and tablet could control exactly the same visual synth controls, connecting to the same namespace. After thousands of hours of connecting and performing with different devices, it was discovered that the tablet could provide the greatest coverage of a real-time name space in the least amount of time. For example, the following interface:



represents about 8200 nodes in a switched ontology of C++ variables in the underlying code. With the old version of Tr3, it took one day to map the interface over an existing C++ namespace. Because scalar ranges remap, it didn't matter what size the tablet was.

The problem was that the ontology was static, which was fine for a printed template on a Wacom tablet. However touchable surfaces, like the iPad, allow interface elements to be pinched, rescaled, and moved. A dynamic ontology would allow sub-elements flow within the parent - akin to how web-pages flow with CSS (cascading style sheets).

Cascading observers can be used with any hierarchy of real-time interaction. For example, let's define a full body motion capture suit, which we'll call an avatar:

```
avatar (torso (mouth nose neck (head (eyes (left right) ears (left right)))
           shoulders (left right).arms (elbow (wrist ( hand ( fingers[5]))))
           waist (legs (left right).knee (ankle (foot ( toes[5]))))));
```

And let's decorate each element with angular orientation o p q, radius r, and position x y z:

```
avatar^(o p q r x y z);
```

When a torso changes position or orientation, everything attached to that torso also change their position and orientation. Moving just the arm will affect the elbow, hand, and fingers.

So, let's decorate each element of the avatar with a routine to adjust itself, based on the parent's position, like so:

```
avatar^@(..){adjust(.. .);}
```

The @.. means observe parent. When the parent's state changes, the observer calls `adjust(.. .)`; that passes the parent node and self to a wrapper around a c++ routine to adjust self's `o p q x y z`. The data structure looks something like this:

```
avatar @(..){adjust(.. .); { o p q r x y z;
torso @(..){adjust(.. .); { o p q r x y z;
neck @(..){adjust(.. .); { o p q r x y z;
head @(..){adjust(.. .); { o p q r x y z;
eyes @(..){adjust(.. .); { o p q r x y z;
  left @(..){adjust(.. .); { o p q r x y z;
  right @(..){adjust(.. .); { o p q r x y z;}}
nose @(..){adjust(.. .); { o p q r x y z;
mouth @(..){adjust(.. .); { o p q r x y z;
ears @(..){adjust(.. .); { o p q r x y z;
  left @(..){adjust(.. .); { o p q r x y z;
  right @(..){adjust(.. .); { o p q r x y z;}}}}
shoulders @(..){adjust(.. .); { o p q r x y z;
left @(..){adjust(.. .); { o p q r x y z;
arms @(..){adjust(.. .); { o p q r x y z;
  elbow @(..){adjust(.. .); { o p q r x y z;
  wrist @(..){adjust(.. .); { o p q r x y z;
  hand @(..){adjust(.. .); { o p q r x y z;
  finger.0 @(..){adjust(.. .); { o p q r x y z;
  finger.1 @(..){adjust(.. .); { o p q r x y z;
  finger.2 @(..){adjust(.. .); { o p q r x y z;
  finger.3 @(..){adjust(.. .); { o p q r x y z;
  finger.4 @(..){adjust(.. .); { o p q r x y z;}}}}}}
right @(..){adjust(.. .); { o p q r x y z;
arms @(..){adjust(.. .); { o p q r x y z;
  elbow @(..){adjust(.. .); { o p q r x y z;
  wrist @(..){adjust(.. .); { o p q r x y z;
  hand @(..){adjust(.. .); { o p q r x y z;
  finger.0 @(..){adjust(.. .); { o p q r x y z;
  finger.1 @(..){adjust(.. .); { o p q r x y z;
  finger.2 @(..){adjust(.. .); { o p q r x y z;
  finger.3 @(..){adjust(.. .); { o p q r x y z;
  finger.4 @(..){adjust(.. .); { o p q r x y z;}}}}}}}}
waist @(..){adjust(.. .); { o p q r x y z;
legs @(..){adjust(.. .); { o p q r x y z;
left @(..){adjust(.. .); { o p q r x y z;
  knee @(..){adjust(.. .); { o p q r x y z;
  ankle @(..){adjust(.. .); { o p q r x y z;
  foot @(..){adjust(.. .); { o p q r x y z;
  toes.0 @(..){adjust(.. .); { o p q r x y z;
  toes.1 @(..){adjust(.. .); { o p q r x y z;
  toes.2 @(..){adjust(.. .); { o p q r x y z;
  toes.3 @(..){adjust(.. .); { o p q r x y z;
  toes.4 @(..){adjust(.. .); { o p q r x y z;}}}}}}
right @(..){adjust(.. .); { o p q r x y z;
  knee @(..){adjust(.. .); { o p q r x y z;
  ankle @(..){adjust(.. .); { o p q r x y z;
  foot @(..){adjust(.. .); { o p q r x y z;
  toes.0 @(..){adjust(.. .); { o p q r x y z;
  toes.1 @(..){adjust(.. .); { o p q r x y z;
  toes.2 @(..){adjust(.. .); { o p q r x y z;
  toes.3 @(..){adjust(.. .); { o p q r x y z;
  toes.4 @(..){adjust(.. .); { o p q r x y z;}}}}}}}}}}
```

As with the visual music synth, we could map the avatar to a tablet interface that abstracts the namespace. Recall the background example of many controllers modifying the same model. That means the avatar could be controlled by a camera based body sensor, iPad, joystick, or any other device with overlapping controls. Since Tr3 breaks loops, the controller within each device can update its state without deadlock. The result is a simple and extensible real-time controller that can span many devices and many performers.