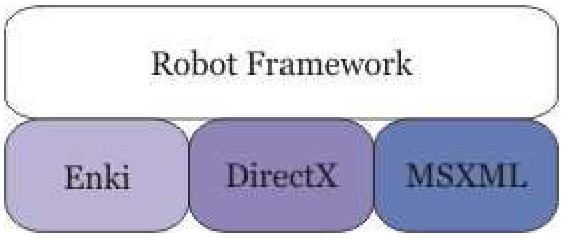
Robot Framework

author: Peter Černo
e-mail: petercerno[at]gmail.com

- **platform:** Win32
- **language:** C++
- **development tools:** Microsoft Visual C++ 2005, Microsoft DirectX SDK

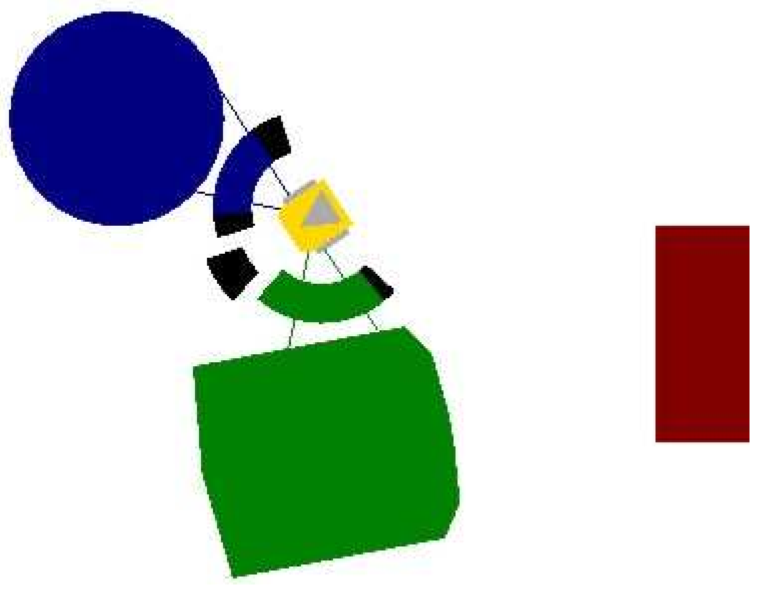
Robot Framework is a framework for simple 2D physics-based robot simulations. It a graphical extension of the open-source **Enki** project based on DirectX.
If you want to experiment with simple robots using neural networks, genetic algorithms or other techniques, you do not need to start from scratch. This framework is both simple to learn in a short time and complex enough to make reliable simulations. I have also created a simple application to demonstrate the power of this framework.

What is Robot Framework (RFW)?



You can imagine Robot Framework as a box that encapsulates **Enki**, **DirectX** and **MSXML** components. RFW provides common interface for all these three parts, but on the other hand it separates their implementation. It uses **strategy pattern**, so you can dynamically (at the run-time) change the way of rendering the objects or loading them. One of the main purposes of the Robot Framework is to visualize simulations made by **Enki** simulator.

What is Robot Neural Simulator (RNS)?



Robot Neural Simulator is a concrete application of **Robot Framework**. I have created this simulator to illustrate the power of the framework. The point of the simulator is this: At first a robot is trained by a user who controls the robot with keyboard. The robot generates a log file where it writes all the data on the sensors and the actions made by the user. The log file is then moved to MATLAB where a neural network is trained to give corresponding actions on given sensor data. This neural network is then moved to the simulator. You can easily turn the control over to this neural network and observe how precisely can the robot reproduce the actions made by the user. To gain a better understanding of how it works I recommend **downloading** this simulator and trying some of the **experiments**.

What is An?

An is a C++ library which provides basic useful bricks for scientific use. This includes the Color type, a collection of several random distributions, some numeric functions such as exponential decay with varying time step, and Formula, a formula parser, byte compiler and interpreter. **Enki** depends on this library.

What is Enki?

Enki is a fast 2D physics-based robot simulator written in C++. It is able to simulate kinematics, collisions, sensors and cameras of robots evolving on a flat surface. It also provides limited support for friction. It is able to simulate groups of robots hundred times faster than real-time on a modern desktop computer.
I had to do some minor modifications to the original Enki library in order to avoid some design problems.
Enki is a part of **Robot Framework** and serves mainly as a physical engine.

What is DirectX?

Microsoft DirectX is a set of low-level APIs for creating games and other high-performance multimedia applications. It includes support for high-performance 2D and 3D graphics, sound, and input.
DirectX is a part of **Robot Framework** and serves mainly as a graphical engine.
You should download and install: **Microsoft DirectX SDK**.

What is MSXML?

Microsoft XML Core Services (MSXML for short) consist of preprogrammed classes and functions that contain code to access and manipulate information in an XML document.
MSXML is a part of **Robot Framework** and serves mainly as an input/output engine.

The documentation that follows is divided into two parts:

In the first part the Robot Neural Simulator application is described from the user point of view. The best way to understand the simulator is to try some of the experiments.

In the second part the basic interfaces and classes of Robot Framework and Robot Neural Simulator are described. This part is destined for readers who are interested in the source code of the application or just want to write their own application.

I. PART

Experiments

At first you have to **download** Robot Neural Simulator (**RNS**) in order to perform these experiments.

1st experiment

In this experiment I have tried to train a robot to avoid the bounds of a rectangular area. When training it I only moved straight forward or turned left. The robot should not be able to turn right. To see the results of this experiment just follow these steps:
1. Run the Robot Neural Simulator application.
2. To the *File Name* edit box write "Scene1.xml" and click on the *Load Scene* button.
3. To the *File Name* edit box write "Robot1.xml" and click on the *Load Robot* button.
4. To the *File Name* edit box write "Robot1Network1.xml" and click on the *Load Network* button.

From the Robot1Network1.xml file you can find out that this neural network has two layers: first input layer has 2 neurons and second hidden layer has 4 neurons. The only transfer function used is *logsig*. The robot has six infrared sensors so this network has six input cells and four output cells. (Each output cell is associated with different key: up, down, left, right)

If everything is loaded successfully then just click on the *Neural Network* radio button and observe how the robot behaves.
If you want to change the position of the robot just click on the *Keyboard* radio button and move the robot with keys.

2nd experiment

In this experiment I have tried to train a robot to keep a certain distance from the bounds of a rectangular area. Again I only moved straight forward or turned left. To see the results of this experiment just follow the steps of the 1st experiment except that you have to load "Robot1Network2.xml" neural network.

3rd experiment

In this experiment I have tried to train a robot to move in a narrow corridor. Again I only moved straight forward or turned left.
The files you have to load are:
Scene: "Scene2.xml"
Robot: "Robot2.xml"
NeuralNetwork: "Robot2Network1.xml" or "Robot2Network2.xml".

4th experiment

In this experiment I have tried to train a robot to move in a narrow corridor. But now the robot is able to turn in both directions.
The files you have to load are:
Scene: "Scene3.xml"
Robot: "Robot3.xml"
NeuralNetwork: "Robot3Network1.xml" or "Robot3Network2.xml".

How to train a robot

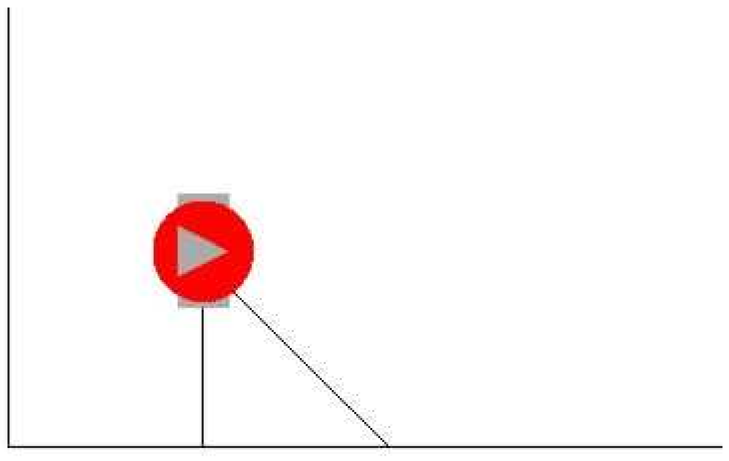
In order to train a robot you must have MATLAB with Neural Network toolbox installed on your computer. I have created two scripts: **feedforward.m** and **savenetwork.m** which are used in this tutorial. Please copy them into your MATLAB work directory.

The first step you have to undertake is designing a scene and a robot. In this tutorial we are working with a simple empty rectangular area and a simple robot with six infrared sensors.

```
<world width = "80.0" height = "100.0"
  startX = "10.0"
  startY = "10.0"
  startAngle = "0.0">
</world>
<neuralrobot wheelDistance = "5.2" robotSpeed = "8.0">
  <object x = "20.0" y = "20.0" r = "2.6"
    angle = "0.0"
    height = "3.0"
    mass = "50.0"
    staticFrictionThreshold = "1.0"
    viscousFrictionTau = "0.5"
    viscousMomentFrictionTau = "0.0"
    collisionAngularFrictionFactor = "0.01"
    color = "Red">
  </object>
  <sensors>
    <irsensor x = "1.0" y = "1.0"
      height = "1.8"
      orientation = "0.0"
      range = "20.0"
      aperture = "0"
      rayCount = "1"
      sensorResponseKernel = "simple" />
    <irsensor x = "1.0" y = "-1.0"
      height = "1.8"
      orientation = "0.0"
      range = "20.0"
      aperture = "0"
      rayCount = "1"
      sensorResponseKernel = "simple" />
    <irsensor x = "0.5" y = "1.0"
      height = "1.8"
```

```
        orientation = "45.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "0.5" y = "-1.0"
        height = "1.8"
        orientation = "-45.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "0.0" y = "1.0"
        height = "1.8"
        orientation = "90.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "0.0" y = "-1.0"
        height = "1.8"
        orientation = "-90.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
    </sensors>
  </neuralrobot>
```

When loaded successfully in Robot Neural Simulator application you should get the following scene:



We are going to train the robot to keep a certain distance from the bounds of the rectangular area. At first we have to specify the name of the log file.
For instance "RobotLog.log" is a good choice. It is necessary that this file does not exist. If it does not then write "RobotLog.log" to the *File Name* edit box.

When you are ready then click on the *Start Log* button. Now the log file is created and the data on the sensors + your actions are being written to this log file in certain time intervals. It is time to move the robot around the rectangular area. After some time if you think that the training is at the end just click on the *Stop Log* button.

Here is a piece of my own log file:

```
00000000 00000000 00000000 013.6866 00000000 057.8567 1.0 0.0 0.0 0.0
00000000 00000000 00000000 016.3869 00000000 053.2265 1.0 0.0 0.0 0.0
00000000 00000000 00000000 014.8516 00000000 055.7677 1.0 0.0 0.0 0.0
00000000 00000000 00000000 015.7606 00000000 056.5351 1.0 0.0 0.0 0.0
00000000 00000000 00000000 015.6614 00000000 054.2562 1.0 0.0 0.0 0.0
0.689666 0.616129 00000000 015.2781 00000000 059.8924 1.0 0.0 0.0 0.0
0.898523 0.780486 00000000 016.7938 00000000 059.1452 1.0 0.0 0.0 0.0
01.09827 01.28201 00000000 014.9952 00000000 053.4388 1.0 0.0 0.0 0.0
01.63977 01.44604 00000000 015.4153 00000000 061.2761 1.0 0.0 0.0 0.0
01.67267 02.05729 00000000 017.3189 00000000 056.6577 1.0 0.0 0.0 0.0
03.09586 03.72857 00000000 017.3809 00000000 062.4889 1.0 0.0 0.0 0.0
005.0089 05.03493 00000000 07.21895 00000000 062.8666 1.0 0.0 1.0 0.0
06.52008 06.79527 00000000 04.09871 00000000 046.0674 1.0 0.0 1.0 0.0
06.03534 08.55567 00000000 010.7992 00000000 026.0934 1.0 0.0 1.0 0.0
04.97141 009.1698 00000000 018.5785 00000000 012.4829 1.0 0.0 1.0 0.0
03.89935 08.00387 00000000 0029.847 00000000 05.10918 1.0 0.0 1.0 0.0
01.00478 03.52438 00000000 045.4979 00000000 0019.916 1.0 0.0 1.0 0.0
00000000 00.77206 00000000 048.2736 00000000 042.9448 1.0 0.0 1.0 0.0
00000000 00000000 00000000 044.5865 00000000 060.3912 1.0 0.0 1.0 0.0
00000000 00000000 00000000 032.9384 00000000 077.0842 1.0 0.0 1.0 0.0
00000000 00000000 00000000 0012.801 00000000 069.5409 1.0 0.0 0.0 0.0
00000000 00000000 00000000 0015.052 00000000 066.0309 1.0 0.0 0.0 0.0
```

As you can see there are six columns for sensors and four columns for user's actions (1.0 = a key is down, 0.0 = a key is up). An important thing to note is that the order of rows is irrelevant. Each row is independent and is representing a self-containing piece of information. You can create few log files and then you can merge them.
When training a robot you have to realize that the only thing the robot can see are the six numbers on its sensors. So if you make different actions on very similar situations (from the robot point of view) then it is very likely that the robot will not be able to understand what you want.

