UNIVERSITÀ DEGLI STUDI DI BRESCIA

FACOLTÀ DI INGEGNERIA

CORSO DI LAUREA SPECIALISTICA IN INGEGNERIA INFORMATICA

CORSO DI ROBOTICA MOBILE

Guida introduttiva all'utilizzo del simulatore Simbad

Studente:

Daniele Petre Matricola : **77686**

Professore:

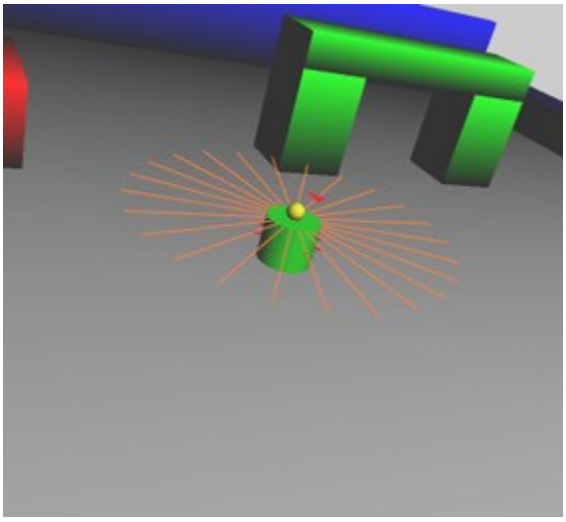
Prof. Riccardo Cassinis

ANNO ACCADEMICO 2011/2012

Indice

Introduzione

In questo documento viene presentato il simulatore di robot 3D Simbad[1].



Si tratta di uno strumento dal semplice utilizzo, scritto in Java, per la simulazione del comportamento di agenti dotati di diversi tipi di sensori e attuatori all'interno di ambienti 3D programmabili a piacere.

Le principali funzionalità con cui viene presentato questo strumento sono:

- Visualizzazione della simulazione in 3D;
- Simulazione singolo o multi robot;
- Sensori visione: telecamera a colori, sensore luce;
- Sensori di distanza: sonar, IR, laser;
- Sensore di contatto;
- Interfaccia grafica di controllo sviluppata con la libreria Java Swing.

Secondo gli autori il simulatore e le sue capacità sono volutamente ridotte per garantire la massima facilità di apprendimento da parte di nuovi utenti. Il software è open source, rilasciato con licenza GPL, anche se l'ultimo rilascio risale all'anno 2007.

Installazione

Per la stesura di questo documento è stata utilizzata l'ultima versione di Simbad disponibile, la 1.4[2].

Per il corretto funzionamento del simulatore è necessario avere installato una macchina virtuale Java[3] (JRE) e le librerie Java3D[4].

E' necessario impostare alcune variabili di sistema[5] indicando la posizione nel file system dei seguenti file e cartelle (ipotizzando che il path di installazione di Java3D sia [path-java3d]):

- *CLASSPATH* [path-java3d]/j3dcore.jar, [path-java3d]/j3dutils.jar e [path-java3d]/vecmath.jar;
- *LD_LIBRARY_PATH* [path-java3d]/j3d/bin.

Simbad non necessita di installazione, si può quindi estrarre dallo zip in qualsiasi cartella (questo non è necessario se si è scaricato direttamente la versione compilata con estensione *.jar*).

Non sono necessari altri accorgimenti, quindi, se tutto è impostato correttamente, un doppio click sul file *simbad-1.4.jar* avvierà il simulatore con un ambiente demo, mostrato in Figura 1.

[1] http://simbad.sourceforge.net/
[2] http://sourceforge.net/projects/simbad/files/simbad/1.4/
[3] http://www.java.com/en/download/index.jsp
[4] http://www.oracle.com/technetwork/java/javase/tech/index-jsp-138252.html
[5] http://www.java3d.org/suninstructions.html

