3D Framework for Machine Learning

A 3D simulation framework using open source projects and proprietary solutions

Kjartan Akil Jónsson

VERKFRÆÐI- OG NÁTTÚRUVÍSINDASVIÐ / SCHOOL OF ENGINEERING AND NATURAL SCIENCES

HÁSKÓLI ÍSLANDS / UNIVERSITY OF ICELAND

3D Framework for Machine Learning

A 3D simulation framework using open source projects and proprietary solutions

Kjartan Akil Jónsson

A 30 credit units Master's thesis

Tutors

Dr. Hjálmtýr Hafsteinsson

Dr. Tómas Philip Rúnarsson

Department of Computer Science

Faculty of Engineering

University of Iceland

Reykjavik, June 2009

3D Framework for Machine Learning

A 3D simulation framework using open source projects and proprietary solutions

A 30 credit units Master's thesis

Department of Computer Science

Faculty of Engineering

University of Iceland

Hjardarhaga 2-6

107 Reykjavik, Iceland

Telephone: + 354 525 4000

Reykjavik, Iceland 2009

ABSTRACT

Science has made dramatic progress within the field of robotics and other types of adaptive machines. The future direction seems to be targeted at learning machines being able to cope in real world environments. The aim of the work presented in this thesis is to create a realistic 3-dimensional world using open source libraries. From the machine learning perspective the goal is to learn how to drive in an all terrain environment, by controlling the vehicles' acceleration and steering. There has been quite some interested in the machine learning community on learning the control of vehicles, both in the simulated and real world. These solutions often use simplified physics, for example by neglecting friction. With this work the aim is to make these learning environments more realistic and hence more challenging for the machine learning methods. This thesis demonstrates a software framework for working with machine learning methods using open standards. The framework uses the Pyro machine learning library and the PhysX physics engine, resulting in a realistic 3D environment for testing and learning artificial intelligence. The environment offers authors of machine learning methods a challenging environment in which to test learning methods.

ÚTDRÁTTUR

Miklar framfarir hafa verið á undanförnum árum á sviði vélmenna og annara aðlagandi tækja. Framtíðartakmarkið er að vélmenni lærir að takast á við verkefni í raunumhverfi. Markmið þessarar ritgerðar er að búa til raunverulegt þrívíddar umhverfi með því að nota opin hugbúnað. Frá sjónarmiði vélræns lærdóms er markmiðið að læra að keyra í hvaða umhverfi sem er, með því að stjóra hröðun og stefnu. Það er mikil áhuga á því að geta stjórnað hegðun bíls bæði í sýndarveruleikaumhverfi og raunumhverfi. Þessar sýndarveruleikalausnir eru oft einfaldar og vantar þar oft hluti eins og núningur. Með þessu verkefni er markmiðið að gera umhverfi fyrir vélrænan lærdóm sem er raunverulegra en þau hafa verið hingað til og þar af leiðandi meira krefjandi fyrir reiknigreindina. Okkar lausn notar vélræna lærdómassafnið Pyro og eðlisfræðisafnið PhysX sem gefur nokkuð raunverulegt þrívíddarumhverfi til að prófa og læra reiknigreind. Forritið gerir notendum kleift að læra hegðun í flóknu og margbreytilegu umhverfi.

PREFACE

I would like to thank my family for all the support in all of the things I take on.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The aim of the work presented in this thesis is to create a realistic 3-dimensional world, using the OGRE graphics engine for all terrain vehicles, and integrate it with existing machine learning frameworks like Pyro. The realism is achieved through the use of a physics library called PhysX. From the machine learning perspective, the goal is to learn how to drive in an all-terrain environment by controlling the acceleration and steering of the vehicles. There has been quite some interest in the machine learning community in learning to control vehicles, both in the simulated and real world. The simulated environments are, however, often simplified to two dimensions, an example of this being the infamous mountain car problem [6]. Furthermore, they use simplified physics, for example by neglecting friction. The aim of this work is to make these learning environments more realistic and hence more challenging for the machine learning methods. The work focuses mainly on the design and implementation of such an environment, with a special focus on making it accessible to current machine learning environments. With this respect, a greater emphasis was put on open source projects rather than proprietary solutions.

1.1 Objectives and Motivation

Many different environments are used for machine learning purposes. These range from being very simple to quite complex. Some of the simplest environments place the target agent in a two-dimensional static space. This allows the agent to converge fast, since the complexity of the environment does not affect the complexity of the learned scope.

Science has taken drastic steps within the field of robotics and other types of adaptive machines. The future direction seems to be targeted at learning machines being able to cope in real-world environments. This requires the tools to follow the same momentum. This kind of evolution calls for the ability to learn in an environment closer to real-world situations. A simple simulation framework does not translate easily to the real world. This means that the learned behavior and the implemented algorithm are not able to cope in the same way as within the simulation, because the real world is stochastic and physically dynamic, being able to change its environment and displacing objects. This is something to which a smart agent must be able to adapt.

The current state of the art for frameworks targeting these kinds of problem is simulating a robot in a three-dimensional (3D) environment using realistic physics. These implementations have robots that can be controlled by a programmer. In some frameworks the user can assemble the robots himself; in others the robots come already predefined. But all of them focus on a professional audience rather than on a fun gaming perspective (refer to appendix “Simulation Systems” for a list of robotic frameworks).

Allowing players to assemble and control machines would be a fun concept. Players could, through this application, compete against each other from a resource standpoint, controlling machines using artificial intelligence or through direct control.

The goal was to create a platform where the users could control machine robots artificially, using artificial intelligence (AI). The framework should allow users to be able to emulate real-world circumstances rather than an ad-hoc representation. This also challenges the learning algorithm to be a fast learner, since the real world does not allow fast-forwarding time, minimizing the time to wait for the learner to converge to the target behavior. The agent must be able to learn in real time.

The framework should also be able to be adaptable so that if a user would want to read a specific set of data from our environment, the framework should be able to provide it, using simulated devices to accomplish this, thus being able to exchange and build a machine from a set of devices. These devices should also be able to provide inputs and outputs, where the input would be used by a machine learning implementation and the output is the action taken by the agent based on the learned behavior.

The limitation of the learning behavior should be the limitation of the devices creating the robot. Thus a robot car using sensors with one meter range should only be able to drive the environment adapting to objects within one meter proximity. If the robot needs to plan ahead, it would require a GPS device that has information about the entire scene. It also limits the robot output since the only thing the agent controls is the rotation of the front wheels and the acceleration speed and direction.

Interfacing against many different devices complicates the AI algorithm if each device requires specific functions of getting and setting data. This resulted in another requirement; the interface of these devices should be abstracted so that a device, for instance an infrared and sonar sensor device, could share an abstracted interface, resulting in limited or no change of code when exchanging these devices on the robot. This also simplifies the complexity of getting or setting device-specific data, allowing the implementation to be focused on machine learning rather than interfacing with the robot.

Being able to try out the different AI implementations that users can think of is important when prototyping. This requires an implementation of a framework that has different kinds of machine learning algorithms already implemented. Creating such an AI framework is quite complex and creating a framework that is usable by others in a simple and effective manner is even more difficult. This requires a framework that is well documented, well implemented and has many examples. From the user's standpoint, it is often difficult to use such a library, especially when the users need to get familiar with the architecture and interface before being able to implement their ideas. This means that learning the architecture better be worth the user's time. Thus, it was preferable that the AI framework would be open source. Mature open source projects often have a strong community. This helps our "Machine Race" community as they can get support from a broader range of sources.

Prototyping is important for both simple and complex solutions when developing. The faster you can try out an idea, the better. Resetting the application by restarting is time-consuming, so being able to implement machine learning ideas within the framework without the need to reset the program or scene is important. Scripting solves this because it is a highly dynamic form of programming, allowing a user to reload changes made to the script on the fly. It was also important that the scripting language had proven background and a strong community so using the programming language would be as easy as possible. The script would also need to support ideas from LISP and other AI programming languages. It was also preferable that the programming language would be easily integrated to the framework and have libraries such as an AI library already developed for

it, since the user would need to use those libraries when developing the machine learning algorithms.

Within a community, people interact with peers. Having community support within the application framework reminds users that they are not alone. The online community would be a common place to communicate with other users by comparing shared experiences, achieved goals, ideas or progress. This also allows a user to to start up their environment from any other computer and view the game state from the online website.

Graphics play an important role for first impressions because users often judge the maturity of a project based on its graphics. It is also important for the experience as it is easier to get immersed into the game if the graphics are appealing. Having appealing graphics was also important in creating an artistic environment. It also required us to make an environment that people could relate to, introducing a high level of immersion. This would attract people to the product and prolong the user experience.

1.2 Contribution

To date, common implementations of AI learning simulations have had static environments. These implementations use scenes that have no or limited dynamic factors like physics that influence the learning behavior. The environment contributes to how well the machine learning implementation converges because it is the surroundings that must be learned to successfully accomplish the task at hand.

This thesis covers development of a platform for testing artificial intelligence implementations within a fully physical environment, using a high end physics library from NVIDIA[1]. This library is used in recent games simulating realistic-behaving scenes. This environment is fully dynamic and allows for challenging machine learning behavior.

Simulators exist that allow users to control “robots” using different programming languages within a scene. These do, however, often not include a ready-to-use machine learning library and require some preparation for setting up the programming workspace.

The framework has an integrated Python[2] scripting language that is known to be easy to use and learn. Python compiles and runs our machine learning algorithms with help from the Pyro[3] library, which has a strong community of experts within the field of robotics and has won awards for excellence in engineering education courseware. The Pyro library has common machine learning algorithms implemented and can be combined to create complex learning behaviors. Users can edit, compile and load the Python script while running the program, allowing them to easily try out and tweak different machine learning methods without reloading the system.

The Pyro architecture abstracts the device driver interface, allowing users to exchange and try different devices without having to alter the AI scripts. This allows a user to try out different devices easily. Different devices can solve the same problem using different methods. This creates a lot of user-specific machines with different types of setup and possible objectives.

---

[1] NVIDIA 2009, <http://www.nvidia.com>

[2] Python 2009, <http://www.python.org>

[3] Pyro 2009, <http://pyrorobotics.org/>

Every device has its own graphical user interface (GUI) notifying the users of the device's current state through visual and numeric presentations. This presentation keeps the user up to date with the device's current state.

A variable watcher and console allows users to view the updated values used by the devices within the runtime environment. The user can also query and evaluate the system values using Python expressions, helping the user to get the value of many state combinations.

The scene is displayed using a rendering library that utilizes high end graphics card processors to render the world and its objects. The graphics library is commonly used in development and deployment of commercial games.

The world has a built-in economic system, allowing users to exchange resources within the framework. This allows users to build, collect and distribute resources through a monetary system. From a game standpoint the monetary system allows users to compete financially through higher-level collaborate game play.

The framework promotes a strong community by offering users a private space as well as a shared documentation area. Here, users can share information, ideas and methods used within the framework.

1.3 Overview

Chapter one, "System Overview", introduces an overview of the application framework; what has been created, how it was assembled and the results.

Chapter two, "System Technology", covers how the technology was created, describing everything within the client-server architecture and its details. The server is described in detail, particularly what the different servers do and why. Then the thesis continues to discuss the client layered architecture, ending with describing how Pyro and its devices are implemented.

Chapter three, "System Functionality", shows the reader how the programming system functions from a user point of view, from creating a robot to creating the script to control it. These ideas will be described using reinforcement learning concepts, showing how "brains" are assigned to robots within the game framework. The chapter ends by showing examples of "brains" and how they can be used.

The Conclusion sums up what has been developed and the project's next steps.

2 SYSTEM OVERVIEW

This chapter contains an overview of the application framework. The first section describes what has been created, then how the implementation of the system was incorporated with regard to the legacy system.

The last section, “Results”, gives an overview of the game concepts used and their functionality within the system, ending up with describing Pyro concepts used by the AI “brain”.

2.1 System Description

The system is a platform for testing artificial intelligence implementations within a dynamic environment called Machine Race. The application uses a high end physics library from NVIDIA used in recent games to simulate a realistic environment. This environment is rendered in 3D, allowing an immersive representation of the simulated scene.

The requirement called for a system where a player could control a robot that was definable by the user in a large world and an active community. The goal of the new system required support of the following features:

- Registration
- Community support
- Network support
- Unlimited landscape
- User controlled world
- Device based machines
- Artificial intelligence
- Resource system

A user begins by creating a profile within the framework. The registration profile is used to store the client’s state in the community. User interaction within the world is stored in a state server. User location and orientation is but a part of the information stored by the servers. The data is stored until the next time the user runs the client and is updated accordingly. The client can alter some state settings himself, such as the machine and device setup.

The user authenticates himself to the community, using the registration profile. The communication is always managed through an accompanying user name. This way, interaction within the community is on a “personal” basis. The community support that is

implemented has four levels across the system that will be explained in turn. The most basic level is that a user can have his/her own profile. This profile stores simple user information such as the user's full name, alias and password.

The second level is that a user can have a game-dependent state stored. This includes machine setup, economy and resources, as well as position and orientation of these entities. This is available through a simple online interface where the user logs into his account on a web page. On this web page (called state), a user has access to alter his economic, resource and machine state directly.

The third level of community support is information sharing, where every player shares information through wiki-style pages.

The fourth and final interaction level is a direct interaction between players in real time through the game client. This level of interaction is currently disabled due to synchronization problems between game clients.

Creating a robot in a world where it could interact without being limited by the world size resulted in the implementation of an unlimited world where the landscape is based on patches. This patch repeats when the user travels across the landscape. Within this landscape, a player could also alter his world. This includs moving objects around and changing the scene layout. This is generically supported by the implementation of the physics library. Each entity within the world has different physical attributes. All of the dynamic entities must have physical topological shapes like boxes, spheres and capsules, but the internal visual representation can be any mesh shape defined. These dynamic shapes can be combined together to create complex physical devices.

The notion of combining entities together to form complex devices is inherently supported by the OGRE3D[4] library. They are "abstract", having no physical behavior, only a visual representation within the world; like a camera or a light bulb. Using the same concept to create complex machines from a collection of devices is possible by using joints in the physics library, resulting in physical devices attached to each other. These devices can have physical behavior; like vehicles driving around in the physical world.

The concept machine is then a collection of these devices that form a user-specified actor. The actor can then be controlled by the player within our world. The control of a machine can be direct or through AI scripts, which can be user-specified through Python and the Pyro library. The user specifies the script behavior by writing Python code. This is simple text-based (ASCII) code written in a file that is compiled during run-time. The user can at any time alter the code and reload it without having to restart the client system. Through these scripts, the user has access to drivers that control the devices that in turn control the entire machine. This allows the machine to behave like a robot with user-specified artificial intelligence called a "brain".

The robot can be controlled directly through Pyro scripts. The direct controls are the user controlling the machine movements using the keyboard and mouse, implemented through the Python "brain" script. Only that this "brain" listens for the keyboard keys and applies movements to the devices just like the AI scripts.

The devices, machines and world objects are all regarded as resources. These resources can be exchanged between users through the online interface or collected. All resources

[4] OGRE3D 2009, <http://www.ogre3d.org/>

can in turn be exchanged between users. This is done through an implemented economic system using basic bookkeeping rules. Every user has many different accounts. These are bookkeeping accounts to manage the user and world economy. Using basic bookkeeping rules allows a user to market, buy, sell and exchange resources. This way, an exchange of an item is "sold" by moving between accounts, altering it from physical entity to in-game currency and vice versa.

2.2 Implementation

The current system is based on a previous B.S thesis. The previous system had to be upgraded to fulfill our goals (see appendix, "Legacy System", on information about the previous system). Updating the system required us to change base components such as the SDK libraries and in some cases add new libraries allowing more enhanced features. The added libraries were:

- Django[5] (server side web framework)
- GeoDjango[6] (server side geographic information system service based on Django)
- PostGIS[7] (server side geographic information system database)
- Hikari[8] (graphical user interface on client side)
- RakNet[9] (client side networking library)
- PhysX[10] (client side physics library)
- NxOgre[11] (client side PhysX wrapper)
- Pyro[12] (abstracted machine learning library on client side)

The online websites run on the Django framework. Working with Django sites is easy once you get used to the different mindsets about the class and table relation, but it gets more complex when you want to use these same ideas with the relational data and general data queries required in our implementation. Django converts Python classes to tables. The programmer defines the class variables and Django builds the table using these variable fields. Each class extends the framework, allowing you to access methods to query the table. The relational aspects get complex because the querying method used is a Django solution and has little relation to the well known SQL syntax. This made working with

---

[5] Django 2009, <http://www.djangoproject.com/>

[6] GeoDjango 2009, <http://geodjango.org/>

[7] PostGIS 2009, <http://postgis.refractions.net/>

[8] Hikari 2009, <http://www.ogre3d.org/wiki/index.php/Hikari>

[9] RakNet 2009, <http://www.jenkinssoftware.com/>

[10] PhysX 2009, <http://developer.nvidia.com/object/physx.html>

[11] NxOgre 2009, <http://nxogre.org/>

[12] Pyro 2009, <http://pyrorobotics.org>

relational data more complex than querying using regular SQL syntax. On the other hand, the code for building sites was simple and straightforward. To get the current state from the game client, communication with the state servers was managed through Pythons XML-RPC protocol.

To save client state in a common world-state database, Django was used. This allowed the use of GeoDjango, which is a geo-spatial framework built on Django. We can then store the game state based on geospatial locations. Thus the system can query the world around the client instead of the entire world state, filtering it on the client's end. Most common databases support Geo spatial queries like PostGIS. Postgres[13] was used because it is free, stable and supported PostGIS extension, allowing us to use Geo spatial data.

Most applications need a graphical user interface for the user to interact with. OGRE3D library comes with CEGUI[14] as the default GUI. CEGUI is a separate open source project that is very powerful, but it was artistically challenging to use this GUI library to make a suitable user interface to the game. We chose to use Hikari GUI library. The Hikari package loads standard Flash[15] files (*.swf) that can register call-backs to, and call, the Flash interface as needed. This allowed us to use standard Flash files as GUI instead of CEGUI. Flash has better artistic tools and a more dynamic interface to work with, allowing for a more informative feedback to the user.

RakNet network code was chosen instead of a proprietary network solution due to problems establishing connections between clients. Using this library, we established more reliable and secure tunnels between clients.

2.3 Results

Implementing these libraries to the legacy technology base improved the technology and the user experience. This section discusses these results from the user's point of view, beginning with the user. Then the machines and its devices are discussed as well as covering how the user can tune his/her machine. The section ends with the community aspects of the framework.

User

When the users have registered they can log in to their "private ccount" page to create a machine. This machine uses devices. These devices can be upgraded through the "private account". With upgrades, the user can enhance the specific functionality of their machine, enabling it to work effectively within the environment. Users can even control more than one machine at a time.

Machine

A user can have one or more machines. Each machine can have different devices. This way, each machine can have user-specified functionality. These machines are created out of independent devices. A user can combine devices to build machines suiting their own needs. The list of devices is available in the section "Devices". This could for instance

[13] Postgres 2009, <http://www.postgresql.org/>

[14] CEGUI 2009, <http://www.cegui.org.uk/>

[15] Flash 2009, <http://en.wikipedia.org/wiki/Adobe_Flash>

create different types of machines. The user could have a vehicle transporting goods and another rover to investigate places that are difficult to maneuver in, simulating excavation of minerals.

Figure 2.1 shows two vehicles driving around the scene, both controlled by the user. In this image, the currently selected machine, “robbie”, has the “AvoidBrain” assigned to it. The user has the device windows; “radio”, “camera”, “infrared” and “AI Pyro” opened, displaying his/her current status and capabilities.

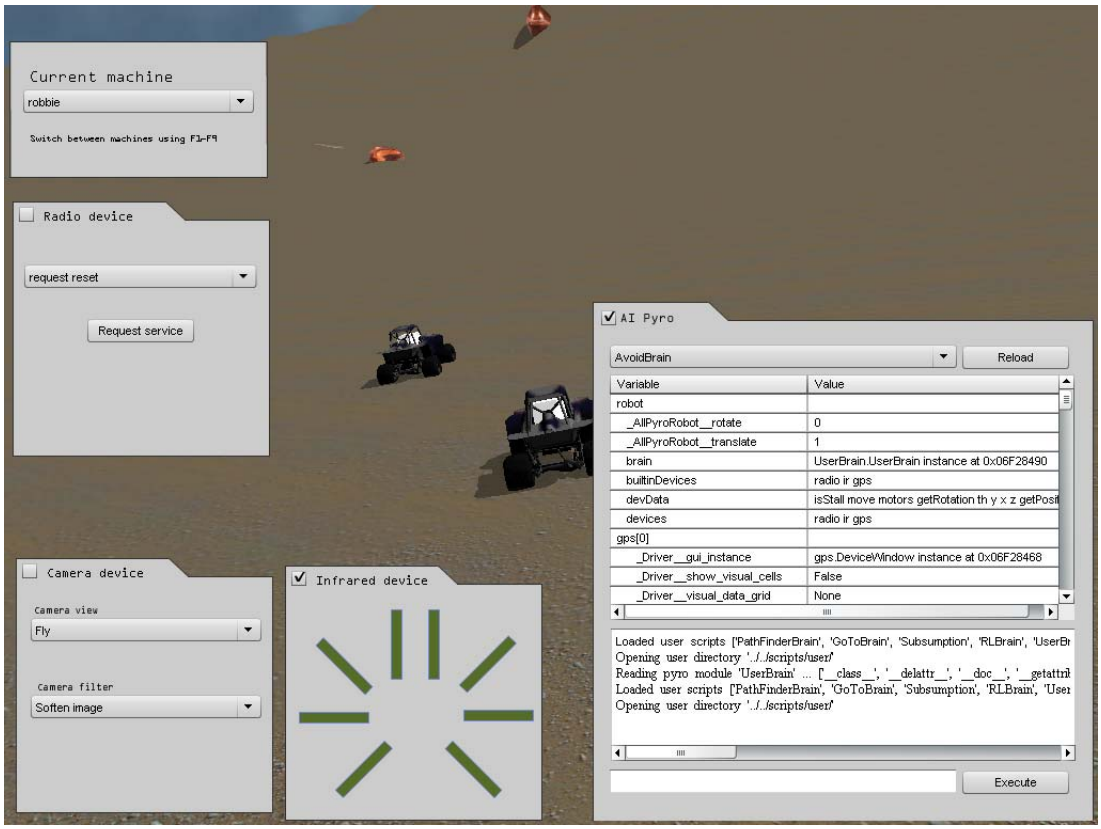


Figure 2.1 In-game scene with two cars driving around

Devices

Devices are building blocks that machines are made of. Each device serves a specific purpose. Devices are grouped into four main categories; base, body, arms and head. All of these groups denote where on the robot the device is applied. The machines are assembled as follows:

> Common to all machines is the base device. This device can either be fixed on a location or dynamic, maneuvering the entire machine around in the environment.
>
> To the base device, a “body” device is added. This device can be used as a container. Currently the IR (infrared) device is supported as a “body” device for sensing the machine’s environment.
>
> The “body” can have “arms” and is used for devices that require more precision when controlling them.

The head device is attached on the machine and is located above all other devices. Example of a head device is the “camera”.

Each device is built based on blueprints. These blueprints create a physical instance. A physical instance can be connected to other physical instances.

Devices are controlled by drivers. Each driver has a graphical user interface window. This window is displayed and updated by the device and drivers.

Brains access abstracted interfaces provided by the device drivers. The device drivers are accessed through the robot interface. For instance, you can access an IR (infrared sensor) device by:

```
robot.ir[0]
```

The above example accesses the robot’s first infrared (IR) device located at index “0”. This provides the AI programmer access to the IR device methods. Through similar schemes, position of the machine can be controlled. Moving a machine is done through:

```
robot.move(rotation-amount, translate-power)
```

Where “`rotation-amount`” ranges from “-1” to “1”, “-1” represents as far to the left the vehicle can rotate and “1” is to the right. “`translate-power`” is a value from “-1” to “1” where “1” applies full torque power to the motored wheels propelling the vehicle forward while “-1” backs the machine up.

Devices behave differently depending on what they are used for. For instance, calling the above function (`robot.move`) will differ between rovers and “terrain vehicles”. A terrain vehicle can only rotate its front wheels while Rovers will rotate the entire vehicle around its own Y axis. Other static-based devices like towers will only apply the “`rotation-amount`” ignoring the “`translation-power`”.

Upgrading

A user can upgrade or downgrade machine devices. Players can either choose to create a device from a blueprint or use an already existing device online at the “purchase” page.

Creating a new device from blueprint allows the user to select from all the available blueprints Machine Race has to offer. This is done by selecting the device blueprint and pressing the “order blueprint device” button on the “purchase” web page.

Artificial intelligence

A user can have many machines. Each machine can be controlled by a “brain”. The default brain is the “UserBrain” which is the player controlling the machine. But users can use more complex scripts that utilize artificial intelligence. This is accomplished by implementing artificial intelligence behavior using Pyro, which is a library accessible through Python. A user can create new brain files that can be loaded through the GUI in real time. The goal of the Pyro robot project is “to provide a programming environment for easily exploring advanced topics in artificial intelligence and robotics without having to

worry about the low-level details of the underlying hardware"[16]. Pyro has a good tutorial and learning section on how to use Pyro as well as explanations on other concepts in Machine Learning. It "won the National Engineering Education Delivery System (NEEDS) 2005 Premier award for excellence in engineering education course ware. Leveraging the pedagogical scalability of the python programming language, Pyro has been adopted and adapted into dozens of undergraduate courses and research projects."[1]

"Brain files" are files that have a specific interface and are stored in the users game scripts directory (see section 'Brains' for more details). The brain class extends the Pyro brain and is initiated when a user applies the brain to a machine. The same brain can be used across different machines because devices are abstracted, making them easily exchangeable. For instance, the number and positions of sensors presented varies from platform to platform. "In order to relieve a programmer from the burden of keeping track of the number of sensors (and a unique numbering scheme), we have created sensor groups: front, left, front-left, etc. Thus, a programmer can simply query a robot to report its front-left sensors in robot units. The values reported will work effectively on any robot, of any size, with any kind of range sensor, yet will be scaled to the specific robot being used" [5]. This abstraction allows different robots to use the same script between different simulated devices, so we can control different machine setups using the same brain.

"Regardless of the kind of drive mechanism available on a robot, from a programmer's perspective, a robot should be able to move forward, backward, turn, and/or perform a combination of these motions (like move forward while turning left). We have created three motion control abstractions: translate, rotate, and move. The latter subsumes both translate and rotate and can be used to specify a combination of translation and rotation. As in the case of range sensor abstractions, the values given to these commands are independent of the specific values expected by the actual motor drivers. A programmer only specifies values in a range -1.0..1.0"[5]. This scheme translates to other devices as well. "As in the case of range sensor abstractions, the values given to these commands are independent of the specific values expected by the actual motor drivers. A programmer only specifies values in a range -1.0..1.0"[5]. This makes all of the devices use similar behavior, accessed using the same uniform interface metaphor. "The abstractions presented above provide a basic, yet important functionality. We recognize that there can be several other devices that can be present on a robot: a gripper, a camera, etc"[5]

Community

The Machine Race framework has support for an active community built into the system architecture. This was done by incorporating wiki-based pages for all the users and forcing them to authenticate themselves before writing anything on the online pages through a login page. This way we hope that users communicate and share ideas as well as game content with each other on a personal basis.

Through the online web page, "State Map", a user can view the world from above. Users can scroll the map, zooming in and out to find other users and their location in the world. Figure 2.2 shows an image of populated rocks in the Machine Race world rendered by Geoserver[17] (compare this to figure 3.3 in the following chapter showing uDig's view of the spatial database).

---

[16] Pyro 2009,<http://pyrorobotics.org/>

[17] GEOServer 2009, <http://geoserver.org/>

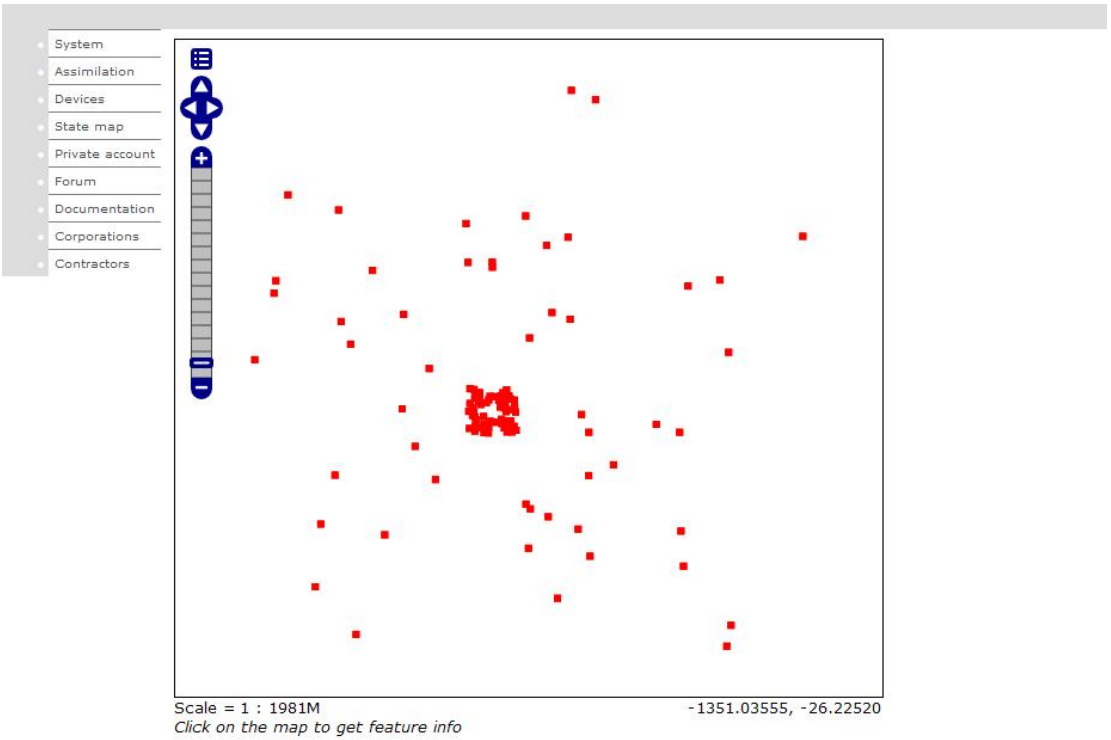


Figure 2.2 Online map showing the world from above

2.4 Summary

This chapter has provided an overview of the application framework, with the first section describing what has been created. Then the implementation of the system was covered, with regard to the legacy system used as basis for this new system.

The last chapter, “Results”, gave an overview of the game concepts used and their functionality within the system, ending with Pyro concepts used by the AI “brain”.

3 SYSTEM TECHNOLOGY

This chapter will cover the underlying technology of the game system in detail. The first section, “Methodologies”, describes how the system was developed, mainly the methods and ideologies used. The “Architecture” section will dive into the client and server aspects and how these communicate.

The server section will describe what servers are running and why, by first covering how the community server site works and what its main function is and then describing what the state server does and how a user can alter its user profile from the game client, using a regular Internet browser. The last section covers the world server and what information it manages.

The client section describes the basic functionality of the game client, focusing on Pyro-centric information. This section covers how the system is installed, updated, executed and rendered, using the open source libraries. And also on how Python and Pyro influence the rendered world through devices.

The development is dependent on a structured development environment. The system is composed of open source projects and proprietary code. All source code is managed using version control systems. These aspects and an overview of what services are outsourced are described in the “Code Control” section.

3.1 Methodologies

The architecture is split into a client and server architecture which is solved independently. The game client is then decoupled and ended up with having a client that has a layered architecture, where each layer focused on a specific functionality. The client has a core system that manages rendering, physics, sound, networking and a script system that handle the game specifics.

A rule of thumb is to not couple the libraries together so that they become codependent on each other. This meant that the interfaces between the layers had to be abstracted. The goal of the abstraction was to exchange libraries easily without affecting the overall information flow.

The server side ended up with having three servers after decoupling, so each server handled its specific function. One handled the community site, another handled the user state while the third handled the world.

After the architecture was decided, agile methodology concepts were used when building the actual game and client framework. In other words; what the system needed was developed, tested and implemented.

3.2 Architecture

The overall architecture is client-server based. The server stores the state that is accessed and distributed to the game clients, where the client has limited access to changing his own state directly unless through session validation.

The system splits the architecture in two categories: client and server. The client is the game client installed on the user's machine and the server gives the client its initial state.

Choosing to use many servers with different purposes resulted in a higher level implementation of load balancing. There are three online servers that have different functions. These servers serve different types of content, i.e. site, state and world content.

The site server registers new users and stores the user's personal information. The state server holds the user's game state and the site server is the community server. The game state server has every entity's state, including the user's state. It handles the economy and resources in the world, except position and orientations of objects. This is what the world server manages. The world server uses a Geospatial Information System to manage spatial queries regarding locations of entities and users.

All these servers delegate information on demand to the client. To communicate with the servers, XML-RPC was used. It is a lightweight remote procedure call format that uses XML to define the functions and arguments. This protocol is used to store the user's state online. Using online servers, any user can access their last state from any computer. The main reason for using online servers is the future plan of supporting multi-player game play.

The packages were abstracted as much as possible, making updates and maintainability as simple as possible, with the core of the client system being our own proprietary C++ code, gluing all the C/C++ libraries together. The focus has been on a clean and straightforward architecture and specialized classes. This way, exchanging the underlying libraries could be done without compromising the overall architecture and potential dependencies.

The OGRE3D library is at the core of the entire system. OGRE starts a rendering loop that updates the rendered frames. The rendering loop also flushes network messages, intersection testing and checks the timer callbacks.

The script interface traps these callbacks from the system. Different callbacks can be registered, notifying the user when the internal state changes.

- Intersection detection
- Timer callbacks
- Network messages
- User interface event, GUI.
- System notification, startup and shutdown

If no callbacks are registered (during start up), no callbacks will be made until the system shuts down. Every call is made through a registered game class that must extend

`GameBase` and be registered through call `SystemHandler.registerGameClass(…)`.

Physical objects are created by forwarding calls to the PhysX wrapper. It handles all the physics-related aspects internally through a separate thread.

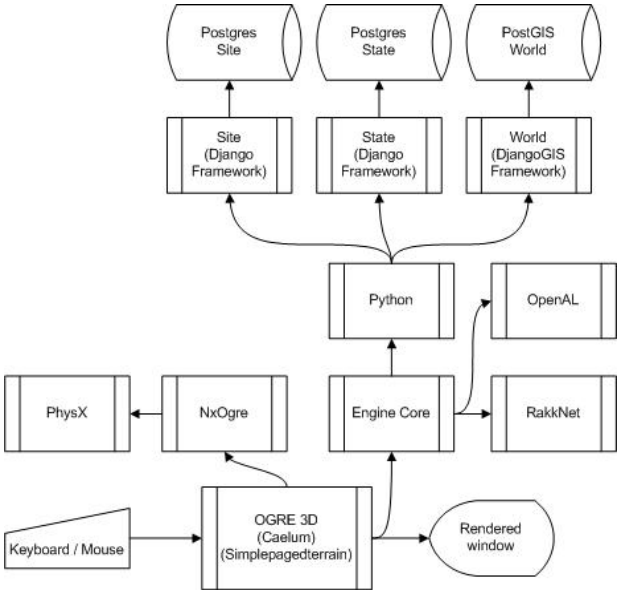


Figure 3.1 Interactions between open source libraries

Figure 3.1 shows an image of the overall architecture with regards to the information flow of the open source packages. Here the OGRE 3D engine runs the main loop rendering the scene and handling both the Physics and I/O while the ”engine core” handles sound, network interactions and Python. Python interacts directly with the online servers, getting and setting the user state. Further details on this interaction are covered in the “Client” section.

3.3 Servers

The system architecture was split into three separate services; site, state and world. This way, each server focuses on its specific functionality. Using this kind of architecture implemented a higher level load balancing, allowing the maintainers to use cheaper online services with lower bandwidth.

Here follows the purpose of these different servers and details regarding them.

3.3.1 Site Server

The community site server[18] is the server that is available for users to view and edit game-related content. The purpose of the site is to invite new players to the game, as well as encouraging a strong and active community. The user can get an overview of the activities involved within the game, read community content and view game capabilities.

[18] Machine Race 2009, http://www.machinerace.com

Users register in the game through the community site. This is done through a registration process that ends with the user being required to confirm the registration. After registering, the user can download the game client from the “Software” page. It includes details on what to download and install so that each new registered client can start playing the game.

The registration process is as follows:

- A user chooses to register by selecting “join”
- Then he signs up, using his full name, e-mail, user name and password
- This information is stored as an inactivated user having a unique identifier key
- An e-mail is sent for confirmation
- The user confirms the registration by opening the mail and clicking the provided link
- The system gets the confirmation through the unique identifier
- The user state is changed to activated user

If the registrant has not confirmed the registration within a week, the account is disabled and later removed. This ensures that the database is not filled with player data that is never used.

Ensuring that every technical aspect works on the fly is difficult. To assist users, we have incorporated information sources of the changes being made and possible solutions to known problems. The page “System Overview” has a F.A.Q, technology section, developer section, general news section and news on patch updates.

