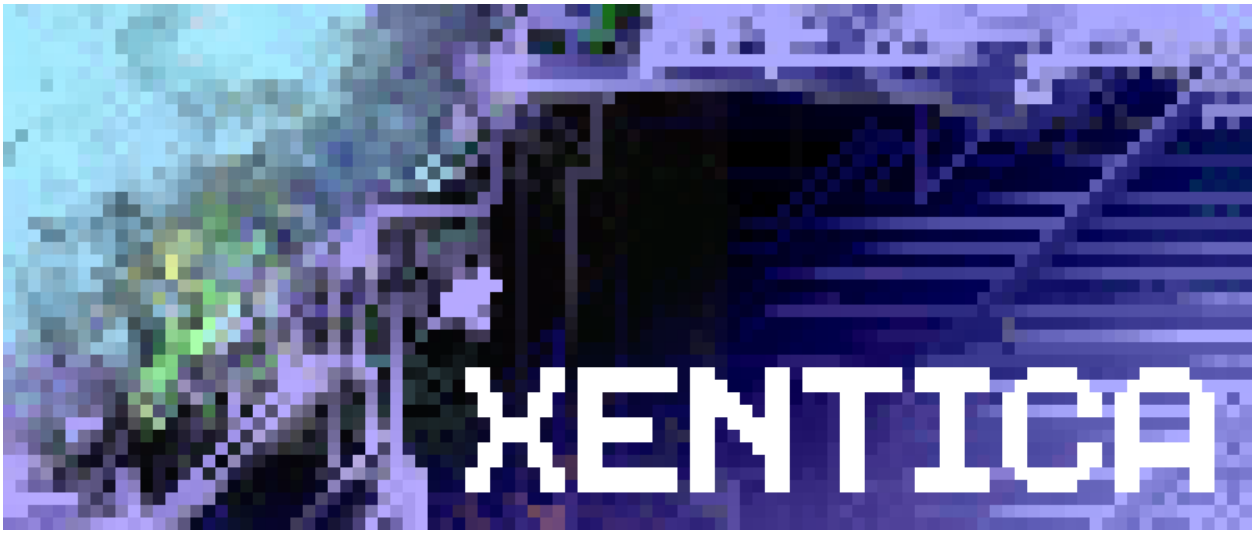
Xentica Documentation

Release 0.1.0

Andrey Zamaraev

Jun 16, 2018

Contents



Xentica is the framework to build GPU-accelerated models for multi-dimensional cellular automata. Given pure Python definitions, it generates kernels in CUDA C and runs them on NVIDIA hardware.

Warning: Current version is a work-in-progress, it works to some degree, but please do not expect something beneficial from it. As planned, really useful stuff would be available only starting from version 0.3.

CHAPTER 1

User Guide

If you brave enough to ignore the warning above, dive right into this guide. Hopefully, you will manage to install Xentica on your system and at least run some examples. Otherwise, just read Tutorial and watch some videos to decide is it worth waiting for future versions.

1.1 Installation Instructions

Xentica is planned to run with several GPU backends in future, like *CUDA*, *OpenCL* and *GLSL*. However, right now, only *CUDA* is supported.

Warning: If your GPU is not CUDA-enabled, this guide is **not** for you. Framework will just not run, no matter how hard you try. You may check the list of CUDA-enabled cards, if you have any doubts.

Note: This page currently containing instructions only for Debian-like systems. If you are on other system, you still can use links to pre-requisites in order to install them. If so, please contact us by opening an issue on GitHub. We could help you if you'll meet some troubles during installation, and also your experience could be used to improve this document.

1.1.1 Core Prerequisites

In order to run CA models without any visualization, you have to correctly install following software.

- NVIDIA CUDA Toolkit

  Generally, you can install it just from your distrubution's repository:

```
$ sudo apt-get install nvidia-cuda-toolkit
```

Although, default packages are often out of date, so in case you have one of those latest cool GPU, you may want to upgrade to the latest CUDA version from official NVIDIA source. We'll hint you with a good article explaining how to do it. But you are really on your own with this stuff.

- Python 3.5+

  Your distribution should already have all you need:

```
$ sudo apt-get install python3 python3-dev python3-pip wheel
```

- NumPy

  Once Python3 is correctly installed, you can install NumPy by:

```
$ pip3 install numpy
```

- PyCUDA

  If CUDA is correctly installed, you again can simply install PyCUDA with `pip`:

```
$ pip3 install pycuda
```

Other pre-requisites should be transparently installed with the main Xentica package.

1.1.2 GUI Prerequisites

If you are planning to run visual examples with Moire GUI, you have to install Kivy framework.

Its pre-requisites could be installed by:

```
$ sudo apt-get install \
  build-essential \
  git \
  ffmpeg \
  libsdl2-dev \
  libsdl2-image-dev \
  libsdl2-mixer-dev \
  libsdl2-ttf-dev \
  libportmidi-dev \
  libswscale-dev \
  libavformat-dev \
  libavcodec-dev \
  zlib1g-dev
```

Then, to install stable Kivy:

```
$ pip3 install Cython==0.25 Kivy==1.10.0
```

On latest Debian distributions you can meet conflicts with `libsdl-mixer`. Then, try to install latest developer version, like:

```
$ pip3 install Cython==0.27.3 git\+https://github.com/kivy/kivy.git
```

If you have any other troubles with that, please refer to the official Kivy installation instructions.

1.1.3 Main package

Xentica package could be installed with:

```
$ pip3 install xentica
```

Note, it does not depends on pre-requisites described above, but you still need to install them properly, or Xentica will not run.

1.1.4 Run Xentica examples

In order to run Game of Life model built with Xentica:

```
$ pip3 install moire
$ wget https://raw.githubusercontent.com/a5kin/xentica/master/examples/game_of_life.py
$ python3 game_of_life.py
```

Or, if you are using `optirun`:

```
$ optirun python3 game_of_life.py
```

1.1.5 Run tests

In order to run Xentica tests:

```
$ git clone https://github.com/a5kin/xentica.git
$ cd xentica/tests
$ ./run_tests.sh
```

Or, if you are using `optirun`:

```
$ optirun python3 run_tests.py
```

Tests will also give you performance metrics. Please update us with them, along with the info about you GPU and Xentica version.