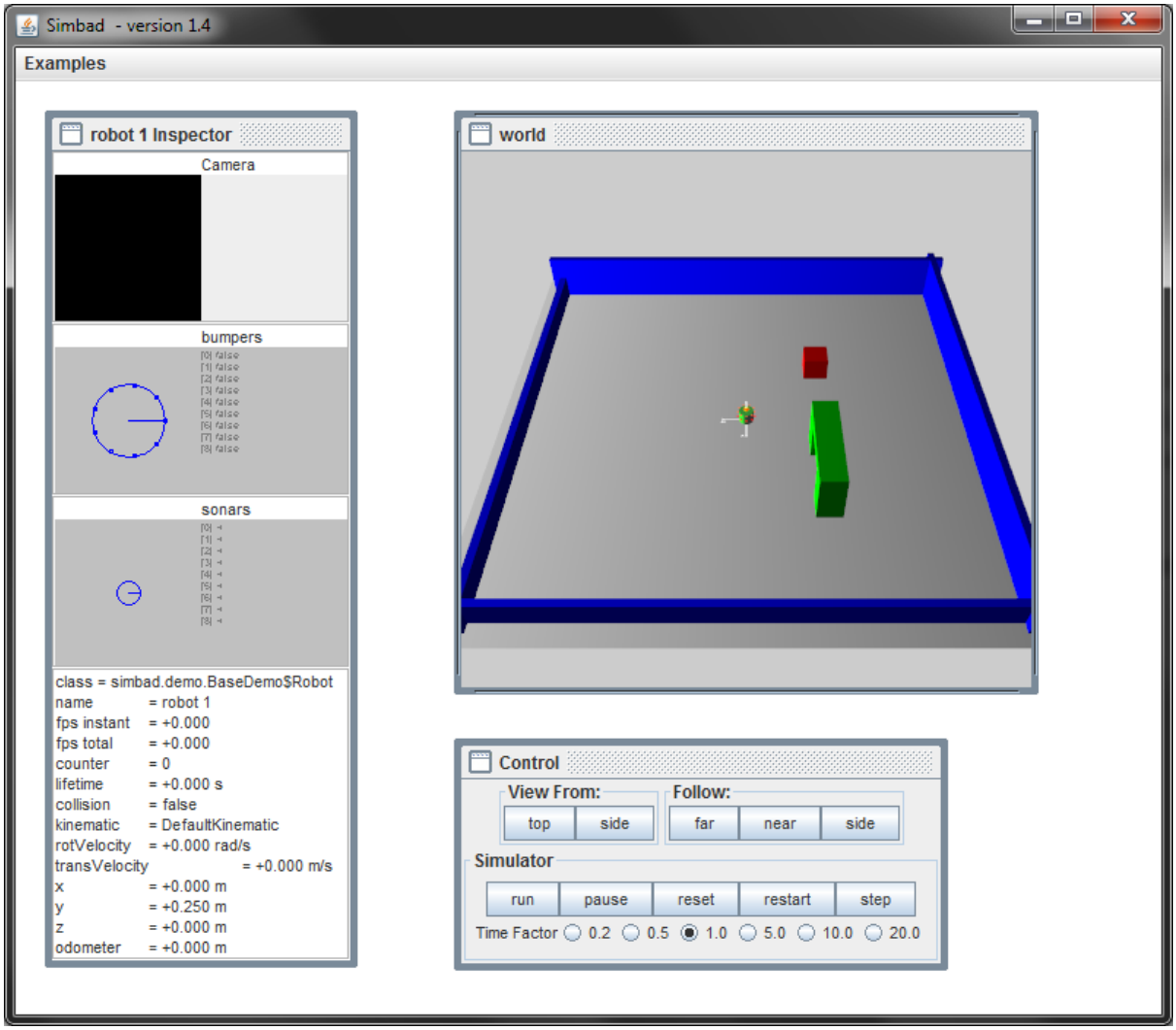


Figura 1 - Primo avvio di Simbad

In caso contrario potrebbe succedere che non venga trovata la libreria java3Dd e venga mostrato il seguente errore:

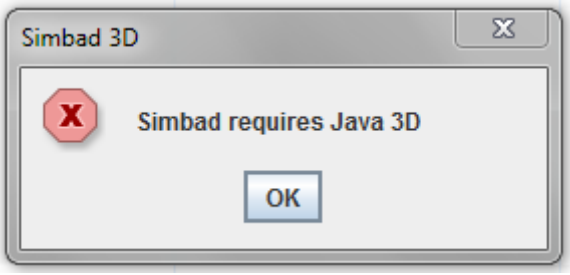


Figura 2 - Errore nell'installazione e impostazione della libreria Java3D

In questo caso è necessario ricontrollare l'assegnamento delle variabili di sistema sopra citate e verificare la corretta installazione di Java3D. E' possibile verificarne il corretto funzionamento, indipendentemente da simbad, all'indirizzo web http://www.java3d.org/howto.html. Se la libreria è installata correttamente verrà visualizzata un applet Java contenente la rappresentazione di una sfera rossa su sfondo nero.

Una nota particolare è da fare per sistemi a 64bit. Dai test effettuati sembra che simbad, nella versione scaricata, sia in grado di funzionare solo con Java3D a 32bit. E' importante quindi, nell'utilizzo del simulatore, impostare sempre l'utilizzo delle librerie a 32bit (ad esempio nella build path di un progetto eclipse).

L'interfaccia grafica del simulatore

Come già visto, quando Simbad viene avviato tramite il file *jar* viene precaricato uno degli agenti di esempio presenti al suo interno. E' possibile caricare diversi altri agenti di esempio dal menu in alto a sinistra *Examples*, visibile in Figura 3.

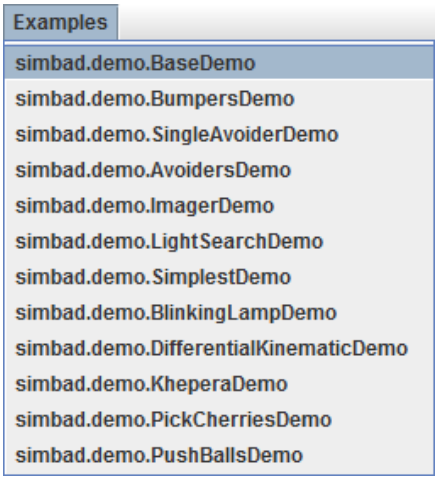


Figura 3 - Menu Examples contenente gli agenti di esempio forniti con il simulatore

In questi esempi vengono messe in mostra le principali funzionalità fornite dalla libreria per la creazione degli agenti da simulare. I codici sorgenti di questi agenti sono forniti nella cartella *src /simbad/demo*. Successivamente al caricamento di uno degli agenti o, come vedremo, in seguito al caricamento di Simbad con un nuovo agente sviluppato, è possibile passare all'effetiva simulazione.

L'interfaccia utente del sistema è abbastanza intuitiva ed è composta da tre principali elementi.

Il primo e più vistoso è il pannello contenente la rappresentazione grafica 3D dell'ambiente simulato, mostrato in Figura 4. Questo si limita quindi a visualizzare il mondo con i suoi eventuali muri, ostacoli e agenti durante il funzionamento. Lo strumento permette solo la manipolazione della visuale tramite l'utilizzo del tasto sinistro del mouse.

In Figura 5 viene invece mostrato il pannello per il controllo del simulatore. I principali controlli riguardano le funzioni per iniziare, fermare o riavviare l'esecuzione della simulazione. E' possibile scegliere tra diverse opzioni per quanto riguarda la velocità di esecuzione modificando il valore del radiogroup *Time factor* rallentando o velocizzando di conseguenza la simulazione in base alle necessità del momento. E' inoltre possibile procedere step-by-step nella simulazione tramite il comando *step*, il sistema attenderà quindi l'input dell'utente prima di eseguire ogni ciclo. Infine sono presenti dei controlli per cambiare il punto di vista della visualizzazione 3D.

