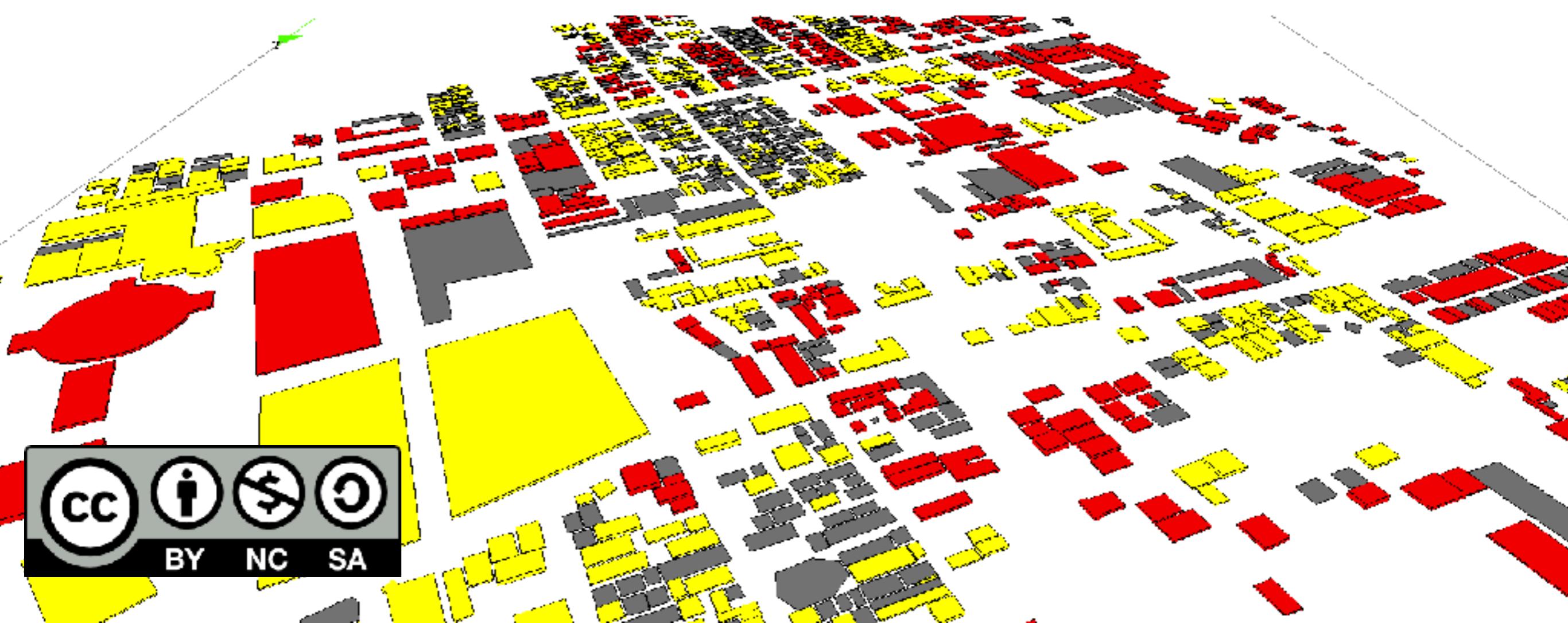


A Practical introduction to GAMA Through a Segregation model

Benoit Gaudou, IRIT (Univ. Toulouse 1), benoit.gaudou@ut-capitole.fr

Patrick Taillandier (INRA, MIA, Toulouse), patrick.taillandier@gmail.com

with the participation of Alexis Drogoul, Jean-Daniel Zucker, Philippe Caillou and Srirāma Bhamidipati



GAMA: How to get documentation ... and ask for help if needed

- ▶ GitHub: <https://github.com/gama-platform/gama>

The screenshot shows the GitHub repository page for 'gama-platform / gama'. The repository has 5,316 commits, 3 branches, 0 releases, and 11 contributors. The 'Issues' link in the sidebar is highlighted with a red box, showing 103 issues. The 'Wiki' link in the sidebar is also highlighted with a red box.

Core plug-in projects of the GAMA platform — Edit

5,316 commits 3 branches 0 releases 11 contributors

Branch: master — gama / +

add a facet when to the focus statement

mathieuBourgais authored 4 days ago latest commit f67f3d2a31

centres.gaml.extensions.hydro Forces every project to comply with the default compiler defined in 2 months ago

dream.gama.opengis Merge branch 'master' of https://github.com/gama-platform/gama.git 3 months ago

idees.gama.features.weka git-svn-id: https://gama-platform.googlecode.com/svn/branches/GAMA CU... a year ago

idees.gama.mapcomparison Forces every project to comply with the default compiler defined in 2 months ago

idees.gama.weka Forces every project to comply with the default compiler defined in 2 months ago

irit.gaml.extensions.database Forces every project to comply with the default compiler defined in 2 months ago

irit.gaml.extensions.test Makes GAMA compatible with a compilation using the JDK 1.8 compiler 2 months ago

irit.maelia.gaml.additions Forces every project to comply with the default compiler defined in 2 months ago

msl.gama.application Should enable (at last) Windows versions to display the left icons in a month ago

msl.gama.core fix an issue for save statement and asc file (case where there is no 10 days ago

msl.gama.display.web Forces every project to comply with the default compiler defined in 2 months ago

Code Issues 103 Pull requests 0 Wiki

Pulse Graphs Settings

HTTPS clone URL <https://github.com/gama-platform/gama.git>

You can clone with [HTTPS](#), [SSH](#), or [Subversion](#).

Clone In Desktop Download ZIP

GAMA: How to get documentation

Contents

I Platform	21
1 Installation	25
Table of contents	25
System Requirements	25
Installation of Java	26
2 Launching GAMA	29
Table of contents	29
Launching the Application	29
Choosing a Workspace	31
Welcome Page	32
3 Headless Mode	35
Table of contents	35
Command	36
Experiment Input File	37
Output Directory	39
Simulation Output	40
Snapshot files	41

3

Primer of the GAML 1.6.1 modeling language

13/08/2014

Structure of a model/program

```
Program myFirstModel
global {
    // Defines all global variables,
    // model initialzation and global behaviors.
}

species mySpecies1
    // Defines variables, behaviors and aspects
    // of agents of the species.

experiment expName
    // Defines the way the model will be executed
    // includes the type of the execution,
    // which global parameters can be modified,
    // and what will be displayed during simulation.
```

Comments

Block comments	<code>/* A block comment starts with the on opening symbol /* . The comment runs until the closing symbol below. */</code>
Inline comments	<code>// This is an inline comment. // The // symbol have to be repeated before every line.</code>

Primitive types

Integer number <code>#value between -2147453648 and 2147453647</code>	<code>int</code>
Real number <code>#absolute value between 4.9*10⁻³⁸ and 1.8*10³⁰⁸</code>	<code>float</code>
String <code>#explicit value "double quotes" or 'single quotes'</code>	<code>string</code>
Boolean value <code>#2 values: true, false</code>	<code>bool</code>

Other types

pair #with the two elements of undefined types pair #with two elements of types type1 and type2 <code>#explicit value using : symbol: e.g. 1:"one"</code>	<code>pair</code>
color <code>#explicit value: rgh(255,0,0) for red. (3 components: Red, Green, Blue)</code>	<code>rgb</code>
point <code>#explicit value: (1,0, 3) or [1,0, 3, 6]. #internal representation with 4 coordinates</code>	<code>point</code>

Variable or constant declaration, affectation

Declaration of a global variable or an attribute <code># Global variables and species attributes can be declared with or without initial value.</code>	<code>// Global variables or species attributes int an_int; string a_string += "my string";</code>
# Local variables need to be initialized when they are declared. <code># explicit declaration of the type # (if the type of the assigned value is different, this value is automatically casted to the declared type)</code>	<code>// Local variables float a_float <- 10.0;</code>
Declaration of a global variable or an attribute with a dynamic value <code>#value computed at each simulation step</code>	<code>// Global variables or species attributes with dynamic value // inc_int is incremented by 1 at each simulation step int inc_int <- 0 update: inc_int + 1;</code>
<code>#value computed each time the variable is used</code>	<code>// random_int has a new random value each time it is used: int random_int >- { rnd(100) };</code>
Definition of a constant	<code>float pi <- 3.14 const: true;</code>
Affectation of a value to a variable <code>Variable <- value or computed expression</code>	<code>// Affectation of a value to an existing variable an_int <- 0;</code>

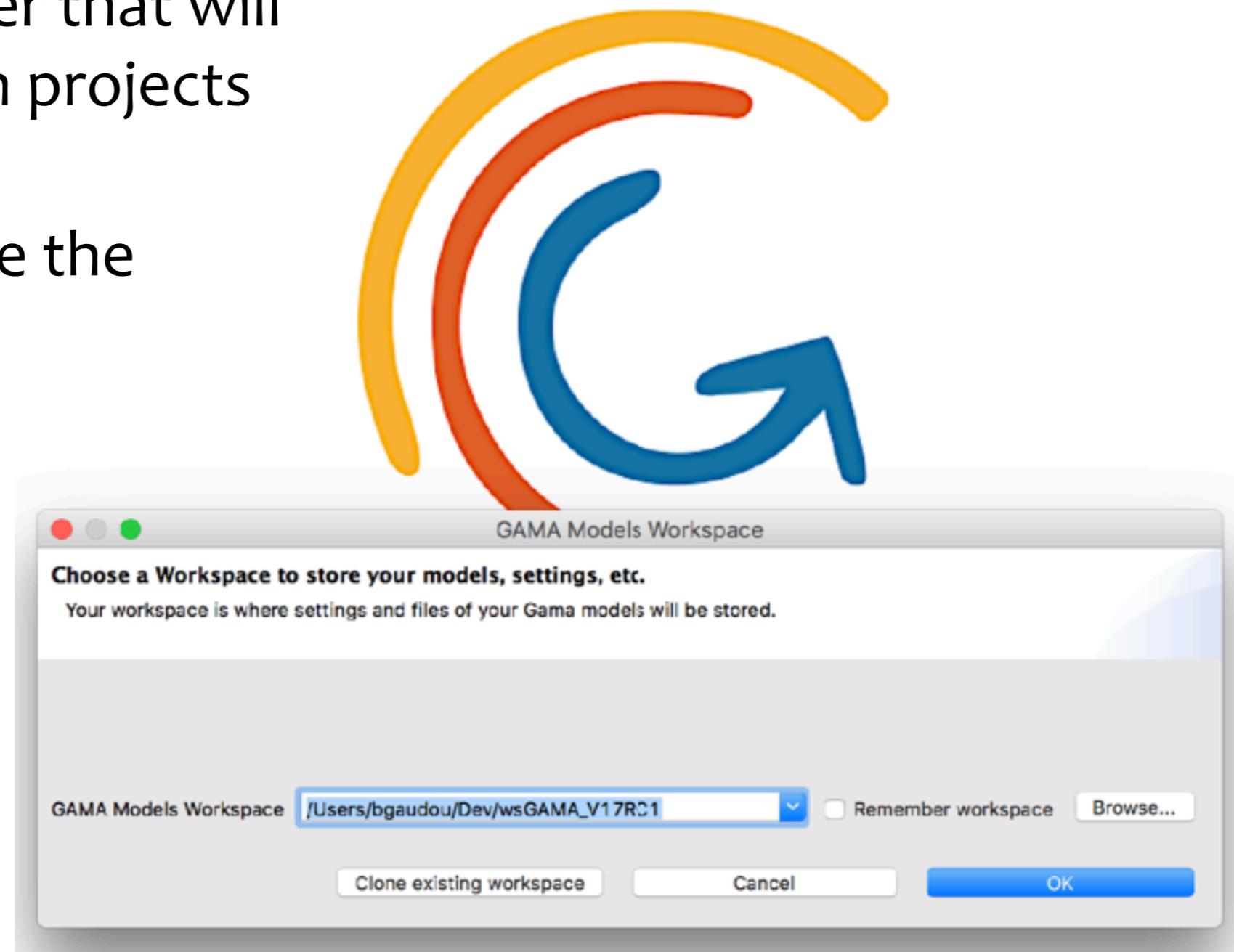
<http://gama-platform.org/>

GAMA Team

1

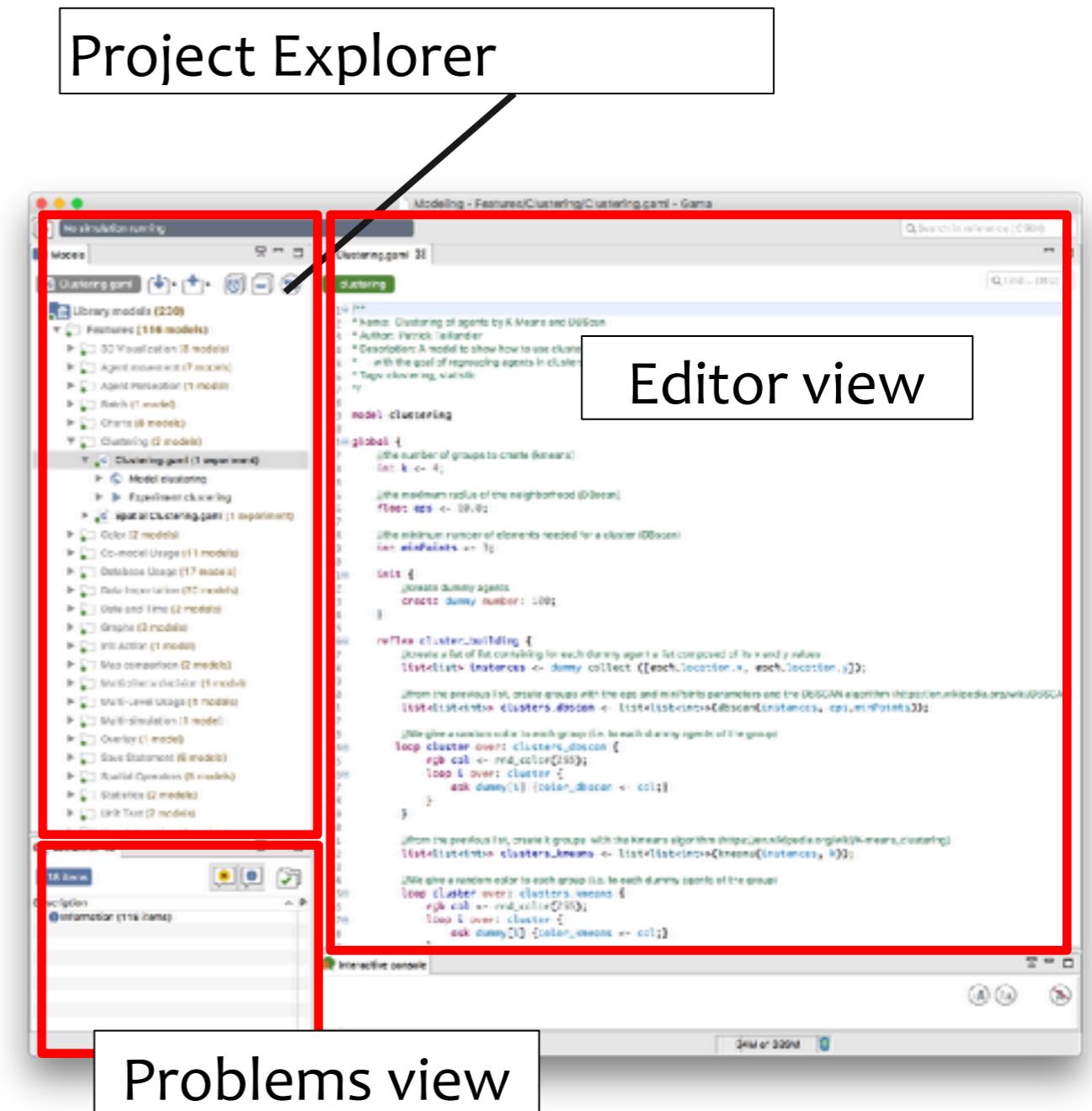
It is now time to run GAMA !

- ▶ First GAMA asks you to choose a workspace.
- ▶ A workspace is a folder that will contain all your own projects and models.
- ▶ You are free to choose the folder you want!



GAMA model files are stored in projects

- ▶ Each project may contain several models, as well as additional resources (GIS data, pictures,...).
- ▶ Belonging to the same project allows models to use each other and have access to the same resources.
- ▶ Projects can be organised in any way, although a default layout (“models”, “doc”, ...) is proposed.



