A Language for Interactive LED Visualization

Hayden Gomes, Emmett McQuinn, Catherine Wah
Fall 2009
Motivation and Goals

- Create language to control LED displays
- Simple to use
- Enable interactive experience
Components

- Interpreter
  - Parser
  - Language design:
    - Encapsulation of LED display
    - Types
    - Functions

- Simulator

- Hardware interface
  - Components
  - Communication protocol
  - Signal converting
Components

Program

Interpreter

Tokenizer (Lex)

Parser (Yacc)

Simulator

LEDs

12/10/2009
Parser

- PLY (Python Lex–Yacc)
  - Extensive error checking
  - Ambiguity resolution via precedence rules
- Output: array representing LED color channels
  - Analogous to skipping Assembly, straight to machine code
Language

- Nested brackets for all commands
  - No ambiguity as to order of operations
- RGB/HSV operations
  - Colorsys module
- Operations performed at current time for simplicity
- Functionality limited by resolution
LED Representation

- 4 : floors
- 70 : LEDs/floor
- 3 : color channels for defining colorspace
- t : visualization program length (default 250)
- $4 \times 70 \times 3 \times t$ array  
  - Passed to simulator and/or hardware controller

- Encapsulated in Display(), along with the current time
# Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Definition</th>
<th>Form(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>x- and y- pixels/ranges of LED array</td>
<td>int * int, (int * int) * (int * int)</td>
</tr>
<tr>
<td>Move</td>
<td>For looking to other LED pixels</td>
<td>int, neg</td>
</tr>
<tr>
<td>Timevar</td>
<td>Positive integer</td>
<td>int</td>
</tr>
<tr>
<td>Num</td>
<td>Float or positive integer</td>
<td>float, int</td>
</tr>
<tr>
<td>Int</td>
<td>Positive integer</td>
<td>int</td>
</tr>
<tr>
<td>Neg</td>
<td>Negative integer</td>
<td>neg</td>
</tr>
<tr>
<td>Float</td>
<td>Floating point</td>
<td>float</td>
</tr>
<tr>
<td>Signal</td>
<td>A wave with color</td>
<td>Numpy.array((3, n))</td>
</tr>
<tr>
<td>Wave</td>
<td>Length n vector defining how signal modulates; n is usually remaining time left in program</td>
<td>Numpy.array((1,n))</td>
</tr>
<tr>
<td>Colorspace</td>
<td>Color channels</td>
<td>(int * int * int)</td>
</tr>
</tbody>
</table>
# Functions (incomplete list)

<table>
<thead>
<tr>
<th>Function</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>timevar -&gt; null</td>
</tr>
<tr>
<td>shift</td>
<td>range * range * move * move * int -&gt; null</td>
</tr>
<tr>
<td>wait</td>
<td>timevar -&gt; null</td>
</tr>
<tr>
<td>set</td>
<td>range * range * signal -&gt; null</td>
</tr>
<tr>
<td>decay</td>
<td>timevar * signal -&gt; signal</td>
</tr>
<tr>
<td>add</td>
<td>signal * signal -&gt; signal</td>
</tr>
<tr>
<td>merge</td>
<td>int * colorspace * wave -&gt; signal</td>
</tr>
<tr>
<td>pulse</td>
<td>timevar -&gt; wave</td>
</tr>
<tr>
<td>sin</td>
<td>num -&gt; wave</td>
</tr>
</tbody>
</table>
Sample programs

- **Rainbow:**
  
  ```
  [time 500]
  [set (0 69) (0 3) [merge 0 (0.8 0.1 0.5) [sin 500]]]
  ```

- **Bars:**
  
  ```
  [time 130]
  [set (0 69) 0 [decay end [merge 2 (0.01 0.5 0.9) [pulse end]]]]
  [set (0 69) 1 [decay end [merge 2 (0.1 0.2 1) [pulse end]]]]
  [set (0 69) 2 [decay end [merge 0 (0.3 0.2 0.8) [pulse end]]]]
  [set (0 69) 3 [decay end [merge 0 (0.7 0.1 0.5) [pulse end]]]]
  [wait 5]
  [shift (0 69) 0 -1 0 70]
  [wait 10]
  [shift (0 69) 1 1 0 70]
  [wait 15]
  [shift (0 69) 2 -1 0 70]
  [wait 5]
  [shift (0 69) 3 1 0 70]
  ```
Language Limitations

- Set-in-space not time
- Lack of variables (except end)
- Signals cannot be saved
- Color normalization
- Amplitude of signal ill-defined
Simulator

- Write pixel values to screen
- Cross Platform (Win/Linux/OSX)
- Draw pixels with OpenGL
- OpenGL's origin is different than array origin, for loops suck in Python so implementation reshuffles array data
- Need thorough verification because we are using it to make inferences about correctness
Simulator ‘99
Simulator 2000™ preview
Hardware

- 4D array to DMX signal converting
- Enttec DMX USB Pro
- Color Kinetics power supplies and fixtures
DMX512

- DMX512 is a standard for digital communication most commonly used for stage lighting
- 512 channels per universe
  - 1 byte per channel
  - 3 channels per fixture
Signal Converting

- Use DMX signaling convention to construct byte array at each time interval
- Transmit to USB Pro through serial USB port connection
- Signaling convention:

<table>
<thead>
<tr>
<th>Size in Bytes</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Start of message delimiter, 0x7E</td>
</tr>
<tr>
<td>1</td>
<td>Label to identify type of message (0x06 for sending)</td>
</tr>
<tr>
<td>1</td>
<td>Data length LSB.</td>
</tr>
<tr>
<td>1</td>
<td>Data length MSB.</td>
</tr>
<tr>
<td>data_length</td>
<td>Data bytes.</td>
</tr>
<tr>
<td>1</td>
<td>End of message delimiter, 0xE7</td>
</tr>
</tbody>
</table>
Hardware Limitations

- 512 channels limit us to 170 fixtures or just shy of 2.5 floors.
  - Possible to have multiple universes but requires additional hardware

- Not easy to translate from specific color to lighting configuration except basics

- Need to run Ethernet wires to connect the floors
  - May need to boost the signal along the way

- Floors were not configured properly to begin with so all fixtures will need to be readdressed.
Future Work

- Integrate optimization techniques
- More informative, language-specific error checking
- Added functionality of language:
  - set-in-time
  - Repeat a pattern
  - Randomizer
- Additional floors
- Simulator 2000
Acknowledgements

- Dave Wargo
- UCSD CSE Department
- Mike Doyle
Demo