After registering, the player can get involved with the online community. Each new player gets his own “contractor” page. This page allows each user to add his/her content through wiki-style syntax. Here, each player can add user-specific text, allowing them to create his/her personal space.

Through the site page each registered user can access their “Personal State” page. The state site stores each player’s personal state. This can be altered by the users themselves. Through these pages, the player can buy, sell, add and remove devices from their machine. A user can have more than one machine created through this interface.

The site server provides only one XML-RPC service. It is the `getRegisteredUser` function and is used by the state server to validate the user name and password.

The site server is hosted on Webfaction using an Ubuntu operating system and the service is provided by the Python-based Django Framework. For more details on the server, refer to the “Code Control” section.

3.3.2 State server

When a user is registering for the first time they must create machines through the online “machine interface”. This machine setup can be altered to suit each player’s needs. This is

done through the state server's "private account" and is accessible through the site page or directly[19].

Users access the state server through a separate log-in section under "Private Account". Here the player needs to log-in using the registered user name and password.

The state server maintains the game state. Game state is every resource state except world positions and community-related aspects. It is responsible for storing and providing the game state on request from clients. This is done through web services such as XML-RPC and regular HTTP web sites.

A resource can be

- a game player, contractor
- a company
- a device
- a machine
- money
- a blueprint
- minerals

Figure 3.2 shows an image of the database relational tables. Here the table "user" is the registered player. The "user" will always be a contractor. But he can be a director of a corporation as well. The corporations have missions and assignments. Just like contractors, each corporation has its own account.

Each resource has an item associated with it, like physical entities, blueprints, makers and raw materials that "extend" the resources, making them share the resource properties of being transferable using transactions.

[19] Machine Race State 2009, <http://state.machinerace.com>

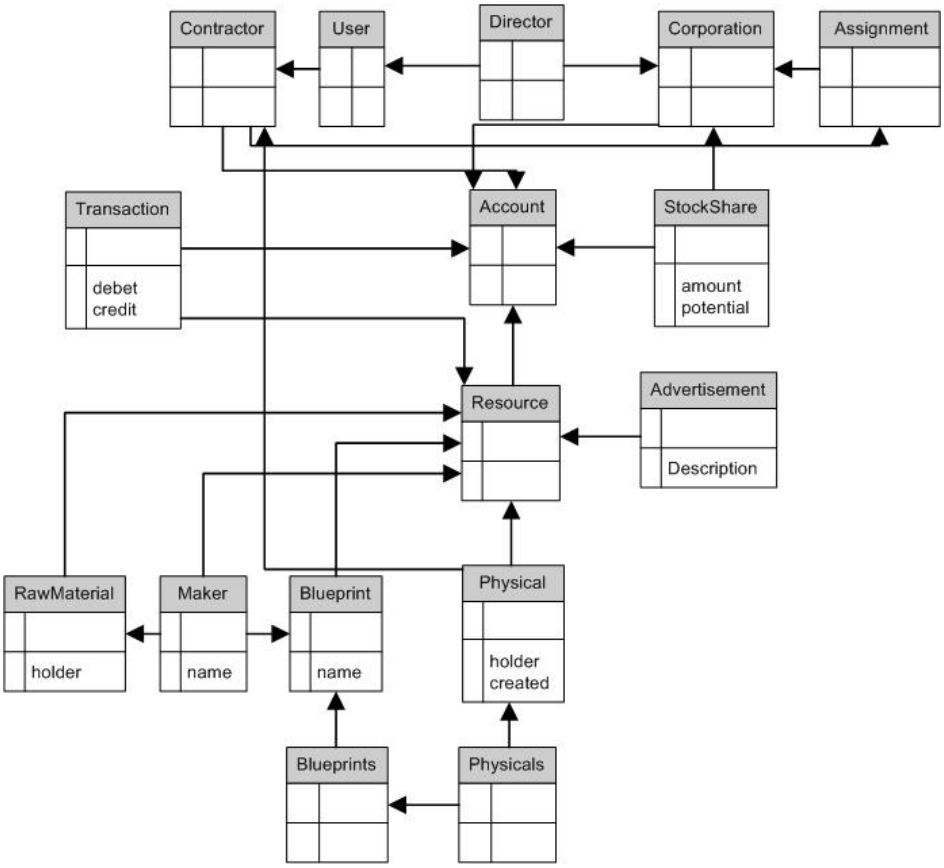


Figure 3.2 Relational database tables used by state

Contractors, as well as companies, have an account. The account has transactions referring to it where each transaction has a value that can have a resource associated with it. When a transaction is made, both parties making the transaction will get a unique transaction number. Both of these accounts will get a transaction mirroring each other's actions. If a resource is involved, the transaction will include the reference to that resource.

Each physical device is associated with a machine. The machine is thus a collection of devices where each machine has a base, body, grip, map and head.

The simplest form of a robot has only a base device. The base device is what everything else is added onto. Currently only one base is supported and it is a vehicle, a four-wheeled device that can drive around the terrain.

The body is the device that can hold or contain other devices, depending on its purpose. Current implementation only supports an infrared device that covers the vehicle area, sensing distances to objects around the vehicle.

The head is a device that is located high above all other devices. This allows the head to be the device that does not have any other devices above or in front of it. An example of a head device is a camera.

Each machine can be changed to suit the needs of the player. This is done online through the ”Private Account“ page and will be updated when the player re-starts the game.

The player can add multiple machines to control at the same time. This is as simple as creating a new machine and assigning devices to it. The machine setup is loaded into the client when the user starts the game. The user is asked to authenticate himself. On success,

his profile is retrieved from the server, containing the machine setup and session information.

The “session” is created upon authentication and is used as an identifier during communication with the server. The user authentication is done by checking the state database if the user name and password match. If no user name is found, we check if the user is a newly registered player. This is done by forwarding the authentication to the site server. The site server will try to authenticate the user by looking him up within its database. If found, it will return the registered user information back to the state server. The state server will then store this information for future authentication. The authentication ends with the sending of the profile and session keys back to the client.

All communication between the game client and the state server is done using account and contractor session keys. These are randomly generated and assigned to the client. They work similarly to the browser session keys. A random string of 128 characters is created, stored and sent to the user as part of the player profile. These keys are then used to authenticate that the user has “logged in” before accessing or altering the player’s state.

This is also a step towards better security since guessing 128 characters is more difficult to accomplish than using logical identification like a user-name or a number. This will make it harder for a user to access and alter the data of other players. This will not prevent man-in-the-middle attacks or identity theft, as is. But future implementations will use secure tunnels (HTTPS).

Here follows a list of XML-RPC calls supported and used by the state server. Every call uses contractor and/or account keys to validate the request. On failure, an exception is raised. This exception is cached and flushed to a local file. Currently, no logging of the failures is done, but future versions will dump all exceptions to a log file that will be read into a statistics database, which is used to analyze player actions.

Here follows the XML-RPC state calls currently used by the client for state synchronization:

- `login`: Authenticates user
- `getContractorBalance`: Returns the contractor’s balance
- `getCorporationBalance`: Checks if the user name is a director
- `getResourceAds`: Returns a list of ads
- `getResourceList`: Returns a list of resources
- `_destroyBlueprint`: Debug function that destroys blueprint
- `createBlueprint`: Creates a new blueprint and vehicle for “account” having name 'name'. If template is set to an existing vehicle blueprint name then we copy those values to our new template, otherwise we create a null-based blueprint. Returns false on failure.
- `getIR`: Returns the infrared sensor (IR) belonging to account “account” having name “name”. If name is None then all the clients IR is returned. Raises `PhysicalException` if the client or IR was not found.

- `getMachine`: Returns the machine belonging to account “account” having name 'name'. If name is “None” then all the client’s machines are returned. Raises PhysicalException if the client or machine was not found.
- `getMachineList`: Returns a list of machines
- `getVehicle`: Returns the vehicle belonging to account “account” having name 'name'. If name is None then all the clients vehicles is returned. Raises `PhysicalException` if the client or vehicle was not found.
- `setBlueprintPart`: Exchanges a blueprint part “type” to “part” of vehicle having “name” of account “account”. If more than one part is owned by the client then we will use the first part found.

The state server is hosted on Webfaction using an Ubuntu operative system, the same server as the site server. The XML-RPC service is run on the Python-based Django Framework.

3.3.3 World Server

The world server’s primary function is to store entities’ position within the world. Geographic Information System (GIS) is used to store this information. This is done through the GeoDjango framework. The GeoDjango framework is a branch of Django web framework.

GIS allows us to work with information that is localized within the range of the user so that it can give the machine devices localized information. This allows the machines to fetch and work with data that is close to their current location.

PostGIS is used as our database server. The server uses open standard protocols. Public research and data collections often use these protocols so little conversion is required when importing common world-data into our server.

Using an open source database we get access to powerful open source tools to edit our world. This is important when working with large amounts of data. Currently uDig[20] is used for editing entity locations in the world.

[20] uDig 2009, <http://udig.refractions.net/>

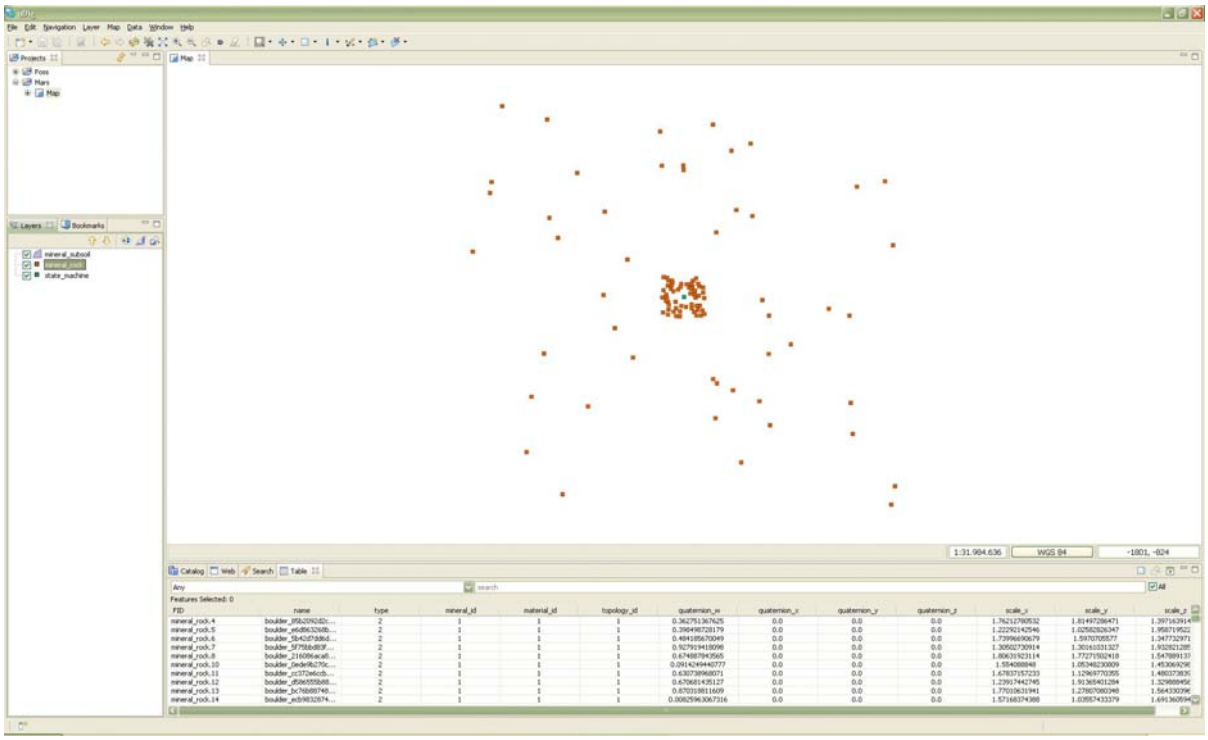


Figure 3.3 uDig view of the GIS database

Figure 3.3 displays current world data through the uDig editor. The world is viewed from above where in the center we have machines (blue dots) surrounded by rocks (brown dots). These rocks are tightly populated closest to the vehicles while more sparse in the outer circle.

GIS information has many layers of information. Current implementation shows three layers. The first layer is the machine locations (blue dots). The second layer is the stones (brown dots), while the third layer is the GPS layer. The GPS layer allows users to store user-defined information to the server database filterable by location, type and value. This can be used by users to store any specific global information that may need to be retrieved at a later state. The last layer is the mineral layer (green layer). Currently, no definitions exist but future implementation will allow devices like spectrometers to check the contents of a location analyzing what minerals are available at a specific location.

The game client connects directly to the world server. The client must log in to the server getting the required session keys. The world server will then take the request and always forward it to site server, making sure that password and user-name are valid. If authentication is successful, the user is added and updated accordingly with the user information.. Then a controller is created using the session key that is returned to the client.

Here follows a list of XML-RPC calls accessed by the client to synchronize its world with the server:

- `clear`: Is a debug function
- `getGPS`: Function allowing user-defined relational data
- `setGPS`: Function allowing user-defined relational data

- `getMachines`: Returns all the machines belonging to controller
- `login`: Authenticates user
- `update`: Will call state server synchronizing newly created data such as new machines creating world positions for them.

The world server needs GeoDjango, which is a branch of Django, to run. Due to a complex installation procedure, it was required that the developer have full control of the operating system. Slicehost was used as our hosting service since it provided full control of a range of operation systems. The service is currently running on a clean install of Ubuntu to which we added Geospatial services such as PostGIS and Geoserver.

3.3.4 Statistics server

The architecture has support for collecting statistics from the clients. This is currently on the server side and is collected from state, site and world when a user interacts with the web pages or calls remote procedures that flush parameters and exception information into a text file. This can be collected and inserted to a table analyzing user interaction.

Currently, this is disabled but future implementation will support detection of faults through exceptions. This exception is caught before we return any values to the user through the XML-RPC protocol. It will be appended to a log while we return default or false values to the user.

3.4 Client

The base client architecture was planned prior to the thesis. This planning focused on the overall architecture. The classes were decoupled as needed, making the open source libraries independent. This planning was done during first stages of development here lessons learned during other game engine development projects were used, helping to form the current design.

The game client runs the game on users' personal computers running XP or Vista. It is downloaded from the site server and installed by the user. The client requires PhysX to be installed on the target system before installing the game. The install package will check if the required version of PhysX, v2.7.4 is installed and abort setup if it is missing. If the installer successfully installs on the client machine, it must ensure that the client binaries are up to date. Like many other complex software solutions, our game client is continuously in development. This requires the game client to be updatable so that every user's software version is in sync with the others. These updates are done automatically using the "Patcher". For a detailed description on how this works and on the installer updating process, refer to appendix, "Setup".

The Machine Race client is the main game application that a user interacts with when running the application. During execution, it connects to the online servers getting world state information. It loads the environment and handles real-time interaction between the user and the virtual machine devices.

- OGRE 3D

- OpenAL[21]
- RakNet
- Python
- PhysX (not open source)

Some of these libraries have so-called “add-on” projects. These projects use the base functionality of the libraries to extend their basic functionality.

- Simplepagedterrain, OGRE
- Caelum, OGRE
- NxOgre, OGRE
- Pyro Robotics, Python
- Hikari, OGRE

Source code for these projects (except PhysX) is available on the World Wide Web through the open source repositories. These open source projects are developed and maintained by a strong community of developers. Releases of the open source libraries are frequent. That is why it is important to keep the system up to date as frequently as possible. Updating often, the programmer avoids dependency updates that break the current build across packages.

The Machine Race code is stored on our Subversion server that uses Cruise Control .NET for our automated build and testing process. On failure, the developer receives an e-mail regarding the fault or error, as well as what was changed. The server notifies everyone using this scheme when a programmer commits code that breaks the automated build and testing process. It is important to have the builds working to easier detect faulty code on changes.

The system is programmed using C++ that connects all the packages together, where Python is integrated to control higher level game functionality. The Python scripts set up the game and alter the system state during callbacks. Meanwhile, the internal application runs the main update loop for time-dependent callbacks that calculates physics properties and renders the current scene. The frequency of Python callbacks is controlled by the script using timer based requests. The script interface can also request to be notified on other updates, such as when the system starts up and shuts down and even intersection between entities.

[21] OpenAL 2009, <http://openal.org/>

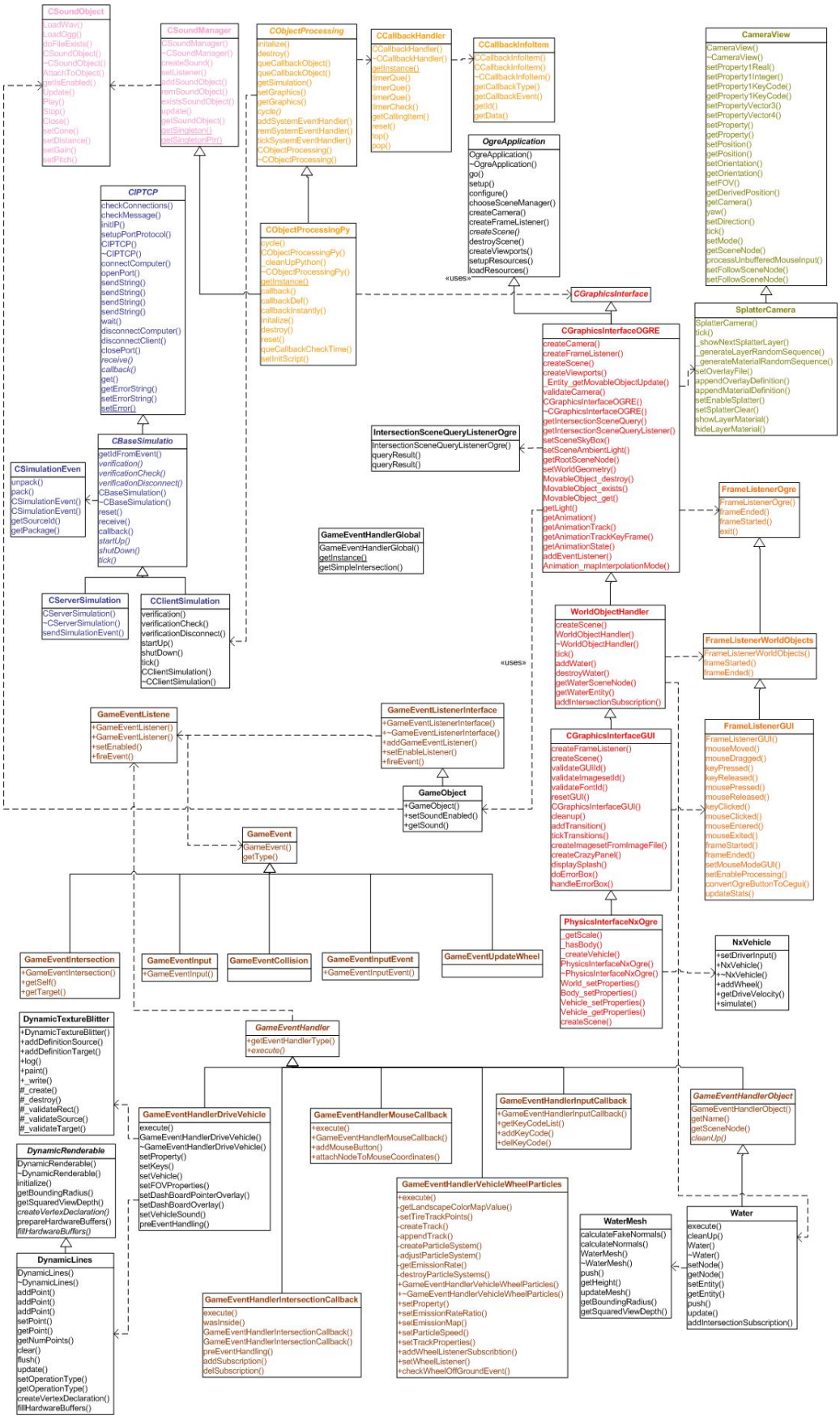


Figure 3.4 UML diagrams of inner classes

Figure 4.4 shows an overview of the system. The following sections describe the different aspects of the architecture in detail, specifically the ”red“ section; `CGraphicsInterfaceOGRE`, `WorldObjectHandler`, `CGraphicsInterfaceGUI`, `PhysicsInterfaceNxOgre`. These classes provide the core functionality of the system. A larger image is available in the appendix “Client UML”.

3.4.1 Rendering

At the core of the system, we have OGRE (Object-Oriented Graphics Rendering Engine). OGRE handles the main loop, calling frame listener methods before and after rendering. Using OGRE API you can override these classes to execute game-dependent calls. The frame listener handles the rendering of the viewport that the current camera displays. The game-dependent features are handled either before or after this view render. This is managed in an abstracted way so that every class is self-contained and focuses on its respective features. The current system supports these layers of functionality; graphics, GUI, world objects and physics. These layers handle game-dependent features. Since these different layers need to be accessed when setting up the game; every layer must be callable from Python. This is provided through the `CGraphicsInterface`. The `CGraphicsInterface` class is overridden by methods of all the classes extending the `CGraphicsInterface` that needs to be accessible from Python.

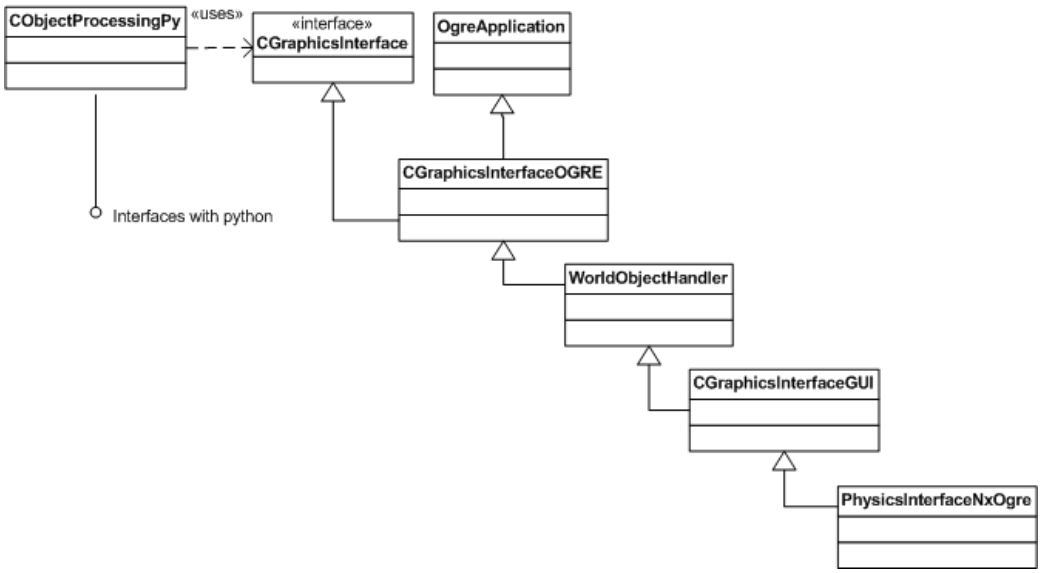


Figure 3.5 UML diagrams of layered library classes

The first layer of the extended class is `CGraphicsInterfaceOGRE`. It is responsible for graphics related calls, handling basic OGRE requirements and features. These include providing the script interface with access to OGRE creation, manipulation and destroy calls. These methods are provided by OGRE to be used for common scene objects such as entities, lights, cameras and other scene management attributes (refer to appendix “Core System Script API” for further details).

When rendering the scene, we want to display entities in our world. OGRE entities are mesh objects defined using an OGRE specific format. Every mesh object has default

material definition and can be altered during run time. The mesh format has texture, normal, tangents and animations. These are defined during mesh export from the proprietary 3D software packages used when modeling the objects.

Materials on these entities are defined using script-based files having the ”.material“ extension. This text-based file format uses the OGRE material definition format that defines the material property that is compiled by OGRE and applied to the final rendered entities on screen. This material script provides the artist with the ability to apply simple materials such as ambient light, colors and textures to more complex effects like real time vertex and fragment programs. These programs can calculate per pixel values for every rendered frame, creating advanced visual effects seen in high-end games.

Simple and advanced effects require external resources. These must be defined where they can be accessed. This is managed using OGRE resource manager. It assumes that all the resource paths are defined in ”resources.cfg“. This text-based file has all the paths to the resources required for our rendered scene. If this fails, an error from OGRE is displayed through a message box or the “Ogre.log” file. But if it succeeds, a mesh is loaded and is wrapped in an entity class. The entity class can control an entities animation, pose and materials during execution (refer to appendix “Core System Script API” for further details).

The entity class is not visible to the scene manager unless it is connected to a node. The nodes can be assembled like a tree structure connecting entities, lights and cameras where the root scene node is the root of the tree. Moving a parent node will affect all the children with the applied movement. The nodes can be assigned to animation key frames or programmatically moving them around the scene.

The scene must have light for us to be able to see anything. Lights can be created that illuminate infinitely or finitely like a sun or spotlight. The lights can have different colors but must be handled exactly as the entities so they must be attached to a scene node for them to affect the scene.

The same rule applies to the cameras. Cameras have basic movements following the same rules as entities. But they also have a field of view. Cameras must also know what area and how they should display the current view. This is fully definable through the script interface (refer to appendix “Core System Script API” for further details).

Since the camera handles how the objects are displayed and from where, it must know what it needs to display within the current view. This is handled internally by OGRE, which supports different scene managers that can be applied to calculate entity obscurity. In OGRE this is quite dynamic because different scene managers can load and unload at run time. The Octree Occlusion based scene manager is currently in use by our system to manage the visible entities.

OGRE renders using both DirectX[22] and OpenGL[23] drivers, thus abstracting driver-specific features to the OGRE programmer. The user can select the preferred driver during startup, as well as the rendering mode and resolution.

---

[22] DirectX 2009, <http://msdn.microsoft.com/en-us/directx/>

[23] OpenGL 2009, <http://www.opengl.org/>

3.4.2 World Handler

The world object handler, `WorldObjectHandler` class, is meant as a placeholder for game-specific methods. These are methods that require special handling by the game application. An example of this is the water effects, where object intersection is detected for the water patch, calculating ripple effects onto the water mesh.

This layer is not used for this distribution, but supported for game-specific objects that may be used in the future.

3.4.3 Graphical User Interface

The next layer is the GUI class layer, or specifically `CGraphicsInterfaceGUI`. It extends the world handler class. The GUI frame listener grabs input from keyboard and mouse interactions and injects it into the GUI components. If the script callback has registered that it is interested in specific keys, callbacks to the script interface will be sent, notifying the key state.

The graphical user interface window uses a class wrapper called Hikari. This open source add-on wraps a Flash OCX[24] library that is used in most flash rendering applications, such as internet browsers. The wrapper creates a GUI by loading the specified flash file into a separate thread. The synchronization of events between the core system and the GUI is managed through an update function during our main loop that draws the rendered flash content to a predefined rendering area in our graphics memory.

The `CGraphicsInterfaceGUI` class provides the game script interface with creation and destruction calls for the GUI windows. It also provides the script interface with a method that registers function names. This method collects all the registered GUI events so that when an event is triggered from the GUI component, it is grabbed in the C++ code and forwards it to the script callback method, executing game dependent code. Through this system, the Hikari wrapper can handle input, output and internal GUI events. The events are defined in the Flash editor application. Using a Flash editor application, an artist can create Flash effects, such as playing a Flash animation, loading external movies and can even calling an externally defined function. These external functions are defined in our script, so that when it calls the external function, we forward the call to the underlying C++ code that in turn calls the Python method directly.

The script interface is also provided with a "`call (…)`" method. The "`call (…)`" method can call methods defined in the Flash GUI. Its parameters are the name of the loaded file, a function name that is defined in the Flash GUI, and a string value. The specified function must exist in the flash file and must also take only one string as an argument. When the script calls this function from the script interface, we forward the call to the flash interface. The call is grabbed by the Flash GUI function that was registered using `ExternalInterfaceCall` (refer to appendix "Core System Script API" for further details). This way, any flash features can be executed by the game mechanics. The GUI programmer can convert the string value to any type of value, integer, float or other types required by the function.

[24] OCX 2009, <http://en.wikipedia.org/wiki/OLE_custom_control>

3.4.4 Physics

The physics layer, `PhysicsInterfaceNxOgre` is the last layer implemented. It is responsible for handling physics using PhysX through the NxOgre wrapper. NxOgre wraps features available in PhysX handling physical behavior of the world. Simple script interface calls are forwarded to the `PhysicsInterfaceNxOgre` (refer to appendix “Core System Script API” for further details).

Physical objects and their behavior can be simple or complex, depending on the geometries involved. A static object is used if collision proximity close to the mesh topology is needed. Dynamic objects must be wrapped in a physical primitive like boxes, spheres, or capsules. Using such simple approximations, the simulation of the topology having the entities will not be exact. For instance, wrapping a vehicle chassis in a box will result in a rollover vehicle lying flat instead of tilted against the hood as expected. This is because the topology is not calculating intersection detection close to the mesh topology, but rather the outline of the box. Instead, we get dynamic objects in our physical world that have a mass and are influenced by other dynamic objects and forces. Dynamic objects also have the ability to be connected to other dynamic objects, using joints that have specified degrees of freedom. For instance, the vehicle is composed of a box having a motor joint connected to thin spherical wheels. The boulders in our landscape are static objects that the player can drive on top of (see image 3.6).

The terrain is generated using “simple paged landscape”. This OGRE “add-on” creates many patches of meshes that are “glued” together. The patches have the ability to have different types of visual resolutions. The far patches have lower visual resolution than the patch area that the camera is closest to. The physical representations of the landscape are static objects. These objects are created out of data provided by “simple paged landscape” with high resolution. Since a large scene is displayed for the user to interact with, we have split up the world in many static patches that are close together. Using the PhysX debug tool, you can see (in image 4.6) how the world is built physically, having two patches. The patch closest to the viewer is gray and the one farther away is slightly darker, colored pink.

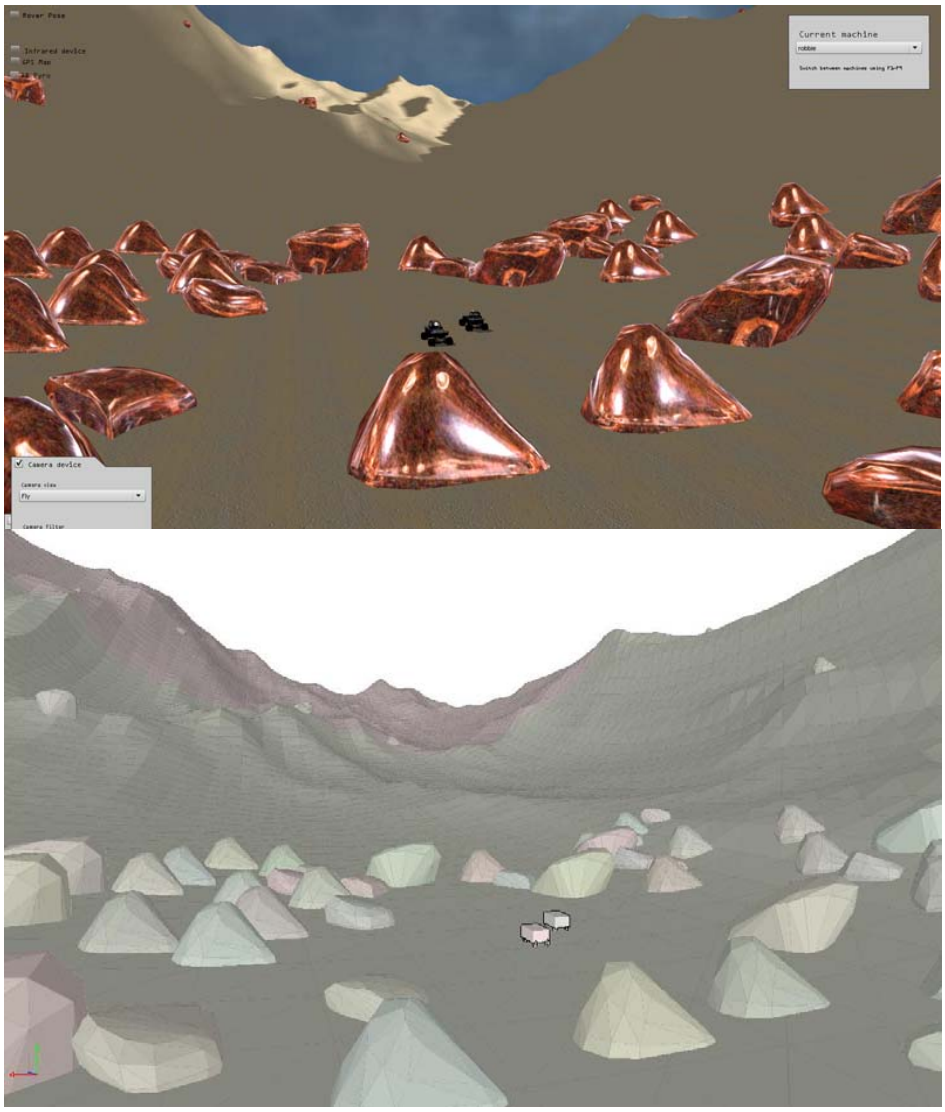


Figure 3.6 View of terrain using OGRE (above) and PhysX debugger (below)

The current implementation has two different types of vehicles, a regular vehicle and a rover. Both physical vehicles are built from four wheels. These wheels are regular ”PhysX vehicle wheels”. They are independent cylinders attached to joint motors. The motor applies a torque force when driving and a rotation force when steering. The currently implemented vehicle only applies torque force on the front wheels, making the vehicle front-wheel driven, thus behaving as a regular vehicle. The rover vehicle behaves the same way, except when breaking and rotating at the same time. Then it will rotate around its own Y axis similar to a Mars rover.

3.4.5 Python

The bridge between Python scripts and our framework is bidirectional. The game script calls defined modules registered by the system (see list below) and the simulation framework calls registered game script callback functions during runtime. This is accomplished by simple Python evaluation functions that take string as an argument and execute it or more advanced calls that register and manage methods and its variables internally.

When starting the system, the Python framework is initiated. The system runs the Python code in class `CObjectProcessingPy` which extends `CObjectProcessing` class that handles the internal game processing calls. Then the modules are registered, making them available to the game script. Here the following modules are registered:

- game
- gameGUI
- gamePhysics
- gameGraphics
- gameSound
- gameObjects
- network
- fileIO

These are all Python modules that can be imported by the game script. Each provided library forwards execution to internal system methods using "`PyObject`" that defines how we handle the call internally. "`PyMethodDef`" sets up the function's name and finally a registration call to "`Py_InitModule`" assigning a function to a module. When all of the modules have been defined we notify the Python system, where it can load the Python game scripts.

Then the first external script file, "`__init__.py`" is loaded and executed. Using this startup file, the game designer can do game-specific initialization and registration. The init file will either do registration of callbacks or import other game scripts modules that do it. These game script files must register the "base game class" called `GameBase`. This class allows the game programmer to register events that may be of interest. It will receive these events from the game system as they occur. These callback events are currently supported:

- Timer callback
- Intersection detection
- Setup
- Destroy
- Network events
- GUI events

If no callbacks are registered (during start up) no callbacks will be made until the system shuts down. Every callback from the game system is made through a registered game class that must extend `GameBase`. This extended game class can override the available methods if they are of interest by the game programmer. But for these methods to be called

back by the system framework, we must register the extended class through call `SystemHandler.registerGameClass (…)`.

When registering the extended game base class with `SystemHandler.registerGameClass (…)` the system is notified on what to call during callbacks. When registered, the system is ready to call the methods available through the `GameBase`. Then a timer callback event is triggered internally in our framework. On these events the script can get and set the system state. This can range from creating game objects with specific attributes to getting game state through XML-RPC calls.

The design of the Python modules interfacing to the system is opted for an easy-to-use script interface, mainly to reduce script calls. The interface provides the user with simple calls that take the provided arguments and do all heavy work internally. For instance, the vehicle assemblage is defined using calls that store the information that is later flushed through the system. Calling the flush method assembles the vehicle using the provided data. If data was not provided the default values are used. This eliminates potential faults when building a vehicle from the script. Because the user has less control in what order they provide the data, we just fill in the blanks if they are missing during vehicle creation. Similar assistance is provided when managing OGRE "node dependent" objects such as entities and lights. Here we automatically create and destroy scene nodes when creating and destroying these objects. But because these objects automatically get a scene node, the game designer has the freedom to manage these objects and utilize node features (refer to appendix "Core System Script API" for further details).

3.4.6 Pyro Robot

Pyro is an open source project that "provide[s] a programming environment for easily exploring advanced topics in artificial intelligence and robotics without having to worry about the low-level details of the underlying hardware."[25] The Pyro library has been in development a long time and is at a stable revision 5.0.0. It is distributed with common machine learning algorithms that are simple to use and have many online examples of implementations created by the Pyro community. The following algorithms are some of the incorporated methods within the package and are ready to be used within our framework:

- Direct/Reactive/Stateless control
- Behavior-based control
- Finite state machines
- Subsumption architectures
- Fuzzy logic
- Reinforcement learning
- Neural networks
- Feed forward networks

[25] Pyro 2009, <http://pyrorobotics.org/>

- Recurrent networks
- Evolutionary systems
- Self-Organizing Map (SOM)
- Resource Allocating Vector Quantizer (RAVQ)
- Cellular automata

The abstraction concept that is used in Pyro plays an important role within our framework. The idea is that all devices are abstracted, so that when using the device we do not bother with the specific capabilities of the devices. This simplifies the implementation of machine learning programs so the user can focus on the actual learning algorithms rather than the device capabilities and the interfaces to them.