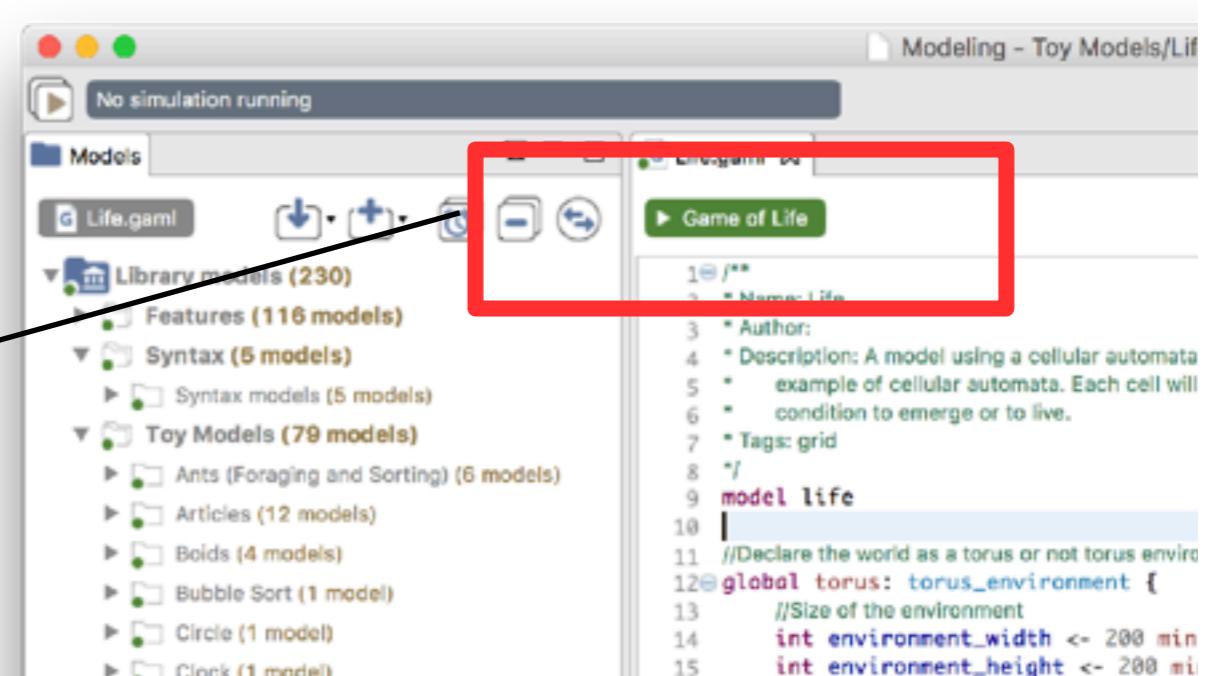
GAMA provides a library of models composed of 4 types of models:

- **Features:** simple models that show how to use some specific features (graph, GIS data, 3D....)
- **Syntax:** models that present the syntax of the GAML language
- **Toy models:** classic agents-based models (ant models, boids, Schelling...)
- **Tutorials:** models linked to online tutorials

Take a look at “Game of life” model in library

- Models library \ Toy models \ Life \ Life.gaml

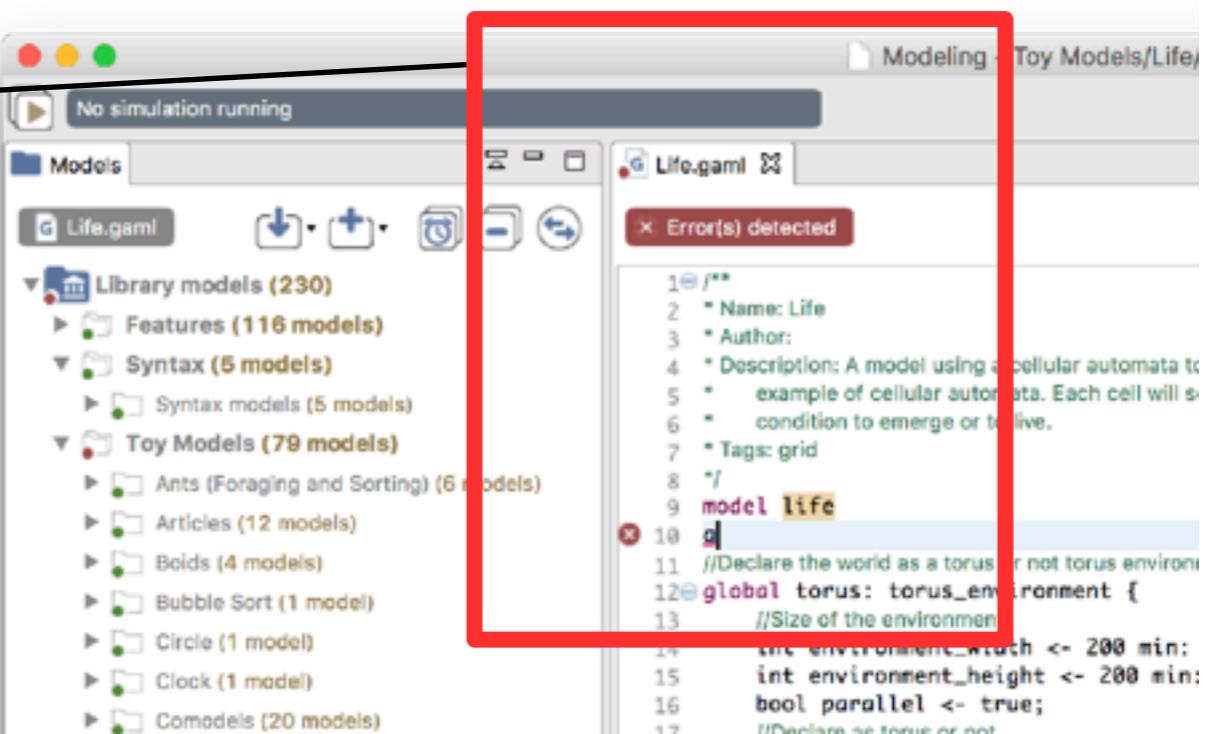
The model can be experimented



A screenshot of the Modeler application interface. On the left, the 'Models' panel shows a tree structure with 'Library models (230)' expanded, revealing categories like 'Features', 'Syntax', and 'Toy Models'. The 'Toy Models' category is selected and expanded, showing sub-models such as 'Ants', 'Articles', 'Boids', 'Bubble Sort', 'Circle', and 'Clock'. In the center, a preview window titled 'Game of Life' displays a green grid pattern. On the right, the code editor shows the GAML code for the 'Game of Life' model. A red box highlights the first few lines of the code.

```
1 /**
2 * Name: Life
3 * Author:
4 * Description: A model using a cellular automata to
5 * example of cellular automata. Each cell will
6 * condition to emerge or to live.
7 * Tags: grid
8 */
9 model life
10
11 //Declare the world as a torus or not torus environment
12@ global torus: torus_environment {
13    //Size of the environment
14    int environment_width <- 200 min
15    int environment_height <- 200 min
16    bool parallel <- true;
17    //Declare no torus or not
```

The model has errors



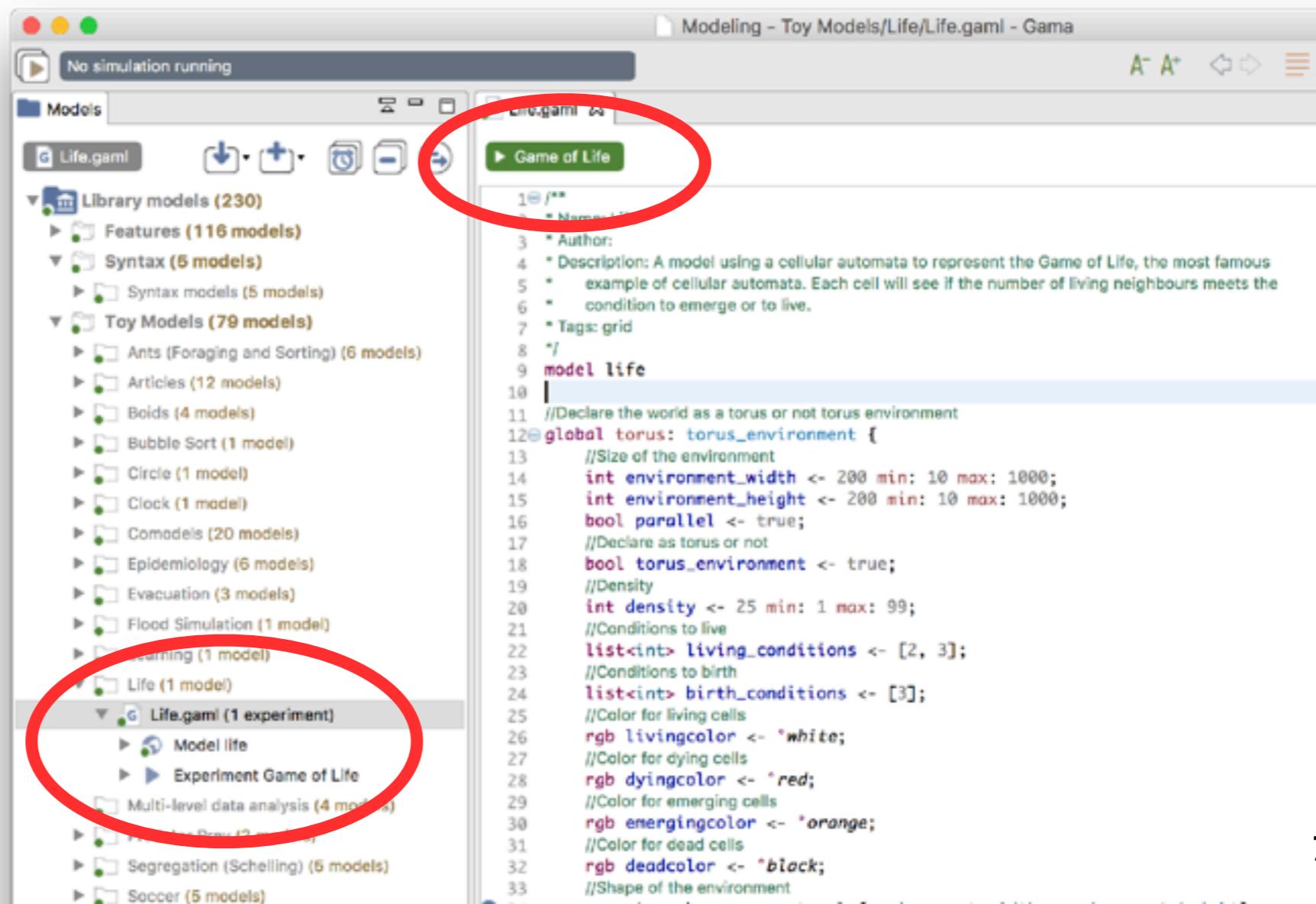
A screenshot of the Modeler application interface, similar to the one above but with error detection enabled. The 'Models' panel and tree structure are identical. In the code editor on the right, a red box highlights the line 'model life' which is underlined with a red squiggle, indicating a syntax error. A message 'x Error(s) detected' is displayed above the code editor.

```
1 /**
2 * Name: Life
3 * Author:
4 * Description: A model using a cellular automata to
5 * example of cellular automata. Each cell will
6 * condition to emerge or to live.
7 * Tags: grid
8 */
9 model life
10
11 //Declare the world as a torus or not torus environment
12@ global torus: torus_environment {
13    //Size of the environment
14    int environment_width <- 200 min
15    int environment_height <- 200 min
16    bool parallel <- true;
17    //Declare no torus or not
```

Launch experiment Game of life

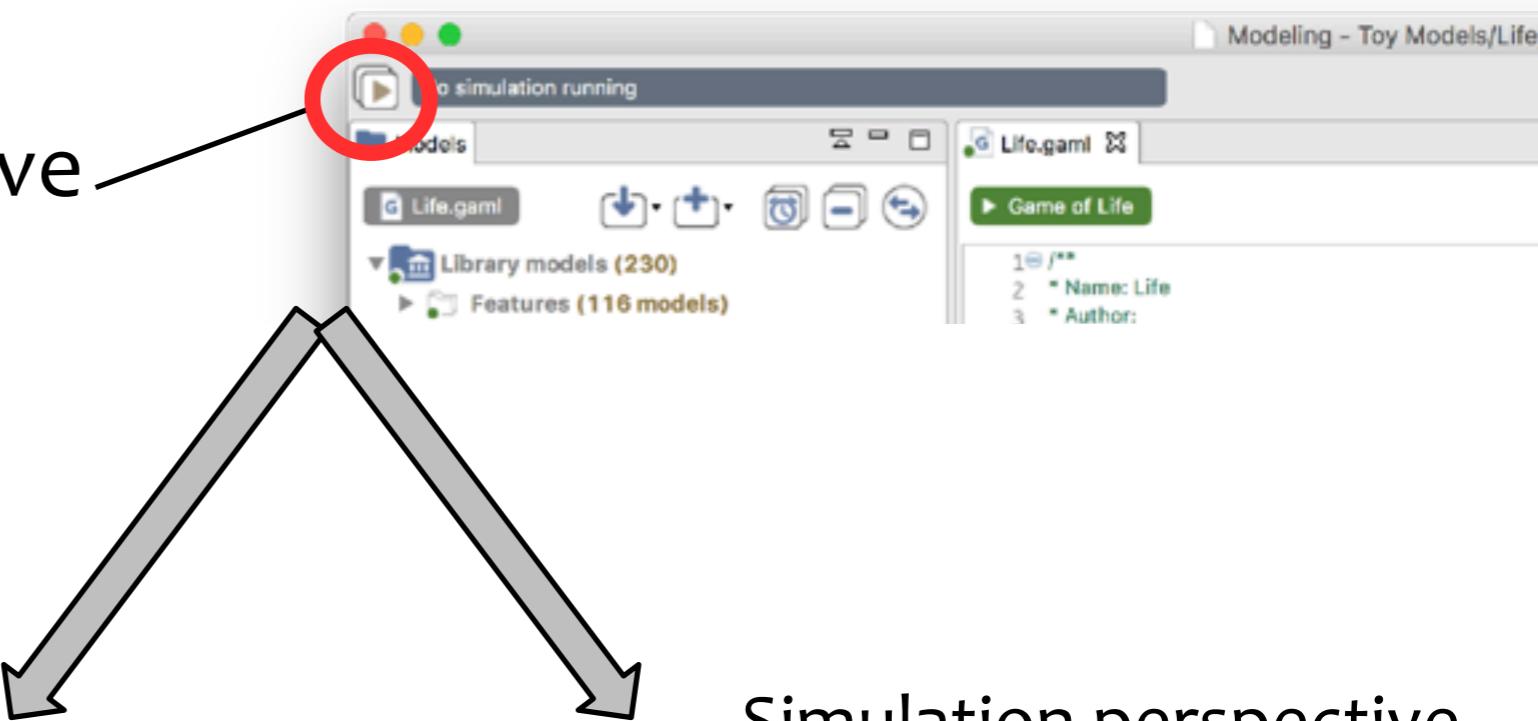
- ▶ An experiment is a way to “run” a model. It can be reached either by:

- ▶ Clicking on an Experiment button
- ▶ in the Project Explorer, under the name of the model