1.2 Tutorial

Tutorial is coming soon. Meanwhile, you may check the official Game Of Life example.

```
"""
A collection of models derived from Conway's Game Of Life.

Experiment classes included.

"""
from xentica import core
from xentica import seeds
from xentica.core import color_effects


class GameOfLife(core.CellularAutomaton):
    """
    The classic CA built with Xentica framework.

    It has only one property called ``state``, which is positive
```

(continues on next page)

(continued from previous page)

```
    integer with max value of 1.

    """

    state = core.IntegerProperty(max_val=1)

    class Topology:
        """
        Mandatory class for all ``CellularAutomaton`` instances.

        All class variables below are also mandatory.

        Here, we declare the topology as a 2-dimensional orthogonal
        lattice with Moore neighborhood, wrapped to a 3-torus.

        """

        dimensions = 2
        lattice = core.OrthogonalLattice()
        neighborhood = core.MooreNeighborhood()
        border = core.TorusBorder()

    def emit(self):
        """
        Implement the logic of emit phase.

        Statements below will be translated into C code as emit kernel
        at the moment of class creation.

        Here, we just copy main state to surrounding buffers.

        """
        for i in range(len(self.buffers)):
            self.buffers[i].state = self.main.state

    def absorb(self):
        """
        Implement the logic of absorb phase.

        Statements below will be translated into C code as well.

        Here, we sum all neigbors buffered states and apply Conway
        rule to modify cell's own state.

        """
        neighbors_alive = core.IntegerVariable()
        for i in range(len(self.buffers)):
            neighbors_alive += self.neighbors[i].buffer.state
        is_born = (8 >> neighbors_alive) & 1
        is_sustain = (12 >> neighbors_alive) & 1
        self.main.state = is_born | is_sustain & self.main.state

    @color_effects.MovingAverage
    def color(self):
        """
        Implement the logic of cell's color calculation.
```

(continues on next page)

(continued from previous page)

```
        Must return a tuple of RGB values computed from ``self.main``
        properties.

        Also, must be decorated by a class from ``color_effects``
        module.

        Here, we simply define 0 state as pure black, and 1 state as
        pure white.

        """
        r = self.main.state * 255
        g = self.main.state * 255
        b = self.main.state * 255
        return (r, g, b)


class GameOfLifeStatic(GameOfLife):
    """
    Game of Life variant with static border made of live cells.

    This is an example of how easy you can inherit other models.

    """

    class Topology(GameOfLife.Topology):
        """
        You can inherit parent class ``Topology``.

        Then, override only necessary variables.

        """

        border = core.StaticBorder(1)


class GameOfLifeColor(GameOfLife):
    """
    Game Of Life variant with RGB color.

    This is an example of how to use multiple properties per cell.

    """

    state = core.IntegerProperty(max_val=1)
    red = core.IntegerProperty(max_val=255)
    green = core.IntegerProperty(max_val=255)
    blue = core.IntegerProperty(max_val=255)

    def emit(self):
        """Copy all properties to surrounding buffers."""
        for i in range(len(self.buffers)):
            self.buffers[i].state = self.main.state
            self.buffers[i].red = self.main.red
            self.buffers[i].green = self.main.green
            self.buffers[i].blue = self.main.blue

    def absorb(self):
```

(continues on next page)

(continued from previous page)

```
        """
        Calculate RGB as neighbors sum for living cell only.

        Note, parent ``absorb`` method should be called using direct
        class access, not via ``super``.

        """
        GameOfLife.absorb(self)
        red_sum = core.IntegerVariable()
        for i in range(len(self.buffers)):
            red_sum += self.neighbors[i].buffer.red + 1
        green_sum = core.IntegerVariable()
        for i in range(len(self.buffers)):
            green_sum += self.neighbors[i].buffer.green + 1
        blue_sum = core.IntegerVariable()
        for i in range(len(self.buffers)):
            blue_sum += self.neighbors[i].buffer.blue + 1
        self.main.red = red_sum * self.main.state
        self.main.green = green_sum * self.main.state
        self.main.blue = blue_sum * self.main.state

    @color_effects.MovingAverage
    def color(self):
        """Calculate color as usual."""
        r = self.main.state * self.main.red
        g = self.main.state * self.main.green
        b = self.main.state * self.main.blue
        return (r, g, b)


class GameOfLife6D(GameOfLife):
    """
    Game of Life variant in 6D.

    Nothing interesting, just to prove you can do it with ease.

    """

    class Topology(GameOfLife.Topology):
        """
        Hyper-spacewalk, is as easy as increase ``dimensions`` value.

        However, we are also changing neighborhood to Von Neumann
        here, to prevent neighbors number exponential grow.

        """

        dimensions = 6
        neighborhood = core.VonNeumannNeighborhood()


class GOLExperiment(core.Experiment):
    """
    Particular experiment for the vanilla Game of Life.

    Here, we define constants and initial conditions from which the
    world's seed will be generated.
```

(continues on next page)

(continued from previous page)

```
    The ``word`` is RNG seed string. The ``size``, ``zoom`` and
    ``pos`` are board contstants. The ``seed`` is a pattern used in
    initial board state generation.

    ``BigBang`` is a pattern when small area initialized with a
    high-density random values.

    """

    word = "OBEY XENTICA"
    size = (640, 360, )
    zoom = 3
    pos = [0, 0]
    seed = seeds.patterns.BigBang(
        pos=(320, 180),
        size=(100, 100),
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
    )


class GOLExperiment2(GOLExperiment):
    """
    Another experiment for the vanilla GoL.

    Since it is inherited from ``GOLExperiment``, we can redefine only
    values we need.

    ``PrimordialSoup`` is a pattern when the whole board is
    initialized with low-density random values.

    """

    word = "XENTICA IS YOUR GODDESS"
    seed = seeds.patterns.PrimordialSoup(
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
    )


class GOLExperimentColor(GOLExperiment):
    """
    Experiment for GameOfLifeColor.

    Here, we introduce ``fade_out`` constant, which is used in
    rendering causing cells slowly fade out.

    Note, it is only aestetic effect, and does not affect real cell
    state.

    """

    fade_out = 10
    seed = seeds.patterns.PrimordialSoup(
```

(continues on next page)

(continued from previous page)

```
        vals={
            "state": seeds.random.RandInt(0, 1),
            "red": seeds.random.RandInt(0, 255),
            "green": seeds.random.RandInt(0, 255),
            "blue": seeds.random.RandInt(0, 255),
        }
    )


class GOLExperiment6D(GOLExperiment2):
    """
    Special experiment for 6D Life.

    Here, we define the world with 2 spatial and 4 looped
    micro-dimensions, 3 cells length each.

    As a result, we get large quasi-stable oscillators, looping over
    micro-dimensions. Strangely formed, but nothing interesting,
    really.

    """

    size = (640, 360, 3, 3, 3, 3)


def main():
    # Finally, an example of how to run model/experiment interactively
    import moire
    ca = GameOfLifeColor(GOLExperimentColor)
    gui = moire.GUI(runnable=ca)
    gui.run()


if __name__ == "__main__":
    main()
```

CHAPTER 2

API Reference

2.1 The Core (xentica.core)

Xentica core functionality is available via modules from this package.

