Framsticks scripting

Szymon Ulatowski Maciej Komosinski

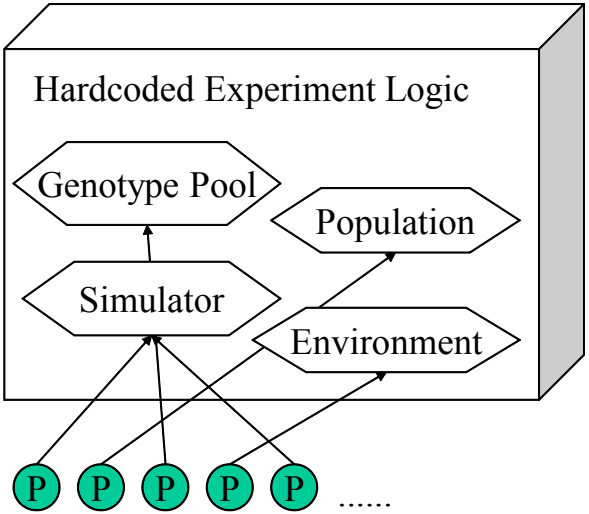
www.framsticks.com

Why use script language?

- In GUI: to automate common/repeating tasks ("macros")
- Algorithmical approach is more straightforward for some purposes
- To simplify internal dependencies
- To give more control to advanced users

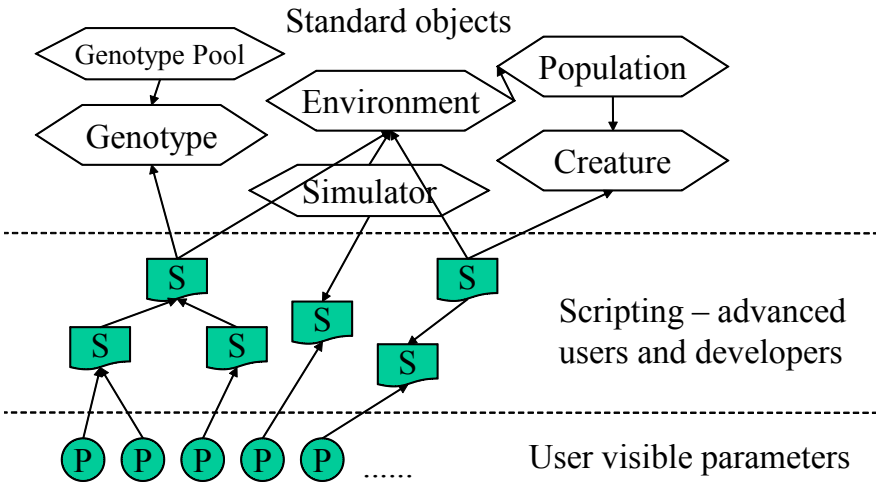
- An advanced user can define new experiments, evaluation methods, neuron types, creature interaction, visualization
- Regular users do not see the involved complexity

Popular Alife Software



User-visible parameters

Framsticks Architecture

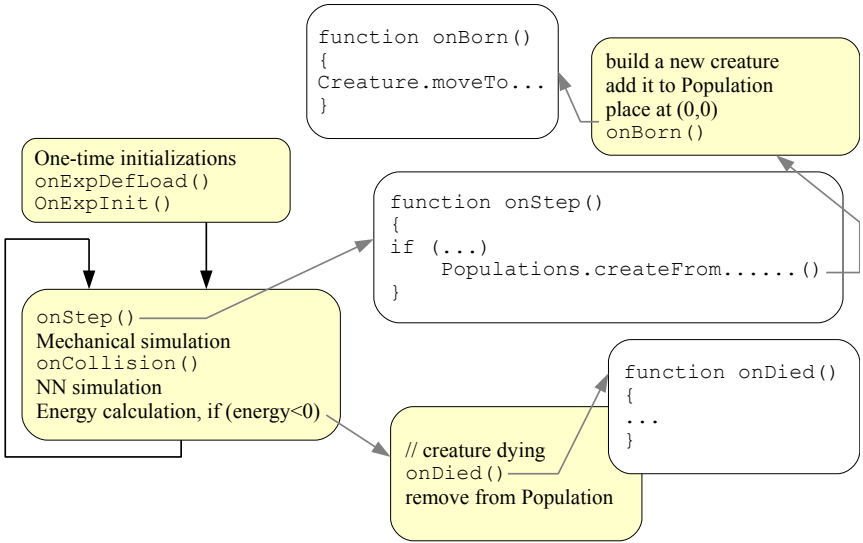


FramScript Language

- JavaScript-like
- 'C' syntax and semantics, untyped variables, functions, can reference Framsticks objects
- Sample code:

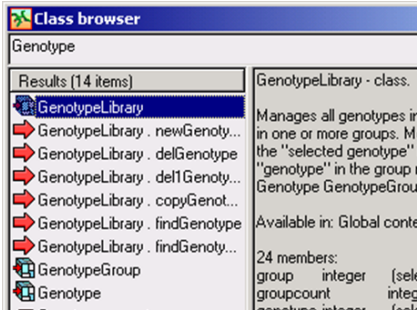
```
function test(angle, mult)
{
    var s = mult * Math.sin(angle);
    return s + ExpProperties.uservalue;
}
```

Event–based architecture: main loop and script handlers

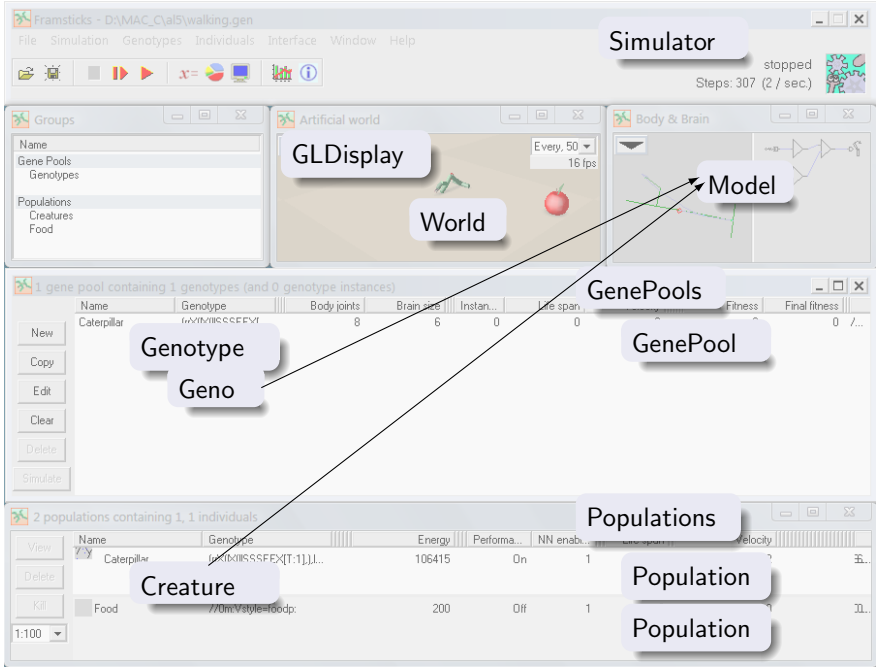


Object Model

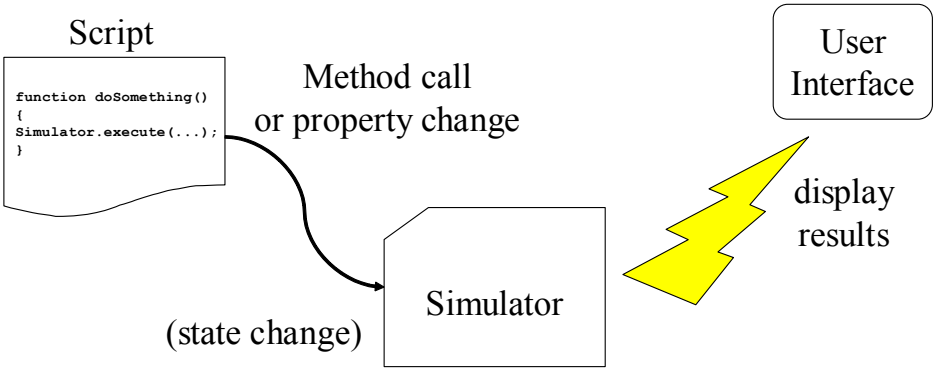
- Core Simulator objects (Genotype, Creature, World, ...)
- User Interface objects (CLI, GLDiplay, Chart, ...)
- General Purpose objects (Math, Collections, ...)