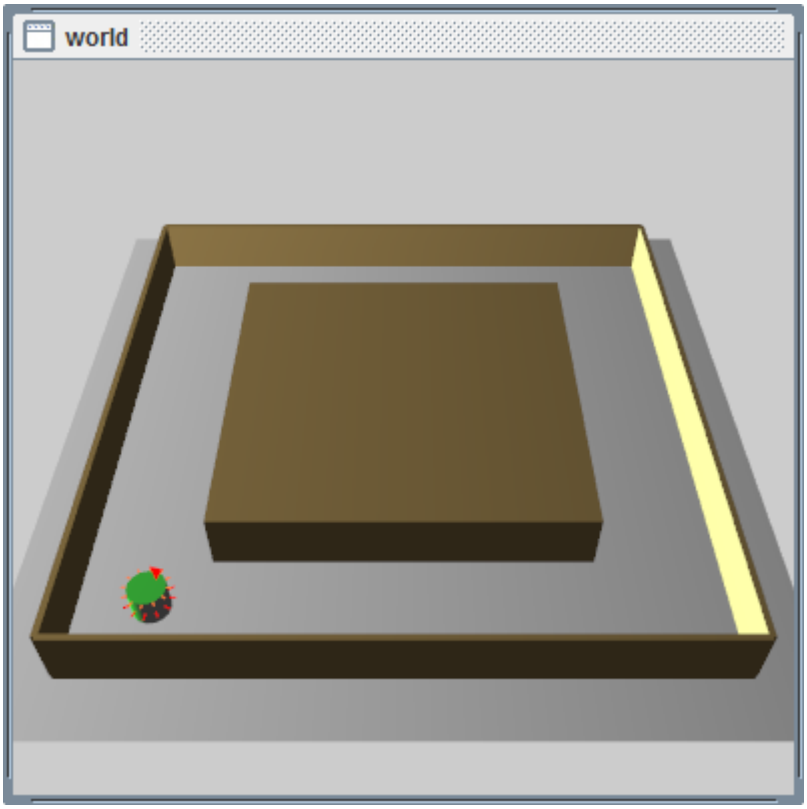


Figura 4 - Pannello per la visualizzazione 3D dell'ambiente simulato

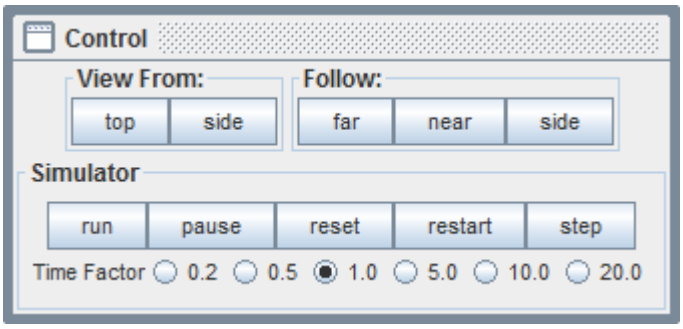


Figura 5 - Pannello di controllo della simulazione

Infine è presente un pannello, visibile in Figura 6, che fornisce molte informazioni utili durante l'esecuzione delle simulazioni. Il pannello viene automaticamente modificato in base alle caratteristiche dell'agente simulato. Vengono infatti forniti sia i valori dei sensori presenti sia una loro intuitiva rappresentazione grafica. Nell'esempio in figura sono presenti 12 sonar e 12 bumper. In seguito vengono riportate altre informazioni più generali riguardanti la simulazione. In particolare vengono riportati dati riguardanti l'agente (classe, velocità di traslazione e rotazione, posizione, ecc) e dati riguardanti gli step temporali della simulazione (tempo trascorso, numero di step simulati).

Bisogna infine aggiungere, riguardo all'interfaccia grafica, che è possibile sia personalizzare le dimensioni di tutti i pannelli sia aggiungerne di nuovi e personalizzati estendendo la classe *JPanel* e utilizzando il metodo *Agent.setUIPanel(JPanel)*.

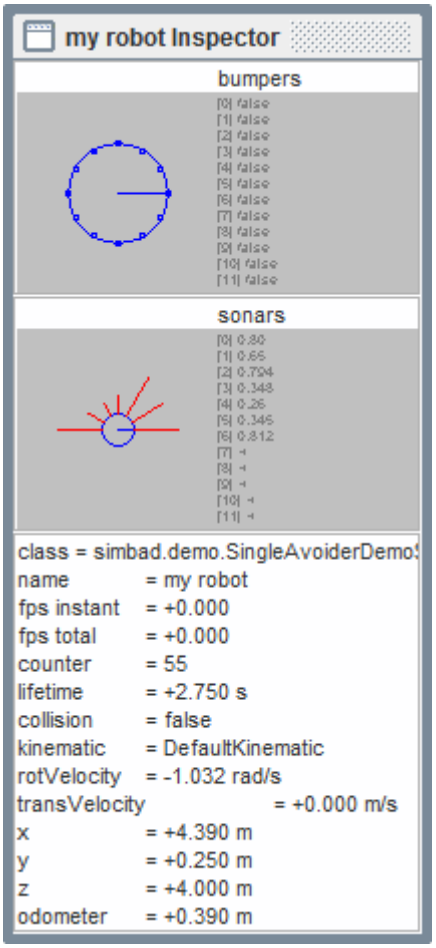


Figura 6 - Pannello per la visualizzazione dei dati della simulazione, sensori e attuatori

Programmazione di simulazioni

L'interfaccia grafica permette di eseguire le simulazioni degli agenti programmati. Nel seguito viene introdotta la meccanica utilizzata per la simulazione e gli strumenti messi a disposizione per la definizione del comportamento degli agenti da simulare.

La parte principale della simulazione è rappresentata dal ciclo di simulazione. I suoi passi vengono eseguiti di continuo per far avanzare lo stato dell'agente all'istante successivo.