Launching an experiment will switch from *Modelling* to *Simulation* Perspective

▶ Allows to change perspective

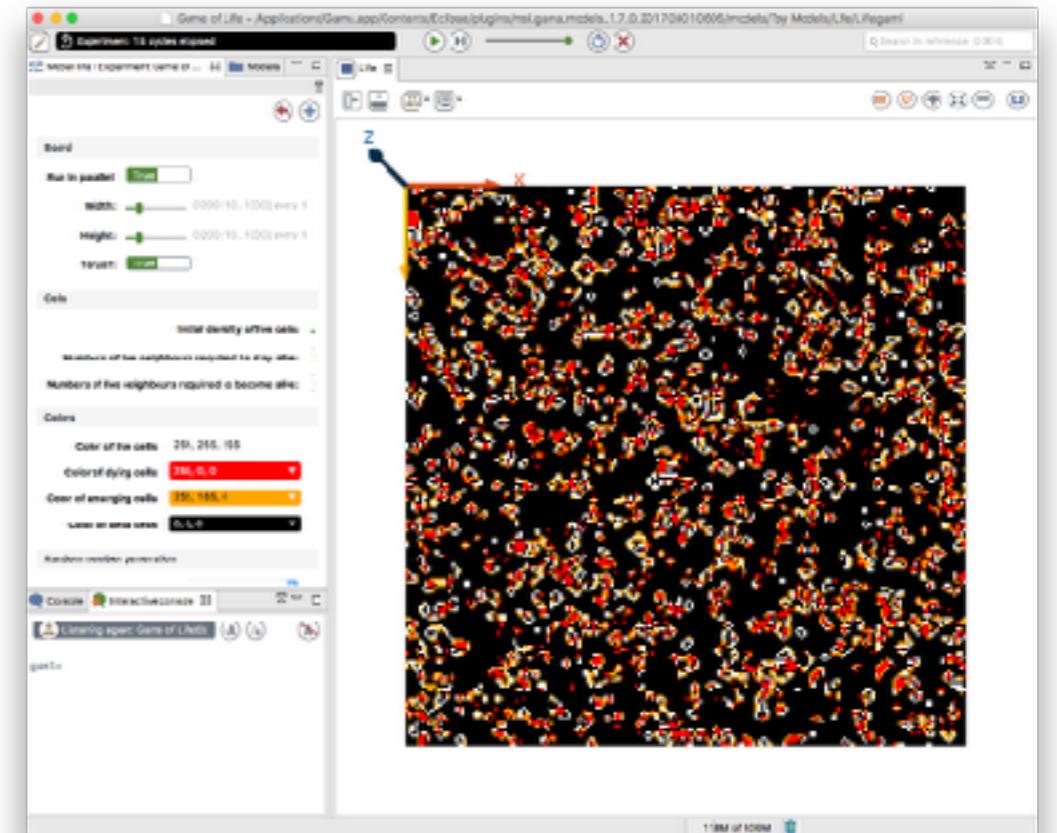


Modelling perspective

Simulation perspective

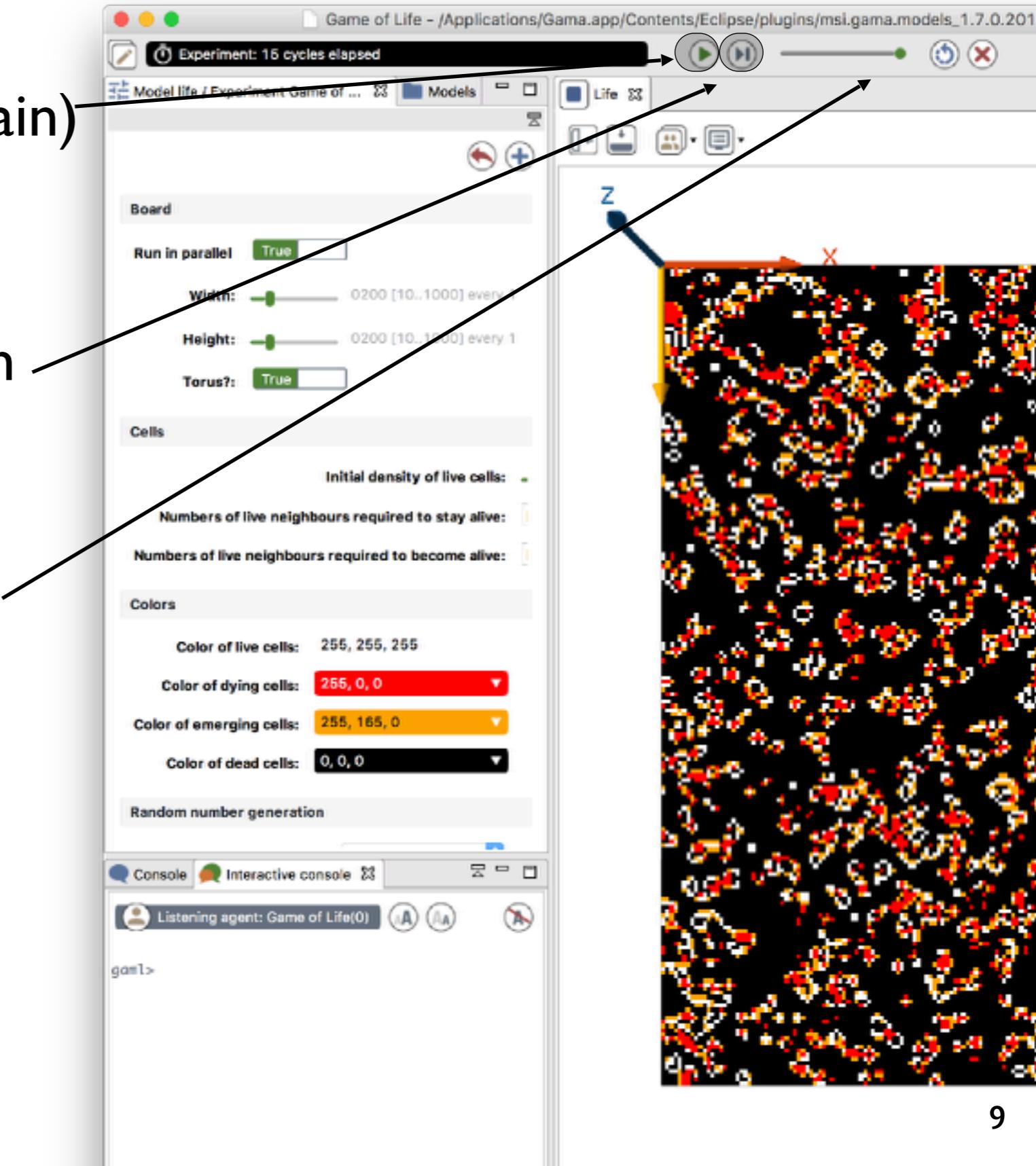
This screenshot shows the Gama IDE in Modelling perspective. The central workspace displays the source code for the "Life.gaml" model, specifically the "Game of Life" section. The code uses Gaml syntax to define the environment, rules for cell survival, and the initial state of the grid. The interface includes a navigation bar at the top, a tree view of available models on the left, and a validation panel at the bottom.

```
1/*  
2 * Name: Life  
3 * Author:  
4 * Description: A model using a cellular automata to represent the Game of Life, the most famous  
5 * example of cellular automata. Each cell will see if the number of living neighbours meets the  
6 * condition to emerge or to live.  
7 * Tag: grid  
8 */  
9 model life  
10 |  
11 |//Declare the world as a torus or not torus environment  
12|global torus: torus_environment {  
13 |int environment_width < 200 min: 10 max: 1000;  
14 |int environment_height < 200 min: 10 max: 1000;  
15 |bool parallel < true;  
16 |}  
17 |//Declare as torus or not  
18 |Bool torus_environment < true;  
19 |//Density  
20 |int density < 25 min: 1 max: 99;  
21 |//Conditions to live  
22 |list<int> living_conditions < [2, 3];  
23 |//Conditions to birth  
24 |list<int> birth_conditions < [3];  
25 |//Color for living cells  
26 |rgb livingcolor < "white";  
27 |//Color for dying cells  
28 |rgb dyingcolor < "red";  
29 |//Color for emerging cells  
30 |rgb emergingcolor < "orange";  
31 |//Color for dead cells  
32 |rgb deadcolor < "black";  
33 |//Shape of the environment  
34 |geometry shape < rectangle(environment_width, environment_height);  
35 |  
36 |//Initialization of the model by writing the description of the model in the console  
37 |init {  
38 |    do description;  
39 |}  
40 |  
41 |//Ask at each life_cell to evolve and update  
42 |reflect generic for {  
43 |    //The computation is made in parallel  
44 |    ask life.cell parallel: parallel {  
45 |        do evolve;  
46 |    }  
47 |}  
48 |//Write the description of the model in the console
```



Exploring the Simulation perspective

- ▶ Start/pause simulation (it will run until pause is clicked again)



- ▶ Step the simulation (it will run one cycle of the simulation)

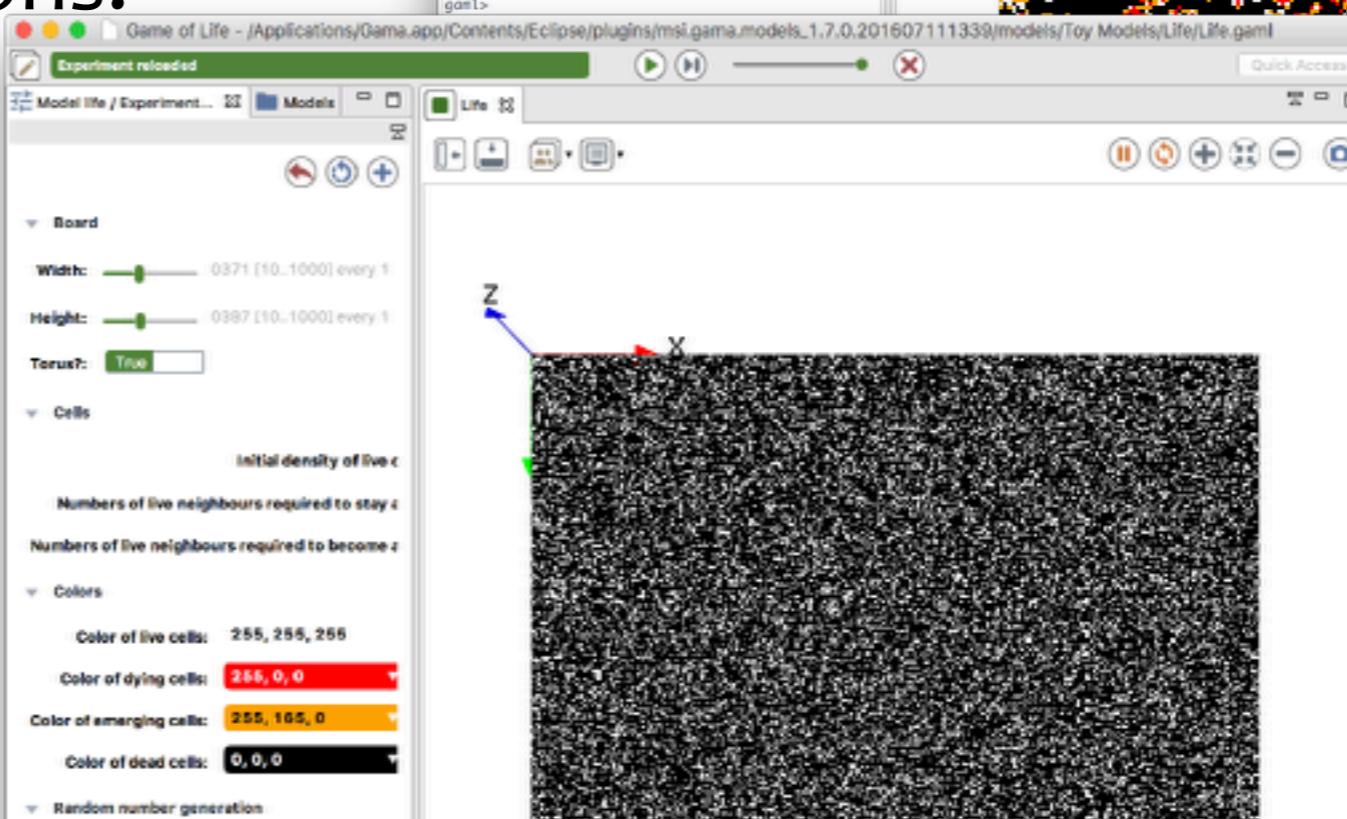
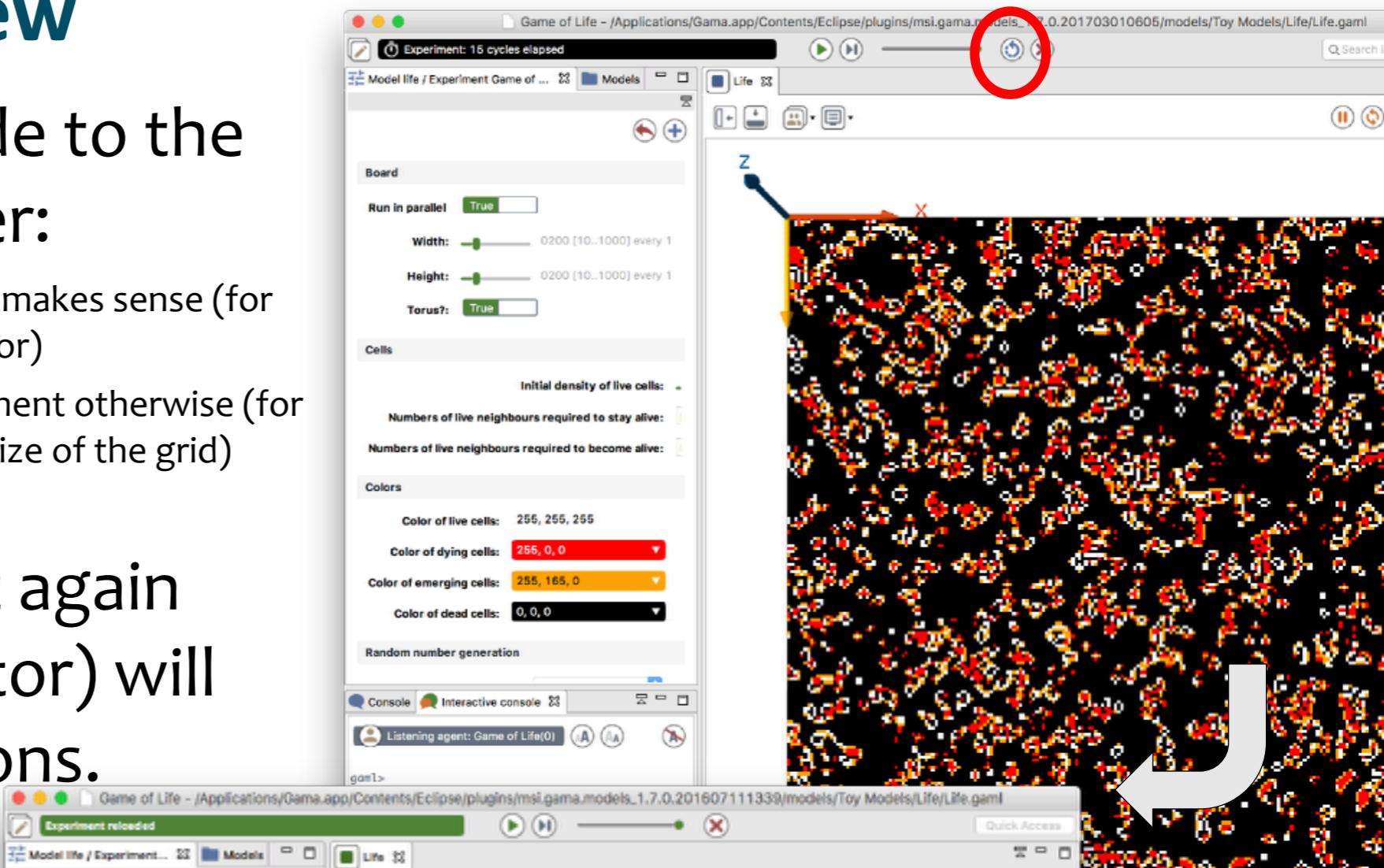
- ▶ Adjust the speed of the simulation

Explore the simulation with parameters modified from Parameters view

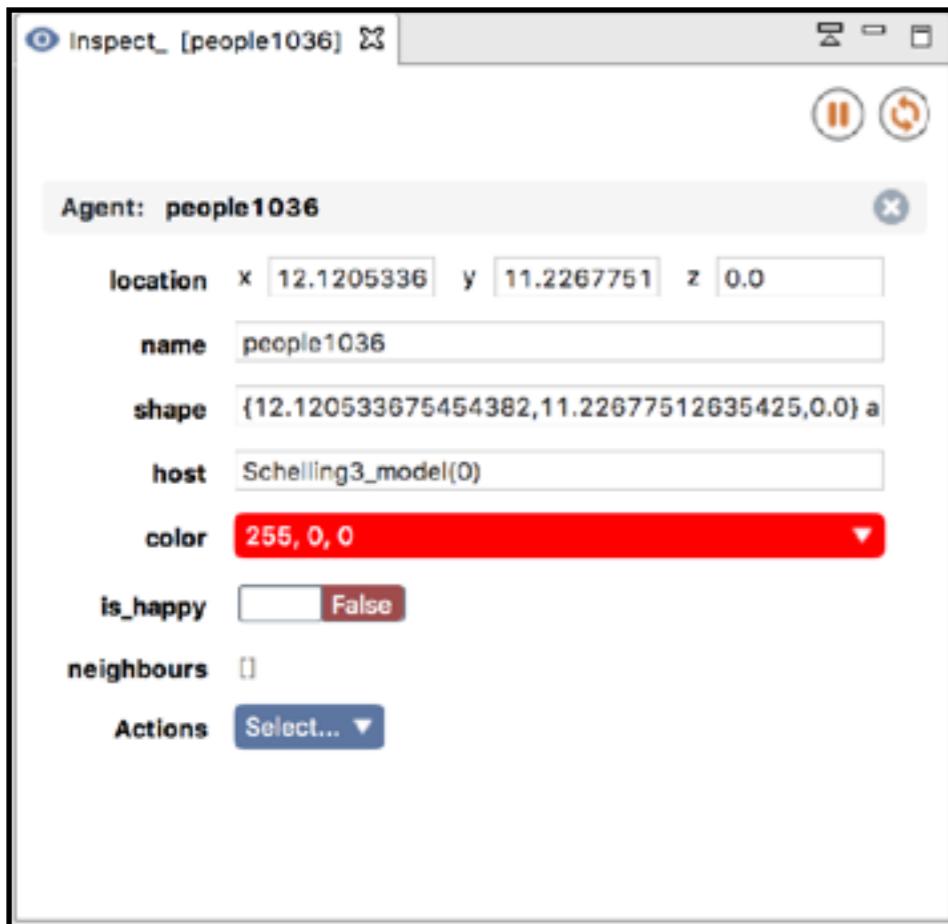
- ▶ The modifications made to the parameters are either:

- ▶ Used for the current simulation when it makes sense (for instance, if the user changes a color)
- ▶ Used when the user reloads the experiment otherwise (for instance, if the user changes the size of the grid)

- ▶ Launching experiment again (from the model editor) will erase the modifications.



GAMA offers 2 views that display information about one or several agents



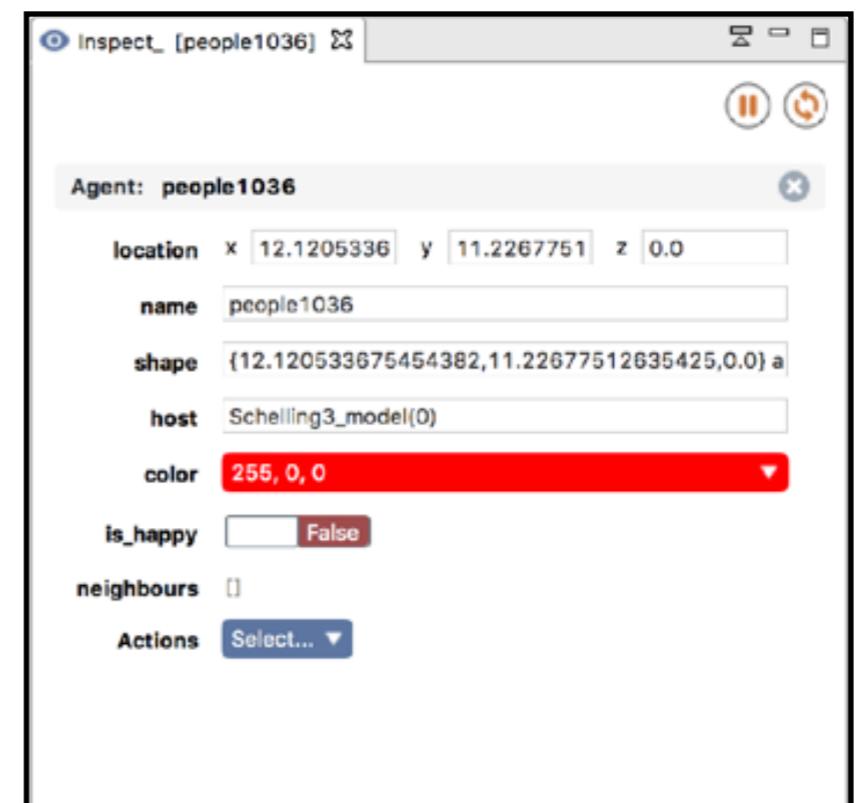
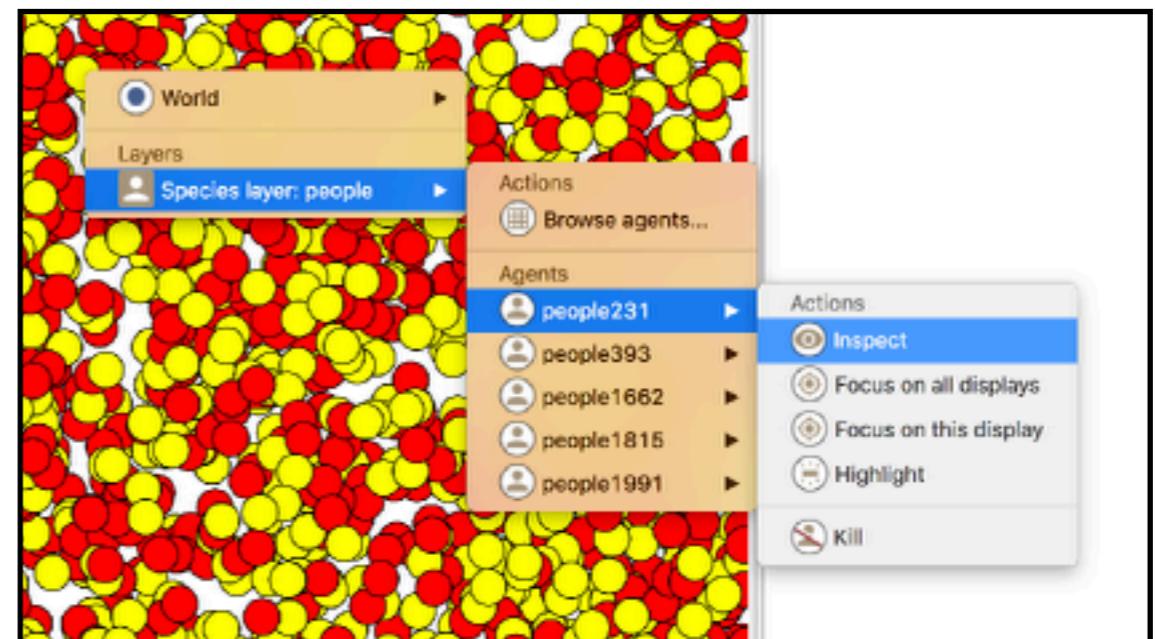
agent inspector