In addition, you may use `core` package as a shortcut to the main classes of the framework.

- **Base classes**
  - `core.CellularAutomaton` → *xentica.core.base.CellularAutomaton*
  - `core.Experiment` → *xentica.core.experiment.Experiment*
- **Lattices**
  - `core.OrthogonalLattice` → *xentica.core.topology.lattice.OrthogonalLattice*
- **Neighborhoods**
  - `core.MooreNeighborhood` → *xentica.core.topology.neighborhood.MooreNeighborhood*
  - `core.VonNeumannNeighborhood` → *xentica.core.topology.neighborhood.VonNeumannNeighborhood*
- **Borders**
  - `core.TorusBorder` → *xentica.core.topology.border.TorusBorder*
  - `core.StaticBorder` → *xentica.core.topology.border.StaticBorder*
- **Properties**
  - `core.IntegerProperty` → *xentica.core.properties.IntegerProperty*
- **Variables**
  - `core.IntegerVariable` → *xentica.core.variables.IntegerVariable*

The classes listed above are all you need to build CA models and experiments with Xentica, unless you are planning to implement custom core features like new lattices, borders, etc.

2.1.1 Base Classes (xentica.core.base)

The module with the base class for all CA models.

All Xentica models should be inherited from *CellularAutomaton* base class. Inside the model, you should correctly define the `Topology` class and describe the CA logic in `emit()`, `absorb()` and `color()` methods.

`Topology` is the place where you define the dimensionality, lattice, neighborhood and border effects for your CA. See *xentica.core.topology* package for details.

The logic of the model will follow Buffered State Cellular Automaton (BSCA) principle. In general, every cell mirrors its state in buffers by the number of neighbors, each buffer intended for one of neighbors. Then, at each step, the interaction between cells are performed via buffers in 2-phase emit/absorb process. More detailed description of BSCA principle is available in The Core section of The Concept document.

`emit()` describes the logic of the first phase of BSCA. At this phase, you should fill cell's buffers with corresponding values, depending on cell's main state and (optionally) on neighbors' main states. The most easy logic is to just copy the main state to buffers. It is esspecially useful when you're intending to emulate classic CA (like Conway's Life) with BSCA. Write access to main state is prohibited there.

`absorb()` describes the logic of the second phase of BSCA. At this phase, you should set the cell's main state, depending on neighbors' buffered states. Write access to buffers is prohibited there.

`color()` describes how to calculate cell's color from its raw state. See detailed instructions on it in *xentica.core.color_effects*.

The logic of the functions from above will be translated into C code at the moment of class creation. For the further instructions on how to use cell's main and buffered states, see *xentica.core.properties*, for the instructions on variables and expressions with them, see *xentica.core.variables*.

A minimal example, the CA where each cell is taking the mean value of its neighbors each step:

```
from xentica import core
from xentica.core import color_effects

class MeanCA(core.CellularAutomaton):

    state = core.IntegerProperty(max_val=255)

    class Topology:
        dimensions = 2
        lattice = core.OrthogonalLattice()
        neighborhood = core.MooreNeighborhood()
        border = core.TorusBorder()

    def emit(self):
        for i in range(len(self.buffers)):
            self.buffers[i].state = self.main.state

    def absorb(self):
        s = core.IntegerVariable()
        for i in range(len(self.buffers)):
            s += self.neighbors[i].buffer.state
        self.main.state = s / len(self.buffers)
```

(continues on next page)

(continued from previous page)

```
    @color_effects.MovingAverage
    def color(self):
        v = self.main.state
        return (v, v, v)
```

class xentica.core.base.**BSCA**

Bases: type

Meta-class for *CellularAutomaton*.

Performs all necessary stuff to generate GPU kernels given class definition.

It is also preparing `main`, `buffers` and `neighbors` class variables being used in `emit()`, `absorb()` and `color()` methods.

append_code(*code*)

Append `code` to kernel's C code.

build_absorb()

Generate `absorb()` kernel.

Returns String with C code for `absorb()` kernel.

build_defines()

Generate `#define` section for all kernels.

Returns String with C code with necessary defines.

build_emit()

Generate `emit()` kernel.

Returns String with C code for `emit()` kernel.

build_render()

Generate `render()` kernel.

Returns String with C code for `render()` kernel.

coords_declared

Check if coordinate variables are declared.

Returns `True` if coordinate variables are declared, `False` otherwise.

declare(*prop*)

Mark property declared.

Parameters **prop** – *Property* subclass instance.

declare_coords()

Mark coordinate variables declared.

deferred_write(*prop*)

Declare a property for deferred write.

The property will be written into a memory at the end of C code generation.

Parameters **prop** – *Property* subclass instance.

define_constant(*constant*)

Remember the constant is defined.

Parameters **constant** – *Constant* instance.

index_to_coord(*i*)
Wrap `lattice.index_to_coord` method.

Parameters **i** – Cell's index.

is_constant(*constant*)
Check if the constant is defined.

Parameters **constant** – *Constant* instance.

Returns `True` if constant is defined, `False` otherwise.

is_declared(*prop*)
Check if `prop` property is declared.

Parameters **prop** – *Property* subclass instance.

Returns `True` if property is declared, `False` otherwise.

is_unpacked(*prop*)
Check if `prop` property is unpacked.

Parameters **prop** – *Property* subclass instance.

Returns `True` if property is unpacked, `False` otherwise.

pack_state(*state*)
Pack state structure into raw in-memory representation.

Returns Integer representing packed state.

unpack(*prop*)
Mark `prop` property unpacked.

Parameters **prop** – *Property* subclass instance.

class xentica.core.base.**CellularAutomaton**(*experiment_class*)
Bases: `object`

Base class for all Xentica models.

Compiles GPU kernels generated by *BSCA* metaclass, initializes necessary GPU arrays and popupates them with the seed.

