The (yet-in) complete guide to Soya 3D

(also known as “the yet-in”)

LAMY Jean-Baptiste “Jiba”

January 20, 2008

Contents

Chapter 1

Introduction

1.1 License

The Yeti picture on the cover was drawn by Jean-noël Lafargue (jn (at) hyperbate (dot) com), and is available under the Free Art License (`http://artlibre.org/licence/lal/en/`).

1.2 What is Soya 3D ?

1.3 History

The actual Soya was originally written by Jiba and Blam. Jiba is the man that has written seven 3D engines:

1. a 3D engine ("Vertige 3D") in visual basic + direct X
2. a 3D engine in visual basic + OpenGL (direct X was too horrible)
3. a second 3D engine in visual basic + OpenGL (rewrite of the previous one)
4. a 3D engine ("Opale.Soya") in Java + OpenGL (my last engine was too big to compile with VB!)
5. a 3D engine ("Opale.Soya 2") in Java + OpenGL (rewrite of the previous one, with my brother Blam)
6. a 3D engine ("Soya" < 0.7) in Python + C + OpenGL (Python was now more appealing for me than Java, still helped by Blam for the C part)
7. a 3D engine ("Soya" >= 0.7) in Python + Pyrex + OpenGL (I was not at ease with C and Blam was gone)

The engine in visual basic have never been published (I haven't got the Internet at this time!). The Java engine have been published and can still be found on the web; in particular they were used in Arkanae. They are no longer maintained.

Then, after trying an aggressive take-over on Soya, Arc Riley has forked the project into PySoy, on June the 6th in 2006. Arc still own the soya3d.org domain name, and use it as a placeholder spreading false and slanderous allegations.

Arc was considering me (=Jiba) as a "bad leader" for the project. After having written the Soya 3D engine and several games (including Arkanae, Slune, Balazar and Balazar Brothers), I consider that I am definitevely a great leader and that the technical directions I have chosen are right. I don't claim being a good manager, though ;-)

You can find more historical information on `http://home.gna.org/oomadness/en/soya/history.html`.

1.4 The Soya project objectives and the Soya spirit

Here are the spirit that Soya follows (or *should* follow ;-)):

- **Soya's goal #1 is to allow to develop as rapidly as possible** 3D games and other 3D apps
  (Rationale: Soya targets "amateur" developpers, who code during their limited free time! Moreover, rapid development allows to gain a precious time that can be re-invested in improving your code or testing your app. For game, it leads to a better gameplay)
- **Soya's goal #2 is to be as easy as possible to learn**, in particular for people with no 3D background at all
  (Rationale: Soya is not a toy; to be easy to learn is only goal #2 since any newbie will, a day, not longer be a newbie)

- **Despite its simplicity, Soya never sacrifices performance and speed**
  (Rationale: speed is important for 3D game!)
- **Soya's API does not necessarily fit to the mathematical, computational or technical reality of the 3D**, e.g. Soya API does not require the use of matrices
  (Rationale: math are not the natural way to represent 3D object. Soya should be useable without an important mathematical background. However, Soya provides matrices, mainly for debug purpose)
- **Soya relies a lot on Python facilities and modules**, e.g. saving 3D model is done through object serialization. As a consequence, Soya won't evolve toward a multi-language 3D engine and will stay Python-centred
  (Rationale: Python modules ease the Soya development, but also the use of Soya, since they are already well-known to Python developers)
- **Soya has plenty of dependencies**
  (Rationale: any good OS has a package system tools today)
- **Soya always assumes by default the most common usecase**
  (Rationale: doing so lead to a substencial time gain)
- **Soya is an "atypic" 3D engine and relies on a certain number of controversal choices** that are somehow debatable ; however, most of these choices are deliberate
  (Rationale: Soya should be seen as a "research project" aiming at "a new way for 3D" ; the objective is not to satisfy anyone, but to be the ideal 3D engine for a few persons. Python does similarly)

1.5 Documentation available

1.5.1 Docstrings

About half of Soya objects and functions have docstrings. You can use pydoc to read them, or just type *e.g.* "help(soya.Body)" in a Python interpreter. You can also browse the doc online at `http://home.gna.org/oomadness/en/soya/pydoc.html`.

1.5.2 Tutorials

The Soya tutorial pack includes many tutorials, demos and examples.

1.5.3 About this doc

The (yet-in)complete guide to Soya 3D, also known as "the yet-in", is still under writing.

1.6 Getting help

You may ask for help either on the Soya's mailing list (soya-user@gna.org, suscribe from `http://mail.gna.org/listinfo/soya-user`) or the #soya IRC channel on FreeNode. Please check if you cannot find the reply to your question in the documentation listed above, though.

Chapter 2

Soya's basics

2.1 Initializing Soya

Initializing Soya is done in three steps, corresponding to these three lines:

```
import soya
soya.path.append("/your/data/path")
soya.init("My 3D app", sound = 1)
```

1. Importing module. Soya has several Python sub-packages, but most of the basic stuff is directly in the soya module.
2. Setting data path. The data directory (`/your/data/path` above) is referred as <data>/ in this documentation; it is expected to contain several subdirectories (see section 3).

   Hint: if your data are in the "data" directory located in the same directory than your script, a common trick for Python script is:

   ```
   soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))
   ```

   and for Python module:

   ```
   soya.path.append(os.path.basename(__file__, "data"))
   ```

3. Creating and showing the 3D display. `soya.init` can take the following arguments (all being optional):

   title is the title of the window (windowed mode only, defaults to "Soya 3D").

   width, height the dimensions of the 3D screen (default to 640, 480).

   fullscreen is true for fullscreen and false for windowed mode (defaults to false).

   resizeable is true for a resizeable window (windowed mode only, defaults to true).

   create_surface is true for creating an OpenGL surface through SDL, and false for using whatever OpenGL that is currently active (in this case it is up to you to initialize OpenGL, *e.g.* with PyGame, see section 13.2; it defaults to true)

   sound is true to initialize 3D sound support (default to false for backward compatibility). There are other sound-related arguments, which are discussed in section 7.3.

You can use soya.set_video(width, height, fullscreen, resizable) to change some of these parameters after initialization.

The rest of this documentation assume that you have initialized Soya correctly. It may also assume that you have imported some common modules (os, sys,...), well, you are probably enough intelligent to understand that ;-).

2.2 Basic classes

The UML schema of figure 2.1 shows Soya's basic classes:

MainLoop is in charge of managing and regulating time (see section 2.7).

CoordSyst is the base class for all 3D objects. It defines a coordinate system, *i.e.* it has a 3D position, orientation and size.

Light is a light.

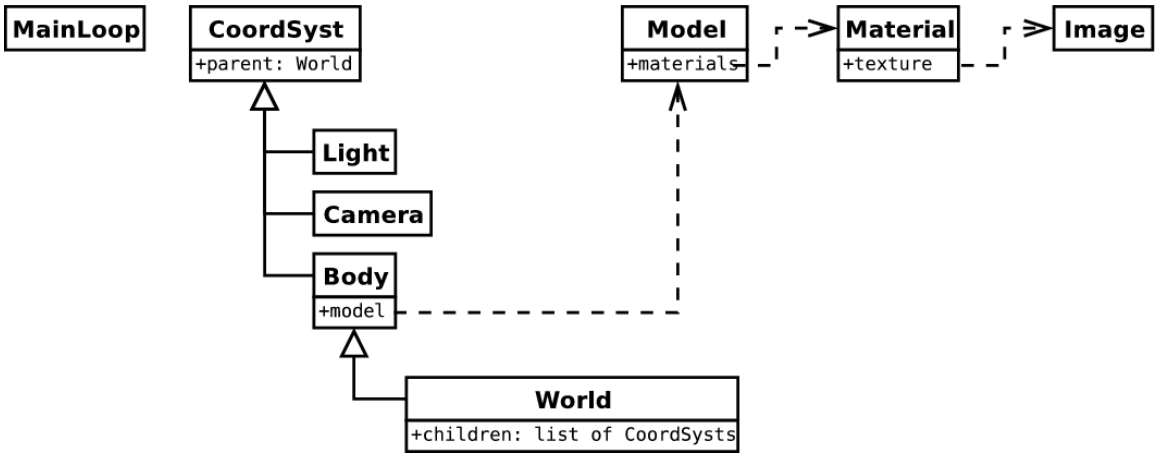


Figure 2.1: Soya's basic classes (UML schema)

Camera is the "eye" from which the 3D scene is viewed. It also acts as the "ear", for 3D sound.

Model (sometimes called "Mesh" in other 3D engines) is a 3D model. Model is actually an "abstract" class, and Soya provides several Model subclasses (SimpleModel, AnimatedModel,...). Models are created either by exporting them from 3D modelers (see chapter 5) or by creating a World, putting Faces in the World and then "compiling" the World into a Model (see chapter 10).

Material defines the attributes of a surface, *e.g.* color or texture.

Image is a 2D image. It is used in particular for textures. Images are usually created using a 2D painting program like The Gimp.

Body (sometimes called "Entity" in other 3D engines, or "Object" in Blender) displays a Model at a specific 3D position. Model cannot be displayed without "emBodying" them; it allows to display the same Model at several location (*e.g.* two identical houses in a town), by creating two Bodies with the same Model.

World (sometimes called "Node" or "Group" in other 3D engines) acts as a grouping container. A World can contain other nested CoordSysts, including other Worlds. When a World is moved, all the CoordSysts it contains are moved too. As a consequence, Soya scenegraph is a tree structure, the root being a World, usually called "scene". World also inherit from Body, and thus can display a Model.

Most of the more advanced Soya classes derive from these.

Hint: some people find odd that World inherits from Body... but this will make sense in section 4.5, please wait ;-)

History: For a long time, Soya has used "folkloric" names. These names are still available as aliases, for backward compatibility (and archeologists :-). They are: Idler for MainLoop, Shape for Model, Volume for Body, Land for Terrain (see section 9.1).

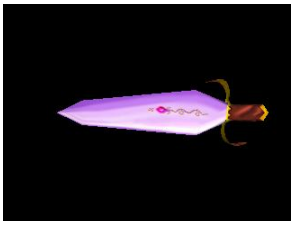
2.3 Your first 3D scene with Soya

We are going to create a basic 3D scene that just displays a Model. First, we need to create the root of the tree, a World we call "scene" (Worlds with no parent are usually called "scene"). Then we load the Model and create a Body that displays it. Then we create a Light and a Camera, and we set the Camera as the *root widget*, *i.e.* the object Soya renders (see chapter 11). Finally, we create the MainLoop and start looping. The scene tree is the following:

```
World scene
  |
  +-- Body sword, displaying the sword Model
  |
  +-- Light light
  |
  +-- Camera camera
```

And here is the code (see tutorial basic-1.py):

```
scene = soya.World()

sword_model = soya.Model.get("sword")

sword = soya.Body(scene, sword_model)
sword.x = 1.0
sword.rotate_y(90.0)

light = soya.Light(scene)
light.set_xyz(0.5, 0.0, 2.0)

camera = soya.Camera(scene)
camera.z = 2.0

soya.set_root_widget(camera)
soya.MainLoop(scene).main_loop()
```



In the next sections, we are going to see in more details the various objects used in this first example.

2.4 Loading Images, Materials and Models

Models are usually not created but loaded from a file (if you want to create Models directly from Soya, see chapter 10). To load a Model, do:

```
your_model = soya.Model.get("your_model_filename")
```

and Soya loads the <data>/models/your_model_filename.data file. The Materials and Images used by the Model are also automatically loaded, from the <data>/materials/ and <data>/images/ directories. Image files are PNG or JPEG, and Material and Model files are raw serialized Python objects, a format that only Soya can read or write.

However, Soya can import Blender models automatically. If a <data>/blender/your_model_filename.blend file exist, and the Model file doesn't exist (or is older), Soya loads the Blender files and caches the resulting Model in <data>/models/your_model_fi This feature is known as "auto-exporters"; for more details on auto-exporters or data management, see chapter 3, and for more details on using Blender with Soya, see chapter 5.

If you call Model.get several times with the same filename argument, Soya doesn't load the Model twice but returns the same (cached) object. This is nice since Models are immutable.

2.5 Displaying Models with Bodies

Models cannot be directly displayed on the screen; you need to put them onto a Body. You can create a Body as following:

```
your_body = soya.Body(parent, model)
```

where parent is the World in which the Body is added, and model is the Model to display (both default to None). For all constructors of 3D objects (derivating from CoordSyst), the first argument is the parent World. It is possible to reparent a CoordSyst, and the current parent World can be accessed through the parent attribute.

The Model displayed by a Body can be get or set through the model attribute:

```
your_body.model = soya.Model.get("your_second_model_filename")
```

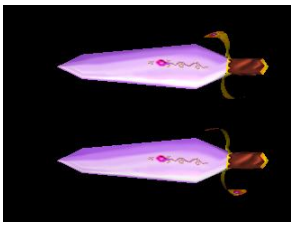
Bodies allow to display several times the same Model. For example, to display two swords instead of one, you can use the following scene tree:

```
World scene
  |
  +-- Body sword1, displaying the sword Model
  |
```

```
+-- Body sword2, displaying the sword Model
|
+-- Light light
|
+-- Camera camera
```

And here is the code (set_xyz and rotate_y are used to position the two Bodies and will be seen in the next section; the code snipplet doesn't include the scene, Light, Camera and MainLoop, which are identical to the "first scene" example):

```
sword_model = soya.Model.get("sword")

sword1 = soya.Body(scene, sword_model)
sword1.set_xyz(1.0, 0.5, 0.0)
sword1.rotate_y(90.0)

sword2 = soya.Body(scene, sword_model)
sword2.set_xyz(1.0, -0.5, 0.0)
sword2.rotate_y(90.0)
```



2.6 Moving, rotating and scaling CoordSysts

Moving, rotating and scaling methods belong to the CoordSyst class, and are inherited to all CoordSyst children classes. This section shows the most common Moving, rotating and scaling methods; for a complete reference see section 2.13.3.

2.6.1 Soya's conventions

- When relevant, Soya always considers X as the right-hand direction, Y as the up direction, Z as the backward direction, and thus -Z as the frontward direction (Soya uses -Z for front in order to keep all coordinate systems right-handed, just to avoid an internal mathematical nighmare).
- All angles are expressed in degrees.
- In Soya, a 3D position is defined by three X, Y, Z values **and** the CoordSyst in which they are expressed. X, Y, Z alone are not enough to make a 3D position.
- By default, a distance of 1.0 is considered as roughly one meter, although you may choose a different convention.