Pyro is an environment- and platform independent interface that allows researchers to focus on AI rather than low level drivers and its interfaces. It also allows you to access these device drivers through a console in real time.

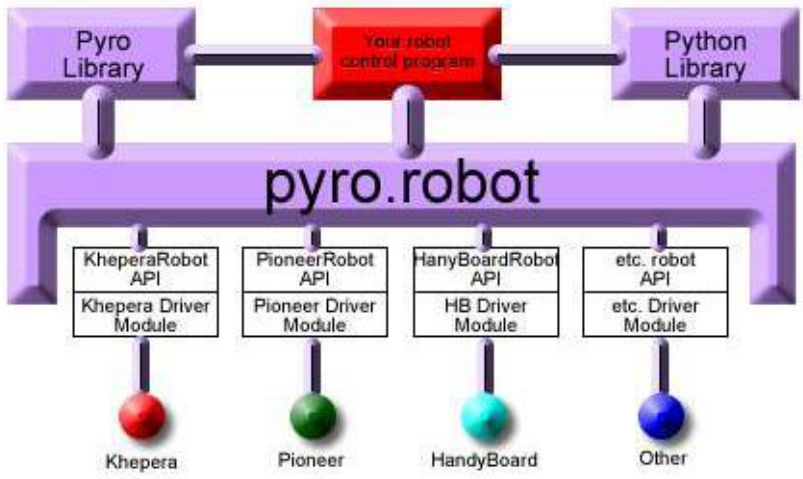


Figure 3.7 Image borrowed from paper [4] showing Pyro architecture

”A user’s control program for a robot is called a brain. Each brain is written by extending the library classes similar to the way a Java programmer writes Java programs. This allows a robot programmer to concentrate mainly on the behavior-level details of the robot. Since the control program is written in Python, the standard Python Library is also available for use” [4]

The concept of the abstracted robot was easy to generalize and implement into the game client. To control a robot in the environment the robot is viewed as a machine. The machine has devices and is controlled by a robot brain. This was implemented so that all devices have a hardware and software representation. The hardware is the physical device representation and has an entity or physical object representing it, while the software driver is the code calling device features. In other words; the driver has access to the device properties and is registered by the robot while the device class interacts with the game system. The driver must extend the `pyrobot.robot.device` while the Device

class is the code that initiates it during start-up, and then the extended `pyrobot.robot.device` device class does the device registration internally within Pyro framework.

Adding and removing devices is done through the state web server. When the machine is loaded during client initialization, the game client retrieves the user's machine setup and then the robot takes control of these devices loading and registering them accordingly into Pyro framework.

After this step, each device is known to the robot and can be controllable by a brain. This means that when a vehicle device is controlled by the brain, the driver will actually apply the torque force. But the device is the one that forwards it to the internal simulated physical representation. The brain (also called the agent) interfaces with the machine driver through a simpler interface, while the device handles the simulation by calling the internal system framework.

Each device class has a GUI, allowing the user to get visual feedback on the status of the device. Depending on the device, different GUI windows are used. Some GUIs control the device driver directly through buttons, text boxes or combo boxes. Other device GUIs display the device status and current data through text or Flash animation. This displayed information comes from the driver that in turn is updated by the device.

Device updates come from the game system and are handled differently, depending on the device requirements. Some drivers may need to query the game framework directly. Others gets updated automatically when the framework state changes. This happens when the internal game framework state changes on intersection detection, as is the case for IR (infrared sensors). On intersection detection, the system calls back (call-back) and enumerates all the machines and its devices checking if we need to update its data. For instance, getting distance to other objects using the IR devices will enumerate through all the machines and its devices, checking if the entities belong to an IR device. This is done every time there is an intersection detection detected through `MachineRace.intersectionCallback (…).` Then, intersections are added (or removed) for the device that the intersecting entity belongs to. Now the IR device knows that one of its sensors is intersecting with another object in the world. But since it is up to the Brain to choose what devices it needs to use to learn about its environment, we do not need to get the exact status of the current intersections unless the Brain uses the device and queries the state of the device intersection (see next section "Device" for more details).

Figure 4.8 displays a Unified Modeling Language (UML) diagram of the interactions between Pyro-dependent classes. Not all of the classes used by the script system are displayed, for instance omitting specific drivers like IR, radio and rover that are extended by `Device.Driver`. The diagram does not show how the brain class is used when programming the agent because it is extended directly from the Pyro internal system.

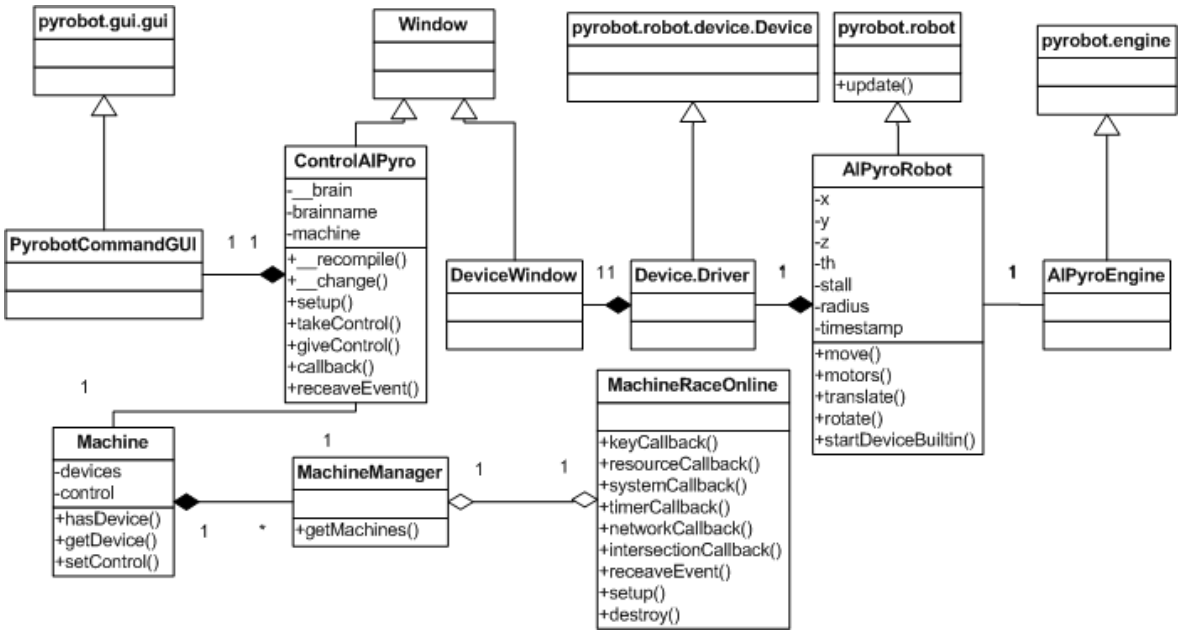


Figure 3.8 UML diagram overview of Pyrorobot dependent classes

Figure 4.8 shows an overview of how Pyrorobot was implemented and what classes it depends on. Our main game class is `MachineRaceOnline`. It extends the `GameBase` class and receives all notifications from the underlying framework. It has access to `MachineManager`, which contains all of the machines simulated in the world. Each machine has many devices and can be controlled through the control instance.

The Pyro implementation uses the class `ControlAIPyro` to control the machine that in turn is controlled by the brain. This class can give and take control of different user-defined brains during run time as well as recompiling and reloading the current brain. The `ControlAIPyro` also has access to `PyrobotCommandGUI`, which handles console commands and allows us to control the machine directly through a command prompt.

When starting the game, the machines are created and populated into the `MachineManager`. During runtime, the user requests a timer-based callback every 100 ms to update the machine brains. This is done through the `MachineRaceOnline.timerCallback (…)` method that enumerates all the control classes calling `ControlAIPyro.callback (…)`. This method will in turn call `brain.isAlive (…)` and `brain.run (…)`, originally handled using threads in the generic Pyro implementation. By calling `brain.run (…)` ourselves we simulate the threading execution using a single threaded approach.

When `brain.run (…)` is called, it will do internal Pyro management and subsequently forward the call to `pyrorobot.brain.Brain.step (…)`. This method is overridden by our own brain implementations. In this call, the AI programmer queries machine devices and forwards values to ML algorithms that in turn apply actions to devices that propagate values to the simulated machine.

These devices are accessed by the brain to control the simulated machine. This is done by accessing the robot variables and methods. When doing this, you are actually accessing `AIPyroRobot`, which extends `pyrorobot.robot`. It has basic functionality such as `move (…)`, `motors (…)` and `rotate (…)` and even *x*, *y*, *z*, *th*, *thr* variables, notifying the vehicle's current position and tilt.

Implementing the Pyro framework into our simulation environment was challenging. The Pyro library does not have any available documentation on implementing the library into an already existing framework. But there are different ways to implement the Pyro, as seen in the Pyro robot directory. Out of the box, the Pyro library internals rely on Python to be multi-threaded and in control of the main loop. The threads run the updates to the brains and devices synchronously, calling sleep methods and timer methods to align time-dependent data.

Integrating Pyro into the framework involved altering core Pyro modules that used timer and threading functions, such as "Brain.py". These modules had to be changed from multithreaded to single threading because the timer and threading calls resulted in the entire system freezing and halting execution due to lack of multithreading support in our Python integration. This required us to make modifications to the Pyro library when integrating it to the distributed Python library that comes with the game. The reason for this is that the main game loop is executed through our core framework rather than by Python. This results in our system giving Python pieces of the execution time through callback methods. When these calls have completed, our game loop continues until the next time a callback is requested, resulting in our Pyro implementation being asynchronous. This required us to remove all the code referring to any thread control.

Originally, Pyro started a brain in a separate thread, updating the brain state through the threading method "`Brain.run (…)`". To adapt the synchronization, our system will call the "`Brain.run (…)`" manually for each running brain that is normally executed by the thread automatically. If `Brain.isAlive (…)` returns "`True`" we call `Brain.run (…)` in our timer callback method. This way, the system "simulates" the threading sequentially for every brain that is running.

The Pyro library is large and required different kinds of implementations of solutions using threads. All these threads must be simulated asynchronous so the system would behave according to the original distribution. Testing implementation integrity could not be automated because the Pyro framework does not include generic tests.

To make sure that the integration was close to the original implementation, the examples provided with the library were tested. This was done by running the examples (after minor alterations) and observing the learned behavior. This kind of testing is a high level of testing that is dependent on individual device implementations. The results, however, were positive because the example brains were learning in a similar way to the original implementation. The following brains were tested:

- `Subsumption`: The behavior is to randomly orient itself in its environment, avoiding obstacles. It matched that behavior perfectly.
- `FSMProgram`: The vehicle drives around in a boxlike shape. This did not match the implementation because the robot did not know how far it should drive before stopping and rotating itself 90 degrees.
- `NNOnlineProgram`: Drives in an environment with many obstacles. It uses the IR sensor to detect collisions, forwarding it to an Artificial Neural Network (ANN). The ANN outputs learned values that converged to the target values.
- `RLBrain`: Has a grid with pits and closed areas. It walks this grid randomly, searching for the goal state. When the goal state is found, it learns the way and

Since the Pyro ideology is based on the assumption that devices are abstracted and that no device implementations are the same,we can conclude that these results are sufficient as a good implementation and further testing must be done.

As far as testing the devices, they should behave the same between brains. So if the device works for one implementation, it should be working for all.

3.4.7 Devices

Devices are the essential building blocks of a robot. Implementing devices is done by the user on the server side through the web interface. The machine definition is loaded by the game client applying it to the current machine. Each device has an entity representation and a device driver. These are interfaced by the robot and accessed by the brain upon request. Below are the devices that are currently supported:

- Rover; a four wheeled vehicle
- IR; infrared device
- Camera; controls the scene camera
- Radio; can place the machine in a new location

Each device implementation has an accompanied test class to test the device internals. These tests extend the Python UnitTest framework. A typical device module has this class structure:

```
class Device :
    def __init__(self, entity) :
        self.entity = entity
        ... device initialization code here ...
    def setVisible (self, b) :
        """This is called when we are un-selecting and selecting the machine"""
    def setDriver (self, d) :
        """ Called by the Driver when it was created """
        self.__driver = d
    def getGUI (self) : return self.__driver.getGUI()
    ... some helper methods here ...
    class Driver(pyrobot.robot.device.Device):
        def __init__(self, dev, type = "... device name here ..."):
            pyrobot.robot.device.Device.__init__(self, type)
            self._dev = dev
            self.__gui_instance = DeviceWindow()
            ... driver initalization code here ...
        def setVisible (self, b) :
```

```
            if b : self.gui = self.__gui_instance
            else : self.gui = None
        def getGUI (self) : return self.__gui_instance
        ... some helper driver methods here ...

from window import Window

class DeviceWindow (Window) :
    """Handles menu"""
    def __init__ (self):
        try :
            GUI.systemGUI.loadLayout ("... the gui flash file ...", 256, 256)
            GUI.systemGUI.setPosition ("... the gui flash file ...", 0, 0)
        except Exception, e: pass
    def clear (self): GUI.systemGUI.delLayout ("... the gui flash file ...")
    def callback (self, source, event, message):
        pass
        ... gui callbacks that is forwarded to a driver or device here ...

import unittest
class TestDevice(unittest.TestCase):
    ... device specific tests here ...
```

Every device has the same or similar code structure as the code above. This is so that the programmer can generalize the calls to the device as needed. The defined methods are used by the system internals when the user is interacting with the different machines.

Using these structure, different types of devices can be implemented, ranging from sensor devices to tool devices such as magnets.

Implementing Device

In what follows we show an example of how to implement a sensor device. The sensor device is the most basic device a robot can have. It helps the robot orient itself in its environment. An infrared device (IR) has infrared sensors. The infrared sensors can calculate the distance to objects within a specified range. Our implementation of the infrared device is such that an entity is created. This entity is applied to the vehicle as a child node of the vehicle. So wherever the vehicle is located, the entity will be at the same location having the same orientation. To this entity, hidden boxes are applied. These boxes are our actual sensors, sensing the amount of intersection by other objects. They orient around the center entity so that they “reach” out.

When creating the machine, hidden boxes are registered to notify the device when they intersect other entities. On intersection, the device gets a callback that an intersection has been detected. When the device knows that an entity is intersecting with another entity, it can query the system using call `getDistanceTo (...)`. This method returns exactly how much of our object is intersecting.

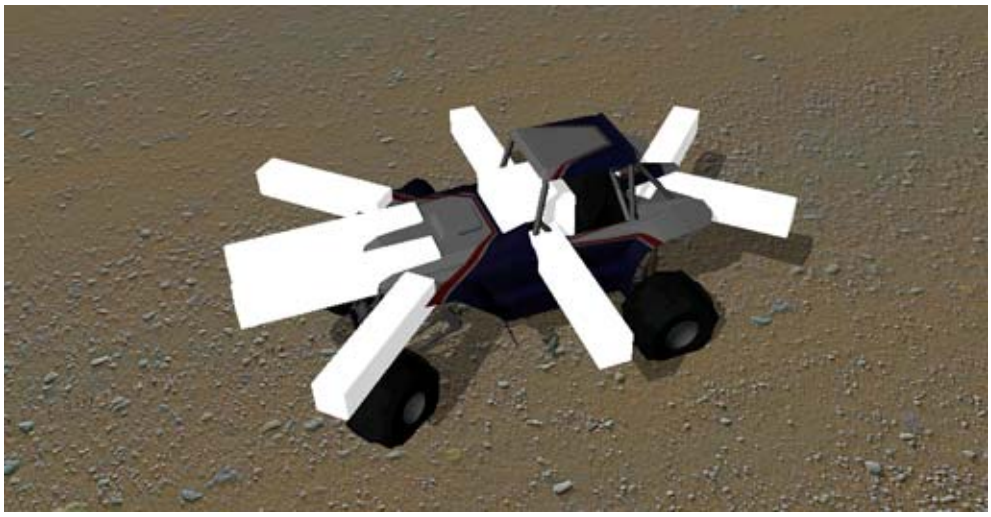


Figure 3.9 Vehicle and visible sensors

Bellow is the code used when calculating distance to intersecting obstacles by the IR sensors. This code is called by the Python code where “self.instance” is a call to vehicle interface:

```
def getSensorValue(self, pos):
    return pyrobot.robot.device.SensorValue(self, self._getVal(pos), pos,
                      (self._ox(pos) / 10.0, # meters scale adjustment
                       self._oy(pos) / 10.0, # meters scale adjustment
                       20.0 / 10.0, # meters scale adjustment
                       self._thr(pos),
                       self.arc),)

def _getVal (self, pos):
    mindist = self.maxvalueraw #assume that no object is near
    if self.__sensored_cd == None : return mindist #no update has been made
    sensors = self.instance.getChildren()
    instance = SceneManager.getEntity(sensors[pos])
    #now we just return the minimum distance
    for e in self.__sensored_cd[pos] :
        p = e.getPosition()
        m = instance.getDistanceTo(e.name)
        if m['distance'] <> None :
            dist = m['distance']
            mindist = min(dist, mindist)
    return mindist
```

All the sensor states are checked when the AI programmer queries the sensor state. The following code is called by `instance.getDistanceTo (e.anem),` where `getDistanceTo` will look up movable object A (moa) and movable object B (mob, or instance) is this particular sensor.

```
Vector3 v;
 AxisAlignedBox aaa;
 AxisAlignedBox aab;
 if (moa && mob)
 {
     result = result + "{";
     aaa = moa->getParentSceneNode()->_getWorldAABB();
     aab = mob->getParentSceneNode()->_getWorldAABB();
     AxisAlignedBox aai = aaa.intersection(aab);
     if (aai.getSize().length() == 0)
     {
         result = "{'coverage':None, 'center':None, 'distance':None}";
         return true;
     }
     v = aai.getSize() / aaa.getSize(); // - sceneNode->getWorldPosition();
     result = result + "'coverage': [" + v.x + ", " +  v.y + ", " + v.z + "]";
     result = result + ",";
     v = aai.getCenter() - moa->getParentSceneNode()->getWorldPosition();
     result = result + "'distance': " + v.length();
     result = result + ",";
     v = aai.getCenter();
     result = result + "'center': [" + v.x + ", " +  v.y + ", " + v.z + "]";
     result = result + "}";
 } else return false; //One of the objects did not exist
 return true;
```

Calling method `getSensorValue(...)` we get the following data:

{'distance': 0.7310, 'center': [-423.31, 116.0280, -602.67899], 'coverage': [0.226189, 1, 0.7235]}

The returned data is evaluated by `_getVal ()`. This dictionary string has detailed information regarding the intersection state of one sensor. Here, the value of “distance” is the percentage of the coverage detected by the internal system. “Center” is the center location of the intersection in world space and “coverage” is the percentage of coverage for each axis, *x*, *y* and *z*.

3.5 Code Control

The development of a simulation framework with internal and external dependencies, requires an organized and concise development environment, especially when some of the dependencies are on external hardware and services such as graphic cards and online servers.

To simplify this, automation of testing processes and hosting services is used. The internal development of the source code is managed structurally, using enterprise development procedures and workflows. The source is stored in the Subversion version control system.

During development, the programmer does local tests and commits when all tests pass successfully. Then a test server will download the recently committed code and recompile the project in its entirety. When the build is complete, automated tests check that every unit test works accordingly. Then we take this build and create an install package when needed. For details on how this works, refer to appendix “Code Control”.

3.6 Summary

This chapter described how the system technology was assembled,what different parts it is composed of and the communication architecture between the client, state server, site server and world server and their particular functionality within the framework.

The client section described the layered functionality of the game client, ending with Python- and Pyro-centric information. The section also covered how the system executes and controls the machine devices using Pyro, using the IR device as an example.

The development dependencies of the framework and the control of its source code were briefly covered in the section “Code Control”.

4 SYSTEM FUNCTIONALITY

This chapter will describe how a user uses the system to build his or her own machine and control it using Pyro.

The following chapters will show what you need to do to get started using the application, and then briefly show how the user prepares a machine, using the online pages.

Then the Pyro ideas used within our simulation is described. This section uses reinforcement learning concepts for reference, ending with how “brains” are developed using the in-game console.

The chapter ends by showing examples of such “brains” and what they can do.

Throughout this chapter the term “machine-race-directory” will be the machine race client directory where the user installed the client and the “user-script-directory” is the machine race directory where the client stores his/hers own scripts and default location is <machine-race-directory>” \scripts\user\”.

4.1 Preparation

This project is about having developed a game platform within open source projects. It is deployable on Windows-based PCs operating on Vista and XP systems. This framework uses C++ and Python as main programming languages, with Python controlling the game logic. The framework runs on systems with graphics hardware with the latest drivers. The current system supports the major 3D card manufacturers

- NVIDIA: Geforce2 or higher required, Geforce 4(non-mx) or higher recommended
- ATI: Radeon 7500 or higher required, Radeon 9600 or higher recommended

However the graphics cards SiS, Intel and S3 cards may or may not be supported or display unexpected results.

To use the system, a user must first register and install the game client. After a successful installation, we prepare the user profile to play around with the machines. Before the user can control a machine, he/she must first create and tune them. This is done through the online interfaces and is described in the sections “Create Machines” and “Purchase Devices”. “Create Machines” covers how a player can create a machine from scratch and “Purchase Devices” describes how a device can be bought and sold, while at the same time keeping track of our economy.

4.1.1 Creating Machines

A machine is defined as a collection of devices. You can choose what devices you want your machine to have in “Private Account”. The “Private Account” area is a collection of

pages showing your current game profile and state. This profile can be altered by the user regarding

- machines
- machine setup
- inventory
- economy

When you first register the online state server creates a default profile. This profile has 100 ISK as startup money and no machines (see image below).

Figure 4.1 Site machine list page

To create a machine, you must first purchase devices to assign the machine blueprint. This is done on the “Purchase” page. This page allows you to buy new or used (old) devices when building the machine.

This section will show an example that creates a new rover device from a blueprint. This is done by selecting a rover device called “4x2 Terrain Rover x1.2” as displayed in figure 5.2.

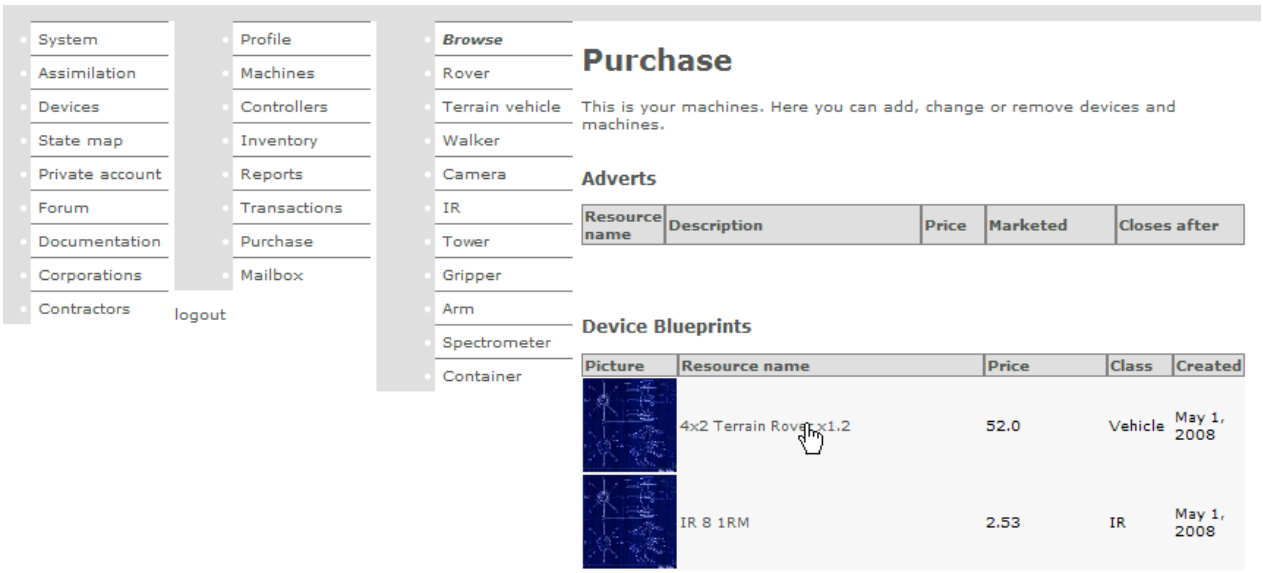


Figure 4.2 Site purchase list page

When selecting this link a new page is displayed, see figure 4.3. This page shows detailed information about the selected device displaying name, age, price and what type it is. Here the user can select “Order blueprint device”. In figure 4.3 we see that the owner “akil” has money to buy the device. If he did not have enough money, the button would be disabled.

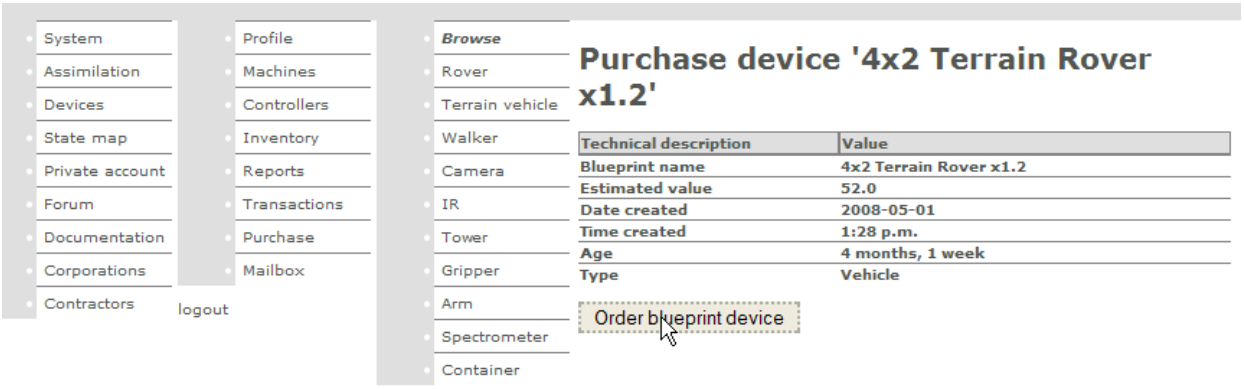


Figure 4.3 Site purchase detail page

Ordering a blueprint device will create and add a new device to “akil’s” resources. Doing so will bill your game account and add the resource to your inventory list.

AI scripts need to use sensors to orient themselves in the environment. Since we are creating a machine that will use AI features, we must add an infrared sensor. This requires us to purchase a sensor device as well. We would need to buy the “IR 8 1RM” sensor device, so we do the same steps for “IR 8 1RM” as we did with “4x2 Terrain Rover x1.2”.

When we have acquired a sensor and a vehicle device we are ready to assemble the actual machine. This is done by going to the machine page and selecting “add”. This will display the machine profile page.

On this page (figure 4.4) we can give our machne a name, “akiller”, and select the “Base” device to be “4x2 Terrain Rover x1.2” with “Body” “IR 8 1RM” device.

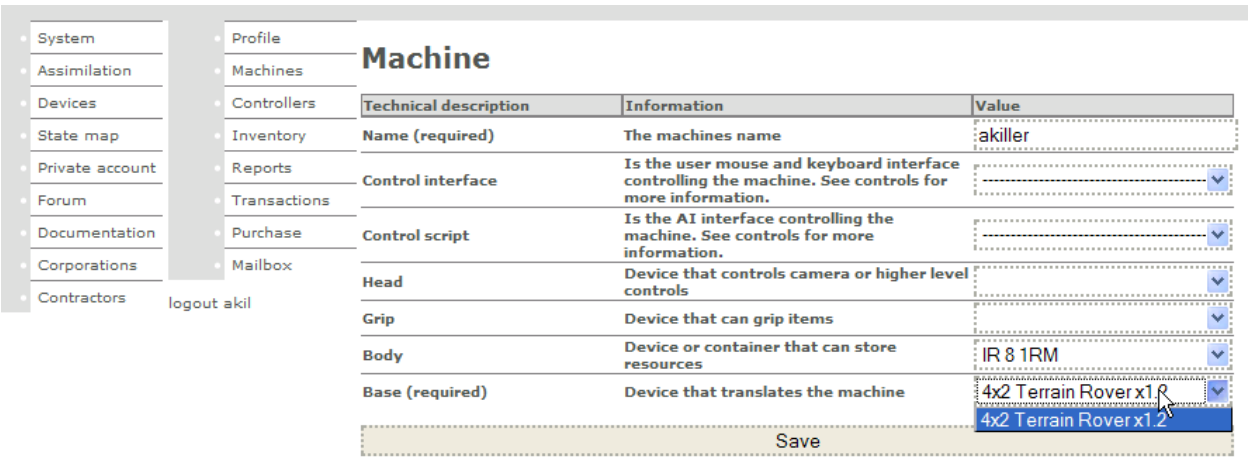


Figure 4.4 Site machine builder page

Now you have created a machine and you can start the game client application. The machine will be loaded into the environment withthe default brain, “UserBrain”, which you can change throughout the game.

You can always edit or remove a created machine by selecting the machine from the “machine list”. When removing the blueprint (by selecting “delete”), we only remove the machine’s blueprint but not the attached devices in the resource list.

Because devices are used to build a machine, the machine in itself is a collection of devices. A user can exchange these devices in his machine. These changes are applied the next time he starts the game application.

A user can create at most twelve machines. The machines will be loaded into the scene at the same time, allowing the user to control them at any time while playing. The user can switch between them, using a drop-down box or pressing keyboard keys “F1” to “F12”.

4.1.2 Purchasing Devices

A user can upgrade or downgrade machine devices through the “machine” page (figure 4.4) by simply selecting what device to edit. All of the user devices are displayed in the inventory list (see image 4.5) and these are the devices that can be assigned to machines. Selecting the machine device, you can only select from a list of devices that are of the same type you are editing. Every device can be edited, except for the base device. It is only changeable when creating a new machine.

When creating a new machine, we can choose from a list of base devices. Every base device that we can choose from is available in the “Base” list from the machine profile.

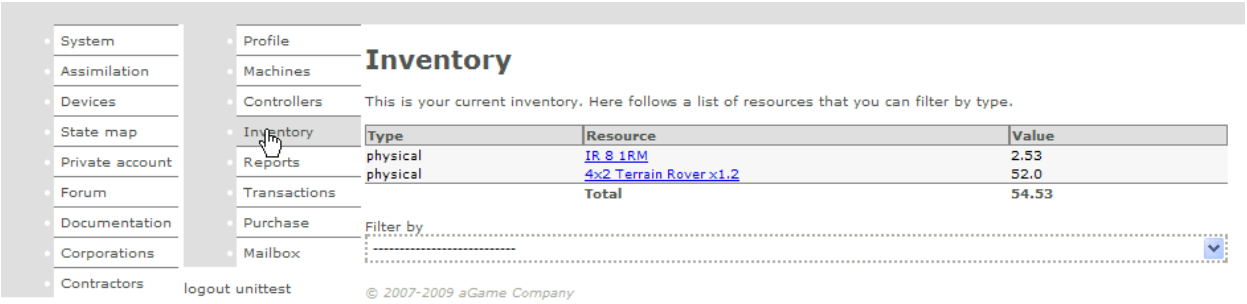


Figure 4.5 Site inventory page

To add devices to your machine, you must purchase the device. This can be done through the “purchase” page. On this page you can buy new and old devices. The new devices are available by creating a new device from a blueprint. This is accomplished by selecting a device blueprint from the selection of devices and press “Order blueprint device”.

You can sell any device that you own by advertising it. This is done by selecting a device in your inventory list and choosing “Sell item”. This will open up a text box and require you to select a valid date when the offer expires.

Every business transaction is tracked through a double-entry bookkeeping system. This transaction list is available through the “Transaction” page. Through this page you can see all of your transactions accumulated over time.

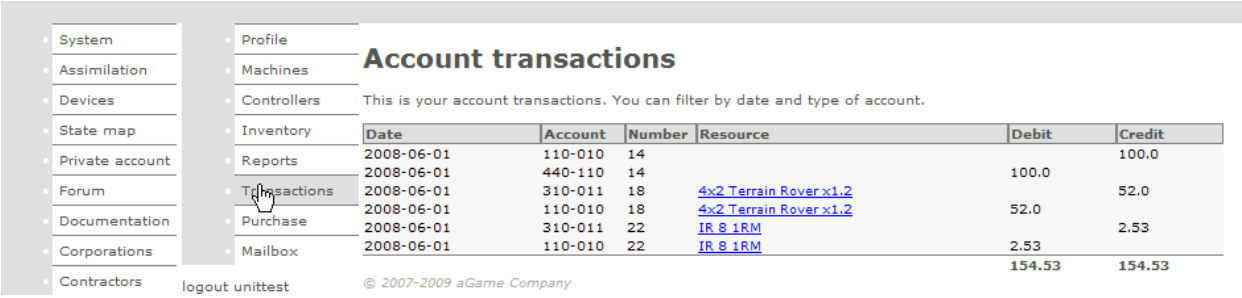


Figure 4.6 Site account transactions page

Every business transaction is tracked through basic bookkeeping rules. Keeping track of transactions allows you to view a resources history as well as giving the game known methods to evaluate the world economy.

4.2 Using Pyro

This section will introduce how you can get started with Pyro within the Machine Race framework. The focus of this chapter is primarily on how you use Pyro and its concepts within our environment. For a comprehensive Pyro documentation refer to

- Official Pyro Robot page[26]
- The Pyro community wiki[27]

Pyro is a Python library that introduces abstracted device concepts to control and manipulate robot devices. It also has some common AI algorithms implemented that can be used out of the box. It is used by researchers, teachers and students applying artificial intelligence concepts and has won awards within the AI community for ease of use.

Pyro works like this; In a text-based Python file you can define a “brain”. The “brain” extends a Pyro-based `Brain` class that controls a machine. This “brain” can combine and use behavior-based algorithms known to the AI community. Currently, we have implemented and tested the following algorithms:

- Direct control
- Sequence control
- Reinforcement learning
- Neural networks

[26] Pyro 2009, <http://pyrorobotics.org/>

[27] Pyro wiki 2009, <http://emergent.brynmawr.edu/emergent>

A user can easily implement these concepts by importing libraries that are currently supported by the Pyro library. These concepts can be tested in our framework by either creating or editing an existing brain. The brain files are in the directory

machine-race-directory/script/user/

Here you find examples that extend the Pyro brain. Each file represents a separate brain behavior. Only one brain can be used by the machine at any given time.

The brain requires sensors for its input so a machine must have a sensor installed. You can easily install them by logging in to your account and adding one. See section “Creating Machines” for further details.

4.2.1 Agent

To describe how the interaction between the machine and its environment works, concepts from reinforcement learning (RL) will be used.

The idea of an agent and its environment is a known concept within the Machine Learning community. “The learner and decision maker is called the agent. The thing it interacts with, comprising everything outside the agent, is called the environment. These interact continually, the agent selecting actions and the environment responding to those actions and presenting new situations to the agent. The environment also gives rise to rewards”[6]. Specifically, the agent in our framework is the Machine Learning algorithm. It is used by the brain to learn and make decisions. This could be an ANN network, an RL agent or even a decision tree algorithm. Figure 5.7 shows how the brain scripts work as an intermediate script. It is primarily used by the AI programmer to assemble the ML algorithms, formatting input and output values between the devices and the algorithms.

The brain script reads the state *S* from the environment, using the machine devices. Then it forwards an altered state, *S** that fits the algorithms input to the agent. In turn, the agent outputs an action *A*, and forwards a fitting action to the machine *A**. The reward *R** is given when the brain script recognizes that the state *S* from the environment is behaving according to the brain’s goals.

The brain is the actual glue that manages the data between the actual learner and the environment.