After initialization, you can run step-by-step simulation and render the field at any moment:

```
from xentica import core
import moire

class MyCA(core.CellularAutomaton):
    # ...

class MyExperiment(core.Experiment):
    # ...

ca = MyCA(MyExperiment)
ca.set_viewport((320, 200))

# run CA manually for 100 steps
for i in range(100):
    ca.step()
# render current timestep
frame = ca.render()
```

(continues on next page)

(continued from previous page)

```
# or run the whole process interactively with Moire
gui = moire.GUI(runnable=ca)
gui.run()
```

Parameters **experiment_class** – `Experiment` instance, holding all necessary parameters for the field initialization.

apply_speed(*dval*)
Change the simulation speed.

Usable only in conduction with Moire, although you can use the `speed` value in your custom GUI too.

Parameters **dval** – Delta by which speed is changed.

load(*filename*)
Load the CA state from `filename` file.

render()
Render the field at the current timestep.

You must call *`set_viewport()`* before do any rendering.

Returns NumPy array of `np.uint8` values, `width * height * 3` size. The RGB values are consecutive.

save(*filename*)
Save the CA state into `filename` file.

set_viewport(*size*)
Set viewport (camera) size and initialize GPU array for it.

Parameters **size** – tuple with width and height in pixels.

step()
Perform a single simulation step.

`timestep` attribute will hold the current step number.

toggle_pause()
Toggle `paused` flag.

When paused, the `step()` method does nothing.

class xentica.core.base.**CachedNeighbor**
Bases: `object`

Utility class, intended to hold main and buffered CA state.

2.1.2 Experiments (xentica.core.experiment)

The collection of classes to describe experiments for CA models.

Experiment is a class with CA parameters stored as class variables. Different models may have a different set of parameters. To make sure all set correct, you should inherit your experiments from *`Experiment`* class.

The quick example:

```
from xentica import core, seeds

class MyExperiment(core.Experiment):
    # RNG seed string
    word = "My Special String"
    # field size
    size = (640, 360, )
    # initial field zoom
    zoom = 3
    # initial field shift
    pos = [0, 0]
    # A pattern used in initial board state generation.
    # BigBang is a small area initialized with high-density random values.
    seed = seeds.patterns.BigBang(
        # position Big Bang area
        pos=(320, 180),
        # size of Big Bang area
        size=(100, 100),
        # algorithm to generate random values
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
   )
```

class xentica.core.experiment.**Experiment**

Bases: object

Base class for all experiments.

Right now doing nothing, but will be improved in future versions. So it is adviced to inherit your experiments from it.

2.1.3 Properties (xentica.core.properties)

The collection of classes to describe properties of CA models.

Warning: Do not confuse with Python properties.

Xentica properties are declaring as class variables and helping you to organize CA state into complex structures.

Each *CellularAutomaton* instance should have at least one property declared. The property name is up to you. If your model has just one value for state (like in most classic CA), the best practice is to call it `state` as follows:

```
from xentica import core

class MyCA(core.CellularAutomaton):
    state = core.IntegerProperty(max_val=1)
    # ...
```

Then, you can use it in expressions of `emit()`, `absorb()` and `color()` functions as:

`self.main.state` to get and set main state;

`self.buffers[i].state` to get and set i-th buffered state;

`self.neighbors[i].buffer.state` to get and set i-th neighbor buffered state.

Xentica will take care of all other things, like packing CA properties into binary representation and back, storing and getting corresponding values from VRAM, etc.

Most of properties will return *DeferredExpression* on access, so you can use them safely in mixed expressions:

```
self.buffers[i].state = self.main.state + 1
```

class xentica.core.properties.**Property**

Bases: *xentica.core.variables.DeferredExpression*

Base class for all properties.

Has a vast set of default functionality already implemented. Though, you are free to re-define it all to implement really custom behavior.

best_type

Get type that suits best to store a property.

Returns tuple representing best type: `(bit_width, numpy_dtype, gpu_c_type)`

bit_width

Get the number of bits necessary to store a property.

Returns Positive integer, a property's bit width.

calc_bit_width()

Calculate the property's bit width.

This is the method you most likely need to override. It will be called from *bit_width()*.

Returns Positive integer, calculated property's width in bits.

coords_declared

Test if the coordinates variables are declared.

ctype

Get C type, based on result of *best_type()*.

Returns C type that suits best to store a property.

declare_once()

Generate C code to declare a variable holding cell's state.

You must push the generated code to BSCA via `self._bsca.append_code()`, then declare necessary stuff via `self._bsca.declare()`.

You should also take care of skipping the whole process if things are already declared.

declared

Test if the state variable is declared.

dtype

Get NumPy dtype, based on result of *best_type()*.

Returns NumPy dtype that suits best to store a property.

set_bsca(*bsca*, *buf_num*, *nbr_num*)

Set up a reference to BSCA instance.

Do not override this method, it is cruicial to inner framework mechanics.

Parameters

- **bsca** – *CellularAutomaton* instance.
- **buf_num** – Buffer's index, associated to property.

- **nbr_num** – Neighbor's index, associated to property.

width
Get the number of memory cells to store a property.

In example, if `ctype == "int"` and `bit_width == 64`, you need 2 memory cells.

Returns Positive integer, a property's width.

class xentica.core.properties.**IntegerProperty**(*max_val*)
Bases: *xentica.core.properties.Property*

Most generic property for you to use.

It is just a positive integer with upper limit of `max_val`.

calc_bit_width()
Calculate bit width, based on `max_val`.

class xentica.core.properties.**ContainerProperty**
Bases: *xentica.core.properties.Property*

A property acting as a holder for other properties.

Currently is used only for inner framework mechanics, in particular, to hold, pack and unpack all top-level properties.

It will be enhanced in future versions, and give you the ability to implement nested properties structures.

Warning: Right now, direct use of this class is prohibited.

calc_bit_width()
Calculate bit width as sum of inner properties' bit widths.

declare_once(*init_val=None*)
Do all necessary declarations for inner properties.

Also, implements the case of off-board neighbor access.

Parameters **init_val** – Default value for the property.

deferred_write()
Pack state and write its value to VRAM.

This method is called from `BSCA` at the end of kernel processing.

set_bsca(*bsca*, *buf_num*, *nbr_num*)
Propagate BSCA setting to inner properties.

unpacked
Test if inner properties are unpacked from memory.

values()
Iterate over properties, emulating `dict` functionality.

