SIMULATION GAME EDITOR (SGE) DOCUMENTATION – ALPHA VERSION

REGIEN Marine, BECU Nicolas – June 2023

Basics

SGE is a python game editor, aimed to create a grid and agent-based simulation without having to start from scratch.

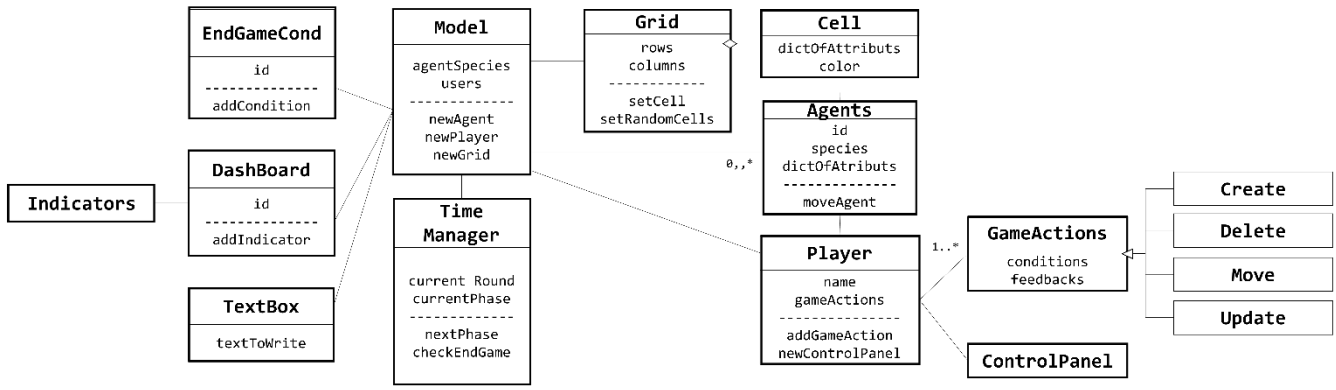
Here are some global explanations to go through this manual:

- At the simplest base, a game is a model and a grid (Steps 1 & 2)
- In python, you need to check the order of the arguments only if you don't put their names before the value.
- Each step is documented by examples, and SGE GitHub can provide a variety of examples.
- You'll find some ressources for pyQT5 applications and lambda functions at the end of the manual

Some acronym explanations:

- int: integrers
- str: string of text between " "
- dict: dictionary { }
- bool: Boolean True or False

Diagramm



Steps and Summary

Step 1: create your Model.....1
Step 2: create your Grid and Cells.....2
Step 3: create your Agents.....4
Step 4: add Admin, Players and GameActions.....6
Step 5: set up Time Management.....8
Step 6: set up EndGame rule and Scores (DashBoard).....10
Step 7: add TextBoxes.....12

Step 1: create your Model

```
myModel=SGModel(1800,900, windowTitle="my Model Title", typeOfLayout ="grid")
```

Create the model

Arguments:

- **width (int):** width of the main window in pixels *(default: 1800px)*
- **height (int):** height of the main window in pixels *(default: 900px)*
- **typeOfLayout ("vertical", "horizontal" or "grid"):** the type of layout used to position the different graphic elements of the simulation *(default: "grid")*
- **x (int, optional):** *used only for grid layout*. defines the number layout grid width *(default: 3)*
- **y (int, optional):** *used only for grid layout*. defines the number layout grid height *(default: 3)*
- **name (str, optional):** the name of the model *(default: "Simulation")*
- **windowTitle (str, optional):** the title of the main window of the simulation. *(default: "myGame")*

```
myModel=SGModel(1500,1000, windowTitle="RehabGame", typeOfLayout ="vertical", name='myFirstModel')
```

```
myModel=SGModel(2000,300,'horizontal')
```

```
myModel=SGModel(700,450,'grid',x=5,y=2)
```

Step 2: create your Grid and your Cells


```
aGrid=myModel.newGrid(7,7,"square",size=60, gap=2,name='grid1')
```

Create a grid.

SGE can have several grids !