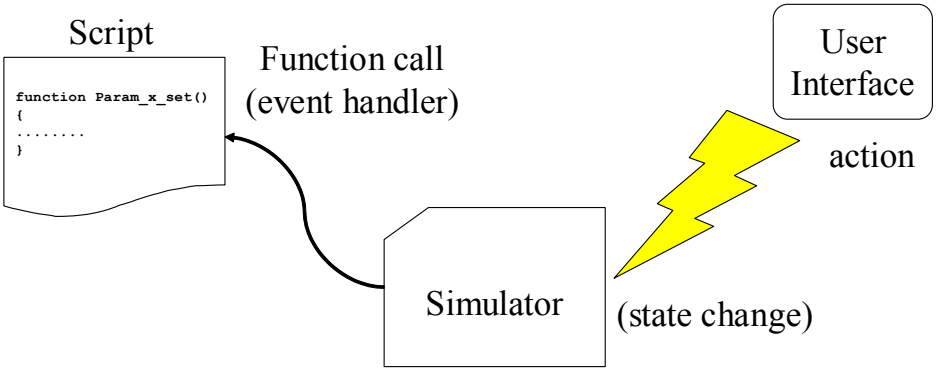
Class reference available in the Framsticks application: Help/Class Browser



Script/Simulator/User



User/Simulator/Script

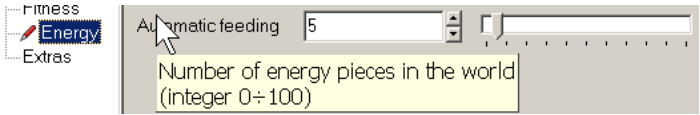


User-visible properties

- Definition:

```
property:
id:feed
name:Automatic feeding
type:d 0 100 5
group:Energy
help:Number of energy pieces in the world
```

- Presentation:



- Script access:

```
ObjectName.feed=12;
function ObjectName_set_feed() // 'feed' changed
{
   // handle modification of the 'feed' variable
}
```

Property types

- **"d" – integer value**
  - "d min max" – integer with constraints
  - "d 0 1" – false/true switch (checkbox) 
  - "d min max ∼name1∼name2∼name3..." – integer presented as named values (combo) 

- **"f" – floating point value**
  - "f min max" – floating point with constraints

Property types

- **"s" – text string**
  - "s 1 maxlength" – multiline text
  - "s ∼value1∼value2∼value3..." – text with suggested default values (combo)
- **"p" – procedure call** (action), a button in GUI

- **"o" / "x"** – object property/unknown type property (not useful in the interface)

Editors

These syntax highlighting editors will make script writing easier:

- Emacs, www.gnu.org/software/emacs
- jEdit, www.jedit.org
- Crimson Editor, www.crimsoneditor.com
- (your favorite editor with syntax highlighting)

...but the recommended tool is Framclipse.

Framclipse: a plugin for Eclipse

- Fully integrated with the Eclipse platform
- Easy installation – "Install New Software" Eclipse feature

Framclipse: a plugin for Eclipse

- Fully integrated with the Eclipse platform
- Easy installation – "Install New Software" Eclipse feature
- Syntax highlighting

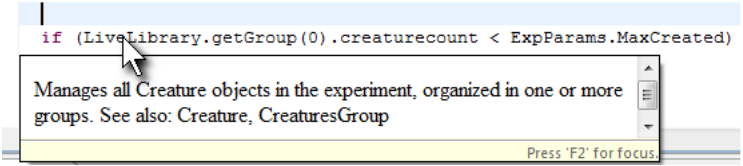
```
//comment
function onExpLoad()
{
  Simulator.message("expdef: onExpLoad not implemented", 3);
}
```

Framclipse: a plugin for Eclipse

- Fully integrated with the Eclipse platform
- Easy installation – "Install New Software" Eclipse feature
- Syntax highlighting

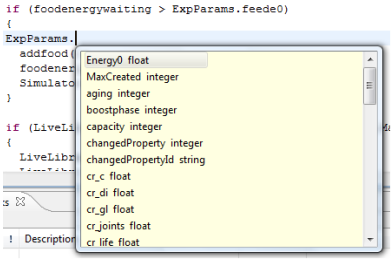
```
//comment
function onExpLoad()
{
  Simulator.message("expdef: onExpLoad not implemented", 3);
}
```

- Documentation popups



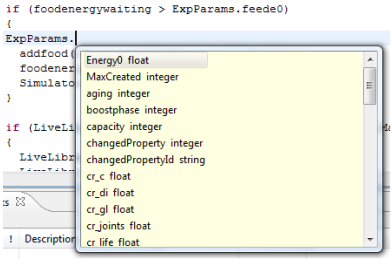
Framclipse: more features

- Content assistance

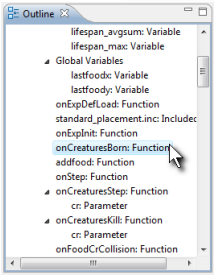


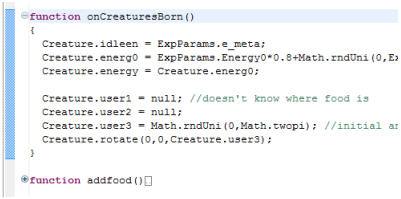
Framclipse: more features

- Content assistance



- Interactive outline
- And much more.