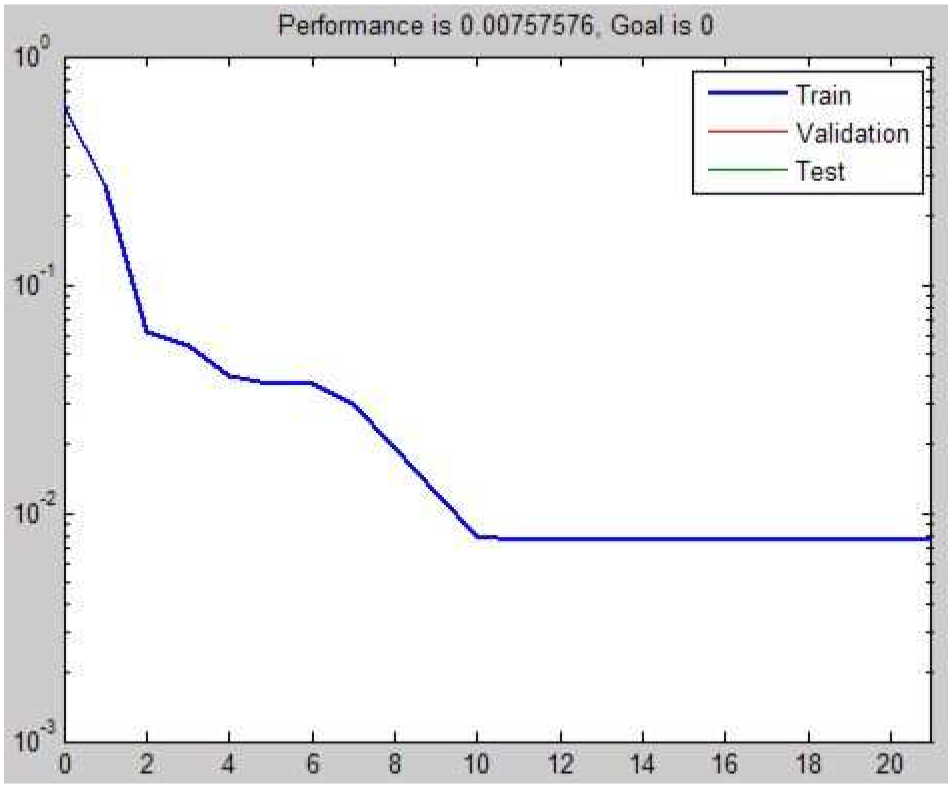
Now we are going to transfer this log file into the MATLAB. You do not need to copy all the lines. It is recommended to delete those lines where the robot is not moving (i.e. the last four columns are zero).

In MATLAB you have to create two matrices. The first matrix *X* contains the columns 1 to 6 of the log file and the second matrix *Y* contains the columns 7 to 10 of the log file. If you have these matrices set then all you need to do is to write the following command:

```
[net, mean, std, PCAmatrix] = feedforward(X.', Y.');
```

Note that X.' is a transposition of matrix X.

The results of the training are showed in a graph like this one:



The result network is stored in the *net* variable. The variables *mean*, *std* and *PCAmatrix* are used to correlate the input data.

You can also define your own architecture of the network. All you have to do is to open "feedforward.m" file and modify the default settings - the row:

```
net = newff(minmax(Xtrans), [2 m], {'logsig', 'logsig'});
```

The *m* variable is the number of output cells (in our case m = 4).
I have tried only these architectures in my own experiments:

```
  [2 m],   {'logsig', 'logsig'}
  [4 m],   {'logsig', 'logsig'}
  [2 4 m], {'logsig', 'logsig', 'logsig'}
  [4 4 m], {'logsig', 'logsig', 'logsig'}
```

The last step is saving this network into an xml file:

```
savenetwork('RobotNetwork.xml', net, mean, std, PCAmatrix);
```

This command saves the network to the "RobotNetwork.xml" file. You can load this network in Robot Neural Simulator application and try how it works. That is all.

But remember, this network is closely associated with the robot you have trained. You cannot use this network with other robots.

First results are often unsatisfactory. You have to gain some practice and experience in order to become a good trainer.

Results and conclusion

One of the main limitations of the simulator is that the robot (and also the feed forward neural network) does not contain any internal states. This means for instance that the robot does not know what it has done few milliseconds before. Its actions are based only on the present data on the sensors.

Another limitation is that the circular cameras are not considered during training. This is mainly because the camera gives you too many information at once. At first you have to transform the raw data from the camera into the form that is appropriate for training neural networks. It is not easy to make up such transformation so I have not implemented one yet.

If we take into account all these constraints then we get some unpleasant consequences: For instance the robot is not able to learn drawing back from a dead end. The robot cannot stop because if it stops then it will never start running again etc.

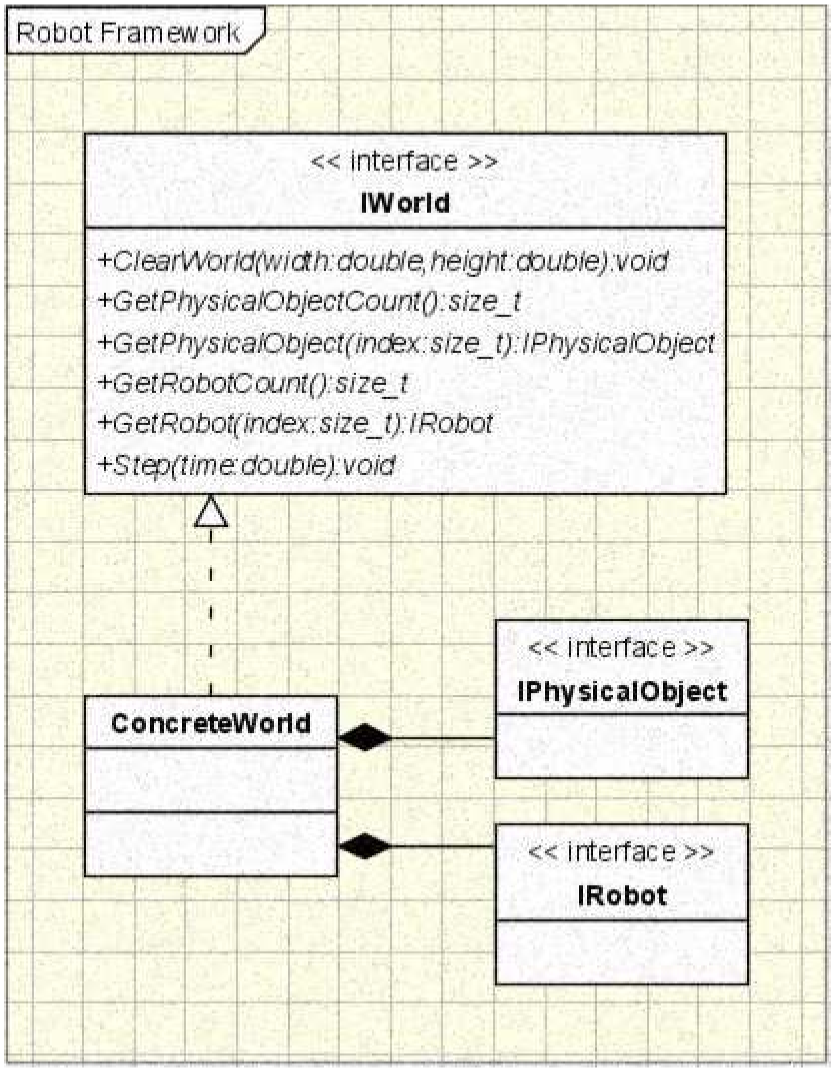
Despite all these restrictions I consider this simulator successful. It has proved that it is possible to train a robot to do some actions only by demonstrating these actions. You need not to specify all the details to the robot.

I think that this technique can be also applied in the physical robots. If you write the control unit of a robot by hand you often cannot treat all the possibilities that can happen. Neural networks are more robust than if-then-else code. They can give good results even if the input data are imprecise.

II. PART

Robot Framework

Robot Framework (**RFW**) is a group of interfaces and classes in the RFW namespace. The core interfaces are: *IWorld*, *IPhysicalObject* and *IRobot*.

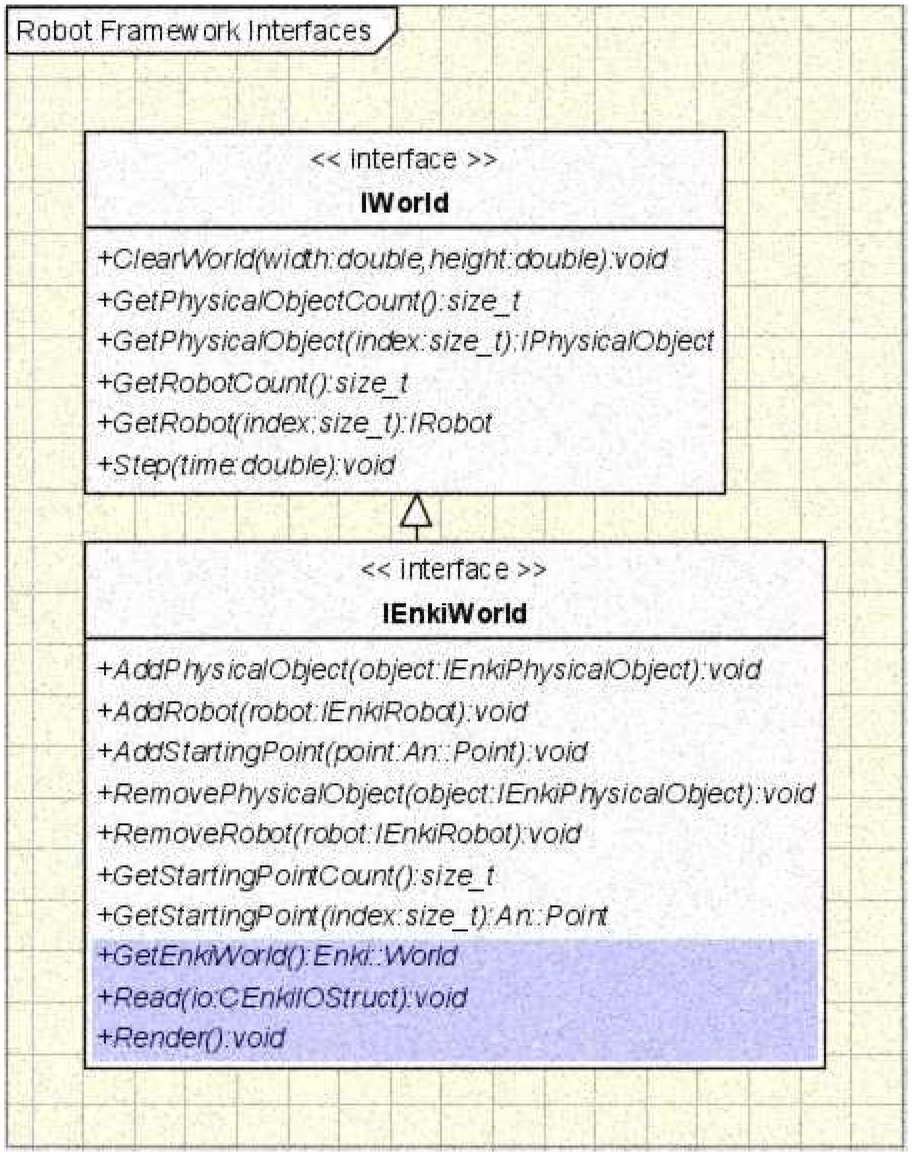

Robot Framework
<< interface >>
IWorld
+ClearWorld(width:double,height:double):void
+GetPhysicalObjectCount():size_t
+GetPhysicalObject(index:size_t):IPhysicalObject
+GetRobotCount():size_t
+GetRobot(index:size_t):IRobot
+Step(time:double):void
ConcreteWorld
<< interface >>
IPhysicalObject
<< interface >>
IRobot

Classes that implements *IWorld* contain (are composed of) *IPhysicalObject*s and *IRobot*s. Note that if you destroy *IWorld* then *IPhysicalObject*s and *IRobot*s are also destroyed. If you think of *IWorld* you should remember that it is only a container of objects and robots. It has no other responsibilities.