Arguments:

- **columns (int):** number of columns (width) *(default: 8)*
- **rows (int):** number of rows (height) *(default: 8)*
- **format ("square", "hexagonal"):** shape of the cells *(default: "square")*
- **color (a color, optional):** background color of the grid *(default: Qt.gray)*.
- **gap (int, optional):** gap size between cells *(default: 3)*
- **size (int, optional):** size of the cells *(default: 30)*
- **name (str):** name of the grid
- **moveable (bool):** grid can be moved by clic and drag *(default: True)*

To customize the cells :

1

```
aGrid.setCells("Resource","2")
```

Set the value of an attribute for all cells

Arguments:

- **aAttribute (str):** name of the attribute to set
- **aValue (str):** value to set the attribute to

2

```
aGrid.setCell("Resource","0",3,4)
```

Set the value of an attribute for a specific cell

Arguments:

- **aAttribute (str):** name of the attribute to set
- **aValue (str):** value to set the attribute to
- **aValueX (int):** a column number
- **aValueY (int):** a row number

3

```
aGrid.setCells_withColumn("ProtectionLevel","Reserve",5)
```

Set the value of an attribute for a specific column

Arguments:

- **aAttribute (str):** name of the attribute to set
- **aValue (str):** value to set the attribute to
- **aColumnNumber (int):** a column number

4

```
aGrid.setCells_withRow("ProtectionLevel","Free",6)
```

Set the value of an attribute for a specific row

Arguments:

- **aAttribute (str):** name of the attribute to set
- **aValue (str):** value to set the attribute to
- **aRowNumber (int):** a row number

5

```
aGrid.setRandomCell("Resource","3")
```

Set the value of an attribute for a random cell

Arguments:

- **aAttribute (str):** name of the attribute to set
- **aValue (str):** value to set the attribute to

6
```
aGrid.setRandomCells("Resource","0",8)
```

Set the value of an attribute for random cells

Arguments:
- **aAttribute (str):** name of the attribute to set
- **aValue (str):** value to set the attribute to
- **numberOfCells (int) :** number of cells

To customize the colors of your cells (POV):

A POV (point of view) permits to associate a attribute value to a color on the game. See examples 1.1 for details

1
```
myModel.newPov("Resource","Resource",{"3":Qt.darkGreen,"2":Qt.green,
"1":Qt.yellow,"0":Qt.white})
```

Declare a new Point of View for cells

Arguments:
- **nameOfPov (str):** name of POV, will appear in the interface
- **aAtt (str):** name of the attribut
- **DictofColors (dict):** a dictionary with all the attribut values, and for each one a Qt.Color
- **listOfGridsToApply (list):** list of grid names where the POV applies *(default: None)*

2
```
myModel.newBorderPov("ProtectionLevel","ProtectionLevel",
{"Reserve":Qt.magenta,"Free":Qt.black})
```

Declare a new Point of View for cells (only border color)

Arguments:
- **nameOfPov (str):** name of POV, will appear in the interface
- **aAtt (str):** name of the attribut
- **DictofColors (dict):** a dictionary with all the attribut values, and for each one a Qt.Color
- **listOfGridsToApply (list):** list of grid names where the POV applies *(default: None)*

```
Sheeps=myModel.newAgentSpecies("Sheeps","circleAgent",{"health":{"good","bad"}})
```

(a) Case of attribute dependent agents

```
Workers=myModel.newAgentSpecies("Workers","triangleAgent1",uniqueColor=Qt.black)
```

(b) Case of unique color agents

Create a new species of Agents

Arguments:

- **aSpeciesName (str):** the species name
- **aSpeciesShape (str):** the species shape (see list below)
- **dictofAttributs (dict):** all the species attributes with all the values **[case (a) only]**
- **aSpeciesDefaultSize (int):** the species shape size *(default: 10)*
- **uniqueColor (Qt.color):** color of the agent **[case (b) only]**

Return:

- a nested dict for the species
- a species

List of possible shapes:

« circleAgent »

« squareAgent »