Script applications in Framsticks

- Neuron definitions
- Experiment definitions
- Advanced 3D visualization
- "Framsticks Theater" presentations
- General purpose scripting (macros)

Neuron definitions

- Neuron: a signal processing unit
- Direct control of the neuron output
- Can use any user defined functions, read from neuron inputs, preserve state in the neuron itself
- Public properties can be used to influence the neuron behavior. Genetic operators will accept and operate
  on those properties

Neuron definitions code

- Initialization function ("constructor") – set the internal state if needed

```
function init()
{ NeuroProperties.somestate=0; }
```

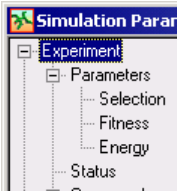
- Working function – calculate the new state

```
function go()
{ Neuro.state=...something...; }
```

- Properties (accessible as "NeuroProperties")
  - Public properties (with flags=0) – initialized based on the genotype data, and available for genetic operators
  - Private properties (with flags=32) – only for a neuron script, for internal purposes

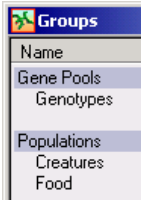
Experiment definition

- Controls the way objects are created in the world and how they are evaluated
- Can create user-visible parameters (ExpProperties) and experiment state variables (ExpState)
- Responsible for saving/restoring experiment state



Experiment definition: Groups

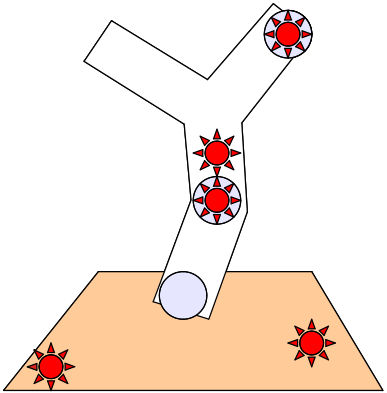
- Interaction with User Interface
- Script access:
  - GenePools
    GenePool
    Genotype
  - Populations
    Population
    Creature



Communication: Signal objects

Why use signals:

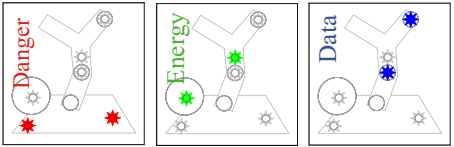
- Optimized for common cases
- Unified programming interface for stationary, 
  neuron-attached, 
  and creature-attached signals 
- Visualization in GUI and Theater



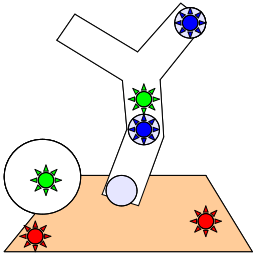
Communication: Signal objects

Signal properties:

- Channel



- Power
- Flavor 
- Value (any object)



Communication: Signal objects

Receiving aggregated power

- Low level, "biological" senses
- Easy, one-line custom neurons:

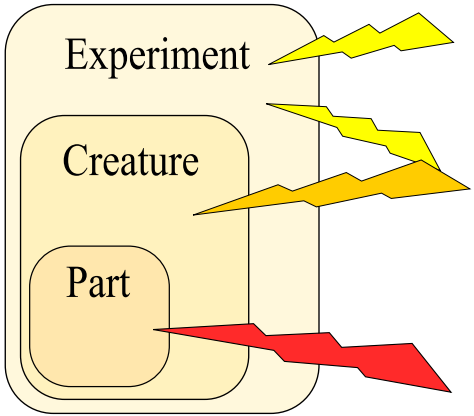
```
Neuro.state=Neuro.signals.
 receiveFilter("channel",
 max_range, flavor, filter);
```

Receiving objects

- Can pass any kind of data
- More sophisticated processing:

```
var n=Creature.signals.
 receiveSet("flock",range);
for(i=0;i<n.size;i++)
 doSomething(n[i].value);
```

Experiment definition: Events



onExpDefLoad(), onExpInit(), onExpLoad(), onExpSave(), onStep()

onBorn(), onDied(), onStep(), onUpdate(), onCrCollision()

onCollision()

Experiment definition: Events

- `onExpDefLoad()`
  The experiment definition is loaded.
  Usual actions:
  - Setup genotypes and creatures groups
  - Initialize experiment parameters and state