I passi del ciclo sono i seguenti e vengono eseguiti per ogni agente "vivo" durante la simulazione:

- Controllo delle collisioni geometriche tra gli oggetti del mondo e l'agente;
- Aggiornamento dei valori dei sensori, se presenti, in base alla nuova posizione dell'agente e alla loro frequenza di aggiornamento;
- Aggiornamento degli attuatori, se presenti, in base alla loro frequenza di aggiornamento;
- Chiamata al metodo *performBehavior* dell'agente;
- Aggiornamento della posizione dell'agente in base ai parametri forniti.

E' da notare che, così come nel caso di strumenti reali, sensori e attuatori non vengono aggiornati a ogni ciclo ma in base alla loro frequenza specifica.

I passi del ciclo quindi sono predefiniti e l'utente, programmando gli agenti, interviene direttamente solo nel penultimo passo, quando viene chiamato il metodo *performBehavior* di ogni agente. Questo metodo dunque è destinato alla descrizione del comportamento desiderato per l'agente durante la simulazione.

Di seguito vengono presentati alcuni dei principali metodi, ereditati dalla classe *Agent*, per ottenere informazioni sull'agente stesso e impostarne le proprietà.

I metodi riguardanti lo stato dell'agente riguardano principalmente la sua posizione e la presenza di collisioni:

- *boolean collisionDetected()*, restituisce true se sono state rilevate collisioni nel ciclo corrente;
- *int getCounter()*, ritorna il numero di cicli eseguiti durante la simulazione;
- *double getOdometer()*, ritorna il valore dell'odometro dell'agente, in metri;
- *double getLifeTime()*, restituisce il numeri di secondi virtuali di vita dell'agente;
- *void getCoords(Point3d coord)*, restituisce le coordinate in metri della posizione attuale dell'agente.

Altri metodi riguardano principalmente le velocità di movimento e di rotazione dell'agente durante i suoi spostamenti:

- *void setRotationalVelocity(double rv)*, imposta la velocità di rotazione in radianti al secondo;
- *void setTranslationalVelocity(double tv)*, imposta la velocità di traslazione in metri al secondo;
- *double getRotationalVelocity()*, restituisce l'attuale velocità di rotazione in radianti al secondo;
- *double getTranslationalVelocity()*, restituisce l'attuale velocità di traslazione in metri al secondo;
- *void moveToStartPosition()*, riporta l'agente nella posizione di partenza.

Questi metodi valgono per la tipologia di agente di base. E' tuttavia possibile costruire agenti con controlli indipendenti per le velocità delle ruote destre e sinistre (Differential Drive), utilizzando la classe DifferentialKinetic e i suoi metodi:

- *setLeftVelocity(double vel)*, imposta la velocità delle ruote di sinistra;
- *setRightVelocity(double vel)*, imposta la velocità delle ruote di sinistra;
- *setWheelsVelocity(double vl, double vr)*, imposta la velocità di tutte le ruote.

Per meglio comprendere la struttura delle API del simulatore viene riportato il diagramma delle classi principali, in Figura 7.

Alla base si trova la classe astratta *BaseObject* per la rappresentazione di un generico oggetto nel mondo della simulazione.

Successivamente le principali classi sono *Agent*, *SensorDevice*, *ActuatorDevice* e *BlockWorldObject*.

La prima rappresenta un agente, con i metodi già visti in precedenza.

In seguito, ereditando da *Device*, si trovano i sensori e gli attuatori assegnabili a un agente, quali sonar, telecamera, lampada, ecc.

Infine la classe *BlockWorldObject* rappresenta un oggetto statico del mondo simulato, può quindi essere un muro o un generico ostacolo. Di default vengono fornite le sottoclassi *Box* e *Arch* per l'inserimento di oscatoli rettangolari e a forma di arco.