Attributes	#	color	location	name	opinion
agents	0	rob (103. 57. 13...	151.454867956...	SimpleAgent0'	0.4425164039...
color	1	rob (254. 120. 1...	137.713065300...	SimpleAgent1'	0.5587268742...
host	2	rob (143. 215. 1...	137.508191731...	SimpleAgent2'	0.6235161370...
location	3	rob (42. 202. 10...	192.256108104...	SimpleAgent3'	0.5563720528...
members	4	rob (226. 120. 9...	197.290308870...	SimpleAgent4'	0.6658831744...
name	5	rob (182. 7. 216...	151.469727905...	SimpleAgent5'	0.6311607884...
opinion	6	rob (25. 117. 11...	125.560744310...	SimpleAgent6'	0.7791995483...
peers	7	rob (46. 78. 75...	175.709297793...	SimpleAgent7'	0.5587268742...
shape	8	rob (44. 98. 229...	133.3868883396...	SimpleAgent8'	0.2130192286...
agents	9	rob (167. 78. 18...	158.936932527...	SimpleAgent9'	0.5029072021...
color	10	rob (191. 76. 40...	17.0288366905...	SimpleAgent10'	0.5932985490...
host	11	rob (68. 193. 19...	149.410029541...	SimpleAgent11'	0.6982848583...
location	12	rob (58. 76. 107...	110.728018127...	SimpleAgent12'	0.4935022410...
members	13	rob (138. 98. 31...	115.423154126...	SimpleAgent13'	0.8093212845...
name	14	rob (99. 91. 145...	120.736089547...	SimpleAgent14'	0.6311607884...
opinion	15	rob (96. 171. 87...	188.925467574...	SimpleAgent15'	0.4816172839...
peers	16	rob (180. 87. 70...	134.349619171...	SimpleAgent16'	0.4935022410...
shape	17	rob (64. 46. 78...	139.225633940...	SimpleAgent17'	0.5932985490...
agents	18	rob (67. 223. 55...	116.062299931...	SimpleAgent18'	0.6964384083...
color	19	rob (189. 93. 24...	140.014702016...	SimpleAgent19'	0.6602867719...

agent browser

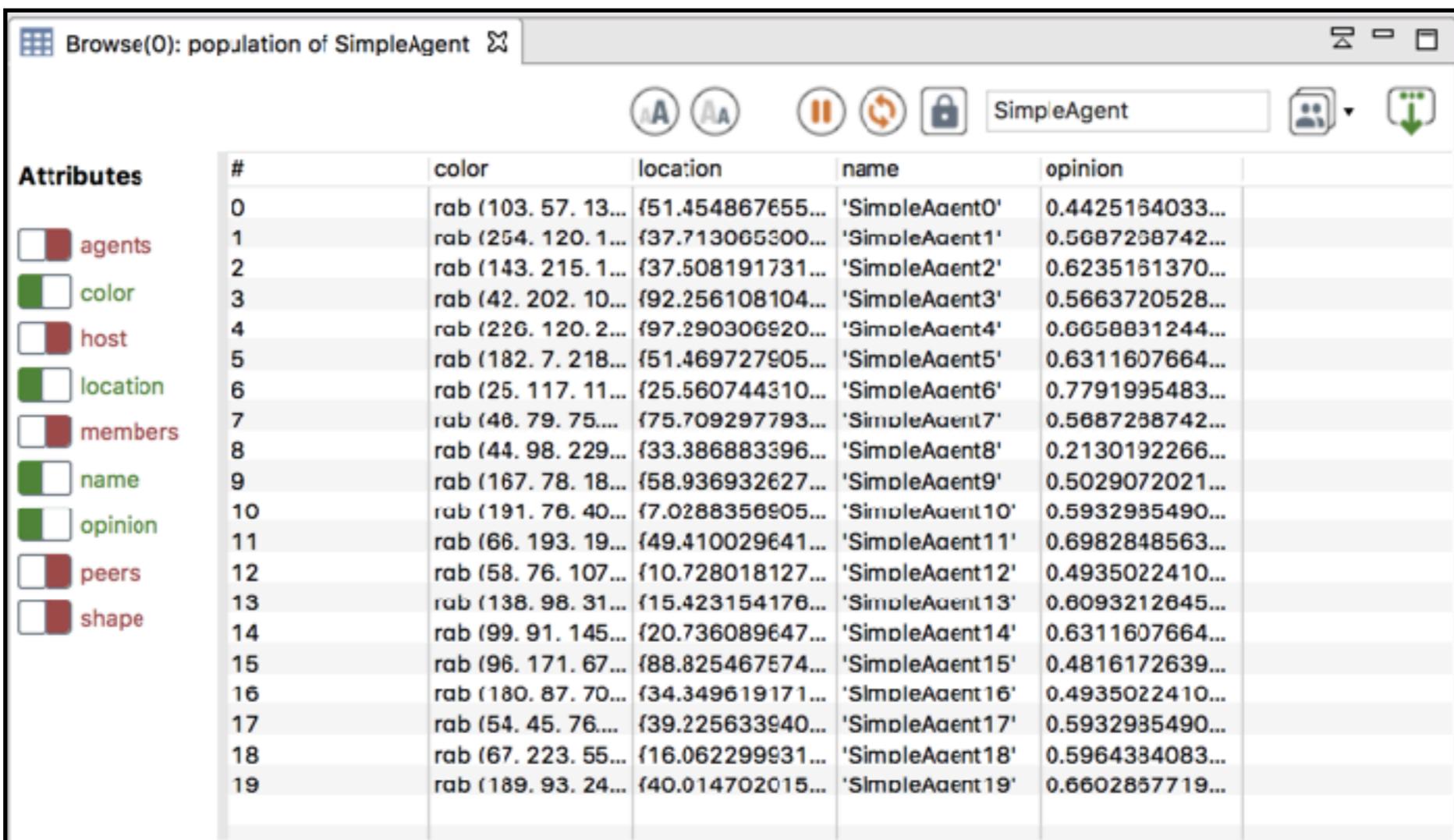
Inspect by right clicking on a agent in a display

- ▶ provides information about one specific agent. It also allows to change the values of its variables during the simulation.
- ▶ It is possible to «highlight» the selected agent.



Inspect informations by agent browser

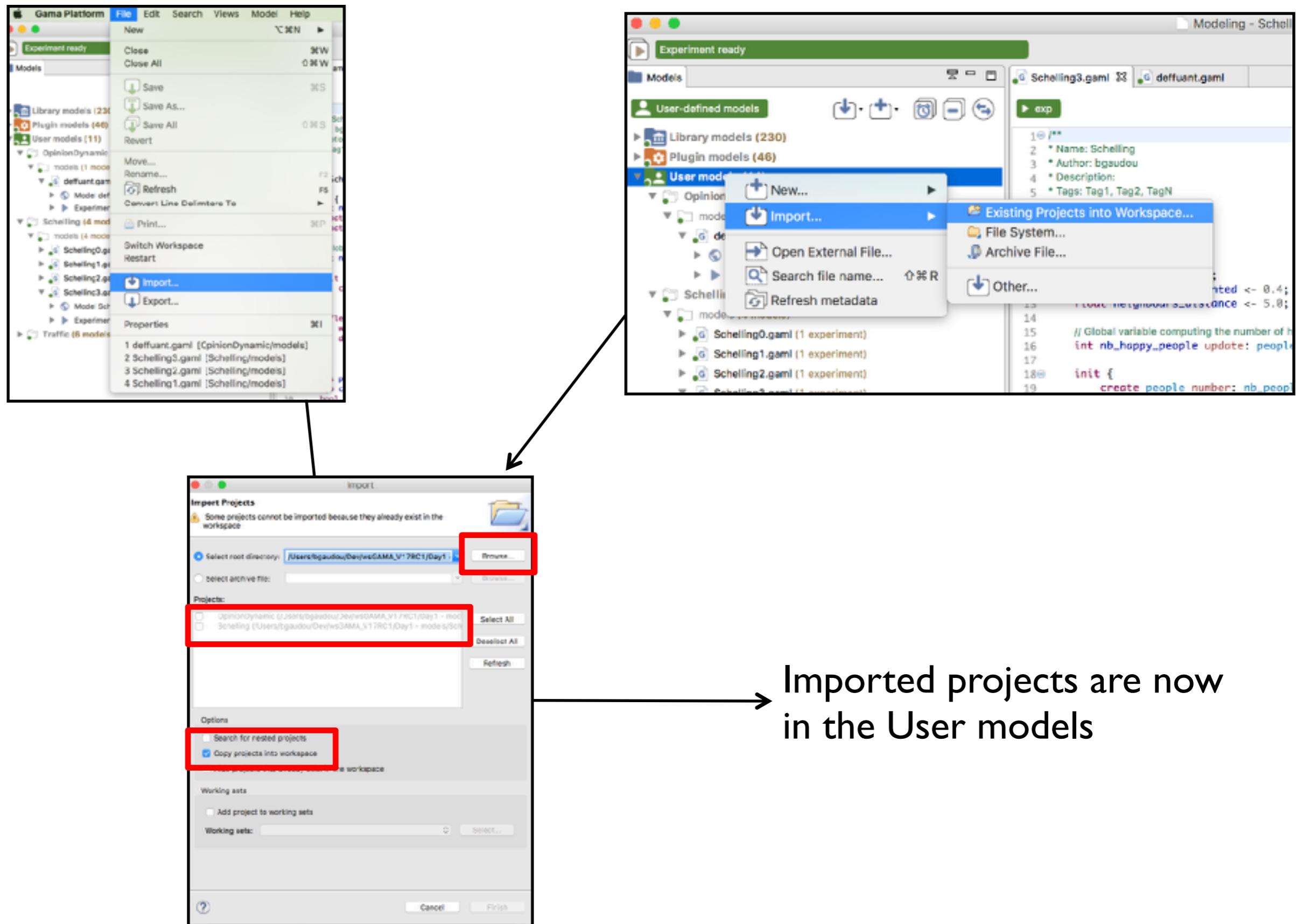
- ▶ The species browser provides informations about all or a selection of agents of a species.
- ▶ The agent browser is available through the **Agents** menu or by right clicking on a by right_clicking on a display



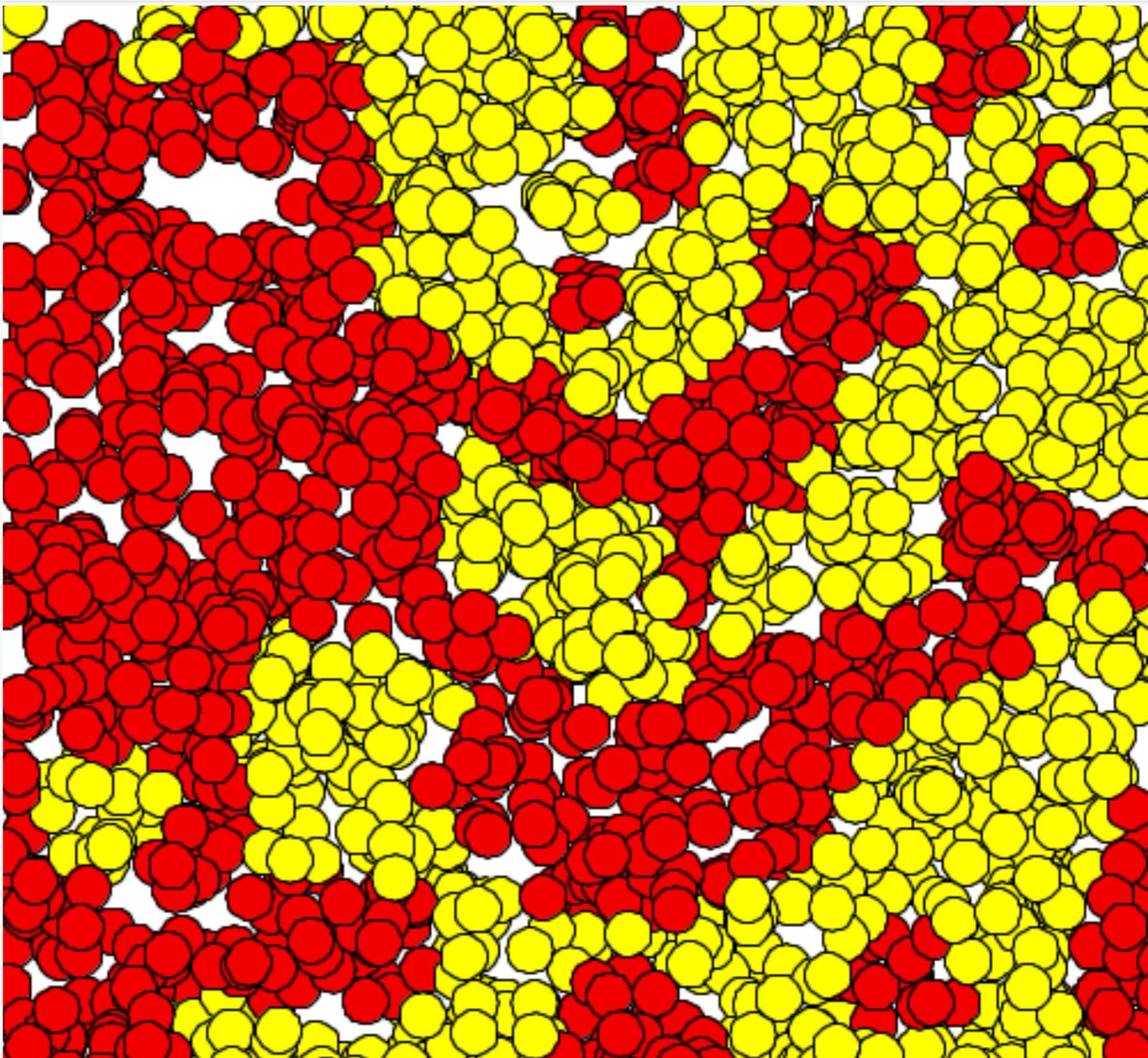
The screenshot shows the JADE Agent Browser interface titled "Browse(0): population of SimpleAgent". The window includes standard operating system controls (minimize, maximize, close) and a toolbar with icons for zoom, search, and file operations. A dropdown menu labeled "SimpleAgent" is open. On the left, there is a sidebar titled "Attributes" with checkboxes for various agent properties: agents (unchecked), color (checked), host (unchecked), location (checked), members (unchecked), name (checked), opinion (checked), peers (unchecked), and shape (unchecked). The main area displays a table with 20 rows of data, each representing a "SimpleAgent" object. The columns are labeled "#", "color", "location", "name", and "opinion". The "color" column contains values like "rab (103. 57. 13..." and "rab (254. 120. 1...". The "location" column contains values like "f51.454867655..." and "f37.713065300...". The "name" column contains values like "'SimpleAaent0'" and "'SimpleAaent1'". The "opinion" column contains numerical values ranging from 0.4425164033... to 0.6602857719... .

Attributes	#	color	location	name	opinion
<input type="checkbox"/> agents	0	rab (103. 57. 13...	f51.454867655...	'SimpleAaent0'	0.4425164033...
<input checked="" type="checkbox"/> color	1	rab (254. 120. 1...	f37.713065300...	'SimpleAaent1'	0.5687288742...
<input type="checkbox"/> host	2	rab (143. 215. 1...	f37.508191731...	'SimpleAaent2'	0.6235151370...
<input checked="" type="checkbox"/> location	3	rab (42. 202. 10...	f92.256108104...	'SimpleAaent3'	0.5663720528...
<input type="checkbox"/> members	4	rab (226. 120. 2...	f97.290306920...	'SimpleAaent4'	0.6658831244...
<input checked="" type="checkbox"/> name	5	rab (182. 7. 218...	f51.469727905...	'SimpleAaent5'	0.6311607664...
<input checked="" type="checkbox"/> opinion	6	rab (25. 117. 11...	f25.560744310...	'SimpleAaent6'	0.7791995483...
<input type="checkbox"/> peers	7	rab (48. 79. 75...	f75.709297793...	'SimpleAaent7'	0.5687288742...
<input type="checkbox"/> shape	8	rab (44. 98. 229...	f33.386883396...	'SimpleAaent8'	0.2130192266...
	9	rab (167. 78. 18...	f58.936932627...	'SimpleAaent9'	0.5029072021...
	10	rub (191. 76. 40...	f7.0288356905...	'SimpleAaent10'	0.5932935490...
	11	rab (66. 193. 19...	f49.410029641...	'SimpleAaent11'	0.6982848563...
	12	rab (58. 76. 107...	f10.728018127...	'SimpleAaent12'	0.4935022410...
	13	rub (138. 98. 31...	f15.423154176...	'SimpleAaent13'	0.6093212645...
	14	rab (99. 91. 145...	f20.736089647...	'SimpleAaent14'	0.6311607664...
	15	rab (96. 171. 67...	f88.825467574...	'SimpleAaent15'	0.4816172639...
	16	rab (180. 87. 70...	f34.349619171...	'SimpleAaent16'	0.4935022410...
	17	rab (54. 45. 76...	f39.225633940...	'SimpleAaent17'	0.5932935490...
	18	rab (67. 223. 55...	f16.062299931...	'SimpleAaent18'	0.5964334083...
	19	rab (189. 93. 24...	f40.014702015...	'SimpleAaent19'	0.6602857719...