2.1.4 Variables (xentica.core.variables)

The collection of classes to declare and use C variables and constants.

If the logic of your `emit()`, `absorb()` or `color()` functions requires the intermediate variables, you must declare them via classes from this module in the following way:

```
from xentica import core

class MyCA(core.CellularAutomaton):
    # ...

    def emit(self):
        myvar = core.IntegerVariable()
```

Then you can use them in mixed expressions, like:

```
myvar += self.neighbors[i].buffer.state
self.main.state = myvar & 1
```

You may also define constants or other `#define` patterns with *Constant* class.

class xentica.core.variables.**DeferredExpression**(*code=''*)

Bases: object

Base class for other classes intended to be used in mixed expressions.

In particular, it is used in base *Variable* and *Property* classes.

Most of the magic methods dealing with binary and unary operators, as well as augmented assigns are automatically overridden for this class. As a result, you can use its subclasses in mixed expressions with ordinary Python values. See the example in module description above.

Allowed binary ops `+`, `-`, `*`, `/`, `%`, `>>`, `<<`, `&`, `^`, `|`, `<`, `<=`, `>`, `>=`, `==`, `!=`

Allowed unary ops `+`, `-`, `~`, `abs`, `int`, `float`, `round`

Allowed augmented assigns `+=`, `-=`, `*=`, `/=`, `%=`, `<<=`, `>>=`, `&=`, `^=`, `|=`

class xentica.core.variables.**Constant**(*name*, *value*)

Bases: *xentica.core.mixins.BscaDetectorMixin*

The class for defining constants and `#define` patterns.

Once you instantiate *Constant*, you must feed it to `BSCA.define_constant()` in order to generate correct C code:

```
const = Constant("C_NAME", "bsca_var")
self._bsca.define_constant(const)
```

Parameters

- **name** – Name to use in `#define`.
- **value** – String, evaluating into `bsca.<value>`, second part of `#define`.

get_define_code()

Get the C code for `#define`.

name

Get the name of constant.

replace_value(*source*)

Replace the constant's value in generated C code.

Parameters **source** – Generated C code.

class xentica.core.variables.**Variable**(*val=None*)

Bases: *xentica.core.variables.DeferredExpression*, *xentica.core.mixins.BscaDetectorMixin*

Base class for all variables.

Most of the functionality for variables are already implemented in it. Though, you are free to re-define it all to implement really custom behavior.

Parameters **val** – Initial value for the variable.

code

Get the variable name as code.

var_name

Get variable name.

class xentica.core.variables.**IntegerVariable**(*val=0*)

Bases: *xentica.core.variables.Variable*

The variable intended to hold a positive integer.

var_type = 'unsigned int'

C type to use in definition.

2.1.5 Color Effects (xentica.core.color_effects)

The collection of decorators for `color()` method, each CA model should have.

The method should be decorated by one of the classes below, otherwise the correct model behavior is not guaranteed.

All decorators are get the `(red, green, blue)` tuple from `color()` method, then process it to create some color effect.

The minimal example:

```
from xentica import core
from xentica.core import color_effects

class MyCA(core.CellularAutomaton):

    state = core.IntegerProperty(max_val=1)

    # ...

    @color_effects.MovingAverage
    def color(self):
        red = self.main.state * 255
        green = self.main.state * 255
        blue = self.main.state * 255
        return (red, green, blue)
```

class xentica.core.color_effects.**ColorEffect**(*func*)

Bases: *xentica.core.mixins.BscaDetectorMixin*

Base class for other color effects.

You may also use it as standalone color effect decorator, it just doing nothing, storing calculated RGB value directly.

To create your own class inherited from *ColorEffect*, you should override `__call__` method, and place a code of color processing into `self.effect`. The code should process a values of `new_r`, `new_g`, `new_b` variables and store the result back to them.

The example:

```
class MyEffect(ColorEffect):

    def __call__(self, *args):
        self.effect = "new_r += 20;"
        self.effect += "new_g += 15;"
        self.effect += "new_b += 10;"
        return super(MyEffect, self).__call__(*args)
```

class xentica.core.color_effects.**MovingAverage**(*func*)

Bases: *xentica.core.color_effects.ColorEffect*

Apply the moving average to each color channel separately.

With this effect, 3 additional settings are available for you in `Experiment` classes:

fade_in The maximum delta by which a channel could *increase* its value in a single timestep.

fade_out The maximum delta by which a channel could *decrease* its value in a single timestep.

smooth_factor The divisor for two previous settings, to make the effect even smoother.

2.1.6 Renderers (xentica.core.renderers)

The collection of classes implementing render logic.

The renderer takes the array of cells' colors and renders the screen frame from it. Also, it is possible to expand a list of user actions, adding ones specific to the renderer, like zoom, scroll etc.

The default renderer is *RendererPlain*. Though there are no other renderers yet, you may try to implement your own and apply it to CA model as follows:

```
from xentica.core import CellularAutomaton
from xentica.core.renderers import Renderer

class MyRenderer(Renderer):
    # ...

class MyCA(CellularAutomaton):
    renderer = MyRenderer()
    # ...
```

class xentica.core.renderers.**Renderer**

Bases: *xentica.core.mixins.BscaDetectorMixin*

Base class for all renderers.

For correct behavior, renderer classes should be inherited from this class. Then at least *render_code()* method should be implemented.

However, if you are planning to add user actions specific to your renderer, more methods should be overridden:

- `__init__()`, where you expand a list of kernel arguments in `self.args`;
- *get_args_vals()*, where you expand the list of arguments' values;
- *setup_actions()*, where you expand a dictionary of bridge actions;

See *RendererPlain* code as an example.

get_args_vals(*bsca*)
Get a list of kernel arguments values.

The order should correspond to `self.args`, with the values themselves as either PyCUDA `GpuArray` or correct NumPy instance. Those values will be used directly as arguments to PyCUDA kernel execution.

Parameters **bsca** – `xentica.core.CellularAutomaton` instance.

render_code()
Generate C code for rendering.

At minimum, it should process cells colors stored in `col` GPU-array, and store the resulting pixel's value into `img` GPU-array. It can additionally use other custom arguments, if any set up.

setup_actions(*bridge*)
Expand bridge with custom user actions.