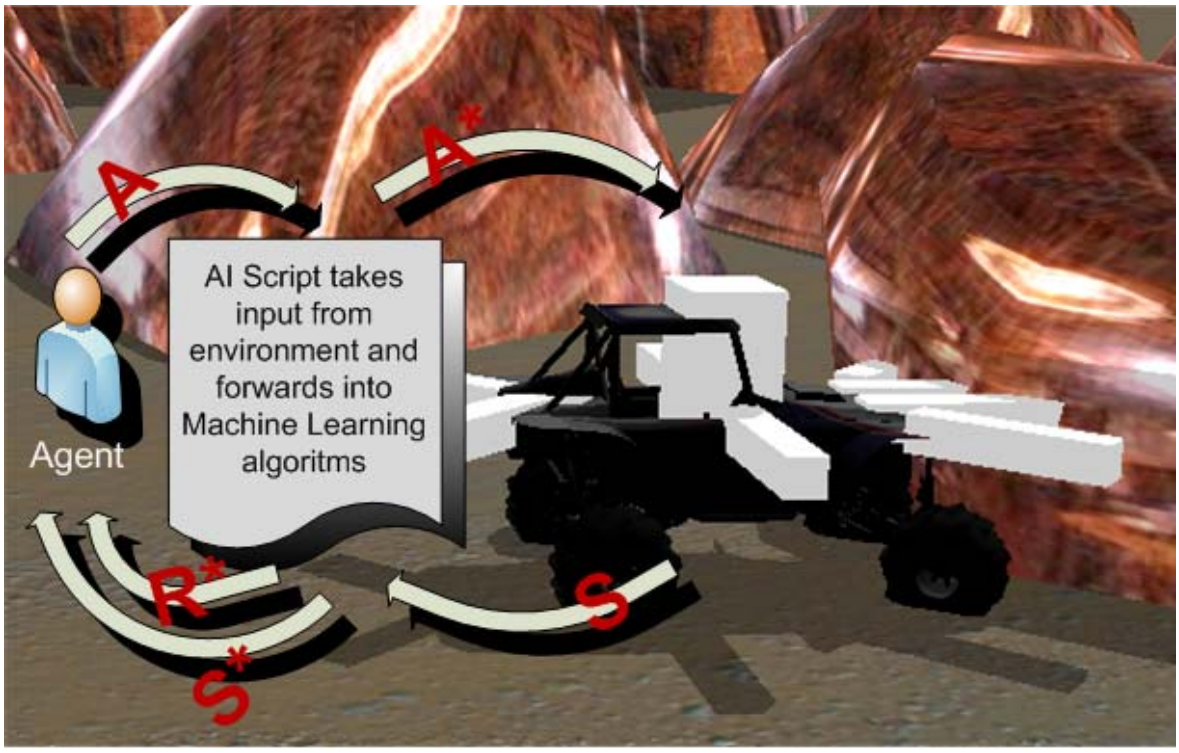


Figure 4.7 Agent and environment state interaction

4.2.2 State

Learning algorithms requires data to learn from its environment. Each machine has a collection of devices. These devices control the machine and can read data from its environment, currently an “8 IR Sensor” is used to sense distances to obstacles. The”8 IR Sensor” is a sensor with eight sensors located around its center, as depicted in figure 4.8.

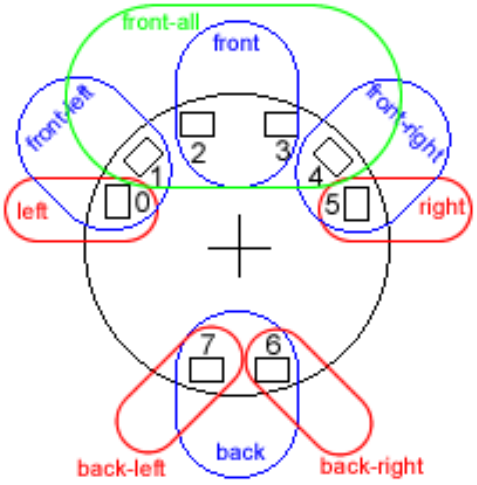


Figure 4.8 Sensor layout taken from emergent.brynmawr.edu

This basic sensor is modeled from “Khepera IR/Light Sensor Groups”. Figure 4.8 shows a sensor,”8 IR Sensor“, from above. The sensors are located around the machine’s center. Each group of sensors has an identifier (name) that you can use to access the state of each

sensor at any given time. When the sensor detects an object, it stores the distance to that object. This is accomplished by each sensor doing a call-back notification to the robot device when an intersection is detected. The intersection detection algorithm uses Axis-Aligned-Bounding-Boxes (AABB) as implemented by OGRE3D library. The intersection detection is not as exact as other intersection detection algorithms because it uses AABB. AABB boundary is the smallest bounding box dependent on the entity (mesh) topology and orientation. this results in narrow long objects having a boundary that can change from a long box to a squared box, as shown in figure 4.9.

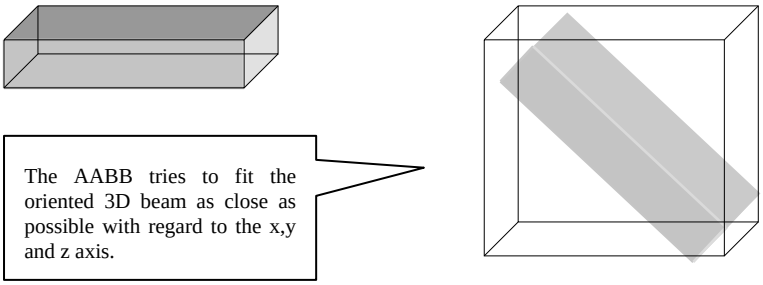


Figure 4.9 Two differently oriented beams and their AABB

On intersection detection, the center of the intersection is calculated for the bounding box A (which is the machine's "sensor box") and B (the obstacle). The new intersection volume C has a center that is used to calculate how much it is overlapping the sensor. The bigger the C box is, the more a sensor is "touching" B. The sensor returns a value between "0" and "1" where "1" means that A is fully overlapping B. This will be covered in more detail in the section "Devices". For examples on using common sensors, see the official Pyro "module contents" page[28] [29]. For more devices, see the official Machine Race[30].

4.2.3 Environment

The environment in which our machine is simulated is very dynamic because it is built using a physical library called PhysX. The world is created with a static and uneven ground that has primitive dynamic objects like rocks in it. The ground forms hills and canyons along the terrain that our machines interact with.

[28] Pyro module contents 2009, <http://Pyrorobotics.org/?page=PyroModulesContents>

[29] Pyro sensors 2009, <http://emergent.brynmawr.edu/emergent/Pyro_20Sensors>

[30] Machine Race upgrading 2009, <http://state.machinerace.com/purchase/>

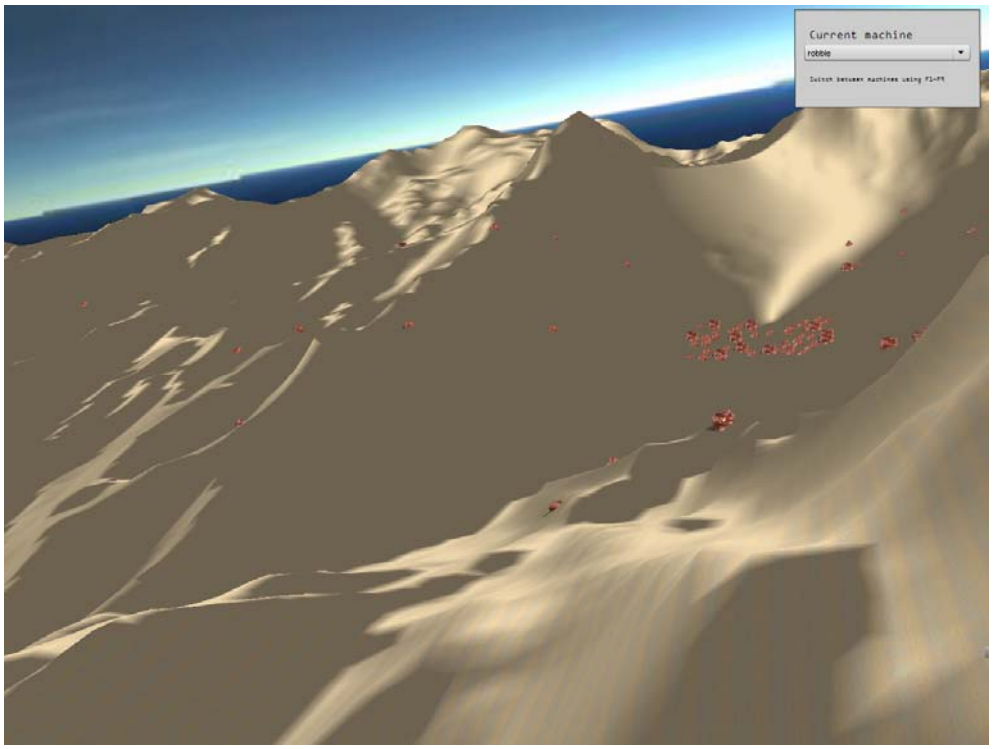


Figure 4.10 Overview of terrain

On the terrain we have dynamic and static objects. These can range from being dynamic machines to static landscape objects like rocks, boulders and stones. The only variable that is different amongst items in the world is the topology and mass which is dependent on the size and density of the object.

All these dynamic objects have an Axis Aligned Bounding Box that is the smallest box around a mesh. The sensor devices used by “8 IR Sensor” can be used to detect intersection with these objects, returning how close they are to the sensors.

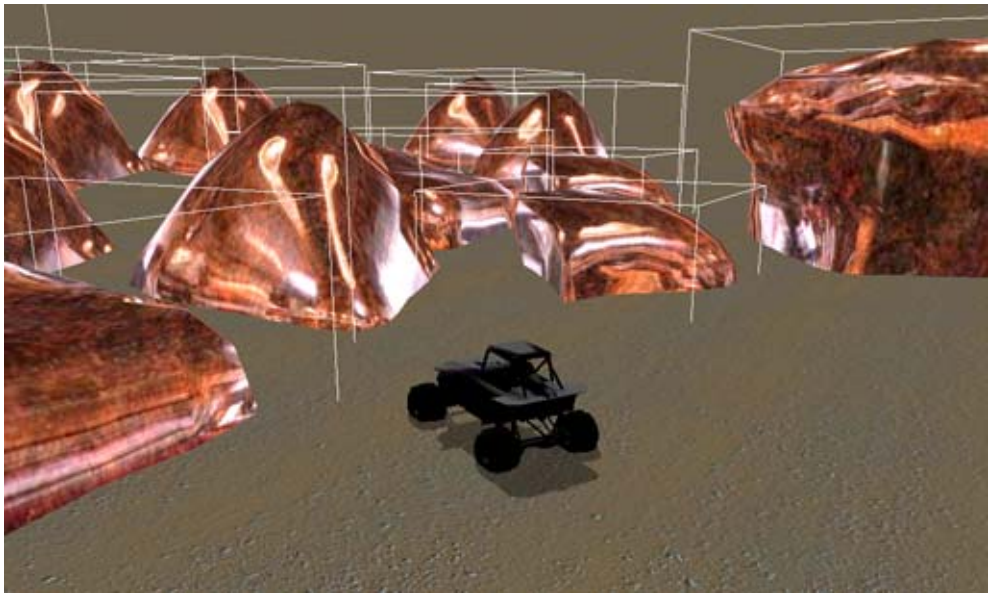


Figure 4.11 Image of bounding box

4.2.4 Output

A machine can have many devices. Each device can be controlled differently, dependending on its functionality. Currently, you can control all the devices as defined by the Pyro robot interface. You can even control a robot's position and orientation by controlling the base device. This is done through the following function calls:

- `move (amount, direction)`; moves vehicle and rotates wheels
- `translate (amount)`; moves the vehicle forward or backward
- `motors (amount, amount)`; gives power to each wheel

Where amount is a value between "-1" and "1" where "1" means that we move the vehicle full throttle and "0" stops it, while "-1" backs the vehicle up. The actual speed is dependent on the machine engine's power.

The wheels' steering direction is a value between "-1" and "1" where "1" is the maximum rotation angle to the left and "0" means that the wheels should be at the center.

When an AI programmer is working with a robot, he/she needs to know if it is stuck. A robot is stuck if it is trying to move but can not do so. Then it is considered stalled. The system assumes that a robot is stuck when the brain tries to move the robot hundred calls without being able to change its (*x*, *y*, *z*) position. Then the system changes the state "`stall`" from "`false`" to "`true`". This state is accessible by the programming interface through the `robot.isStall (…)` method and is used to detect when to correct the machine's position.

4.2.5 Interface

Each machine has its own state. This state can be viewed and controlled through a GUI window. The window, called "AI Pyro", displays the state of all of the devices of the selected machine. Through this window you can query a device state, reload brain, change script, and alter device state in real time. This interface is a window on top of your "workspace", displaying the simulation behind it. It can be opened and closed through the check-box in the top left corner. You can even drag it around by pressing the right mouse button and moving the mouse. To interact with the window components, you must use the left mouse button like in regular applications.

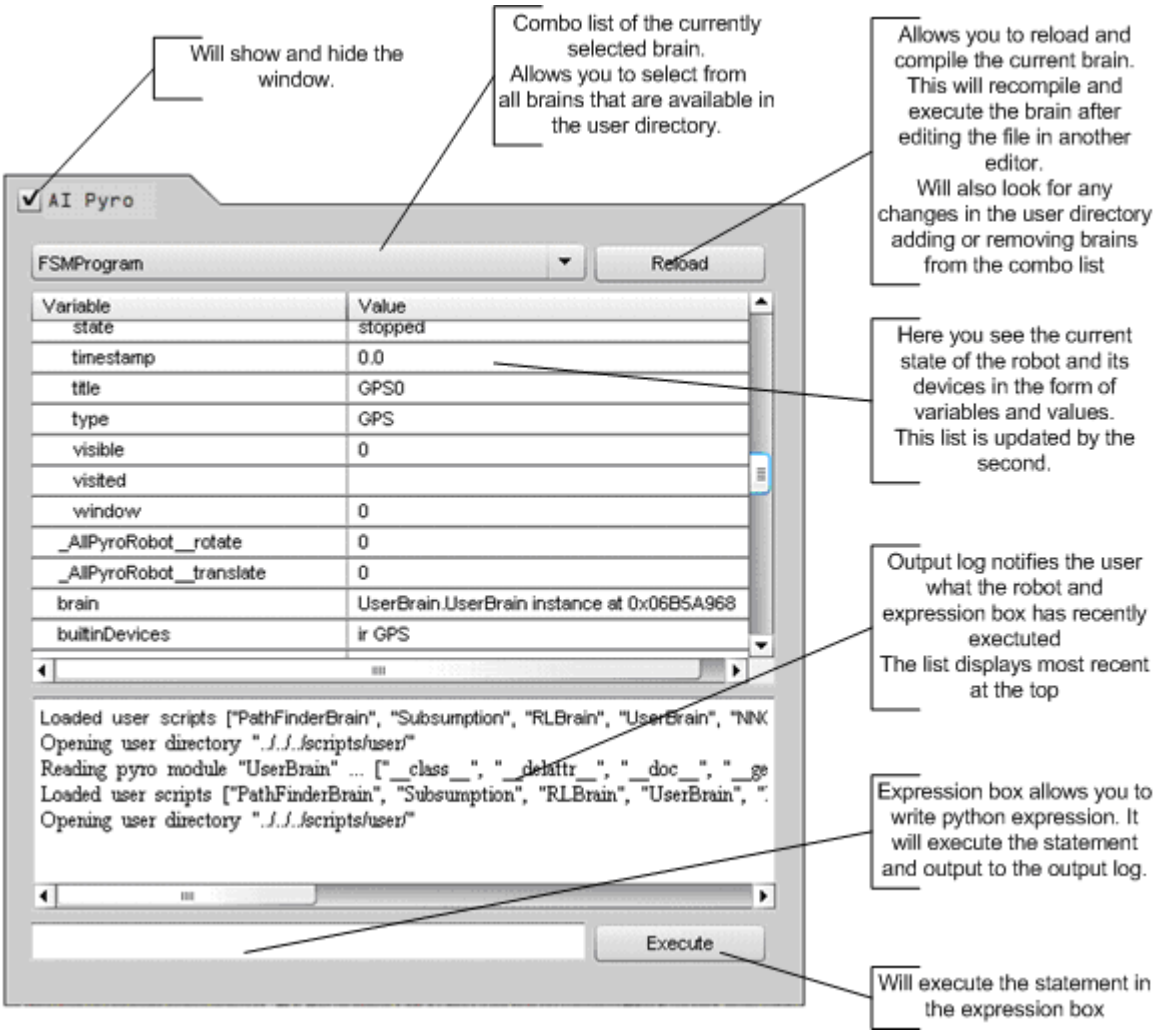


Figure 4.12 Pyro GUI

The GUI allows you to see the current state of all the devices and its variables in run time through the “variable watcher”. It displays a list of all the variables the robot has access to and updates the variables automatically every second. Through this list, you can get the machine’s current position, its tilt and even and exact state of every device in use. Below the “variable watcher” we have a text area. This text area is the “output log”.

The “output log” displays the result from interactions with the “AI Pyro” window. Here you can see the results after reloading or changing a brain, including error or faults during compilation and evaluation of the brain being loaded. If all is well it will display the document text of the loaded brain.

Through the “output log” you will even see the result after executing an expression using the “expression box”. The “expression box” allows a user to evaluate the robot’s current status dynamically. Here you have access to all the variables a robot and its devices is working with. You can even watch variables, execute device commands, open brain files for editing and reset the brain. Here follows a list of commands that you have access to:

Table 4-1 List of commands accessble through the Pyro GUI

Command	Action
run	will execute and run the robot brain
runtilquit	see run
step	will do a robot step
info	will return robot information
!	will display the command history and the index of the command that you can execute through command '!'. See '! <index start> - <index end>' and '! <index>' for more detail.
!!	will execute last command
! <index start>-<index end>	will execute command in history from index start to stop
! <index>	will execute command at index in history. See ! for more information.
$ <expression>	will forward command to operative system console
about	will display about text
reload	will reset engine
reset	see reload
restart	see reload
load brain <name>	will load the specified brain. Omit file extension.
stop	will stop the robot execution
quit	see stop
exit	see stop
bye	see stop
edit	opens the currently executed brain in notepad editor. The program will freeze until the editor has been closed. This may result in unexpected behavior. It is recommended to open and edit the file manually instead. When done, you can reload it dynamically through the reload command or button.
watch <expression>	will add the expression to the watch expression list. And execute and output the result of the expression each time a command is executed by the expression

	editor
unwatch <expression>	will remove the expression from the watched expression list
browse <expression>	currently not implemented
<python expression>	will execute any python expression displaying a traceback error description on failure. Here you can define and work with variables as well as using current robot and device variables in your expressions.

This GUI allows users to evaluate dynamically any part of the robot's execution at run time. For instance, to check what devices are currently available, we enter the following into the command prompt:

```
>>> robot.builtinDevices
['ir', 'gps']
```

This returns the built-in devices loaded by the machine. You can even access these device methods and attributes by calling them directly.

```
>>> robot.ir[0].name
ir[0]
>>> robot.range["front-all"]
```

The last command will return a list of all front sensors. To get the sensor values you can query them separately through call

```
>>> robot.range["front-all"][0].distance()
```

This will return the state of the assigned range sensor (in this case the front row of IR sensors). The query returns the intersection state of all the front sensors relative to the obstacles in front of the machine.

If you get stuck after a collision or accident you may need to reorient the machine. You can restore the machine's position from a brain or through the console by calling.

```
>>> robot.radio[0].getServices() #returns all available services
["request reset", "may-day stall"]
>>> robot.radio[0].callService("request reset")
>>> robot.radio[0].callService("request transport", {"x":1,"y":2,"z":3})
>>> robot.radio[0].callService("request move-to", {"x":1,"y":2,"z":3})
```

You can access some services through the radio GUI while others need more data to complete the request. To reset the machine, call the "request reset" service. It lifts and then drops the vehicle on to the current location, resetting its orientation.

The "request transport" command is unofficial and will translate the machine from the current location [x, y, z] to [x+delta_x, y+delta_y, z+delta_z] units and drop the machine on that location. While "move-to" will move the vehicle to [x, y, z] in world units.

When running a brain script, you may want to edit and make changes to it. Currently, we support opening the currently running brain using Notepad. This can be done at runtime by calling

```
>>> edit
```

This will open Notepad with the brain code that is currently executing. When done, you can close the file and reload the brain by pressing the “reload” button on the GUI interface. Note that calling “edit” may result in the system crashing since the Notepad application will stop all current processes within the framework until the application is closed.

4.2.6 Brains

“Any program that controls a robot (physical or simulated) is referred to as a brain. It is written in Python using Pyro classes.

Programming a brain usually involves extending existing class libraries. The libraries help simplify robot-specific features and provide insulation from the lowest level details concerning hardware drivers. In fact, the abstraction provided uniformly accommodates the use of actual physical robots or their simulations even though vastly different drivers and communication protocols may be used underneath the abstraction layer. Consequently, a robot experimenter need only concentrate on the behavior-level details of the robot. Robots may feature a disparate set of sensors and movements, and yet, depending on the specific robots or simulators used, Pyro provides a uniform way of accessing those features without getting bogged down by low-level details.”[31]. Specifically, a brain implementation is a class that extends the `pyrobot.brain.Brain` class. The `Brain` class has access to some methods. Here follows a list of the most common methods used:

- `__init__`; is called when the Brain is initiated
- `step`; is called every 100 ms, here is where you handle the data to input into the algorithms
- `callback`; is called when registered events (not trapped by other calls) notifies you on the event.
- `activated`; is called when the brain is activated through the GUI interface
- `deactivated`; is called when the brain is deactivated or another brain is selected.

[31] Pyrobot, 2009, <http://pyrorobotics.org/?page=Pyro_20Brains>

Each python file that is used as a brain must have an `INIT` function defined. This function registers the brain in the system that starts the stepping “process”. All the brains used by the system are located in the directory

machine-race-directory/script/user/

Using the “AI Pyro” window interface you can easily select and execute a brain (refer to section “Interface” for more details).

Figure 4.13 shows a UML diagram displaying the currently implemented example brains.

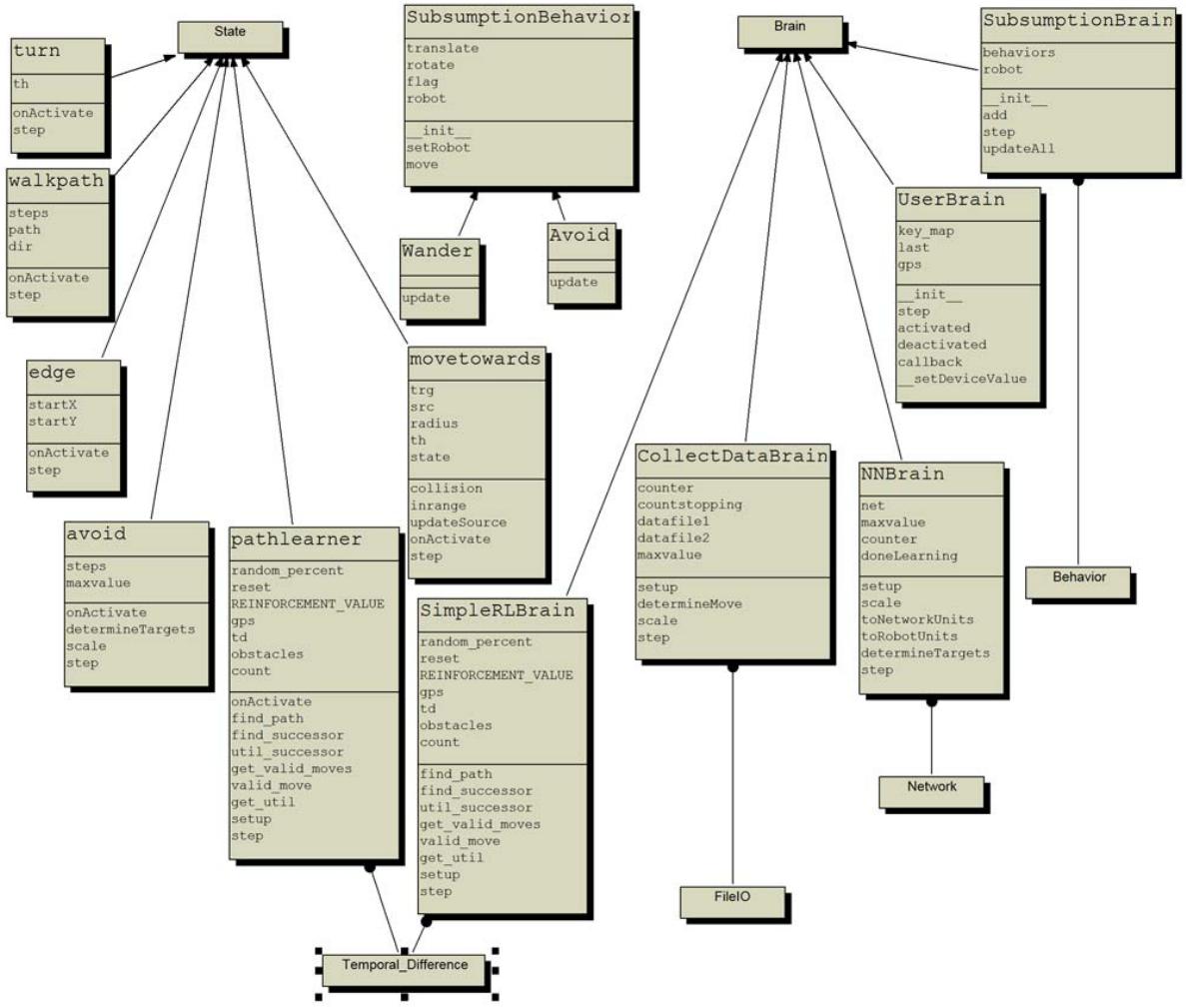


Figure 4.13 Pyro brain implementation UML

The UML in figure 4.13 shows an overview of some of the example brains used and how they relate with regards to base classes. The "`Brain`" class is the main class that we extend for most behaviors, while "State" classes are used to store a state that is used during different behaviors (see section "Brain examples" for more details).

Here the `SubsumptionBrain` uses `SubsumptionBehavior` brains `Wander` and `Avoid` to implement a layered behavior best described on the Pyro community site. The turn and edge classes are used in the example "Finite state program".

`NNBrain` class uses neural network behavior and will be covered in the section "Learning to Avoid Using Neural Networks". `CollectDataBrain` is a brain that collects data into a neural network that is then saved.

`UserBrain` is a brain that uses no specific AI algorithms. In this example we cover how to control a vehicle directly, using callbacks that are triggered using the keyboard.

`SimpleRLBrain` is an implementation of using reinforcement learning to analyze a predefined grid using random search methods in a grid system. While `walkpath`, `avoid`, `pathlearner` is an implementation of searching the same grid and then walking the learned path, using reinforcement learning methods switching between states.

Technical note: The official Pyro package distributed through the Pyro community uses threading. Our implementation does not use threading since our Python compilation is integrated without threads. The original Pyro implementation would run each brain independently in a separate thread calling step. . This threading is simulated by a calling step in a callback function. When the step method has executed the brain´s time step it will execute the next brain's time step until all brains have executed a single step. Then Python returns to the game system, executing internal game dependent features until the next time the brain-callback-timer is called

4.3 Brain Examples

Here follows simple examples of different brains that are distributed with the framework. Most brain examples, except the Mountain Car Problem, are taken from, or inspired by, the official Pyro Robotics page[32]. These examples are mainly meant as an overview of features and possibilities using our framework.

4.3.1 Simple Control

This example will show how we can move a machine forward with 30% power. For every step Pyro brain translates the "robot" vehicle using 30% of the motor engine max power.

The example will not use any AI. Below is a code example where the AI programmer extends the `pyrobot.brain.Brain` class that is initiated and returned through the `INIT` function:

```
"""Simple brain that moves machine 30 percent of the max power of the vehicle"""
```

[32] Pyro 2009, <http://pyrorobotics.org>

```
from pyrobot.brain import Brain
class SimpleBrain(Brain):     # ← define the robot's brain class
     def step(self):          # ← only method you have to define is the step method
      self.robot.translate(0.3)     # ← go forward
def INIT(engine):             # ← create a brain for the robot
   return SimpleBrain('SimpleBrain', engine)
```

When loading this code into the machine, you will notice a message in the "output log" displaying the quoted string message above "Simple brain that moves machine 30 percent of the max power of the vehicle". This text is read from the python module "doc string". This is the first part of the file that is commented, and describes the brain. This "doc string" is useful for developers when they want to add some description of the developed brain. It is also useful for the user when selecting different brains through the GUI interface.

In the `SimpleBrain.step (…)` method we see a call to `self.robot.translate (…)` this tells the robot to move forward, using 30% of its maximum engine power. This will be executed the moment the underlying robot calls the `INIT (…)` method, returning the "`SimpleBrain`" class for registration as the current brain.

4.3.2 User Control

This example will show how to control a machine directly, using keyboard arrow keys. The example will cover how to register the keys and manage callbacks. Controlling a vehicle using a keyboard is done by calling the robot devices directly. Subsequently we will not use machine learning libraries to control the vehicle.

For this example, we must override the class initialization methods (`__init__`) because we want to register the keyboard keys. Since every brain must initiate the underlying Pyro brain, the programmer initiates the user `UserBrain` by calling our overloaded parent initialization method, making sure that Pyro registers and starts up the base brain dependencies as well.

```
Brain.__init__(self, 'UserBrain', engine)
```

Then we register keys "left", "right", "up" and down through method

```
SystemHandler.keyEnabled(k, True)
```

Where *k* is the key and "`True`" denotes that this key should notify when it is pressed or released.

Note that in this example, nothing is done when the brain is stepping. This is because we want to have full control of the vehicle through keys rather than an evaluated behavior. Our behavior is controlled instead by the user through the registered keys that call the callback method. This method in turn calls `__setDeviceValue,` which evaluates the

arguments to calculate if the user wants to drive forward, backwards or turn the vehicle. Finally, this is applied to the vehicle through the call

```
self.robot.move(self.last["MOVE"], self.last["ROTATE"])
```

When the user selects or deselects the "`UserBrain`" we want it to stop the vehicle. To apply this, the programmer overrides the methods `activated (…)` and `deactivated (…)` methods and sets the machine movement and rotation to "0".

```
"""
Brain script forwarding key actions to rover.
This script allows users to take manual control of the vehicle.
"""
from pyrobot.brain import Brain
import SystemHandler
class UserBrain(Brain):
    key_map = {"LEFT":"ROTATE_LEFT", "RIGHT":"ROTATE_RIGHT", \
           "UP":"MOVE_FORWARD", "DOWN":"MOVE_BACKWARD"}
    last = {"MOVE":0, "ROTATE":0}
    gps = None
    def __init__(self, engine):
        Brain.__init__(self, 'UserBrain', engine)
        #register the keyboard keys
        for k in self.key_map.keys() : SystemHandler.keyEnabled(k, True)
    def step(self): pass
    def activated (self):
        """called each time we activate the brain"""
        self.robot.move(0, 0)
    def deactivated (self):
        """called each time we deactivate the brain"""
        self.robot.move(0, 0)
    def callback (self, source, event, message):
        self.__setDeviceValue(message, event)
    def __setDeviceValue (self, action, enable) :
        action = self.key_map[action]
        if action == "ROTATE_LEFT" : self.last["ROTATE"] = 1*int(enable)
        elif action == "ROTATE_RIGHT" : self.last["ROTATE"] = -1*int(enable)
        elif action == "MOVE_FORWARD" : self.last["MOVE"] = 1*int(enable)
        elif action == "MOVE_BACKWARD" : self.last["MOVE"] = -1*int(enable)
        self.robot.move(self.last["MOVE"], self.last["ROTATE"])

def INIT(engine):
    brain = UserBrain(engine=engine)
    return brain
```

Using concepts from this section, you can implement machine learning brains that allow a user to take over control of the robot while maneuvering the environment. Or add direct user control of other devices like the radio to reset the location of the machine.

4.3.3 Simple Avoid Brain

This simple brain avoids obstacles. On every call to step we read the sensors getting the shortest distance to the object. If the shortest distance is "`minSide`" then we stop the vehicle and rotate it away from the obstacle. If no obstacle exists we drive forward with 0.5 (50%) power.

We query the minimum distance of the front sensors by checking the left and right front distances detected by the sensors.

```
min([s.distance() for s in self.robot.range["front-right"]])
```

Here, we use sensors to read the surrounding environment and take appropriate action. The code below will drive an object and rotate it if it senses an obstacle in front of its left and right sensors.

```
"""
A Simple avoid brain.
"""
from pyrobot.brain import Brain
class Avoid(Brain):
    def wander(self, minSide):
        # if approaching an obstacle to the left side, turn right
        if min([s.distance() for s in self.robot.range["front-left"]]) < minSide:
            self.move(0,-0.3)
        # if approaching an obstacle to the right side, turn left
        elif min([s.distance() for s in self.robot.range["front-right"]]) < minSide:
            self.move(0,0.3)
        else: # go forward
            self.move(0.5, 0)
    def step(self):
        self.wander(1)
def INIT(engine):
   return Avoid('Avoid', engine)
```

Using sensors we can query a machine's distance to its environment through calls to `self.robot.range` with these attributes.

- `left`
- `right`

- `front-left`
- `left-front`
- `front-right`
- `back-left`
- `back-right`
- `back`
- `back-all`
- `front`
- `front-all`
- `all`

Through this interface, we can orient the machine according to specific rules. For instance we could create a machine that tries to move through a collection of obstacles in a landscape. But an enhanced implementation could remember the location of the obstacles encountered and use this information when planning its path around the landscape. This could be accomplished by remembering the location of the vehicle within the world using a grid. To get the vehicle location, we can query and store the robot's location using calls `robot.x`, `robot.y`, `robot.z` and save this data into the grid.

4.3.4 Learning to Avoid Using Neural Networks

This brain is more or less taken as is from the collection of examples provided by the Pyro community (where changes are marked "compared to original"). It teaches a neural network avoidance by moving the machine in an environment withobstacles and forwarding the sensor readings and appropriate behavior into the neural network.

The appropriate behavior is pre-programmed through the `determineTargets (…)` call and this is the target behavior that we want the neural network to learn. The progress is printed out to the log ("xMessage.log" in the machine-race-directory/log directory). Here you can see an output produced by the print statement and what the target values should be, compared to what the neural network has learned. For details on neural networks, refer to the Pyro documentation.

```
"""
Learning on-the-fly to avoid obstacles using neural networks.
"""
from pyrobot.brain import Brain
from pyrobot.brain.conx import *
from time import *

class NNBrain(Brain):
    def setup(self):
        self.net = Network()
```

```
        self.net.addLayers(3,2,2)
        self.net.setEpsilon(0.25)
        self.net.setMomentum(.1)
        self.maxvalue = self.robot.range.getMaxvalue()
        self.counter = 0
        self.doneLearning = 0
    # Scale the range readings to be between 0 and 1.  Also make close
    # obstacles register high readings, and distance obstacles low readings.
    def scale(self, val):
        return (1 - (val / self.maxvalue))
    # The robot will get translate and rotate values in the range [-0.5,0.5],
    # but the neural network will generate outputs in the range [0,1].
    def toNetworkUnits(self, val):
        return (val + 0.5)
    def toRobotUnits(self, val):
        return (val - 0.5)
    def determineTargets(self, left, front, right):
        if front < 1.0:
            print "front"
            return([0.0, 0.5])
        elif left < 1.0:
            print "left"
            return([0.2, -0.5])#compared to original we increase 0.0 to 0.2
        elif right < 1.0:
            print "right"
            return([0.2, 0.5]) #compared to original we increase 0.0 to 0.2
        else:
            print "clear"
            return([0.3, 0.0]) #compared to original we reduce 0.5 to 0.3
    def step(self):
        if self.doneLearning:
            self.net.setLearning(0)
        else:
            self.net.setLearning(1)
        # Set inputs
        left  = min([s.distance() for s in self.robot.range["front-left"]])
        front = min([s.distance() for s in self.robot.range["front"]])
        right = min([s.distance() for s in self.robot.range["front-right"]])
        inputs = map(self.scale, [left, front, right])
        trnTarget,rotTarget = self.determineTargets(left, front, right)
        targets = [self.toNetworkUnits(trnTarget), self.toNetworkUnits(rotTarget)]
        # Learn
        # input and output are the names of the layers
        self.net.step(input=inputs,output=targets)
        trnActual = self.toRobotUnits(self.net['output'].activation[0])
        rotActual = self.toRobotUnits(self.net['output'].activation[1])
        # Check if the robot is stuck and give it a kick to unjam it
        if self.robot.stall:
```

```
            print "stuck--reversing"
            self.move(-0.5, 0.0)
            #compared to original, removed sleep call
        else:
            if self.doneLearning:
                print "move", trnActual, rotActual
                self.move(trnActual*1.0, rotActual*2.0)#compared to original
            else:
                print "step", self.counter, "target", trnTarget, rotTarget, \
                    "network", trnActual, rotActual
                self.move(trnTarget*0.5, rotTarget*2.0)#compared to original
            self.counter += 1
def INIT(engine):
    return NNBrain('NNBrain', engine)
```

This example does not use the learned behavior collected by our neural network. But using the learned target values would be as easy as taking the learned values that are printed out to `log (trnActual, rotActual)` and controlling the vehicle using these values instead.

Collecting data from the environment and forwarding these different values into the neural network allows many possible complex behaviors to be learned.

4.3.5 Finite State Program

When implementing a behavior we may want to use different techniques depending on the current state. For instance, for a brain that is moving in a boxed pattern, we may want to split the behavior so that the brain's first mission is to move the vehicle forward. In this state, the vehicle's only focus is to move a distance larger than one. When this mission is accomplished, the vehicle changes state by calling `self.goto('turn')`.