Import existing projects into the workspace



Introduction toGAMA through an example: Segregation Model



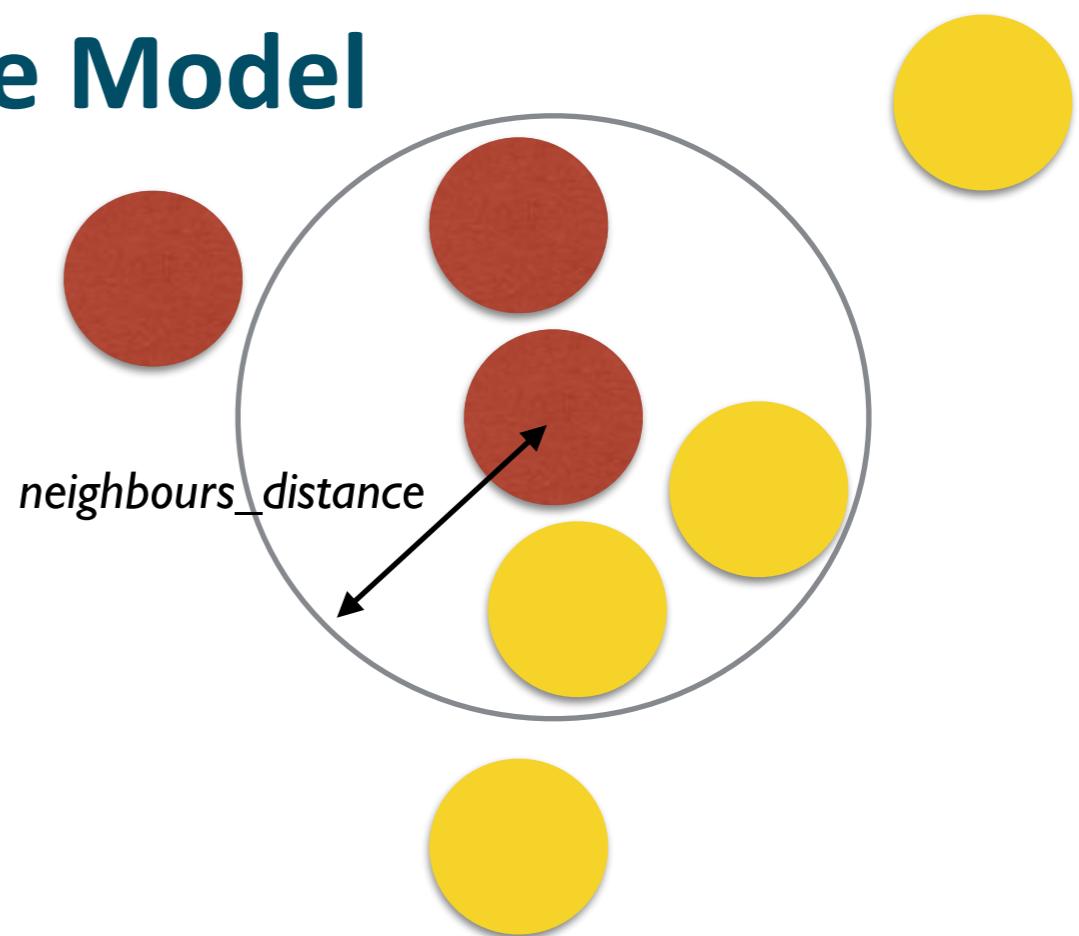
Urban Segregation Model proposed by Schelling

- ❖ In 1969, Schelling introduced a model of segregation in which individuals of two different colours, positioned on a grid (abstract representation of a district), choose where to live based on a preferred percentage of neighbours of the same colour.
- ❖ Using coins on a board, he showed that a small preference for one's neighbours to be of the same colour could lead to total segregation.
- ❖ It is a good example of a generative model, where the emergence of a phenomenon here, segregation) is not directly predictable from the knowledge of individual



Proposed implementation of the Model

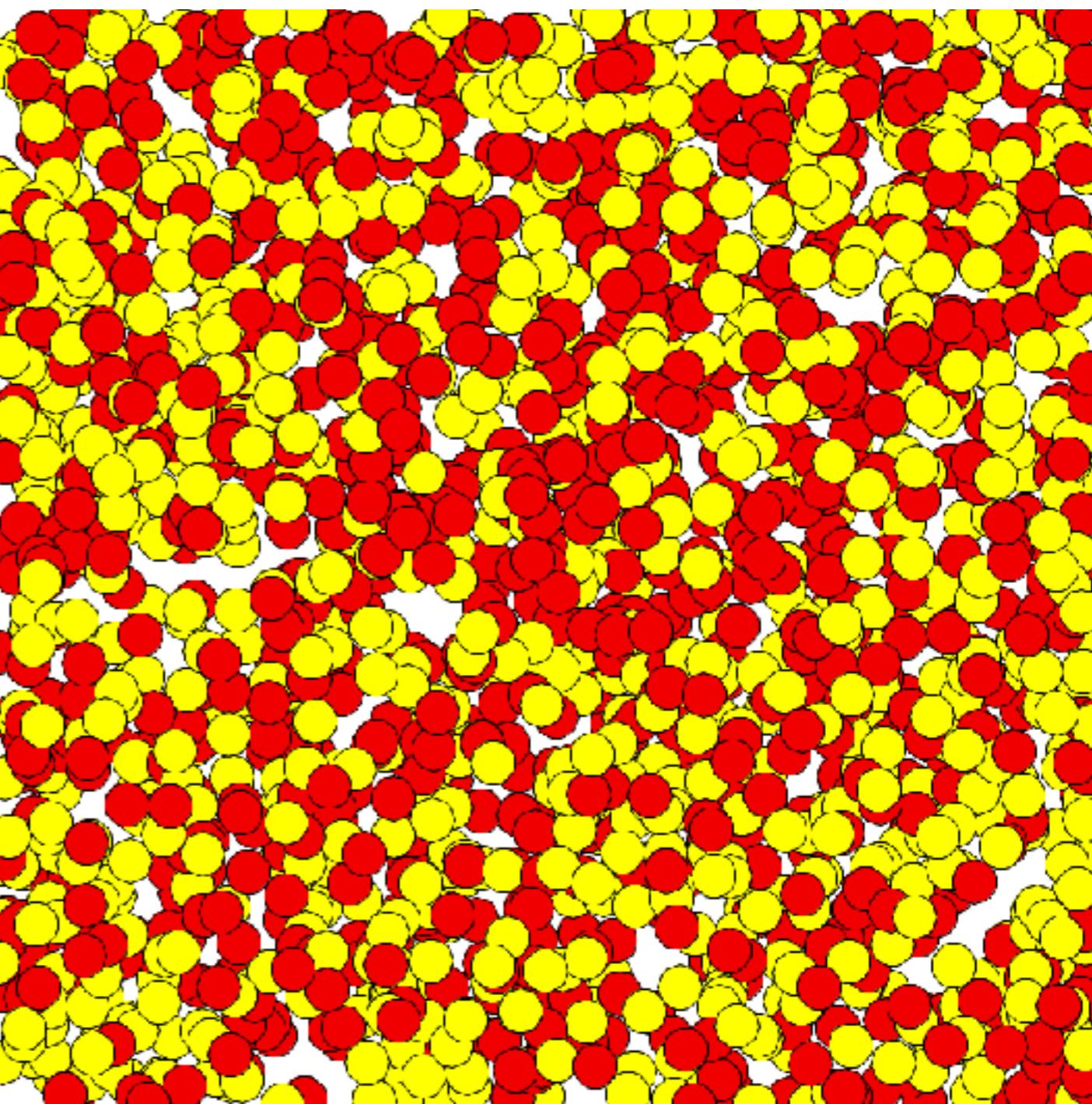
- ❖ **People agents** of 2 different colors (red and yellow) live in a continuous environment
- ❖ At each simulation step, each people agent:
 - computes if it is happy: it is happy if the rate of people agents at a distance *neighbours_distance* of the same color is higher or equals to the threshold *similar_rate_wanted*
 - if it is not happy, it moves to a random location



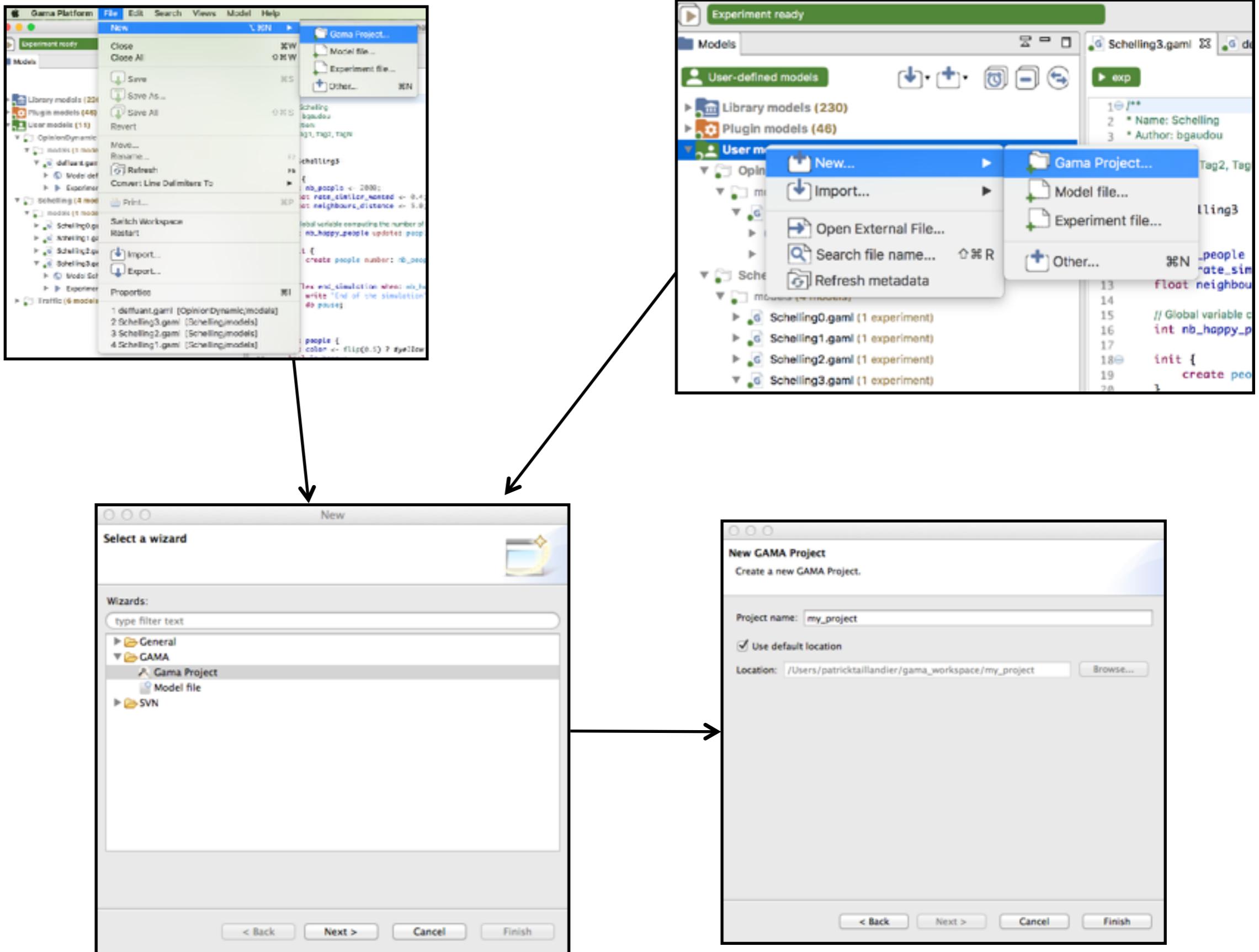
$\text{Similar_rate} = 1/3 = 0.333$
happy if $\text{similar_rate} \geq \text{similar_rate_wanted}$

Step 1: definition and display of the people species

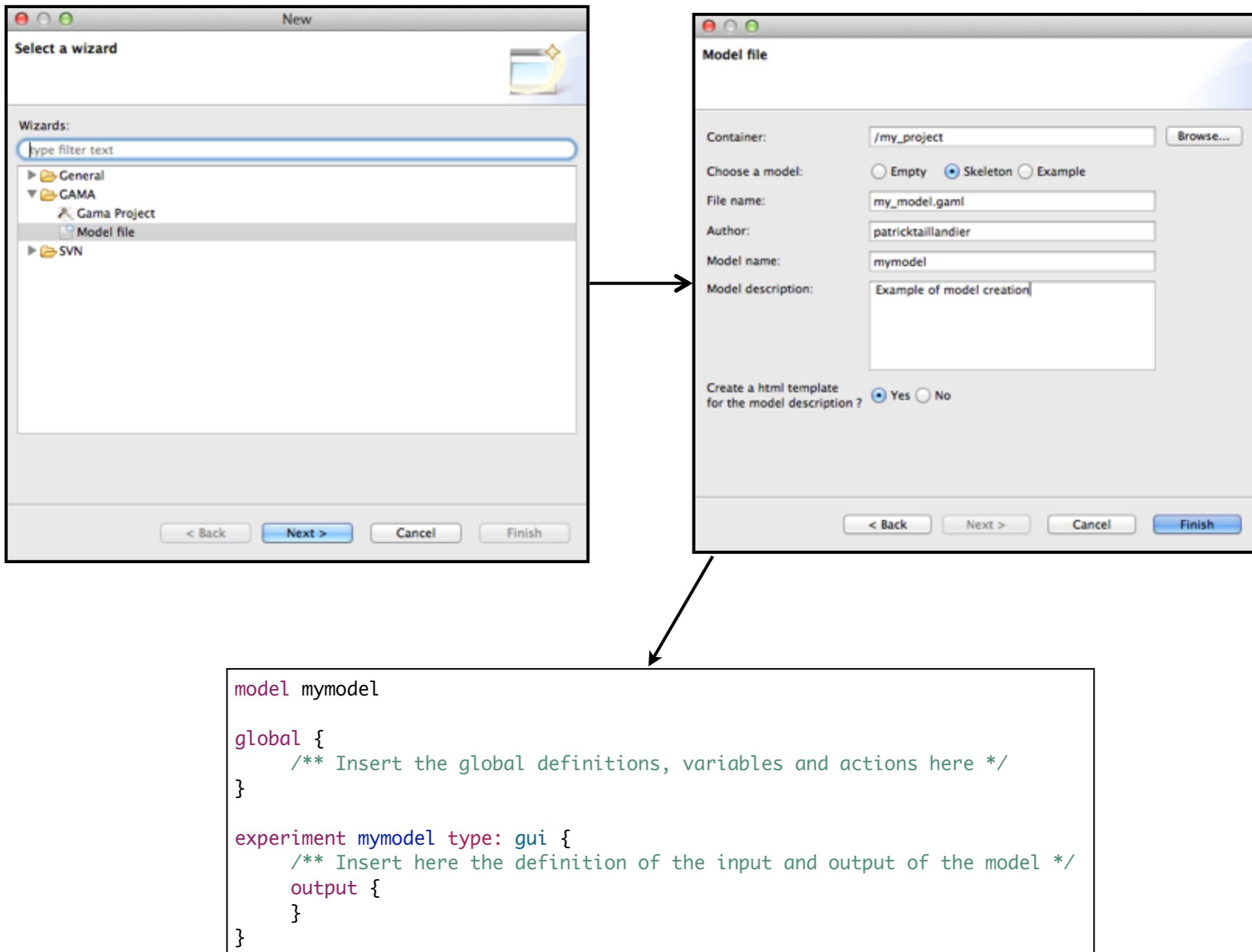
- ❖ Objectives:
 - Definition of the *people* species
 - Creation of 2000 people agents randomly located in the environment
 - Display of the agents



Creation of a new project



Creation of a new model file



Introduction to the GAMA Modelling Language - GAML

- ▶ The role of GAML is to support modellers in writing **models**, which are specifications of **simulations** that can be executed and controlled during **experiments**, themselves specified by experiment plans.
- ▶ Agents in GAML are specified by their **species**, which provide them with a set of **attributes** (*what they know*), **actions** (*what they can do*), **behaviours** (*what they actually do*) and also specifies properties of their **population**, for instance its **topology**
- ▶ **Everything is an agent** in GAML: the model itself (called the *world*), the agents defined in it, the experiments...

Therefore, the structure of a model in GAML is simply a set of *species declaration statements*

- ▶ 3 types of species declaration statements are supported:
- ▶ **Global (unique)**: global attributes, actions, dynamics and initialisation.
- ▶ **Species and Grid**: agent species. Several species statements can be defined in the same model.
- ▶ **Experiment** : simulation execution context, in particular inputs and outputs. Several experiment statements can be defined.

```
model my_model

global {
    /** Insert the global definitions,
     * variables and actions here
     */
}

species my_species{
    /** Insert here the definition of the
     * species of agents
     */
}

experiment my_model type: gui {
    /** Insert here the definition of the
     * input and output of the model
     */
}
```

A statement represents either a declaration or an imperative command

- ▶ It consists in a **keyword**, followed by a list of **facets** (some of them mandatory), ended by “;” or a **block** of statements.
- ▶ A **facet** is a keyword, followed by “：“, and an **expression**. Note that the keyword of the first facet can usually be omitted. If the statement is a declaration, the first facet contains an **identifier**.
- ▶ A **block** is a set of statements enclosed into curly brackets (“{” and “}”)

2 ways to write commentaries (texts that are not just part of the model but here for information purpose):

- //... : for one line. Example : //this is a commentary
- /* ... */ : can be used for several lines. Example : /* this is as well a commentary */

A species declaration statement begins with a special keyword and is composed of several declarative statements

► Syntax:

keyword identifier {

[Attributes declaration]

[Action declaration]

[Behaviours declaration]

[Aspect declaration]

}

```
species my_species{  
    string a_variable;  
    action do_something{...}  
    reflex a_behaviour{...}  
    aspect my_representation{...}  
}
```

where **keyword** is either *global*,
species, *grid* or *experiment*. The
identifier is not used for *global*.

A typical model structure

```
Model new
global{
    // attribute declaration
    // action declaration
    // behaviour declaration
    // aspect declaration
}

species my_species{
    // attribute declaration
    // action declaration
    // behaviour declaration
    // aspect declaration
}

experiment my_experiment type:gui{
    // attribute declaration
    // action declaration
    // behaviour declaration
    // aspect declaration
    // output declaration
}
```

Only in experiments, output declarations allow to define how the outcomes of simulations should be saved or displayed

Segregation model 1: People Species definition

- **To do :** define the species *people*:

- **Solution :**

```
species people {  
}
```

Species can declare attributes (*what their agents know*) by using *declarative statements* beginning with a data type name.