You can do it as follows:

```
class MyRenderer(Renderer):
    # ...

    @staticmethod
    def my_awesome_action():
        def func(ca, gui):
            # do something with ``ca`` and ``gui``
        return func

    def setup_actions(self):
        bridge.key_actions.update({
            "some_key": self.my_awesome_action(),
        })
```

Parameters **bridge** – `xentica.bridge.Bridge` instance.

class xentica.core.renderers.**RendererPlain**(*projection_axes=None*)
Bases: *xentica.core.renderers.Renderer*

Render board as 2D plain.

If your model has more than 2 dimensions, a projection over `projection_axes` tuple will be made. The default is two first axes, which corresponds to `(0, 1)` tuple.

static apply_move(*bsca*, **args*)
Apply field move action to CA class.

Parameters **bsca** – `xentica.core.CellularAutomaton` instance.

static apply_zoom(*bsca*, *dval*)
Apply field zoom action to CA class.

Parameters

- **bsca** – `xentica.core.CellularAutomaton` instance.
- **dval** – Delta by which field is zoomed.

get_args_vals(*bsca*)
Extend kernel arguments values.

static move(*dx*, *dy*)
Move over game field by some delta.

Parameters

- **dx** – Delta by x-axis.
- **dy** – Delta by y-axis.

render_code()
Implement the code for render kernel.

setup_actions(*bridge*)
Extend bridge with scroll and zoom user actions.

static zoom(*dzoom*)
Zoom game field by some delta.

Parameters **dzoom** – Delta by which field is zoomed.

2.1.7 Exceptions (xentica.core.exceptions)

The collection of exceptions, specific to the framework.

exception xentica.core.exceptions.**XenticaException**
Bases: Exception

Basic Xentica framework exception.

2.1.8 Mixins (xentica.core.mixins)

The collection of mixins to be used in core classes.

Would be interesting only if you are planning to hack into Xentica core functionality.

class xentica.core.mixins.**BscaDetectorMixin**
Bases: object

Add a functionlality to detect BSCA class instances holding current class.

All methods are for private use only.

2.2 The Topology (xentica.core.topology)

This package helps you build the topology for CA models.

All xentica.core.CellularAutomaton subclasses **must** have Topology class declared inside. This class describes:

- dimensions: the number of dimensions your CA model operates on.
- lattice: the type of lattice of your CA board. Built-in lattice types are available in *xentica.core.topology.lattice* module.
- neighborhood: the type of neighborhood for a single cell. Built-in neighborhood types are available in *xentica.core.topology.neighborhood* module.
- border: the type of border effect, e.g. how to process off-board cells. Built-in border types are available in *xentica.core.topology.border* module.

In example, you can declare the topology for a 2-dimensional orthogonal lattice with Moore neighborhood, wrapped to a 3-torus, as follows:

```
class Topology:

    dimensions = 2
    lattice = core.OrthogonalLattice()
    neighborhood = core.MooreNeighborhood()
    border = core.TorusBorder()
```

2.2.1 Lattice (xentica.core.topology.lattice)

The collection of classes describing different lattice topologies.

All classes there are intended to be used inside `Topology` for `lattice` class variable definition. They are also available via *xentica.core* shortcut. The example:

```
from xentica.core import CellularAutomaton, OrthogonalLattice

class MyCA(CellularAutomaton):
    class Topology:
        lattice = OrthogonalLattice()
        # ...
    # ...
```

class xentica.core.topology.lattice.**Lattice**

Bases: *xentica.core.topology.mixins.DimensionsMixin*, *xentica.core.mixins.BscaDetectorMixin*

Base class for all lattices.

For correct behavior, lattice classes should be inherited from this class. You should also implement the following functions:

- *index_to_coord_code()*
- *index_to_coord()*
- *coord_to_index_code()*
- *is_off_board_code()*

See the detailed description below.

coord_to_index_code(*coord_prefix*)

Generate C code for obtaining cell's index by coordinates.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters **coord_prefix** – The prefix for variables, containing coordinates.

Returns A string with C code calculating cell's index. No assignment, only a valid expression needed.

index_to_coord(*idx*, *bsca*)

Obtain cell's coordinates by its index, in pure Python.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters

- **idx** – Cell's index, a positive integer, or NumPy array of indices.

- **bsca** – xentica.core.CellularAutomaton instance, to access field size and number of dimensions.

Returns Tuple of integer coordinates, or NumPy arrays of coords per each axis.

index_to_coord_code(*index_name*, *coord_prefix*)
Generate C code to obtain coordinates by cell's index.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters

- **index_name** – The name of variable containing cell's index.
- **coord_prefix** – The prefix for resulting variables, containing coordinates.

Returns A string with C code, doing all necessary to process index variable and store coordinates to variables with given prefix.

is_off_board_code(*coord_prefix*)
Generate C code to test if the cell's coordinates are off board.

This is an abstract method, you must implement it in *Lattice* subclasses.

Parameters **coord_prefix** – The prefix for variables, containing coordinates.

Returns A string with C code testing coordinate variables. No assignment, only a valid expression with boolean result needed.

width_prefix = '_w'
The prefix to be used in C code for field size constants.

class xentica.core.topology.lattice.**OrthogonalLattice**
Bases: *xentica.core.topology.lattice.Lattice*

N-dimensional orthogonal lattice.

Points are all possible positive integer coordinates.

coord_to_index_code(*coord_prefix*)
Implement cell's index obtaining by coordinates in C.

See *Lattice.coord_to_index_code()* for details.

index_to_coord(*idx*, *bsca*)
Implement coordinates obtaining by cell's index in Python.

See *Lattice.index_to_coord()* for details.

index_to_coord_code(*index_name*, *coord_prefix*)
Implement coordinates obtaining by cell's index in C.

See *Lattice.index_to_coord_code()* for details.

is_off_board_code(*coord_prefix*)
Implement off board cell obtaining in C.

See *Lattice.is_off_board_code()* for details.

supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
Overridden value for supported dimensions.

2.2.2 Neighborhood (xentica.core.topology.neighborhood)

The collection of classes describing different neighborhood topologies.

All classes there are intended to be used inside `Topology` for `neighborhood` class variable definition. They are also available via *xentica.core* shortcut. The example:

```
from xentica.core import CellularAutomaton, MooreNeighborhood

class MyCA(CellularAutomaton):
    class Topology:
        neighborhood = MooreNeighborhood()
        # ...
    # ...
```

class xentica.core.topology.neighborhood.**Neighborhood**

Bases: *xentica.core.topology.mixins.DimensionsMixin*

Base class for all neighborhood topologies.