The mission of the vehicle is to rotate itself. In this state, the vehicle's only mission is to rotate 90 degrees, checking the vehicle's current rotation during each call to step. When this step is reached, we call `self.goto('edge')`

```
"""
Robot goes forward and then slows and backup up when it detects something
"""
from pyrobot.geometry import *   # import distance function
from pyrobot.brain.behaviors import State, FSMBrain

class edge (State):
    def onActivate(self): # method called when activated or gotoed
        self.startX = self.robot.x
        self.startY = self.robot.y
    def step(self):
        x = self.robot.x
        y = self.robot.y
```

```
            dist = distance( self.startX, self.startY, x, y)
            if dist > 1.0:
                self.goto('turn')
            else:
                self.robot.move(.3, 0)

    class turn (State):
        def onActivate(self):
            self.th = self.robot.th
        def step(self):
            th = self.robot.th
            r  = angleAdd(th, - self.th)
            if r > 90 :
                self.goto('edge')
            else:
                self.robot.move(0, .2)
    def INIT(engine):
        brain = FSMBrain(engine=engine)
        # add a few states:
        brain.add(edge(1)) # 1 makes it active
        brain.add(turn())
        return brain
```

Using state-based brains, we can split up a behavior into smaller missions. Using this kind of technique allows us to build smaller focused states that are triggered by their environment.

For instance, we could create a machine that learns how to move around in an environment until it gets stalled. When the machine gets stalled, we move the machine to a state where it resets itself and checks that it is no longer stalled. When it can move itself again, we change back to the learning state.

4.3.6 Mountain Car Problem

Python has the ability to import external packages. These are easily integrated into the system by copying the libraries into the directory

machine-race-directory\common\python\lib\

When copied to this location, we can easily import it into the "brain" using the regular "`import`" statement provided by Python. An external library was required for solving the Mountain Car problem because the Pyro RL implementation only had a Temporal Difference (TD) implementation rather than a general RL interface. This example will use an external package from RLAI[33].

[33] RLAI 2009, <http://rlai.cs.ualberta.ca/RLAI/RLtoolkit/RLtoolkit1.0.html>

Solving the mountain car problem required a flexible implementation of Reinforcement Learning Algorithm. This was found using the RLToolkit, incidentally the RLToolkit also had a 2D implementation of the mountain car problem as an example, originally developed by Richard Sutton. This example was used as a base to our solution.

RLToolkit's `RLInterface` was used to solve this problem because it has a general implementation of the RL concepts, making it easy to apply reinforcement learning techniques to a wide range of problems.

The `RLInterface` requires the programmer to define two primary functions that is used when learning.

environmentFunction(action) ==> state, reward

agentFunction(state, reward) ==> action

Both of these functions feed each other through the RLToolkit package. The `environmentFunction (…)` takes an action created by agentFunction returning the state (sensation) and reward, while the agentFunction takes input, state and reward, created by `environmentFunction (…)`.

During initialization, the RL programmer creates these functions and registers them with the RLToolkit so that the toolkit knows what to compute during the learning episodes. A learning episode consists of executing the agent function and then the environment function. Calling the agent function, the learner will make a decision based on the current state. This action is then applied to the environment vehicle that gathers information about the new state.

In this example, the learning agent is a vehicle trying to drive up a mountain hill. To accomplish this, the agent must gain enough momentum. "The difficulty is that gravity is stronger than the car's engine, and even at full throttle, the car cannot accelerate up the steep slope. The only solution is to first move away from the goal and up the opposite slope"[6] This is done by driving the vehicle backwards until it can not climb further up the hill. "Then, by applying full throttle, the car can build up enough inertia to carry it up the steep slope even though it is slowing down the whole way. This is a simple example of a continuous control task where things have to get worse in a sense (farther from the goal) before they can get better. Many control methodologies have great difficulties with tasks of this kind unless explicitly aided by a human designer."[6] To learn this process a learning state called "learn" was created. This class extends the `FSMBrain` class that enables us to switch between different states. The learning state is a class wrapper that handles the RL learning through `RLinterface`. The brain begins by registering our main RL functions `mountainAgent (…)` and `mountainEnv (…)`.

```
self.rli = RLinterface(mountainAgent, mountainEnv)
```

Then we set up up the vehicle and remember the time (which is used to calculate the speed) and our position (used when learning). But the first thing we need to handle is the vehicle's orientation. This is done by the applying the"rotate"state.

```
self.goto('rotate')
```

Since the vehicle is in a 3D enviroment, we must adjust the vehicle's orientation to turn towards our goal. This is managed by using the "`rotate`" class. The "`rotate`" class is another state-based class that manages the rotation of the vehicle. Since one brain state can only be applied at any one time the "`learn`" and "`rotate`" class will call each other when they think the state must be altered. The learner will call the "rotate" state when the vehicle is oriented more than 45 degrees from the goal.

```
if (abs(prefered_th-self.robot.th)) > 45 : self.goto ("rotate")
```

Here "`prefered_th`" is 90 degrees and `self.robot.th` returns the current vehicle rotation. Using this method, we can make sure that the vehicle is always oriented and driving towards the *z* axis, while learing to climb the hill. The actual learning is done when the `learn.step (...)` method is called. This will in turn forward the call to `rli.step (...)` which handles RL learning. `rli.step (...)` calls the function `mountainAgent (…)` and `mountainEnv (…)`. The method `mountainAgent (…)` is the actual learning agent. These functions are regular method functions that the AI programmer uses to define the actual behavior of the agent and environment. The behavior is an action taken by the learner based on the current state. This action is chosen by the learner, based on the provided settings set by epsilon and the learned action based on the current state. The `mountainAgent (…)` method takes state *s* and reward *r* as an input value using Sarsa to learn the desired behavour based on the environment function. When calling the RL Sarsa algorithm these inputs, *r* and *s*, are fused into the algorithm along with the action, *a*. This combination is stored and used to calculate the learned action. This action value is returned by the `mountainAgent (...)` to be used as an input to the `mountainEnv (...)` method.

The method `mountainEnv (...)` will actually apply the action, *a*, to the vehicle. While doing so it calculates the current position, *p* and velocity, *v*. These for a tuple

```
(p, v)
```

The tuple is the actual state that is returned back to the agent, using the registered call `mountainAgent (...)`. Below is the actual code implementing the learner state class:

```
class learn(State):
    rli = None
    def getRotation(self) :
        a = self.robot.devData["getRotation"]()[0]
        return a
    def onActivate(self): # method called when activated or goto
        if self.rli == None :
```

```
            setupEnv ()
            #lets use RL toolkit
            self.rli = RLinterface(mountainAgent, mountainEnv)
            #lets setup the machine
            self.initPos()
            #we need to keep track of time for speed calculations
            self.last_time = time.time()
            #we probably need to adjust the vehicle to face right direction
            self.goto('rotate')
            #lets get the position of our wall
            self.posx = SceneManager.getEntity ("target").getPosition()[0]

    def initPos (self) :
        #we always want to learn from this position
        self.setPosition (287.43, 0, 209)
        #we want the vehicle to start with breaks on turning wheels front
        self.robot.move (0,0)
        #lets assume that the vehicle is not stalled to begin with
        self.robot.stall = False

    def getVelocity (self) :
        """
        This calculates the velocity of the vehicle and used
        by the learner to know how fast we are going while learning
        """
        b = [self.robot.x, self.robot.y]
        a = self.last_pos
        time_now = self.robot.timestamp
        timedelta = time_now - self.last_time
        distance = math.sqrt (abs(math.pow(a[0]-b[0],2) + math.pow(a[1]-b[1],2)))
        self.last_pos = b
        if distance == 0.0 : return 0.0
        if timedelta == 0.0 : return 0.0
        velocity  = float(distance) / float(timedelta)
        self.last_time = time_now
        return velocity

    def getPos (self) :
        """
        Since we are only learning one dimension lets return that only
        This is used by rli.mountainEnv
        """
        return self.robot.z
    def setPosition (self, x,y,z) :
        """This sets the vehicle position in world space using the radio device"""
        dic = {"x":x,"y":y,"z":z}
        self.robot.radio[0].callService("request move-to", dic)
        self.last_time = self.robot.timestamp
```

```
            self.last_pos = [self.robot.x, self.robot.y]

        def step(self) :
            """This is executed every 100 ms while this state is used"""
            global rotdir
            #call this to do the actual learning,
                  #step will call RL toolkit methods mountainEnv, mountainAgent
            self.rli.step ()
            if self.robot.stall :
                #we are stalled lets reset the vehicle
                if rotdir > 0 : carbrain.initPos()
            #lets correct the vehicle orientation if of course
            if (abs(prefered_th-self.robot.th)) > 45 : self.goto ("rotate")

        def isPassedGoal (self) :
            """Checks if we have passed our goal, this is used by rli.mountainEnv"""
            return self.robot.x > self.posx

    def INIT(engine):
        global carbrain
        fsm = FSMBrain( engine=engine )
        carbrain = learn(1)
        # add behaviors, lowest priorities first:
        fsm.add( rotate (0) )
        fsm.add( carbrain )
        return fsm
```

This class has some helper functions. The method `getVelocity (...)` is used to calculate the velocity of the vehicle by using the time elapsed and distance traveled. While the method `getPos (...)` gets the current position along the *z* axis. `initPos (…)` is used to reset the vehicle at the starting position. `onActivate (...)` is the method called when the brain is executed for the first time and will it initialize the RL interface along with setting initial data.

To incoporate the mountain car example provided by the RLToolkit, we had to scale some values fitting the example with the world scale. These values had to do with the fact that data collected by the robot was in world scale, but the RLToolkit example used radians. The original implementation placed the agent at the bottom of a sinus curve, at “-0.5”, with the range from “-1.2” to “0.6”. From there it was supposed to reach a goal at location “0.5”.

The simulated world that was used for our implementation is much larger than this. We place the vehicle at location 287 (or more specifically x=287, z=209), wanting it to reach a goal located at 320. The range that the vehicle used to gain momentum on was from 120 to 320 along the z axis.

The original implementation shows the results using a state value chart that showed the learner’s progress based on state value of the learned scope. Using `dataFlush (…)` method, the user can flush the current state value used by the machine by calling the following commands using the “AI Pyro”:

```
>>> import MountainCarBrain
Ok
>>> MountainCarBrain.dataFlush()
```

This will write the state value data onto an xMessages.log file in the log directory. Using this data, a graph can be created that shows the current state value, see figure 4.14.

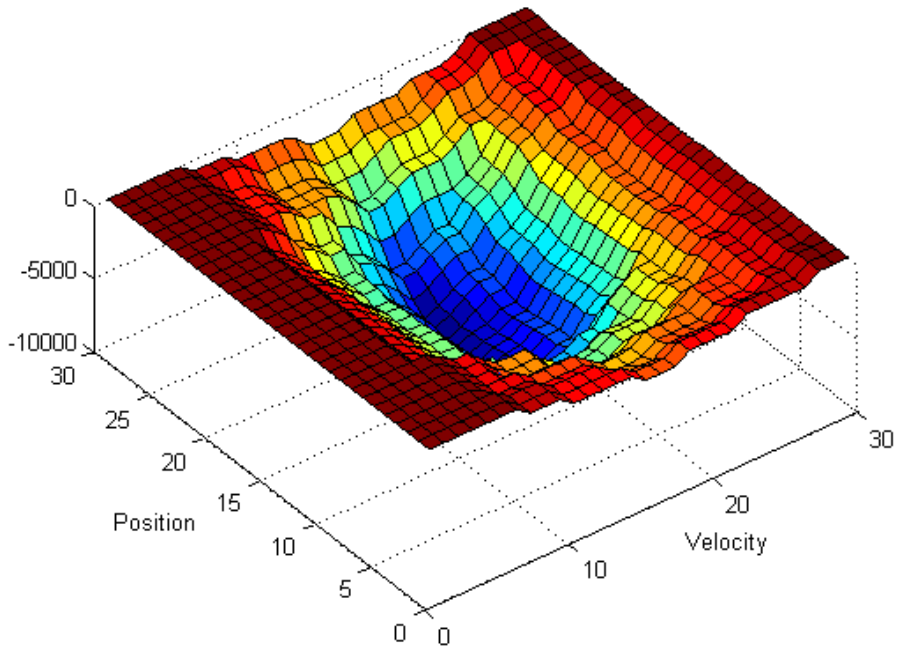


Figure 4.14 Mountain car state value graph

The results of this graph show that the RLToolkit state value alters during the episodes, similar to the original graph provided by the example (see appendix, "Mountain Car").

Learning the orientation of the vehicle in world space would be interesting as well. It is possible by reading the *th* value and passing it within the position *p* and velocity *v* tuple.

```
(p, v, th)
```

Additionally, the vehicle must have a state that orients itself towards a particular location rather than along the *z* axis and that is possible by altering the rotation code to calculate the required rotation to the goal while rotating, stopping rotation when it reaches "0" degrees.

4.4 Summary

This chapter has shown the reader how the system functions from a user point of view, everything from installing the game to creating a machine from scratch.

This chapter also shows how the user could program and assign a script that fully controls the machine devices. Some of these ideas were described using reinforcement learning concepts for reference, showing how “brains” are assigned within the game framework. The chapter ended by showing examples of such “brains” and how they can be used.

5 CONCLUSION

The final chapter begins with an overview of the results, then the future of the project will be covered and finally we will consider problems that were left unsolved. In the section "Problems Left Unsolved" we review what could be improved.

5.1 Discussion

The author of this thesis has programmed a framework for working with machine learning methods using open standards. The framework uses The Pyro machine learning library, resulting in an environment for testing and learning artificial intelligence. The framework offers authors of machine learning methods a challenging dynamic environment.

The scene uses rendering solutions used by commercial games to display the world to the user, allowing designers to use advanced graphics technology for advanced effects.

It also provides users with an integrated community, allowing a collaborative and competitive framework. Connecting the user to an online community withan economic subsystem, we provide a vibrant and fun environment for them to compete, using machine learning methods.

The software is accessible online through download from two different sources. The differences between them are that the first is a simple stand-alone application where the online dependency has been removed. The second one provides users with the framework as it is meant to be used.

Offline: http://www.kjartanjonsson.com/ MSThesisOffline.zip

Online: http://www.machinerace.com

5.2 Future Work

Pyro has good tutorials, many examples and a clean architecture. The Pyro packages and algorithms are, in this author's opinion, not very well documented as an API. Neither are the interfaces to devices. Implementing Pyro into the framework was challenging, however, since the Pyro library is designed to control the main threads, rather than to be a passive thread waiting for execution. To solve this, we had to remove any threading created by the library. Removing these threading interactions could result in unexpected behavior since interaction between the different library modules may be broken. But due to the lack of testing routines, we can not guarantee that results are comparable with the Pyro system. However, we do get expected results from the current test implementations and must use more examples to test the additional features of Pyro.

Implementing a threading version of Python script execution could result in the library being implemented in a cleaner fashion. It could also offer us to do multitasking, so that fetching data from network sources would not temporarily suspend the system and offer

other features, like the ability to open a text editor while the system processes world graphics, rather than halting.

We have not yet implemented all of the different devices supported by the Pyro abstract interface such as mapping and vision features. Adding these would give the machine a wider range of application. Pyro also assumes in some cases that it controls the world. Through this, it has all the information about the current world state. This is sometimes used in some examples like reinforcement learning.

The artwork in our world needs further development, as well as the rendering effects that are possible to create using OGRE3D.

Currently, no sound is used in the game world but it is supported by the framework using OpenAL. To enable this, we simply call the sound library to play the sounds accordingly. We also need to provide triggers to the scripting interface so that we register a sound to be played during a particular event. This provides further immersion to the player.

The current architecture of the entire system assumes an underlying networking architecture where the position of the entities is synchronized with the online world server through “real time” servers. The users connect to these real time servers to synchronize their positions with each other. The real time servers in turn connect to the world server, updating the current state so that all users are synchronized. Currently, version 0.3.0 has no synchronization enabled. Further work must be done in dead reckoning for this to be usable and version 0.9.0 has some enhancements to support this. We also want to use HTTPS instead of HTTP when communicating, using XML-RPC and providing users with further security, eliminating the possibility of “man in the middle” attacks.

These corrections and other implementations are still under development.

REFERENCES

1. Blank, D, Kumar, D, Meeden, L, Yanco, H 2006, 'AI Magazine', *Robots and Robotics in Undergraduate AI Education, Special Issue of AI Magazine*, vol. 27 no. 1, pp. 39-50.

2. Fossum, T & Snow, J 2006, 'How Platform-Independent is Pyro?', Department of Computer Science, Potsdam NY

3. Blank, D, Kumar, D, Meeden, L, Yanco, H 2005, 'Pyro: An Integrated Environment for Robotics Education'

4. Blank, D, Kumar, D, Meeden, L, Yanco, H 2004 'Pyrobot: A Python-based Versatile Programming Environment for Teaching Robotics'

5. Blank, D, Yanco, H, Kumar, D, Meeden, L 'Paper Avoiding the Karel-the-Robot Paradox: A framework for making sophisticated robotics accessible'

6. Sutton, RS & Barto, AG 1998 *Reinforcement Learning: an introduction*, The MIT Press, London

APPENDIX – LEGACY SYSTEM

The bulk of technology development has taken place since 2003. The system framework was used in a 2005 B.S thesis "Tuning physics based systems using Interactive Evolution". The paper studied how well a vehicle can adapt to the user requirements based on feedback when driving.

The base technology had a vehicle controlled by the user driving in a physical environment.

We built on this technology, creating our current framework. Adaptation mainly involved upgrading, adding and exchanging current technology.

Legacy technology

The current technology was built on my B.S project. The B.S project was in turn based on a game engine framework that has been in development since 2003.

This framework uses open source libraries as base components.

All packages are based on (or similar to) Open Source LGPL licenses. These libraries provide basic game features such as physics, sound, graphics and networking functionality.

All these libraries are glued together using C++ in the core system.

The state of art when the M.S started was

- Python (client side)
- OpenAL (client side)
- ODE (client side)
- OgreODE wrapper (client side)
- OGRE3D (client side)
- CEGUI (client side)

Python is used to change the game state. It is controlled by our system. We call it on initialization and provide interface calls to the core system. These include access to graphics, sound, network and GUI functions.

Most importantly, we provide Python with the ability to register callbacks. These callbacks are called by the internal system on events that are registered during python initialization. This way, the game designer can register what features he wants to subscribe to and gets notified on changes. Here the game designer makes desired state changes through interface calls to the system.

OpenAL is the audio library. It is supported but not used extensively. The implementation is such that each movable game object in the word has access to a game object. This game object in turn has a sound instance. The sound instance assigned to the object updates the position and orientation of the sound, returning to the user in a sensation of relational sound in three dimensions. We notify the sound instance of any additional changes such as collisions playing collision sounds.

ODE was used as a physics library. It is platform-independent and very powerful. The community is strong and much work has been done recently by the community to improve features and performance. The OgreODE wrapper was lightweight and simple to use.

But we chose to exchange ODE (and OgreODE wrapper) because it was too heavy in execution and did not support network play (entity state correction).

Figure 2.1 Jeep used during "Tuning physics based systems using interactive evolution"

PhysX was chosen because it was free to use and had a professional user base. It supported state correction better than ODE and resulted in less overhead on the overall system frame rate.

OGRE3D is the graphics library and the rendering system. It is heavily integrated into the system and drives the main loop, altering between rendering and internal game processes. The version used (Eihort, v1.4) had the Open Source project called CEGUI integrated as the official GUI.

The B.S project "Tuning physics based systems using interactive evolutions" used Python to tune the vehicle properties. Calls into the system were made to the physics system assigning the new values. The user could drive around an uneven terrain that was created from a height map. The size of the terrain was fixed approximately five hundred times the size of the vehicle.

The world objects were placed onto the world by the game scripts. The scripts loaded an XML file containing the location of the target entities. This included gates that denoted where the driver should drive.

To the left you see an image of the vehicle that was used during the experiment. A, B, C and D shows the actual physic ODE entities used during the experiment.

The problem we encountered was the lack of tools to work with in creating the world. The XML file was manually edited and the vehicle had fixed properties.

Here follows a graph of how we connected the modules together.

OGRE3D is the graphics library and the rendering system. It is heavily integrated into the system and drives the main loop, alternating between rendering and internal game processes. The version used (Eihort, v1.4) had the Open Source project called CEGUI integrated as the official GUI.

The B.S project "Tuning physics based systems using interactive evolutions" used Python to tune the vehicle properties. Calls into the system were made to the physics system assigning the new values. The user could drive around an uneven terrain that was created from a height map. The size of the terrain was fixed approximately five hundred times the size of the vehicle.

The world objects were placed onto the world by the game scripts. The scripts loaded an XML file containing the location of the target entities. This included gates that denoted where the driver should drive.

To the left you see an image of the vehicle that was used during the experiment. A, B, C and D shows the actual physic ODE entities used during the experiment.

The problem we encountered was the lack of tools to work with in creating the world. The XML file was manually edited and the vehicle had fixed properties.

Here follows a graph of how we connected the modules together.

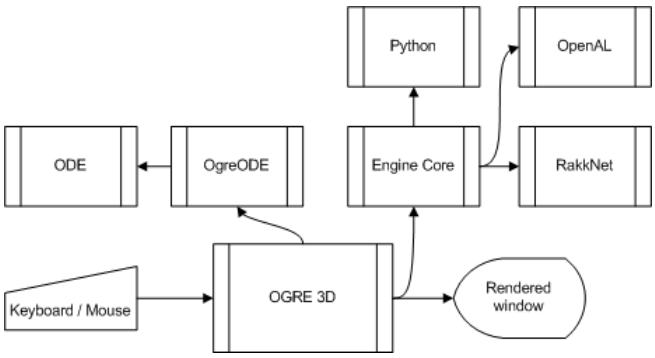


Figure 2.2 Interactions between open source libraries

The technology base was enough to do subsequent research but we found it heavy in execution compared to the limited size of the world.

The system supported peer to peer communication allowing for simple entity state synchronization. This was through simple TCP/IP libraries using simple strings as protocol.

APPENDIX – SETUP

The system requires us to run a collection of binaries on the client machine. These must be installed and maintained. Here follows an overview of the installer, how it is created, maintained and distributed. We also describe how the patcher works and distributes the newest code releases.

Installer

The installer is created during the automated build process. It is compiled after a successful build of the “game script”.

The “game script” build will get the newest files and run game dependent tests. It checks the unit-tests created for game dependent features. On success it will create a directory called “game-clean” having all the game files. Then it will remove all the development files used when creating the game such as the original modeling files, compiled python files and workspace files.

The “game-clean” directory now contains a clean distribution of files. This collection of files is used by the installer build process.

We make a copy to the “game-clean” directory that is named “game-export”. We then run Wix to create the installer. Using Wix, we run “Tallow”. The “Tallow” program walks the “game-export” directory creating an XML file called “machineraceonline-files.wxs”. This XML file contains a list of all the files needed for the install without GUIDs. The GUIDs is stored in a pre-created XML file called “machineraceonline-uuid.wxi”. We use Python to parse both “machineraceonline-files.wxs” and “machineraceonline-uuid.wxi” and copy the GUIDs to newly created “machineraceonline-files.wxs”. This file is then saved as “machineraceonline-files.wxi”.

We run candle.exe on “machinerace-install.wxs” that defines our setup process as well as including “machineraceonline-files.wxi” that creates a configuration script.

Then we run light.exe, which creates the actual setup file, “MachineRace.msi”.

Finally we test the install by running the newly created MSI file in console mode. After the install we run the game tests once more checking that we can actually run the new install and that all the required files are included. We end by uninstalling and distributing the patch files.

We will copy this distribution and move the files to a directory called “patches” and “patch-delta”. The “patches” directory contains the last set of files used during the build process. We copy the files to this directory, checking for changes in the distribution between the newly created files in “game-clean” and “patches”, copying the changed or new files to “patch-delta”.

The “patch-delta” files are then manually moved to the FTP server, 'ftp://patch.machinerace.com'.

Patcher

When a user starts the game, he runs a program named “Patcher”. The patcher is “Win32” version of “GNU Wget” released under GPL license. The Patcher is a separate program that updates the game distribution.

It is pre-configured to connect to an FTP server comparing the local directory with the online release. If there are new files online we will download them to the game directory.

Patcher will walk the online directory, checking if the file on the FTP server does not match the game directory with regard to date or file size. Then the patcher will overwrite the local game directory files that do not match. If it does not exist we simply add it. The patcher will never remove any files from the game directory.

If it downloaded a newer version of itself, that is “patcher.exe”, it will rename and delete itself during next the execution.

When the patcher has checked all files it will start the game client, “client.exe”, ending with terminating its own execution.

If the patcher fails to download, or during download, it will continue and run “client.exe”.

APPENDIX – MACHINE LEARNING LIBRARIES

When considering libraries for the project we wanted to implement Python-based libraries. All drivers were already controlled through Python and allowed for a cleaner architecture.

Implementing AI is also done at higher levels where Python is currently implemented, so implementing the machine learning library through Python made sense.

There are many Python libraries that could be considered for implementation as artificial algorithm libraries. Most of the libraries offered a limited set of machine learning algorithms, so they were not interesting as an implementation.

Bayesian library

Pebl is a Bayesian network based learning library. Its learning is based on prior knowledge and observations. With the focus on Bayesian learning only, it is limited by the different types of algorithms that we could try out and play around with. Divmod is more focused on classifying Internet site content. Open Bayes is a Python-based library that focuses on Bayes classification. The project is at a halt as of version 0.2.

Neural Network based libraries

Fast Artificial Neural Network Library (FANN) is a fast neural network implementation with a basic to advanced neural network tool set. It supports Python binding but is limited in using Neural Network principles. ffnet is another Neural Network library supporting "easy-to-use feed-forward neural network" in Python. It has basic functionality to train and use neural networks. PyAIML is an interpreter for AIML (artificial intelligence Markup Language) and is implemented using Python. It "describes a class of data objects called AIML objects and partially describes the behavior of computer programs that process them. AIML objects are made up of units called topics and categories, which contain either parsed or un-parsed data."

The AIML syntax follows XML based syntax with keywords within brackets.

Orange is data-mining software supporting Python scripts accessing C++ components. It has a higher level programming language where you program visually through so-called Orange Widgets. It reads data through tab-delimited files and C4.5 files as well as other types. The software is focused on large data sets that are classified using Bayesian classification algorithms.

Forward (-backward) chaining systems

PyKE is a Python-based knowledge inference engine (expert system). It supports backward and forward chaining (goal directed) inference. Using the library you define rules and relationships that the engine learns and uses in queries. It is relatively simple to implement and useful for data that is not numerically dynamic. FuXi is another forward-chaining production system based on Pychinko. It uses Notation 3 Description Logic Programming. It is based on Charles Forgy´s Rate algorithm.

"The working memory is a set of items which (in most systems) represent facts about the system's current situation - the state of the external world and/or the internal problem-solving state of the system itself. Each item in WM is called a working memory element,or a WME.

The production memory is a set of productions (i.e., rules). A production is specified as a set of conditions, collectively called the left-hand side (LHS), and a set of actions, collectively called the right-hand side (RHS)." - Chimezie Ogbuji (ReadMe.txt file)

PyCLIPS is a CLIPS wrapper. CLIPS is an expert system tool that learns based on created rules, similar to FuXi and PyKE. The CLIPS library has been in development a long time (1984). CLIPS uses its own structured language when defining the rules. Pychinko is similar to PyCLIPS, FuXi and PyKE, a forward-chaining rule engine where you create rules.

Support Vector Machines

LIBSVM is a library for Support Vector Machines in Python. It supports many different interfaces like Python. PyML focuses on SVMs and related kernel methods using object oriented methodologies. It reads in data through files and analyzes data. It requires the Numpy library and matplotlib. Natural Language Toolkit is a toolkit for natural language processing.

Other algorithms

Gambit is a framework for construction and analysis of finite extensive and strategic games. It contains a library of game theory software and tools for analysis of game theory. Numenta implements hierarchical temporal memory system (HTM) patterned after the human neocortex. It is a new library targeted at a professional audience. Future systems will include vision systems, robotics, data mining and analysis as well as failure analysis and prediction. PySCeS is a Python Simulator for cellular Systems.

APPENDIX – DOMAIN OF MACHINE RACE

Machine Race is a race between machines. This is viewed both from the competition and evolutionary standpoint.

Resource is a commodity of value that can be exchanged. They are 'devices' and minerals.

Corporation is a collection of resources that a board of directors controls together. A corporation can create and delegate activities. Users can own a corporation through stock.

Contractor is a person in the game that has assets and can take assignments.

Director is a person in the game that among others controls a corporation. Persons control the company through voting on actions to take.

Stock is a share in a corporation. A share is a resource; owner, corporation or amount of shares

Activity is an understanding that a contractor works for a corporation. An activity can be advertised and assigned to more than one person. Many users can take on the same activity. At the completion of the activity, the corporation must give some kind of reward if it was within the specified period or a worker was first to complete it.

Blueprint is a description of a physical representation. Blueprints can be created, bought and sold. Blueprints are resources.

Machine is a collection of devices controlled by the user.

Device is a physical or abstract entity having a specific functionality. A device is controlled by a driver and always has a visual representation if itself.

Driver is a software interface used by a robot to control a device.

Robot is a machine that has a brain.

Engine is a machine.

Brain is a script loaded by a robot and uses Pyrobot machine learning library to control drivers that in turn control a machine.

Physical is an instance of blueprint. Each physical representation has age (date of birth) damage (dependent on what blueprint it is based on to be implemented), device, holder (i.e. the user) and an owner (the one that owns it).

Maker is a thing that makes or a manufacturer. A maker can transform resources into a physical representation of a blueprint. Its input is always according to its blueprint specification. A maker is a resource.

Advertisement is in the game where people sell resources through ads. Only items in ads can be sold. It is the action of making generally known, a calling to the attention of the public.

Surface is the ground where the game progresses on a planet (excluding indoor scenes). The surface has resources/materials, depending on where on a planet there are different resources to find.

Minerals are a basic item that is gathered from the environment.

Enterprise is the system in its entirety.

State is all the properties belonging to an entity or player.

Player is a registered user.

User is a person that uses the system.

Client is the run-time system that is executed by the user on his/her local computer.

Bookkeeping is resources managed using double-entry bookkeeping practices.

World is the game world within the client game system where a player interacts with machines.

The enterprise is the race between machines. This is viewed both from the competition and evolutionary standpoint.

APPENDIX – CODE CONTROL

Creating a project of this scale and complexity, it is important that the source code is managed appropriately so that it is backed up and never lost or corrupted..

Version control

Developing any large scale application is complex and tools needed to control the code were used. We chose to use CVS because it was already well integrated into many open source tools; specifically Eclipse.

We moved to Subversion (and still moving) because CVS was not dynamic enough for directory and file changes. The tools were better and more flexible for a changing environment.

- CVS
- Subversion

Continuous Integration

We also needed to make sure that the codebase was runable on different machine specifications. We choose to use autonomous testing in our continuous build process.

Cruise Control .NET was used to detect changes and check out the newest codes, rebuild the solution running all the tests on a test machine and finally deploy the final version. The tests were layered. If the build was successful we moved to next tests.

The following tests were made:

1. System tests
2. Game script tests
3. Installer tests
4. Server tests

Currently, there is poor testing of the actual game code and Pyrobot integration. When the build has completed successfully, we have all the changes of the source code and new installer file available for release.

Releasing to the public, we decided that we needed full control. We choose to release new releases manually for more deployment control.

Open source

We are running an open source database, PostgreSQL. This database may have updates that need installment.

Django is constantly being developed and new releases should be updated appropriately.

Currently, we are running services on Webfaction that handle these updates. See section "Outsourcing" for more information.

Proprietary

Maintaining the code base involves adding features, fixing faults and errors as well as updating compatibility when dependency code changes.

We use Subversion that handles version control of our code development. The Subversion service is running at the hosting service Webfaction. We will back up locally by checking for changes and committing it into another source control system (Sourceforge Enterprise Edition).

The entire system currently runs an automated testing process. This is achieved through CruiseControl .NET. It is an automated build and testing system that detects changes in repository. An event is fired when it detects a change in the source code. All code is downloaded and recompiled. When the compilation is completed we start the next step of running unit tests. It is important that all new code (and old) has good unit tests that cover the entire developed feature.

Outsourcing

We have chosen to outsource maintenance of the following services:

- Hosting PostgreSQL database
- Hosting Django based projects
- Hosting Subversion source code repository
- Hosting static files like media and setup files
- Hosting Trac for ticket handling

Users connect the systems databases through the World Wide Web (WWW) using remote procedure calls (RPC) as depicted below.

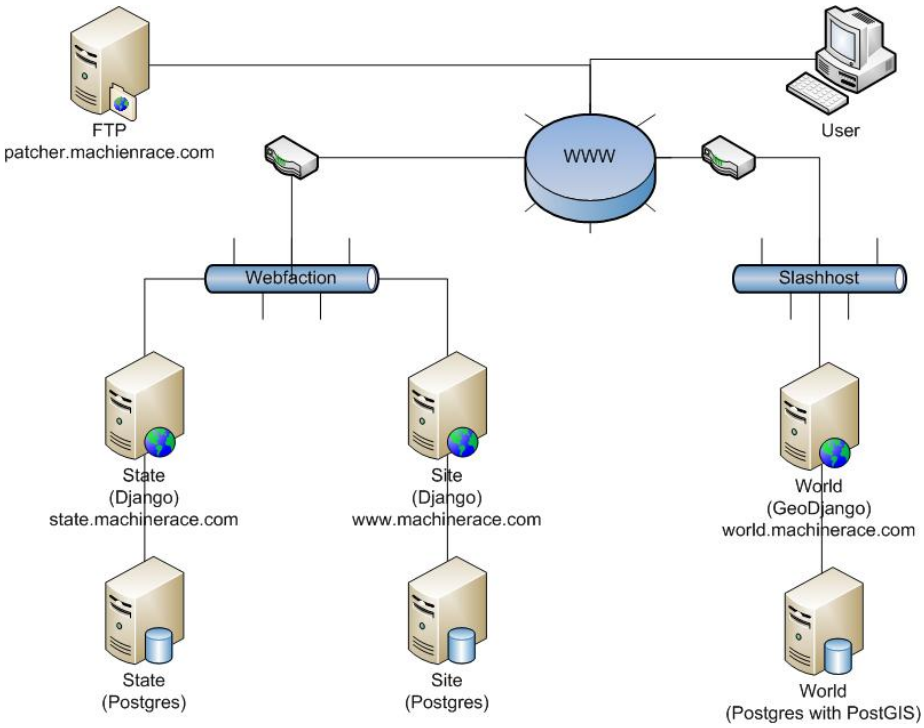


Figure 3.1 Online server architecture

Webfaction services

Webfaction is a cheap hosting company that is currently handling our sites; state, site.

They handle hardware maintenance and can easily make updates of these software solutions:

- PostgreSQL
- Django web framework
- Subversion

We are currently running on Dual Xeon 2.4 GHz with 4 GB RAM that Webfaction maintains.

Slicehost services

Slicehost is another cheap hosting company that is currently handling GIS related applications of our sites.

- PostgreSQL with PostGIS extension
- GEODjango web framework
- GEOServer

Running Ubuntu 8.04 (Hardy Heron) on 256 RAM. With 10 GB RAID-10 disks and 4-core servers that supports 100 GB bandwidth.

Maintaining the open source projects

This technology has been in development since 2003. During this time we have evaluated and tested different technologies to form the best suitable solution. The current technology is mainly based on the LGPL licensed except for PhysX (that is free for use).

Here we list the technique we are using to develop the game.

Core technology

The libraries we use are glued together using C++.

- OGRE 3D, v1.4 Eihort: is a rendering engine with a clean cut architecture and documentation. We use it for presenting our data visually.
- PhysX, v2.7.4: is a physics library currently used by next generation games. We use it for representing our physical objects.
- Python, v2.5: is a flexible scripting language. We use it for game logic.
- Raknet, v3.x: is a network library. We use it for real time data synchronization.
- OpenAL, v1.1: is a audio library. We use it as sound library (currently disabled).
- Django Project, v0.97: is a python based web framework. We use it to power this site and game state information.

Add-on projects

- Simplepagedterrain, v, OGRE
- Caelum, v0.2, OGRE
- Nxogre, v0.9-x, OGRE
- Hikari, v0.3, OGRE
- Wikimedia snippet, July 23, 2008, Django Project
- Django-registration, v0.6, Django Project
- GEODjango, v1.0, Django Project
- Pyro Robotics, v5.0.0, Python

Support technology

We are also using these tools to reduce development time and increase project organization:

- Cruise Control for .Net is an Automated Continuous Integration server. We use it to run Nant scripts, testing everything from game code to server up time.
- NAnt is a build and test tool. We use it for unit testing.
- Source Forge Enterprise Edition is an collaborative development platform. We use it as CVS / SVN servers but also as a collaboration and organizational overview tool.
- PostgreSQL is a free database server. We use it for serving this web site as well as game state information.
- Google Apps for administration
- Trac for faults and error handling (open for all users)

Here follows an additional list of tools used during software development:

- Python 2.5, for scripting and tests
- Eclipse 3.3.2, for Python development
- PyDev, Python enabled Eclipse
- PyNSource, reverse engineering
- CELayoutEditor 0.5, when using CEGUI
- TWiki is a wiki. We use it for serving internal web site
- Adobe Flash CS3 Professional (limited usage), creating GUI content

For graphics and art we used these products:

- Terragen, creating landscape files for tests
- L3DT, creating landscape files for tests

APPENDIX – CORE SYSTEM SCRIPT API

SystemHandler (version 1.17)

index
d:\cruisecontrol\game-client-system\game\common\scripts\systemhandler.py

Modules

fileIO game network

Classes

FileIO
GameBase

class **FileIO**

Methods defined here:

__init__(self, file, option='r')
Opens a file and appends it to the open file collection on success
if failed we return string describing error. On success we return NULL string and we added the file to the file collection
Options are
w: will allow write and read on a file removing current content
r: will allow read only on a file
a: will append to current file
rw:will write into current content allowing reading as well

close(self)
Closes a file and removes it from the open file collection

read(self)
Read a file

write(self, content)
Write to a file

class **GameBase**

Is the game base framework that the main game class must extend. Must be registered using call registerGameClass(...)