These interfaces are too general so they are not used directly. We have to add some dependencies in order to get useful interfaces.

RFW Interfaces

At first *IEnkiWorld* interface is introduced:



Robot Framework Interfaces
<< interface >>
IWorld
+ClearWorld(width:double,height:double):void
+GetPhysicalObjectCount():size_t
+GetPhysicalObject(index:size_t):IPhysicalObject
+GetRobotCount():size_t
+GetRobot(index:size_t):IRobot
+Step(time:double):void
<< interface >>
IEnkiWorld
+AddPhysicalObject(object:IEnkiPhysicalObject):void
+AddRobot(robot:IEnkiRobot):void
+AddStartingPoint(point:An::Point):void
+RemovePhysicalObject(object:IEnkiPhysicalObject):void
+RemoveRobot(robot:IEnkiRobot):void
+GetStartingPointCount():size_t
+GetStartingPoint(index:size_t):An::Point
+GetEnkiWorld():Enki::World
+Read(io:CEnkiIOStruct):void
+Render():void

The most important methods are: *GetEnkiWorld*, *Read* and *Render*.
GetEnkiWorld method connects *IEnkiWorld* to **Enki**, *Read* connects *IEnkiWorld* to **MSXML** and *Render* connects *IEnkiWorld* to **DirectX**.

Other important methods are: *AddPhysicalObject* and *AddRobot*. These methods should not be called directly by the user. Instead they should be placed in the constructor of the object (robot).

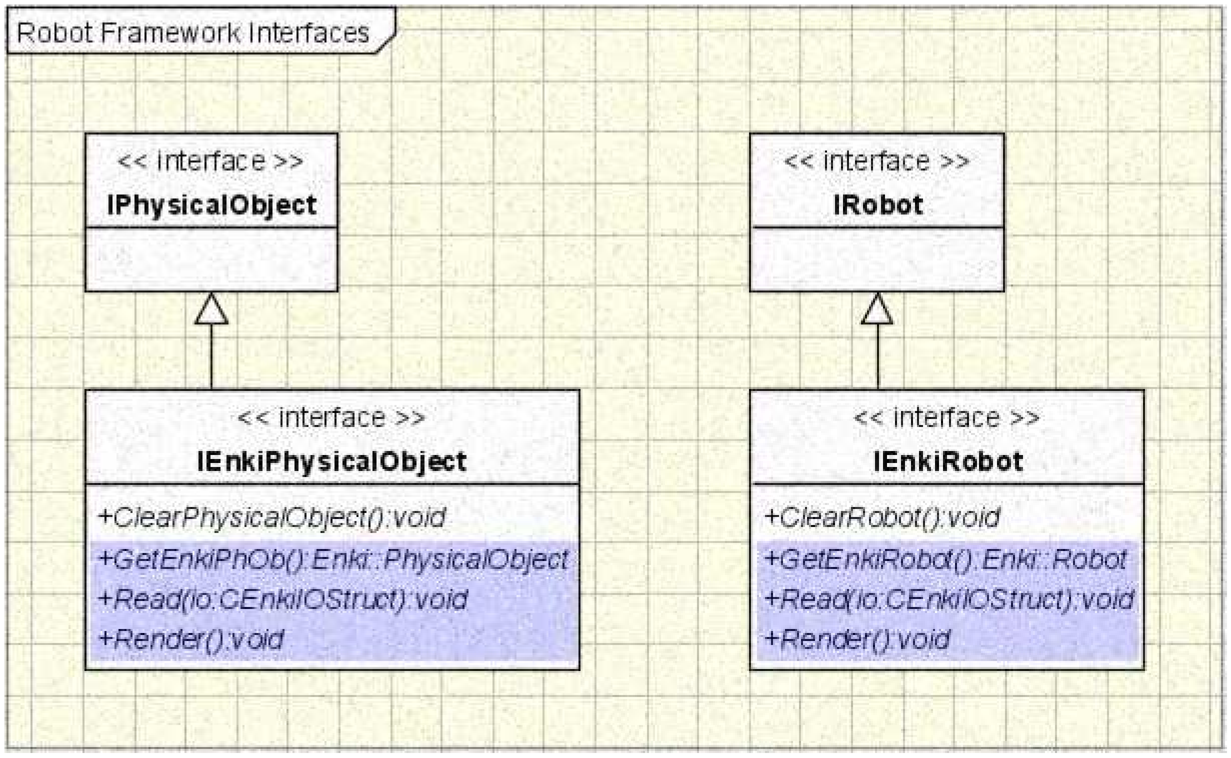
For instance the creation of CEnkiRobot should look like this:

```
m_EnkiRobot = new RFW::CEnkiRobot(m_EnkiWorld);
```

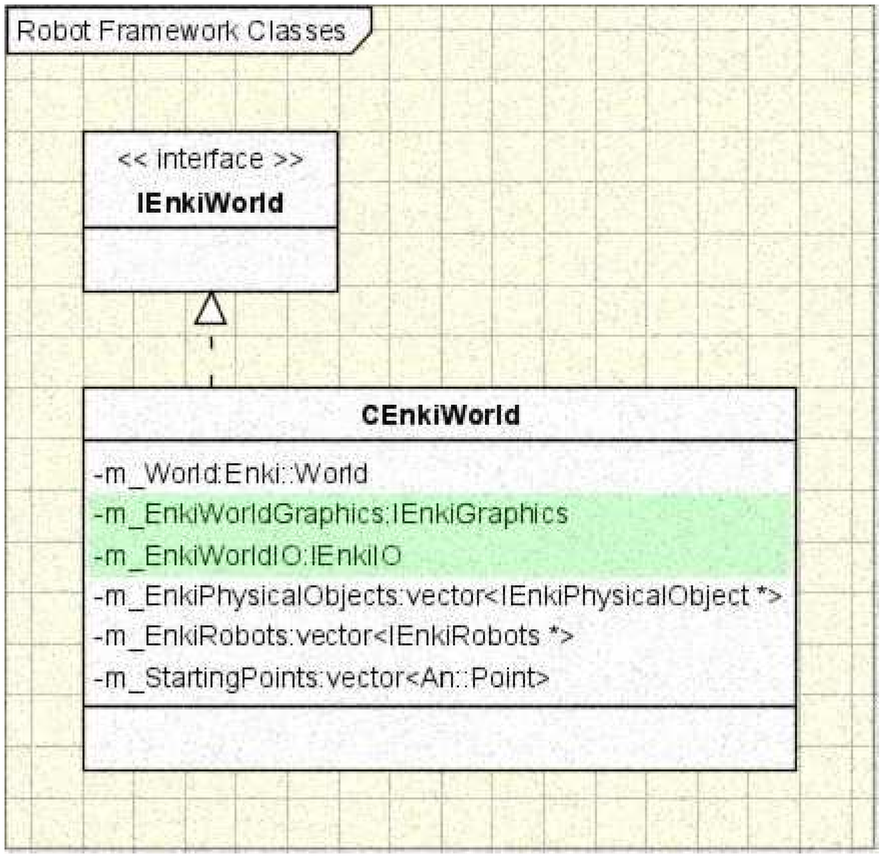
This code creates an instance of *RFW::CEnkiRobot* class and then it inserts this instance to the m_EnkiWorld and makes all other necessary things.
m_EnkiRobot now belongs to m_EnkiWorld so it is prohibited to call destructor on m_EnkiRobot. Instead you should call m_EnkiWorld.RemoveRobot method.

Note on naming convention: We write *IEnkiWorld* in order to underline the fact that this interface depends somehow on Enki physical engine. Despite this interface is full-valued RFW interface. This also relates to all other interfaces (classes) starting with IEnki (CEnki) prefix. Enki classes always start with Enki:: prefix.

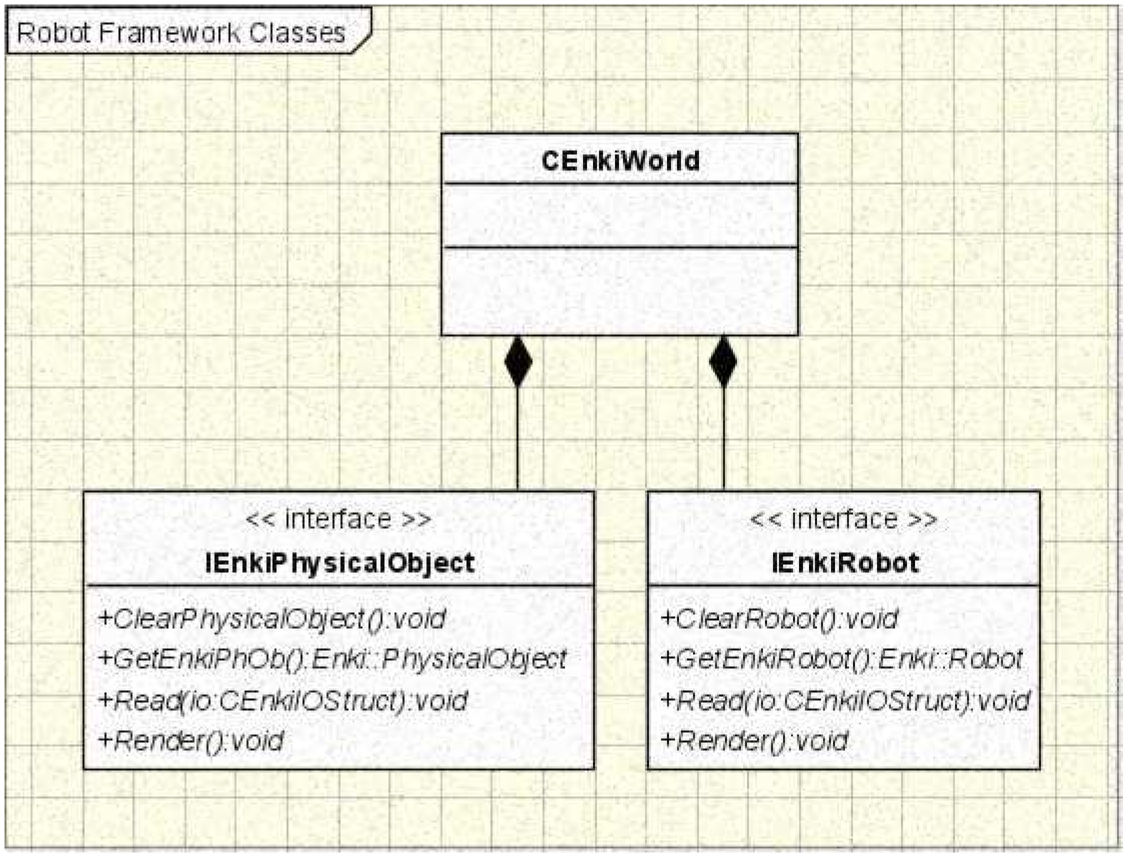
IEnkiPhysicalObject and *IEnkiRobot* interfaces are analogous:


Robot Framework Interfaces
<< interface >>
IPhysicalObject
<< interface >>
IRobot
<< interface >>
IEnkiPhysicalObject
+ClearPhysicalObject():void
+GetEnkiPhOb():Enki::PhysicalObject
+Read(io:CEnkiIOStruct):void
+Render():void
<< interface >>
IEnkiRobot
+ClearRobot():void
+GetEnkiRobot():Enki::Robot
+Read(io:CEnkiIOStruct):void
+Render():void