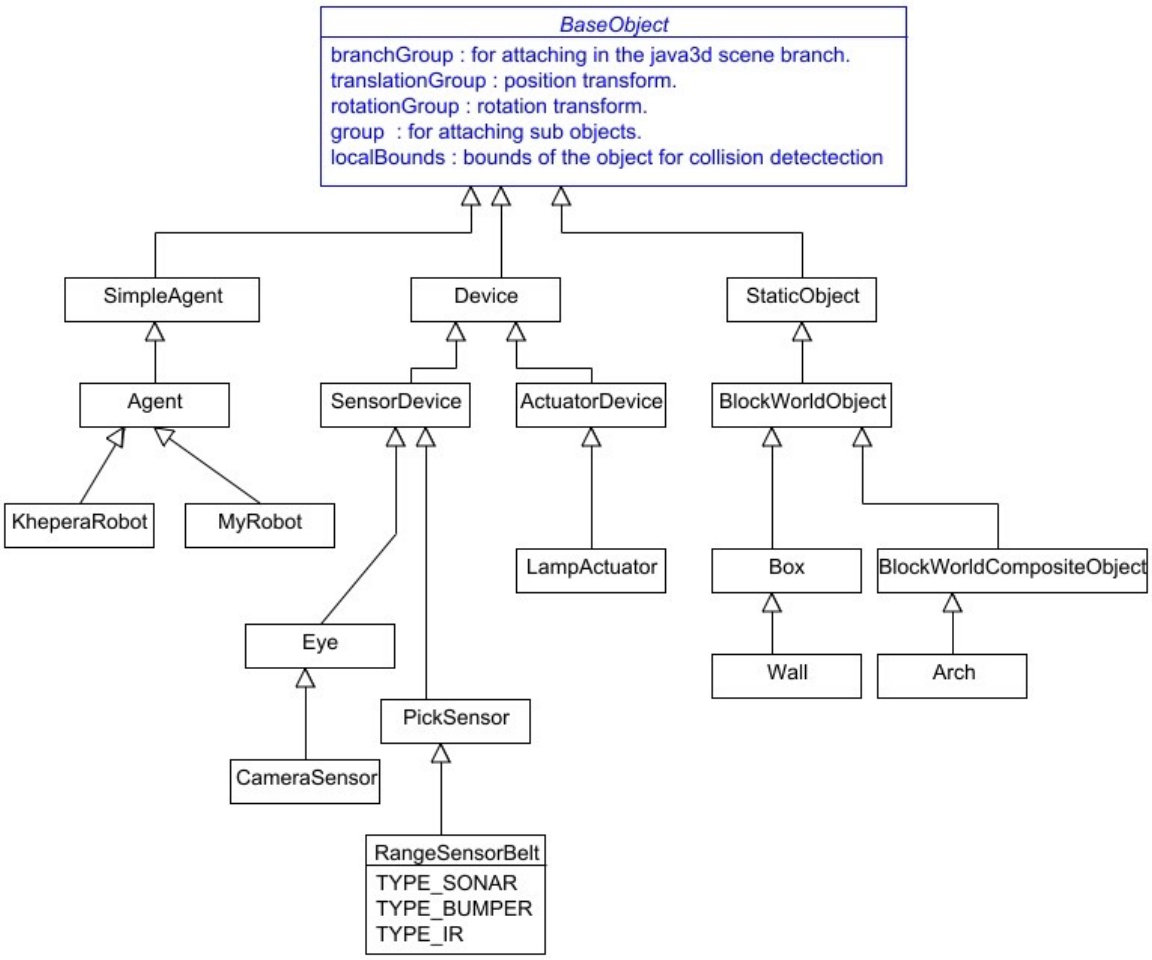


Figura 7 - Diagramma delle principali classi di Simbad

Agente di esempio

Per completare la descrizione del simulatore Simbad viene proposto un esempio in cui è stato implementato un agente dotato di otto sensori di distanza, una telecamera e una lampada di segnalazione all'interno di un ambiente quadrato con diversi ostacoli, come visibile in Figura 8.

Per comodità il codice di questo esempio, a differenza degli esempi proposti da Simbad, è stato suddiviso in diversi file come da standard Java. In particolare le classi implementate sono tre:

- *AmbienteTutorial*, classe contenente la descrizione del mondo della simulazione;

- *Tutorial*, classe principale per l'esecuzione del simulatore con l'ambiente e l'agente implementati;
- *TutorialRobot*, implementazione dell'agente della simulazione.