- `onExpInit()`
  New experiment started.
  - Clear all state information

Experiment definition: Events

- `onBorn()`
  A new creature was born (as a result of script or user action).
  Usual handling:
  - Adjust placement
  - Initialize creature's properties
- `onGroupNameBorn(creature_arg)`
  Group-specific handler for onBorn()

Experiment definition: Events

- `onDied()`
  The creature has died. Usual handling:
  - Update performance data about its genotype.
- `onGroupNameDied(creature_arg)`
  Group-specific handler for onDied()

Experiment definition: Events

- `onStep()`
  Called before each simulation step, once
- `onGroupNameStep(creature_arg)`
  Called for each creature in a group

- `onUpdate()`
  Called periodically as requested by the "performance sampling period" parameter. Creature's performace fields (distance, velocity) have been updated.
- `onGroupNameUpdate(creature_arg)`
  Group-specific handler for onUpdate()

Experiment definition: Events

- `onGroupNameCollision()`
  Called for each creature part touching another creature. Usual handling:
  - Energy transfer (see "standard.expdef")
  - Update performance
- `onGroupNameCrCollision()`
  Similar, for creature-level collisions

Experiment definition: Events

- `onExpSave()`
  Should save experiment state using the "File" object. Sample usage:

```
File.writeComment("my parameters");
File.writeObject(sim_params.*);
```

- `onExpLoad()`
  Restore the experiment state from the file created in the "ExpSave" event. The "Loader" object can parse and load multiple objects.

```
Loader.addClass(sim_params.*);
Loader.run();
```

Experiment definition: Events

UserEvent: An action users can launch from the context menu,
eg. "Place food" and "Drop food" found in most standard experiments (defined in **standard_events.inc**).

- `queryEventNames()`
  Return the supported event names (a vector of text strings).
- `onUserEvent(type,point,vector)`
  Handle the event. Point and vector are 3-element vectors describing the world location where the user action occurred.

Experiment definition: Events


```
function queryEventNames()
{
  return ["Place food", "Drop food"];
}

function onUserEvent(type, point, vector)
{
  var p = WorldMap.intersect(point, vector);
  if (p)
  {
    addfood();
    var z = p[2];
    if (type == 1) z += 10;
    Creature.moveAbs(p[0]-Creature.size_x/2,
                     p[1]-Creature.size_y/2,z);
  }
}
```

Experiment definition: Low-level mouse interaction

- `onMouseClick(options, xy, point, vector)`
- `onMouseMove(options, xy, point, vector)`
- `onMouseUnclick(options, xy, point, vector)`

Arguments:

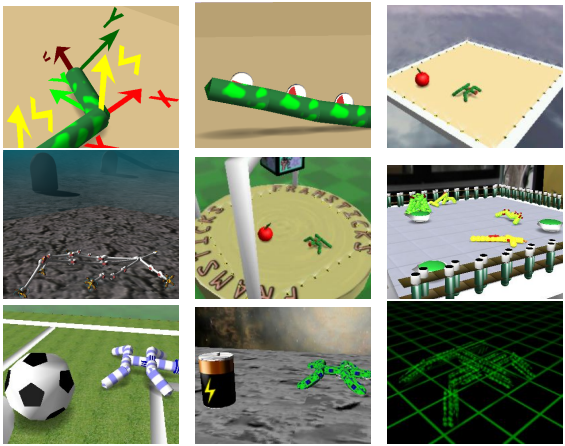
- **options** – mouse button and double click identification
- **xy** – 2D coordinates (xy[0] and xy[1])
- **point, vector** – 3D "line of click" defined by point and vector

Advanced 3D visualization

- OpenGL – 3D rendering engine
- PLIB/SSG – Scene graph
- Script:
  - Create or load geometry and textures for all objects (environment, creatures)
  - Update object properties (positions)

Advanced 3D visualization: style definition

- Style: predefined visualization functions with associated graphics files



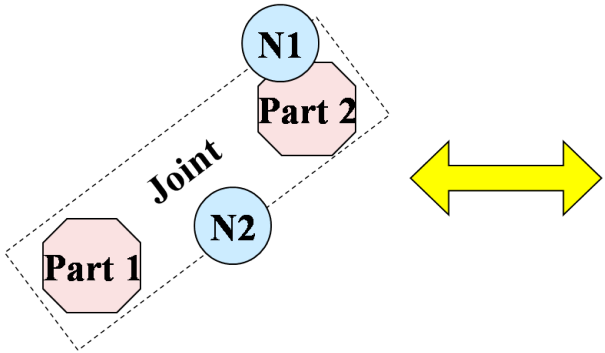
Examples: XYZ, pressure, standard, spooksticks, arena, laboratory, football, space, matrix