► Syntax:

`data_type identifier;`

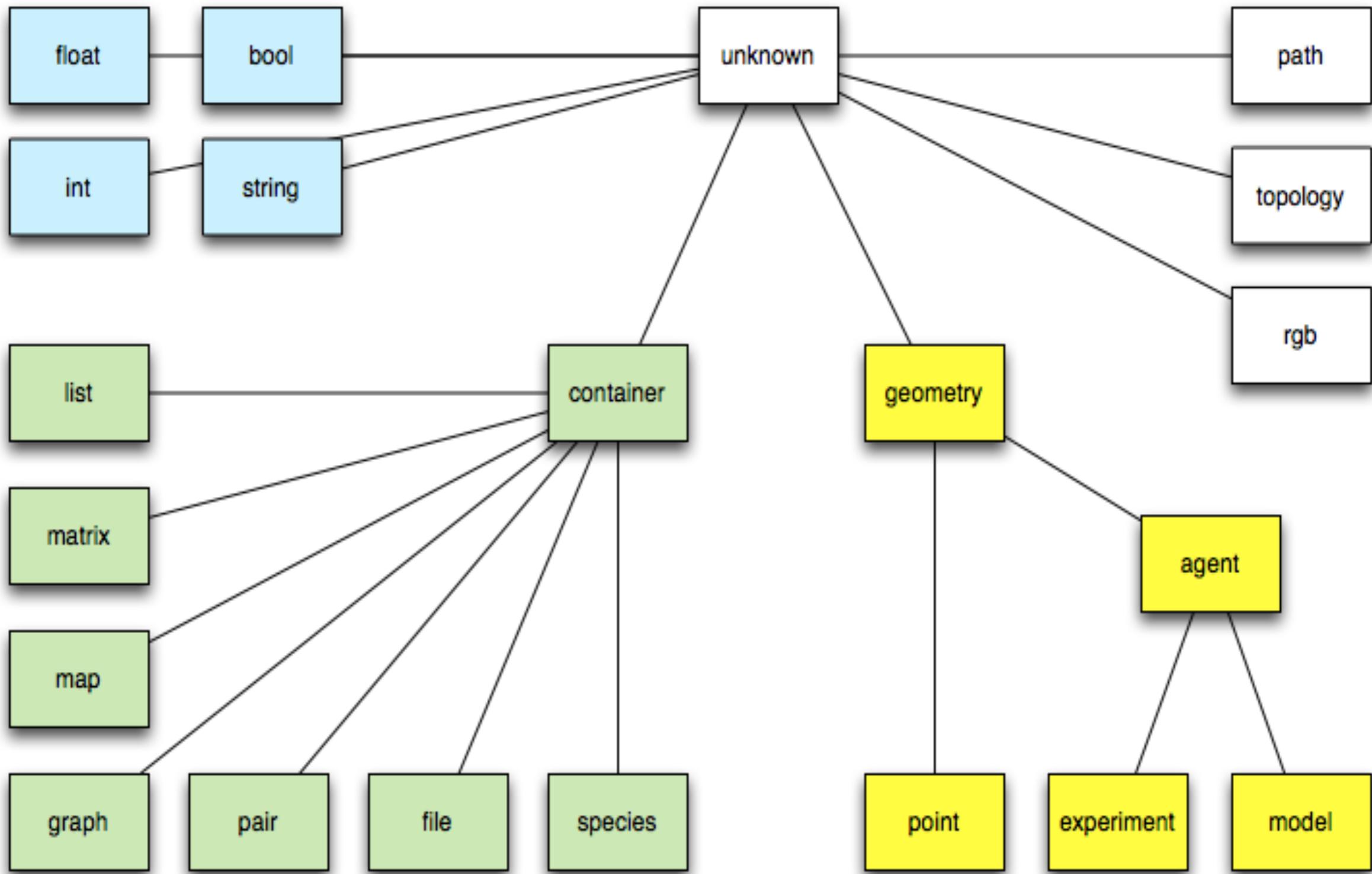
- `data_type` refers to the name of a built-in type or a species declared in the model.
- The **identifier** can be any combination of letters and digits (plus the underscore, "`_`") that does not begin with a digit.
- Optional facets:

- `<-` : initial value
- `update` : value computed at each simulation step
- `function` or `-> + {..}` : value computed each time the variable is called
- `min` : min value
- `max` : max value

All GAMA agents are provided with some built-in attributes :

- ***name*** (*string*)
- ***shape*** (*geometry*)   
- ***location*** (*point*) : centroid of its shape

The hierarchy of data types in GAML is the following:



Examples of data types in GAML

Primitives types:

► Integer:

```
int density <- 25 min: 1 max: 99;
```

► Float

```
float speed_of_agents min: 0.1 <- 2.0 ;
```

► Bool

```
bool state <- (rnd(100)) < density;
```

► String

```
string message <- "This is a message";
```

Complex types:

- List: a composite datatype holding an ordered collection of values

```
list a_list_of_numbers<- [0,1,2,3];
```

- Map: a composite datatype holding an ordered collection of pairs

```
map pair_of_key_and_value<- [{"key": ::5}];
```

- Matrix: represents either a two-dimension array (matrix) or a one-dimension array (vector), holding any type of data (including other matrices).

```
matrix mat_2_3 <- ([  
    ["c11", "c12", "c13"],  
    ["c21", "c22", "c23"]  
]);
```

Segregation model 1: People Species attribute definition

- **To do :** define two attributes for the species *people*:
 - **color** : type: *rgb* (colour), init value (<-): *red* with a probability of *0.5*, *yellow* otherwise
 - **is_happy**: type: *bool*, init value: *false*
- **Solution :**

```
species people {  
    rgb color <- flip(0.5) ? #red : #yellow;  
    bool is_happy <- false;  
}
```

Define a colour attribute of type *rgb* (color) of which the initial value has a probability of 0.5 to be red or yellow

Note : the **flip(proba)** operator to test a probability. i.e.
returns true with a probability of *proba*

The symbol **#** allows
also to define a colour

Species definition - *aspect*

- ❖ An *aspect* represent A possible display for a species of agents : ***aspect aspect_name {...}***
- ❖ In an *aspect* block, it is possible to display (as layers) using the ***draw*** statement:
 - a geometry/shape: for example, the agent shape
 - a image: for example, an icon
 - a text

Segregation model 1: People Species circle aspect

- **To do :** define a new aspect called *circle*:
 - ➡ draw a circle of radius 2m with the colour *color*

- **Solution :**

```
species people {  
    //attribute definition  
  
    aspect circle {  
        draw circle(2) border: #black color: color;  
    }  
}
```

This aspect allow to display each people agent as circle of radius 2m using the *color* attribute to define its colour

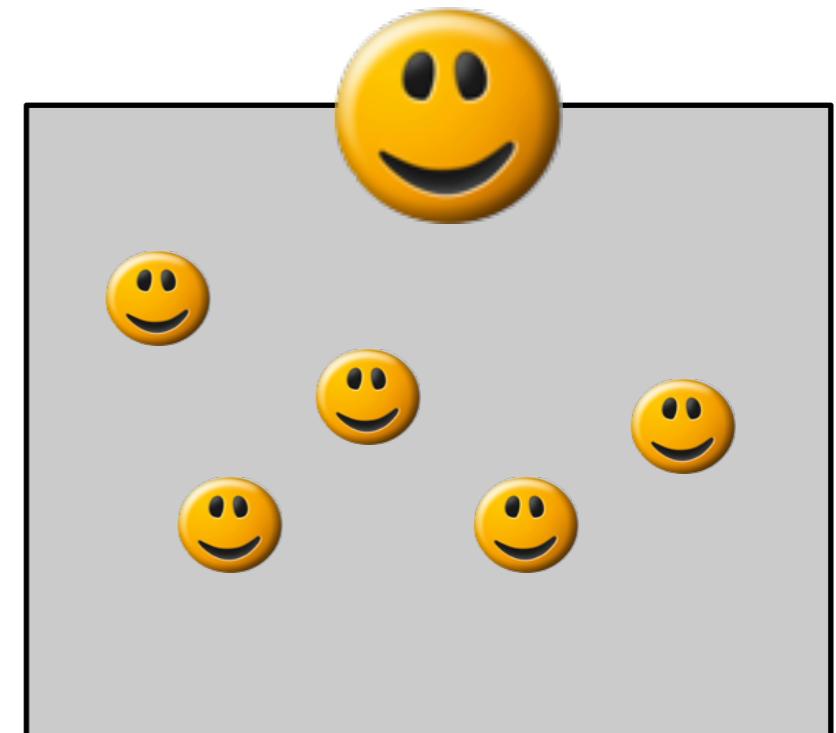
Now, we have finished to define the people species that has several attributes and that can be displayed as a circle.

Next step: creation of people agents !

global block

❖ Global block

- Define a specific agent (called *world*)
- Represent all that is global to the model:
 attributes, actions, reflexes....
- Initialise the simulation (*init block*): when
 the experiment button is pushed, the
 world agent is created and then
 activates its *init* block
- The geometry of the *world* agent (*shape*)
 defines the size of the environment in
 which all the agents are localised. By
 default the shape of the *world* agent
 is a square of 100m size
- Define the nature of the environment:
 torus or not (by default, not torus)



Segregation model 1: Global attribute definition

- **To do :** define three attributes in the global section :
 - **nb_people** : type: *int* (integer), init value (<-): 2000
 - **rate_similar_wanted**: type: *float*, init value: 0.4
 - **neighbours_distance**: type: *float*, init value: 5.0
- **Solution :**

```
global {  
    int nb_people <- 2000;  
    float rate_similar_wanted <- 0.4;  
    float neighbours_distance <- 5.0;  
}
```

Init block

- ▶ For each species, an init block can be defined
- ▶ Allows to execute a sequence of statements at the creation of the agents
- ▶ Activated only once when the agent is created, after the initialisation of its variables, and before it executes any reflex

▶ Syntax:

```
init{  
    <<statements>>  
}
```

- ▶ Only one instance of init per species

```
global {  
    .....  
    //Only executed when world agent is created  
    init {  
        write "Executing initialisation";  
    }  
}
```

Agent creation

❖ Creation of agents : use of the statement: **create species_name +**

- *number* : number of agent to create (*int*, by default, 1)
- *from* : GIS or Raster file (*string or file*)
- *with* : allows to give initial values to the agent variables
- *returns*: list of created agents

Note: By default, agents are **randomly** placed in the environment (except when the facet **from:** + S/G is used)



Note: The *create* statement can be used in all init/actions/reflex of the model, not only in the global section

Segregation model 1: Global init

- **To do :** define a global init block that :
 - ➡ `create nb_people people agents`

- **Solution :**

```
global {  
    //Attribute definition  
  
    init {  
        create people number: nb_people;  
    }  
}
```

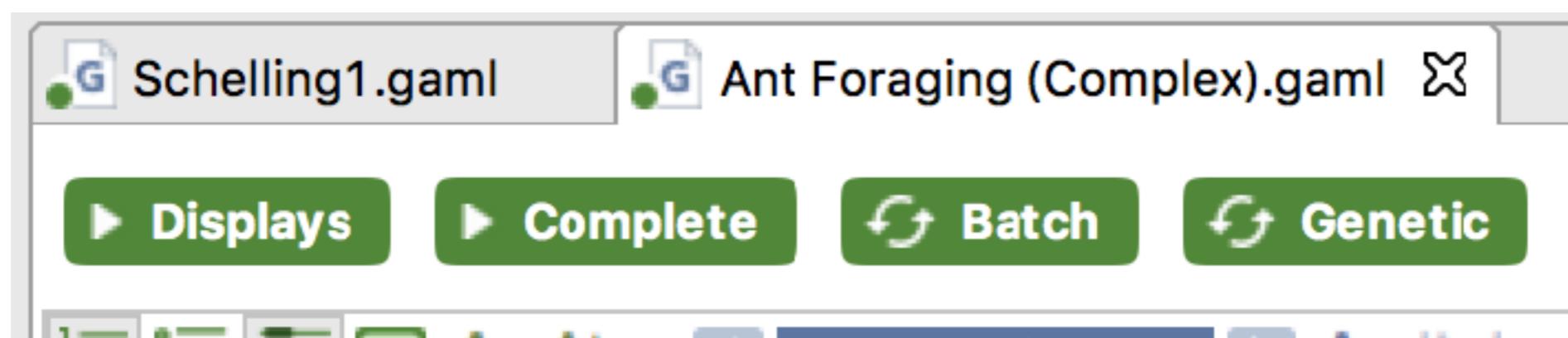
Now, we have finished to define global attributes and
to create people agents.

Next step: add a display to view the people agents

!

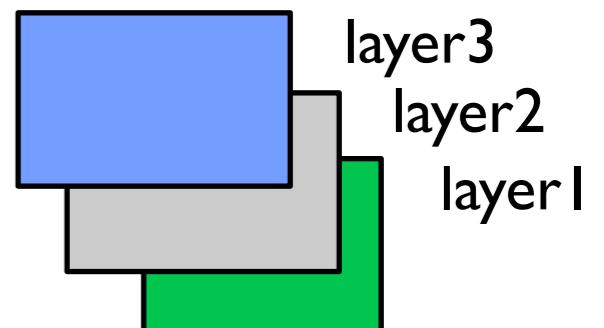
experiment block

- ❖ An experiment is a specific agent that allows define the execution context of simulations, in particular the parameters and outputs of the simulation
- ❖ Several experiment can be defined (one experiment bouton is created per experiment)
- ❖ Define by : **experiment xp_name type: gui/batch {...}**
 - *gui* : one simulation with graphical interface.
 - *batch* : experiment plan: set of simulations without graphical interface



***experiment* block: output definition**

- ❖ The *output* block has to be defined in an *experiment* block
- ❖ It allows to define displays:
 - A refreshing rate can be defined: facet **refresh: nb (int)**
 - Each *display* can contain different displays:
 - ▶ list of agents :
agents layer_name value: agents aspect: my_aspect;
 - ▶ Agent species (all the agents of the species) :
species my_species aspect: my_aspect
 - ▶ Grids: optimised display of grids:
grid grid_name lines: my_color;
 - ▶ Images:
image layer_name file: image_file;
 - ▶ Charts: see later



Segregation model 1: Output definition

- **To do :** define a display called *map* that displays :
 - ➡ The *people* agents with their aspect *circle*

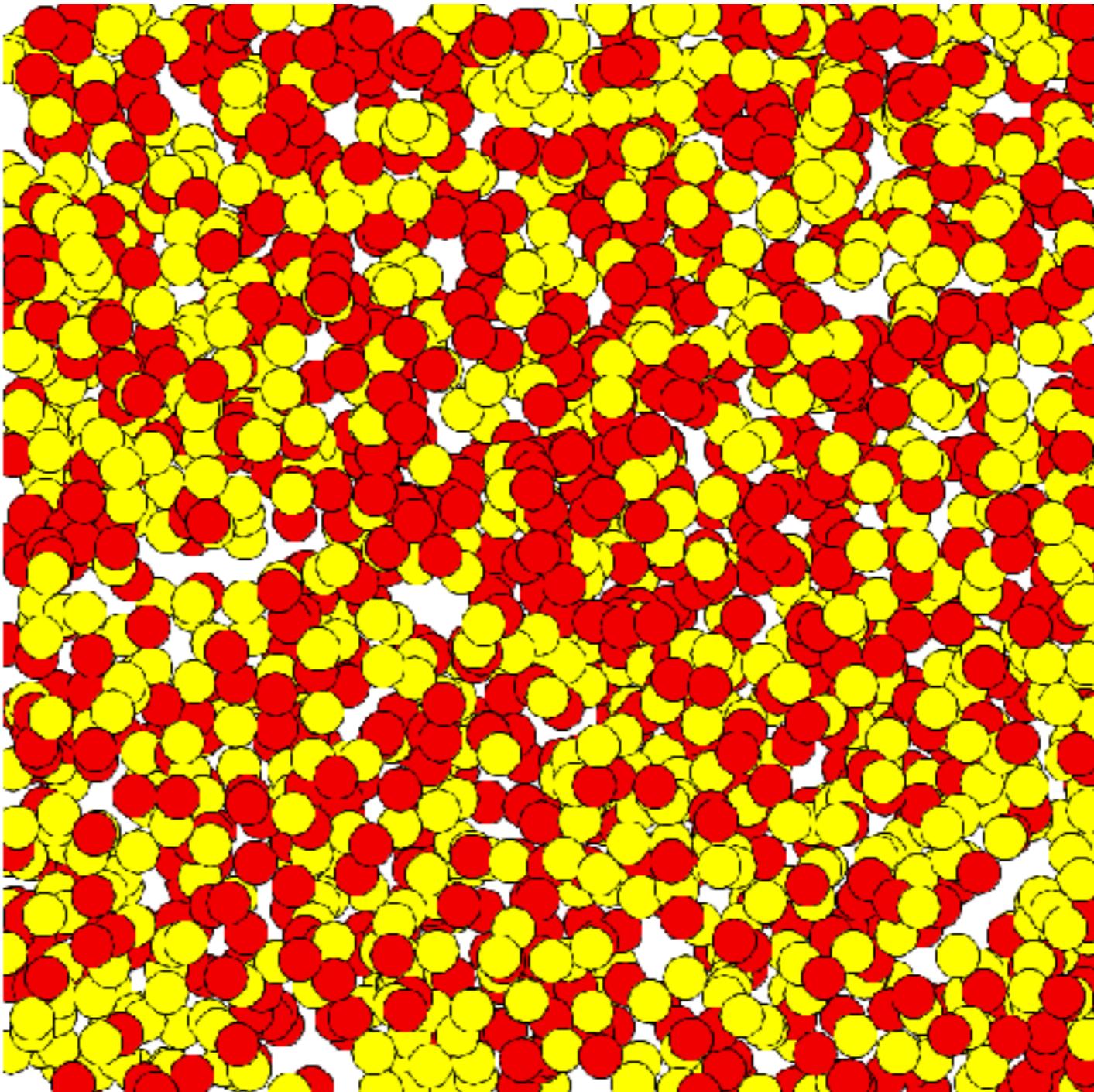
- **Solution :**

```
experiment main_xp type: gui {  
    output {  
        display map {  
            species people aspect: circle;  
        }  
    }  
}
```

Now, we have finished to define a display to view the people agents.

Next step: Test the model !