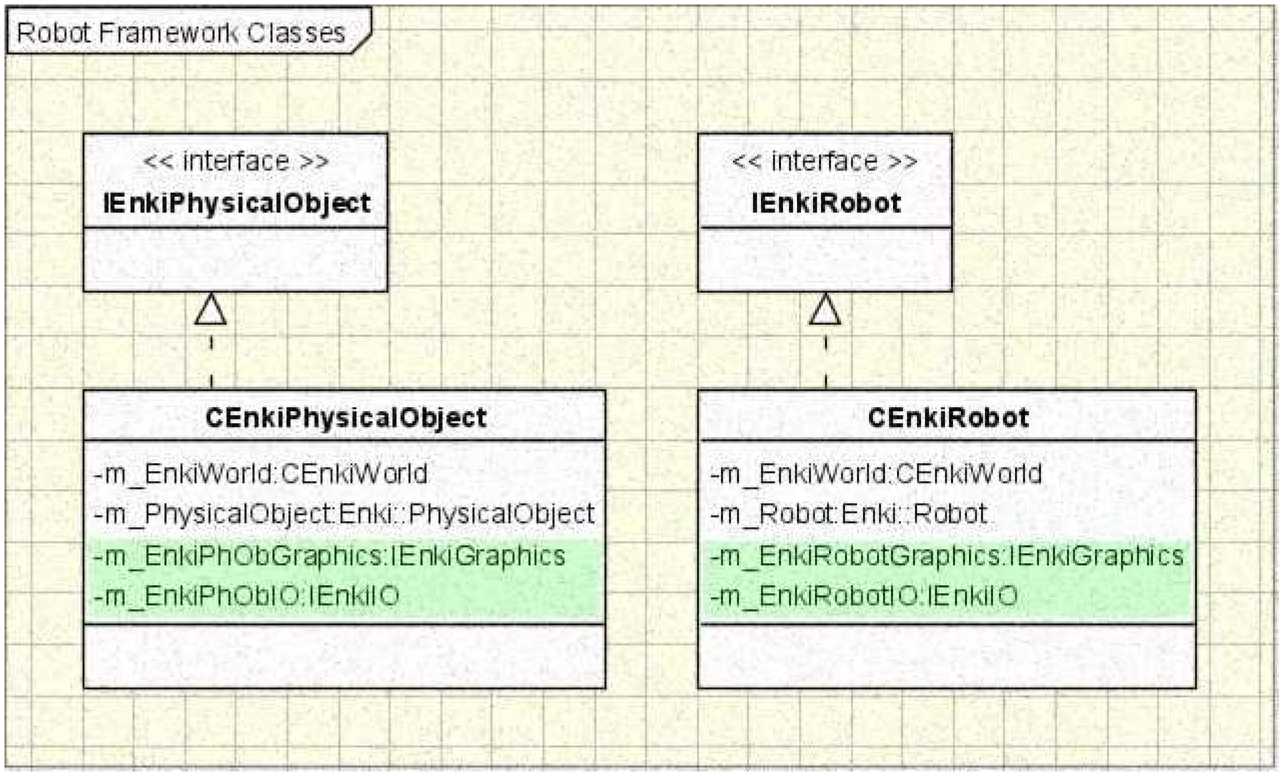
RFW Classes


Robot Framework Classes
<< interface >>
IEnkiWorld
CEnkiWorld
-m_World:Enki::World
-m_EnkiWorldGraphics:IEnkiGraphics
-m_EnkiWorldIO:IEnkiIO
-m_EnkiPhysicalObjects:vector<IEnkiPhysicalObject *>
-m_EnkiRobots:vector<IEnkiRobots *>
-m_StartingPoints:vector<An::Point>

CEnkiWorld class implements *IEnkiWorld* interface. It is something like a container of *IEnkiPhysicalObject*s and *IEnkiRobot*s.


Robot Framework Classes
CEnkiWorld
<< interface >>
IEnkiPhysicalObject
+ClearPhysicalObject():void
+GetEnkiPhOb():Enki::PhysicalObject
+Read(io:CEnkiIOStruct):void
+Render():void
<< interface >>
IEnkiRobot
+ClearRobot():void
+GetEnkiRobot():Enki::Robot
+Read(io:CEnkiIOStruct):void
+Render():void

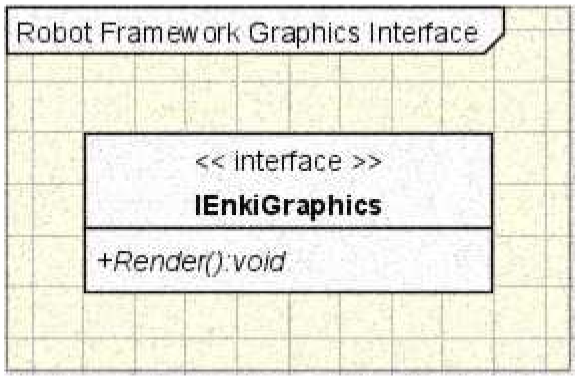
CEnkiWorld class is also able to generate graphic output (*Render()*) and load data from an xml document (*Read(CEnkiIOStruct &)*). The responsibility for rendering and loading data is moved to other classes due to *strategy pattern* (see the green rectangle). *CEnkiWorld* class knows only the interfaces: *IEnkiGraphics* and *IEnkiIO*.



These classes are similar to *CEnkiWorld* class. *CEnkiRobot* class is only a demonstration class. It cannot be used directly. It only shows you how to create your own robotic classes. The reason is simple: At first you have to write a concrete Enki robot class that extends *Enki::Robot* class. Then you can write a wrapper RFW class.

Classes like *CEnkiWorld*, *CEnkiPhysicalObject*, *CEnkiRobot* should not be subclassed.

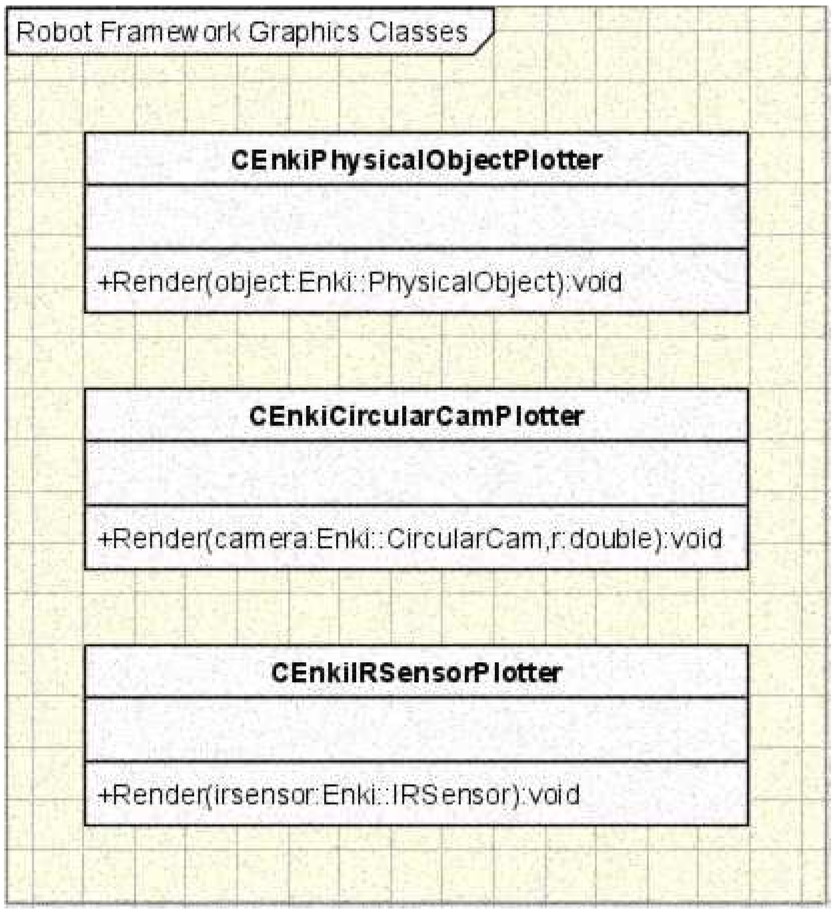
RFW Graphics



IEnkiGraphics is an interface for all classes that are responsible for rendering something.

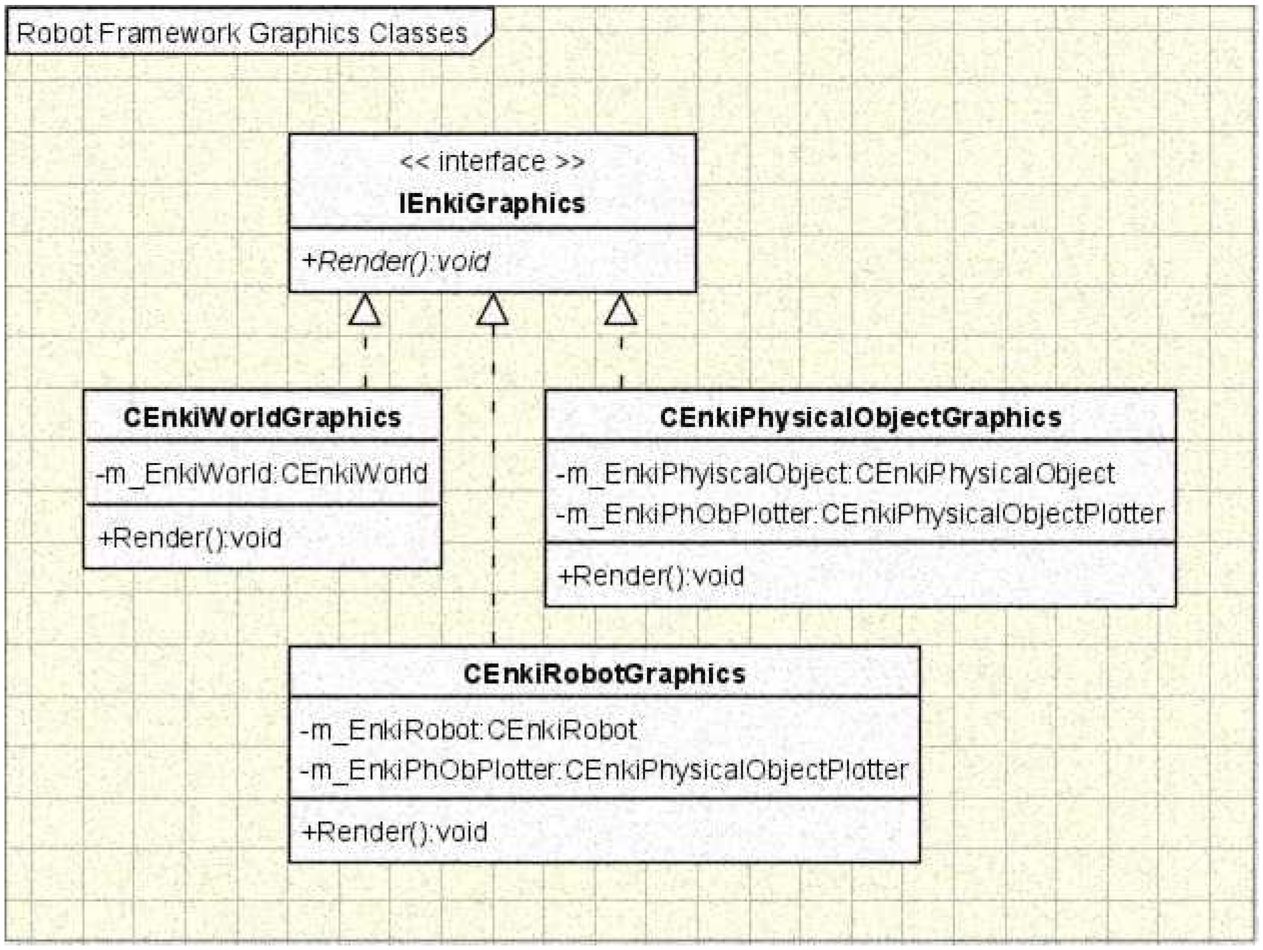
RFW Graphics classes

At first auxiliary plotter classes are introduced:


Robot Framework Graphics Classes
CEnkiPhysicalObjectPlotter
+Render(object:Enki::PhysicalObject):void
CEnkiCircularCamPlotter
+Render(camera:Enki::CircularCam,r:double):void
CEnkiIRSensorPlotter
+Render(irsensor:Enki::IRSensor):void

These classes are very useful because they can render basic Enki primitives: objects, sensors and cameras. The object that they render is passed to them through a parameter in their *Render* method.

