A 'Hello World' for the LambdaTable
by Chris Falk (cfalk2@uic.edu)
This is a simple Electro application for the LambdaTable. By continuously reading in data from the tracker, the program draws markers for each active puck on the table, accurately representing their position and orientation. In the following sections, each bit of code is described, and at the end the program is presented in its entirety. Instructions are provided on how to turn on the LambdaTable and run your code. Refer to the last page for any referenced diagrams.

```plaintext
new_x, new_y, new_a = 0, 0, 0
myhost = "*"
myport = 7000
udp = {}
device = {}
```

These are global variables which will be used in later subroutines. The `new_*` variables refer to the coordinates of a puck, and are initialized here. `myhost`, `myport`, and `udp` are needed to set up a UDP data stream from the tracking cluster of the LambdaTable. The variable `device` is initialized to be an empty table. It will later hold an entry for each puck on the table.

```plaintext
function create_scene()
camera = E.create_camera(E.camera_type_orthogonal)
pivot = E.create_pivot()
E.parent_entity(pivot, camera)
dsp_x0, dsp_y0, dsp_w, dsp_h = E.get_display_union()
E.set_entity_position(pivot, dsp_x0 + dsp_w/2, dsp_y0 + dsp_h/2, 0)
brush_line = E.create_brush()
brush_fill = E.create_brush()
E.set_brush_color(brush_line, 0.2, 0.2, 0.5, 1.0)
E.set_brush_color(brush_fill, 0.8, 0.8, 0.8, 1.0)
end
```

This function sets up the scene graph with the camera as the root, and a pivot as its only child. (Fig 1 shows the complete hierarchy of objects. The device objects will be explained later on.) `dsp_x0`, `dsp_y0`, `dsp_w`, `dsp_h` are initialized to the position, width, and height of the pixel size of the LambdaTable display. In the coordinate system of Electro, the origin is in the center of the viewable area, with `dsp_x0` and `dsp_y0` being the bottom left corner. This can be seen in Fig 3. In the next line, the position of `pivot` is set to (0,0), in the coordinate system of the camera. Then the brushes for the puck markers are initialized.
function mark_sprite(id)
    local id_string  = E.create_string(string.format("%d", id))
    local pivot      = E.create_pivot()
    E.set_entity_scale   (id_string, 32, 32, 32)
    E.set_entity_position(id_string, 96, 96, 0)
    E.set_string_line    (id_string, brush_line)
    E.set_string_fill    (id_string, brush_fill)
    E.parent_entity(id_string, pivot)
    return pivot
end

Here, the labels for each puck are set up. The LambdaTable distinguishes each puck from one another, giving them ID's starting at 1 up to the number of devices it is configured for. Therefore, each marker will be labeled with the appropriate string object.

function make_sprite(filename, trans, unlit)
    local image = E.create_image(filename)
    local brush = E.create_brush()
    E.set_brush_flags(brush, E.brush_flag_transparent, trans)
    E.set_brush_flags(brush, E.brush_flag_unlit, unlit)
    E.set_brush_color(brush, 1.0, 1.0, 1.0, 1.0)
    E.set_brush_image(brush, image)
    local sprite = E.create_sprite(brush)
    E.set_entity_position(sprite, -128, -128, 0)
    return E.create_sprite(brush)
end

This function creates a sprite in the following sequence. First, an image is created from the input filename. Then a brush is created, using the image as its texture map. Finally, a sprite is created using the brush.

function show_device(dev, type)
    for i=1, 3 do
        E.set_entity_flags(dev[i], E.entity_flag_hidden, not (i == type))
    end
end

In show_device, the dev parameter is a table holding sprites corresponding to different marker states. One puck may have three different sprites associated with it – the default, one for a left button press, and one for a right button press. type is a number from the tracker that indicates which state the puck is in, and therefore which sprite should be displayed.