End of step 1

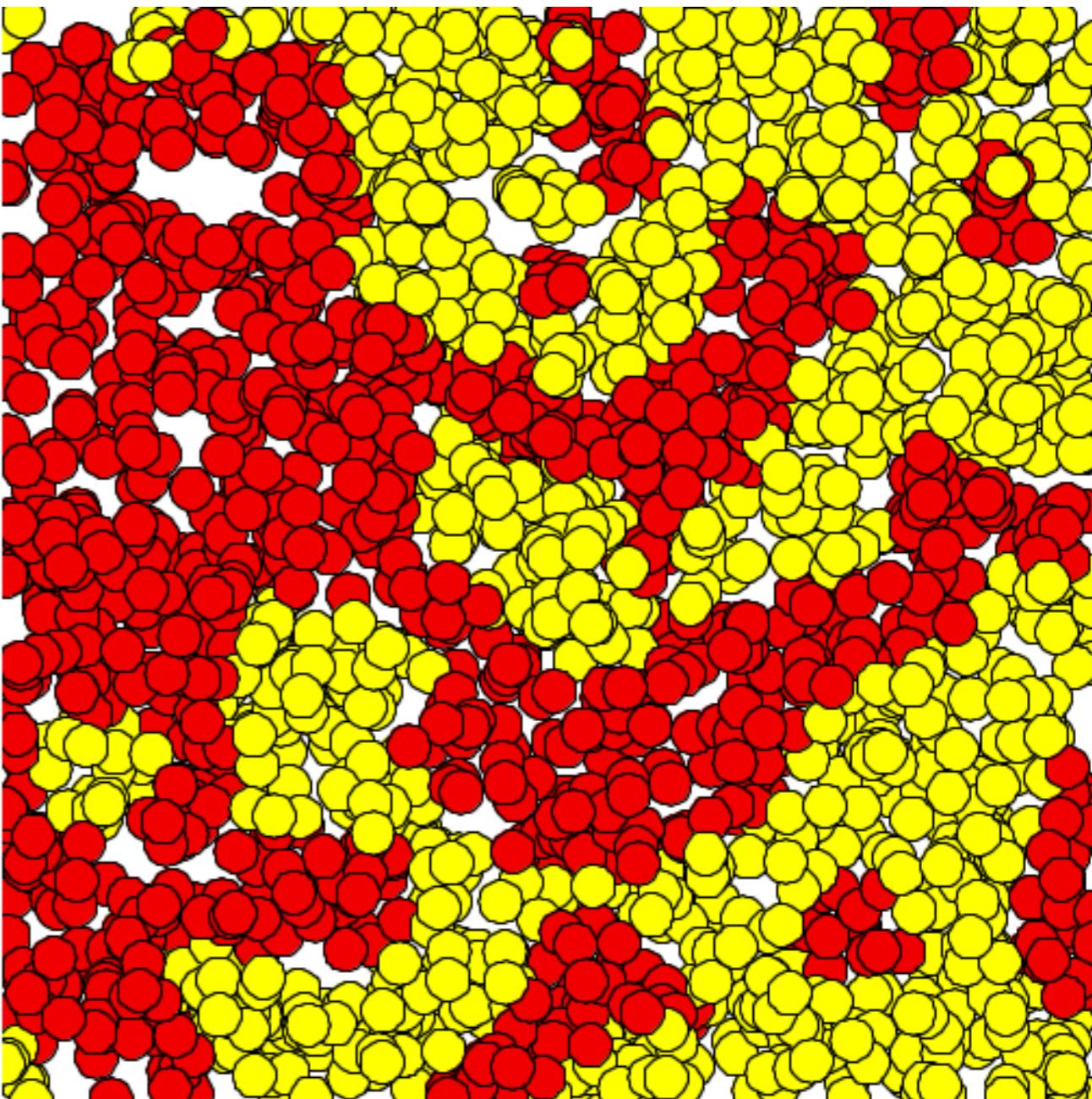


Now it's time to define the dynamic of the models !

Step 2: definition of the people agent dynamic

❖ Objectives:

- Definition of the neighbours attribute for the people agents
- Definition of a computing neighbours similarity behaviour for the people agents
- Definition of a moving behaviour for the people agents



Species are provided with a simple behavioural structure, based on reflexes (*what they actually do*)

- ▶ A reflex is a sequence of statements that can be executed, at each time step, by the agent.

```
reflex name when: condition{
```

```
[statements]
```

```
}
```

- ▶ If no facet **when** are defined, it will be executed every time step.
- ▶ If there is one, it is executed only if the boolean expression evaluates to true.
- ▶ It is a convenient way to specify the behaviour of the agents.

Note: The init block is a specific reflex that is activated only once at the creation of the agent

The scheduler of GAMA

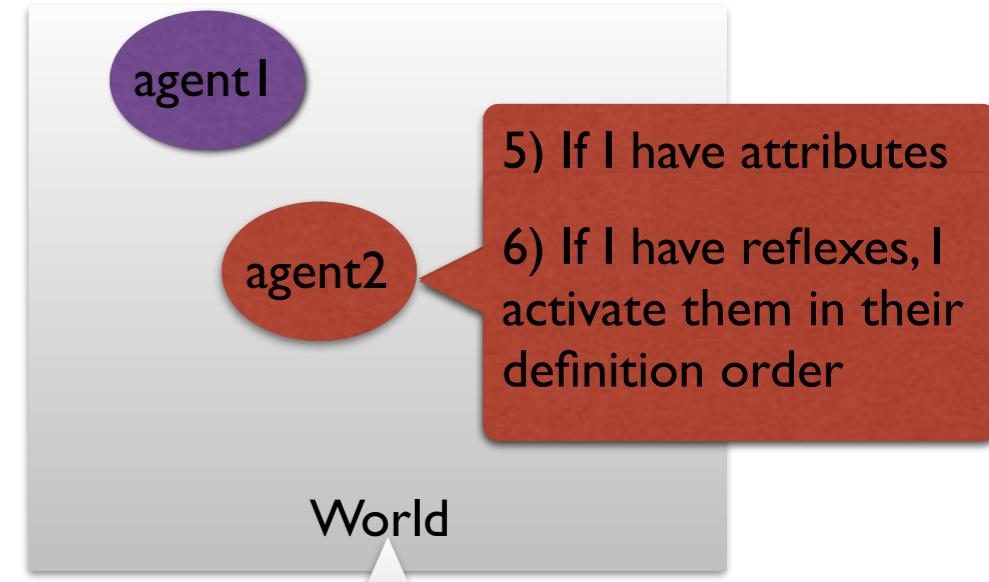
► The basic scheduler of GAMA works as follows:

- GAMA activates the world agent (global) then all the other agents according to their order of creation
- When an agent is executed, first its update its attributes (facet **update** of the attributes), then it activates its reflexes in their definition order

► Of course the Scheduler can be easily tuned through the GAML language:

- modification of the order of activation of the agents (than can be dynamic)
- Fine activation of the agents using actions (e.g.: agent1 executes a first action, then agent2 executes an action, then agent1 executes again another action....)

3) If I have attributes with a
4) If I have reflexes, I activate them in their definition order



2) If I have reflexes, I activate them in their definition order

Note: GAMA offers some specific control architectures (finite state machine, task-oriented architectures...) that can be added to species

Segregation model 2: Neighbours definition for people species

- To do : define an attribute for the people species called *neighbours*:

→ Type: list of *people* agents; update at each simulation step with the *people* agents that are at a distance lower or equal to *neighbours_distance*

- Solution :

```
species people {
    //other attributes
    list<people> neighbours update: people at_distance neighbours_distance;
    //aspect definition
}
```

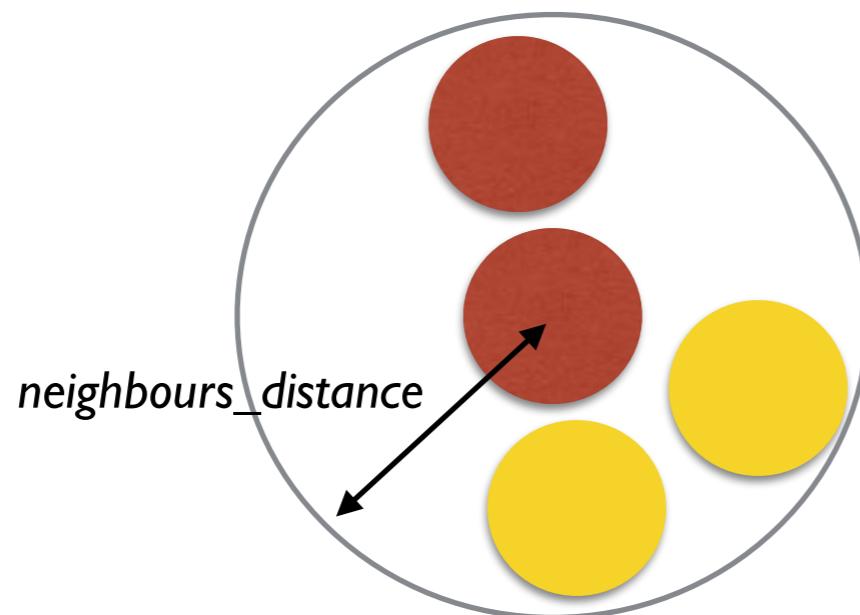


Note 1: the « *a_container at_distance distance* » operator returns all the agents from *a_container* (that can be a species or a list of agents) that are at distance lower or equal to *distance* to the current agent (the one using the operator).

Note 2: the « <> » symbol allows to specify the type of elements contained in a list

Segregation model 2: Computing Similarity reflex for the people species

- To do : define a reflex called *computing_similarity* for the people species:
 - if the neighbours is empty, set the *rate_similar* to 1.0
 - Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the *rate_similar* to the number of similar neighbours divided by the number of neighbours



$\text{Similar_rate} = 1/3 = 0.333$
happy if $\text{similar_rate} \geq \text{similar_rate_wanted}$

What we need here:

- test statement (if-else)
- modifying the value of an attribute
- local variables (to make the code more readable)
- list operators (is empty, nb of elements...)

Introduction to some classic GAML Statements

► Test statement

```
if (my_condition1){...}  
else if (my_condition2){...}  
...  
else {...}
```

```
if (my_money <= 50000) {  
    write "I am poor";  
} else if (my_money > 50000 and my_money < 200000)  
{  
    write "I am ok";  
} else {  
    write "I am rich";  
}
```

► Attribute value assignment statement

```
my_attribute <- new_value;
```

```
my_money <- 100000;
```

► Definition of local variables: variables that just exist inside a block. These variables are deleted from the computer memory at the end of the block

```
type my_local_variable <- init_value;
```

```
int nb_month <- 12;
```

Segregation model 2: Compute Similarity reflex for the people species

- To do : define a reflex called *compute_similarity* for the people species:
 - if the neighbours is empty, set the *rate_similar* to 1.0
 - Otherwise, compute the number of neighbours, then the number of neighbours with the same colour as the agent, then set the *rate_similar* to the number of similar neighbours divided by the number of neighbours

Solution :

```
species people {
    //attributes
    reflex computing_similarity {
        float rate_similar <- 0.0;
        if empty(neighbours) {
            rate_similar <- 1.0;
        } else {
            int nb_neighbours <- length(neighbours);
            int nb_neighbours_sim <- neighbours count (each.color = color);
            rate_similar <- nb_neighbours_sim /nb_neighbours ;
        }
        is_happy <- rate_similar >= rate_similar_wanted;
    }
    //aspect definition
}
```

Note: for list operators, the keyword *each* represents each element of the list

returns true if the list *neighbours* is empty

nb of elements in the list *neighbours*

count the number of elements (from the list *neighbours* for which the colour is equal to the current agent colour

is_happy is set to true if *rate_similar* (computed just before) is higher or equal than the global variable *rate_similar_wanted*

Segregation model 2: Moving reflex for the people species

- **To do :** define a reflex called *moving* for the people species:
 - activated only if the agent is not happy (*not is_happy*)
 - The agent changes its location by an random location in the world
- **Solution :**

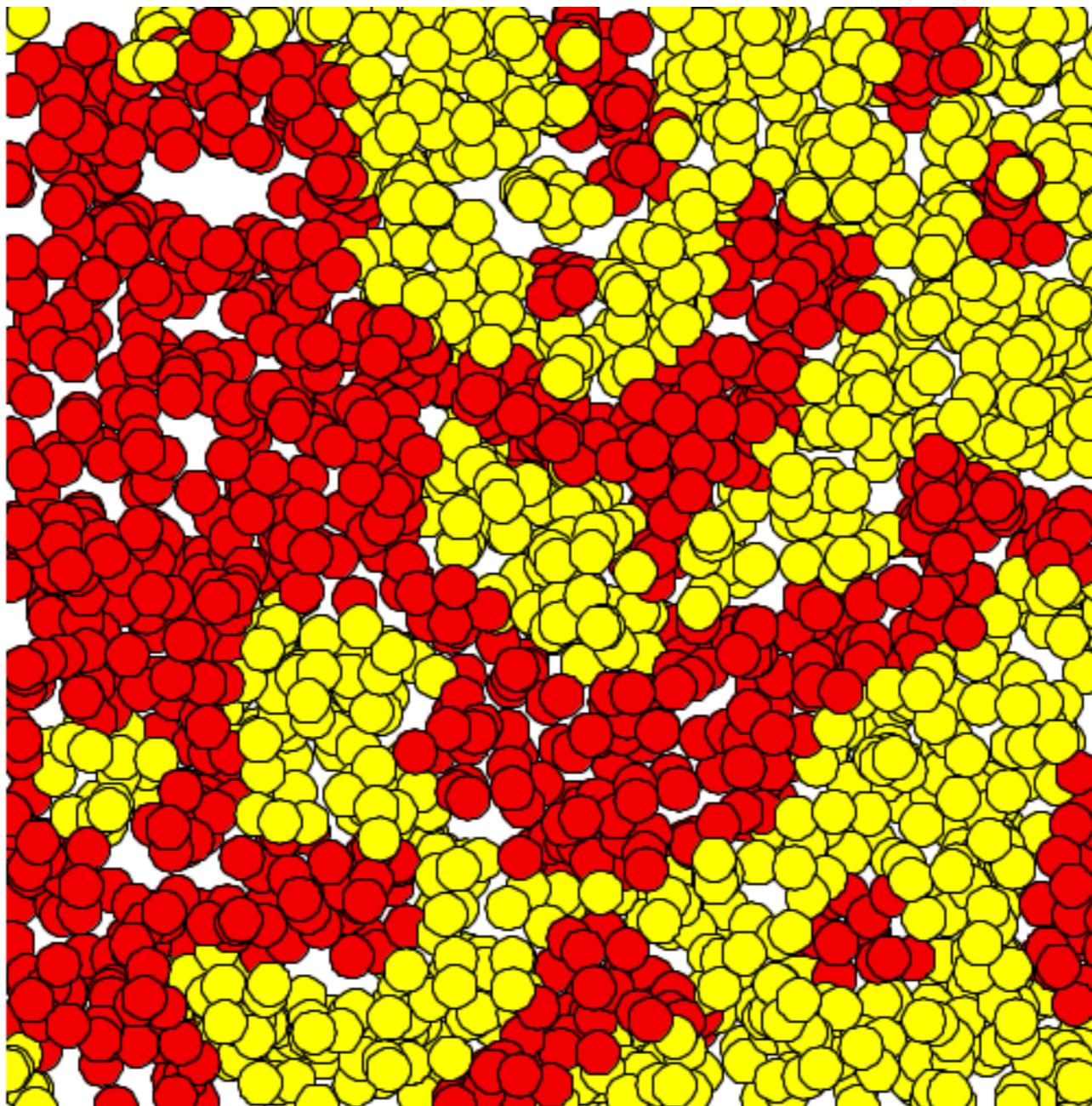
```
species people {
    //attributes
    reflex computing_similarity {
        ...
    }
    reflex moving when: not is_happy {
        location <- any_location_in(world);
    }
    //aspect definition
}
```

the *any_location_in* operator returns a random location (point) inside a geometry or the geometry of an agent.

Note 1: all GAMA agents are provided with a **location** attribute that defined the coordinate of the centroid of its shape. When the location is modified, the shape of the agent is translated to the new location and when the shape is modified, the location is re-computed.

Note2: the world (global) agent can be accessed by the « **world** » keyword

End of step 2

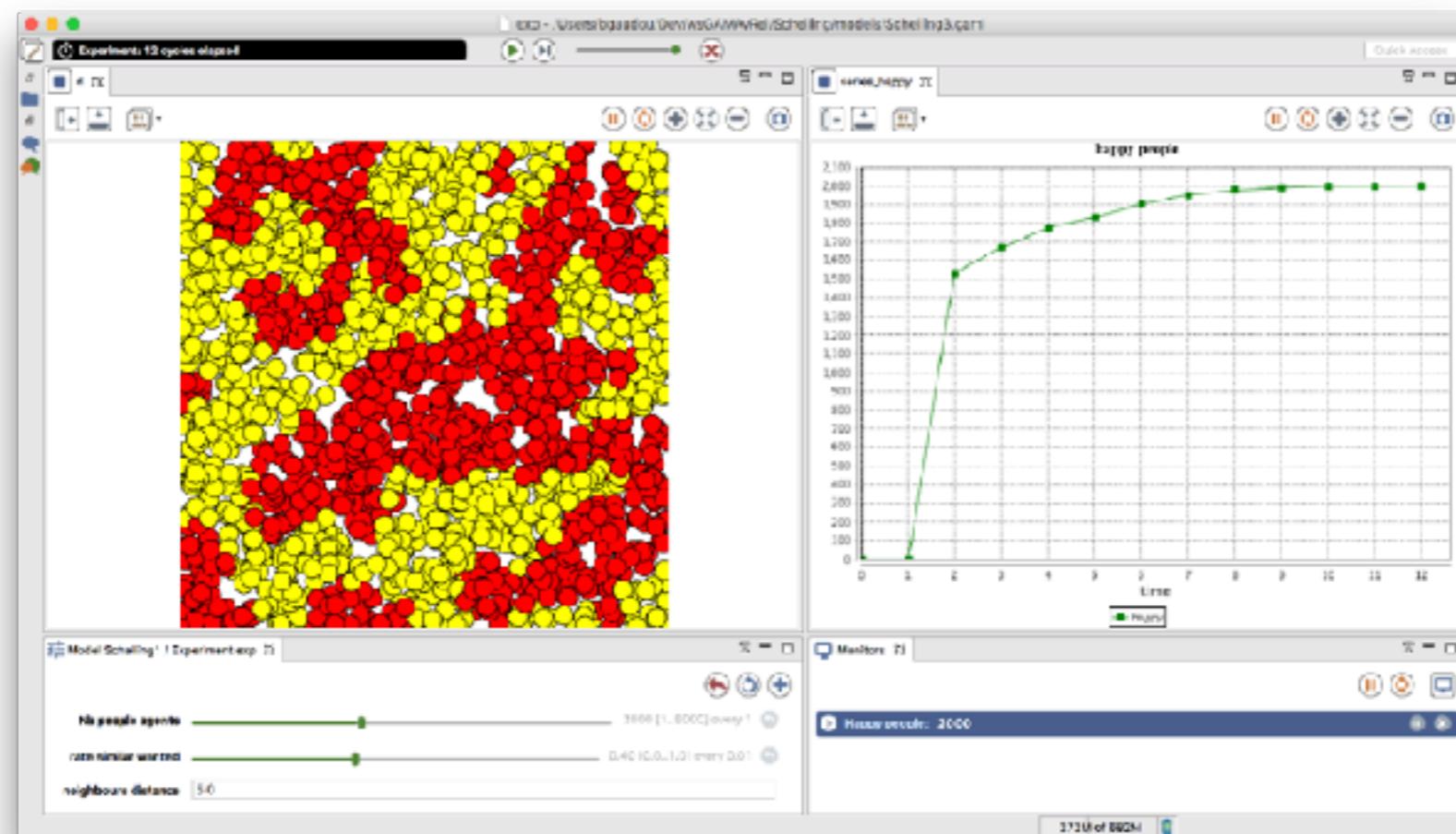


Now it's time to define some new parameters and new outputs for the model !