2.6.2 Moving

The most basic method for moving an object is to set his x, y or z attributes. set_xyz sets x, y and z in a single call.

```
coord_syst.x = 1.0
coord_syst.set_xyz(1.0, 2.0, 3.0) # Set x to 1.0, y to 2.0 and z to 3.0
```

When using methods that take object arguments (and not raw X, Y and Z values), Soya automatically performs coodinate system conversion if needed. To move a CoordSyst at the same position than another one, use the move method:

```
coord_syst.move(coord_syst2)
```

Translation can be done by the add_vector method (which is aliased to the += operator). The vector constructor accepts the parent (any CoordSyst), and then the X, Y and Z coordinates. For example, to move coord_syst one step on its right:

```
coord_syst.add_vector(soya.Vector(coord_syst, 1.0, 0.0, 0.0))
```

And to move coord_syst one step on the scene's right:

```
coord_syst.add_vector(soya.Vector(scene, 1.0, 0.0, 0.0))
```

Finally, add_mul_vector(proportion, vector) is a faster equivalent to add_vector(proportion * vector), which often used in advance_time.

2.6.3 Rotating

rotate_x(angle) (aliased to rotate_vertical), rotate_y(angle) (aliased to rotate_lateral) and rotate_z(angle) (aliased to rotate_incline) perform rotation around the CoordSyst's **parent** X, Y and Z axes. The turn_* methods (turn_x, turn_lateral,...) are identical but they refer to the CoordSyst **local** X, Y and Z axes, and not its parent ones. All angles are in degrees.

```
coord_syst.rotate_y(90.0)
```

rotate_axis(angle, axis) performs a rotation around an axis defined by the origin (0, 0, 0) and the Vector axis.

```
coord_syst.rotate_axis(90.0, soya.Vector(scene, 0.0, 1.0, 0.0))
```

rotate(angle, a, b) performs a rotation around an axis that pass through a and b (either CoordSysts or Points).

```
coord_syst.rotate(90.0, scene, soya.Point(scene, 0.0, 1.0, 0.0))
```

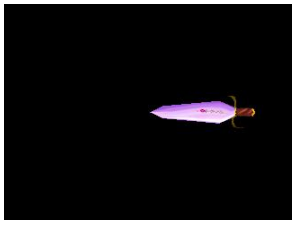
Finally, the very handy look_at(direction) method rotates a CoordSyst so as it looks toward direction (either a Vector, a Point or a CoordSyst); the -Z direction is considered as "front", and look_at tries to maintain the Y direction as up (which is usually what one expects):

```
arrow.look_at(enemy)
```

2.6.4 Scaling

To scale a CoordSyst, use the scale method, which accepts three arguments, the X, Y and Z scale factors. Negative values can be used for mirroring.

```
coord_syst.scale(0.5, 0.5, 0.5)
```



The scale_x, scale_y and scale_z attributes are the current X, Y and Z scale factors (*e.g.* 0.5 in the previous example).

2.7 Time management and the MainLoop

In Soya, the time is divided in *round*, each round having the same theorical duration (by default, 30 milliseconds). "Theorical duration" means that a given round may be shorter or longer, but the mean duration is constant. The following three methods of CoordSysts are automatically called as time goes on (see figure 2.2):

begin_round() is called at the beginning of each round, for each CoordSyst. begin_round may *e.g.* perform collision detection, read events, determine the CoordSyst next move, and compute a speed vector.

advance_time(proportion) is called in proportion as time goes on; the proportion agument is the proportion of the round that has passed (1.0 for a complete round, 0.5 for halt a round,... during a round, the sum of the proportion arguments in the different calls to advance_time, is always 1.0). advance_time is in charge of *e.g.* applying the speed vector computed by begin_round.

end_round() is called at the end of the round. It is rarely used.

The 3D rendering may occurs at any instant, possibly in the middle of a round, as if it was simultaneous (although Soya uses a single thread). This time managing system yields a very smooth and soft animation: *e.g.* if two third of a round has passed, two third of the movement will be done when the rendering occurs.

MainLoop is responsible for cutting the time as exposed above, and it does a good job. The MainLoop object constructor accpets one (or more) World arguments, which are the root scenes. Then the MainLoop is started by calling MainLoop.main_loop, and it loops until you call MainLoop.stop(arg); arg will be the value returned by MainLoop.main_loop (see section 2.13.1 for more details on MainLoop).

For example, here is a subclass of Body that rotates continuously over the Y axis (see tutorial basic-2.py):

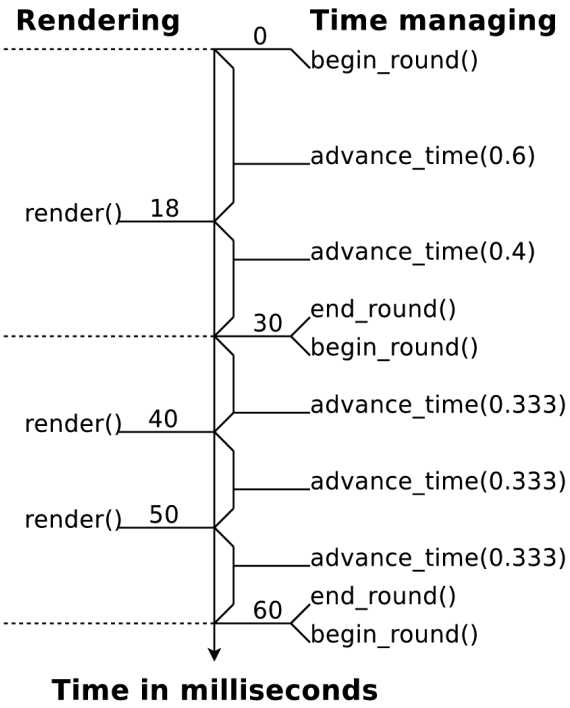
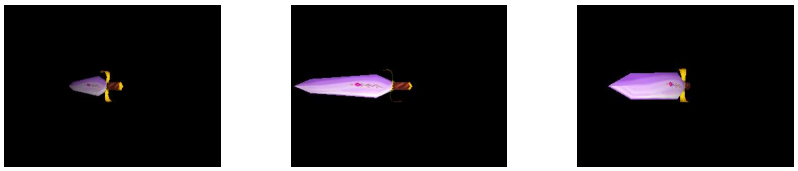


Figure 2.2: Time division in Soya

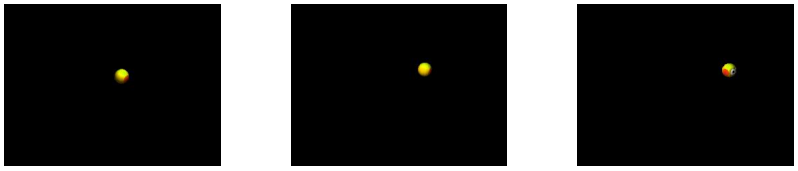
```
class RotatingBody(soya.Body):
    def advance_time(self, proportion):
        soya.Body.advance_time(self, proportion)

        self.rotate_y(proportion * 5.0)
```



As the rotation is always the same, we don't need a begin_round. The advance_time method calls the super implementation, and then rotates the object. Notice how the rotation angle takes into account the proportion argument (*e.g.* if half of the round has passed, half of the rotation is performed).

A more complex example is a randomly moving Body (see tutorial basic-3.py):

```
class RandomlyMovingBody(soya.Body):
    def __init__(self, parent = None, model = None):
        soya.Body.__init__(self, parent, model)
        self.rotation_speed = 0.0
        self.speed = soya.Vector(self, 0.0, 0.0, -0.2)

    def begin_round(self):
        soya.Body.begin_round(self)
        self.rotation_speed = random.uniform(-25.0, 25.0)

    def advance_time(self, proportion):
        soya.Body.advance_time(self, proportion)
        self.rotate_y(proportion * self.rotation_speed)
        self.add_mul_vector(proportion, self.speed)
```



This RandomlyMovingBody has too additional attributes: the rotation speed (in degrees, around the Y axis), and the speed vector. The speed vector is expressed in the RandomlyMovingBody coordinate system itself (remind that the -Z direction is the front).

begin_round computes a new random rotation speed (from -25.0 to 25.0); the speed vector doesn't need update since it is expressed in the RandomlyMovingBody coordinate system (*i.e.* rotating or moving the RandomlyMovingBody will rotate or move the vector). advance_time applies the rotation and speed vector; self.add_mul_vector(propotion, self.speed) is equivalent to self.add_vector(proportion * self.speed), but faster.

2.8 Grouping objects in Worlds

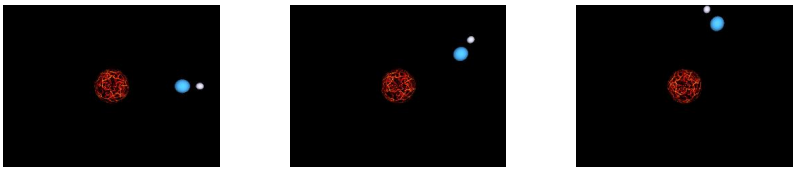
A World is a Body that can also have children CoordSysts nested in it (including other Worlds). When the World is moved, rotated or scaled, all the children CoordSysts are moved, rotated or scaled. For example, we can use nested World for representing celestial objects like suns/stars, planets and satellites. In this example, CelestialObject inherits from World, and continuously rotates (similarly to the RotatingBody example we've seen previously), but it can also contain other CoordSyst. As the CelestialObject rotates, the CoordSysts it contains are moved too.

For three CelestialObjects, the sun, the earth and the moon, the scene tree is:

```
World scene
  |
  +-- CelestialObject sun, displaying the sun Model
  |     |
  |     +-- CelestialObject earth, displaying the earth Model
  |           |
  |           +-- CelestialObject moon, displaying the moon Model
  |
  +-- Light light
  |
  +-- Camera camera
```

And here is the code (see tutorial nested-worlds-1.py):

```
class CelestialObject(soya.World):
    def advance_time(self, proportion):
        soya.World.advance_time(self, proportion)
        self.rotate_y(proportion * 2.0)

sun   = CelestialObject(scene, soya.Model.get("sun"))

earth = CelestialObject(sun  , soya.Model.get("earth"))
earth.x = 2.0

moon  = CelestialObject(earth, soya.Model.get("moon"))
moon.x = 0.5

camera = soya.Camera(scene)
camera.y = 4.0
camera.look_at(soya.Vector(scene, 0.0, -1.0, 0.0)) # Looks downward
soya.set_root_widget(camera)
```



Notice that the moon position (X = 0.5) is relative to its parent coordinate system (the earth), and not to the scene.

The easiest way to add a CoordSyst into a World is to pass the World as the first argument to the CoordSyst's constructor (as done previously):

```
earth = CelestialObject(sun, soya.Model.get("earth"))
```

However, it is also possible to use the add method:

```
earth = CelestialObject(None, soya.Model.get("earth"))
sun.add(earth)
```

Then, CoordSysts can be removed by the remove method:

```
sun.remove(earth)
```

The CoordSyst.parent attribute is the World currently containing the CoordSyst; this attribute is read-only (use remove and add to reparent a CoordSyst):

```
if moon.parent is earth: print "it's OK"
```

The World.children attribute is a list containing all the children CoordSysts (you **should not modify this list** directly, use add and remove!).

```
print earth in sun.children # => true
print moon  in sun.children # => false
```

Worlds can also be iterated as list:

```
for coord_syst in sun:
    print "the sun contains", coord_syst
```

Soya also provide handy recursive methods. World.recursive() returns all the nested CoordSysts, recursively, and CoordSyst.is_inside(World) can be used to check recursively for inclusion:

```
print earth.is_inside(sun) # => true
print moon .is_inside(sun) # => true
```

World also has various methods for searching children, using predicate or the CoordSyst.name attribute. For example, to search (recursively) the scene for all Bodies that display the moon Model:

```
scene.search_all(lambda coord_syst: isinstance(coord_syst, soya.Body)
                                and coord_syst.model is soya.Model.get("moon"))
```

Finally, Worlds can be turned into Model using the to_model() method. Models are faster than Worlds, but they are immutable (this will be discussed in detail in chapter 10).

2.9 Math computation: Point

As stated above, Soya defines a 3D position by three X, Y, Z values **and** the CoordSyst in which they are expressed. Soya provides Point objects for encapsulating the X, Y, Z coordinates and the CoordSyst. Using Points, Soya automatically converts the X, Y, Z coordinates from a CoordSyst to another when needed.

A Point represent a 3D position. To create a Point, use the constructor:

```
soya.Point(CoordSyst, x, y, z) -> Point
```

For example, this creates the Point located at X=1.0, Y=0.0, Z=0.0 in the earth (using the solar system example previously seen).

```
soya.Point(earth, 1.0, 0.0, 0.0)
```

The constructors are very similar to the CoordSyst's (and subclasses') ones, the first argument being the parent object. However, Point can be created in any CoordSyst, and not only World, and they are not considered as "3D objects". For example, they are not listed in World.children:

```
world = soya.World()
point = soya.Point(world, 1.0, 0.0, 0.0)
print world.children # => [] (empty list)
```

The distance between two CoordSysts or Points can be computed by the distance_to method:

```
print moon.distance_to(sun) # Notice that moon and sun are not defined in the same CoordSyst
```

Point provides the same moving methods than CoordSyst (see 2.6.2).

The parent attribute of a Point can be used to get or set the CoordSyst in which the X, Y, Z values are defined. If parent is set, X, Y, Z are left unmodifed; if you want to perform a manual coordinate system conversion, use the convert_to method:

```
moon_center = soya.Point(moon, 0.0, 0.0, 0.0)
moon_center.convert_to(sun)
print "In the sun coordinate system, the center of the moon is", moon_center
# => In the sun coordinate system, the center of the moon is <Point 1.98, 0.0, 0.5
#    in <CelestialObject, model=<SimpleModel sun>>>
```

The % operator performs a similar conversion, but not in place (*i.e.* without modifying the object) : obj % coord_syst means obj expressed in the coord_syst coordinate system, *i.e.* either obj itself if obj.parent is already coord_syst, or a newly created Point. The following is thus equivalent to the previous example:

```
print "In the sun coordinate system, the center of the moon is", moon % sun
```

2.10 Math computation: Vector

A Vector represent a 3D vector; it can be used for translation or angular computation.

Hint: Vector inherits from Point, only to avoid dupplicating some internal code, although this inheritance relation is mathematically absurd ;-).