definition do_timer(dt)
    local socketEmpty = false
    local dgram      = false
    repeat
        poll = udp:receivefrom(60)
        if poll == nil then
            socketEmpty = true
        else
            dgram = poll
        end
    end
end
if dgram then
    if tonumber(string.sub(dgram, 1, 10)) == 2 then
        update = true
        msg_id = tonumber(string.sub(dgram, 11, 20))
        msg_xpos = tonumber(string.sub(dgram, 21, 30))
        msg_ypos = tonumber(string.sub(dgram, 31, 40))
        msg_angle = tonumber(string.sub(dgram, 41, 50))
        msg_type = tonumber(string.sub(dgram, 51, 60))

        if not device[msg_id] then
            local id = msg_id
            device[id] = {}
            device[id][1] = make_sprite("marker.png", true, true)
            device[id][2] = make_sprite("marker_left.png", true, true)
            device[id][3] = make_sprite("marker_right.png", true, true)
            device[id].id = mark_sprite(id)
            device[id].pv = E.create_pivot()
            E.parent_entity(device[id][1], device[id].pv)
            E.parent_entity(device[id][2], device[id].pv)
            E.parent_entity(device[id][3], device[id].pv)
            E.parent_entity(device[id].id, device[id].pv)
            E.parent_entity(device[id].pv, pivot)
        end
        show_device(device[msg_id], msg_type)
        new_x = msg_xpos * dsp_w + dsp_x0
        new_y = msg_ypos * dsp_h + dsp_y0
        new_a = msg_angle
        local x, y, a = new_x, new_y, new_a
        E.set_entity_rotation(device[msg_id].pv, 0, 0, a)
        E.set_entity_position(device[msg_id].pv, x, y, 0)
    end
end
until socketEmpty
return update
end

As stated in the Electro documentation, if idling is enabled, the `do_timer` callback is invoked regularly. In this function, the udp socket object is continuously polled, and each received datagram is parsed into `msg_id`, `msg_xpos`, `msg_ypos`, `msg_angle`, and `msg_type`. Then, if the device table does not have an existing entry for the particular puck corresponding to `msg_id`, an entry is created, resulting in a graph such as Fig 1. Otherwise, the `show_device` subroutine is called to update which markers should be displayed. The position and rotation of the markers are also updated. Note that the received position values will fall between 0 and 1, as shown in Fig 2. Angle values should be interpreted with 0 degrees being 'north', 90 degrees being 'west', and so on.

```lua
function do_start()
    create_scene()
    E.set_typeface("VeraBd.ttf", 0.0001, 0.03125)
    -- Open port to listen for track data
    E.print_console("Binding to host ", myhost, ", port ", myport, "\n")
    udp, err = socket.udp()
```
do_start is the 'main' function of the program. It starts by calling create_scene and setting up the scene graph. Then it sets up UDP communication with the LambdaTable tracker. The socket udp() function creates a UDP socket object. setsockname binds the UDP object to a local address. All of these functions are part of the LuaSocket library, documented at http://www.cs.princeton.edu/~diego/professional/luasocket/.

Finally, idling is enabled, which ensures that the do_timer function will be called.

Full source of ui-test.lua

```lua
new_x, new_y, new_a = 0, 0, 0
myhost = "*
myport = 7000
udp = {}
device = {}

function create_scene()
camera = E.create_camera(E.camera_type_orthogonal)
pivot = E.create_pivot()

E.parent_entity(pivot, camera)
dsp_x0, dsp_y0, dsp_w, dsp_h = E.get_display_union()
E.set_entity_position(pivot, dsp_x0 + dsp_w/2, dsp_y0 + dsp_h/2, 0)
brush_line = E.create_brush()
brush_fill = E.create_brush()
E.set_brush_color(brush_line, 0.2, 0.2, 0.5, 1.0)
E.set_brush_color(brush_fill, 0.8, 0.8, 0.8, 1.0)
end

function mark_sprite(id)
local id_string = E.create_string(string.format("%d", id))
local pivot = E.create_pivot()

E.set_entity_scale (id_string, 32, 32, 32)
E.set_entity_position(id_string, 96, 96, 0)
E.set_string_line (id_string, brush_line)
E.set_string_fill (id_string, brush_fill)
E.parent_entity(id_string, pivot)
return pivot
end

function make_sprite(filename, trans, unlit)
local image = E.create_image(filename)
local brush = E.create_brush()
E.set_brush_flags(brush, E.brush_flag_transparent, trans)
E.set_brush_flags(brush, E.brush_flag_unlit, unlit)
```
E.set_brush_color(brush, 1.0, 1.0, 1.0, 1.0)
E.set_brush_image(brush, image)