Methods defined here:

destroy(self)
First thing called when game system is destroyed

intersectionCallback(self, objectName, event, sourceStateTuple)
objectName is the name of the entity who detected the instersection. event is
what actully happened and sourceTargetTuple is an array of all the objects that
are currently intersecting this movable object.
For instance is may get this movableObjectName -> robot_1, status -> Intersecting,
sourceStateTuple -> ["robot_2","true"] for an intersecting robot_2 intersecting with
robot_1.

keyCallback(self, bkeyDown, strKey)
The function called when the user presses or releases keyboard buttons after the key is registered through keyEnabled(...).
When bKeyDown equals false then the user released the button. strKey is the string representation of the key in question.

networkCallback(self, event, data)
The function called when the system gets network package from an external resource. Is enabled after a call to networkConnect(...).
Event is an id (not used) and data is the a string that was receaved.

resourceCallback(self)
This callback will be called every time the system loads or unloads something (note must be notified by the system to turn up here).

setup(self)
First thing called after registration

systemCallback(self, event, message)
This callback will be called every time the system wants to say something. Messages expected are:
Event 10 means that we are loading something. Message is formated as a dictionary {'name':'Page_0_1', 'precent':51.32}
Event 11 means that we are unloading something. Message is formated as a dictionary {'name':'Page_0_1', 'precent':21}

timerCallback(self, trap, message)
The function called when a user requested a callback through funciton timerCallBack(...).
trap is the trap id sent through the timerCallBack call and message is the associated string.

Functions

addIntersectionDetection(movableObjectName)
Adds the movableobjectname to intersection detection. This object will be called back when ever another object
intersects it. The other object must exist in the list as well. Raises SystemHandlerError if we have not created
the base class or SystemIntersectionError if we could not add it to the system intersection list.

cleanUpAndExit()

delIntersectionDetection(movableObjectName)
Removes the movableobjectname from the list interested in intersection detection. Raises SystemHandlerError if we have not created
the base class or SystemIntersectionError if we could not remove from the system intersection list.

getGameClass()

keyEnabled(key, b=True)
Adds a key to be catched by our callback handler where key is a string. Refer to constants module for available keys.
Raises SystemKeysError on failure.

networkConnect(port, strIp)
Connects this application to port 'port' at ip 'strIp' having the ip as string. We will get all network callbacks and messages through the class
that extends the GameBase class. The method that receaves the call is networkCallback(...)

networkSend(str)
If networkConnect was successfull then this call sends str to connected computer. Raises SystemNetworkError on failure.

receaveMenuEvent(caller, event, param)
This will call the overrided receaveMenuEvent function made by implementer

registerGameClass(c)
Registers a class that extends the GameBase class. If no game class is registered then it will register c and call c.setup() initalizing the call.

timerCallBack(msec, trap=0, message='')
Timer request to be notified after msec with trap and message. The callback will be catched by the class that was registered using
SystemManager.registerGameClass(...) and the method timerCallback(...) will be called.

Data

__author__ = 'Kjartan A Jonsson'
__copyright__ = 'Copyright (c) 2007 aGameCompany'
__license__ = 'aGameCompany'
__source__ = '$Source: /cvsroot/game-client-system/scripts/SystemHandler.py,v $'
__version__ = '$Revision: 1.17 $'

Author

Kjartan A Jonsson

GUI (version 1.13)

index
d:\cruisecontrol\game-client-system\game\common\scripts\gui.py

This is a module that handles the graphical user interface provided by Ogre through CEGUI.
http://www.cegui.org.uk/wiki/index.php/Falagard_System_base_widgets_reference

Modules

gameGUI

Classes

SystemGUI
exceptions.Exception(exceptions.BaseException)
GUIError

class **SystemGUI**

Methods defined here:

addEventSubscriber(self, windowName, eventName, funcName, funcArgs)
Assigns a event subscriber for the widget for a layout must have been successfully been loaded through call loadGUI()
where windowName is a window that exists in the loaded layout. eventName is a legal event name (see note bellow).
functionName is a python string that will be executed on call. This is must be a defined function or
method. This must be accessable from the __init__.py file meaning that if the call is callable from the setup
script then it will be successfully called (raises an python error otherwise)
args is the python (functionName) method or function arguments passed to the function. Can be dictionary string.
Raises GUIError on failure.
The type of events are dependent on the widget called. Subscribing an event to a widget that does not support it will
crash the system. To get a list of events supported you can look at the following links. The public attributes has a list of events
that will be called. Usually you can use the name preciding 'Event'. For instance button has an name EventClicked, you call with event
name being 'Clicked'.
button (http://www.cegui.org.uk/api_reference/classCEGUI_1_1PushButton.html#65d6b712c449f7a9244dcc9d40c0d825)
window (http://www.cegui.org.uk/api_reference/classCEGUI_1_1Window.html)
editbox (http://www.cegui.org.uk/api_reference/classCEGUI_1_1Editbox.html)
Bug: Subscribing an event to a widget that does not support it will crash the system

addItem(self, windowName, itemName)
Appends item last to a window. Widget exists and is of type TaharezLook/Combobox, TaharezLook/Listbox
Item is added to the widget, windowName is the name of the widget and itemName is the name
of the item added (as displayed in window). Raises GUIError on failure

addLayout(self, fileName)
Loads a legal CEGUI formated layout where fileName is name of the file and
must exist be accessable in resource paths.
Passing no fileName will unload the previusly loaded layout.

call(self, windowName, func, arg)
Used in Hikari interface to forward the call to the flash window. Returns None on failure.
Where func is the defined external function in the flash control and arg is a string.

clearItems(self, windowName)
Clears items from a window. Widget exists and is of type TaharezLook/Combobox, TaharezLook/Listbox
Item is added to the widget, windowName is the name of the widget. Raises GUIError on failure

delLayout(self, windowName)
Loads a legal CEGUI formated layout where fileName is name of the file and
must exist be accessable in resource paths.
Passing no fileName will unload the previusly loaded layout.

getCheckboxSelected(self, windowName)
Returns selected items from a window. Widget exists and is of type TaharezLook/Checkbox
Returns True/False. Raises GUIError on failure

getProperty(self, windowName, PropertyName)
Direct access to gui widgets window property in a loaded layout.
Refer to CEGUI reference at http://www.cegui.org.uk/api_reference/
or http://www.cegui.org.uk/api_reference/namespaceCEGUI_1_1WindowProperties.html
Raises GUIError if windowName does not exist in current layout or property is
not present in this widget.

getSelectedItems(self, windowName)
Returns selected items from a window. Widget exists and is of type TaharezLook/Combobox, TaharezLook/Listbox
The selected items are returned in a list. Raises GUIError on failure

getText(self, windowName)
Gets a gui text value in a loaded layout having window with name windowName.
Raises GUIError if windowName does not exist in current layout.

loadImageSet(self, fileName)
Loads an image set from file. The fileName is the name of a file loaded through the ogre resource manager.
If the imageset is loaded we dont do anything. If it is not found we notify failure.
Raises GUIError on failure.

loadLayout(self, fileName, w=256, h=256)
Loads a legal CEGUI formated layout where fileName is name of the file and
must exist be accessable in resource paths.
Passing no fileName will unload the previusly loaded layout.

setPosition(self, windowName, x, y)

setProperty(self, windowName, PropertyName, PropertyValue)
Direct access to gui widgets window property in a loaded layout.
Refer to CEGUI reference at http://www.cegui.org.uk/api_reference/
or http://www.cegui.org.uk/api_reference/namespaceCEGUI_1_1WindowProperties.html
Raises GUIError if windowName does not exist in current layout or property is
not present in this widget.

For Hikari
play (start, stop, loop, rewind)
transparent (bool) : uses alpha channel
show (bool) : hides/shows window
opacity (real) : sets window opacity

setText(self, windowName, text)
Sets a gui text value in a loaded layout having window with name windowName.
Raises GUIError if windowName does not exist in current layout.

Data

__author__ = 'Kjartan A Jonsson'
__copyright__ = 'Copyright (c) 2007 aGameCompany'
__license__ = 'aGameCompany'
__source__ = '$Source: /cvsroot/game-client-system/scripts/GUI.py,v $'
__version__ = '$Revision: 1.13 $'
systemGUI = <GUI.SystemGUI instance at 0x00B98CD8>

Author

Kjartan A Jonsson

sound (version 1.4)

index
d:\cruisecontrol\game-client-system\game\common\scripts\sound.py

Sound library wrapper

Notes on how animation and materials (collisions) sounds work:
We create and define sounds using the sound objects. And we can assign it as Material and/or Animation.
If this is done we will play the sound depending on what was assigned. If we assign it as Animation we will play the sound wh
en the animation is started.
If it was defined as Material we will play it as default sound on collision between objects.
A Material can also be assigned more than one sound and is done though a call where we define two Materials for each sound.
This will be played when these two objects collide with each other.

Some notes on Animation:
When playing an animation it is best to define the sound object to loop if we have an looping animation. Also if we
do not stop the current animation the sound will not stop. This means that before changing the animation we need to stop
the current playing animation so the sound stops correctly. This is handled currently by entity.setAnimation. It calls
stop animation if we have played one before.

Modules

gameSound

Classes

exceptions.Exception(exceptions.BaseException)
SoundError
Sound

class **Sound**

Methods defined here:
__init__(self, name, file, loop)
where id is the id of our camera, if camera id is 0 then we are accessing a new instance of default camera
attachToSceneNode(self, sceneNode)
destroy(self)
getFile(self)
getName(self)
play(self)
setAnimationSound(self, entity, animationName)
Will play this sound with specified sound settings when animtion is played through the python interface
setCone(self, innerAnge, outerAngle, outerGain)
setDistance(self, referenceRad, maxRad)
setGain(self, gain, gmin, gmax)
setMaterialSound(self, materialA, materialB=None)
Will play this sound with specified sound settings when materialA collides with materialB.
if materialB is not set we use this sound as the default collision sound for materialA.
setPitch(self, pitch)
stop(self)

Data and other attributes defined here:
sceneNode = None

Author

Kjartan A Jonsson

SceneManager (version 1.64)

index
d:\cruisecontrol\game-client-system\game\common\scripts\scenemanager.py

```
This module behaves as a singleton allowing calls to the game graphics system.
It will attempt to clean upp all the created objects before shutdown calling all
subsequent lighs, organics, entities, etc .. created to be deleted.
```

Modules

gameGraphics

Classes

Animation
AnimationTrack
Camera
KeyFrame
MovableObject
    Entity
    Light
    Particles
Organic
TextureBlitter
exceptions.Exception(exceptions.BaseException)
    SceneError
    AnimationError
    CameraError
    EntityError
    LightError
    MovableObjectError
    OrganicError
    ParticleError
    TextureBlitterError

class **Animation**

```
Animation class is a placeholder for animating movable objects. Many movable objects can be
animated by the animation class by adding them using Animation.addTrack (...) and supplying
the class of the object extended by MovableObject class.
```

Methods defined here:

__init__(self, name, length)
```
This class should not be instanced directly use SceneManager.addAnimation(...)
```
addTrack(self, movableObject)
```
Creates a new AnimationTrack automatically associated with a object extending MovableObject 'movableObject'
This method creates a standard AnimationTrack, but also associates it with a target MovableObject which wil
Returns the instance of the create AnimationTrack class that knows its index. Where the index is a positive
```
pause(self)
```
Pauses the animation.
```
play(self, bLoop=True)
```
Plays the animation. If bLoop equals true then we loop the animation
```
setInterpolationMode(self, bUsingInterpolation=True)
```
Creating animation class we use interpolation by default. This is changed if bUsingInterpolation is False,
then we will use Spline to interpolate the animation frames.
```
setWeight(self, w)
```
Sets the weight (influence) of this animation.
```
stop(self)
```
Stops the animation.
```

Data and other attributes defined here:

name = None

class **AnimationTrack**

```
AnimationTrack is the class that holds a many KeyFrames. The animation track handles the animation
for the supplied movable object.
Use Animation class to create animation tracks. Animation can have zero to many AnimationTracks.
To remove animation track call SceneManager.delAnimation(...).
```

Methods defined here:

__init__(self, index, attrib, movableObject)
```
This class should not be instanced directly use SceneManager.addAnimation(...) and then Animation.addTrack(
```
addKeyFrame(self, timePos)
```
Creates a key frame for animation of the movable object. Where tiemPos must be a positive integer value rep
It is better to create KeyFrames in time order. Creating them out of order can result in expensive reorderi
Note that a KeyFrame at time index 0.0 is always created for you, so you don't need to create this one.
Note that the timePos must also be lower than the total time of animation (animation lenght).
```

KeyFrame is used to define the actual scale, translation and rotation in an animation.
Returns instance of KeyFrame on success but raises AnimationError on failure.

Data and other attributes defined here:
index = None

class **Camera**

Request for refactoring. Currenlty we handle the camera creation and cleanup. Ogre should do it instead.

Methods defined here:

__init__(self)
Is meant to give access to the one default camera

attach(self, entityName=None)
This will attach the camera to an entity having name entityName, the entity must exist. If no entity is defined then the camera is detached and "free".
When attached you can still control the camera but in the entitys local space. You will be able to set lookAt and lookAtEntity where the camera will follow the attached entity and look at the targets.
Calling followEntity will detach the camera from the object (restoring it).
Deleting an entity havning a camera attached will delete the default camera :(.

disable(self)
Will remove any behavoirs on camera.

followEntity(self, name='')
Where name is null or a loaded movable object. Critical: pass null value (disabling) before unloading the entity. So we are not following at the deleted entity (will cause crash).

lookAtDirection(self, x, y, z)
Turns the camera to the specified direction

lookAtEntity(self, name='')
Where name is null or a loaded movable object. Critical: pass null value (disabling) before unloading the entity. So we are not looking at the deleted entity (will cause crash).

setFOV(self, fov=0.80000000000000004)
Sets field of view, 0.8 is a good value

setFlyFreeControl(self, moveScale=None, rotationScale=None)
Will enable fly free camera control for camera. If moveScale and rotationScale is provided these values will be used in transition per unit.

setFlyFreeControlKeys(self, MoveLeft, MoveRight, MoveForward, MoveBackward, MoveUp, MoveDown, RotateLeft, RotateRight)
Sets what keys are used when we control the camera in free fly mode.

setFollowProperties(self, followFactor, followRotationFactor, height, distance, lookAhead=100)
Applies the follow scene node factors. How the scene node is followed.

setOrientation(self, w, x, y, z)
Rotates the camera

setPosition(self, x, y, z)
Positions the camera

setTargetEntity(self, name='')
Where name is null or a loaded movable object. Critical: pass null value (disabling) before unloading the entity. So we are not looking at the deleted entity (will cause crash).

useCamera(self, cameraName=None)
This will change the camera from the current to the specified 'cameraName'. The camera named 'cameraName' must exist in the system (you can create through loading external scenes, like through call loadScene(...))
Critical: pass null value (disabling) before unloading the scene that created the camera.
Passing None (or '') will select the default camera.
If the camera does not exist we raise an SceneError.

class **Entity**(MovableObject)

Handles a mesh, use SceneManager addEntity instead of calling this directly

Methods defined here:

__init__(self, name, file=None)
This class should not be instanced directly use SceneManager.addEntity(...)

__str__(self)
Formated string describing name, file, animation names and position of the Entity

getAllAnimationNames(self)
Returns a list of all the animations supported by this entity.

playAnimation(self, name, loop=False, enabled=True)
Plays entity animation and returns None on success. Else raises an EntityError.

setMaterial(self, MaterialName)
Sets this entites material. Replaces the material currently in use.

stopAnimation(self, name)
Plays entity animation and returns None on success. Else raises an EntityError.

Data and other attributes defined here:
file = ''

Methods inherited from MovableObject:

attach(self, movableobject, position=[])
Attaches the 'movableobjectname' to this movable object. The object must exists and be loaded into ogre.
When attaching an object it will detach the objects parent scene node from its parent. Then it will be attached to this objects parent scene node (inheriting its translations + applying its current transitio
Cameras are not supported.

getChildren(self)

getDistanceTo(self, toObject)
Returns the distance to the object 'toObject'

getOrientation(self, world=False)

Returs the movable objects orientation in the form of list having four values being [w, x, y, z].
If world is true then we will get the orientation after all applied orientation to parents.
If world is false then we get the orientation in relation to parent.

getPitch(self)
Returns movable objects 'degree' around x axis

getPosition(self, world=False)
Returns movable objects position in the form of a list having three float values [x, y, z]
If world is true then we will get the final position after all applied translation to parents.
If world is false then we get the translation in relation to parent.

getRoll(self)
Returns movable objects 'degree' around z axis

getScale(self)
Returns movable objects scale in the form of a list having three float values [x, y, z]

getSize(self)
Returns movable objects size in the form of a list having three float values [x, y, z] after scale
(the bounding box size).

getYaw(self)
Returns movable objects 'degree' around y axis

rotate(self, w, x, y, z, time=None, args=None)
Rotates an object. Behaves exactly as setOrientation

scale(self, x, y, z)
Scales relative to current scale

setOrientation(self, w, x, y, z, time=None, args=None)
Sets the orientation of our object. If time is set then we interpolate linearly from current orientation to
during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a cubic (Ogre::Quaternion::Squad) funct
values each having four values that will be used to calculate the interpolation. The arg format is as follo
Method raises MovableObjectError on failure and None on success.

setPitch(self, degree)
Rotates object 'degree' around x axis

setPosition(self, x, y, z, type='Absolute', time=None, args=None)
Sets the position of our object, where type is either 'Absolute' or 'Relative'. Where 'Relative' uses the o
and sets the position relative to that. Absolute moves the object in worlds space. If time is set then we i
location to the desired (x, y, z) position during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a spline (Ogre::SimpleSpline) function.
list values each having three values that will be used to calculate the interpolation. The arg format is as
Method raises MovableObjectError on failure and None on success.

setRoll(self, degree)
Rotates object 'degree' around z axis

setScale(self, x, y, z, relative=False)
Scales relative to current scale

setVisible(self, b=True, cascade=True)
if b is True then we set this object visable and all its children

setYaw(self, degree)
Rotates object 'degree' around y axis

showBoundingbox(self, b=True)
if b is true then we display this object bounding box

translate(self, x, y, z, time=None, args=None)
Moves movable object relative to its parent

Data and other attributes inherited from MovableObject:
name = None

class **KeyFrame**

```
KeyFrame is the class that positions an animated movable object by setting the subsiquent transition, rotation or sca
specified time position.
Use AnimationTrack class to create key frames. AnimationTrack can have zero to many key frames.
To remove animation track call SceneManager.delAnimation(...).
```

Methods defined here:

__init__(self, id, attrib, timePos)
This class should not be instanced directly use SceneManager.addAnimation(...) and then Animation.addTrack(
and lastly AnimationTrack.addKeyFrame(...)

setRotation(self, q, x, y, z)
Sets the rotation applied by this keyframe. Returns None on success but raises AnimationError on failure

setScale(self, x, y, z)
Sets the scaling factor applied by this keyframe to the animable object at it's time index. Returns None o
but raises AnimationError on failure

setTranslate(self, x, y, z)
Sets the translation associated with this keyframe. Returns None on success but raises AnimationError on fa

---

Data and other attributes defined here:
id = 0

class **Light**(MovableObject)

```
##############################################
#    Light
#       Representation of a dynamic light source in the scene
##############################################
```

Methods defined here:

__init__(self, name)
This class should not be instanced directly use SceneManager.addLight(...)

__str__(self)
Formated string describing name, file, animation names and position of the Light

setDiffuseColor(self, r, g, b)

setDirection(self, x, y, z)
Sets the light to directional light mode and direction in which a light points.

setFalloffRange(self, innerAngle, outerAngle, falloff)
Sets the light to spotlight mode.
Sets the falloff between the inner and outer cones of the spotlight.
Where:
innerAngle is the angle in degrees covered by the bright inner cone The inner cone applicable only to D
outerAngle is the angle in degrees covered by the outer cone
falloff is the rate of falloff between the inner and outer cones. 1.0 means a linear falloff, less mean

---

Methods inherited from MovableObject:

attach(self, movableobject, position=[])
Attaches the 'movableobjectname' to this movable object. The object must exists and be loaded into ogre.
When attaching an object it will detach the objects parent scene node from its parent. Then it will
be attached to this objects parent scene node (inheriting its translations + applying its current transitio
Cameras are not supported.

getChildren(self)

getDistanceTo(self, toObject)
Returns the distance to the object 'toObject'

getOrientation(self, world=False)
Returs the movable objects orientation in the form of list having four values being [w, x, y, z].
If world is true then we will get the orientation after all applied orientation to parents.
If world is false then we get the orientation in relation to parent.

getPitch(self)
Returns movable objects 'degree' around x axis

getPosition(self, world=False)
Returns movable objects position in the form of a list having three float values [x, y, z]
If world is true then we will get the final position after all applied translation to parents.
If world is false then we get the translation in relation to parent.

getRoll(self)
Returns movable objects 'degree' around z axis

getScale(self)
Returns movable objects scale in the form of a list having three float values [x, y, z]

getSize(self)
Returns movable objects size in the form of a list having three float values [x, y, z] after scale
(the bounding box size).

getYaw(self)
Returns movable objects 'degree' around y axis

rotate(self, w, x, y, z, time=None, args=None)
Rotates an object. Behaves exactly as setOrientation

scale(self, x, y, z)
Scales relative to current scale

setOrientation(self, w, x, y, z, time=None, args=None)
Sets the orientation of our object. If time is set then we interpolate linearly from current orientation to
during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a cubic (Ogre::Quaternion::Squad) funct
values each having four values that will be used to calculate the interpolation. The arg format is as follo
Method raises MovableObjectError on failure and None on success.

setPitch(self, degree)
Rotates object 'degree' around x axis

setPosition(self, x, y, z, type='Absolute', time=None, args=None)
Sets the position of our object, where type is either 'Absolute' or 'Relative'. Where 'Relative' uses the o
and sets the position relative to that. Absolute moves the object in worlds space. If time is set then we i
location to the desired (x, y, z) position during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a spline (Ogre::SimpleSpline) function.
list values each having three values that will be used to calculate the interpolation. The arg format is as
Method raises MovableObjectError on failure and None on success.

setRoll(self, degree)
Rotates object 'degree' around z axis

setScale(self, x, y, z, relative=False)
Scales relative to current scale

setVisible(self, b=True, cascade=True)
if b is True then we set this object visable and all its children

setYaw(self, degree)
Rotates object 'degree' around y axis

showBoundingbox(self, b=True)
if b is true then we display this object bounding box

translate(self, x, y, z, time=None, args=None)
Moves movable object relative to its parent

Data and other attributes inherited from MovableObject:

name = None

class **MovableObject**

Class representing a primitive object in the scene graph, is never used through direct initalization

Methods defined here:

__init__(self)
Abstract class initalization call that raises an error when called directly

__str__(self)
Formated string describing name and position of the object

attach(self, movableobject, position=[])
Attaches the 'movableobjectname' to this movable object. The object must exists and be loaded into ogre.
When attaching an object it will detach the objects parent scene node from its parent. Then it will
be attached to this objects parent scene node (inheriting its translations + applying its current transitio
Cameras are not supported.

getChildren(self)

getDistanceTo(self, toObject)
Returns the distance to the object 'toObject'

getOrientation(self, world=False)
Returs the movable objects orientation in the form of list having four values being [w, x, y, z].
If world is true then we will get the orientation after all applied orientation to parents.
If world is false then we get the orientation in relation to parent.

getPitch(self)
Returns movable objects 'degree' around x axis

getPosition(self, world=False)
Returns movable objects position in the form of a list having three float values [x, y, z]
If world is true then we will get the final position after all applied translation to parents.
If world is false then we get the translation in relation to parent.

getRoll(self)
Returns movable objects 'degree' around z axis

getScale(self)
Returns movable objects scale in the form of a list having three float values [x, y, z]

getSize(self)
Returns movable objects size in the form of a list having three float values [x, y, z] after scale
(the bounding box size).

getYaw(self)
Returns movable objects 'degree' around y axis

rotate(self, w, x, y, z, time=None, args=None)
Rotates an object. Behaves exactly as setOrientation

scale(self, x, y, z)
Scales relative to current scale

setOrientation(self, w, x, y, z, time=None, args=None)
Sets the orientation of our object. If time is set then we interpolate linearly from current orientation to
during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a cubic (Ogre::Quaternion::Squad) funct
values each having four values that will be used to calculate the interpolation. The arg format is as follo
Method raises MovableObjectError on failure and None on success.

setPitch(self, degree)
Rotates object 'degree' around x axis

setPosition(self, x, y, z, type='Absolute', time=None, args=None)
Sets the position of our object, where type is either 'Absolute' or 'Relative'. Where 'Relative' uses the o
and sets the position relative to that. Absolute moves the object in worlds space. If time is set then we i
location to the desired (x, y, z) position during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a spline (Ogre::SimpleSpline) function.
list values each having three values that will be used to calculate the interpolation. The arg format is as
Method raises MovableObjectError on failure and None on success.

setRoll(self, degree)
Rotates object 'degree' around z axis

setScale(self, x, y, z, relative=False)
Scales relative to current scale

setVisible(self, b=True, cascade=True)
if b is True then we set this object visable and all its children

setYaw(self, degree)
Rotates object 'degree' around y axis

showBoundingbox(self, b=True)
if b is true then we display this object bounding box

translate(self, x, y, z, time=None, args=None)
Moves movable object relative to its parent

Data and other attributes defined here:
name = None

class **Organic**

```
##############################################
#    Organic
#        A object that interacts with its enviroment
##############################################
```

Methods defined here:
__init__(self, name, type, dic)
This class should not be instanced directly use SceneManager.addOrganic(...)

__str__(self)

getPropertyKeys(self, type)
Returns all the keys needed to be set by the specified type to be a successfull call to the system.

getType(self)
Returns the Organic type. The supported types are Grass and WoblyMesh

class **Particles**(MovableObject)

```
Class defining particle system based special effects
```

Methods defined here:
__init__(self, name, effectName)

This class should not be instanced directly use SceneManager.addParticles(...)

Methods inherited from MovableObject:

__str__(self)
Formated string describing name and position of the object

attach(self, movableobject, position=[])
Attaches the 'movableobjectname' to this movable object. The object must exists and be loaded into ogre.
When attaching an object it will detach the objects parent scene node from its parent. Then it will
be attached to this objects parent scene node (inheriting its translations + applying its current transitio
Cameras are not supported.

getChildren(self)

getDistanceTo(self, toObject)
Returns the distance to the object 'toObject'

getOrientation(self, world=False)
Returs the movable objects orientation in the form of list having four values being [w, x, y, z].
If world is true then we will get the orientation after all applied orientation to parents.
If world is false then we get the orientation in relation to parent.

getPitch(self)
Returns movable objects 'degree' around x axis

getPosition(self, world=False)
Returns movable objects position in the form of a list having three float values [x, y, z]
If world is true then we will get the final position after all applied translation to parents.
If world is false then we get the translation in relation to parent.

getRoll(self)
Returns movable objects 'degree' around z axis

getScale(self)
Returns movable objects scale in the form of a list having three float values [x, y, z]

getSize(self)
Returns movable objects size in the form of a list having three float values [x, y, z] after scale
(the bounding box size).

getYaw(self)
Returns movable objects 'degree' around y axis

rotate(self, w, x, y, z, time=None, args=None)
Rotates an object. Behaves exactly as setOrientation

scale(self, x, y, z)
Scales relative to current scale

setOrientation(self, w, x, y, z, time=None, args=None)
Sets the orientation of our object. If time is set then we interpolate linearly from current orientation to
during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a cubic (Ogre::Quaternion::Squad) funct
values each having four values that will be used to calculate the interpolation. The arg format is as follo
Method raises MovableObjectError on failure and None on success.

setPitch(self, degree)
Rotates object 'degree' around x axis

setPosition(self, x, y, z, type='Absolute', time=None, args=None)
Sets the position of our object, where type is either 'Absolute' or 'Relative'. Where 'Relative' uses the o
and sets the position relative to that. Absolute moves the object in worlds space. If time is set then we i
location to the desired (x, y, z) position during time 'time'. Where time is in milli seconds.
If arg is set then we interpolate using the arg values as input to a spline (Ogre::SimpleSpline) function.
list values each having three values that will be used to calculate the interpolation. The arg format is as
Method raises MovableObjectError on failure and None on success.

setRoll(self, degree)
Rotates object 'degree' around z axis

setScale(self, x, y, z, relative=False)
Scales relative to current scale

setVisible(self, b=True, cascade=True)
if b is True then we set this object visable and all its children

setYaw(self, degree)
Rotates object 'degree' around y axis

showBoundingbox(self, b=True)
if b is true then we display this object bounding box

translate(self, x, y, z, time=None, args=None)
Moves movable object relative to its parent

Data and other attributes inherited from MovableObject:

name = None

class **TextureBlitter**

Methods defined here:

__del__(self)
Destroys the texture blitter.

__init__(self, textureName, fileName, resourceGroup, width, height)
Creates the instance and texture file. The textureName will be the name refered by material scripts and
file name is the name of the file which we start from (this can be an empty image (64x64). The fileName mus
be accessable from the resource group 'resourceGroup'. The width and height must be larger than 0 and repre
the final images size.
This instance must be destroyed before exiting the application. Otherwise it causes a crash during unload.

__str__(self)

addSource(self, name, x, y, w, h, texture, resourceGroup='General', flipX=False, flipY=False)
Adds a sticker source definition to the collection and sets its dimensions.
Where texture is the name of the texture file and x, y, w, h is the location within the
texture where we take the image.
If flipX is true then it will load this image and flip it in the x axis.
If flipY is true then it will load this image and flip it in the y axis.

addTarget(self, name, x, y, w, h)
Adds a sticker location to the collection and sets its dimensions.
Where targetName is the name to identify this location. x and y is the relative
location on the bitmap (a number between 0 and 1) and w and h is the relative
width and height of the target image.

paint(self, sourceName, targetName)
Will set the sticker onto target location.

Data and other attributes defined here:
height = None
id = None
name = None
resourceGroup = None
width = None

Functions

addAnimation(name, length=None)
Creates an animation which can be used to animate movable objects. Passing length=None tells the system to use a
Remarks:
An animation is a collection of 'tracks' which over time change the position / orientation of Node objects. In t
likely have tracks to modify the position / orientation of SceneNode objects, e.g. to make objects move along a
You don't need to use an Animation object to move objects around - you can do it yourself using the methods of t
However, when you need relatively complex scripted animation, this is the class to use since it will interpolate
generally make the whole process easier to manage.
A single animation can affect multiple Node objects (each AnimationTrack affects a single Node). In addition, th
can be affected by multiple animations, athough this is more useful when performing skeletal animation (see Skel
Note that whilst it uses the same classes, the animations created here are kept separate from the skeletal anima
Parameters:
name is the name of the animation, should be unique within it's parent (e.g. Skeleton)
length is the length of the animation in seconds. Passing None tels the system that we are going to use an alrea
Returns instance of the created Animation class on success but raises SceneError on failure.

addBlitterTexture(textureName, fileName, resourceGroup, width, height)
Creates an blitter texture and adds it to our blitter texture collection

addEntity(name, file)
Adds a entity with name 'name' using mesh resource file 'file'.
The entity can only be removed by calling SceneManager.delEntity(<name>).
Returns a instance on success or raises an SceneError if it already exists or anything else went wrong.

addGrass()

addLight(name, type=0)
Adds a light with name 'name'. The light is returned on success else SceneError is raised.
The light can only be removed by calling SceneManager.delLight(<name>).
Returns a instance on success or raises an SceneError if it already exists or anything else went wrong.
Different types of light include the following
* Point light sources give off light equally in all directions, so require only position not direction.
* Directional lights simulate parallel light beams from a distant source, hence have direction but no position.
* Spotlights simulate a cone of light from a source so require position and direction, plus extra values for fal
By default adding a light is a point light. Setting a direction by calling setDirection converts it to a directi
If setting a fall off using the method setFalloffRange (...) then converts it to a spot light.

addOrganics(name, type, dic)
Adds a organic with name 'name' having type 'type'. Where type is either 'Grass' or 'WoblyMesh'.
The organic using the following attributes passed as a dictionary:
Grass => "GrassWidth", "GrassHeight", "Material", "Radius", "EdgeDistribution", "PositionX", "PositionZ"
WoblyMesh => "MeshName","px","py","pz","sz","sy","sz","ow","ox","oy","oz","Displacement","Density","TimeDensity"
The entity is returned on success else SceneError is raised. The entity can only be removed by calling SceneMana
Returns a instance on success or raises an SceneError if it already exists or anything else went wrong

addParticles(name, file)
Adds a particle system named 'name' base on file 'file' where the file is of type .particle and is available in
Returns a instance on success or raises an SceneError if it already exists or anything else went wrong.

addTrees()

delAnimation(name)
Destroys an animation with name 'name'. The entire tracks and keyframes will be removed from the system. Any mov
Returns None on success but raises SceneError on failure.

delBlitterTexture(name)
Will remove and destroy the blitter from our collection

delEntity(name)
Removes a entity with name 'name' from the system. Must have been created using call addEntity (...).
Returns None on success or raises an SceneError if it does not exist

delLight(name)
Removes a light with name 'name' from the system. Must have been created using call addLight (...)
Returns None on success or raises an SceneError if it does not exist

delOrganics(name)
Removes a organics with name 'name' from the system. Must have been created using call addOrganics (...).
Returns None on success or raises an SceneError if it does not exist or system failed to remove it

delParticles(name)
Removes a particles with name 'name' from the system. Must have been created using call addParticles (...)
Returns None on success or raises an SceneError if it does not exist

getAnimation(name)
Returns an animation having name 'name'. Raises SceneError if the animation does not exist.

getAnimationNames()
Returns a list of all lights currently loaded in the system

getBlitterTextureNames()
Return all the names of our created blitter textures

getCamera()
Returns the default camera created by the system.

getContentQuery(filter=None)
Returns a dictionary on the scene manager content. Current supported filters are:
None : returns everything
The returned dictionary has keys 'Light', 'Camera', 'Animation', 'Entity'

getEntity(name)
Returns the instance of entity having name 'name'. Returns a instance or raises an SceneError if it does not exi

getEntityNames()
Returns a list of all entities currently loaded in the system. Returns a list.

getLight(name)
Returns the instance of entity having name 'name'

getLightNames()
Returns a list of all lights currently loaded in the system

getOrganicNames()
Returns a list of all organics currently loaded in the system. Returns a list

getOrganics(name)
Returns the instance of organic having name 'name'. Returns a instance or raises an SceneError if it does not ex

getParticleNames()
Returns a list of all particles currently loaded in the system. Returns a list

getParticles(name)
Returns the instance of particles having name 'name'. Returns a instance or raises an SceneError if it does not

hasAnimation(name)
Returns true if the animation with name 'name' exists

hasEntity(name)
Returns a true if entity with name 'name' exists.

hasLight(name)
Returns a true if light with name 'name' exists

hasOrganics(name)
Returns a true if organics with name 'name' exists

hasParticles(name)
Returns true if the particles with name 'name' exists

loadScene(fileName=None)
Loads a scene of supported formats. The files must be accessable to the Ogre resource loader.
Supported types are .osm (oFusion files created through the oFusion max plugin for Ogre3d).
Passing None or '' will unload the prevously loaded scene. Only one scene can be loaded at a time.
Raises SceneError on failure.

setAmbientLight(r, g, b)
Sets the scenes ambient light where r,g,b are values between 1 and 0. Returns None

setCompostionEffect(effect)
Sets the composition effect (last processing of the image) to effect where effect names can be one of defined ef

setFog(r, g, b, density=0.001, fogmode=3)
Sets fog in the scene where fogmodes are (None=0,exp=1,exp2=2,linear=3)

setShadowDistance(far, fadestart=0, fadeend=0)
Sets the shadow distance where

setSkyBox(strMaterialName=None, bEnable=True, scale=50)
Sets the scenes skybox. strMaterialName is a loaded resource material defined as a cube texture. Returns None
If strMaterialName is set to an valid material and bEnable is true then we will texture the sky.
If strMaterialName is set to None and bEnable is true then we will do a procedural sky.
If strMaterialName is set to None and bEnable is false then we will make the sky black.