« ellipseAgent1 » / « ellipseAgent2 »

« rectAgent1 » / « rectAgent2 »

« triangleAgent1 » / « triangleAgent2 »

« arrowAgent1 » / « arrowAgent2 »

To customize the colors of your agents (POV): [case (a) only]

```
Sheeps.newPov("Sheeps -> Health","health",{'good':Qt.blue,'bad':Qt.red})
```

Create a new POV for Agents

Arguments:

- **nameOfPov (str):** name of POV, will appear in the interface
- **concernedAtt (str):** name of the attribute concerned by the declaration
- **DictofColors (dict):** a dictionary with all the attribute values, and for each one a Qt.Color

Case (b) doesn't need it.

Possible actions on agents at initialization :

1

```
m1=myModel.newAgent(aGrid,Sheeps,3,7,
        aDictofAttributs={"health":{"good"},"hunger":{"bad"}})
```

Create a new Agent

Arguments:

- **aGrid (instance):** the grid you want your agent in
- **aAgentSpecies (instance):** the species of your agent
- **ValueX (int):** column position in grid (default: Random)
- **ValueY (int):** row position in grid (default: Random)
- **aID (int):** a identification number (automatically generated)
- **aDictOfAttribute (dict):** the species attributes (automatically generated)

Return:

- a new nest in the species dict for the agent
- an agent

2

```
m2.setValueAgent('health','good')
```

Update an Agent attribute value

Arguments:

- **attribute (str):** attribute concerned by the update
- **value (str):** value previously declared in the species, to update

Admin mode :

By default there is automatically a player called Admin. The legend admin is used to control in admin mode and serves as legend

```
globalLegend = myModel.newLegendAdmin("Global Legend", showAgentsWithNoAtt=True)
```

Create the legend / Admin control panel

Arguments:

- **Name (str):** name of the Legend (default: legend)
- **showAgentsWithNoAtt (bool):** display of species that don't have attributes *(default: False)*

To add a Player :

```
Player1=myModel.newPlayer("Player 1")
```

Create a new Player

Arguments:

- **name (str):** name of the Player (will be displayed)

Create Actions:

1

```
createA1=myModel.newCreateAction(Workers, 20)
```

Add a Create GameAction to the game

A Create Action allows players to create entities on the grid

Arguments:

- **anObjectType (instance):** a AgentSpecies or the keyword "Cell"
- **a Number (int):** number of utilization, could use "infinite"
- **aDictOfAcceptedValue (dict):** attribute with value concerned, could be None

```
createA1.addFeedback(lambda: a1.moveAgent(aGrid,numberOfMovement=3))
```

Add a Feedback to an Action (common to every Game Action)

Arguments:

- **feedback (lambda function, optional):** Feedback action performed only if the actions are executed

An advanced explanation about lambda functions can be found at the end of this documentation.

2

```
myModel.newDeleteAction(Workers, "infinite")
```

Add a Delete GameAction to the game

A Delete Action allows players to delete entities on the grid

Arguments:

- **anObjectType (instance):** a AgentSpecies
- **a Number (int):** number of utilization, could use "infinite"
- **aDictOfAcceptedValue (dict):** attribute with value concerned, could be None

3
```
myModel.newUpdateAction('Cell', 3, {"Resource": "3"})
```

Add a Update GameAction to the game
A Update Action allows players to change the value of the attributes on entities

Arguments:
- **anObjectType (instance):** a AgentSpecies or the keyword "Cell"
- **a Number (int):** number of utilization, could use "infinite"
- **aDictOfAcceptedValue (dict):** attribute with value concerned, could be None

4
```
myModel.newMoveAction(Workers, 1)
```

Add a Move GameAction to the game
A Move Action allows players to move entities on the grid

Arguments:
- **anObjectType (instance):** a AgentSpecies
- **a Number (int):** number of utilization, could use "infinite"
- **aDictOfAcceptedValue (dict):** attribute with value concerned, could be None