Advanced 3D visualization: objects

- **GeomBuilder**
  - Create/modify scene graph(SSG nodes manipulation)
  - Special support for VisualModel
- **VertexBuilder**
  - Create any shape (low level: points, polygons)
- **Loader**
  - Load 3D object files (SSG)
- **Material**
  - Create/modify object attributes – colors, textures (SSG)

Advanced 3D visualization: objects

- **VisualModel**
  - Mapping: creature element ↔ scene node



- *Creature root node*
  - *Part1*
    - *N1*
  - *Part2*
  - *Joint*
    - *N2*

Advanced 3D visualization: script functions

- `onLoad()`
- `stylename_model_build()` – called once, before the visual object is created
- `world_build()` – called when the world parameters change
- `modelviewer_build_empty()`
- `modelviewer_build()`

Advanced 3D visualization: script functions

- `stylename_elementname_build()` – create a 3D object associated with the current element
- `stylename_elementname_update()` – update the 3D object according to the current properties

- stylename: "default", unless set to another name in the model (eg. "food", "manipulator")
- elementname: "part", "joint", "neuro" unless set to another name in the model element

Advanced 3D visualization: script sample #1


```
function default_flag_build()
{...}
```

Listing: Style definition

Genotype

```
...
p:3,1.72,6,Vstyle=flag
...
```

Result

Advanced 3D visualization: script sample #2

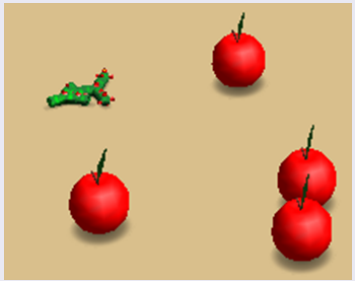

```
function food_part_build()
{...}
```

Listing: Style definition

Genotype

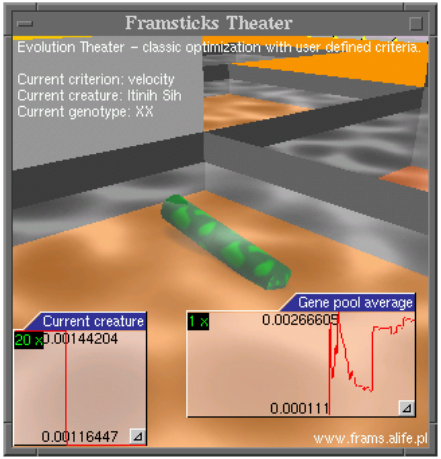
```
//0
m:Vstyle=food
p:
```

Result



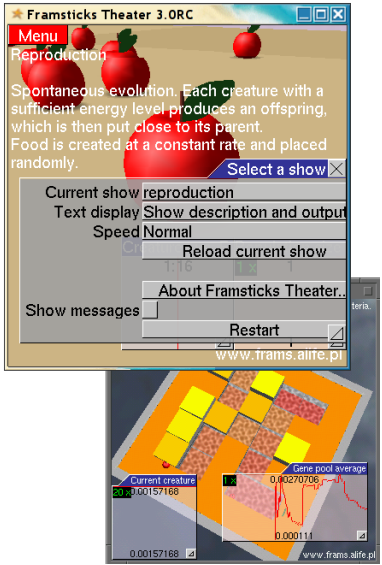
Framsticks Theater presentations

- Setup the simulator for a predefined experiment
- Provides a simple, attractive graphical user interface
- Can define user-visible parameters
- Can create and use additional GUI objects



Framsticks Theater presentations

- UI capabilities:
  - **ShowProperties** object (user-visible parameters and actions presented in the menu)
  - **GLDisplay.banner="text"** (simple text display)
  - **GLDisplay.createWindow(style, title, client)**
  - **GLDisplay.newCreatureCharts()**
  - **GLDisplay.newSimStatsCharts()**



- Events:
  - `onLoad()`, `onResize()`, `onShowStep()`, `onSimStep()`, `onSelectionChange()`
- Objects:
  - `GLDisplay`, `Camera`, `Window`, `DynaChart`

General purpose scripting

- Used for repeating tasks and testing
- Can access rarely used features without a user interface
- Command line simulator: all user actions are actually a script
- GUI version: user scripts can be accessed from the "Parameters" window

Let's create a neuron that occasionally generates noise.