To create a Vector, you can use either the constructor, or the vector_to method, which create a vector from a beginning position and an end position (both being CoordSyst or Point):

```
soya.Vector(CoordSyst, x, y, z) -> Vector
CoordSyst_or_Point.vector_to(CoordSyst_or_Point) -> Vector # Aliased to the >> operator
```

For example, to move character one unit forward, we first create the speed vector, which is Z=-1 in the character CoordSyst (-Z being the front direction in Soya convention), and then we use add_vector:

```
speed = soya.Vector(character, 0.0, 0.0, -1.0)
character.add_vector(speed)
```

Like Point, Vector are not considered as "3D objects" and, in particular, they are not listed in World.children. Vector provides the same moving methods than CoordSyst (see 2.6.2).

The length method returns the length of the Vector; you can use the set_length(new_length) method to scale the Vector to the given length. The following example moves the moon 1.0 unit toward the sun (remind that add_vector perform a translation):

```
vector = moon.vector_to(sun)
vector.set_length(1.0)
moon.add_vector(vector)
```

Vector provides also some methods for dealing with rotations and angles. The angle_to method returns the angle between two Vectors; the following example computes the angle between the sun and the moon, at the earth position:

```
print (earth >> sun).angle_to(earth >> moon) # >> is the same than vector_to
```

For Vector, add_vector performs a vectorial addition (since translating a vector doesn't change it and is thus useless). The dot_product and cross_product methods compute what you can expect from them.

Hints: Creating many Points or Vectors is time-consuming, since they are Python object. You can increase the performance by re-using the same objects instead of creating new ones.

2.11 The eye: Camera

2.12 Enlight your scene: Light

In Soya, colors are always four-value tupples of the form (red, green, blue, alpha). Each component ranges from 0.0 to 1.0. The alpha component is the transparency (0.0 means fully transparent, 1.0 means fully opaque). The alpha component is always mandatory, even if it is not used (*e.g.* for Light colors).

2.13 Basic object reference

2.13.1 MainLoop

MainLoop is in charge of managing and regulating time (see section 2.7). The current running MainLoop can be accessed as soya.MAIN_LOOP.

Constructor:

MainLoop(scene1, scene2,...) -> MainLoop

Noticeable attributes are:

fps the frame rate (number of frame per second, a usefull speed indicator; read only).

running true if the MainLoop is running (read only).

next_round_tasks a list of callable (taking no arg) that will be called once, just after the beginning of the next round. You can add callable if you want.

scenes the Worlds associated to the MainLoop. These Worlds begin_round, advance_time and end_round will be called regularly (as well as the ones of all objects inside the Worlds, of course).

round_duration the duration of a round, in second. Round is the time unit. It is granted that **all** rounds correspond to a period lasting round_durection (though the different period may not be regularly spread over time). (default to 0.030, *i.e.* 30 milliseconds).

min_frame_duration: minimum duration for a frame. This attribute can be used to limit the maximum FPS to save CPU time; *e.g.* having FPS higher than 30-40 is usually useless. Default is 0.020, which limits FPS to 40 in theory and to about 33 in practice (I don't know why there is a difference between theory and practice !).

Noticeable methods are:

main_loop() -> return_value starts the MainLoop. This method returns only after MainLoop.stop is called, and it returns the argument given to MainLoop.stop.

stop(return_value=None) stops the main loop. The stop doesn't occur immediately, but at the end of the next iteration. MainLoop.stop causes MainLoop.main_loop to returns; return_value is the (optionnal) value that MainLoop.main_loop will return.

reset() rester the internal time counter. You need to call MainLoop.reset if your program has paused, and you don't want the MainLoop to compensate the time loss by accelerating. This is usually the case when you put a game in pause mode.

update() calling regularly MainLoop.update is an alternative to MainLoop.start (see section 13.1).

begin_round()

advance_time(proportion)

end_round() default implementation calls all the corresponding methods of all scenes in the MainLoop.

render() called when it is time to render; default implementation calls soya.render that does the job.

2.13.2 Model

Model is a 3D model (sometimes called "Mesh" in other 3D engines). Model is actually an "abstract" class, and Soya provides several Model classes (SimpleModel, AnimatedModel,...). Models are created either by exporting them from 3D modelers (see chapter 5) or by creating a World, putting Faces in the World and then "compiling" the World into a Model (see chapter 10). Models are considered as immutable, as a single model can be shared and used by several Body; if you want to modify a Model in your code, you'll have to modify the World that has generated it, and then to turn the World into a new Model.

Models are not created by calling the Model constructor directly, they are loaded from a file or created from a World, respectively:

Model.get("filename") -> Model (see section 3)

World.to_model() -> Model

Noticeable attributes are:

filename the name of the file the model was loaded from (without path or extension).

materials a tuples of the materials the model uses.

2.13.3 CoordSyst

CoordSyst is the base class for all 3D objects. It defines a coordinate system, *i.e.* it has a 3D position, orientation and size. Constructor is:

CoordSyst(parent=None) -> CoordSyst where parent is the World in which the CoordSyst will be added (use None for no addition).

Noticeable attributes are:

parent the World that contains this CoordSyst (None if no such parent; read-only, use World.remove and World.add to reparent a CoordSyst).

x, y, z the X, Y and Z coordinates (defaults to 0.0, 0.0, 0.0).

scale_x, scale_y, scale_z the X, Y and Z scaling factors.

visible if false, the object is not displayed (defaults to true).

solid if false, the object is not taken into account for collision and raypicking (see section 8, defaults to true).

static if true, the object is considered as static (doesn't move), and Soya take that into account for optimizing rendering (default to false, may be modified automatically due to auto_static, see below).

auto_static if true, Soya automatically determines and sets the static attribute (defaults to true).

matrix, root_matrix, inverted_root_matrix the underlying 4x4 matrix, the root matrix (*i.e.* the multiplication of all matrices from scene.matrix up to CoordSyst.matrix), and the inverse of the root matrix (for debugging or hacking purpose only).

left_handed true if the CoordSyst is left_handed (read only, for debugging or hacking purpose only).

Noticeable methods are:

get_root() -> World get the root parent of the CoordSyst (the scene).

is_inside(coord_syst) returns true if the CoordSyst is inside coord_syst, *i.e.* both CoordSysts are the same, or coord_syst is a World that (recursively) contains the CoordSyst.

position() -> Point creates a Point in the same parent and at the same place than the CoordSyst.

distance_to(position) -> float returns the distance between the CoordSyst and position (another CoordSyst or a Point).

vector_to(position) -> Vector creates a Vector that starts at the CoordSyst position, and ends at the given position (another CoordSyst or a Point; aliased to the >> operator).

set_identity() resets the CoordSyst position, orientation and scaling.

get_sphere() -> (Point, float) returns a sphere (defined by the center Point, and the radius) that includes all elements in the CoordSyst.

get_box() -> (Point, Point) returns a sphere (defined by two corners) that includes all elements in the CoordSyst.

interpolate(state1, state2, factor) moves, rotates and scales the CoordSyst by interpolating between the two CoordSystStates state1 and state2. factor indicates the weight of the two CoordSystStates (0.0 means state1, 1.0 state2, and 0.5 half-way). XXX details interpolation in an other chapter; this feature is not yet stable.

set_xyz(x, y, z) set the x, y and z attributes in a single call.

move(position) moves the CoordSyst at the same place than position (another CoordSyst or a Point).

add_vector(vector) translates the CoordSyst by the given Vector (aliased to the += operator) .

add_mul_vector(k, vector) translates the CoordSyst by k times the given Vector (equivalent to, but faster than, add_vector(k * vector)).

add_xyz(x, y, z) translates the CoordSyst by (x, y, z) (expressed in the CoordSyst's parent coordinate system).

shift(x, y, z) translates the CoordSyst by (x, y, z) (expressed in the CoordSyst coordinate system).

rotate_x(angle), rotate_vertical(angle) rotates arround the CoordSyst's parent X axis (like of you rotate the head vertically).

rotate_y(angle), rotate_lateral(angle) rotates arround the CoordSyst's parent Y axis (like of you rotate the head laterally).

rotate_z(angle), rotate_incline(angle) rotates arround the CoordSyst's parent Z axis(like of you roll the head).

turn_x(angle), turn_vertical(angle) rotates arround the CoordSyst local X axis.

turn_y(angle), turn_lateral(angle) rotates arround the CoordSyst local Y axis.

turn_z(angle), turn_incline(angle) rotates arround the CoordSyst local Z axis.

rotate(angle, a, b) rotates around the axis defined by the a and b position (CoordSysts or Points).

rotate_axis(angle, axis) rotates around the axis defined by the origin (0, 0, 0) and the Vector axis.

rotate_xyz(angle, a_x, a_y, a_z, b_x, b_y, b_z) rotates around the axis defined by (a_x, a_y, a_z) and (b_x, b_y, b_z).

rotate_axis_xyz(angle, axis_x, axis_y, axis_z) rotates around the axis defined by the origin (0, 0, 0) and the (axis_x, axis_y, axis_z) Vector.

look_at(target) rotates the CoordSyst so as his front (*i.e.* -Z) direction points toward the given target, and tries to maintain the Y direction as the up direction.

look_at_x(target) is similar to look_at, but makes the X direction looking at the target, instead of -Z.

look_at_y(target) is similar to look_at, but makes the Y direction looking at the target, instead of -Z.

scale(x, y, z) scales the CoordSyst by x, y and z.

set_scale_factors(scale_x, scale_y, scale_z) sets the scale_x, scale_y and scale_z attributes in a single call.

get_dimension() -> (float, float, float) returns the width, height and depth dimension of the CoordSyst.

set_dimension(width, height, depth) scales the CoordSyst so as its dimensions are the given width, height and depth.

CoordSyst1 % CoordSyst2 -> Point returns a Point at the same place than CoordSyst1, but in the CoordSyst2 coordinate system (the returned value may be CoordSyst1 itself if it is already in CoordSyst2, or a newly created Point).

2.13.4 Body

Inherits from: **CoordSyst**.

A Body displays a Model at a specific 3D position. Model cannot be displayed without "emBodying" them; it allows to display the same Model at several location (*e.g.* two identical houses in a town), by creating two Bodies with the same Model.

Constructor is:

Body(parent=None, model=None, opt=None) -> Body where parent and model are obvious, and opt is an optional argument passed to the model (for AnimatedModel, it can be a list of the mesh names to attach, see chapter 4).

Noticeable attributes are:

model the Model the Body displays.

deforms the list of Deform applied to the Body (default to an empty list; you should not modify the list directly, but use the add_deform and remove_deform methods; see section 9.7).

The following attribute is only available if the Body's Model is an AnimatedModel (see chapter 4):

attached_meshes the list of the attached meshes names (only with AnimatedModel, see chapter 4).

Noticeable methods are:

add_deform(deform) applies the given Deform to the Body (see section 9.7).

remove_deform(deform) removes the given Deform from the Body (see section 9.7).

The following methods are only available if the Body's Model is an AnimatedModel (see chapter 4):

attach(mesh_name1, mesh_name2,...) attaches the meshes of the given names.

detach(mesh_name1, mesh_name2,...) detaches the meshes of the given names.

is_attached(mesh_name) -> int returns true if the mesh named mesh_name is attached.

animate_blend_cycle(animation_name, weight=1.0, fade_in=0.2) plays the animation of the given name in cycle, with the given weight (usefull is several animations are cycled simultaneously, which is possible), and fade_in is the time (in second) needed to reach the full weight, in order to avoid a brutal transition. The animation will **not** start at its beginning, but at the current global animation time, which is shared by all cycles (use animate_reset if you want to start a cycle at its beginning).

animate_clear_cycle(animation_name, fade_out=0.2) stops cycling the animation of the given name; fade_out is the time (in second) needed to stop the animation.

animate_execute_action(animation_name, fade_in=0.2, fade_out=0.2) plays the animation of the given name once; fade_in and fade_out are the time (in second) needed to reach full weight, and to stop the animation, in order to avoid brutal transitions.

animate_reset() immidiately stops **all** animations, and resets the cycle animation time, *i.e.* future animations played with animate_blend_cycle will restart from their beginning.

set_lod_level() set the current LOD level (only if the Cal3D model file support it).

2.13.5 World

Inherits from: **Body, SavedInAPath**.
Constructor and loading class methods are:

World(parent=None, model=None, opt=None) -> World equivalent to Body's constructor.

World.load(filename) loads a World (see section 3.7.1).

World.get(filename) loads a World, using a cache of already loaded Worlds (see section 3.7.1).

Noticeable attributes are:

children the list of children CoordSyst directly nested in the World. Do not modify the list (use the add and remove methods).

filename the World's filename (relative to the <data>/worlds/ directory; defaults to None).

atmosphere the Atmosphere, defining the World atmospheric properties like fog, sky or background color (defaults to None, see section 9.6).

model_builder the ModelBuider, *i.e.* the object responsible for turning the World into a Model (defaults to None; in this case, a default ModelBuilder is used, see chapter 10). ModelBuider can be used to add shadows, cell-shading,...

The following attributes are only available if the World's Model is an AnimatedModel (see chapter 4):

attached_coordsysts the list of the CoordSysts attached to a bone, containing (CoordSyst, bone_id, option_flags) tuples (only with AnimatedModel, see chapter 4).

Noticeable methods are:

add(CoordSyst) adds the given CoordSyst inside the World.

insert(index, CoordSyst) is similar to add, but insert the CoordSyst at the given index in the children list.

remove(CoordSyst) removes the given CoordSyst from the World.

recursive() -> list of CoordSysts returns the recursive list of children CoordSysts, *i.e.* the children list, plus the children list of the nested Worlds, and so on.

search(predicate) -> CoordSyst searches (recursively) for a CoordSyst that satisfies the given predicate; predicate is callable that take a CoordSyst argument and that return true of false.

search_all(predicate) -> list of CoordSysts is like search, but returns the list of all CoordSysts that satisfy the predicate.

search_name(name) -> CoordSyst searches (recursively) for a CoordSyst whose name attribute is the given name (aliased to World[name]).

subitem(namepath) -> CoordSyst returns the CoordSyst denoted by namepath. namepath is one or more names separated by dots, *e.g.* "character.head.mouth".

to_model() -> Model turns the World into a Model (see chapter 10).

raypick(origin, direction, distance=-1.0, half_line=1, cull_face=1, p=None, v=None) -> (Point, Vector) performs a raypicking, and returns either a (impact_point, normal_at_the_impact) tuple, or None (see section 8.1).

raypick_b(origin, direction, distance=-1.0, half_line=1, cull_face=1) -> int performs a raypicking, and returns 1 if there is a collision, and 0 if there is not (see section 8.1).