Classes that implements *IEnkiGraphics* interface follow:

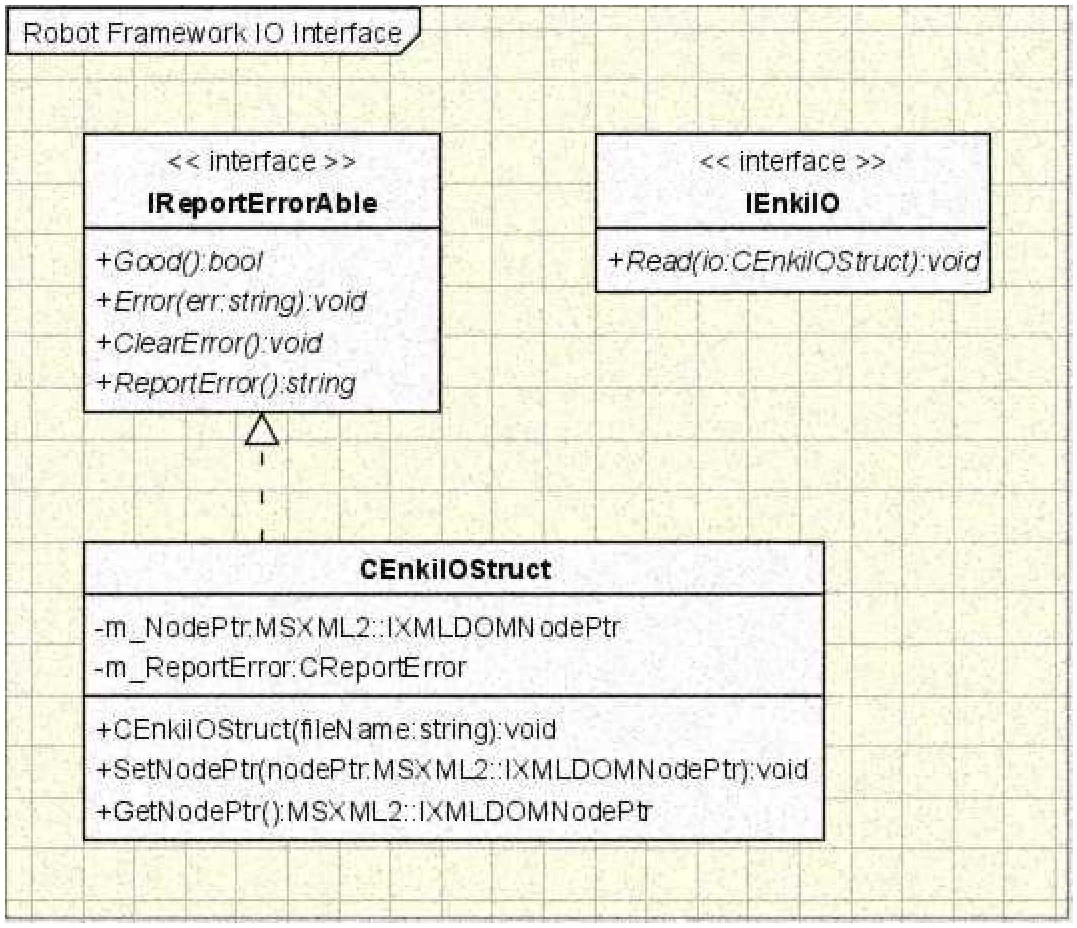


Robot Framework Graphics Classes
<< interface >>
IEnkiGraphics
+Render():void
CEnkiWorldGraphics
-m_EnkiWorld:CEnkiWorld
+Render():void
CEnkiPhysicalObjectGraphics
-m_EnkiPhyiscalObject:CEnkiPhysicalObject
-m_EnkiPhObPlotter:CEnkiPhysicalObjectPlotter
+Render():void
CEnkiRobotGraphics
-m_EnkiRobot:CEnkiRobot
-m_EnkiPhObPlotter:CEnkiPhysicalObjectPlotter
+Render():void

For instance *CEnkiWorldGraphics* class is responsible for rendering *CEnkiWorld* class. This is a very tight unidirectional relationship. *CEnkiWorld* class knows nothing about *CEnkiWorldGraphics* class. It only knows *IEnkiGraphics* interface.

The other classes are analogous: *CEnkiPhysicalObjectGraphics* class is responsible for rendering *CEnkiPhysicalObject* class. *CEnkiRobotGraphics* class is only a demonstration class like *CEnkiRobot* class.

RFW Input/Output


Robot Framework IO Interface
<< interface >>
IReportErrorAble
+Good():bool
+Error(err:string):void
+ClearError():void
+ReportError():string
<< interface >>
IEnkiIO
+Read(io:CEnkiIOStruct):void
CEnkiIOStruct
-m_NodePtr:MSXML2::IXMLDOMNodePtr
-m_ReportError:CReportError
+CEnkiIOStruct(fileName:string):void
+SetNodePtr(nodePtr:MSXML2::IXMLDOMNodePtr):void
+GetNodePtr():MSXML2::IXMLDOMNodePtr

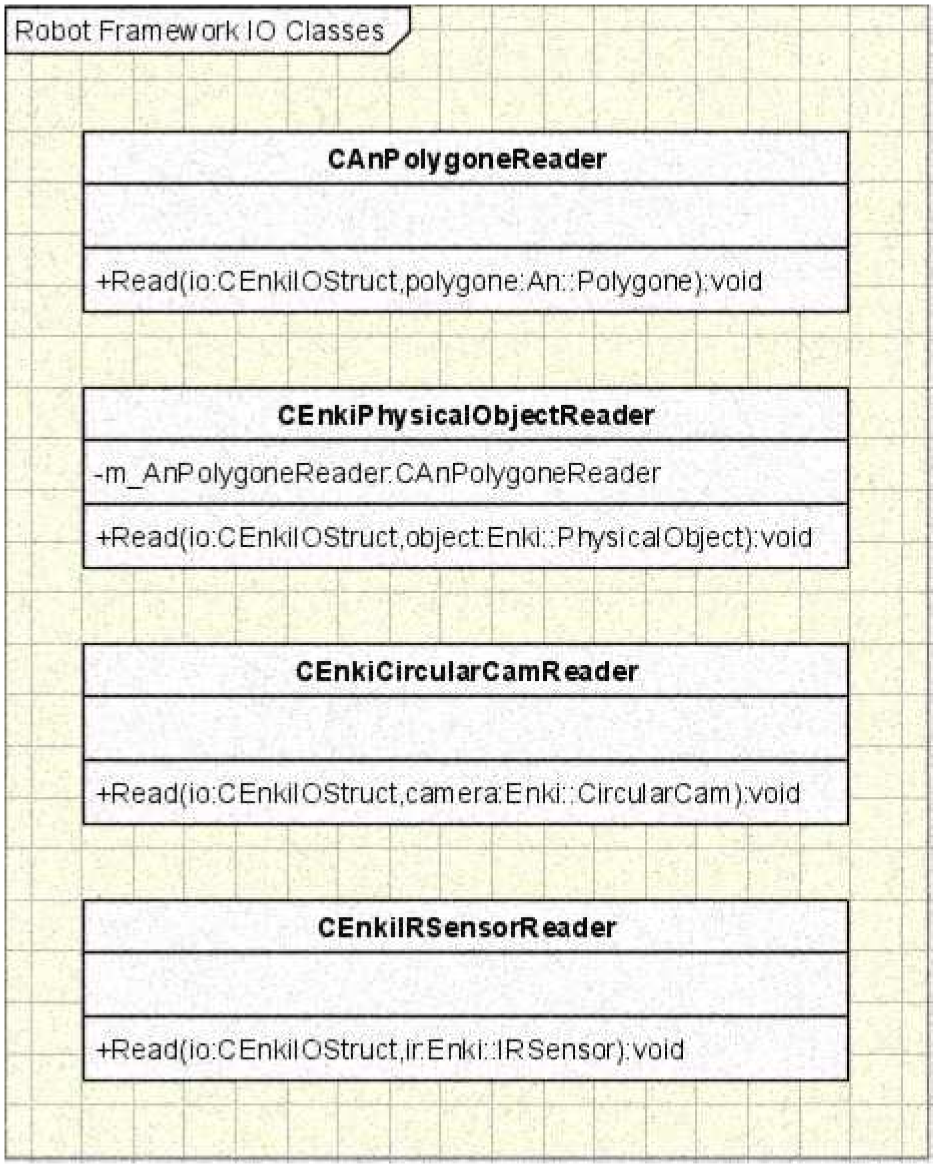
Classes that implement *IReportErrorAble* interface are able to report errors.

CEnkiIOStruct encapsulates a handle to an xml file and implements *IReportErrorAble* interface too. This class is something like a connection between **Robot Framework** and **MSXML**.

IEnkiIO is an interface for all classes that are responsible for loading something from an xml document.

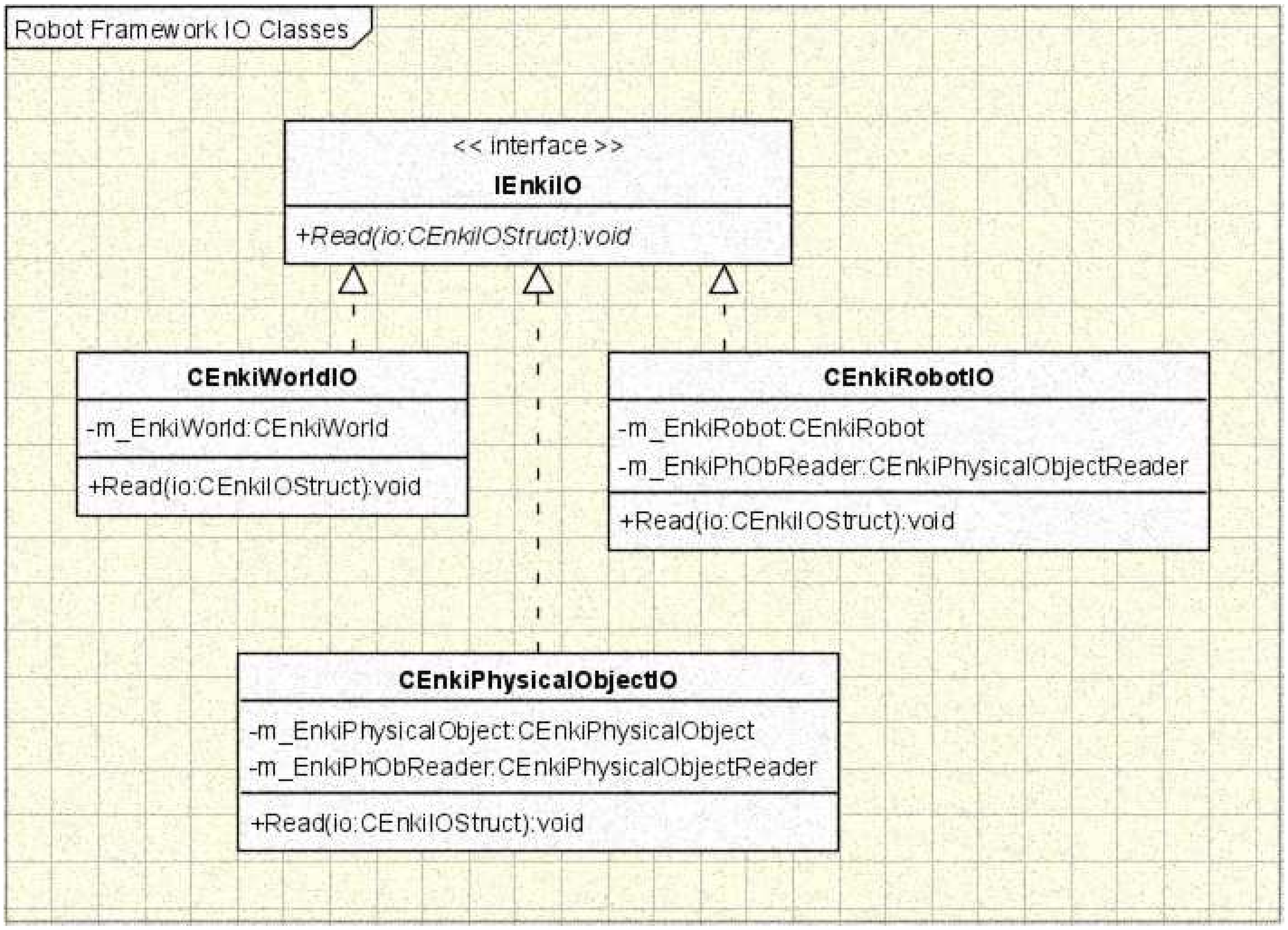
RFW Input/Output classes

At first auxiliary reader classes are introduced:


Robot Framework IO Classes
CAnPolygoneReader
+Read(io:CEnkiIOStruct,polygone:An::Polygone):void
CEnkiPhysicalObjectReader
-m_AnPolygoneReader:CAnPolygoneReader
+Read(io:CEnkiIOStruct,object:Enki::PhysicalObject):void
CEnkiCircularCamReader
+Read(io:CEnkiIOStruct,camera:Enki::CircularCam):void
CEnkiIRSensorReader
+Read(io:CEnkiIOStruct,ir:Enki::IRSensor):void

These classes are able to read basic Enki primitives: objects, sensors and cameras. The object that they read is passed to them through a parameter in their *Read* method.
The *io* parameter serves as a handle to an xml file.