For correct behavior, neighborhood classes should be inherited from this class. You should also implement the following functions:

- *neighbor_coords()*
- *neighbor_state()*

See the detailed description below.

neighbor_coords(*index*, *coord_prefix*, *neighbor_prefix*)

Generate C code to obtain neighbor coordinates by its index.

This is an abstract method, you must implement it in *Neighborhood* subclasses.

Parameters

- **index** – Neighbor index, a non-negative integer less than number of neighbors.
- **coord_prefix** – The prefix for variables containing main cell's coordinates.
- **neighbor_prefix** – The prefix for resulting variables containing neighbor coordinates.

Returns A string with C code doing all necessary to get neighbor state from RAM. No assignment, only a valid expression needed.

neighbor_state(*neighbor_index*, *state_index*, *coord_prefix*)

Generate C code to obtain neighbor state by its index.

This is an abstract method, you must implement it in *Neighborhood* subclasses.

Parameters

- **neighbor_index** – Neighbor index, a non-negative integer less than number of neighbors.
- **state_index** – State index, a non-negative integer less than number of neighbors for buffered states or -1 for main state.
- **coord_prefix** – The prefix for variables containing neighbor coordinates.

Returns A string with C code doing all necessary to process neighbors's coordinates and store them to neighbor coordinates variables.

num_neighbors = None
Number of neighbors, you must re-define it in sub-classes.

topology = None
A reference to `Topology` holder class, will be set in `BSCA` metaclass.

class xentica.core.topology.neighborhood.**OrthogonalNeighborhood**
Bases: *xentica.core.topology.neighborhood.Neighborhood*

Base class for neighborhoods on orthogonal lattice.

It is implementing all necessary *Neighborhood* abstract methods, the only thing you should override is `dimensions()` setter. In `dimensions()`, you should correctly set `num_neighbors` and `_neighbor_deltas` attributes.

neighbor_coords(*index*, *coord_prefix*, *neighbor_prefix*)
Implement neighbor coordinates obtaining by its index, in C.

See *Neighborhood.neighbor_coords()* for details.

neighbor_state(*neighbor_index*, *state_index*, *coord_prefix*)
Implement state obtaining by neighbor/state index, in C.

See *Neighborhood.neighbor_coords()* for details.

supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
Any number of dimentions is supported, 100 is just to limit your hyperspatial hunger.

class xentica.core.topology.neighborhood.**MooreNeighborhood**
Bases: *xentica.core.topology.neighborhood.OrthogonalNeighborhood*

N-dimensional Moore neighborhood implementation.

The neighbors are all cells, sharing at least one vertex.

dimensions
Get a number of dimensions.

class xentica.core.topology.neighborhood.**VonNeumannNeighborhood**
Bases: *xentica.core.topology.neighborhood.OrthogonalNeighborhood*

N-dimensional Von Neumann neighborhood implementation.

The neighbors are adjacent cells in all possible orthogonal directions.

dimensions
Get a number of dimensions.

2.2.3 Border (xentica.core.topology.border)

The collection of classes describing different types of field borders.

All classes there are intended to be used inside `Topology` for `border` class variable definition. They are also available via *xentica.core* shortcut. The example:

```
from xentica.core import CellularAutomaton, TorusBorder

class MyCA(CellularAutomaton):
    class Topology:
        border = TorusBorder()
        # ...
    # ...
```

class xentica.core.topology.border.**Border**
Bases: *xentica.core.topology.mixins.DimensionsMixin*

Base class for all types of borders.

You should not inherit your borders directly from this class, use either *WrappedBorder* or *GeneratedBorder* base subclasses for this.

class xentica.core.topology.border.**WrappedBorder**
Bases: *xentica.core.topology.border.Border*

Base class for borders wrapping the field into different manifolds.

For correct behavior, you should implement *wrap_coords()* method.

See the detailed description below.

wrap_coords(*coord_prefix*)
Generate C code to translate off-board coordinates to on-board ones.

This is an abstract method, you must implement it in *WrappedBorder* subclasses.

Parameters **coord_prefix** – The prefix for variables containing cell's coordinates.

class xentica.core.topology.border.**GeneratedBorder**
Bases: *xentica.core.topology.border.Border*

Base class for borders generating states of the off-board cells.

For correct behavior, you should implement *off_board_state()* method.

See the detailed description below.

off_board_state(*coord_prefix*)
Generate C code to obtain off-board cell's state.

This is an abstract method, you must implement it in *GeneratedBorder* subclasses.

Parameters **coord_prefix** – The prefix for variables containing cell's coordinates.

class xentica.core.topology.border.**TorusBorder**
Bases: *xentica.core.topology.border.WrappedBorder*

Wraps the entire field into N-torus manifold.

This is the most common type of border, allowing you to generate seamless tiles for wallpapers.

supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
Any number of dimentions is supported, 100 is just to limit your hyperspatial hunger.

wrap_coords(*coord_prefix*)
Impement coordinates wrapping to torus.

See *WrappedBorder.wrap_coords()* for details.

class xentica.core.topology.border.**StaticBorder**(*value=0*)
Bases: *xentica.core.topology.border.GeneratedBorder*

Generates a static value for every off-board cell.

This is acting like your field is surrounded by cells with the same pre-defined state.

The default is just an empty (zero) state.

off_board_state(*coord_prefix*)
Impement off-board cells' values obtaining.

See *GeneratedBorder.off_board_state()* for details.

```
supported_dimensions = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
```

2.2.4 Mixins (xentica.core.topology.mixins)

The collection of mixins to be used in core classes.

Would be interesting only if you are planning to hack into Xentica core functionality.

class xentica.core.topology.mixins.**DimensionsMixin**

Bases: object

The base functionality for classes, operating on a number of dimensions.

Adds `dimensions` property to a class, and checks it automatically over a list of allowed dimensions.

allowed_dimension(*num_dim*)

Test if particular dimensionality is allowed.

Parameters **num_dim** – Numbers of dimensions to test

Returns Boolean value, either dimensionality is allowed or not.

dimensions

Get a number of dimensions.

supported_dimensions = []

A list of integers, containing supported dimensionality. You must set it manually for every class using *DimensionsMixin*.

2.3 The Seeds (xentica.seeds)

The package helping you to construct initial CA state (seed).

Classes from modules below are intended for use in `Experiment` classes.

For example to initialize the whole board with random values:

```
from xentica import core, seeds

class MyExperiment(core.Experiment):
    # ...
    seed = seeds.patterns.PrimordialSoup(
        vals={
            "state": seeds.random.RandInt(0, 1),
        }
   )
```

2.3.1 Patterns (xentica.seeds.patterns)

Module containing different patterns for CA seed initialization.