RaypickContext(center, radius, RaypickContext=None, items = None) -> RaypickContext creates a RaypickingContext. RaypickingContext are used to perform several raypicking in the same region, faster than by calling raypick or raypick_b (see section 8.1).

The following methods are only available if the World's Model is an AnimatedModel (see chapter 4):

attach_to_bone(CoordSyst, bone_name) attaches the given CoordSyst (which is understood to be a direct child of the World) to the bone of the given name. When the bone is moved by the animation, the CoordSyst moves too.

detach_from_bone(CoordSyst) detaches the given CoordSyst from a bone.

2.13.6 Camera

Camera is the "eye" from which the 3D scene is viewed. It also acts as the "ear", for 3D sound. It also inherits from Widget (see section 11).

Inherits from: **CoordSyst**, **Widget**.

Constructor is:

Camera(CoordSyst) -> Camera

Noticeable attributes are:

front the minimum distance at which 3D objects can be seen (defaults to 0.1; cannot be 0.0).

back the maximum distance at which 3D objects can be seen (defaults to 100.0). If the back / front ratio is too big, you loose precision in the depth buffer.

fov the field of vision (or FOV), in degrees. Default is 60.0.

left, top, width, height the viewport rectangle, in pixel. Use it if you want the Camera to render only on a part of the screen. It defaults to the whole screen.

partial if true, the Camera is considered to use only a part of the screen, and not the whole screen, in particular for clearing purpose. Clearing a partial Camera is slower, but it doesn't clear the whole screen (defaults to false).

ortho if true, the Camera uses orthogonal perspective; if false (default) it uses real perspective.

listen_sound true if the Camera is used as the "sound listener". A single Camera can be used so at the same time (defaults to true, see section 7.4).

to_render the world that is rendered by the Camera. Default is None, which means the root scene (as returned by get_root()).

master the master Widget (see section 11.2).

Noticeable methods are:

set_viewport(left, top, width, height) sets left, top, width and height in a single call.

get_screen_width(), get_screen_height() -> int gets the width and the height of the rendering screen, in pixel.

coord2d_to_3d(x, y, z, reused_Point=None) -> Point converts 2D coordinates X and Y (in pixel, *e.g.* mouse position) into a Point. Z is the Point Z coordinates (in the Camera coordinate system) ; it should be negative and defaults to -1.0. reused_Point is an optionnal Point that is used to store the result, if you want to avoid the creation of a new Point object and prefer reuse an existant one (for speed purpose).

coord3d_to_2d(Point) -> (x, y) converts a Point (or a CoordSyst) into 2D screen coordinates X, Y (in pixel).

render_to_material(Material, what) renders the camera to a Material's texture. 'what' is one of GL_RGBA, GL_LUMINANCE, GL_ALPHA.

is_in_frustum(CoordSyst) -> int returns true if the given CoordSyst is inside the Camera's frustum.

2.13.7 Light

Inherits from: **CoordSyst**.
Constructor is:

Light(CoordSyst) -> Light

Noticeable attributes are:

constant the constant attenuation of the Light (defaults to 1.0). This attenuation factor is not influenced by the distance.

linear the linear attenuation of the Light (defaults to 0.0). This attenuation factor is proportional to the distance.

quadratic the quadratic attenuation of the Light (defaults to 0.0). This attenuation factor is proportional to the square distance.

ambient the ambient color of the Light (defaults to no ambient, *i.e.* black or (0.0, 0.0, 0.0, 1.0)). This part of the Light is not affected by the Light's orientation or attenuation.

diffuse the diffuse color of the Light (defaults to white, *i.e.* (1.0, 1.0, 1.0, 1.0)). This color is the "main color" of the Light.

specular the specular color of the Light (defaults to white, *i.e.* (1.0, 1.0, 1.0, 1.0)). This color is used for the bright part of the object.

directional if true, the Light is directional (*e.g.* a sun). If false (default), The position of a directional Light doesn't matter, and only the constant component of the attenuation is used.

angle if angle is < 180.0, the Light is a spotlight; angle being the angle of the splot (defaults to 180.0).

exponent modifies how a spotlight Light is spread over space.

top_level if true, the Light pass through Portal (see section ; defaults to false).

cast_shadow if true, the Light casts shadows on Model with shadow enabled (default is true).

shadow_color the color of the shadows casted by the Light (default is semi-transparent black, *i.e.* (0.0, 0.0, 0.0, 0.5)).

2.13.8 Point

Constructor is:

Point(CoordSyst, x, y, z) -> Point

Noticeable attributes are:

x, y, z the X, Y and Z coordinates (defaults to 0.0, 0.0, 0.0).

parent the CoordSyst in which the Point is defined.

Noticeable methods are:

get_root() -> World get the root parent of the Point (the scene).

position() -> Point creates a Point in the same parent and at the same place than the Point.

distance_to(position) -> float returns the distance between the Point and position (another CoordSyst or a Point).

vector_to(position) -> Vector creates a Vector that starts at the Point position, and ends at the given position (another CoordSyst or a Point; aliased to the $>>$ operator).

set_xyz(x, y, z) set the x, y and z attributes in a single call.

move(position) moves the Point at the same place than position (another CoordSyst or a Point).

add_vector(vector) translates the Point by the given Vector (aliased to the += operator) .

add_mul_vector(k, vector) translates the Point by k times the given Vector (equivalent to, but faster than, add_vector(k * vector)).

add_xyz(x, y, z) translates the Point by (x, y, z) (expressed in the CoordSyst's parent coordinate system).

copy() -> Point returns a copy of the Point

clone(other) changes in place the Point so as it is a clone of other (a Point or a CoordSyst).

convert_to(CoordSyst) converts in place the Point to the CoordSyst coordinates system. The x, y and z coordinates are modified, and the Point's parent is set to the given CoordSyst.

Point % CoordSyst -> Point returns a Point at the same place than the Point, but in the CoordSyst coordinate system (the returned value may be the Point itself if it is already in the right CoordSyst, or a newly created Point).

Point + Vector, Point - Vector -> Point translates the Point by the Vector.

2.13.9 Vector

Vector inherits from Point for implementation and internal purpose, although it can be seen as a mathematical absurdity. Constructor is:

Vector(CoordSyst, x, y, z) -> Vector

CoordSyst_or_Point.vector_to(CoordSyst_or_Point) -> Vector (aliased to >>).

Noticeable methods are:

length() -> float returns the length of the Vector.

set_length(float) scales the Vector so as its length is the given value.

normalize() scales the Vector so as its length is 1.0.

dot_product(Vector) -> float returns the dot product of two Vectors.

cross_product(Vector, reused_Vector = None) -> Vector returns the cross product of two Vectors; if reused_Vector is given, the result will be written in it instead of creating a new Vector.

angle_to(Vector) -> float returns the angle between the two Vectors (in degrees).

set_start_end(start, end) changes the Vector in place so as it starts and ends at the given start and end (Point or CoordSyst).

Vector + Vector, Vector - Vector -> Vector vectorial addition.

float * Vector scales the Vector.

2.13.10 Interesting methods for overriding

When overriding a Soya method, **don't forget to call the super implementation**!

CoordSyst.begin_round() (see section 2.7).

CoordSyst.advance_time(proportion) (see section 2.7).

CoordSyst.end_round() (see section 2.7).

CoordSyst.added_into(newparent) is called whenever the CoordSyst is added into a new World, or removed from its current World (in this case, newparent is None).

CoordSyst.loaded() is called **after** the object was loaded from a file. Notice that, if you want to perform some hacking on file loading that involves **several** objects, overriding CoordSyst.loaded is safer than CoordSyst.__setstate__, since other objects may not be fully initialized when CoordSyst.__setstate__ is called.

World.add(coordsyst)

Chapter 3

Managing data

3.1 Data path

Soya stores each class of object in a separate subdirectory in <data>, the data path given at the initialization. <data> is expected to contains the following subdirectories:

<data>/images contains image files (PNG or JPEG; for JPEG you should use the .jpeg extension, and not .jpg).

<data>/materials contains Soya Materials.

<data>/models contains Soya Models.

<data>/animated_models contains Soya AnimatedModels (in Cal3D format, using a directory per AnimatedModel, see chapter 4).

<data>/worlds contains Soya Worlds. These Worlds can either be though as 3D scenes.

<data>/blender contains Blender models (see chapter 5).

<data>/sounds contains sound files (WAV or OGG Vorbis, see chapter 7).

<data>/fonts contains Fonts (see section 11.1).

These objects are the objects Soya can load; all of them inherit from SavedInAPath (excepted for Blender model, which are not Soya objects, of course). Other Soya objects can be saved, but not directly. For example, you cannot save just a Body in a file, but you can include a Body inside a World, and then you can save the World.

Known bug: Currently, Camera cannot be saved in files.

3.2 File formats

All Soya-specific objects (Materials, non-animated Models, Worlds and their content) are saved through serialization. Soya currently supports two file formats: Pickle and Cerealizer. Pickle (actually cPickle) is integrated into Python, and can save any object you may create, however **Pickle is not secure** for networking game. Cerealizer (`http://home.gna.org/oomadness/en/cerealizer`) is secure, but it requires you to register manually the class that are safe to read from a file.

The default file formats is to save files with Pickle, and to load either Pickle or Cerealizer files (Soya can determine automatically the format of a file; if Cerealizer is not installed, loading Cerealizer file is of course disabled). However, you are encouraged to use Cerealizer for security purpose.

To set the file formats, use the following function:

set_file_format(saving_format, loading_formats = None) where saving_format is the format for saving files (either a function with a signature like pickle.dumps, or a module with a dumps function), and loading_formats is the format for loading files (either a function with a signature like pickle.loads, or a module with a loads function), or a list of formats. If loading_formats is None, the loading formats are left unmodified.

The actual default (which may change) is equivalent to:

```
import cPickle, Cerealizer
set_file_format(cPickle, [cerealizer, cPickle]) # if Cerealizer is available
set_file_format(cPickle, cPickle)               # if Cerealizer is not available
```

To use only Pickle (for compatibility with older apps):

```
set_file_format(cPickle, cPickle)
```

To use Cerealizer while still being able to read cPickle files:

```
set_file_format(cerealizer, [cerealizer, cPickle])
```

To use only Cerealizer – **this is the only configuration safe for networking**:

```
set_file_format(cerealizer, cerealizer)
```

If you use Cerealizer you have to declare which class is safe for saving / loading. Soya automatically register Soya's classes, but you have still to register your derived classes. This can be done as following:

```
class YourClass(soya.CoordSyst):
    ...
cerealizer.register(YourClass)
```

If your class inherits from SavedInAPath, usually World, and you want YourWorld.get to work properly, you should do:

```
class YourWorld(soya.World):
    ...
cerealizer.register(YourWorld, SavedInAPathHandler(YourWorld))
```

And, if you want your objects to be saved in the directory <data>/your_worlds/ instead of <data>/worlds/, do:

```
class YourWorld(soya.World):
    DIRNAME = "your_worlds"
    _alls = weakref.WeakValueDictionary()
    ...
```

3.3 Saving objects

In Soya, saving an object is done in two steps (see tutorial basic-savingfile-pickle-1 and basic-savingfile-cerealizer-1):

```
obj.filename = "your_object"
obj.save()
```

The object is saved in the corresonding subdirectory in the first data path (*i.e.* soya.path[0]), and the file is named <filename>.data (*e.g.* <data>/worlds/your_object.data). To save the object again, just call obj.save. Images, Sounds, Fonts and AnimatedModels cannot be saved; as Soya is not able to modify them, it would be a non-sense.

When saving a reference to a SavedInAPath object (*i.e.* an Image, a Material, a Model, an AnimatedModel, a World, a Sound or a Font), if the object has a filename, only the filename will be saved. If it has not, the object will be saved normally. In the following example, the data of the sword Model are not saved in the scene's file:

```
scene = soya.World()
body = soya.Body(scene, soya.Model.get("sword"))
scene.filename = "sword_scene"
scene.save()
```

When the scene will be loaded, the sword Model will be loaded by calling soya.Model.get("sword"). As a consequence, you'll have to distribute the sword Model along with the scene.

3.4 Loading objects

Objects can be loaded with one of these class methods:

SavedInAPath.load(filename) -> SavedInAPath

SavedInAPath.get(filename) -> SavedInAPath

The difference between load and get is that load always return a new object, whereas get return the same object when it is called several time with the same filename. Images, Models, Sounds and Fonts are immutable in Soya, and thus get is the preferred method for loading them. For example, to load the sword_scene saved above:

```
scene = soya.World.load("sword_scene")
```

See also tutorial basic-loadingfile-1.

3.4.1 Extended filenames

The @ character is used to indicate some optional parameters in a filename:

For Fonts, it can be used to indicate the horizontal and vertical Font size (the horizontal size follows the @, then an x, then the vertical size). It can be used to create several Font objects of various sizes from a single Font file. *E.g.* :

```
font = soya.Font.get("indigo.ttf@20x30")
```

For Models and Worlds, it can be used to generate several Worlds and Models from a single Blender file (see section 5.5).

3.5 Auto-exporters and automatic conversions

When loading data, Soya can automatically perform the following conversions:

- Image -> Material
- Blender model -> World
- World-> Model
- Blender model -> AnimatedModel

Concretely, it means that, if you load (with either load or get) a Material that doesn't exist, and if there is an Image with the same filename, Soya will create a new Material, using this Image as texture, and saved (for future use). If the Material already exists, but there is a more recently modified Image with the same filename, Soya will load the Material, and automatically update the texture.

Similarly, when loading a World, Soya searches for a Blender model with the same filename, and, if needed, will export it to Soya (Notice that this require Blender to be installed). When loading a Model, Soya searches for a World (and thus for a Blender model), and, if needed, will load the World and re-turn it to a Model (Models are generated from Worlds, see chapter 10). This feature is known as "Soya's auto-exporters".

Auto-exporters are really nice for development, however they can be annoying in the final version of an application, for example one doesn't expect a game to start blender for re-exporting the model (when installing the game, the timestamp of the various files may be changed, messing up the whole auto-exporters system). To prevent that, you can disable auto-exporters as following:

```
soya.AUTO_EXPORTERS_ENABLED = 0
```

A common trick consists in enabling auto-exportes only for Subversion / CVS sources:

```
APPDIR = os.path.dirname(sys.argv[0]) # os.path.dirname(__file__) for a Python module
soya.AUTO_EXPORTERS_ENABLED = os.path.exists(os.path.join(APPDIR, ".svn"))
```

3.6 Where can i obtain Models?

First, you can design model yourself, using a 3D modeler like Blender (see chapter 5), or within Python scripts (*e.g.* for geometrical model; see chapter 10). Free models are also available:

- Nekeme Prod., an association for Free Game (as in Free Speech) Jiba is a member of, maintains the Free Data Repository. The FDR is a database of free resources for games, including 3D models, images, musics, sounds,... It can be browsed at `http://fdr.nekeme.net/`.
- You can also re-use the model of an existing free game, such as Balazar Brother (`http://home.gna.org/oomadness/en/balazar_brother/index.html`).