Classes that implements *IEnkiIO* interface follow:

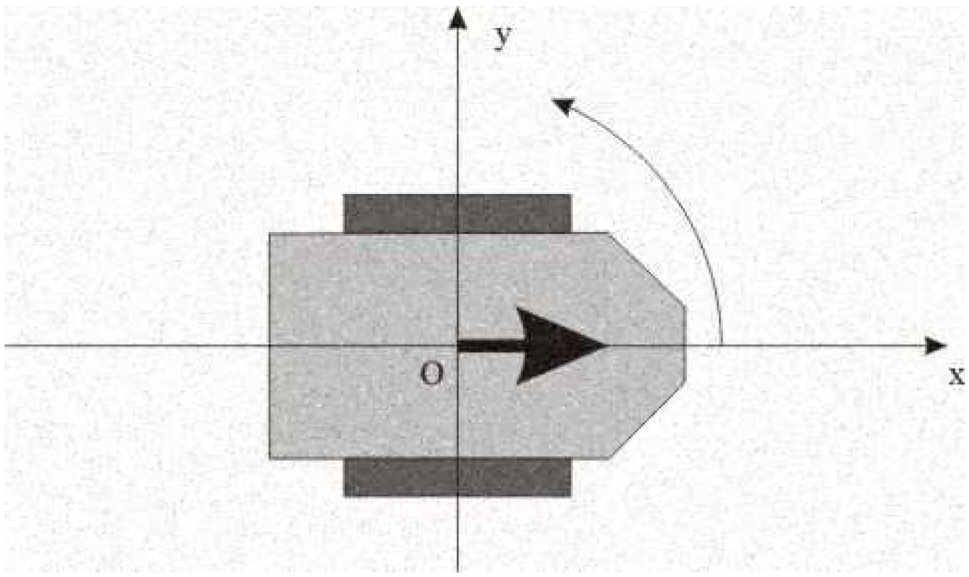

Robot Framework IO Classes
<< interface >>
IEnkiIO
+Read(io:CEnkiIOStruct):void
CEnkiWorldIO
-m_EnkiWorld:CEnkiWorld
+Read(io:CEnkiIOStruct):void
CEnkiRobotIO
-m_EnkiRobot:CEnkiRobot
-m_EnkiPhObReader:CEnkiPhysicalObjectReader
+Read(io:CEnkiIOStruct):void
CEnkiPhysicalObjectIO
-m_EnkiPhysicalObject:CEnkiPhysicalObject
-m_EnkiPhObReader:CEnkiPhysicalObjectReader
+Read(io:CEnkiIOStruct):void

For instance *CEnkiWorldIO* class is responsible for reading *CEnkiWorld* class from an xml file. This is a very tight unidirectional relationship. *CEnkiWorld* class knows nothing about *CEnkiWorldIO* class. It only knows *IEnkiIO* interface.

The other classes are analogous. *CEnkiPhysicalObjectIO* class is responsible for reading *CEnkiPhysicalObject* class from an xml file. *CEnkiRobotIO* class is only a demonstration class like *CEnkiRobot* class.

RFW XML files

At first coordinate system is introduced:



Let us imagine a physical object or a robot. Its boundary is defined as a polygon or a circle. If it is a polygon then the boundary vertices must be sorted in a **counterclockwise order**. The centroid of the boundary is always at the beginning of the coordinate system. The front side of the object (robot) is always in the direction of an x axis.
All the coordinates are in *cm* and the speed is in *cm/s*. The framework is precise in a sense that if you set the speed of a robot for instance to 5 cm/s and robot is moving 10 seconds then it moves by 50 cm.

A polygon is defined easily. For instance:

```
<polygone m = "9">
  -10.0  -10.0
  +10.0  -10.0
  +12.0   -7.0
  +12.5   -1.5
  +12.5   +1.5
  +12.0   +7.0
  +10.0  +10.0
  -10.0  +10.0
  -11.0    0.0
</polygone>
```

Be ware of the counterclockwise order of the vertices.

A physical object is a little more complicated:

```
<object x = "80.0" y = "40.0"
  color = "Maroon"
  height = "10.0"
  angle = "0.0"
  mass = "200.0"
  staticFrictionThreshold = "1.0"
  viscousFrictionTau = "0.1"
  viscousMomentFrictionTau = "0.0"
  collisionAngularFrictionFactor = "0.01">
  <polygone m = "4">
    -10.0  -10.0
    +10.0  -10.0
    +10.0  +10.0
    -10.0  +10.0
  </polygone>
</object>
```

This is a physical object with circular boundary:

```
<object x = "20.0" y = "60.0" r = "10.0"
  color = "Navy"
  height = "10.0"
  angle = "0.0"
  mass = "100.0"
  staticFrictionThreshold = "1.0"
  viscousFrictionTau = "0.1"
  viscousMomentFrictionTau = "0.0"
  collisionAngularFrictionFactor = "0.01">
</object>
```

If you want to understand what exactly all these parameters mean you should look how the Enki physical engine works.

World just encapsulates physical objects. You can also define starting position for a robot.

```
<world width = "100.0" height = "100.0"
  startX = "10.0"
  startY = "10.0"
  startAngle = "45.0">
  <object x = "40.0" y = "25.0"
    color = "Lime"
    height = "10.0"
    angle = "-45.0"
    mass = "100.0"
    staticFrictionThreshold = "1.0"
    viscousFrictionTau = "0.1"
    viscousMomentFrictionTau = "0.0"
    collisionAngularFrictionFactor = "0.01">
    <polygone m = "9">
      -10.0  -10.0
      +10.0  -10.0
      +12.0   -7.0
      +14.0   -1.5
      +14.0   +1.5
      +12.0   +7.0
      +10.0  +10.0
      -10.0  +10.0
      -12.0    0.0
    </polygone>
  </object>
  <object x = "80.0" y = "40.0"
    color = "Red"
    height = "10.0"
    angle = "45.0"
    mass = "100.0"
    staticFrictionThreshold = "1.0"
    viscousFrictionTau = "0.1"
    viscousMomentFrictionTau = "0.0"
    collisionAngularFrictionFactor = "0.01">
    <polygone m = "4">
      -10.0  -10.0
      +10.0  -10.0
      +15.0  +10.0
      -10.0  +10.0
    </polygone>
  </object>
  <object x = "20.0" y = "60.0" r = "5.0"
    color = "Blue"
    height = "10.0"
    angle = "0.0"
    mass = "50.0"
    staticFrictionThreshold = "1.0"
    viscousFrictionTau = "0.1"
    viscousMomentFrictionTau = "0.0"
    collisionAngularFrictionFactor = "0.01">
  </object>
</world>
```

Circular camera and infrared sensor follow:

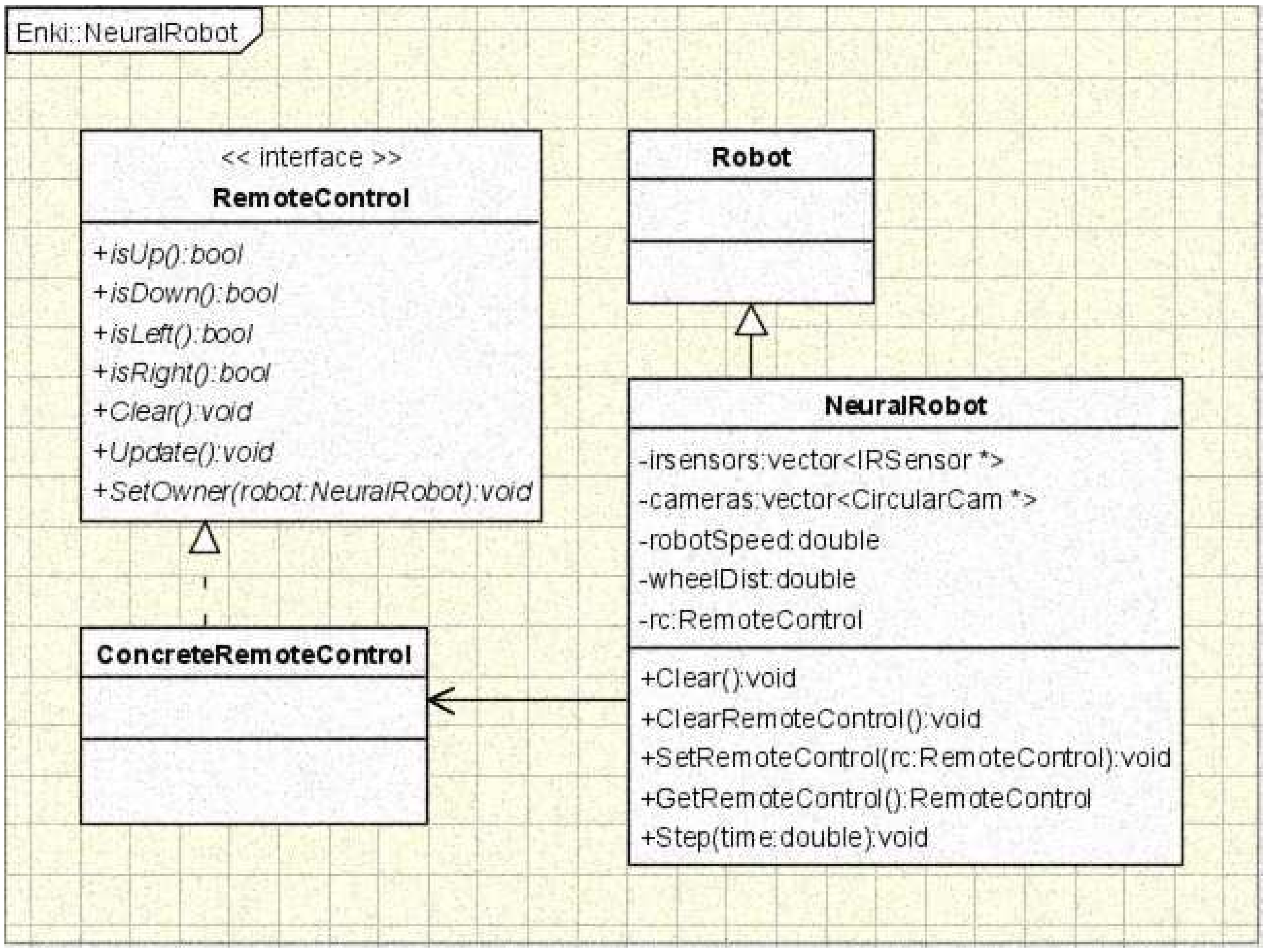
```
<camera x = "2.0" y = "0.0"
  height = "0.5"
  orientation = "0.0"
  fieldOfView = "15.0"
  pixelCount = "50" />

<irsensor x = "1.3" y = "1.3"
  height = "1.8"
  orientation = "45.0"
  range = "20.0"
  aperture = "0"
  rayCount = "1"
  sensorResponseKernel = "simple" />
```

If you want to know where a camera or a sensor is placed on a robot then always imagine the coordinate system showed at the beggining of this page.

Robot Neural Simulator

Robot Neural Simulator (**RNS**) is a concrete application of Robot Framework (**RFW**).

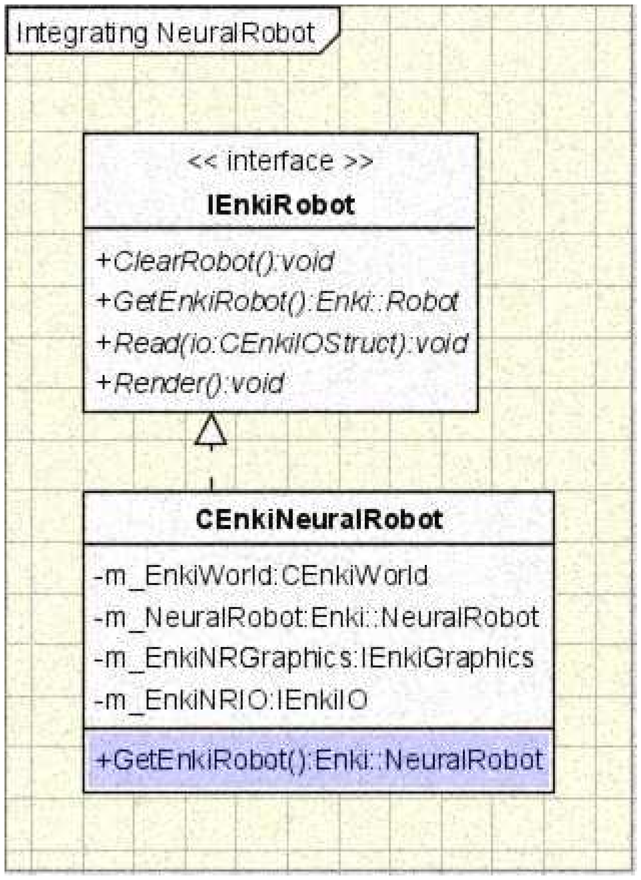
At first *Enki::NeuralRobot* class is introduced:



Enki::NeuralRobot class extends *Enki::Robot* class. This class represents a simple robot with two wheels and some infrared sensors and cameras. The robot is controlled via remote control. Every concrete remote control class implements *Enki::RemoteControl* interface. I have implemented two remote classes: *CKeyboardRemoteControl* class (through this class you can control a robot by keys) and *CNeuralNetworkRemoteControl* class (through this class a robot is controlled by a neural network).