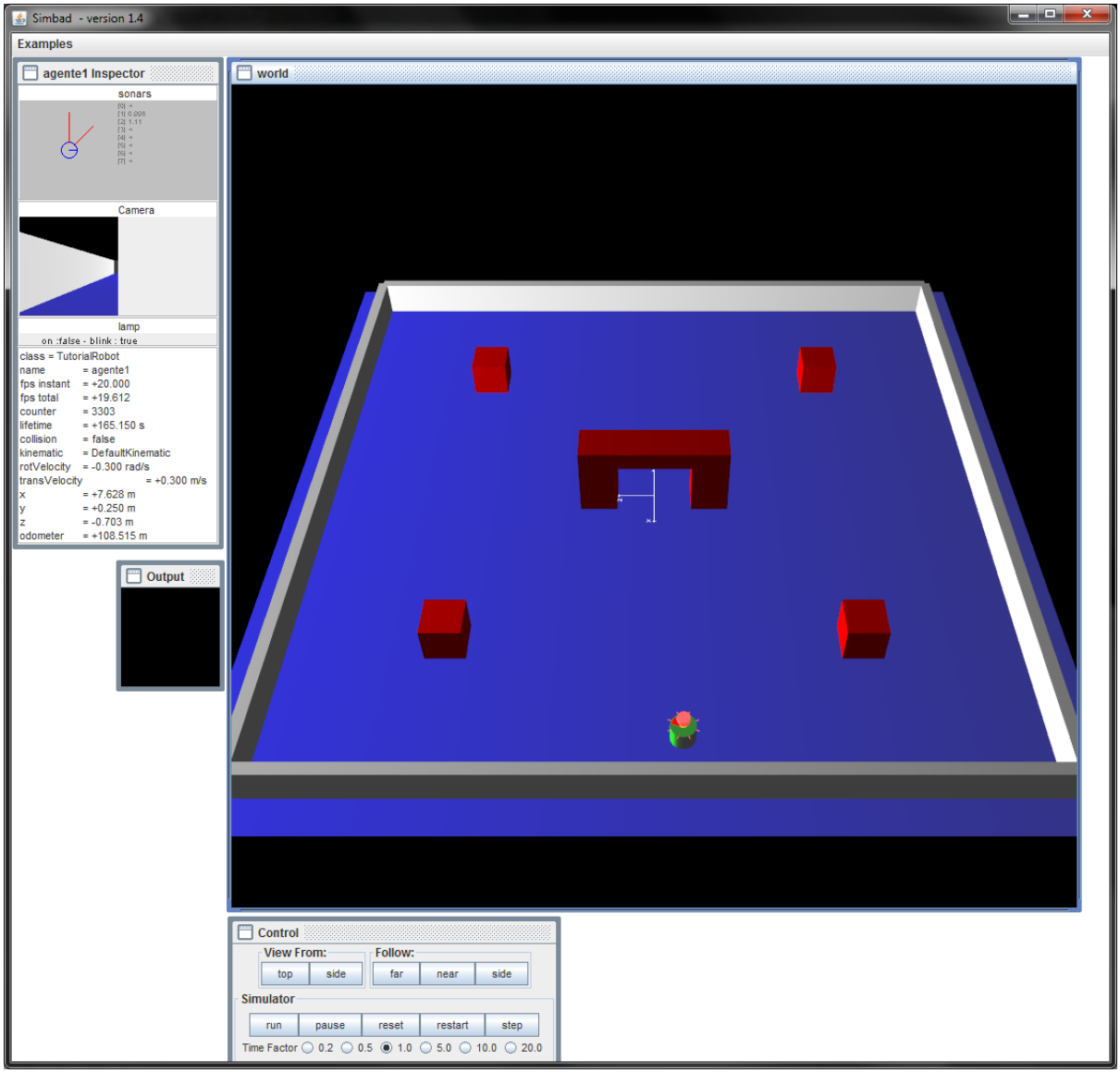


Figura 8 - Agente di esempio durante l'esecuzione

Viene analizzata per prima la classe *AmbienteTutorial*. Di seguito il codice sorgente completo:

```
1.  import javax.vecmath.Vector3d;
2.  import javax.vecmath.Vector3f;
3.  import simbad.sim.Arch;
4.  import simbad.sim.Box;
5.  import simbad.sim.EnvironmentDescription;
6.  import simbad.sim.Wall;
7.
8.  public class AmbienteTutorial extends EnvironmentDescription {
9.    public AmbienteTutorial() {
10.       // proprietà  ambiente
11.       light1IsOn = true;
```

```
12.       light2IsOn = false;
13.       wallColor = ligthgray;
14.       floorColor = blue;
15.       backgroundColor = black;
16.       boxColor = red;
17.       archColor = red;
18.       showAxis(true);
19.       worldSize = 20;
20.
21.       // muri
22.       Wall w1 = new Wall(new Vector3d(9, 0, 0), 18, 1, this);
23.       w1.rotate90(1);
24.       add(w1);
25.       Wall w2 = new Wall(new Vector3d(-9, 0, 0), 18, 1, this);
26.       w2.rotate90(1);
27.       add(w2);
28.       Wall w3 = new Wall(new Vector3d(0, 0, 9), 18, 1, this);
29.       add(w3);
30.       Wall w4 = new Wall(new Vector3d(0, 0, -9), 18, 1, this);
31.       add(w4);
32.
33.       // ostacoli
34.       Box b1 = new Box(new Vector3d(-5, 0, -5), new Vector3f(1, 1, 1), this);
35.       Box b2 = new Box(new Vector3d(-5, 0, 5), new Vector3f(1, 1, 1), this);
36.       Box b3 = new Box(new Vector3d(5, 0, 5), new Vector3f(1, 1, 1), this);
37.       Box b4 = new Box(new Vector3d(5, 0, -5), new Vector3f(1, 1, 1), this);
38.       add(b1);
39.       add(b2);
40.       add(b3);
41.       add(b4);
42.       Arch a1 = new Arch(new Vector3d(0, 0, 0), this);
43.       a1.rotate90(1);
44.       add(a1);
45.   }
46. }
```

La prima cosa da notare è che la classe eredita da *EnvironmentDescription*. Come si vedrà in seguito infatti, è necessaria una classe di questo tipo per inizializzare Simbad e la simulazione.

Successivamente vengono impostate varie proprietà dell'ambiente quali la presenza di luci, i colori degli oggetti e dello sfondo e la dimensione dell'ambiente stesso. Per l'impostazione di queste proprietà non sono presenti dei metodi e bisogna quindi modificare direttamente le apposite variabili.

In seguito vengono creati quattro oggetti *Wall*, per la rappresentazione dei muri di contorno (in grigio nell'immagine), quattro ostacoli quadrati - *Box* - e un ostacolo a forma di arco - *Arch*. Questi vengono quindi posizionati all'interno dell'ambiente utilizzando le coordinate sotto forma di oggetti *Vector3d*.

Com'è possibile vedere, quindi, questa classe è molto semplice e descrive l'ambiente, le sue proprietà e gli oggetti in esso contenuti.

Viene ora proposto il codice sorgente completo della classe successiva, *Tutorial*:

```
1.  import javax.swing.JInternalFrame;
2.  import javax.vecmath.Vector3d;
3.  import simbad.gui.Simbad;
4.
5.  public class Tutorial
6.  {
```

```
7.      public static void main(String[] args)
8.      {
9.        System.setProperty("j3d.implicitAntialiasing", "true");
10.       AmbienteTutorial ambiente = new AmbienteTutorial();
11.       ambiente.add(new TutorialRobot(new Vector3d(0, 0, 0), "agente1"));
12.       Simbad frame = new Simbad(ambiente, false);
13.
14.       frame.setExtendedState(frame.getExtendedState()|Simbad.MAXIMIZED_BOTH);
15.
16.       JInternalFrame[] frames = frame.getDesktopPane().getAllFrames();
17.       JInternalFrame control = frames[0];
18.       JInternalFrame inspector = frames[1];
19.       JInternalFrame world = frames[2];
20.
21.       inspector.setLocation(1, 1);
22.       world.setLocation(inspector.getWidth()+5, 1);
23.       world.setSize( frame.getHeight()-control.getHeight()-70, frame.getHeight()-
    control.getHeight()-70);
24.       control.setLocation(world.getLocation().x, frame.getHeight()-
    control.getHeight()-65);
25.     }
26. }
```

Questa classe contiene il metodo main per l'esecuzione della simulazione con l'ambiente *AmbienteTutorial* e un agente *TutorialRobot*.

La classe principale che viene utilizzata è *Simbad*. Questa classe inizializza il vero e proprio simulatore e richiede come parametro una classe di tipo *EnvironmentDescription*. Viene quindi creata un'istanza di AmbienteTutorial. Il secondo parametro, booleano, indica se la simulazione debba essere eseguita in *background mode*, cioè senza l'utilizzo dell'interfaccia grafica precedentemente presentata. Questa modalità è utile nel caso si vogliano effettuare numerose simulazioni poichè viene eliminato l'overhead dell'interfaccia grafica e l'esecuzione risulta quindi molto più veloce. Nell'esempio il parametro viene impostato a *false*.

Un altro passaggio importante è l'utilizzo del metodo *add()* dell'oggetto *ambiente*. Questo infatti permette di inserire un agente all'interno dell'ambiente stesso, nella posizione indicata. Alla linea 11 viene quindi creato un agente di tipo *TutorialRobot*, denominato *agente1*, e inserito nella posizione (0,0,0) all'interno di *ambiente*.