3.7 Object reference

3.7.1 SavedInAPath

SavedInAPath is an abstract mix-in class, used by Image, Material, Model, AnimatedModel, World, Sound and Font.

Noticeable class attributes are:

DIRNAME the data subdirectory used for saving the instance of this class, *e.g.* "models" for Model.

Noticeable attributes are:

filename the object's filename (relative to the <data>/<class>/ directory); if the extension is .data, it is not present in the filename. If the object has a filename, when saving other Soya objects that refer to it, only the filename will be saved (this allow to share *e.g.* a Model between two World scenes). If the object has no filename, the object will be dupplicated in any other file that refer to it.

Noticeable class methods are:

load(filename) -> SavedInAPath loads the object saved in the <data>/<class>/<filename> file. Depending of the object, it may perform automatic conversion.

get(filename) -> SavedInAPath is similar to load, but, if called several times with the same with the same filename, it returns the same (cached) object instead of loading it twice.

availables() -> list of strings returns the list of the filename of all the objects available in the <data>/<class>/ directory.

Noticeable methods are:

save(absolute_filename=None) saves the object in <data>/<class>/<filename>, or in absolute_filename if given. Some objects (namely, Images, Sounds, Fonts and AnimatedModel) cannot be saved by Soya, and can only be loaded.

loaded() called when the object is loaded from a file; you may override it.

Chapter 4

Animated models

4.1 Loading animated model

AnimatedModels are loaded as usual, excepted that the model is in the Cal3D format, and not the Soya one. Auto-exporter works as usual with Blender, and generates Cal3D model as required. For example to load the Cal3D model located in <data>/animated_models/balazar/ (corresponding to <data>/blender/balazar.blend, if it exists[1]):

```
sorcerer_model = soya.AnimatedModel.get("balazar")
```

4.2 Displaying the model

AnimatedModel can be attributed to Body or World as any other Models:

```
sorcerer = soya.Body(scene)
sorcerer.shape = sorcerer_shape
```

4.3 Attaching meshes

A Cal3D model can be composed of several meshes, that can be attached (i.e. displayed) or not. Non-attached meshes are not visible. The AnimatedModel.meshes attribute is a dict mapping mesh names to their numerical ID. By default, all meshes are attached, but you can detach or attach some of them, as following:

```
print "Available mesh names:", sorcerer_shape.meshes.keys()
sorcerer.detach("helmet")
sorcerer.attach("armor")
```

Notice that displaying AnimatedModels is much slower than non-animated Model, due to the animation-related computation, even if you actually don't play animation.

4.4 Playing animations

An AnimatedModel can have several animations ; the AnimatedModel.animations attribute is a dict mapping animation names to their numerical ID:

```
print "Available animation names:", sorcerer_shape.animations.keys()
```

To start playing an animation in loop, use the animate_blend_cycle(animation_name, weight = 1.0, fade_in = 0.2) method (see tutorial character-animation-1.py).

```
sorcerer.animate_blend_cycle("walk")
```

[1] "Balazar" is just the name of my sorcerer!

Notice that several animation can be played simultaneously, and blended together. The weight argument indicate the weight of the animation ; it defaults to 1.0 but may be changed if there are several simultaneous animations. The fade_in argument is the time (in second) to reach the full weight (in order to avoid a brutal transition, which corresponds to fade_in = 0.0).

Then, the animation can be stopped by the animate_clear_cycle(animation_name, fade_out = 0.2) method. The fade_out argument is the time (in second) required to fully stop the animation.

```
sorcerer.animate_clear_cycle("walk")
```

Hint: When you start playing an animation with animate_blend_cycle, the animation may not start at the beginning, but at the current Cal3D internal counter value. The animate_reset() method can be used to reset this internal counter, so as cycled animations starts at their beginning.

```
sorcerer.animate_reset() # Ensure we start the walking animation at its beginning
sorcerer.animate_blend_cycle("walk")
```

Finally, a last method exists for animation: animate_execute_action(animation_name, fade_in = 0.2, fade_out = 0.2). It plays an animation once, without the possibility to blend several animation at the same time. However, I use it rarely, since it is not possible to interrupt the animation before its end.

4.5 Attaching objects to bones

Soya allows you to attach Soya CoordSyst to the animated bones. This is extremely usefull if you want to add items to an animated character, like a sword. It can also be used for manual collision detection, since it allows to know where is located a part of the AnimatedModel. We are going to add a sword to the sorcerer. For that, we first need the sorcerer to be a World (instead of just a Body), and then we create a World corresponding to the right hand of the sorcerer. The right hand is added inside the sorcerer ; this is **mandatory** for attaching objects to bone !

```
sorcerer = soya.World(scene, sorcerer_shape)
right_hand = soya.World(sorcerer)
```

Now, using the World.attach_to_bone(CoordSyst) method, we attach the right hand to the sorcerer's bone named "right_hand" (this is the name given to the bone in Blender). This causes the righ hand World's position and orientation to be automatically updated according to the bone's position and orientation.

```
sorcerer.attach_to_bone(right_hand, "right_hand")
```

And finally, we create a sword Body in the right hand World :

```
sword = soya.Body(right_hand, soya.Model.get("sword"))
sword.rotate_z(180.0) # rotate and position the sword as needed
sword.set_xyz(0.05, 0.1, 0.0)
```

The resulting scene tree is the following (see tutorial character-animation-2.py):

```
World scene
  |
  +-- World sorcerer, displaying the Balazar AnimatedModel
  |     |
  |     +-- World right_hand, attached to the bone named "right_hand"
  |           |
  |           +-- Body sword, displaying the sword Model
  |
  +-- Light light
  |
  +-- Camera camera
```

You can detach a CoordSyst from a bone using the World.detach_from_bone(CoordSyst) method.

Hint: objects that are attached to bone must always be parented to the World that have the AnimatedModel, even if the bones are inserted one inside the other in Blender. For instance, you may have the following scene tree:

```
World scene
  |
  +-- World sorcerer, displaying the Balazar AnimatedModel
        |
        +-- World right_arm, attached to the bone named "right_arm"
        |
        +-- World right_hand, attached to the bone named "right_hand"
```

4.6 Object reference

4.6.1 AnimatedModel

Most of the attribute can be set in Python (*e.g.* animated_model.sphere = (0.0, 0.0, 0.0, 2.0)), in the .cfg Cal3D file (by adding *e.g.* sphere=0.0 0.0 0.0 2.0) or in the Blender file (by adding in a text buffer called 'soya_params' *e.g.* sphere=0.0 0.0 0.0 2.0, see section 5.4).

Noticeable attributes are:

animations a dictionary mapping animation names to their numerical ID.

meshes a dictionary mapping mesh names to their numerical ID.

materials a list of the Materials used by the AnimatedModel.

sphere a (x, y, z, radius) tuple defining a culling sphere for the animated model (a radius of -1.0 disable sphere culling, which is the default). You can set the sphere property for increasing performance.

double_sided if true, draw both sides of each face of the model. Default is true. It can be disabled for increasing performance.

shadow if true, the model casts shadows. Default is false.

cellshading if true, cell-shading is enabled. To enable cell-shading, use set_cellshading() or modify the Cal3D .cfg file or the Blender file.

Noticeable methods are:

set_cellshading(shader = DEFAULT_SHADER, outline_color = BLACK, outline_width = 4.0, outline_attenuation
enable cellshading, with the given shader and outline properties. Set outline_width to 0.0 to disable outline.

Chapter 5

Blender for Soya

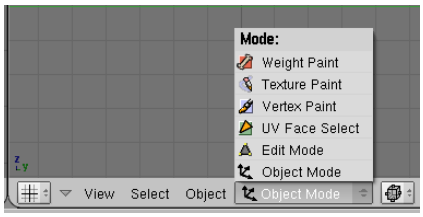
5.1 Modeling in Blender

The following Blender tutorial is very short and Soya-oriented. XXX add here some links to other Blender tutorials.

5.1.1 Drawing the mesh structure

The first step is to create the mesh structure in Blender. I usually start by adding a cube (Menu add->mesh->cube), and then I deform it, *e.g.* by selecting vertices (select them with the mouse right button) and then moving (press the "g" key), rotating (press "r") or scaling (press "s") them. You can also extrude (press "e", or press space and then click the edit->extrude menu) some vertices or faces (to select a face, select all of its vertices).

Blender has several modes; the first one is the object mode, and allows to move the different mesh objects you have (*e.g.* the cube). If you want to modify a mesh, select it (by clicking on it with the right mouse button) and change to the edit mode. A third mode, the face mode, will be used later for applying texture.

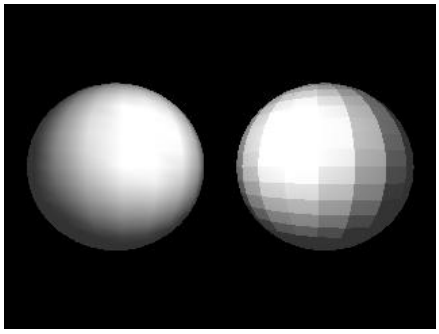


When drawing the mesh, you should use the Blender axis convention (X is right, Y is front and Z is up); the exporter automatically rotates the Model so as it uses Soya's convention (X is right, Y is up and Z is back).

Warning: Only triangles or quads are supported; lines, points or more complex faces are not.

5.1.2 Smooth or solid lighting

Then you have to choose between smooth or solid lighting. Smooth ligthing should be used for objects that are smooth by nature (although they are, as any 3D model, made of face). The following picture shows two spheres; the left one uses smooth lighting, and the right one uses solid lighting.



Notice that the smooth or solid lighting can be set on a per-face basis. In addition, Soya automatically remove the smooth lighting between two faces that make an angle higher than 80 degrees (this value can be changed by setting the max_face_angle parameter). This effect is not visible in Blender, but usually corresponds to what you expect.

5.1.3 Designing textures

Textures can be done in any 2D bitmap image editor, such as The Gimp. Textures should be PNG or JPEG images (with a .jpeg extension), and they should be saved in the <data>/images/ directory. Soya support RGB and RGBA images, as well as indexed colors. The **dimensions of the image must be powers of two** (*e.g.* 8, 16, 32, 64, 128, 256, 512,... pixels), but the image doesn't need to be a square.

Hint: Soya automatically check if the texture image has an alpha channel or not; however The Gimp (as well as other painting programs) sometimes automatically add an undesired alpha channel. Since alpha texture are slower that non-alpha one, and possibly buggy when two of them overlap (see "known bug" below), you should ensure it is not the case. If needed, remove the alpha channel ("flatten image " in The Gimp).

Hint: In Blender, the length of the name given to a texture is limited to about 19 characters. As the exporter assume that this name is the name of the corresponding image file (which is the default value in Blender), you should avoid long filenames for textures.

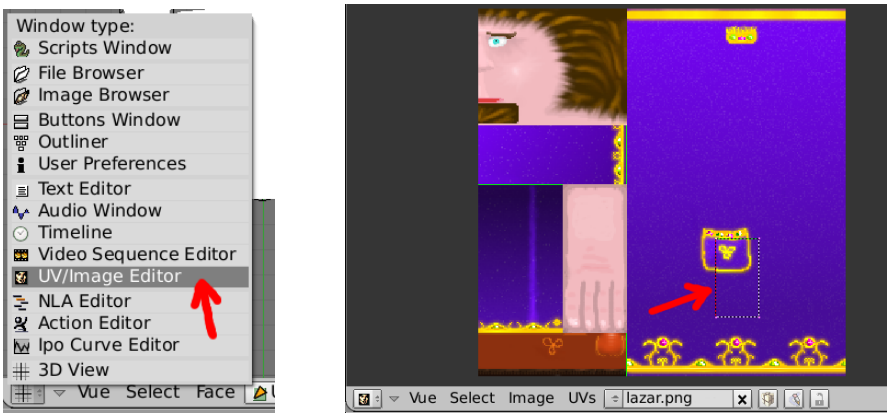
Known bug: When two semi-transparent objects overlap, because one of them is in front of the other, Soya may not render them well, i.e. the farther object in front of the other one. There are two possible workaround for this bug:

- For objects like a grid, an herbage or grass, use a texture with a "mask", instead of a semi-transparent texture. A texture with a "mask" can have transparency, but not semi-transparency. Soya automatically uses a mask for texture with an alpha channel, with all alpha values being either 0.0 or 1.0 (255 in the Gimp integer notation).
- For special effects like spells or explosion, use additive blending (see section 10.1). In facts, which object is rendered first as no impact with additive blending, as a consequence the bug is now harmless. Moreover, additive blending often makes explosions or spells more impressive.

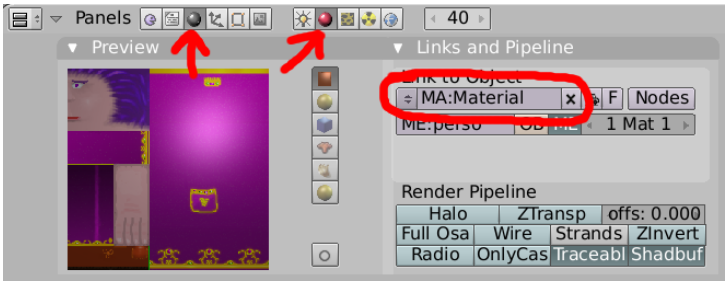
When an UV image is mapped to a face in Blender, Soya automatically exports it using the Material of the same name than the image filename, and creates this Material from the image if it doesn't exist yet.