Step 3: definition of new parameters and outputs

❖ Objectives:

- Definition of a new global variable to compute the number of happy people at each simulation step
- Definition of an ending condition (when all people are happy)
- Definition of parameters
- Definition of a new monitor to follow the number of happy people
- Definition of a chart to follow the evolution of the number of happy people



Segregation model 3: computation of the number of happy people

- **To do :** define a global attribute called *nb_happy_people*:
 - ➡ Type: *int*; update at each simulation step with the number of *people* agents that are happy
- **Solution :**

```
global {  
    // other attributes  
    int nb_happy_people <- 0 update: people count each.is_happy ;  
    ...  
}
```

We are now able to know at every step the number of happy people

Next step: pause the simulation when all the people agents are happy!

For that we need to use the pause action of the world agent!

Segregation model 3: Simulation Ending condition

- **To do :** define a global reflex called *end_simulation*:
 - ➔ is activated only when the number of happy people is equal to the number of people
 - ➔ call the « pause » action of the world agents that pauses the simulation
- **Solution :**

```
global {  
    //attributes and init  
    -  
    reflex end_simulation when: nb_happy_people = nb_people {  
        do pause;  
    }  
}
```

An action in GAML is a capability available to the agents of a species (*what they can do*)

- ▶ It is a block of **statements** that can be used and reused whenever needed. An action can accept arguments.
- ▶ An action can return a result (statement **return**).

```
return_type action_name (var_type arg_name,...) {  
    [statements]  
    [return value;]  
}
```

```
action action_simple {  
    write "action simple";  
}
```

«write» allow to display a message in console view (best for debugging models)

Action that returns a value

- ▶ Some actions are directly available (built-in action, i.e. primitive) for all agents (e.g. die action) or to specific agents (pause action of the world agents)

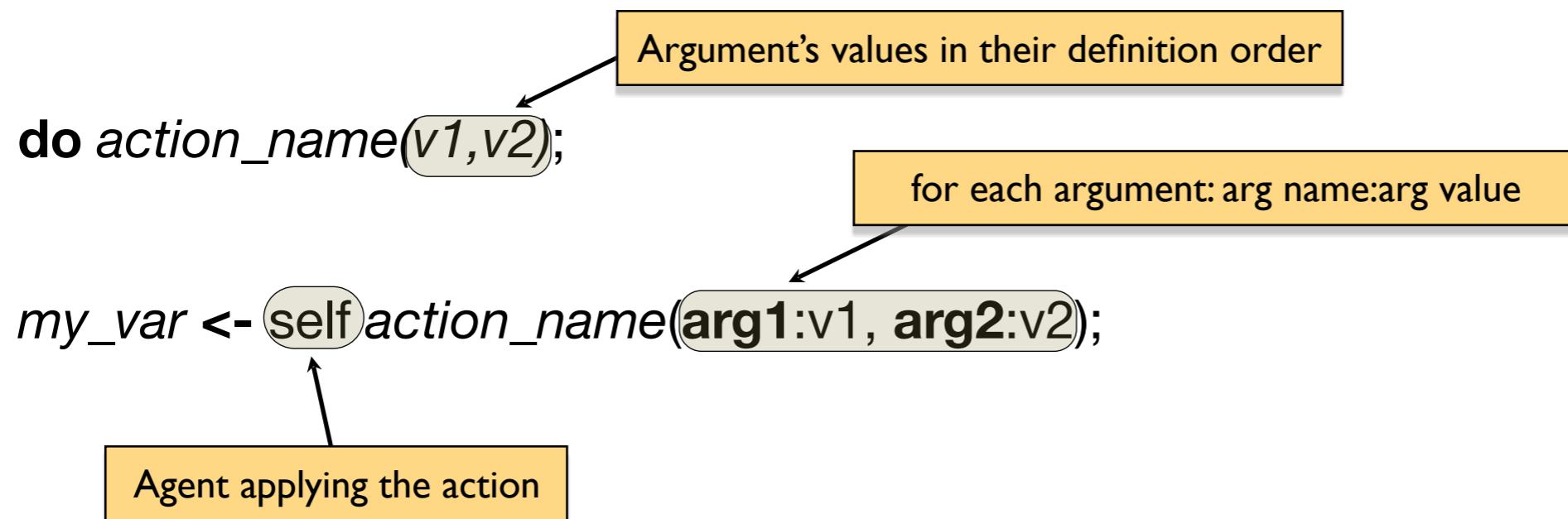
Type of the return value

```
int sum (int a <- 100, int b) {  
    return a + b;  
}
```

Definition 2 arguments : the first one named «a», type integer and default value is «100»; the second named «b», type integer

return a value, and finish the action

Different ways to call an action in GAML



Examples:

```
do action_simple;
```

```
int d <- self add(10,100);
```

```
int d <- self add(b:100);
```

Segregation model 3: Simulation Ending condition

- **To do :** define a global reflex called *end_simulation*:
 - is activated only when the number of happy people is equal to the number of people
 - call the « pause » action of the world agents that pauses the simulation
- **Solution :**

```
global {  
    //attributes and init  
    -  
    reflex end_simulation when: nb_happy_people = nb_people {  
        do pause;  
    }  
}
```

The simulation is now pause when all the agent are happy

Next step: definition of parameters!

experiment block: parameter definition

❖ Parameter :

```
parameter legend var: var_name category: my_cat;
```

- Allow to give to the user the possibility to define the value of a global attribute
- *legend*: string to display
- *var_name*: reference to a global attribute
- *category*: string (use to better organise the parameters) - optional

Example



```
experiment main_experiment type:gui{  
    parameter "Nb people agents" var: nb_people min: 1 max: 5000 step: 1;  
}
```

It is possible to define here the
min and max values of a
parameter as well as its
discretisation step

Segregation model 3: Parameter definition

- **To do :** define 3 parameters:
 - attribute: *nb_people*, legend: « nb of people »
 - attribute: *rate_similar_wanted*, legend: « rate similar wanted », min: 0.0, max: 1.0
 - attribute: *neighbours_distance*, legend: « neighbours distance », step: 1.0

- **Solution :**

```
experiment main_xp type: gui {  
    parameter "nb of people" var: nb_people;  
    parameter "rate similar wanted" var: rate_similar_wanted min: 0.0 max: 1.0;  
    parameter "neighbours distance" var: neighbours_distance step: 1.0;  
  
    output {  
        ...  
    }  
}
```

The user can now modify the value of the global attributes through parameters

Next step: definition of a monitor and a chart!

experiment block: monitor definition

- ❖ A monitor is an output allowing to display the current value of an expression
- ❖ The data to display have to be defined inside the **output** block:
monitor legend value: value

Example

```
experiment main_experiment type:gui{
    //...parameters
    output {
        monitor "Infected people rate" value: infected_rate;

        //...display
    }
}
```



Segregation model 3: Monitor concerning the number of happy people

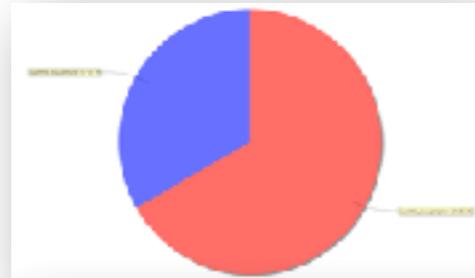
- **To do :** define a monitor to follow the evolution of the number of happy people
- **Solution :**

```
experiment main_xp type: gui {  
    //parameter definition  
  
    output {  
        monitor "nb of happy people" value: nb_happy_people;  
        //display definition  
    }  
}
```

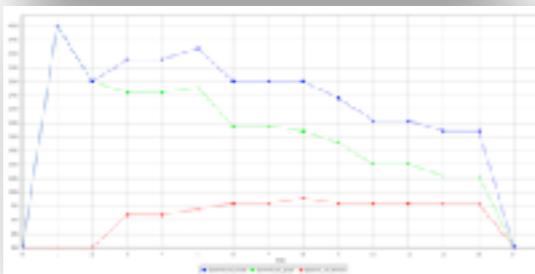
experiment block: chart definition

- ❖ GAMA allows to display several type of charts :

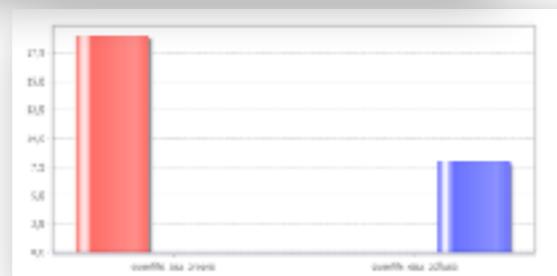
- Pie



- Series



- Histogram



- XY chart

- ❖ A chart is a layer in a display: **chart legend type: chart_type**

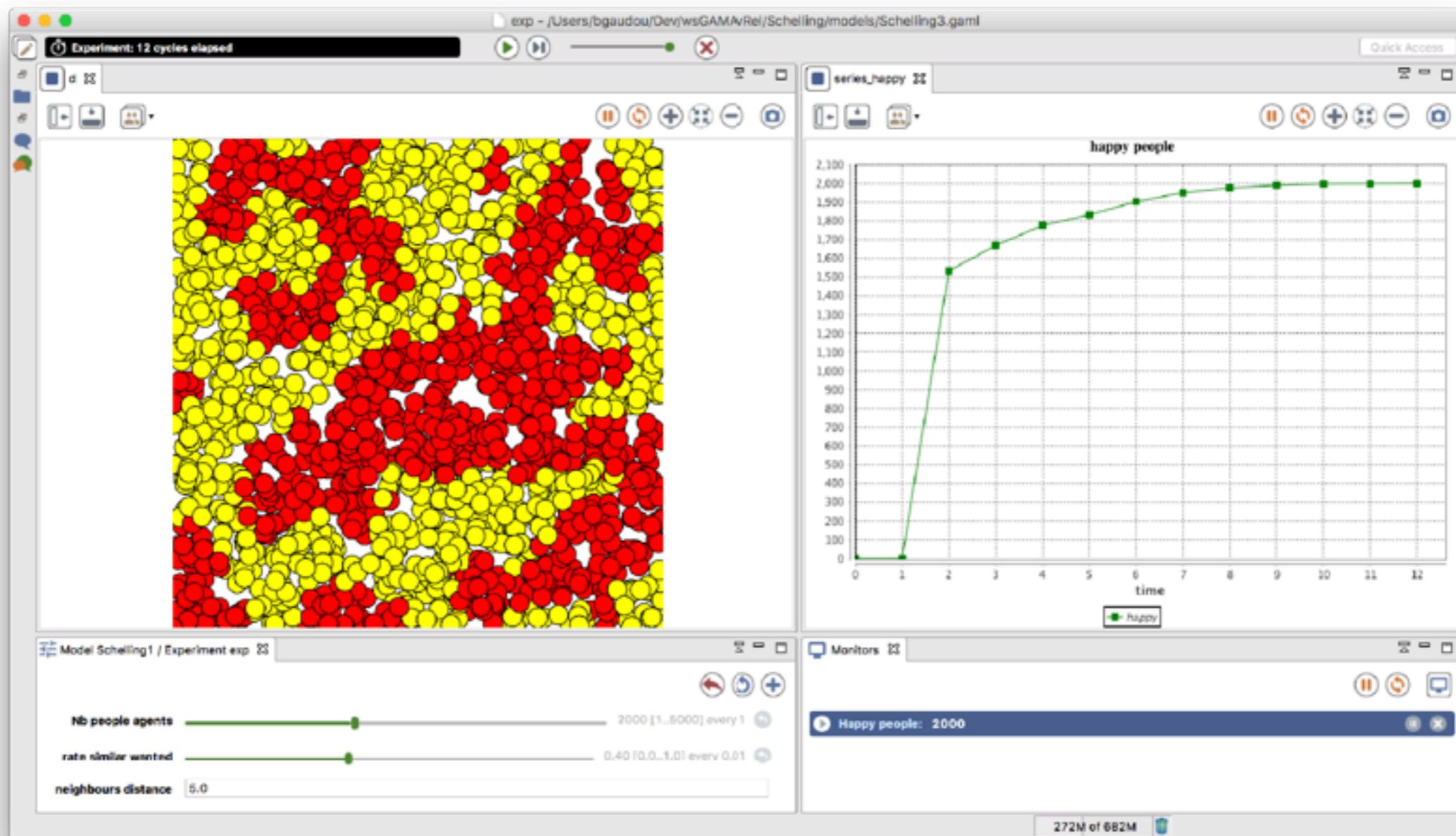
- ❖ The data to display have to be defined inside the **chart** block:
data legend value: value color: colour

Segregation model 3: Chart concerning the number of happy people

- **To do :** define a chart in a new display called *chart* to follow the evolution of the number of happy people.
 - chart name: « evolution of the number of happy people », type: *series*
 - data: "nb of happy people", value: nb_happy_people, color: green
- **Solution :**

```
experiment main_xp type: gui {  
    // parameter definition  
  
    output {  
        // display monitor  
        // map display definition  
        display chart {  
            chart "evolution of the number of happy people" type: series{  
                data "nb of happy people" value: nb_happy_people color: #green;  
            }  
        }  
    }  
}
```

End of step 3



It's time to play with model !

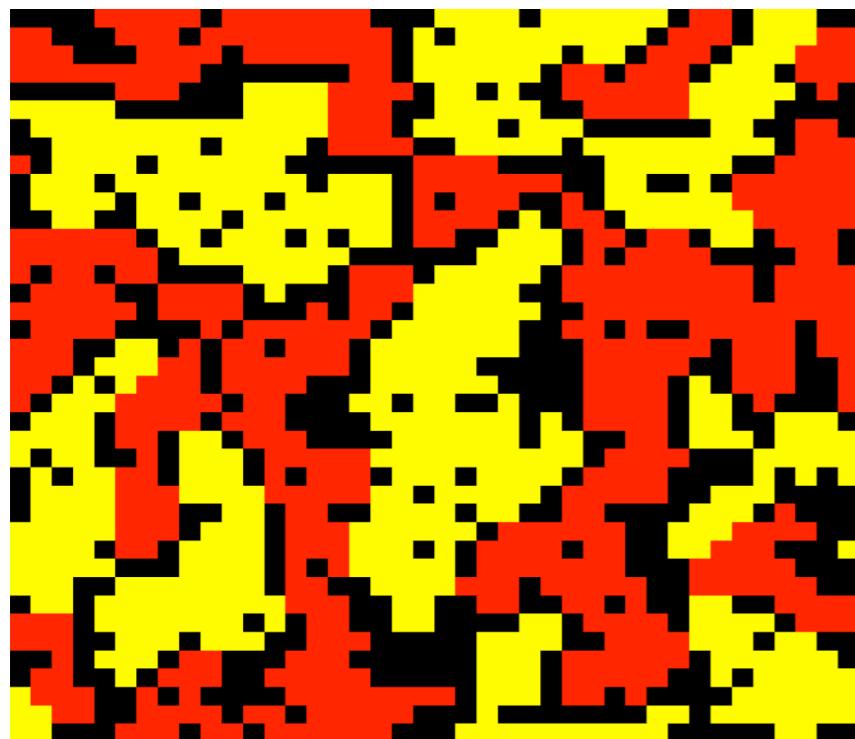
Questions: what is the impact on the

number of clusters of the

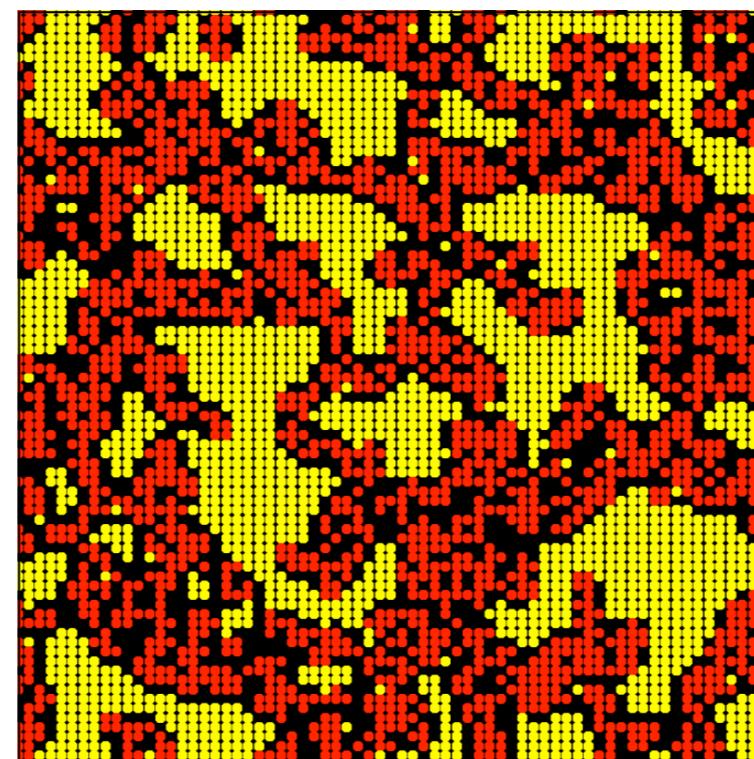
rate_similarity_wanter parameter?

Same question for the *neighbours_distance* parameter

Other implementations of the model are possible!



Cellular automata



Agents and grid



Agents and GIS