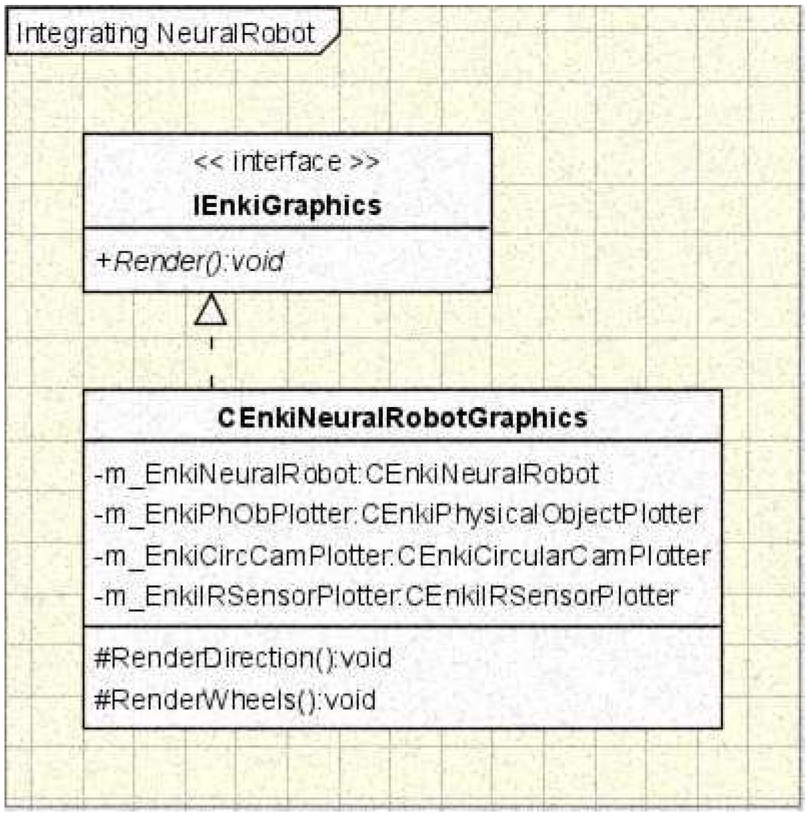
NeuralRobot in RFW

If we want to integrate *Enki::NeuralRobot* into the Robot Framework we have to create several classes: *CEnkiNeuralRobot*, *CEnkiNeuralRobotGraphics* and *CEnkiNeuralRobotIO*.

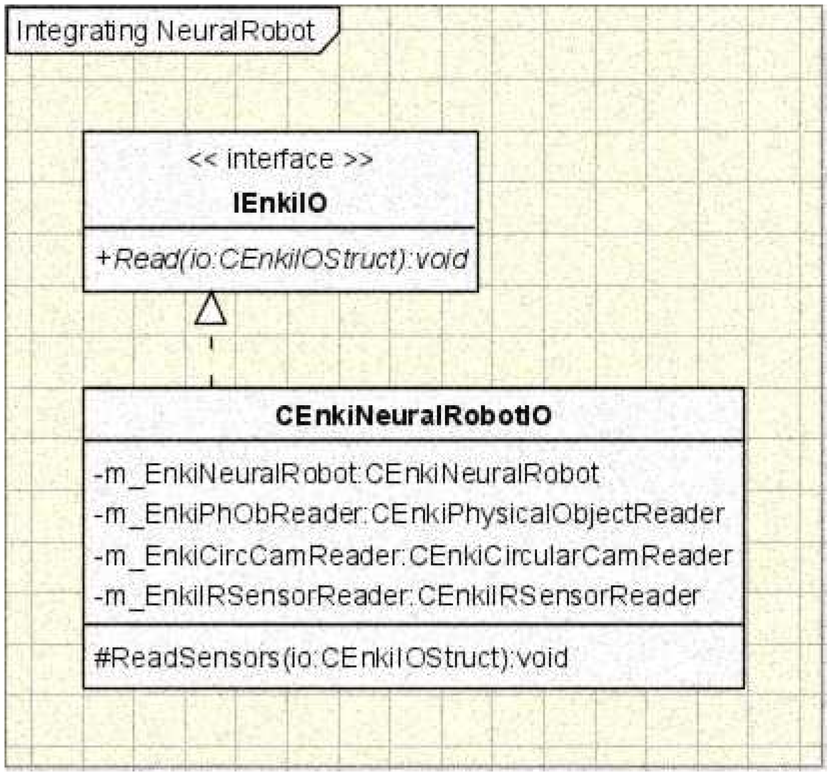

Integrating NeuralRobot
<< interface >>
IEnkiRobot
+ClearRobot():void
+GetEnkiRobot():Enki::Robot
+Read(io:CEnkiIOStruct):void
+Render():void
CEnkiNeuralRobot
-m_EnkiWorld:CEnkiWorld
-m_NeuralRobot:Enki::NeuralRobot
-m_EnkiNRGraphics:IEnkiGraphics
-m_EnkiNRIO:IEnkiIO
+GetEnkiRobot():Enki::NeuralRobot

The most important thing to note is that *GetEnkiRobot* method returns pointer to the *Enki::NeuralRobot* class.

CEnkiNeuralRobotGraphics class is responsible for rendering *CEnkiNeuralRobot* class.


Integrating NeuralRobot
<< interface >>
IEnkiGraphics
+Render():void
CEnkiNeuralRobotGraphics
-m_EnkiNeuralRobot:CEnkiNeuralRobot
-m_EnkiPhObPlotter:CEnkiPhysicalObjectPlotter
-m_EnkiCircCamPlotter:CEnkiCircularCamPlotter
-m_EnkiIRSensorPlotter:CEnkiIRSensorPlotter
#RenderDirection():void
#RenderWheels():void

CEnkiNeuralRobotIO class is responsible for reading *CEnkiNeuralRobot* from an xml file.


Integrating NeuralRobot
<< interface >>
IEnkiIO
+Read(io:CEnkiIOStruct):void
CEnkiNeuralRobotIO
-m_EnkiNeuralRobot:CEnkiNeuralRobot
-m_EnkiPhObReader:CEnkiPhysicalObjectReader
-m_EnkiCircCamReader:CEnkiCircularCamReader
-m_EnkiIRSensorReader:CEnkiIRSensorReader
#ReadSensors(io:CEnkiIOStruct):void

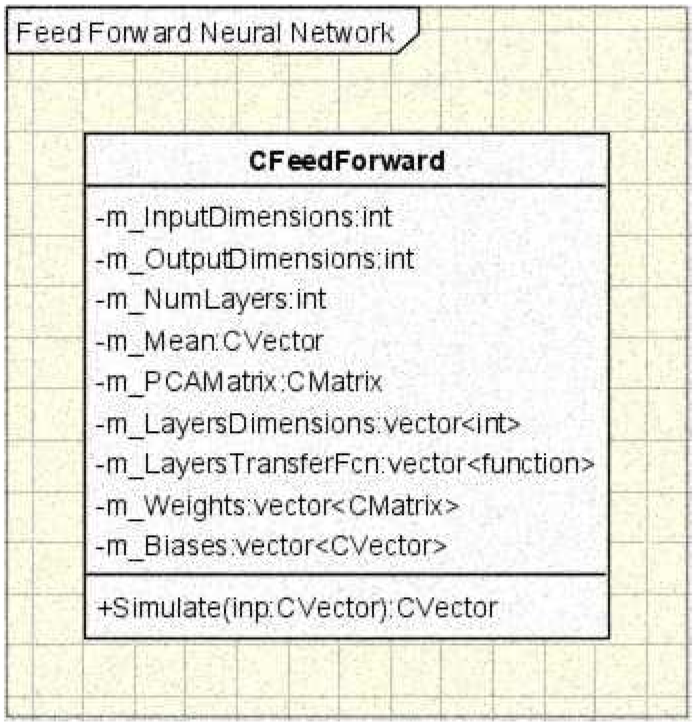
An example of an xml file follows:

```xml
  <neuralrobot wheelDistance = "5.2" robotSpeed = "15.0">
    <object x = "20.0" y = "20.0"
      angle = "0.0"
      height = "3.0"
      mass = "50.0"
      staticFrictionThreshold = "1.0"
      viscousFrictionTau = "0.5"
      viscousMomentFrictionTau = "0.0"
      collisionAngularFrictionFactor = "0.05"
      color = "Gold">
      <polygone m = "6">
        -2.5  -2.5
        +2.0  -2.5
        +3.0  -2.0
        +3.0  +2.0
        +2.0  +2.5
        -2.5  +2.5
      </polygone>
    </object>
    <sensors>
      <irsensor x = "1.0" y = "1.5"
        height = "1.8"
        orientation = "90.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "1.3" y = "1.3"
        height = "1.8"
        orientation = "45.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "1.6" y = "0.6"
        height = "1.8"
        orientation = "0.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "1.6" y = "-0.6"
        height = "1.8"
        orientation = "0.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "1.3" y = "-1.3"
        height = "1.8"
        orientation = "-45.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "1.0" y = "-1.5"
        height = "1.8"
        orientation = "-90.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "-1.5" y = "-1.0"
        height = "1.8"
        orientation = "-180.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <irsensor x = "-1.5" y = "1.0"
        height = "1.8"
        orientation = "-180.0"
        range = "20.0"
        aperture = "0"
        rayCount = "1"
        sensorResponseKernel = "simple" />
      <camera x = "0.0" y = "1.0"
        height = "0.5"
        orientation = "60.0"
        fieldOfView = "45.0"
        pixelCount = "50" />
      <camera x = "0.0" y = "-1.0"
        height = "0.5"
        orientation = "-60.0"
        fieldOfView = "45.0"
        pixelCount = "50" />
      <camera x = "2.0" y = "0.0"
        height = "0.5"
        orientation = "0.0"
        fieldOfView = "15.0"
        pixelCount = "50" />
    </sensors>
  </neuralrobot>
```

As you can see this robot has got eight infrared sensors and three cameras.

Feed Forward

The feed forward neural networks are the simplest type of artificial neural networks devised. The information moves in only one direction, forward, from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network.


Feed Forward Neural Network
CFeedForward
-m_InputDimensions:int
-m_OutputDimensions:int
-m_NumLayers:int
-m_Mean:CVector
-m_PCAMatrix:CMatrix
-m_LayersDimensions:vector<int>
-m_LayersTransferFcn:vector<function>
-m_Weights:vector<CMatrix>
-m_Biases:vector<CVector>
+Simulate(inp:CVector):CVector

The *Simulate* method simply transforms the input vector into the output vector. The pseudo-code of this method follows:

```
  CFeedForward::Simulate(CVector x): CVector
  begin

(1) x := x - m_Mean
(2) x := x / m_Std  // element by element
(3) y := m_PCAMatrix * x

(4) for Layer := 0 to m_NumLayers - 1 do begin
(5)   y := m_Weights[Layer] * y + m_Biases[Layer]
(6)   y := m_LayersTransferFcn[Layer](y)
(7) end

    return y;
  end
```

The rows (1), (2), (3) transform the input vector x into the vector y. *m_PCAMatrix* is the matrix of PCA analysis. This transformation of the input data is optional, but the feed forward networks give better results when trained on correlated data.