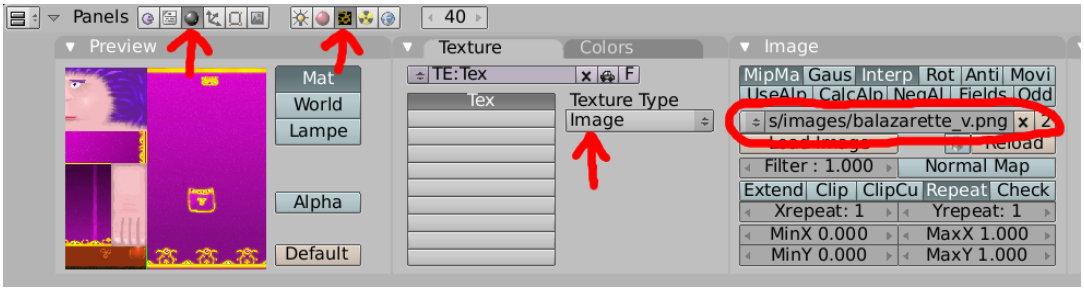
5.1.4 Applying the texture to the model

In Blender, enter in face mode, and select the faces you want to apply the texture to (press "a" to select all faces). Then go the the UV/image editor window, and use the Image->Open menu to open your texture. Finally, position the UV coordinates by moving the vertex of the triangle or quad over the texture.



When exporting to AnimatedModel (i.e. to Cal3D format, using BlenderCal), each material need to be associated to the corresponding texture. This can be done by selecting the right material in the material panel (see the first screenshot below), and then by selecting in the texture panel, the "Image" texture type and the right image (see the second screenshot below).

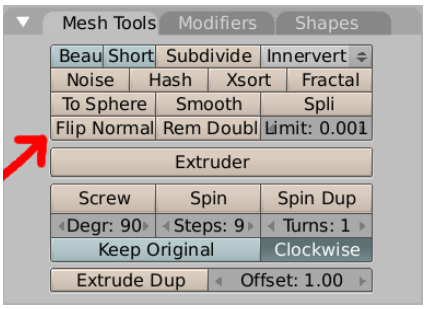


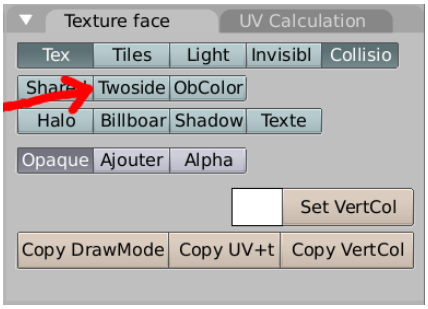


5.1.5 Face's sides

By default, Soya and Blender shows only one side of each face (Blender shows both side in some draw type modes, but not in the final rendering). Which side is visible depends on the normal of the face. If a face shows the wrong side, select it (by selecting all of its vertices in edition mode, or by selecting it in face mode, and then go to edition mode), and then click the "flip normal" button.

If you want to show both sides, select the face in face mode, and then click the "twoside" button.



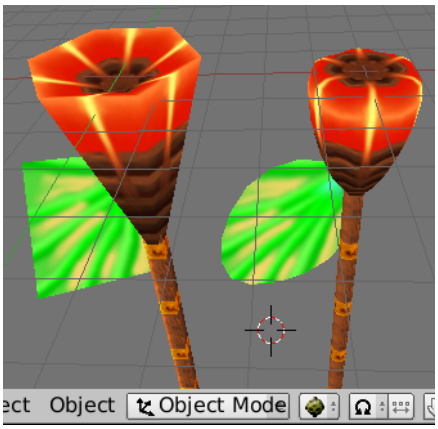


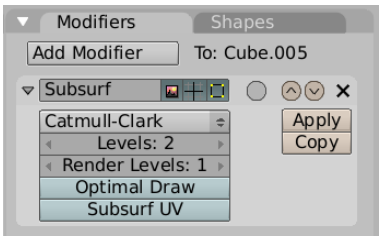
5.1.6 Adding face colors

You can also add per-face colors, although this feature is not commonly used. Soya exports them as per-vertex colors (since Soya doesn't support per-face colors).

5.1.7 SubSurf

In Blender, SubSurf can be used to automatically increase the details of a Model. The following picture show the same model without and with SubSurf:





When exporting non-animated Model, Soya automatically take SubSurf into account. However, this is not the case for AnimatedModel. A common trick is to apply the SubSurf on the Model (by clicking the "Apply" button); notice that applying SubSurf destroys any vertex group that you may have created, as a consequence it should be done **before** defining vertex groups.

5.1.8 Adding an armature

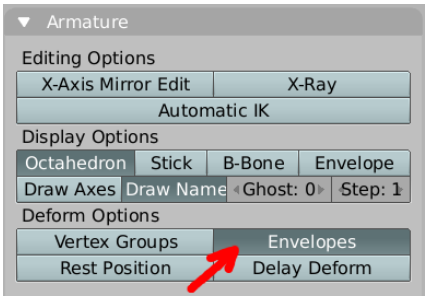
Blender's armatures are the skeletons used for animation. An armature is thus required only for AnimatedModel (although it may be used on non-animated Model, for generating several Soya Models being the same Blender model at different animation frame).

To add an armature, choose the Add->Armature menu, and then draw the armature's bones.

5.1.9 Linking bones to vertices

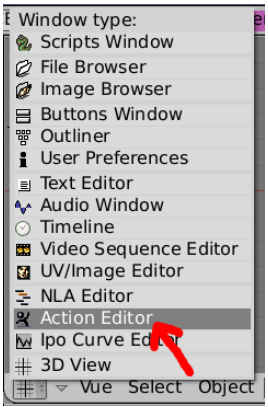
The second step for AnimatedModels is to link the armature's bones to the vertices. This can be done:

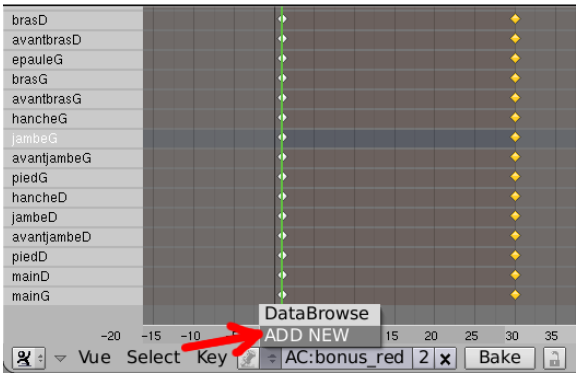
- Using vertex groups: select both the model and the armature (in that order), and click the Object->Parent->Make parent->Armature->Create from closest bones. Vertex groups corresponding to the various bones will be automatically created, and can then be refined manually. However, it seems that the recent version of Blender (2.41) are not as efficient for this than older version.
- Using envelopes: process as above, but choose "Don't create group" instead of "Create from closest bones", and then check the "envelopes" button in the armature's properties.



5.1.10 Adding animations

In Blender, animations are called "actions". Go to the "Action editor" window, add a new action, and give it a name. In the 3D window, select the armature and enter in the pose mode. Then move and rotate the bones, and add a keyframe with the Pose->Insert keyframe menu.





5.2 Auto-exporter

The easiest way to export Blender model is to use the auto-exporter (see section 3.5).

Simply save your Blender model in <data>/blender/ (*e.g.* <data>/blender/your_model.blend). Then in Soya, load the World or the Model of the same name (without extension), *e.g.*:

```
soya.Model.get("your_model")
soya.AnimatedModel.get("your_model")
```

Soya will automatically launch Blender, export the model, cache it in the <data>/worlds/, <data>/models/ and <data>/animated_ directories, and quit Blender in a fraction of second.

If the Blender file is updated after that, Soya will automatically re-export it.

5.3 Blender features exported to Soya

Here is a summary of the following Blender features that are correctly exported to Soya:

Mesh structure

Face UV image (mapped to Face.material, see section 10).

Vertex UV texture coordinates (mapped to Vertex.tex_x and Vertex.tex_y, see section 10).

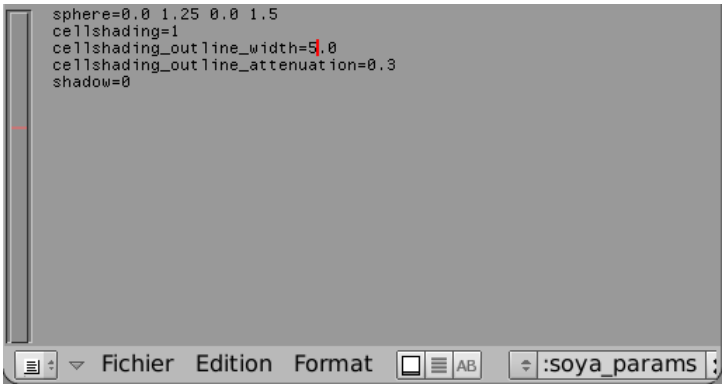


Figure 5.1: Using "parameter text buffer" in Blender

Face_twoside (mapped to Face.double_sided, see section 10).

Smooth or solid lighting (mapped to Face.smooth_lit, see section 10).

Face colors (mapped to Vertex.color, see section 10).

For non-animated Model only:

SubSurf

For AnimatedModel only:

Armature (also called "skeleton").

Animation (called "action" in Blender).

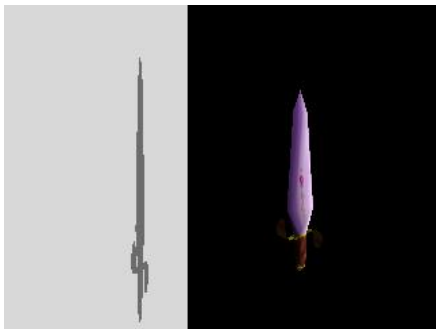
AnimatedModel are exported to the Cal3D file format using BlenderCal (included in Soya source ; it requires Blender >= 2.42a).

5.4 Adding Soya-specific attributes in Blender

Soya also provides some features that are not supported by Blender. These features can be defined using a "parameter text buffer" (see figure 5.1). First, create a text buffer in Blender, and name it "soya_params". This text buffer contains Soya-specific informations, given as "key=value" pairs. The following pairs are supported:

scale=2.0 scales the Model (in the three directions).

shadow=1 activate shadows on the Soya Model (use 0 for disabling shadow, which is the default; see picture below).



cellshading=1 activate the cellshading on the Soya Model (use 0 for disabling cellshading, which is the default). A cellshaded Model use a different, more cartoon-like, lighting algorithm, and can have an outline (see the picture below; the sword on the left has not cellshading enabled, the sword on the right has it).

cellshading_shader="filename" the name of the Material used as the "shader" (defaults to soya.SHADER_DEFAULT_MATERIA Only the texture of the Material is used; it should be a 1-pixel-wide alpha texture. This texture is then added over the normal Model texture, with the top-most pixels being added over the darker parts of the Model and the bottom-most pixels over the bright parts.

cellshading_outline_width=1.0 the width of the cellshading outline (default to 0.0, this parameter is used only if cellshading is activated). If the width is 0.0, no outline is added to the Model.

cellshading_outline_color=red,green,blue,alpha the color of the cellshading outline (default to black, this parameter is used only if cellshading is activated).

cellshading_outline_attenuation=0.3 the attenuation of the cellshading outline, with regard to the distance (defaults to 0.3).

animation=blender_action_name the name of a Blender action; the corresponding action will be set current before exporting (defaults to None).

animation_time=3.0 the frame number sets before exporting (this parameter is expected to be used with the animation one).

max_face_angle=80.0 the maximum angle between two smooth-lit Faces. If the angle between two Faces is higher than the given value, the two Faces won't be considered as smooth (with regard to each other), even if they are marked as smooth in Blender. In other 3D engine (including Blender), you need to dupplicate vertices for disabling smoothing; however Soya takes into account the angle between each Face, and does that automatically for you. Default is 80.0; you can disable this feature by setting it to 360.0.

keep_points_and_lines=1 if true, points and lines are kept in the Model (by default, Soya drops them, and keep only triangles and quads).

material_oldname=newname replaces the Material named "oldname" by the Material named "newname".

config_text=blender_text_buffer_name also read the Blender text buffer of the given name.

config_file=file also read the given file (as if it was a parameter buffer).

5.5 Generating several Soya models from a single Blender file

The parameter text buffers named "soya_params" is always parsed. It is possible to export several Soya Model from a single Blender file, using additional parameter text buffers. In this case, the Model filename in Soya is "blender_filename@additional_parame
For example, if you have a <data>/blender/sword.blend Blender model, with the following "soya_params" text buffer:

```
cellshading=1
cellshading_outline_width=1.0
```

the following text buffer called "big":

```
scale=2.0
```

and the following text buffer called "blue":

```
material_sword=sword_blue
```

Then,

- soya.Model.get("sword") loads the Model and parses only the "soya_params" buffer.
- soya.Model.get("sword@big") loads the Model and parses the "soya_params" and the "big" buffers, and thus scales the Model by 2.
- soya.Model.get("sword@blue") loads the Model and parses the "soya_params" and the "blue" buffers, and thus replaces the "sword" Material by the "sword_blue" Material.

The animation and animation_time can be used to generate several non-animated Model corresponding to the various frames of one or more Blender actions. For example, to generate a non-animated Model that is a statue of Balazar running, at the frame 2:

```
animation=run
animation_time=2.0
```

5.6 Exporting Soya model to Blender

The script soya/soya2blender.py (in the Soya sources) can import a Soya Model in Blender. You have to modify the end of the script to choose the Model to import, and then run the script manually in Blender.

Known bug: It seems that soya2blender.py doesn't export texture well, so you'll have to re-set the texture.

5.7 What about other 3D modelers ?

In addition to Blender, the following modelers are supported by Soya:

- 3DSMax (see soya/_3DS2soya.py).
- MilkShake 3D (see ms3D2soya.py).
- OBJ/MTL (see objmtl2soya.py).

Chapter 6

Event handling

6.1 Getting events

XXX Soya's event system is still quite primitive and need a rewrite.

soya.process_event() computes and returns all events that have occured since the last call to process_event.

An event is a tuple ; the first value of an event tuple is a constant from soya.sdlconst, and the following values depend of the event type. Event types are:

```
(sdlconst.KEYDOWN, key, mods[, unicode_key])
(sdlconst.KEYUP, key, mods)
(sdlconst.MOUSEMOTION, x, y, x_relative, y_relative, state)
(sdlconst.MOUSEBUTTONDOWN, button, x, y)
(sdlconst.MOUSEBUTTONUP, button, x, y)
(sdlconst.JOYAXISMOTION, axis, value)
(sdlconst.JOYBUTTONDOWN, button)
(sdlconst.JOYBUTTONUP, button)
(sdlconst.VIDEORESIZE, width, height)
(sdlconst.VIDEOEXPOSE)
(sdlconst.QUIT)
```

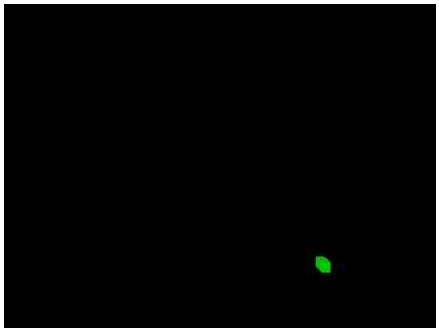
unicode_key is present only if soya.set_use_unicode(1) has been called.