Le successive istruzioni, irrilevanti ai fini della simulazione, hanno lo scopo di cambiare la dimensione e la posizione della finestra del simulatore e dei pannelli dell'interfaccia grafica. Sono stati utilizzati per evitare di ridimensionare e spostare manualmente i componenti grafici del simulatore a ogni esecuzione.

Infine il codice sorgente della classe dell'agente, *TutorialRobot*:

```
1.  import javax.vecmath.Vector3d;
2.  import simbad.sim.Agent;
3.  import simbad.sim.LampActuator;
4.  import simbad.sim.RangeSensorBelt;
5.  import simbad.sim.RobotFactory;
6.
7.  public class TutorialRobot extends Agent {
8.      private static final int    DISTANZA_ATTENZIONE = 1;
9.      private static final double TRASLAZIONE_VELOCE  = 0.3;
10.     private static final double TRASLAZIONE_LENTA   = 0.1;
11.     private static final double ROTAZIONE_LENTA = 0.3;
12.     private static final int    ROTAZIONE_VELOCE    = 1;
```

```
13.     private static final int    NUMERO_SENSORI  = 8;
14.     private static final double VELOCITA_CROCIERA   = 0.8;
15.     private RangeSensorBelt sonars; // sonar o bumper
16.     private LampActuator lamp;
17.     private double direzione = 1;
18.
19.     public TutorialRobot(Vector3d position, String name)
20.     {
21.         super(position, name);
22.         sonars = RobotFactory.addSonarBeltSensor(this, NUMERO_SENSORI);
23.         lamp = RobotFactory.addLamp(this);
24.         RobotFactory.addCameraSensor(this);
25.     }
26.
27.     public void performBehavior()
28.     {
29.         resetLamp();
30.
31.         if (collisionDetected())
32.         {
33.             lamp.setOn(true);
34.             return;
35.         }
36.
37.         if (!hasFrontalHit())
38.         {
39.             prosegui();
40.             if (Math.random() > 0.75)
41.                 direzione = -direzione;
42.             return;
43.         }
44.
45.         if (isTooClose())
46.         {
47.             lamp.setBlink(false);
48.             lamp.setOn(true);
49.             setTranslationalVelocity(TRASLAZIONE_LENTA);
50.             setRotationalVelocity(ROTAZIONE_VELOCE*impostaDirezioneRotazione());
51.         }
52.         else
53.         {
54.             lamp.setBlink(true);
55.             lamp.setOn(false);
56.             setTranslationalVelocity(TRASLAZIONE_VELOCE);
57.             setRotationalVelocity(ROTAZIONE_LENTA*direzione);
58.         }
59.     }
60.
61.     private void resetLamp()
62.     {
63.         lamp.setBlink(false);
64.         lamp.setOn(false);
65.     }
66.
67.     private void prosegui()
68.     {
69.         setTranslationalVelocity(VELOCITA_CROCIERA);
70.         setRotationalVelocity(0);
71.     }
72.
73.     private boolean hasFrontalHit()
74.     {
75.         return sonars.hasHit(0) || sonars.hasHit(1) ||  sonars.hasHit(NUMERO_SENSORI
    -1);
76.     }
77.
```

```
78.     private boolean isTooClose() {
79.         boolean leftClose  = sonars.getFrontLeftQuadrantMeasurement() < DISTANZA_ATT
    ENZIONE/(double)(2);
80.         boolean rightClose = sonars.getFrontRightQuadrantMeasurement() < DISTANZA_AT
    TENZIONE/(double)(2);
81.         boolean frontClose = sonars.getFrontQuadrantMeasurement() < DISTANZA_ATTENZI
    ONE;
82.         return leftClose || rightClose || frontClose;
83.     }
84.
85.     private double impostaDirezioneRotazione()
86.     {
87.         double rotazione = 1;
88.         if (sonars.getFrontLeftQuadrantMeasurement() < sonars.getFrontRightQuadrantM
    easurement())
89.             rotazione = -rotazione;
90.         direzione = rotazione;
91.         return rotazione;
92.     }
93. }
```

La classe *TutorialRobot*, come già detto, descrive l'agente della simulazione, eredita quindi dalla classe *Agent*.

Nel codice inizialmente vengono definite alcune costanti di comodo per i valori delle diverse velocità utilizzate nella descrizione del comportamento dell'agente.

Il costruttore, dopo la chiamata a *super()*, inizializza i sensori e attuatori dell'agente. In particolare viene utilizzata *RobotFactory* la quale è una classe di appoggio per fornire l'agente dei diversi sensori e attuatori presenti nella libreria. Alcuni dei metodi che si possono trovare sono i seguenti:

- *addBumperBeltSensor()*, aggiunge una cintura di bumper;
- *addCameraSensor()*, aggiunge una telecamera;
- *addLamp()*, aggiunge una lampada;
- *addLightSensor()*, aggiunge un sensore di luce;
- *addSonarBeltSensor()*, aggiunge una cintura di sonar;
- *setDifferentialDriveKinematicModel()*, modifica il robot dotandolo di Differential Drive.

Successivamente al costruttore si trova l'override del metodo *Agent.performBehaviour* il quale verrà chiamato a ogni ciclo della simulazione come descritto in precedenza.

E' da notare che in questo caso non viene utilizzato il metodo *Agent.initBehaviour*, il quale verrebbe utilizzato solamente durante la fase di inizializzazione del simulatore, poichè non sono necessari particolari accorgimenti per l'agente implementato.