Data

BTN_0 = '0'
BTN_1 = '1'
BTN_2 = '2'
BTN_3 = '3'
BTN_4 = '4'
BTN_5 = '5'
BTN_6 = '6'
BTN_7 = '7'
BTN_8 = '8'
BTN_9 = '9'
BTN_A = 'A'
BTN_ABNT_C1 = 'ABNT_C1'
BTN_ABNT_C2 = 'ABNT_C2'
BTN_ADD = 'ADD'
BTN_APOSTROPHE = 'APOSTROPHE'
BTN_APPS = 'APPS'
BTN_AT = 'AT'
BTN_AX = 'AX'
BTN_B = 'B'
BTN_BACK = 'BACK'
BTN_BACKSLASH = 'BACKSLASH'
BTN_C = 'C'
BTN_CALCULATOR = 'CALCULATOR'
BTN_CAPITAL = 'CAPITAL'
BTN_COLON = 'COLON'
BTN_COMMA = 'COMMA'
BTN_CONVERT = 'CONVERT'
BTN_D = 'D'
BTN_DECIMAL = 'DECIMAL'
BTN_DELETE = 'DELETE'

BTN_DIVIDE = 'DIVIDE'
BTN_DOWN = 'DOWN'
BTN_E = 'E'
BTN_END = 'END'
BTN_EQUALS = 'EQUALS'
BTN_ESCAPE = 'ESCAPE'
BTN_F = 'F'
BTN_F1 = 'F1'
BTN_F10 = 'F10'
BTN_F11 = 'F11'
BTN_F12 = 'F12'
BTN_F13 = 'F13'
BTN_F14 = 'F14'
BTN_F15 = 'F15'
BTN_F2 = 'F2'
BTN_F3 = 'F3'
BTN_F4 = 'F4'
BTN_F5 = 'F5'
BTN_F6 = 'F6'
BTN_F7 = 'F7'
BTN_F8 = 'F8'
BTN_F9 = 'F9'
BTN_G = 'G'
BTN_GRAVE = 'GRAVE'
BTN_H = 'H'
BTN_HOME = 'HOME'
BTN_I = 'I'
BTN_INSERT = 'INSERT'
BTN_J = 'J'
BTN_K = 'K'
BTN_KANA = 'KANA'
BTN_KANJI = 'KANJI'
BTN_L = 'L'
BTN_LBRACKET = 'LBRACKET'
BTN_LCONTROL = 'LCONTROL'
BTN_LEFT = 'LEFT'
BTN_LMENU = 'LMENU'
BTN_LSHIFT = 'LSHIFT'
BTN_LWIN = 'LWIN'
BTN_M = 'M'
BTN_MAIL = 'MAIL'
BTN_MEDIASELECT = 'MEDIASELECT'
BTN_MEDIASTOP = 'MEDIASTOP'
BTN_MINUS = 'MINUS'
BTN_MULTIPLY = 'MULTIPLY'
BTN_MUTE = 'MUTE'
BTN_MYCOMPUTER = 'MYCOMPUTER'
BTN_N = 'N'
BTN_NEXTTRACK = 'NEXTTRACK'
BTN_NOCONVERT = 'NOCONVERT'
BTN_NUMLOCK = 'NUMLOCK'
BTN_NUMPAD0 = 'NUMPAD0'
BTN_NUMPAD1 = 'NUMPAD1'
BTN_NUMPAD2 = 'NUMPAD2'
BTN_NUMPAD3 = 'NUMPAD3'
BTN_NUMPAD4 = 'NUMPAD4'
BTN_NUMPAD5 = 'NUMPAD5'
BTN_NUMPAD6 = 'NUMPAD6'
BTN_NUMPAD7 = 'NUMPAD7'
BTN_NUMPAD8 = 'NUMPAD8'
BTN_NUMPAD9 = 'NUMPAD9'
BTN_NUMPADCOMMA = 'NUMPADCOMMA'
BTN_NUMPADENTER = 'NUMPADENTER'
BTN_NUMPADEQUALS = 'NUMPADEQUALS'
BTN_O = 'O'
BTN_OEM_102 = 'OEM_102'
BTN_P = 'P'
BTN_PAUSE = 'PAUSE'
BTN_PERIOD = 'PERIOD'
BTN_PGDOWN = 'PGDOWN'
BTN_PGUP = 'PGUP'
BTN_PLAYPAUSE = 'PLAYPAUSE'
BTN_POWER = 'POWER'
BTN_PREVTRACK = 'PREVTRACK'
BTN_Q = 'Q'

BTN_R = 'R'
BTN_RBRACKET = 'RBRACKET'
BTN_RCONTROL = 'RCONTROL'
BTN_RETURN = 'RETURN'
BTN_RIGHT = 'RIGHT'
BTN_RMENU = 'RMENU'
BTN_RSHIFT = 'RSHIFT'
BTN_RWIN = 'RWIN'
BTN_S = 'S'
BTN_SCROLL = 'SCROLL'
BTN_SEMICOLON = 'SEMICOLON'
BTN_SLASH = 'SLASH'
BTN_SLEEP = 'SLEEP'
BTN_SPACE = 'SPACE'
BTN_STOP = 'STOP'
BTN_SUBTRACT = 'SUBTRACT'
BTN_SYSRQ = 'SYSRQ'
BTN_T = 'T'
BTN_TAB = 'TAB'
BTN_U = 'U'
BTN_UNDERLINE = 'UNDERLINE'
BTN_UNLABELED = 'UNLABELED'
BTN_UP = 'UP'
BTN_V = 'V'
BTN_VOLUMEDOWN = 'VOLUMEDOWN'
BTN_VOLUMEUP = 'VOLUMEUP'
BTN_W = 'W'
BTN_WAKE = 'WAKE'
BTN_WEBBACK = 'WEBBACK'
BTN_WEBFAVORITES = 'WEBFAVORITES'
BTN_WEBFORWARD = 'WEBFORWARD'
BTN_WEBHOME = 'WEBHOME'
BTN_WEBREFRESH = 'WEBREFRESH'
BTN_WEBSEARCH = 'WEBSEARCH'
BTN_WEBSTOP = 'WEBSTOP'
BTN_X = 'X'
BTN_Y = 'Y'
BTN_YEN = 'YEN'
BTN_Z = 'Z'
PATH_LOG = ''
REG_CALLBACK_INTERSECTION = 100
REG_CALLBACK_NETWORK = 105
REG_CALLBACK_REGISTERED = 106
REG_CALLBACK_SYSTEM = 104
REG_CALLBACK_TIMER = 101
REG_SYSTEM_SETUP = 200
SYSTEMEVENT_LOAD = 10
SYSTEMEVENT_UNLOAD = 11
__author__ = 'Kjartan A Jonsson'
__copyright__ = 'Copyright (c) 2007 aGameCompany'
__license__ = 'aGameCompany'
__source__ = '$Source: /cvsroot/game-client-system/scripts/SceneManager.py,v $'
__version__ = '$Revision: 1.64 $'
false = 0
true = 1

Author

Kjartan A Jonsson

PhysicsManager (version 1.41)

index
d:\cruisecontrol\game-client-system\game\common\scripts\physicsmanager.py

```
This module behaves as a singleton allowing calls to the game physics system
```

Modules

SceneManager gameGraphics gamePhysics

Classes

Tool
Vehicle
SceneManager.Entity(SceneManager.MovableObject)
    Body
exceptions.Exception(exceptions.BaseException)
    PhysicsError
    BodyError
    ToolError
    VehicleError

class **Body**(SceneManager.Entity)

Method resolution order:
    Body
    SceneManager.Entity
    SceneManager.MovableObject

Methods defined here:

__init__(self, dic)
```
    class initalization call that raises an error when called directly
```
attach(self, body, pos=[0, 0, 0], lforce=0, aforce=0)
```
    Attaches the body to vehicles chassis position in local space where
    - force is the strenght of the attachement. Force must be positive or 0 value
    - pos is the position and
    - body is the body instance
```
getData(self)
getEntity(self)
getOrientation(self)
getPosition(self)
setOrientation(self, w, x, y, z)
setPosition(self, x, y, z)

Data and other attributes defined here:

file = None
id = None
name = None

Methods inherited from SceneManager.Entity:

__str__(self)
```
    Formated string describing name, file, animation names and position of the Entity
```
getAllAnimationNames(self)
```
    Returns a list of all the animations supported by this entity.
```
playAnimation(self, name, loop=False, enabled=True)
```
    Plays entity animation and returns None on success. Else raises an EntityError.
```
setMaterial(self, MaterialName)
```
    Sets this entites material. Replaces the material currently in use.
```
stopAnimation(self, name)
```
    Plays entity animation and returns None on success. Else raises an EntityError.
```

Methods inherited from SceneManager.MovableObject:

getChildren(self)
getDistanceTo(self, toObject)
```
    Returns the distance to the object 'toObject'
```
getPitch(self)
```
    Returns movable objects 'degree' around x axis
```
getRoll(self)
```
    Returns movable objects 'degree' around z axis
```
getScale(self)
```
    Returns movable objects scale in the form of a list having three float values [x, y, z]
```
getSize(self)
```
    Returns movable objects size in the form of a list having three float values [x, y, z] after scale
    (the bounding box size).
```
getYaw(self)
```
    Returns movable objects 'degree' around y axis
```
rotate(self, w, x, y, z, time=None, args=None)
```
    Rotates an object. Behaves exactly as setOrientation
```
scale(self, x, y, z)
```
    Scales relative to current scale
```

setPitch(self, degree)
Rotates object 'degree' around x axis
setRoll(self, degree)
Rotates object 'degree' around z axis
setScale(self, x, y, z, relative=False)
Scales relative to current scale
setVisible(self, b=True, cascade=True)
if b is True then we set this object visable and all its children
setYaw(self, degree)
Rotates object 'degree' around y axis
showBoundingbox(self, b=True)
if b is true then we display this object bounding box
translate(self, x, y, z, time=None, args=None)
Moves movable object relative to its parent

class **Tool**

Methods defined here:
__init__(self, dic)
class initalization call that raises an error when called directly
attach(self, body, pa=[], pb=[])
Will attach body to this body pa, pb is the delta from center of object
getData(self)
getEntity(self)
setAction(self, a)
Where a is an integer action type that is know by the tool
Common values are in the range of -1 to 1
setBallisticsEffect(self, entity=None, sound=None, psys=None, ppos=None)
Sets the pre effect when the tool does not function due to some internal state (missing bullets or power le
Where :
entity is the ballistics entity created
anim is name of the animation that this entity body currently has defined, returns error if not fou
sound is WeaponSound of the sound file to be played on success, returns error if not found. Default
psys is the name of the defined particle system, returns error if not found. Default is None not us
ppos is the position of the particle system in local coordinates (body position is null). Default i
This functions allows total control of what effect to be played out when the tool is used.
setDefaultEffect(self, anim=None, sound=None, psys=None, ppos=None, msec=0)
Sets the default effect when the tool is idle.
Where :
anim is name of the animation that this entity body currently has defined, returns error if not fou
sound is WeaponSound of the sound file to be played on success, returns error if not found. Default
psys is the name of the defined particle system, returns error if not found. Default is None not us
ppos is the position of the particle system in local coordinates (body position is null). Default i
sec is tool effect time and is 0 by default.
This functions allows total control of what effect to be played out when the tool is used.
setEnabled(self, b)
setNeoEffect(self, anim=None, sound=None, psys=None, ppos=None, msec=0)
Sets the neo (new ballistics or creation of) effect when the tool does executes the tool (more specifically
Where :
anim is name of the animation that this entity body currently has defined, returns error if not fou
sound is WeaponSound of the sound file to be played on success, returns error if not found. Default
psys is the name of the defined particle system, returns error if not found. Default is None not us
ppos is the position of the particle system in local coordinates (body position is null). Default i
sec is tool effect time and is 0 by default.
This functions allows total control of what effect to be played out when the tool is used.
setPostEffect(self, anim=None, sound=None, psys=None, ppos=None, msec=0)
Sets the post effect when the tool has finished executing the neo effect (more specifically after neo msec)
Where :
anim is name of the animation that this entity body currently has defined, returns error if not fou
sound is WeaponSound of the sound file to be played on success, returns error if not found. Default
psys is the name of the defined particle system, returns error if not found. Default is None not us
ppos is the position of the particle system in local coordinates (body position is null). Default i
sec is tool effect time and is 0 by default.
This functions allows total control of what effect to be played out when the tool is used.
setPreEffectFailure(self, anim=None, sound=None, psys=None, ppos=None, msec=0)
Sets the pre effect when the tool does not function due to some internal state (missing bullets or power le
Where :
anim is name of the animation that this entity body currently has defined, returns error if not fou
sound is WeaponSound of the sound file to be played on success, returns error if not found. Default
psys is the name of the defined particle system, returns error if not found. Default is None not us
ppos is the position of the particle system in local coordinates (body position is null). Default i
sec is tool effect time and is 0 by default.
This functions allows total control of what effect to be played out when the tool is used.
setPreEffectSuccess(self, anim=None, sound=None, psys=None, ppos=None, msec=0)
Sets the pre effect when the tool successfully functions
Where :
anim is name of the animation that this entity body currently has defined, returns error if not fou
sound is WeaponSound of the sound file to be played on success, returns error if not found. Default
psys is the name of the defined particle system, returns error if not found. Default is None not us
ppos is the position of the particle system in local coordinates (body position is null). Default i
sec is tool effect time and is 0 by default.
This functions allows total control of what effect to be played out when the tool is used.
setRechargeEffect(self, anim=None, sound=None, psys=None, ppos=None, msec=0)
Sets the pre effect when the tool does not function due to some internal state (missing bullets or power le
Where :
anim is name of the animation that this entity body currently has defined, returns error if not fou
sound is WeaponSound of the sound file to be played on success, returns error if not found. Default
psys is the name of the defined particle system, returns error if not found. Default is None not us
ppos is the position of the particle system in local coordinates (body position is null). Default i
sec is tool effect time and is 0 by default.

This functions allows total control of what effect to be played out when the tool is used.
Neither effects can be running when this is played

Data and other attributes defined here:
file = None
id = None
name = None

class **Vehicle**

Is the static and dynamic representation of our vehicle. The class requires calls to setup the vehicle data (will use
and then during calls makeDynamic and makeStatic we create the actual vehicle based on this data.
Vehicle parts are represented using bolts and an vehicle part. If the vehicle mesh is omitted then no mesh and bolds a
Vehicle parts are connected using bolts if mesh is NULL then no visual representation (entity) is created

Methods defined here:

__init__(self, name)
Creates a physical vehicle having name 'name' without wheels. The chassis is a default object and is
changed by calling setChassis(...).

addAxis(self, type, meshName, zposition)
Will add an axis of type 'type' having mesh 'meshName'. The Mesh must be available
to the ogre resources.

addEnginePowerStep(self, ratio)
Adds a power step to the engine so that it will output value ratio*power at this step. The step position is
with an internal call that increases and decreases with values set by calling setEngine(...).
This value must be 0 to 1 where 1 will be the maximum value set. There can be many steps taken for an engin
reach 1. The first value should be 0 while the last is 1.
You can clear all steps by calling delEnginePowerSteps(...).
Raises VehicleError on failure.

addWheel(self, type, meshName)
Raises VehicleError on failure.

attach(self, body, pos=[0, 0, 0], lforce=0, aforce=0)
Attaches the body to vehicles chassis position in local space where
- force is the strenght of the attachement. Force must be positive or 0 value
- pos is the position and
- body is the body instance

delEnginePowerSteps(self)
Clears engine steps added during call of addEnginePowerStep(...). Will revert to linear increase of power.

delWheel(self, type)
Removes the wheel from current vehicle setup. Raises VehicleError if the wheel does not exist.
Raises VehicleError if the wheel does not exist.

destroy(self)
Removes the vehicle and current attached parts to the vehicle such as wheel, engine, suspension, chassis, e

getCorrection(self)
Gets this vehicles correct possition and rotation. This is used to update the network simulated
vehicles. Returns [[x, y ,z],[w, x, y, z]] formated string.

getDeadreconing(self)

getMaterial(self, part)
Returns the assinged material of the requested part. None if no material was defined for the part.
The successfull return value is material name

getName(self)

getOrientation(self)
Returns a quaternion of the vehicles orientation

getPartName(self, part)
Gets this vehicles generated name of the parts entity. The Parameter part must be one of
these supported strings: 'chassi' and 'tire'

getPosition(self)
Gets the vehicles position in global space.

getRotation(self)
Returns a array having [x,y,z] degree rotations
__MovableObjectProperty_Yaw = 4
__MovableObjectProperty_Roll = 5
__MovableObjectProperty_Pitch = 6

isDynamic(self)

makeDynamic(self)
Makes the actual vehicle according to the specified settings set through the set calls.
The representation is dynamic meaning it will be drivable and behave as an vehicle depending
on the settings.

makeStatic(self)
Makes the an static vehicle according to the specified settings set through the set calls.
The representation is static meaning it will NOT be drivable.

setAxisBolt(self, type, boltName, meshName, offset=[0, 0, 0])
Creates and attaches an bolt to the axis. The axis 'type' is either 'FrontLeft', 'FrontRight', 'RearLeft' o
The 'boltName' is used to connect another vehicle part to this bold.
The bolt is sort of a market that is used to identify the location.
'meshName' is the visual representation. The meshName must be an resource available mesh or null. If null n
is used.
'offset' is a three dimensional location relative to the parent 'chassis' center (or in other words, parent
Raises VehicleError on failure

setBrake(self, axis, force=100)
Sets the vehicle brake force. May not be negative null value and axis is either 'Front' or 'Rear'
Raises VehicleError on failure.

setChassis(self, mesh, mass, massoffset=[0, -1, 0])
Sets the vehicles chassis to be mesh having mass and an massoffset.
Where mass must be a positive value. The mesh must exist in resources.

setChassisBolt(self, boltName, meshName, offset=[0, 0, 0])
Creates and attaches an bolt to the chassis. The 'boltName' is used to connect another vehicle part to this
The bolt is sort of a market that is used to identify the location.

'meshName' is the visual representation. The meshName must be an resource available mesh or null. If null n
is used.
'offset' is a three dimensional location relative to the parent 'chassis' center (or in other words, parent
Raises VehicleError on failure

setControlArmBehavior(self, axisType, controlArmTravel)
Sets the control arm behavoir where
axisType is either 'FrontLeft', 'FrontRight', 'RearLeft', 'RearRight' and
controlArmTravel is larger than zero.
Raises VehicleError on failure

setControlArmMotion(self, axisType, boltType, meshName, pivotOffset=[0, 0, 0], behavior='Connect')
Creates an visual representation of the control arm (can be omitted). Where
axisType is either 'FrontLeft', 'FrontRight', 'RearLeft', 'RearRight'
boltType is created through calls setChassisBolt and setAxisBolt
meshName if name of file that is accessable by ogre resource system.
pivotSystem has three float values positioning the pivot of the vehicle part (where pivot is the axis which
behavoir is either 'Connect' ... TBD
If this call is not called then no visual representation will be created for this vehicle part but it will
function properly.
Raises VehicleError on failure

setCorrection(self, posrot)
Corrects the vehicle position and orientation if needed where posrot is a string having
format [[x, y ,z],[w, x, y, z]]. This string is turned into
two arrays. First having 3 values (x,y,z) and the second having quaternion (w,x,y,z).
Returns true if correction was made false if no correction was needed.

setDeadreconing(self, data)

setDriveShaftMotion(self, axisType, boltType, meshName, pivotOffset=[0, 0, 0], behavior='Connect')
Creates an visual representation of the drive shaft (can be omitted). Where
axisType is either 'FrontLeft', 'FrontRight', 'RearLeft', 'RearRight'
boltType is created through calls setChassisBolt and setAxisBolt
meshName if name of file that is accessable by ogre resource system.
pivotSystem has three float values positioning the pivot of the vehicle part (where pivot is the axis which
behavoir is either 'Connect' ... TBD
If this call is not called then no visual representation will be created for this vehicle part but it will
function properly.
Raises VehicleError on failure

setEngine(self, power, increase, decrease)
Sets the vehicles engine properties where power is the maximum power through output and increase / decrease
engine increase and decrease its output. These are values must be larger than 0 (good values are 60, 0.8, 1
will go from 0% to 100% during gas intake and gas halt. Where at 100% it will give defined power.
Call addPowerStep to tune how much the engine will output at these steps.
Raises VehicleError on failure.

setMaterial(self, part, material)
Sets this vehicles generated entities to te specified material 'material' of the vehicle.
The parameter 'part' must be one of these supported strings: see getPartName.
Raises VehicleError on failure. Stores the material on success and can be reassigned through call useMateri

setOrientation(self, w, x, y, z)
Sets the vehicles orientation of type quaternion in global space.

setPosition(self, x, y, z)
Sets the vehicles position in global space. The y position will be calculated "ground height + y". If y is

setShockAbsorberBehavior(self, axisType, shockAbsorberDamper, shockAbsorberTarget)
Sets the control arm behavoir where
axisType is either 'FrontLeft', 'FrontRight', 'RearLeft', 'RearRight' and
shockAbsorberDamper is larger than zero and
shockAbsorberTarget is between zero and one.
Raises VehicleError on failure

setShockAbsorberMotion(self, axisType, boltType, meshName, pivotOffset=[0, 0, 0], behavior='Connect')
Creates an visual representation of the shock absorber (can be omitted). Where
axisType is either 'FrontLeft', 'FrontRight', 'RearLeft', 'RearRight'
boltType is created through calls setChassisBolt and setAxisBolt
meshName if name of file that is accessable by ogre resource system.
pivotSystem has three float values positioning the pivot of the vehicle part (where pivot is the axis which
behavoir is either 'Connect' ... TBD
If this call is not called then no visual representation will be created for this vehicle part but it will
function properly.
Raises VehicleError on failure

setSteer(self, SteerServoIncrease=4, SteerServoDecrease=4, SteerMax=0.5999999999999998, RotateOwnAxisOnStop=0)
Sets the servo steering of the wheels. Where SteerServoIncrease is the speed in which we turn the wheel and
speed the wheels restores themselfs when player stops steering. SteerMax is the maximum angle (in radians)
The speed in which we turn must be not null positive values and SteerMax maximum value is 1.4 (80 degrees).
Raises VehicleError on failure.

setSuspensionBehavior(self, axisType, suspensionSpring)
Sets the control arm behavoir where
axisType is either 'Front' or 'Rear' and
suspensionSpring is larger than zero.
Raises VehicleError on failure

setSuspensionMotion(self, axisType, boltType, meshName, pivotOffset=[0, 0, 0], behavior='Connect')
Creates an visual representation of the suspenion (can be omitted). Where
axisType is either 'FrontLeft', 'FrontRight', 'RearLeft', 'RearRight'
boltType is created through calls setChassisBolt and setAxisBolt
meshName if name of file that is accessable by ogre resource system.
pivotSystem has three float values positioning the pivot of the vehicle part (where pivot is the axis which
behavoir is either 'Connect' ... TBD
If this call is not called then no visual representation will be created for this vehicle part but it will
function properly.
Raises VehicleError on failure

setTireBehavior(self, axis, LateralAsymptoteSlip, LateralAsymptoteValue, LateralExtremumSlip, LateralExtremumValue, LongitudalAsymptoteSlip, LongitudalExtremumValue)
The behavior of the tire is dependent on these values. They are used to calculate the grip with relation to
See PhysX documentation on NxWheelShape for detailed information.
'axis' is either 'Front' or 'Rear'.
LateralAsymptoteSlip, LateralAsymptoteValue, LateralExtremumSlip, LateralExtremumValue.

LongitudalAsymptoteSlip, LongitudalAsymptoteValue, LongitudalExtremumSlip, LongitudalExtremumValue.
Must be positive values, LateralExtremumSlip < LateralAsymptoteSlip and LongitudalExtremumSlip < Longitudal
Raises VehicleError on failure.

setTireGrip(self, axis, longitudal, lateral)
Sets the vehicle tires grip on 'axis' where axis is either 'Front' or 'Rear'.
The grip is the stiffness factor dependent on force.
Must have positive values. Raises VehicleError on failure.

setWheelBehavior(self, axisType, mass, massinv)
Sets the wheel behavior through mass and mass inverted values. Where
axisType is either 'FrontLeft', 'FrontRight', 'RearLeft', 'RearRight' and
mass and massinv is not negative values.

setWheelParticles(self, particleMaterial='', channel=0, delta=0)
Sets the vehicles wheel particle material. The material must be loaded or loadable by Ogre.
The material will be set to the wheel imidiatly and used when driving depending on settings provided.
Where particleMaterial is the name of an existing accessable ogre material
and channel is the channel to be assigned the particles (channels are integer values from 0 to 3)
and delta is the delta value to be added where we start our emission from (for instance if rotation
power is .3 and this is .5 then output will be .8)
If particleMaterial is NULL ('') then the particles for channel 'channel' is removed.
Raises VehicleError on failure.

setWheelTracks(self, materialName)
Sets vehicles wheel track material. The material named 'materialName' must be loaded by Ogre.
The material named 'materialName' will imidiatly be used on current tracks.
If null then wheel tracks are disabled.

useMaterial(self)
Will use the previus assigned material on all the objects. This is mainly so we do not need to
get the information again (we just reuse it).
Raises VehicleError if dictionary was not set at all (no succesfull call to setMaterial was made)

Data and other attributes defined here:
file = None
name = None
types = {'FrontLeft': [0, False], 'FrontRight': [0, True], 'RearLeft': [1, False], 'RearRight': [1, True]}

Functions

addBall(name, meshfile, mass=10, position=[0, 0, 0], static=False, radius=None)
Creates a ball having using meshfile as the representation.
If mass is not set then it will use 10 as default.
If radius is not set then it will use meshes radius as default radius
If position is not set then we use default 0,0,0

addBox(name, meshfile, mass=10, position=[0, 0, 0], static=False, size=None)
Creates a ball having using meshfile as the representation.
If mass is not set then it will use 10 as default.
If size is not set then it will use meshes size as default size
If position is not set then we use default 0,0,0
If static is true then we create a static object else not.

addCapsule(name, meshfile, mass=10, position=[0, 0, 0], static=False, dim=None)
Creates a capsule having using meshfile as the representation.
If mass is not set then it will use 10 as default.
If position is not set then we use default 0,0,0
If static is True then we create an static mesh else not.
If dim is set then we do not use the mesh dimension but the array of values that
must have 2 values [x,y]

addCar(name)
Creates a vehicle instance and returns it to the caller.

addGunTool(name, meshfile, force=1000, mass=10, distance=1000, position=[0, 0, 0])
Creates a gun tool having using meshfile as the representation.
Where distance is the range of our gun. Objects outside the range is not effected
If mass is not set then it will use 10 as default.
If size is not set then it will use meshes size as default size
If position is not set then we use default 0,0,0
If static is true then we create a static object else not.

addMagnetTool(name, meshfile, force=1000, mass=10, distance=1000, position=[0, 0, 0])
Creates a gun tool having using meshfile as the representation.
Where distance is the range of our gun. Objects outside the range is not effected
If mass is not set then it will use 10 as default.
If size is not set then it will use meshes size as default size
If position is not set then we use default 0,0,0
If static is true then we create a static object else not.

addMesh(name, meshfile, mass=10, position=[0, 0, 0], static=False)
Creates a capsule having using meshfile as the representation.
If mass is not set then it will use 10 as default.
If position is not set then we use default 0,0,0
If static is True then we create an static mesh else not.
Note that the mesh must be convex and follow PhysX requirements.

addPath(name, spline, materialName, deltaY=0.29999999999999999, width=20, orientation=[1, 0, 0, 0])
Adds a bezier path with name 'name' having zero points.
The path sides will be as close to the ground as possible.
Raises SceneError on failure.
Returns None on success.

addPrism(name, meshfile, mass=10, position=[0, 0, 0], dim=[1, 1, 5], static=False)
Creates a capsule having using meshfile as the representation.
If mass is not set then it will use 10 as default.
If position is not set then we use default 0,0,0
If static is True then we create an static mesh else not.
Dim must be an array of three values defining the width, height and side count

addRocketTool(name, meshfile, force=1000, mass=10, distance=1000, position=[0, 0, 0])
Creates a rocket tool having using meshfile as the representation.

Where distance is the speed of the rocket.
If mass is not set then it will use 10 as default.
If size is not set then it will use meshes size as default size
If position is not set then we use default 0,0,0
If static is true then we create a static object else not.

addTool(tooltype, name, meshfile, force=1000, mass=10, distance=1000, position=[0, 0, 0])
Creates a tool having using meshfile as the representation.
Where distance is the range of our gun. Objects outside the range is not effected
If mass is not set then it will use 10 as default.
If size is not set then it will use meshes size as default size
If position is not set then we use default 0,0,0
If static is true then we create a static object else not.

createWorld(HasFloor='', bHasGravity=True)
HasFloor takes a file name that represent its area. If config file is '' (nothing) then we create a floor at hei
Creates a physical world returning None. Raises PhysicsError on failure

debugEnabled(b, ip='127.0.0.1')

delBody(name)
Destorys the body and removes from scene. An body is created through calls
addBall(), addBox(), addMesh(), addCapsule(), addPrism()

delCar(name, dic)

delPath(name)
Destroys a bezier path with name 'name'
Raises SceneError on failure.
Returns None on success.

delTool(name)
Creates a tool having using meshfile as the representation.
Where distance is the range of our gun. Objects outside the range is not effected
If mass is not set then it will use 10 as default.
If size is not set then it will use meshes size as default size
If position is not set then we use default 0,0,0
If static is true then we create a static object else not.

destroyWorld()
Destroys a physical world and all of its physical contents if it exists returning None.
Raises PhysicsError on failure

getBody(name)

getCar(name)
Returns the car having id 'name' if it exists. Else returns None.

getCarNames()
Return all the loaded vehicle names

getMarker()
A marker is an object that follows the ground in front of the camera (used for editing).
A marker is created through call useMarker()
Returns the position of the terrain marker

isCarPresent(name)

setAreaProperty(dic, x=None, y=None)
NOT SUPPORTED
Sets an area property flusing the 'dic' into the system. The 'dic' attribute and values must be supported
otherwise we raise SceneError exception.
x and y is the page we assign the attributes onto. If omited we assign over the entire area.
'dic' should use supported attributes and values. If we are describing something we should describe it fully.
Currently we support 'define' of 'grass' and 'tree' where
'tree' input values are 'DetailLevelHigh' 'DetailLevelMedium', 'DetailLevelLow', 'DetailLevelCount', 'MeshNa
'grass' input values are 'Density', 'DetailLevel', 'ColorMap', 'DensityMap', 'SizeMaxX','SizeMaxY','SizeMinX

setAreaRegion(descr)
descr is a xml file that we load with the region configuration.

setCar(name, dic, deadreconing=None)

useMarker(meshName=None)
A marker is an object that follows the ground in front of the camera (used for editing).
Will enable/disable a terrain marker over a terrain. The marker is enabled if meshName is defined.
Passing None or '' will disable the marker.
Raises PhysicsError on failure.

Data

__author__ = 'Kjartan A Jonsson'
__copyright__ = 'Copyright (c) 2007 aGameCompany'
__license__ = 'aGameCompany'
__version__ = '$Revision: 1.41 $'

Author

Kjartan A Jonsson

APPENDIX – SIMULATION SYSTEMS

This chapter will briefly cover the background of different simulation frameworks that have resemblances to our implementation.

1.1 Simulators

Simulation frameworks are by no means a new concept. They are used throughout the field of computer science ranging from commercial games like the “Sims” to complex tools used in hospitals where doctors use external goggles to view patient internals during surgery. They also range from being one dimensional to three dimensional, simulating simple sound pulses, to first person military training simulations.

Our focus will be on simulators that try to simulate the world from a visual and physical standpoint supporting some kind of programmatic or scriptable AI interface into the system to control robot devices. These simulators are available as open source projects or as proprietary solutions.

None of the following projects depend on an internet connection nor do they require user registration before using their products. However, most of these simulator applications focus on professional simulation of robots rather than simulation from a games perspective with resources and underlying monetary system.

1.1.1 Webots

Webots is developed by Cyberbotics. "Webots is a three-dimensional mobile robot simulator. It was originally developed as a research tool for investigating various control algorithms in mobile robotics."

It has a complete IDE, debugging and structured information scheme for programming and viewing the simulation in real time.

When simulating the physical environment it uses ODE (Open Dynamics Engine), where we opted to exchange ODE for PhysX because of ODE state change was to heavy in computation.

Like our implementation each robot has a controller program. It handles initialization, run time, reset and shutdown of the robots.

Using Webots you can transfer the code to a real robot similarly to our Pyrobot library. Webots supports devices that can both rotate and translate the robot. It has a wide range of sensors like camera, accelerometers, distance sensor, light sensors, touch sensor, force sensor, data emitters, data receivers and a GPS device where it remembers its and other robots’ location. These devices can be adjusted by the user.

In Webots you program in C/C++, Java, URBI, Python, MATLAB and you can control many different types of robots, both virtually within the simulation framework and

physically, using real world robots. It contains an extensive set of Machine Learning algorithms.

1.1.2 Marilou Robotics Studio

Marilou is developed by anyKode and is a simulation framework where you can control a virtual or physical robot. The library support a range of languages (except Python) such as C/C++, VB#, J#, C#, CLI and URBI allowing the user to work with the same language and software tools on real robots.

It has a range of devices ready to use and different kinds of worlds with physical attributes such as shock, friction, and rebound, behavior with infrared or ultrasound. Just like our implementation, you can build your own robot through complex hierarchy assemblies.

There is no generic implementation of Machine Learning Algorithms in the distribution. The environment is free for home usage but there are costs for professional and teaching applications. The studio is open source and can also be used on Linux.

1.1.3 Freebots

Is a lightweight robot simulation project that was released April 9, 2008 as an open source project on SourceForge.net.

It uses OpenGL/Inventor graphics and ODE physics for its simulation.

The application does not have a user interface when interacting and developing within the framework, but the code is lightweight and written in Python.

1.1.4 Player Project

Is a popular professional open source robot simulation framework and is open source based on Player/Stage or Player/stage/Gazebo projects. It has Python support as well as C/C++, Java, TCL (as well as an unofficial version using URBI).

"The Player Project creates Free Software that enables research in robot and sensor systems. The Player robot server is probably the most widely used robot control interface in the world. It supports a wide variety of hardware, also C, C++ and Python programming languages are officially supported, Java, Ada, Ruby, Octave, and others are supported by third parties. Its simulation backends, Stage and Gazebo, are also very widely used." - Player Project Wiki

The stage simulator is a 2D view of the robot(s) and its devices. It provides users with many different devices to try out. The stage world is defined through scripts where you lay out obstacles from a 2D perspective.

The Gazebo project is the 3D simulator available for the Player Project. It supports dynamic environments using ODE and it uses Ogre3dGRE3D for rendering.

This project has a similar structure to Pyrobot. See home page for further details

1.1.5 Microsoft Robotics Studio

Microsoft Robotics Studio is a visual programming tool used by professionals, researchers and for education. It has 3D simulation using PhysX, and supports a wide range of physical robots.

The programming environment is through .NET that in turn supports languages C#, VB#, J# and IronPython.