6.2 Converting mouse 2D coordinates to 3D coordinates

Mouse coordinates are returned as pixel values by process_event(). These 2D pixel values can be converted into 3D coordinates using Camera.coord2d_to_3d(x, y, z = -1.0) -> Point. As mouse coordinates are2D, the Z value cannot be guessed; if not given, it default to -1.0. Remind that, if you want the mouse to be in front of the Camera, you need $z < 0.0$.

For example, you can use the following example to display a 3D cursor (see the mouse-1 tuto, and the raypicking-2 tuto for an example of drag-drop):

```
class Cursor(soya.Body):
  def __init__(self, parent, model = None):
    soya.Body.__init__(self, parent, model)

  def begin_round(self):
    soya.Body.begin_round(self)
    for event in soya.process_event():
      if event[0] == soya.sdlconst.MOUSEMOTION:
        self.mouse_pos = camera.coord2d_to_3d(event[1], event[2], -15.0)
        self.move(self.mouse_pos)
```



6.3 Converting 3D coordinates to 2D coordinates

3D coordinates can be converted to 2D pixel values using Camera.coord3d_to_2d(position) -> (x, y), where position is either a Point or a CoordSyst.

Chapter 7

Sounds

Soya's sound API is very similar to the API for 3D objects.

7.1 Loading sounds

Soya support currently the following sound file formats:

- WAV (through the Python wave module)
- OGG Vorbis (requires the PyOgg and PyVorbis Python module)

Sound files should be placed in your `<data>/sounds` directory (see chapter 3). You can load a sound by doing:

```
sound = soya.Sound.get("my_sound.wav")
```

Soya uses a kind of streaming to not have to read the whole sound file before starting playing. The Sound object contains only the sound raw data.

7.2 Playing sounds: SoundPlayer

To play the Sound, you need to put it into a SoundPlayer, a subclass of CoordSyst that plays a Sound in a 3D environment (SoundPlayer is to Sound what Body is to Model):

```
sound_player = soya.SoundPlayer(parent, sound)
```

That's all! The Sound will be played at the position of the SoundPlayer, and Soya will automatically take care of the Doppler effect. Moving the SoundPlayer (or his parent, of course) will move the source of the Sound; see the sound-1.py tutorial for an example. When the Sound is over, by default Soya automatically removes the SoundPlayer from its parent. To stop playing the Sound before the end, just remove manually the SoundPlayer from its parent.

To play a background music in loop, at no particular 3D position:

```
sound_player = soya.SoundPlayer(parent, sound, loop = 1, play_in_3D = 0)
```

Hint: the sound and play_in_3D attributes are currently read-only (this may change in the future), and thus you'll have to set them when calling the constructor.

Hint: when a World containing a SoundPlayer is saved, Soya saves the current playing position. When the World will be loaded, the sound will restart **at (about) the same position**, and not at the beginning.

Hint: as WAV files are not seekable though the wave Python module, loading SoundPlayer that are playing WAV can be slow, particularly for big file. As a consequence, you should prefer the OGG Vorbis format for big files like music.

7.3 Sound initialization

soya.init accept the following optional sound-related parameters:

sound is true to initialize 3D sound support (default to false for backward compatibility).

sound_device is the OpenAL device names, the default value should be nice (default tries native, esd, sdl, alsa, arts, and null devices, in order).

soud_frequency is the sound frequency, in Hz (defaults to 44100).

sound_reference_distance is the reference distance for sound attenuation (defaults to 1.0). Increase this value if you find that sounds far from the cameraare too much attenuated.

sound_doppler_factor can be used to increase or decrease the Doppler effect (defaults to 0.01, which sounds a nice value).

Additionally, the set_sound_volume function can be used to control the global sound volume, ranging from 0.0 (no sound) to 1.0 (default and maximum value):

```
soya.set_sound_volume(0.5)
```

Use soya.get_sound_volume() to get the current sound volume.

7.4 Sound and multiple Cameras

Soya uses the Camera as the "ear" from which sounds are listened. However, you can have only a single "ear" at the same time. If you have several Cameras, you have to choose the one that will act as the "ear". This can be done through the listen_sound attribute of the Camera: if this attribute is false, the Camera doesn't act as a "ear" (the default value is true):

```
camera.listen_sound = 0
```

7.5 Object reference

7.5.1 Sound

Inherits from: **SavedInAPath**.
Loading Sounds:

Sound.get(filename) -> Sound loads a Sound from the <data>/sounds/ directory; filename should include the sound extension (*e.g.* .wav or .ogg).

Noticeable attributes are:

filename the sound's filename (relative to the <data>/sounds/ directory).

stereo is true if the sound is stereo, and false if the sound is mono.

7.5.2 SoundPlayer

Inherits from: **CoordSyst**.
Constructor is:

SoundPlayer(parent, sound, loop, play_in_3D, gain, auto_remove) -> SoundPlayer the parameters directly match the attributes.

Noticeable attributes are:

sound is the Sound to play (read-only).

loop if true, the sound restarts from the beginning when it ends (defaults to false).

play_in_3D if true, the sound is played as a 3D sound; if false, as a 2D sound (read-only; defaults to true). Notice that OpenAL cannot play stereo sound in 3D, and you'll get an error if you try that.

gain is the volume of the Sound, ranging from 0.0 to 1.0 (default 1.0).

auto_remove if true, the SoundPlayer is automatically removed when the sound ends (excepted in cases of looping!, defaults to true)

Noticeable methods are:

ended() this method is called when the sound is over. You may override it; the default implementation removes the SoundPlayer from its parent if its auto_remove attribute is true.

Chapter 8

Collision detection and physics

8.1 Raypicking

8.2 Collision (ODE support)

8.3 Physic engine

8.4 Object reference

Chapter 9

Advanced Soya objects

Chapter 10

Modeling

The Soya modelling system allows to create Soya model from Python code, without using Blender or any other 3D modeller. It is also used for writing exporters for 3D modellers.

Soya Model are created by putting several Faces (triangles or quads) in a World, and then converting the World into a Model

10.1 Materials

A material

10.2 Basic Models: cube and sphere

The soya.cube and soya.sphere module provide functions for creating cube and sphere. These functions simply call lower lower functions for creating cubic or pherical models.

10.3 Faces and vertices

In Soya, each model is made of Faces.

10.4 Modelifiers

Warning: Only triangles or quads are supported; lines, points or more complex faces are not.

10.5 Static lighting

10.6 Object reference

10.6.1 Image

10.6.2 Material

10.6.3 Vertex

10.6.4 Face

10.6.5 ModelBuilder

Chapter 11

Font, text, and widget systems

11.1 Fonts and text drawing

11.2 Widgets

11.3 Pudding

11.4 Object reference

11.4.1 Font

11.4.2 Label3D

Chapter 12

Tofu network and game engine

Tofu is a client-server network and game engine for Soya. Main features are:

- single and multi-player mode,
- interpolation of character animation and position,
- persistent world support,
- a player can control one or several characters,
- one or several players can play with the same client (see figure 12.1).

12.1 Principles

12.1.1 Players, PlayerID, Mobiles, Levels

Players are the human players. The Player class represent the human player, and thus it is used **never** used client-side, since the client doesn't have the player data.

PlayerIDs are used to identify Players. Contrary to Players, PlayerIDs are available both at client and server-side. The default PlayerID class provides only two attributes: filename (which is the name of the Player) and password. You may extend the class with additional attributes, such as the character the Player has chosen (*e.g.* Tux, Gnu, and so on). A new Player is automatically created when a new filename is given.

Mobiles are every objects that can move or that may be changed during the game. In particular, the chararacters the Player plays are Mobiles. Bots, *i.e.* characters played by the computer, are also Mobiles. Mobile inherits from soya.World.

Levels are a part of the game's universe. A Mobile is located in a single level at a given time. Levels inherits from soya.World.

Uniques are objects that have a unique identifier (UID), which can be used to identify the object on the server or any client. The UID can be acessed by the uid attribute. Mobiles and Levels are Uniques.

12.1.2 Actions, messages and states

In client-server mode, Tofu sends various information over the network in order to maintain the state of the Mobiles identical on the server and on each client. Tofu distinguishes three types of information (see figure 12.3):

Actions are the action the human player or a bot decide to perform. Human player actions are generated by the client (by reading *e.g.* the keyboard events) and sent to the server. Examples are: start jumping, start walking, stop walking,... Many actions are given as "start" or "stop something", in order to reduce the amount of actions sent.

Messages are information sent by the server to the clients. Messages are only rarely used, in particular they should not be used for sending positionning information, or any other information that evolves often. Examples are information about life lost, or a bonus taken.

States are also information sent by the server to the clients. Contrary to messages, states are for information that evolves so quickly that it is not possible to send all the information. Usually, states are used only for position and orientation of the characters. As a consequence, states are interpolated when needed.

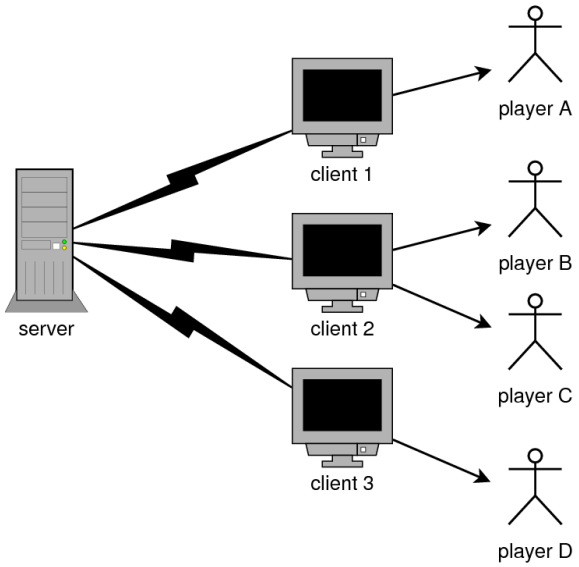


Figure 12.1: Tofu client-server model

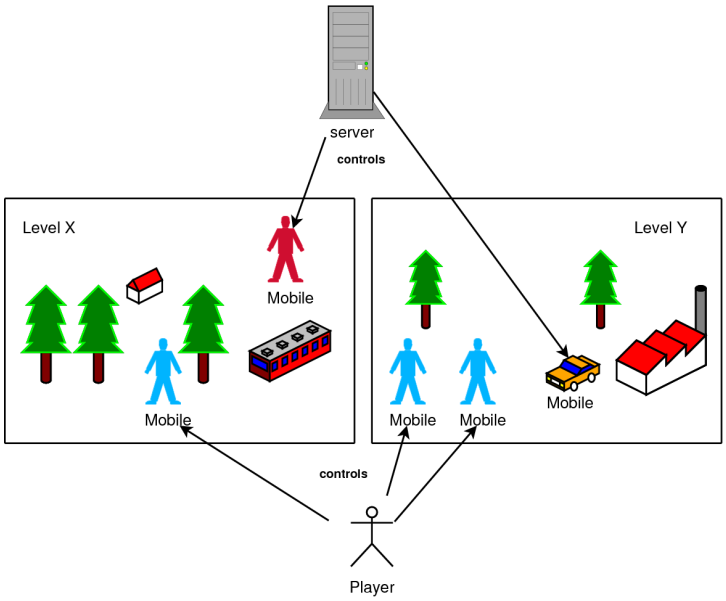


Figure 12.2: Players, Mobiles and Levels

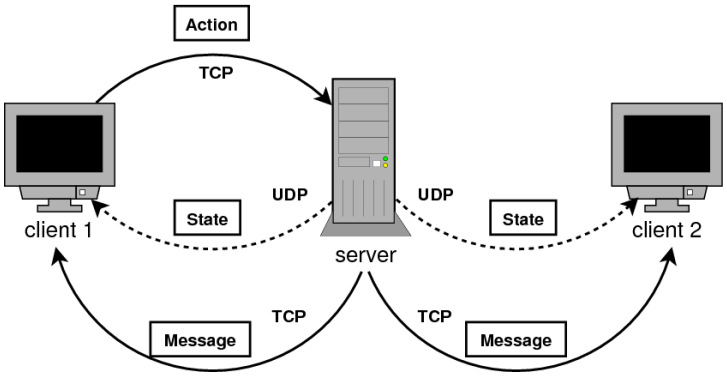


Figure 12.3: Actions, states and messages

Actions, messages and states are not classes; Tofu just uses raw strings for them.

For sending information, Tofu uses a mix of TCP and UDP sockets (For those that are not specialists in network, like me, TCP and UDP are two network protocols. The main difference between them is that, with TCP, you are garanteered that the data sent over the network arrives, whereas with UDP, the data may never arrive. However, UDP is faster). Tofu sends actions and messages with TCP, and states with UDP. If a state is lost, it will be counterweighed by interpolation.

12.1.3 Persistence: Data path and game path

Tofu automatically saves the game data using Cerealizer with it is shut down.

With Tofu, there is two path for saving data. **The normal Soya path** is used to load Images, Materials, Models, Worlds, Fonts, Sounds,... as usual. It is also used for loading new unmodified Levels, *i.e.* when a player enters in the level for the first time.

The Tofu game path is used for loading and saving players and modified levels, *i.e.* levels saved after a player enters in it and possibly alter the level. The game path is obtained by joining tofu.SAVED_GAME_DIR and tofu.GAME. tofu.SAVED_GAME_DIR is the directory where all games are saved, and tofu.GAME is the subdirectory where the current game is saved.

These variables can be used in various ways:

In single player mode one usually wants a game per player, which can be obtained by using the player name as the game name.

In single client-server mode :

For persistent universe a single game is needed (use any name of your choice).

For game with "match" a game is required for each match.

However, notice that a Tofu server cannot manage several games at a time. For instance, for a game with "match", you have to run one server for each match.

12.1.4 Single player, server and client modes

With Tofu, the **same** code is used for single player, server and client modes. This eases game programming, because you have a single program to write.

You can switch to any mode, by calling tofu.set_side("single"), tofu.set_side("server") or tofu.set_side("client").