Add Actions to the Player:

```
Player1.addGameAction(createA1)
```

Add an action to the Player *(common to every GameAction)*

Arguments:
- **aGameAction (instance):** myModel.createAction instance

Create the Player Control Panel:

Like the global legend for Admin mode, each player need its own ControlPanel to play

```
Player1ControlPanel = Player1.newControlPanel(
    "Player 1 Actions", showAgentsWithNoAtt=True)
```

To create a Player Control Panel (only with the GameActions related elements)

Arguments:
- **Name (str):** name of the Control Panel, displayed
- **showAgentsWithNoAtt (bool):** display of species that don't have attributes *(default: False)*

How to select your players :

In the game there are 2 options to switch between admin and player(s)

1
```
userSelector=myModel.newUserSelector()
```

To create an User Selector in your game. Functions automatically with the players declared in your model.

2
```
myModel.currentPlayer='Player 1'
```

At initialization, you can choose witch player is active.

Arguments:
- **Name (str):** name of the players. Could be "Admin" for Admin mode.

Time Manager explanation:

*In SGE the Game time is organized by **an infinite sequence of rounds, composed of phases and run by class TimeManager**. These phases could be Game Phases (for player to play according to Game Actions predefined in step 4) **or** Model Phases (model activities or events that can occur in between Game Phases or at the beginning/end of rounds)*

To build a Game Phase:

Reminder: Game Actions are explained in step 4

```
myModel.timeManager.newGamePhase('Phase 1',[Player1,Player2])
```

To add a Game Phase in a round
Arguments:

- **name (str):** name displayed on the TimeLabel
- **activePlayers (str):** list of players concerned about the phase *(default: all)*

To build a Model Phase:

1 *Create a Model Action*

```
aModelAction4 =myModel.newModelAction(lambda: aGrid.setRandomCells("landUse","forest",2))
```

To add a model action which can be executed during a modelPhase
Arguments:

- **actions (lambda function):** Actions the model performs during the phase (add, delete, move...)
- **conditions (lambda function, optional):** Actions are performed only if the condition returns true
- **feedbacks (lambda function, optional):** Feedback actions performed only if the actions are executed

An advanced explanation about lambda functions can be found at the end of this documentation.

Alternatives :

```
aModelAction4.addCondition(lambda: myModel.getCurrentRound()==3)
```

```
aModelAction4.addFeedback(lambda: i1.setResult(i1.result + 5))
```

```
aModelAction2.addModelAction(anAction)
```

Important : one Model Action can have several actions, conditions and feedbacks. In this case you must put them in an array between […] !

2 *Build a Model Phase*

```
myModel.timeManager.newModelPhase(aModelAction4)
```

To add a Model Phase in a round
Arguments:

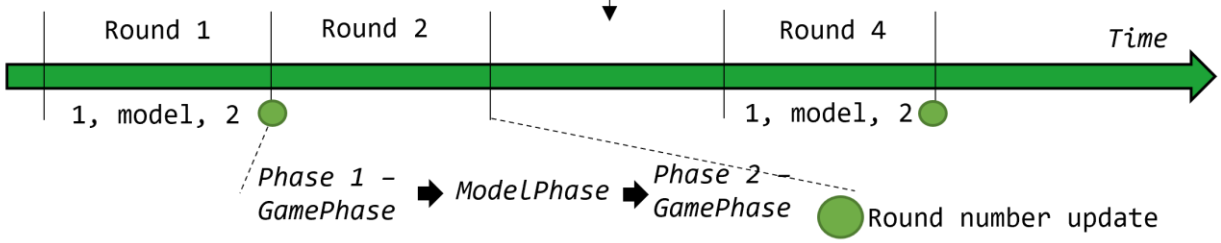
- **Model actions (lambda function or instances):** list of model actions
- **conditions (lambda function, optional):** actions are performed only if the condition returns true
- **name (str):** name displayed on the TimeLabel

You also can skip newModelAction() and directly describe your ModelAction in newModelPhase()

Creation of the « Timeline »:

The Phases order declaration is essential. Each round will be build by phases in the order of the script.

```
myModel.timeManager.newGamePhase('Phase 1', [Player1, Player2])
myModel.timeManager.newModelPhase(lambda: aGrid.setRandomCells("landUse","shrub",2))
myModel.timeManager.newGamePhase('Phase 2', [Player1, Player2])
```



Add a Time Label:

```
GameRounds=myModel.newTimeLabel("My Game Time",Qt.white,Qt.black,Qt.red)
```

To add the Time Label in the game

Arguments:

- **name (str):** name displayed on the TimeLabel
- **backgroundColor (Qt Color):** color of the background *(default: Qt.white)*
- **borderColor (Qt Color):** color of the border *(default: Qt.black)*
- **textColor (Qt Color):** color of the text *(default: Qt.black)*

```
DashBoard=myModel.newDashBoard(borderColor=Qt.black,textColor=Qt.red)
```

Create the DashBoard of the game
Arguments:

- **title (str):** title of the widget *(default: "Phases&Rounds")*
- **displayRefresh (str):** type of refresh *(default: 'instantaneous')*
- **backgroundColor (Qt Color):** color of the background *(default: Qt.transparent)*
- **borderColor (Qt Color):** color of the border *(default: Qt.black)*
- **textColor (Qt Color):** color of the text *(default: Qt.black)*

To add indicators:

```
i1=DashBoard.addIndicator("sumAtt",'cell','Resource')
```

Add an Indicator on the DashBoard
Arguments:

- **method (str):** name of the method / list in the documentation
- **entity (str):** "cell" or "agent" or None *(only for score, see below)*
- **color (Qt color):** text color *(default: Qt.black)*
- **attribute (str):** concerned attribute
- **value (str, optionnal):** for certain methods, a value is required
- **indicatorName (str, optionnal):** name displayed on the dashboard
- **isDisplay (bool):** display on the dashboard *(default: True)*

List of possible methods:
[for cells:]
- **sumAtt,avgAtt** : sum and average
- **minAtt,maxAtt** : minimum and maximum of a attribute
- **nbEqualTo,nbWithLess,nbWithMore** : logic test with a value
- **nb** : count of entities

[for agents:]
- nb : count of entities

Add an Score on the DashBoard
Set entity to None.

Alternatives:

```
ind=DashBoard.addIndicator_Sum('cell','Resource')
ind=DashBoard.addIndicator_Avg('cell','Resource')
ind=DashBoard.addIndicator_Min('cell','Resource')
ind=DashBoard.addIndicator_Max('cell','Resource')
ind=DashBoard.addIndicator_EqualTo('cell','Resource','2')
ind=DashBoard.addIndicator_WithLess('cell','Resource','3')
ind=DashBoard.addIndicator_WithMore('cell','Resource','1')
ind=DashBoard.addIndicator_Nb('cell','Resource')
```

Arguments:

- **entity (str):** "cell" or "agent"
- **attribute (str):** concerned attribute
- **value (str, optionnal):** for certain methods, a value is required
- **indicatorName (str, optionnal):** name displayed on the dashboard
- **isDisplay (bool):** display on the dashboard *(default: True)*

To finish the step :

```
DashBoard.showIndicators()
```

```
endGameRule=myModel.newEndGameRule(numberRequired=2)
```

Create the EndGame Rule Board of the game
Arguments:

- **title (str):** header of the board, displayed *(default: EndGame Rules)*
- **numberRequired (int):** number of completed conditions to trigger EndGame

To add EndGame condition:

1

```
endGameRule.addEndGameCondition_onIndicator(i1,"equal",90,
                                            name="Resource equal to 90")
```

Create an EndGame Condition with an Indicator
Arguments:

- **indicator (instance):** indicator concerned
- **logicalTest (str):** logical test concerned, see list on documentation
- **objective (int):** objective value to do logical test with
- **name (str):** name of the end game condition, displayed *(default: "Indicator based condition")*
- **color (Qt.color):** text color *(default: black)*
- **isDisplay (bool):** is displayed in the EndGameRule board *(default: True)*