The neuron should occasionally generate random output, otherwise it should work like the regular "N" neuron.

Let's create a neuron that occasionally generates noise.

The neuron should occasionally generate random output, otherwise it should work like the regular "N" neuron.

General neuron information (inside the "class:" section)

```
class:
name:Nn
longname:Noisy neuron
prefinputs:-1
prefoutput:1
code:~
..... the script code will be here....
~
```

One public parameter – "Error rate" (probability of response perturbation)

```
property:
id:e
name:Error rate
type:f 0.0 0.1
```

Let's create a neuron that occasionally generates noise.

The neuron should occasionally generate random output, otherwise it should work like the regular "N" neuron.

General neuron information (inside the "class:" section)

```
class:
name:Nn
longname:Noisy neuron
prefinputs:-1
prefoutput:1
code:~
..... the script code will be here....
~
```

One public parameter – "Error rate" (probability of response perturbation)

```
property:
id:e
name:Error rate
type:f 0.0 0.1
```

```
function go()
{
    var s=Neuro.weightedInputSum;
    if (Math.rnd01 < NeuroProperties.e)
        s=(Math.rnd01*2)-1.0;
    Neuro.state=s;
}
```

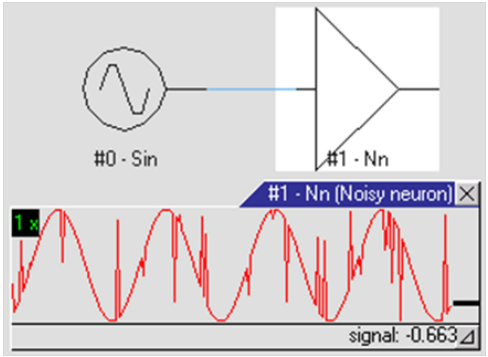
Listing: Work function

Complete source and demonstration

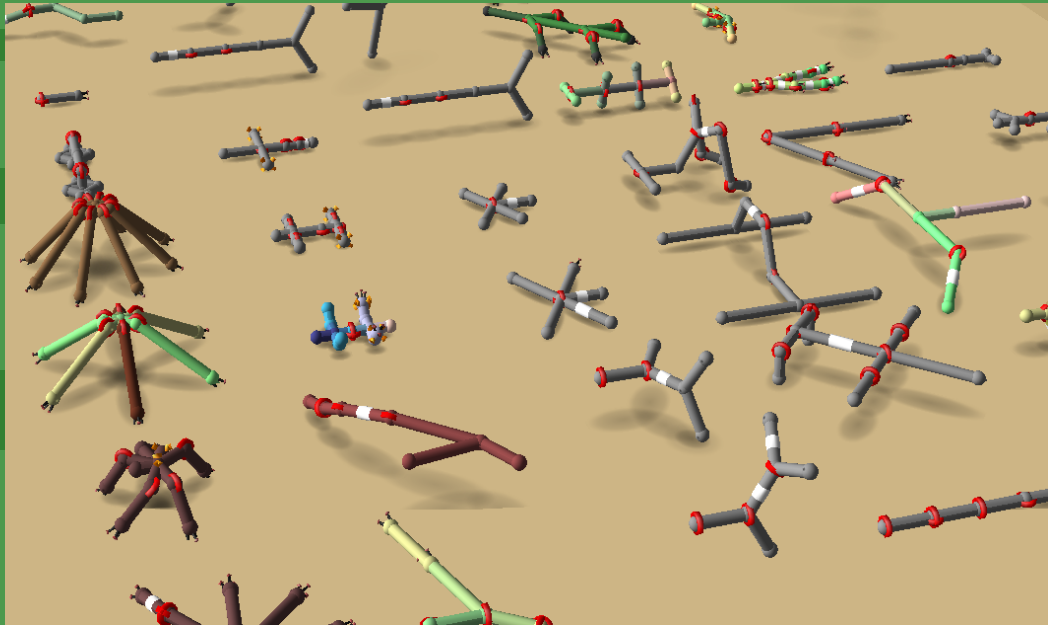
The "noisy.neuro" file

```
class:
name:Nn
longname:Noisy neuron
prefinputs:-1
prefoutput:1
code:~
function go()
{
   var s = Neuro.weightedInputSum;
   if (Math.rnd01 < NeuroProperties.e)
      s = (Math.rnd01 * 2) - 1.0;
   Neuro.state=s;
}
~

property:
id:e
name:Error rate
type:f 0.0 0.1
```



Let's place all creatures in a grid (e.g. for a presentation)