Nel metodo *performBehaviour* viene quindi definito il comportamento che l'agente deve tenere nelle diverse situazioni che possono presentarsi durante la simulazione. In questo caso, essendo l'agente fornito di bumper (sensori di contatto) e sonar (sensori di distanza), vengono considerati i seguenti stati:

- collisione, i bumper indicano che l'agente ha toccato uno dei muri o uno degli ostacoli;
- avvicinamento, i sonar della parte frontale indicano che l'agente di sta avvicinando a un ostacolo;

- distanza di attenzione, i sonar della parte frontale indicano che l'agente è eccessivamente vicino a un ostacolo e deve quindi procedere con attenzione;
- nessuno dei precedenti, l'agente può continuare a muoversi a velocità sostenuta lungo la direzione corrente senza pericoli immediati.

Per rendere più leggibile il metodo *performBehaviour* sono stati aggiunti alcuni metodi di appoggio per piccole parti della logica del comportamento dell'agente e di gestione dei sensori e attuatori. In particolare:

- *resetlamp()*, spegne totalmente la lampada dell'agente;
- *prosegui()*, imposta la velocità di crociera dell'agente e indica di procedere lungo la direzione corrente;
- *hasFrontalHit()*, indica se i tre sonar frontali stanno rilevando degli oggetti (se questi sono troppo lontani non vengono rilevati e i sensori non forniscono alcun dato);
- *isTooClose()*, se *hasFrontaHit()* restituisce true allora vengono valutati i dati dei sonar per capire se la distanza dell'agente dall'ostacolo è troppo ristretta. Viene dato peso maggiore al sonar frontale, mentre la media dei sonar di destra e sinistra viene misurata con un peso dimezzato rispetto a quello frontale;
- *impostaDirezioneRotazione()*, osserva la posizione dell'agente rispetto all'ostacolo e imposta la direzione di rotazione nel senso che porta l'agente il più velocemente possibile lontano dall'ostacolo stesso. Indica quindi la direzione di rotazione ottima per evitare l'ostacolo.

Introdotti i metodi di appoggio è possibile tornare alla descrizione del metodo *performBehaviour*, nel quale, per evitare un eccesso di if annidati è stata spesso utilizzato il *return*. Il metodo è molto semplice e si compone dei seguenti passi:

1. Reset della lampada nel caso che al ciclo precedente fosse accesa o lampeggiante;
2. Se viene rilevata una collisione viene accesa la lampada e l'agente viene lasciato fermo nella posizione in cui si trova;
3. Se non ci sono rilevazioni da parte dei sonar frontali viene fatto proseguire l'agente lungo la direzione corrente alla velocità di crociera;
    - Per aggiungere una fonte di non determinismo viene in questo punto deciso, al 75% di possibilità, di cambiare l'ultima direzione di rotazione utilizzata. Questo espediente però viene utilizzato solamente quando l'agente si trova a distanze non critiche dagli ostacoli. L'obiettivo è di avere una maggiore varietà nei percorsi dell'agente all'interno dell'ambiente di esempio.
4. Arrivati a questo punto significa che i sonar frontali hanno effettuato delle rilevazioni. Se queste indicano che l'agente è troppo vicino all'ostacolo allora viene accesa la lampada con luce fissa, viene impostata una lenta velocità di traslazione e una elevata velocità di rotazione nel senso ottimo per allontanarsi il più velocemente possibile dall'ostacolo;
5. Se la distanza non è critica allora viene accesa la lampada con luce lampeggiante e viene ridotta la velocità di traslazione a un valore medio e viene impostata una piccola velocità di rotazione in modo che l'agente devi leggermente la direzione rispetto a quella corrente. In questo caso viene mantenuta la direzione di rotazione prescelta durante la fase di crociera in modo casuale. Questa direzione potrebbe far deviare l'agente verso l'ostacolo piuttosto che farlo allontanare. Il risultato, come detto, è un comportamento non deterministico nella

visita dell'ambiente da parte dell'agente. Nonostante questo "fattore di rischio", nelle prove effettuate, con i valori di velocità di traslazione e rotazione riportati nel codice, l'agente collide con gli ostacoli molto raramente.

Nella Figura 8, proposta precedentemente, è possibile osservare l'agente in una fase in cui i sonar rilevano il muro nelle vicinanze, quindi la lampada è lampeggiante (con colore rosso), la velocità di traslazione viene moderata e viene deviato il percorso con una piccola velocità di rotazione. In questo caso in senso antiorario (-0,3 rad/s), portando quindi l'agente ad avvicinarsi al muro piuttosto che ad allontanarsi. Quando l'agente risulterà troppo vicino all'ostacolo allora rileverà l'errore e imposterà un'alta velocità di rotazione nel verso che lo porta ad allontanarsi il più velocemente possibile.

Conclusioni

In questo documento è stato presentato il simulatore Simbad. Tramite l'utilizzo del linguaggio Java esso permette di programmare e quindi simulare il comportamento di agenti autonomi all'interno di ambienti 3d. E' possibile eseguire le simulazioni sia passo-passo, con visualizzazione in tempo reale della ambiente, sia in modalità *batch,* per l'esecuzione di un numero elevato di simulazioni senza la richiesta di interventi da parte dell'utente.

Come visto nell'esempio proposto, lo strumento è risultato di facile utilizzo sia per la parte di simulazione sia per la parte di programmazione di un agente relativamente banale. L'utilità teorica e didattica di questa libreria è quindi indubbia.

Sono presenti però alcuni risvolti non ottimali. Il primo di tutti consiste nel fatto che l'ultima versione di Simbad, utilizzata per questo documento, risale al 2007. Nonostante questo tutte le funzionalità provate sono risultate funzionanti. E' da segnalare un'iniziale difficoltà dell'installazione su una macchina a 64 bit, ormai piuttosto comune, a differenza del 2007. Per quanto riguarda le API di programmazione dell'agente bisogna dire che queste risultano a volte oscure, complice la documentazione un po' scarna o assente per alcune classi. Inoltre le funzionalità risultano leggermente limitate ma questo è dovuto al volere dei creatori di Simbad nel mantenere lo strumento di semplice e immediato utilizzo. Trattandosi però in un framework, oltretutto open source, nulla vieta all'utente di implementare e aggiungere nuove funzionalità.