The current version was released in July 2008, but has been used in advanced projects like DARPA Urban Grand Challenge, MySpace back-end and Indiana university network coordinator. See home page for further details

1.2 References

The following is a list of references and more Python-based machine learning libraries.

playerstage.sourceforge.net

Pebl, http://code.google.com/p/pebl-project/

Divmod, http://divmod.org/trac/wiki/DivmodReverend

Open Bayes, http://www.openbayes.org/

Fast Artificial Neural Network Library (FANN), http://leenissen.dk/fann/

ffnet - http://ffnet.sourceforge.net/

PyAIML, http://pyaiml.sourceforge.net/

Orange, http://www.ailab.si/orange/

PyKE, http://pyke.sourceforge.net/

FuXi, http://code.google.com/p/python-dlp/wiki/FuXi

PyCLIPS, http://pyclips.sourceforge.net/web/

Pychinko, http://www.mindswap.org/~katz/pychinko/

ANTLRv3, http://www.antlr.org/

LIBSVM, http://www.csie.ntu.edu.tw/~cjlin/libsvm/

PyML, http://pyml.sourceforge.net/

Natural Language Toolkit, http://nltk.sourceforge.net/index.php/Main_Pag

Gambit, http://gambit.sourceforge.net/

PySCeS, http://pysces.sourceforge.net/

Myro, http://wiki.roboteducation.org/Myro_Development

List of Python libraries, http://swik.net/Ai+python

APPENDIX – MOUNTAIN CAR

Here is the entire Mountain Car implementation used by the brain MountainCarBrain.

```
"""
       Author: Kjartan Akil Jonsson, aGame Company
       Brain script learning to reach a goal on top of a mountain.
       This script uses RLtoolkit and is based on code originally by Rich Sutton

"""

# This was taken from example program in RLtoolkit named mountainDemoG
# (mountainAgent and mountaintEnv) for reinforcement learning with linear function
# approximation. The learning algorithm used is sarsa(lambda) with traces.
# Written by Rich Sutton 12/17/00
# Converted to Python in April, 2004
# Modified to use the RL interface in April 2004
# Modified for Machine Race May 2009

# For this example we need some system tools
# Import scene and system tools as well as Pyrobot brain

import SceneManager
import SystemHandler
from pyrobot.brain import Brain

total_episode_reward = 0    #collect rewards for statistics
prefered_th = 90            #this is the prefered vehicle rotation against the mountain
from RLtoolkit.RLinterface import RLinterface
from RLtoolkit.Tiles.tiles import *
from RLtoolkit.traces import *
from RLtoolkit.utilities import *

##############################################################################################
# Here begins the code that is taken from RLToolkit\examples\mountainEnv.py
# Changes to the original code is noted using KAJ NOTE
##############################################################################################
# Mountain car environment
#  functions getStats -> steps, episodes, episodesteps
#        and getState -> position, velocity, action       return current globals
# RL interface function is mountainEnv(a=None)
#   use   rli = RLinterface(mountainAgent, mountainEnv)
# initialize with setupEnvsetupEnv()

from math import *
from RLtoolkit.utilities import *
```

```
minPosition = 120                # Minimum car position, KAJ NOTE: was -1.2 denoting max position in
the curve
maxPosition = 320                # Maximum car position (past goal), KAJ NOTE: was 0.6 denoting bottom
of curve
maxVelocity = 10.07              # Maximum velocity of car, KAJ NOTE: was 0.07
goalPosition = 0.5               # Goal position - how to tell we are done, KAJ NOTE: not used
velocity = 0.0
position = 0.0
action = 1
steps = episodesteps = episodes = 0
randomStarts = False


def mountainEnv(a=None):
    if a == None:               # start of new episode
        return s0()             # return initial sensation
    else:
        s, r = do(a)            # do action
        return s, r             # return new sensation and state


def setupEnv():
    global steps, episodesteps, episodes, velocity, position, action
    velocity = 0.0
    position = 0.0
    action = 1
    steps = episodesteps = episodes = 0


def s0():
    "Returns initial state of mountain car for a new episode"
    global steps, position, velocity, episodes, episodesteps, action, data
    data = [[0 for i in range(resolution)] for j in range(resolution)]    #KAJ NOTE: added to populate
data
    #lasts, lastr, action = None, 0, 1
    action = 1
    if steps != 0:
        episodes += 1
    episodesteps = 0
    position, velocity = mCarInit()
    #print "Start of episode, sensation is", position, velocity
    return position, velocity


time_since_last_flush = 0
def do(a):
    "Does the specified action and returns a new state and reward"
    global action, total_episode_reward, time_since_last_flush
    action = a
    p, v = mCarStep(a)                  # do the action
    global position, velocity, steps, episodesteps, episodes
    position = p
    velocity = v
    steps += 1
    episodesteps += 1
    r = -1                          # get reward (always -1 in MC)
    if mCarGoal(p):                 # if we've reached the goal
```

```
        sp = 'terminal'                  # set the new state to indicate that
        print str(carbrain.robot.timestamp).replace(".0","") + ", " + str(total_episode_reward)
        total_episode_reward = 0 #KAJ NOTE: added to reset the reward
    else:
        sp = (p, v)              # otherwise new state is position and velocity
    #print "New state is", sp, "reward is", r
    total_episode_reward = total_episode_reward + r
    return sp, r


def mCarHeight(position):           #KAJ NOTE: Not used by MC implementation
    return sin(3 * position)


def mCarSlope(position):            #KAJ NOTE: Not used by MC implementation
    return cos(3 * position)


def mCarStep(a):
    "Applies the action and calculates the new position and velocity"
    global velocity, position
    oldp, oldv = position, velocity
    velocity = carbrain.getVelocity()   #KAJ NOTE: minmax(velocity + (.001 * factor)  + (-.0025 * cos(3
* position)), maxVelocity)
    position = carbrain.getPos()         #KAJ NOTE: original implementation the position was calculated
using position += velocity
    thrust = 0                          #KAJ NOTE: thrust is added because it is used by our vehicle
    if a == 0 :                          #KAJ NOTE: this checks if the vehicle is "stuck". If not then we
rotate it
        if math.fabs(velocity) < 0.01 :
            carbrain.goto("rotate")


    if a==0:                             # action is backward thrust
        factor = -1
        thrust = -1
    elif a == 1:                        # action is coast
        factor = 0
    else:                               # action is forward thrust
        factor = 1
        thrust = 1
    carbrain.robot.move(thrust, 0)
    return position, velocity


def mCarGoal (p):
    "Is the car past the goal?"
    return carbrain.isPassedGoal()      #KAJ NOTE: was 'return p >= goalPosition'


def mCarInit ():
    "Set car to its initial position for an episode"
    global randomStarts
    if randomStarts:
        position = randomInInterval(minPosition, goalPosition)
        velocity = randomInInterval(- maxVelocity, maxVelocity)
    else:
        carbrain.initPos()              #KAJ NOTE: was 'position = -0.5'
        position = carbrain.getPos()
```

```
        velocity = 0.0
    #print "Initial state is", position, velocity
    return position, velocity

def curStats():
    global steps, episodes, episodesteps
    return steps, episodes, episodesteps

def curState():
    global position, velocity, action
    return position, velocity, action

############################################################################################################
# Here begins some tool functions used to output state value data taken from
############################################################################################################


def calcD(Fs):
    "calculate distances for 3d window"
    global theta, m
    d = -10000.0
    for a in range(m):
        s = 0.0
    for v in Fs[a]:
        s += theta[v]
    if s > d:
        d = s
    return d

def dataFlush():
    global minPosition, maxPosition, maxVelocity, m, theta, numTilings, resolution
    global posWidth, velWidth, data
    _minPosition = float(minPosition)
    _maxPosition = float(maxPosition)
    res = resolution
    Fs = [[0 for item1 in range(numTilings)] for item2 in range(m)]
    for i in range(res):
        pos = _minPosition + ((_maxPosition - _minPosition) * (float(i) / (res-1)))
        for j in range(res):
            vel = (2.0 * float(maxVelocity) * (float(j) / (res-1))) - float(maxVelocity)
            loadF(Fs, pos, vel)
            if mCarGoal(pos):
                d = 0.0
            else:
                d = calcD(Fs)
            data[i][j] = d
    print str(data)

############################################################################################################
# Here begins the code that is taken from RLToolkit\examples\mountainAgent.py
# Changes to the original code is noted using KAJ NOTE
############################################################################################################
```

```
# Mountain car agent -- uses Sarsa learning with replace traces
# (Re)Initialize with mountainEnv.setupAgent()
# Functions for setting agent parameters: setAlpha, setEpsilon
# RL interface function is mountainAgent(s, r=None)
#    e.g. rli = RLinterface(mountainAgent, mountainEnv)


numTilings = 10                         # the number of tilings
posWidth = (maxPosition-minPosition)/2 #1.7 / 8             # tile width for position
velWidth = 8 #0.14 / 8                  # tile width for velocity
n = 8192                          # number of parameters in theta (mem size)
m = 3                             # number of actions (back, coast, forward)
epsilon = 0.01                          # probability of random action
alpha = 0.5                       # step size parameter
lambd = 0.9                       # trace-decay parameter
gamma = 1.0                       # discount rate
resolution = 30

lasts = None
lasta = 1
lastr = 0
traceH = TraceHolder(n, 0.01, 1000)
F = [[0 for item1 in range(numTilings)] for item2 in range(m)]     # vector of eligibility traces
theta = [0. for item in range(n)]
qValues = [0. for item in range(m)]                          # array of action values
cTable = CollisionTable(4096, 'safe')  #was 2048

# Mountain car agent -- uses Sarsa learning with replace traces
# (Re)Initialize with mountainEnv.setupAgent()
# Functions for setting agent parameters: setAlpha, setEpsilon
# RL interface function is mountainAgent(s, r=None)
#    e.g. rli = RLinterface(mountainAgent, mountainEnv)

def mountainAgent(s, r=None):
    """Main agent function to send to RLinterface"""
    global lasts, lasta, lastq, traceH, qValues
    if r == None:                       # initialize traces at start of episode
        traceH.decayTraces(0)
        lasts, lastr, lasta = None, 0, 1
    if s != 'terminal':
        a = choose(s)
        if r !=None:               # not initial state
            learnSarsa(lasts, lasta, r, s, a)
        lasts, lasta = s, a
        lastq = qValues[a]
        return a
    else:
            return None

def setupAgent():
    "(Re)Initialize agent"
```

```
    global traceH, F, qValues, theta, lasts, lasta, lastr, cTable
    lasts, lasta, lastr = None, 1, 0
    traceH = TraceHolder(n, 0.01, 1000)
    F = [[0 for item1 in range(numTilings)] for item2 in range(m)]
    qValues = [0.0 for item in range(m)]
    theta = [0.0 for item in range(n)]
    cTable = CollisionTable(4096)

def choose (s):
    "Chooses next action"
    global epsilon, m, qValues
    pos, vel = s                                # break state up into state vars
    for a in range(m):
        loadFeatures(F, pos, vel, a)                # compute feature sets for new state with a
        qValues[a] = computeQ(a)                # compute action values
    chooseA = egreedy(epsilon, m, qValues)
    # print "Agent chose action", chooseA, "qValues", qValues
    return chooseA

def learnSarsa(lastS, lastA, r, s, a):
    "Learns using sarsa"
    global lastq, alpha, gamma, lambd, numTilings, traceH, F, m, theta, qValues
    #print "learnSarsa", lastS, lastA, r, s, a
    if lastS != None:
        delta = r - lastq
        delta += gamma * qValues[a]
        amt = delta * (alpha / numTilings)
        for i in traceH.getTraceIndices():
            theta[i] += amt * traceH.getTrace(i)
        traceH.decayTraces(gamma * lambd)
        traceH.replaceTraces(F[a])
        #traceH.accumulateTraces(F[a])
        #alist = range(m)
        #alist.remove(a)
        #traceH.replaceTracesZero(F[a], [F[act] for act in alist])
    lastq = qValues[a]

def setAlpha(new):
    "reset alpha - works from other modules"
    global alpha
    alpha = new

def setEpsilon(new):
    "reset epsilon - works from other modules"
    global epsilon
    epsilon = new

def computeQ (a):
    "compute value of action for current F and theta"
    q = 0
    for i in F[a]:
        q += theta[i]
```

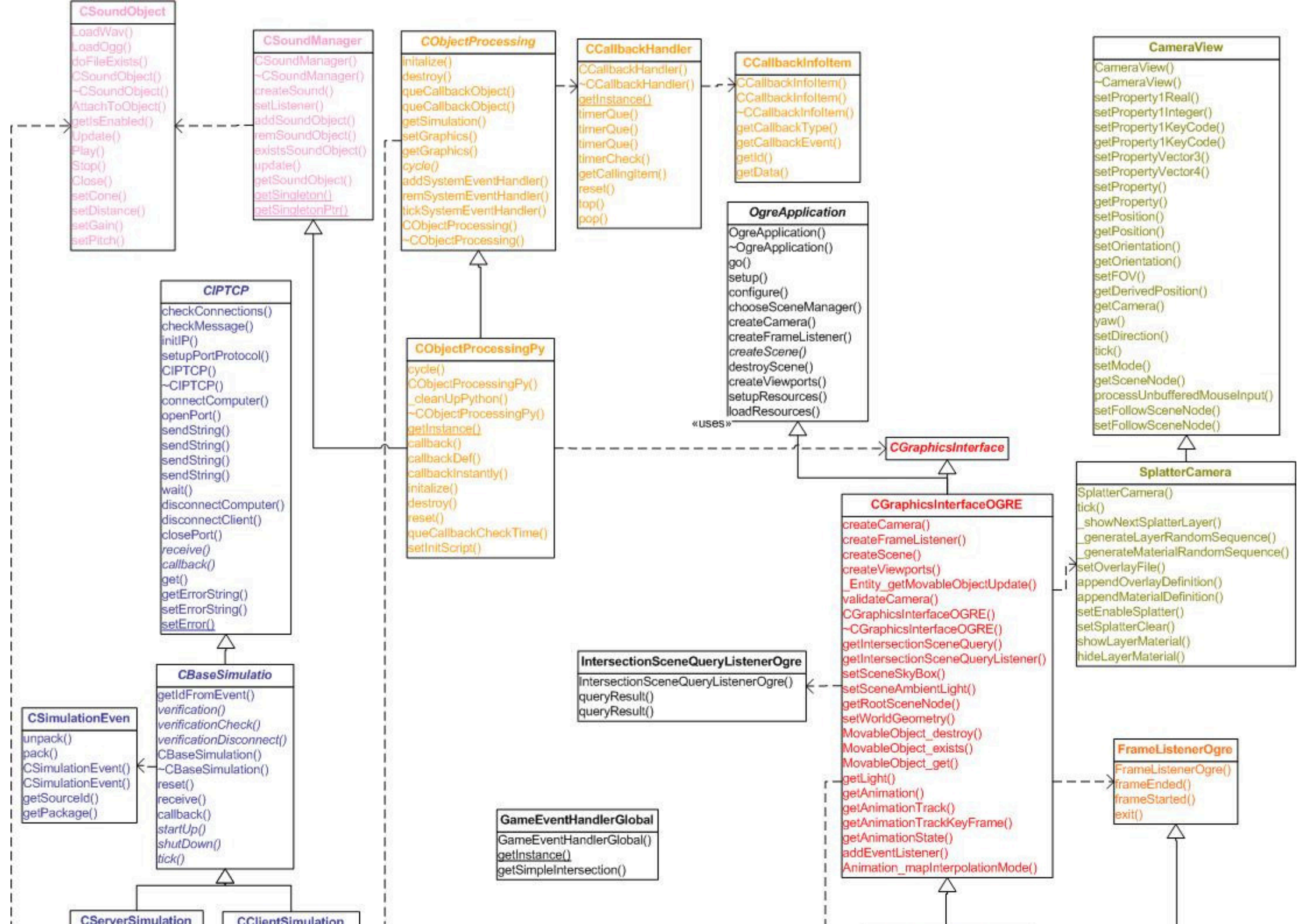
```
    return q

def loadFeatures (F, pos, vel, a):
    "Compute feature sets for action at current state"
    global cTable, posWidth, velWidth
    statevars = [pos / posWidth, vel / velWidth]
    loadtiles(F[a], 0, numTilings, cTable, statevars, [a])

def loadF (F, pos, vel):
    "Compute feature sets for each action at current state"
    global m, cTable, posWidth, velWidth
    statevars = [pos / posWidth, vel / velWidth]
    for a in range(m):
        loadtiles(F[a], 0, numTilings, cTable, statevars, [a])

def updateTheta (amt):
    for i in traceH.getTraceIndices():
            theta[i] += amt * traceH.getTrace(i)

##########################################################################################################
# Here begins our integration of the example code
##########################################################################################################



from pyrobot.brain import Brain
import random
import time
import datetime
import math
from pyrobot.brain.behaviors import State, FSMBrain

rotdir = -1

class rotate (State):
    def onActivate(self):
        self.state = "rotation"

    def step(self):
        global rotdir
        th = prefered_th
        delta_th = th-self.robot.th
        rotdir = -1
        if delta_th > 0 : rotdir = 1

        if delta_th < -180 :
            delta_th = 360 + delta_th
        elif delta_th > 180 :
            delta_th = delta_th - 360
        change_th = math.fabs(delta_th)
        velocity = carbrain.getVelocity()
        if velocity < 0 : return None #dont do any rotation while backing up
```

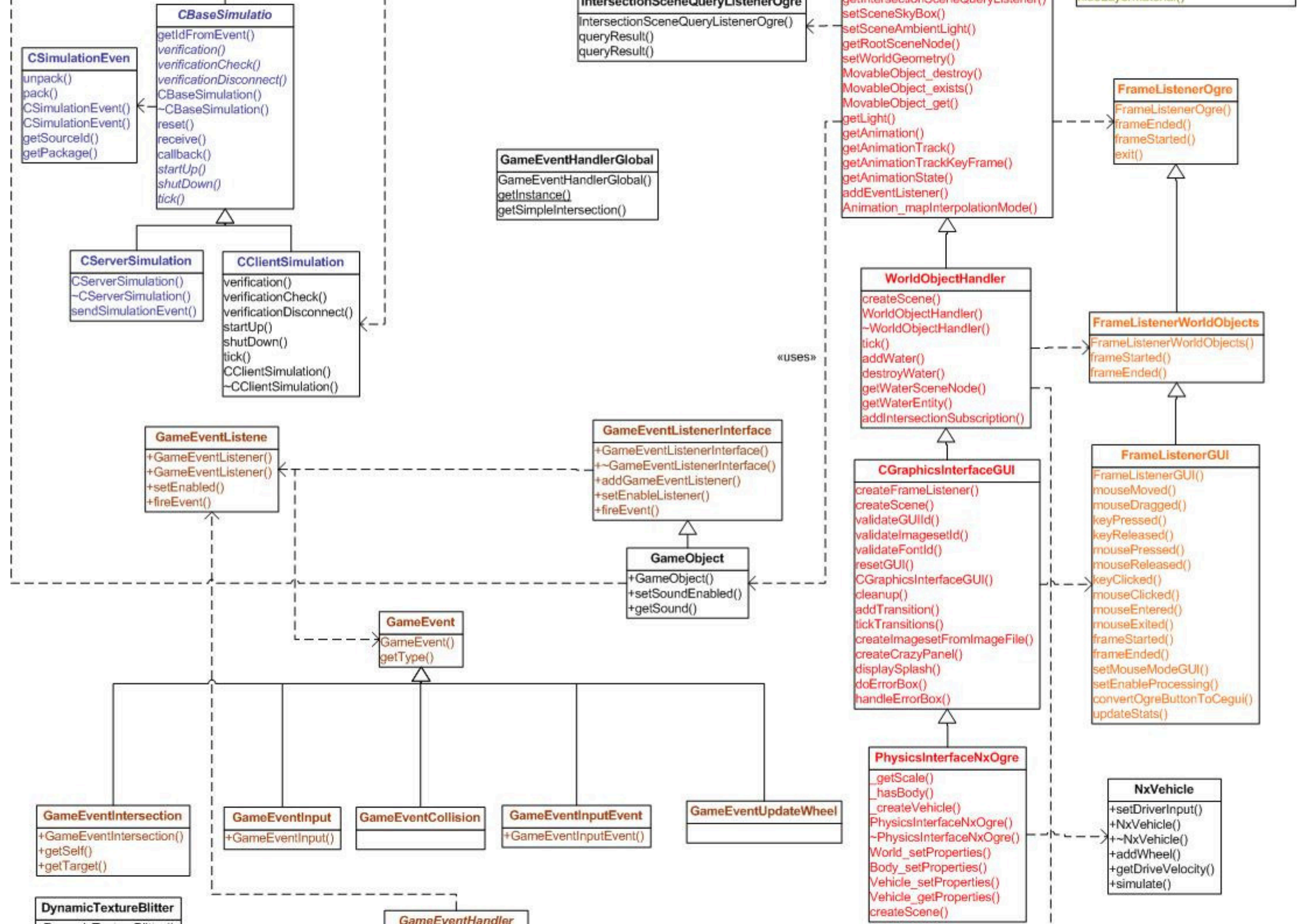
```
        if velocity == 0.0 :
            velocity = 0.000001
            if self.robot.stall :
                carbrain.initPos()
        if change_th < 10 :
            self.state = "drive"
            self.goto('learn')
        elif change_th > 45 and math.fabs(velocity) < 1 :
            self.state = "rotation"
            self.robot.move (0, delta_th)
        else :
            self.state = "rotation"
            if math.fabs(velocity) < 1 : rot = rotdir
            else : rot = (delta_th/360.0)
            self.robot.move (1, rot)


class learn(State):


    rli = None
    def getRotation(self) :
        a = self.robot.devData["getRotation"]()[0]
        return a


    def onActivate(self): # method called when activated or gotoed
        if self.rli == None :
                    setupEnv ()
                    self.rli = RLinterface(mountainAgent, mountainEnv)  #lets use RL toolkit
                    self.initPos()                                      #lets setup the machine
                    self.last_time = time.time()                          #we need to keep track of time
for speed calculations
                    self.goto('rotate')                                  #we probably need to adjust the
vehicle to face right direction
                    self.posx  =  SceneManager.getEntity  ("target").getPosition()[0]  #lets  get  the
position of our wall


    def initPos (self) :
        self.setPosition (287.43, 0, 209)                            #we always want to learn from this
position
        self.robot.move (0,0)                                         #we want the vehicle to start with
breaks on turning wheels front
        self.robot.stall = False                                    #lets assume that the vehicle is not
stalled to begin with


    def getVelocity (self) :
        """
        This calculates the velocity of the vehicle and used by the learner to know how fast we are
going while learning
        """
        b = [self.robot.x, self.robot.y]
        a = self.last_pos
        time_now = self.robot.timestamp
        timedelta = time_now - self.last_time
        distance = math.sqrt (abs(math.pow(a[0]-b[0],2) + math.pow(a[1]-b[1],2)))
        self.last_pos = b
        if distance == 0.0 : return 0.0
```

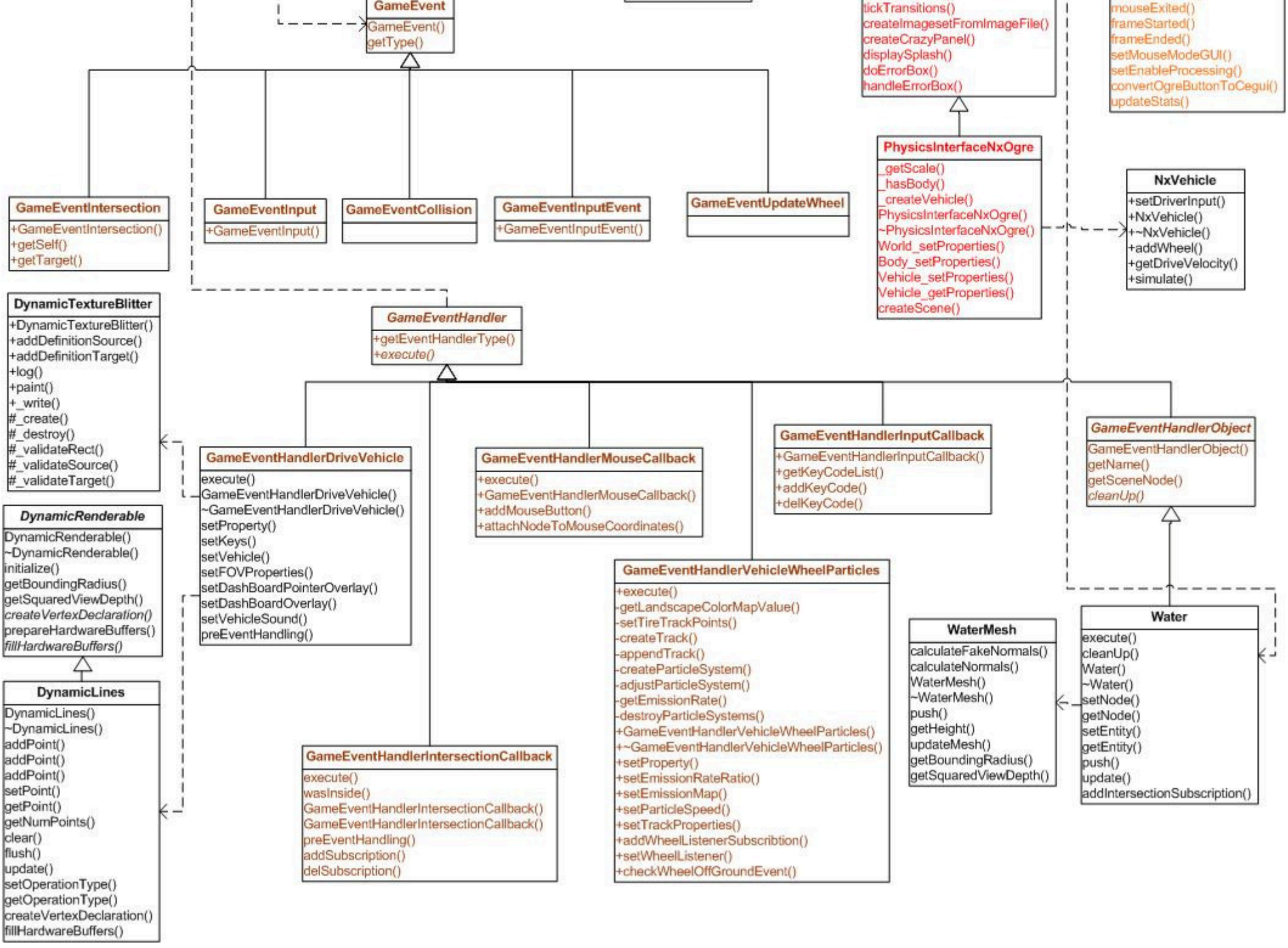
```
        if timedelta == 0.0 : return 0.0
        velocity  = float(distance) / float(timedelta)
        self.last_time = time_now
        return velocity

    def getPos (self) :
        """
        Since we are only learning one dimension lets return that only
        This is used by rli.mountainEnv
        """
        return self.robot.z

    def setPosition (self, x,y,z) :
        """This sets the vehicle position in world space using the radio device"""
        dic = {"x":x,"y":y,"z":z}
        self.robot.radio[0].callService("request move-to", dic)
        self.last_time = self.robot.timestamp
        self.last_pos = [self.robot.x, self.robot.y]

    def step(self) :
        """This is executed every 100 ms while this state is used"""
        global rotdir
        self.rli.step () #call this to do the actual learning, step will call RL toolkit methods
mountainEnv, mountainAgent
        if self.robot.stall :
            if rotdir > 0 : carbrain.initPos()  #we are stalled lets reset the vehicle
        if (abs(prefered_th-self.robot.th)) > 45 : self.goto ("rotate") #lets correct the vehicle
orientation if of course

    def isPassedGoal (self) :
        """Checks if we have passed our goal, this is used by rli.mountainEnv"""
        return self.robot.x > self.posx


def INIT(engine):
    global carbrain
    fsm = FSMBrain( engine=engine )
    carbrain = learn(1)
    # add behaviors, lowest priorities first:
    fsm.add( rotate (0) )
    fsm.add( carbrain )
    return fsm
```

APPENDIX – CLIENT UML

Here is an enlarged version of the Game Client UML diagram shown in figure 3.4.


CSoundObject
LoadWav()
LoadOgg()
doFileExists()
CSoundObject()
~CSoundObject()
AttachToObject()
getIsEnabled()
Update()
Play()
Stop()
Close()
setCone()
setDistance()
setGain()
setPitch()
CSoundManager
CSoundManager()
~CSoundManager()
createSound()
setListener()
addSoundObject()
remSoundObject()
existsSoundObject()
update()
getSoundObject()
getSingleton()
getSingletonPtr()
CObjectProcessing
initialize()
destroy()
queCallbackObject()
queCallbackObject()
getSimulation()
setGraphics()
getGraphics()
cycle()
addSystemEventHandler()
remSystemEventHandler()
tickSystemEventHandler()
CObjectProcessing()
~CObjectProcessing()
CCallbackHandler
CCallbackHandler()
~CCallbackHandler()
getInstance()
timerQue()
timerQue()
timerQue()
timerCheck()
getCallingItem()
reset()
top()
pop()
CCallbackInfoItem
CCallbackInfoItem()
CCallbackInfoItem()
~CCallbackInfoItem()
getCallbackType()
getCallbackEvent()
getId()
getData()
CameraView
CameraView()
~CameraView()
setProperty1Real()
setProperty1Integer()
setProperty1KeyCode()
getProperty1KeyCode()
setPropertyVector3()
setPropertyVector4()
setProperty()
getProperty()
setPosition()
getPosition()
setOrientation()
getOrientation()
setFOV()
getDerivedPosition()
getCamera()
yaw()
setDirection()
tick()
setMode()
getSceneNode()
processUnbufferedMouseInput()
setFollowSceneNode()
setFollowSceneNode()
OgreApplication
OgreApplication()
~OgreApplication()
go()
setup()
configure()
chooseSceneManager()
createCamera()
createFrameListener()
createScene()
destroyScene()
createViewports()
setupResources()
loadResources()
«uses»
CIPTCP
checkConnections()
checkMessage()
initIP()
setupPortProtocol()
CIPTCP()
~CIPTCP()
connectComputer()
openPort()
sendString()
sendString()
sendString()
sendString()
wait()
disconnectComputer()
disconnectClient()
closePort()
receive()
callback()
get()
getErrorString()
setErrorString()
setError()
CObjectProcessingPy
cycle()
CObjectProcessingPy()
_cleanUpPython()
~CObjectProcessingPy()
getInstance()
callback()
callbackDef()
callbackInstantly()
initalize()
destroy()
reset()
queCallbackCheckTime()
setInitScript()
CGraphicsInterface
SplatterCamera
SplatterCamera()
tick()
_showNextSplatterLayer()
_generateLayerRandomSequence()
_generateMaterialRandomSequence()
setOverlayFile()
appendOverlayDefinition()
appendMaterialDefinition()
setEnableSplatter()
setSplatterClear()
showLayerMaterial()
hideLayerMaterial()
CGraphicsInterfaceOGRE
createCamera()
createFrameListener()
createScene()
createViewports()
_Entity_getMovableObjectUpdate()
validateCamera()
CGraphicsInterfaceOGRE()
~CGraphicsInterfaceOGRE()
getIntersectionSceneQuery()
getIntersectionSceneQueryListener()
setSceneSkyBox()
setSceneAmbientLight()
getRootSceneNode()
setWorldGeometry()
MovableObject_destroy()
MovableObject_exists()
MovableObject_get()
getLight()
getAnimation()
getAnimationTrack()
getAnimationTrackKeyFrame()
getAnimationState()
addEventListener()
Animation_mapInterpolationMode()
IntersectionSceneQueryListenerOgre
IntersectionSceneQueryListenerOgre()
queryResult()
queryResult()
FrameListenerOgre
FrameListenerOgre()
frameEnded()
frameStarted()
exit()
CBaseSimulatio
getIdFromEvent()
verification()
verificationCheck()
verificationDisconnect()
CBaseSimulation()
~CBaseSimulation()
reset()
receive()
callback()
startUp()
shutDown()
tick()
CSimulationEven
unpack()
pack()
CSimulationEvent()
CSimulationEvent()
getSourceId()
getPackage()
GameEventHandlerGlobal
GameEventHandlerGlobal()
getInstance()
getSimpleIntersection()
CServerSimulation
CClientSimulation


CBaseSimulatio
getIdFromEvent()
verification()
verificationCheck()
verificationDisconnect()
CBaseSimulation()
~CBaseSimulation()
reset()
receive()
callback()
startUp()
shutDown()
tick()
CSimulationEven
unpack()
pack()
CSimulationEvent()
CSimulationEvent()
getSourceId()
getPackage()
CServerSimulation
CServerSimulation()
~CServerSimulation()
sendSimulationEvent()
CClientSimulation
verification()
verificationCheck()
verificationDisconnect()
startUp()
shutDown()
tick()
CClientSimulation()
~CClientSimulation()
IntersectionSceneQueryListenerOgre
IntersectionSceneQueryListenerOgre()
queryResult()
queryResult()
GameEventHandlerGlobal
GameEventHandlerGlobal()
getInstance()
getSimpleIntersection()
setSceneSkyBox()
setSceneAmbientLight()
getRootSceneNode()
setWorldGeometry()
MovableObject_destroy()
MovableObject_exists()
MovableObject_get()
getLight()
getAnimation()
getAnimationTrack()
getAnimationTrackKeyFrame()
getAnimationState()
addEventListener()
Animation_mapInterpolationMode()
FrameListenerOgre
FrameListenerOgre()
frameEnded()
frameStarted()
exit()
WorldObjectHandler
createScene()
WorldObjectHandler()
~WorldObjectHandler()
tick()
addWater()
destroyWater()
getWaterSceneNode()
getWaterEntity()
addIntersectionSubscription()
FrameListenerWorldObjects
FrameListenerWorldObjects()
frameStarted()
frameEnded()
«uses»
GameEventListene
+GameEventListener()
+GameEventListener()
+setEnabled()
+fireEvent()
GameEventListenerInterface
+GameEventListenerInterface()
+~GameEventListenerInterface()
+addGameEventListener()
+setEnableListener()
+fireEvent()
GameObject
+GameObject()
+setSoundEnabled()
+getSound()
CGraphicsInterfaceGUI
createFrameListener()
createScene()
validateGUIId()
validateImagesetId()
validateFontId()
resetGUI()
CGraphicsInterfaceGUI()
cleanup()
addTransition()
tickTransitions()
createImagesetFromImageFile()
createCrazyPanel()
displaySplash()
doErrorBox()
handleErrorBox()
FrameListenerGUI
FrameListenerGUI()
mouseMoved()
mouseDragged()
keyPressed()
keyReleased()
mousePressed()
mouseReleased()
keyClicked()
mouseClicked()
mouseEntered()
mouseExited()
frameStarted()
frameEnded()
setMouseModeGUI()
setEnableProcessing()
convertOgreButtonToCegui()
updateStats()
GameEvent
GameEvent()
getType()
GameEventIntersection
+GameEventIntersection()
+getSelf()
+getTarget()
GameEventInput
+GameEventInput()
GameEventCollision
GameEventInputEvent
+GameEventInputEvent()
GameEventUpdateWheel
PhysicsInterfaceNxOgre
_getScale()
_hasBody()
_createVehicle()
PhysicsInterfaceNxOgre()
~PhysicsInterfaceNxOgre()
World_setProperties()
Body_setProperties()
Vehicle_setProperties()
Vehicle_getProperties()
createScene()
NxVehicle
+setDriverInput()
+NxVehicle()
+~NxVehicle()
+addWheel()
+getDriveVelocity()
+simulate()
DynamicTextureBlitter
GameEventHandler


GameEvent
GameEvent()
getType()
tickTransitions()
createImagesetFromImageFile()
createCrazyPanel()
displaySplash()
doErrorBox()
handleErrorBox()
mouseExited()
frameStarted()
frameEnded()
setMouseModeGUI()
setEnableProcessing()
convertOgreButtonToCegui()
updateStats()
GameEventIntersection
+GameEventIntersection()
+getSelf()
+getTarget()
GameEventInput
+GameEventInput()
GameEventCollision
GameEventInputEvent
+GameEventInputEvent()
GameEventUpdateWheel
PhysicsInterfaceNxOgre
_getScale()
_hasBody()
_createVehicle()
PhysicsInterfaceNxOgre()
~PhysicsInterfaceNxOgre()
World_setProperties()
Body_setProperties()
Vehicle_setProperties()
Vehicle_getProperties()
createScene()
NxVehicle
+setDriverInput()
+NxVehicle()
+~NxVehicle()
+addWheel()
+getDriveVelocity()
+simulate()
DynamicTextureBlitter
+DynamicTextureBlitter()
+addDefinitionSource()
+addDefinitionTarget()
+log()
+paint()
+_write()
#_create()
#_destroy()
#_validateRect()
#_validateSource()
#_validateTarget()
GameEventHandler
+getEventHandlerType()
+execute()
GameEventHandlerDriveVehicle
execute()
GameEventHandlerDriveVehicle()
~GameEventHandlerDriveVehicle()
setProperty()
setKeys()
setVehicle()
setFOVProperties()
setDashBoardPointerOverlay()
setDashBoardOverlay()
setVehicleSound()
preEventHandling()
GameEventHandlerMouseCallback
+execute()
+GameEventHandlerMouseCallback()
+addMouseButton()
+attachNodeToMouseCoordinates()
GameEventHandlerInputCallback
+GameEventHandlerInputCallback()
+getKeyCodeList()
+addKeyCode()
+delKeyCode()
GameEventHandlerObject
GameEventHandlerObject()
getName()
getSceneNode()
cleanUp()
DynamicRenderable
DynamicRenderable()
~DynamicRenderable()
initialize()
getBoundingRadius()
getSquaredViewDepth()
createVertexDeclaration()
prepareHardwareBuffers()
fillHardwareBuffers()
GameEventHandlerVehicleWheelParticles
+execute()
-getLandscapeColorMapValue()
-setTireTrackPoints()
-createTrack()
-appendTrack()
-createParticleSystem()
-adjustParticleSystem()
-getEmissionRate()
-destroyParticleSystems()
+GameEventHandlerVehicleWheelParticles()
+~GameEventHandlerVehicleWheelParticles()
+setProperty()
+setEmissionRateRatio()
+setEmissionMap()
+setParticleSpeed()
+setTrackProperties()
+addWheelListenerSubscribtion()
+setWheelListener()
+checkWheelOffGroundEvent()
WaterMesh
calculateFakeNormals()
calculateNormals()
WaterMesh()
~WaterMesh()
push()
getHeight()
updateMesh()
getBoundingRadius()
getSquaredViewDepth()
Water
execute()
cleanUp()
Water()
~Water()
setNode()
getNode()
setEntity()
getEntity()
push()
update()
addIntersectionSubscription()
DynamicLines
DynamicLines()
~DynamicLines()
addPoint()
addPoint()
addPoint()
setPoint()
getPoint()
getNumPoints()
clear()
flush()
update()
setOperationType()
getOperationType()
createVertexDeclaration()
fillHardwareBuffers()
GameEventHandlerIntersectionCallback
execute()
wasInside()
GameEventHandlerIntersectionCallback()
GameEventHandlerIntersectionCallback()
preEventHandling()
addSubscription()
delSubscription()

APPENDIX – MOUNTAIN CAR

Graphs collected for comparison, where the first row is after approximetly 1 minute while the last image is taken after approximetly one hour:

State value graph provided by Mountain car example in RLToolkit	Mountain car problem solved using Machine learning framework.