Each pattern class having one mandatory method `generate()` which is called automatically at the initialization stage.

Patterns are intended to use in `Experiment` classes. See the example of general usage above.

class xentica.seeds.patterns.**RandomPattern**(*vals*)
Bases: object

Base class for random patterns.

Parameters **vals** – Dictionary with mixed values. May contain descriptor classes.

generate(*cells*, *bsca*)
Generate the entire initial state.

This is an abstract method, you must implement it in *RandomPattern* subclasses.

Parameters

- **cells** – NumPy array with cells' states as items. The seed will be generated over this array.
- **cells_num** – Total number of cells in cells array.
- **field_size** – Tuple with field sizes per each dimension.
- **index_to_coord** – Function translating cell's index to coordinate.
- **pack_state** – Function packing state into single integer.

class xentica.seeds.patterns.**BigBang**(*vals*, *pos=None*, *size=None*)
Bases: *xentica.seeds.patterns.RandomPattern*

Random init pattern, known as *"Big Bang"*.

Citation from The Concept:

> *"A small area of space is initialized with a high amount of energy and random parameters per each quantum. Outside the area, quanta has either zero or minimum possible amount of energy. This is a good test for the ability of energy to spread in empty space."*

The current implementation allows to generate a value for every cell inside specified N-cube area. Cells outside the area have zero values.

Parameters

- **vals** – Dictionary with mixed values. May contain descriptor classes.
- **pos** – A tuple with the coordinates of the lowest corner of the Bang area.
- **size** – A tuple with the size of Bang area per each dimension.

generate(*cells*, *bsca*)
Generate the entire initial state.

See *RandomPattern.generate()* for details.

class xentica.seeds.patterns.**PrimordialSoup**(*vals*)
Bases: *xentica.seeds.patterns.RandomPattern*

Random init pattern, known as *"Primordial Soup"*.

Citation from The Concept:

> *"Each and every quantum initially has an equally small amount of energy, other parameters are random. This is a good test for the ability of energy to self-organize in clusters from the completely uniform distribution."*

The current implementation allows to populate the entire board with generated values.

Parameters **vals** – Dictionary with mixed values. May contain descriptor classes.

generate(*cells*, *bsca*)
Generate the entire initial state.

See *RandomPattern.generate()* for details.

class xentica.seeds.patterns.**ValDict**(*d*, *parent=None*)
Bases: object

Wrapper over Python dictionary.

It can keep descriptor classes along with regular values. Then, on the item getting, the necessary value is automatically obtaining either directly or via descriptor logic.

Readonly, you should set all dictionary values at the class initialization.

Example of usage:

```
>>> from xentica.seeds.random import RandInt
>>> from xentica.seeds.patterns import ValDict
>>> d = {'a': 2, 's': RandInt(11, 23), 'd': 3.3}
>>> vd = ValDict(d)
>>> vd['a']
2
>>> vd['s']
14
>>> vd['d']
3.3
```

Parameters

- **d** – Dictionary with mixed values. May contain descriptor classes.
- **parent** – A reference to class holding the dictionary. Optional.

items()
Iterate over dictionary items.

keys()
Iterate over dictionary keys.

2.3.2 RNG (xentica.seeds.random)

The module for package-wide RNG.

The main intention is to keep separate deterministic random streams for every *CellularAutomaton* instance. So, is you're initialized RNG for a particular CA with some seed, you're get the guarantee that the random sequence will be the same, no matter how many other CA's you're running in parallel.

class xentica.seeds.random.**LocalRandom**(*seed=None*)
Bases: object

The holder class for the RNG sequence.

It is incapsulating both standart Python random stream and NumPy one.

Once instantiated, you can use them as follows:

```
from xentica.seeds.random import LocalRandom

random = LocalRandom()
```

(continues on next page)

(continued from previous page)

```
# get random number from standard stream
val = random.std.randint(1, 10)
# get 100 random numbers from NumPy stream
vals = random.numpy.randint(1, 10, 100)
```

load(*rng*)
Load random state from the class.

Parameters **rng** – *LocalRandom* instance.

class xentica.seeds.random.**RandInt**(*min_val*, *max_val*)
Bases: object

Class, generating a sequence of random integers in some interval.

It is intended to be used in Experiment seeds. See the example of initializing CA property above.

Parameters

- **min_val** – Lower bound for random value.
- **max_val** – Upper bound for random value.

2.4 The Bridge (xentica.bridge)

The bridge between Xentica and GUI interface.

This package contains all necessary stuff to connect Xentica framework to custom interactive visualization environments.

Right now, only one environment (Moire) is available. This is the official environment, evolving along with the main framework. You are free to implement your own environments. If so, please make a PR on Github and we'll include your solution to the bridge.

Bridge functions are automatically used when you run the simulation like this:

```
import moire
ca = MyCellularAutomaton(MyExperiment)
gui = moire.GUI(runnable=ca)
gui.run()
```

2.4.1 Base (xentica.bridge.base)

This module contains the main class to be used in custom bridges.

Methods from *Bridge* class should be used in other bridges

class xentica.bridge.base.**Bridge**
Bases: object

Main bridge class containing basic functions.

static exit_app(*ca*, *gui*)
Exit GUI application.

static noop(*ca*, *gui*)
Do nothing.

static speed(*dspeed*)
Change simulation speed.

static toggle_pause(*ca*, *gui*)
Pause/unpause simulation.

static toggle_sysinfo(*ca*, *gui*)
Turn system info panel on/off.

2.4.2 Moire (xentica.bridge.moire)

Module with the bridge to Moire UI.

class xentica.bridge.moire.**MoireBridge**
Class incaplulating the actions for Moire UI.

`[` Speed simulation down.

`]` Speed simulation up.

`SPACEBAR` Pause/unpause simulation.

`F12` Toggle system info.

`ESC` Exit app.

2.5 Utilities (xentica.utils)

The collection of utilities.

2.5.1 Formatters (xentica.utils.formatters)

The collection of formatters.

xentica.utils.formatters.**sizeof_fmt**(*num*, *suffix=''*)
Format the number with the humanized order of magnitude.

In example, `11234` become `11.23K`.

Parameters

- **num** – The positive integer.
- **suffix** – Additional suffix to add to formatted string.

Returns Formatted number as a string.

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

x

Index

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z