2

```
targetCell = aGrid.getCell(1,5)
endGameRule.addEndGameCondition_onEntity(targetCell,'Resource',
                                         "greater",2)
```

Create an EndGame Condition based on an Entity state
Arguments:

- **indicator (instance):** an entity (a cell)
- **attribute (str):** attribute concerned
- **logicalTest (str):** logical test concerned, see list on documentation
- **objective (int):** objective value to do logical test with
- **name (str):** name of the end game condition, displayed *(default: "Entity based condition")*
- **color (Qt.color):** text color *(default: black)*
- **isDisplay (bool):** is displayed in the EndGameRule board *(default: True)*

3

```
endGameRule.addEndGameCondition_onGameRound("less",10)
```

Create an EndGame Condition on the time (game rounds)
Arguments:

- **logicalTest (str):** logical test concerned, see list on documentation
- **objective (int):** objective value to do logical test with
- **name (str):** name of the end game condition, displayed *(default: "Entity based condition")*
- **color (Qt.color):** text color *(default: black)*
- **isDisplay (bool):** is displayed in the EndGameRule board *(default: True)*

To finish the step:

```
endGameRule.showEndGameConditions()
```

Step 7: add TextBoxes

```
TextBox = myModel.newTextBox(
    title='Game is starting', textToWrite="Welcome in your game !")
```

Create a text box
Arguments:
- **textToWrite (str):** displayed text in the widget *(default: "Welcome in the game!")*
- **title (str):** name of the widget *(default: "Text Box 1")*

Customization:

1
```
TextBox.addText("The first text!", toTheLine=True)
```

Add text following the existing one
Arguments:
- **text (str):** displayed text
- **toTheLine (bool):** if true, skip a line before adding

2
```
TextBox.setNewText("This is an another text.")
```

Erase previous text and replace it with the new text
Arguments:
- **text (str):** new text

3
```
TextBox.setTitle('a new title !')
```

Replace the title by a new title
Arguments:
- **title (str):** new title

4
```
TextBox.setTextFormat(size=12)
```

Customize the font and the size of the text
Arguments:
- **fontName (str):** desired font *(default : Verdana)*
- **size (int):** desired size of the text *(default : 12)*

5
```
TextBox.setTitleColor("green")
```

Set the color
Arguments:
- **color (str):** desired color

6
```
TextBox.setTitleSize(14)
```

Set the size
Arguments:
- **size (int):** desired size

Lambda functions

In python a lambda function is useful when you want to employ a function from a class within another class' function.

If the answer to the question: "is the action associated with the object you're worling on?" is false, you surely need a lambda function.

How to build a lambda function :

- Lambda: [a]
- Any object previously declared in the file [b]
- Any function from this object class [c]

```
myModel.newModelAction(lambda: aGrid.setRandomCells("landUse","shrub",2))
                        [a]     [b]        [c]
```

Function [c] will be executed for the object [b] only when the modelAction will be performed

PyQT5 CheatSheet

Application Action Bar

Every game client has a Action Bar like this:

Zoom management

Next Phase

Points of view

Acknowledgement

Have you ever dreamed of an editor that easily turns your simulation grid-based game following an agent-based approach ideas into reality?

Simulation Game Editor (SGE) is a Python based solution powered by PyQt5. It aims to help game modellers to create a simulation grid-based game following an agent-based approach without having to develop the model: you just need to change some variables.

SGE is unique from other pre-existent simulation tools: it implements the notion of point of views, game actions and player time round directly in the model code, like an all-in-one pack. SGE aims to create distributed asymmetric simulations: every participant will be immersed in a different point of view according to their personal aptitudes and understandings.

Calculate, test, develop but faster.

SGE is developped under the supervision of LIENSs Laboratory (La Rochelle University, France) within different research projects. SGE answer to an academic need in serious game to have simplier ways to create serious games. SGE first version was presented at ISAGA 2023.

Authors and contributors

***RÉGIEN**, Marine*
***BECU**, Nicolas*
***OSSANT**, Alexis*

UMR LIENSs 7266, La Rochelle Université