12.2 Using the Tofu network engine

12.2.1 Setting up

First, you need to import Soya, Tofu and Cerealizer, and to define soya.path and tofu.SAVED_GAME_DIR (in a real game, you'll probably want to make tofu.SAVED_GAME_DIR user-configurable, *e.g.* by reading it from a configuration file; here we use a temporary directory for tutorial purpose).

```
import sys, os, os.path
import soya, soya.tofu as tofu, cerealizer

soya.path.append(os.path.join(os.path.dirname(sys.argv[0]), "data"))

tofu.SAVED_GAME_DIR = "/tmp/tofu_demo"
```

Warning: Tofu automatically imports and enables Cerealizer, and disables Pickle (see section 3.2) for security reasons; as a consequence, **all Soya data files MUST be in the Cerealizer file format!**

12.2.2 Creating the PlayerID class

The PlayerID class is used to identify Players. The default tofu.PlayerID has just a filename (*i.e.* the player's name) and a password. If you need additional attributes, you can extend PlayerID. For example, for adding a character_name attribute:

```
class PlayerID(tofu.PlayerID):
  def __init__(self, filename, password, character_name = "tux"):
    tofu.PlayerID.__init__(self, filename, password)
    self.character_name = character_name

  def dumps(self):
    return tofu.PlayerID.dumps() + len(self.character_name) + "\n" + self.character_name
  @classmethod
  def loads(Class, s):
    self = tofu.PlayerID.loads(Class, s)
    length = int(s.readline())
    self.character_name = s.read(length)
    return self

tofu.LOAD_PLAYER_ID = PlayerID.loads
```

The dumps method returns the PlayerID saved in a string, and the loads class method returns a PlayerID loaded from a file object s. The loads class method must be given to Tofu in the global variable tofu.LOAD_PLAYER_ID.

You may also use Cerealizer for saving and loading PlayerID. It is very handy for test, but should be avoided, because it might comprise the server security (Tofu uses Cerealizer only in the server->client direction, whereas PlayerIDs are sent from client to server. By sending e.g. Unique or SavedInAPath objects with the PlayerID, one might corrupt the game currenlty played on the server). To use Cerealizer, use the following dumps and loads methods:

```
  def dumps(self): return cerealizer.dumps(self)
  @classmethod
  def loads(Class, s): return cerealizer.load(s)
```

12.2.3 Creating the Player class

The Player class is used for representing the Player in server and single modes. Player is responsible for creating the Player first Mobiles, and putting them in the right Levels. Player can also stores Player stats, like score, that are managed on server-side.

When created, Player receive a PlayerID. Here is an example of a basic Player class:

```
class Player(tofu.Player):
  def __init__(self, player_id):
    tofu.Player.__init__(self, player_id)

    mobile = Mobile()
    mobile.level = tofu.Level.get("first_level")
    mobile.set_xyz(100.0, 0.0, 100.0)
    self.add_mobile(mobile)

tofu.CREATE_PLAYER = Player
cerealizer.register(Player, soya.cerealizer4soya.SavedInAPathHandler(Player))
```

This Player class creates a single Mobile, and puts it in the Level called "first_level", at the given coordinates. Player.add_mobile(mobi gives the control of the Mobile to the Player. Your Player class must be cerealizable.

Warning: You must not call Level.add_mobile (see below) in the Player class. Level.add_mobile will be automatically called when the Player logs in.

The following methods may be overriden in the Player class:

add_mobile(mobile) is called when the player gets the control of a new mobile

remove_mobile(mobile) is called when the player loose the control of a mobile (for example because the Mobile is dead)

login(socket, udp_address) is called when the Player connects to the game

logout(save=1) is called when the Player disconnects from the game ; the save argument indicates wether the Player should be saved or not.

killed(save=0) is called when the Player no longer controls any Mobile, and thus is considered as dead. The save argument indicates wether the Player should be saved or not; it defaults to false.

12.2.4 Creating the MainLoop class

Tofu provides a MainLoop that extends Soya's MainLoop with everything required for networking. The following methods can be overriden:

init_interface() is called when the game is starting, in single or client mode. It can be used for setting up the game interface, for example for creating a camera and a life-bar widget.

12.2.5 Creating the Level class

The Level class represents a game Level. It inherits from soya.World and tofu.Unique, and you have to extend this class. Your Level class must be cerealizable.

```
class Level(tofu.Level):
  def __init__(self):
    tofu.Level.__init__(self)

cerealizer.register(Level, soya.cerealizer4soya.SavedInAPathHandler(Level))
```

The following methods can be overriden:

add_mobile(mobile) is called when a mobile is added in the Level

remove_mobile(mobile) is called when a mobile is removed from the Level

set_active(active) is called when the Level is activated (active is true) or inactivated (active is false). A level is considered as active if and only if there is at least one Mobile controlled by a human Player in the Level.

12.2.6 Creating the Mobile class

The Mobile class represents anything that evolves or changes in a Level. In particular, this definition includes the characters controlled by the Player, as well as bots (characters controlled by computer) and several traps (such as moving platforms). Mobile inherits from soya.World and tofu.Unique, and you have to extend this class. Your Mobile classes must be cerealizable.

Mobile has the following interesting attributes:

bot true if the Mobile is a bot, *i.e.* the Mobile is controlled by a computer and not a Player.

local true if the Mobile is controlled locally, and not by a remote server or client.

player_name the filename of the Player that controls the Mobile, if any. For bots, an empty string.

level the Level the Mobile is inside.

12.2.6.1 Owning and loosing control

A Mobile can be controlled by different Player or computer during its life. To give the control of a Mobile to a Player, call Player.add_mobile(mobile); and call Player.remove_mobile(mobile) for removing the control and turning the Mobile into a bot. In client-server mode, Player.add_mobile and Player.remove_mobile must be called server-side, usually it is called in Mobile.do_collision().

The following methods are related to the control transfert, and should be overriden:

control_owned() is called when the current program gets the control of the Mobile, *i.e.* Mobile.local becomes true. For example, if the Mobile is controlled by a local Player (*i.e.* if Mobile.bot is false), this method can be overriden in order to makes the Camera following and looking at the Mobile.

control_lost() is called when the current program loose the control of the Mobile, *i.e.* Mobile.local becomes false.

12.2.6.2 Generating actions

Actions are the action the human player or a bot decide to perform. Actions are generated on client-side for Mobile controlled by human Players, and on server-side for bots.

Tofu doesn't have an action class; actions are simply represented by raw string. For instance, you can use "<" for representing the "start turning left" action, ">" for "start turning right", "J" for "start jumping", and so on. More complex action examples can be "U12" for "use the magical item of UID 12".

The following methods are used for generating actions:

generate_actions() is called every round, and is in charge of generating the actions for the Mobile. You have to override this method, in order to generate actions from keyboard and mouse events (for Mobiles controlled by human Players) or artificial intelligence (for bots). When actions are generated, you must call send_action(action) for each of them (you can call send_action() several time in generation_actions()).

send_action(action) sends the given action. The action argument must be a string.

12.2.6.3 Doing actions

Actions are performed sever-side, by the following method that must be overriden:

do_action(action) is called for every action. The action argument is the string that generate_actions() has given to send_action(). For instance, if the action is "J" and corresponds to "start jumping", do_action() should modify the Mobile speed, in order to make it jump.

When the current state of the object has been modified, you should call the following method:

set_current_state_importance(importance) sets the importance of the current Mobile state. In client-server mode, the importance is used as a hint for determining which states should be sent through network. Importance can be 0 (no importance at all), 1 (small importance, corresponding to a change that the client is able to interpolate) or 2 (important change, requiring to send a state as soon as possible).

12.2.6.4 Generating and applying states

In client-server mode, states are send from the server to the client. The state usually indicates the position and the orientation of the Mobile. However, a state is not sent for each round. Tofu automatically determines for which round a state is sent, using the values given to set_current_state_importance() as hints.

The following methods are related to states and must be overriden:

get_network_state() is called on server-side when Tofu decides to send a state. It should returns the states of the Mobile, as a string.

read_network_state(file_object) is called on client-side when Tofu receives a state. It should read the states from the given file object.

12.2.6.5 Dealing with physics

Physics computation is performed both client and server side. In client mode, do_physics is in charge of interpolating between states.

You must override the following method:

do_physics() is called every round, and is in charge of doing physics computation. For instance, if the Mobile is jumping and thus has a positive vertical speed, do_physics should apply the vertical speed.

12.2.6.6 Dealing with collisions

Collisions we are speaking about here are collisions between Mobiles, or important collision that must be handled server-side. Normal "collision", like a collision with a static wall, can be dealt with during the physics step.

do_collisions() is called every round, and is in charge of checking important collisions. Important collisions include collision between two Mobiles (*e.g.* a character Mobile touching and taking a bonus Mobile, or two characters colliding), or any collision that has an influence on the game (loosing life point, being teleported to a new Level,...).

In do_collision, you may want to call set_current_state_importance() too (see section 12.2.6.3).

Changing the Level of a Mobile should occur in do_collision. It can be done as following:

```
mobile.level.remove_mobile(mobile)
new_level.add_mobile(mobile)
```

12.2.6.7 Generating messages

Message are similar to action, but they are sent from the server to the clients. Messages often results from do_collision(), for example if a Mobile has taken a life bonus, all clients must be informed that the Mobile's life has increased. This can be done through message.

The following method is used for sending a message:

send_message(message) sends the given message. The message argument must be a string, *e.g.* you may use "L9" as a message meaning "the number of life of the Mobile is now 9".

12.2.6.8 Doing messages

Messages are performed client-side, by the following method that must be overriden:

do_message(message) is called for every message. The message argument is the string that has been given to send_message(). For instance, if the message is "L9" and corresponds to "the number of life is now 9", do_message() may update a lifebar.

12.2.6.9 Conclusion

The following table indicates which methods is called on which mode.

methods (*= called every round)	single mode	server mode	client mode
control_owned	X	X	X
control_lost	X	X	X
generate_actions (*)	X	X (for bots)	X (for human player)
send_action	X	X	X
do_action	X	X	
do_physics (*)	X	X	X
set_current_state_importance	X	X	X
get_network_state		X	
read_network_state			X
do_collisions (*)	X	X	
send_message	X	X	
do_message	X		X

12.2.6.10 Tofu default implementations

Tofu provides several default Mobile implementations with interpolation:

12.2.6.10.1 SpeedInterpolatedMobile is a Mobile that has is moved according to a speed.

The speed is not just a vector, but a CoordSystSpeed, a special object that implement a "speed matrix". Any modification applied to the speed will be performed on the Mobile **every rounds**. For example, for making the Mobile go forward:

```
mobile.speed.z = -0.1
```

For making the Mobile turning:

```
mobile.speed.turn_y(10.0)
```

To stop the Mobile, just reset the speed:

```
mobile.speed.z = 0.0
mobile.reset_orientation_and_scaling()
```

SpeedInterpolatedMobile has the following noticeable attributes:

speed the speed (see above).

last_state the last state of the Mobile. last_state is a CoordSystState object, which has a position but also an orientation and a scaling.

next_state the next state of the Mobile, *i.e.* the state it will after at the end of the current round. next_state is a CoordSystState object. next_state is often used for collision detection in do_physics.

12.2.6.10.2 AnimatedMobile AnimatedMobile is a Mobile that uses animations from an AnimatedModel. The current animation is automatically sent through network using states.

AnimatedMobile has the following attributes:

animable the object that is animated and has an AnimatedModel. If not given, it defaults to the Mobile itself. You can use a different object by setting the "animable" attribute to any other Body.

AnimatedMobile has the following method:

set_animation(animation) starts playing the given animation, and stops playing the previous one.

12.2.6.10.3 RaypickCollidedMobile RaypickCollidedMobile is a Mobile that uses raypicking for basic collision detection (walls and ground).

12.2.6.10.4 RaypickCollidedMobileWithGravity Same as RaypickCollidedMobile, but with gravitational force.

game examples	Mobile classes to extend
a racing game	SpeedInterpolatedMobile, RaypickCollidedMobileWithGravity
a spatial simulation	SpeedInterpolatedMobile, RaypickCollidedMobile
a character-based game	SpeedInterpolatedMobile, RaypickCollidedMobileWithGravity, AnimatedMobile

12.2.7 Starting the game

12.3 About Tofu sources

Tofu sources are entirey in Python. The source (ab)uses of a technic I called "side-oriented programming", which defined in the soya.tofu.sides module. When a method is prefixed by "@side("XXX")" (where XXX is single, server, client, or any combination of them), it means that the method exists only in the corresponding modes. The implementation of side-programming itself is in sides.py.

12.4 Object reference

Chapter 13

Using Soya with...

13.1 External GUI systems (Tk, Wx,...)

Many GUI systems provide their own main loop. In this case, you should create the Soya's MainLoop object as usual, but instead of calling `MainLoop.main_loop()`, you should call `MainLoop.update()` repeatedly, usually in a kind of timer, if possible about once per 25 or 30 millisceonds. `MainLoop.update` manages time similarly than `MainLoop.main_loop`, but it cannot regulate time, as a consequence, you should use `MainLoop.main_loop` whenever possible.

13.1.1 Tkinter

Using Tkinter, this can be done as following, with `after`:

```
class Window(Tkinter.Tk):
    def __init__(self):
        Tkinter.Tk.__init__(self)
        self.after(30, self.update_soya)

    def update_soya(self):
        self.after(30, self.update_soya)
        soya.MAIN_LOOP.update()
```

See tutorial soya-with-tk-1.py for an example.

13.2 PyGame

Using Soya on PyGame surface is possible, by initializing first the PyGame surface, and the initializing Soya as following:

```
soya.init(create_surface = 0)
```

However, this is of little interest, since PyGame doesn't seem to be able to blit on OpenGL surface.

Chapter 14

Extending Soya in Python

14.1 Direct calls to OpenGL

14.2 Writing new Materials

14.3 Writing new CoordSysts

14.4 Object reference

Chapter 15

Hacking the Soya sources

The source of the development version of Soya can be found on our Subversion repository: `http://gna.org/svn/?group=soya`. Soya is written in Pyrex, Python and there is still a small part in C.

You can propose patches on the Soya mailing list. However, I (=Jiba, the Soya maintainer) am a very occupied guy with a small memory :-). If you don't get any feedback after while, it probably doesn't mean you proposition has been rejected, but rather that no one got the time for looking at it, and it has been forgotten... in this case just insist!

Do not hesitate to ask for Subversion write access, too.

Hint: Compiling Soya takes quite a long time. However, you can speed up the compilation by disabling GCC's optimization, as following:

```
export CFLAGS=-O0
```

15.1 Dealing with Segfaults

Sometimes, Soya may crash and cause segmentation faults. It corresponds to error occurring at the Pyrex level. In this case, do as following to obtain a backtrace :

```
gdb python
run ./your_soya_script.py
[wait until the script crashes]
bt
[the backtrace is here]
```