Script header

General script information (inside the "script:" section)

```
script:
name:Gallery
help:Create a grid of creatures from the first genotype list
code:~
..... the script will be here....
~
```

Script source


```
var gridx = 6.0, gridy = 6.0; // grid spacing
var n = GenePools[0].size;
if (n==0)
{
  Simulator.print("This script needs some genotypes "+"in the first group.");
  return;
}
var side = 0+(Math.sqrt(n)+0.999);
Simulator.print("Have "+n+" genotype(s), it will be "+side+" by "+side+" grid");

var g=0;
for(var i=0;i<side;i++)
  for(var j=0;j<side;j++)
  {
    if (g >= GenePools[0].size)
       return;
    Populations[0].createFromGeno(GenePools[0][g].geno);
    var x = gridx*i;
    var y = gridy*j;
    var z = WorldMap.getHeight(x,y);
    Creature.moveAbs(x-Creature.size_x/2, y-Creature.size_y/2, z);
    g++;
  }
```

Let's create an interesting evolutionary experiment

- Creatures gain energy by eating (touching) food
- Each creature with a sufficient amount of energy produces an offspring, which may be a clone or a mutant
- New food pieces are created in the world and placed randomly

General experiment information (inside the "expdef:" section)

```
expdef:
name:Asexual reproduction in the world
info:~
Each creature with a sufficient amount of energy
produces an offspring, which is then put close to its parent.
Food is created at a constant rate and placed randomly.
~
code:~
...this will contain the script...
~
```

Experiment parameters

These parameters will be public, available in GUI and for other components

```
property:
id:p_mutate
name:Mutation probability
type:f 0 100

property:
id:reprEnergy
name:Reproduction energy
type:f 0 10000
group:Energy
help:Creature energy required to produce an offspring

property:
id:e_meta
name:Idle metabolism
type:f 0 1
group:Energy
help:Each creature consumes this amount of energy in one time step

...
```

Key functions


```
global foodenergywaiting;

function onExpDefLoad()
{
  // define genotype and creature groups
  GenePools.clear();
  GenePool.name="Unused";
  Populations.clear();
  Population.name="Creatures";
  Population.nnsim=1;
  Population.enableperf=1;
  Population.colmask=4;
  Populations.addGroup("Food");
  Population.colmask=5;
  Population.nnsim=0;
  Population.enableperf=0;

  // initialize experiment parameters
  ExpProperties.p_mutate=0.5;
  ExpProperties.reprEnergy=1001;
  ExpProperties.e_meta=0.1;
  ExpProperties.feedrate=0.2;
  ExpProperties.feede0=200;
  foodenergywaiting=0.0;
}
```

```
function onExpInit()
{
  Populations.clearGroup(0);
  Populations.clearGroup(1);
  foodenergywaiting=0.0;
  var cr=Populations.createFromString(
    "...(initial genotype)...");
  cr.name="Initial creature";
  place_randomly(cr);
}

function onStep()
{
  foodenergywaiting += ExpProperties.feedrate;

  if (foodenergywaiting > ExpProperties.feede0)
  {
    addFood();
    foodenergywaiting=0.0;
  }

  if (Populations[0].size==0)
  {
    Simulator.print("no more creatures");
    Simulator.stop();
  }
}
```

Creature management


```
function onCreaturesBorn()
{
 Creature.idleen=ExpProperties.e_meta;
 Creature.energ0=1000;
 Creature.energy=Creature.energ0;
}

function onFoodCollision()
{
 var e=Collision.Part2.ing;

 Collision.Creature1.energy_m =
   Collision.Creature1.energy_m+e;

 Collision.Creature2.energy_p =
   Collision.Creature2.energy_p+e;
}
```

```
function onCreaturesStep(cr)
{
 if (cr.energy>=ExpProperties.reprEnergy)
 {
   var newcreature;
   if (Math.rnd01 > ExpProperties.p_mutate) // exact copy
   {
     GenePools.getFromCreatureObject(cr);
     newcreature=Populations[0].createFromGenotype();
   }
   else
   { // mutation
     GenePools.getFromCreatureObject(cr);
     GenePools.mutate();
     newcreature=Populations[0].createFromGenotype();
   }
   if (newcreature!=null)
   {
     newcreature.energ0=cr.energy/2;
     newcreature.energy=newcreature.energ0;
     newcreature.moveAbs(
       cr.pos_x+(cr.size_x-newcreature.size_x)/2,
       cr.pos_y+(cr.size_y-newcreature.size_y)/2,
       cr.pos_z+(cr.size_z-newcreature.size_z)/2);
     cr.energy=cr.energy/2;
   }
 }
}
```