local sprite = E.create_sprite(brush)
E.set_entity_position(sprite, -128, -128, 0)

return E.create_sprite(brush)
end

function show_device(dev, type)
  for i=1, 3 do
    E.set_entity_flags(dev[i], E.entity_flag_hidden, not (i == type))
  end
end

function do_timer(dt)
  local socketEmpty = false
  local dgram
  local update      = false

  repeat
    poll = udp:receivefrom(60)
    if poll == nil then
      socketEmpty = true
    else
      dgram = poll
    end

    if dgram then
      if tonumber(string.sub(dgram, 1, 10)) == 2 then
        update = true
        msg_id    = tonumber(string.sub(dgram, 11, 20))
        msg_xpos  = tonumber(string.sub(dgram, 21, 30))
        msg_ypos  = tonumber(string.sub(dgram, 31, 40))
        msg_angle = tonumber(string.sub(dgram, 41, 50))
        msg_type  = tonumber(string.sub(dgram, 51, 60))

        if not device[msg_id] then
          local id   = msg_id
          device[id] = {}

          device[id][1] = make_sprite("marker.png", true, true)
          device[id][2] = make_sprite("marker_left.png", true, true)
          device[id][3] = make_sprite("marker_right.png", true, true)
          device[id].id = mark_sprite(id)
          device[id].pv = E.create_pivot()

          E.parent_entity(device[id][1], device[id].pv)
          E.parent_entity(device[id][2], device[id].pv)
          E.parent_entity(device[id][3], device[id].pv)
          E.parent_entity(device[id].id, device[id].pv)
        end

        show_device(device[msg_id], msg_type)

        new_x = msg_xpos * dsp_w + dsp_x0
        new_y = msg_ypos * dsp_h + dsp_y0
        new_a = msg_angle

        local x, y, a = new_x, new_y, new_a
        E.set_entity_rotation(device[msg_id].pv, 0, 0, a)
        E.set_entity_position(device[msg_id].pv, x, y, 0)
      end
    end
  end

  until socketEmpty

  return update
end
end

function do_start()
    create_scene()
    E.set_typeface("VeraBd.ttf", 0.0001, 0.03125)
    -- Open port to listen for track data
    E.print_console("Binding to host '" .. myhost .. ", port " .. myport .. "...
"
    udp, err = socket.udp()
    ret, err = udp:setsockname(myhost, myport)
    udp:settimeout(.001)
    myip, myport = udp:getsockname()
    E.print_console("Waiting packets on " .. myip .. ",:.. myport .. ",...
"
    E.enable_timer(true)
end

do_start()

Using the LambdaTable

To test any applications on the LambdaTable, you need to first have an account. Speak with your professor about this. Once you have your account, refer to http://www.ev1.uic.edu/cole/usingtable/ to help you set up your work environment and operate the table.

To run your application, it's convenient to write a short script. Here's an example...

```
#!/bin/sh
cd /home/evl/cfalk2/electro
/home/evl/public/electro/electro-table ui-test.lua -m
```
Diagrams

Scene Graph Hierarchy (Fig 1)

Lambda Table Coordinate System (Fig 2)

Electro Coordinate System with Lambda Table dimensions (Fig 3)