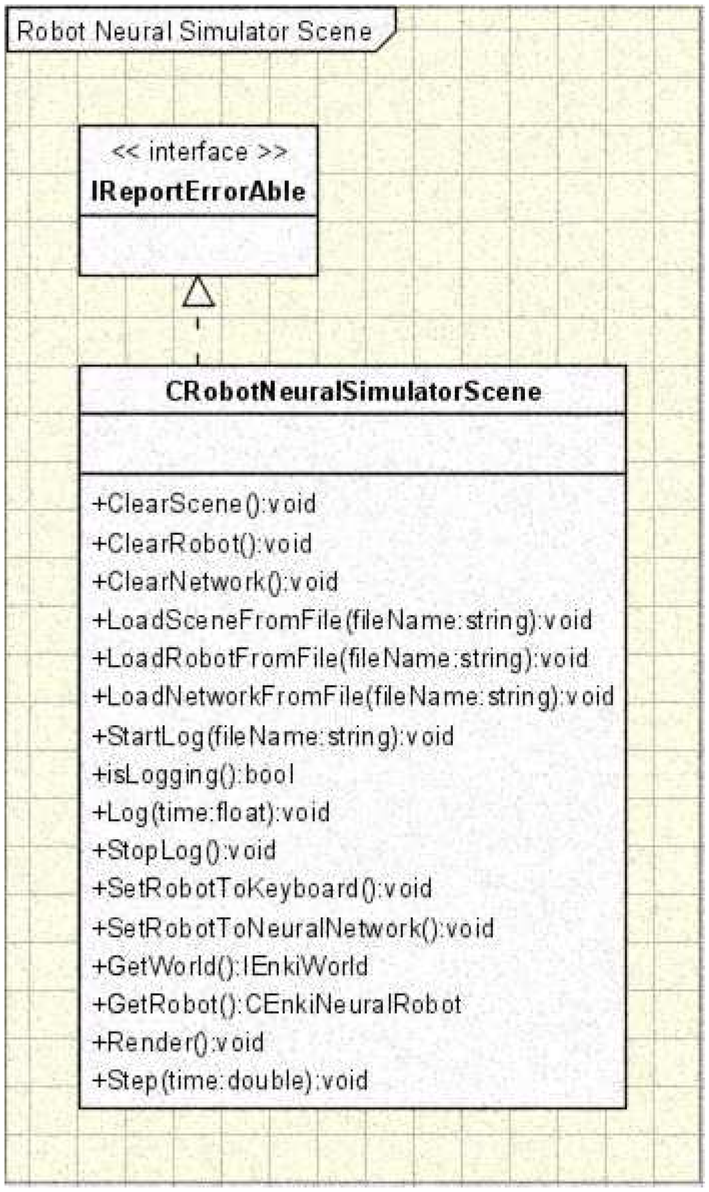
The rows (4), (5), (6), (7) are the main computation of the network. We recognize three types of transfer functions: *logsig*, *tansig* and *purelin*.

An example of an xml file follows:

```
<network inputDimensions = "6" outputDimensions = "4" numLayers = "3">
  <mean>
    <vector m = "6">
       0.5469
       0.8383
      14.9469
      24.2931
      27.4923
      39.5549
    </vector>
  </mean>
  <std>
    <vector m = "6">
       1.4228
       2.2431
      13.2114
      14.9573
      18.6651
      21.5919
    </vector>
  </std>
  <PCAmatrix>
    <matrix m = "6" n = "6">
      -0.6129  -0.6111   0.2967   0.2694   0.1270   0.2720
       0.0337   0.1230  -0.4286   0.5603  -0.4914   0.4944
      -0.0786  -0.0334  -0.6607   0.1831   0.7211  -0.0497
       0.3468   0.0235   0.1716  -0.3121   0.3307   0.8017
      -0.1551  -0.4163  -0.5119  -0.6501  -0.3351   0.0740
       0.6874  -0.6605   0.0078   0.2443  -0.0225  -0.1751
    </matrix>
  </PCAmatrix>
  <layers>
    <layer index = "1" dimensions = "4" transferFcn = "logsig" />
    <layer index = "2" dimensions = "4" transferFcn = "logsig" />
    <layer index = "3" dimensions = "4" transferFcn = "logsig" />
  </layers>
  <weights>
    <inputlayer to = "1">
      <matrix m = "4" n = "6">
        -2.5202  -0.2487   0.5199   4.9593   -0.2628  -1.5376337509
        -1.8132  -3.7116  -3.3079  20.3096  -20.5454  20.6921671054
         3.5972  -2.0743  -2.6365   2.8116   -2.2578   6.4847634567
         0.4913   0.4940   0.0806  -0.5194   -0.3200  -0.8301912843
      </matrix>
    </inputlayer>
    <hiddenlayer to = "2" from = "1">
      <matrix m = "4" n = "4">
        23.6662   39.9447   41.9081  -83.0140367806
        55.5267  -27.9243   45.4143   35.5339920444
         4.3360   27.2725  -23.6810   45.0458177605
       -48.7727   33.5489    3.8649  -56.0159550367
      </matrix>
    </hiddenlayer>
    <hiddenlayer to = "3" from = "2">
      <matrix m = "4" n = "4">
         8.1293    2.2256   10.8377    1.1275475536
        -3.0373   -6.8360    0.2909    3.9822623431
       -44.1881  -66.0722    3.9148  -99.4670877679
       -30.0430   41.3335  -37.7869   86.7369517566
      </matrix>
    </hiddenlayer>
  </weights>
  <biases>
    <bias to = "1">
      <vector m = "4">
         0.5537
        -2.3086
         6.6602
        -0.5831
      </vector>
    </bias>
    <bias to = "2">
      <vector m = "4">
         17.6842
        -64.1610
        -13.8322
         14.2032
      </vector>
    </bias>
    <bias to = "3">
      <vector m = "4">
          4.5828
        -10.2205
         68.0925
        -44.2838
      </vector>
    </bias>
  </biases>
</network>
```

This xml code was not written by a human. The network was generated in MATLAB. I have written a special MATLAB function that can convert such network into an xml file.

RNS Scene

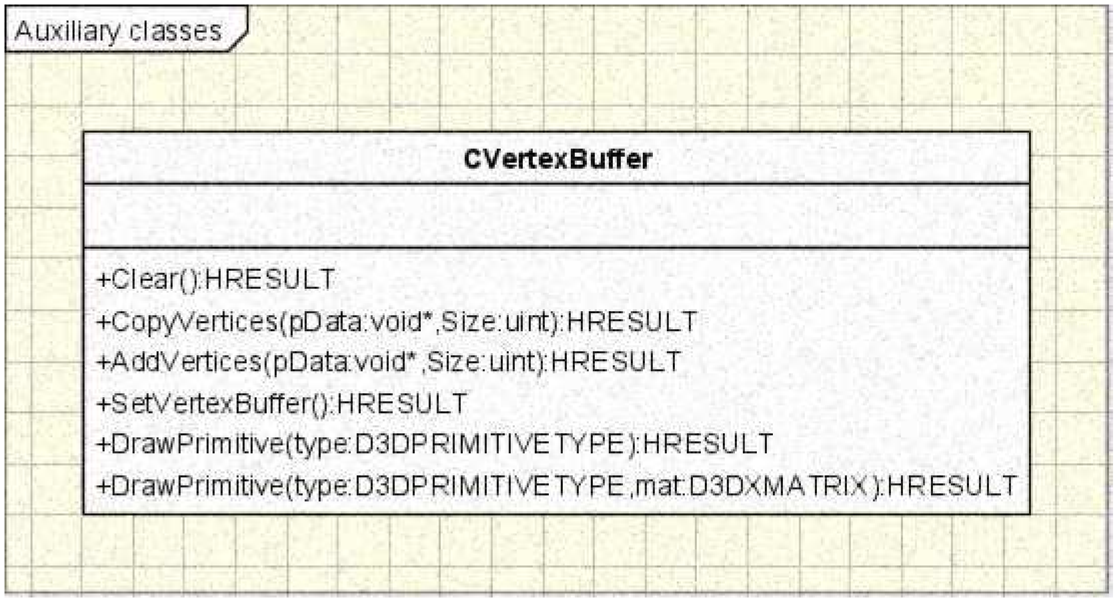

Robot Neural Simulator Scene
<< interface >>
IReportErrorAble
CRobotNeuralSimulatorScene
+ClearScene():void
+ClearRobot():void
+ClearNetwork():void
+LoadSceneFromFile(fileName:string):void
+LoadRobotFromFile(fileName:string):void
+LoadNetworkFromFile(fileName:string):void
+StartLog(fileName:string):void
+isLogging():bool
+Log(time:float):void
+StopLog():void
+SetRobotToKeyboard():void
+SetRobotToNeuralNetwork():void
+GetWorld():IEnkiWorld
+GetRobot():CEnkiNeuralRobot
+Render():void
+Step(time:double):void

CRobotNeuralSimulatorClass does not belong to RFW. It is a specific class for Robot Neural Simulator. The interface of this class defines what a user can do in RNS. Calling these methods in any order at any time should not cause a fall of a program or a run-time error.

Auxiliary classes

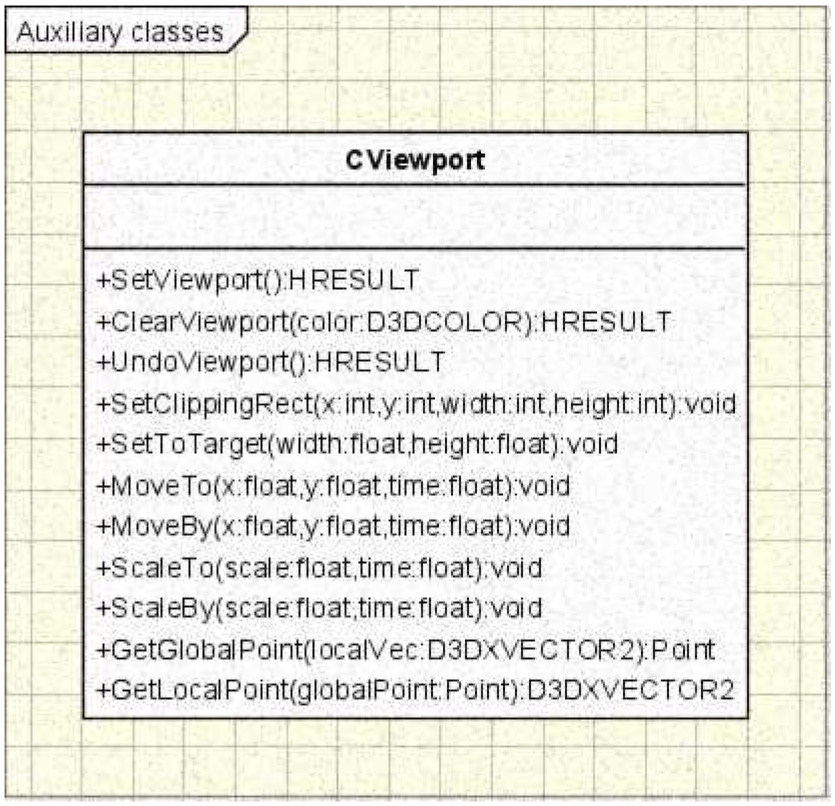
There are many useful classes used in Robot Neural Simulator. I will mention only two classes: *CVertexBuffer* class and *CViewport* class.

CVertexBuffer class encapsulates Direct3D Vertex Buffer.


Auxiliary classes
CVertexBuffer
+Clear():HRESULT
+CopyVertices(pData:void*,Size:uint):HRESULT
+AddVertices(pData:void*,Size:uint):HRESULT
+SetVertexBuffer():HRESULT
+DrawPrimitive(type:D3DPRIMITIVETYPE):HRESULT
+DrawPrimitive(type:D3DPRIMITIVETYPE,mat:D3DXMATRIX):HRESULT

You can simply copy some vertices there and then you can draw it. In addition you can specify a transformation matrix when drawing. The main advantage of this class is that we have only one vertex buffer through the whole application. Or if you want to rewrite the simulator for instance for the OpenGL you just need to rewrite this class.

CViewport class is an interface between logical coordinates of the simulator and physical coordinates on the screen.


Auxiliary classes
CViewport
+SetViewport():HRESULT
+ClearViewport(color:D3DCOLOR):HRESULT
+UndoViewport():HRESULT
+SetClippingRect(x:int,y:int,width:int,height:int):void
+SetToTarget(width:float,height:float):void
+MoveTo(x:float,y:float,time:float):void
+MoveBy(x:float,y:float,time:float):void
+ScaleTo(scale:float,time:float):void
+ScaleBy(scale:float,time:float):void
+GetGlobalPoint(localVec:D3DXVECTOR2):Point
+GetLocalPoint(globalPoint:Point):D3DXVECTOR2

Thanks to this class you can for instance transform a position of the mouse to the logical coordinates of the simulator and vice versa. You can imagine this class as a rectangle on the screen of the application where all the objects of the simulator world are rendered.
There are also some useful methods. For instance by *MoveTo* method you can move the viewport fluently to some specific point.

All these auxiliary classes work independently of robot framework.