The 2004 Report of the IGDA's Artificial Intelligence Interface Standards Committee

Alexander Nareyek, Börje F. Karlsson, Ian Wilson, Marcin Chady, Syrus Mesdaghi, Ramon Axelrod, Nick Porcino, Nathan Combs, Abdennour El Rhalibi, Baylor Wetzel, and Jeff Orkin (eds)

Welcome to our committee's report of 2004, which covers the activities of the committee from June 2003 to June 2004. The first section provides a general introduction and overview while the following sections present details on the committee's single working groups. Feel free to post questions, comments or suggestions to our feedback forum.[1]

Sections:

- Preface
- General Issues
- Working Group on World Interfacing
- Working Group on Steering
- Working Group on Pathfinding
- Working Group on Finite State Machines
- Working Group on Rule-based Systems
- Working Group on Goal-oriented Action Planning
- Support Team

[1] The current text is a PDF rendering (with minimal formatting) of the latest version of the report available on the Wayback Machine - Internet Archive (https://archive.org/web/), the original report was located at http://www.igda.org/ai/report-2004/report-2004.html

1. Preface

On June 17, 2002, our committee on standards for artificial intelligence interfaces (AIISC) was officially launched within the International Game Developers Association (IGDA). The committee's goals are to provide and promote interfaces for basic AI functionality, enabling code recycling and outsourcing thereby, and freeing programmers from low-level AI programming as to assign more resources to sophisticated AI. Standards in this area may also lay grounds for AI hardware components in the long run.

This is a report of the second year of our committee on standards for artificial intelligence interfaces (AIISC). The committee was launched on June 17, 2002 within the International Game Developers Association (IGDA). The committee's goals are to provide and promote interfaces for basic AI functionality, enabling code recycling and outsourcing thereby, and freeing programmers from low-level AI programming as to assign more resources to sophisticated AI. Standards in this area may also lay grounds for AI hardware components in the long run.

The pace of posting on the SourceForge forums (our work is coordinated via a project page at SourceForge) has slowed down this year, resulting in a total of 1067 posts compared to 1777 last year, but some working groups have established sub-groups, which mostly communicate by e-mail. The number of committee members has slightly increased, and we have 69 members now. Interestingly enough, this year saw the first hardware company joining the committee, dedicated to provide specialized hardware for AI techniques like pathfinding. The committee's members are assigned to working groups, which work on the following topics: World Interfacing, Steering, Pathfinding, Finite State Machines, Rule-based Systems and Goal-oriented Action Planning. There is a support team as well, mostly composed of students, which do a great job in supporting the working groups with summaries, documentation and so on. Overviews of the work of all these units during the last year can be found in the following sections.

Our yearly GDC roundtable on AI Interface Standards turned out to be great again, and we see a continuously growing perception of the need for game AI interface standards among game developers. Again, we got some very useful feedback from the roundtable's audience. Our projection of what we wanted to achieve until GDC - a first draft of our standards - was, however, too optimistic as it turned out. We will try to get it ready until next GDC, but realistically, some groups advance toward this goal faster than others, and we might not be able to cover drafts for every group. The time that we have at GDC is highly limited, and we will most likely focus on a subset of groups in the presentation for next year anyway because many participants mentioned that they felt that there was not enough time for the huge volume of information.

Finally, if you are interested in joining the committee, we would be happy to receive your application. Please follow our membership application information on the internet: http://www.igda.org/ai/

I would like to thank everyone who contributed to our committee's progress and this report. The committee is based on voluntary work and we would be nothing without the altruistic support of our members and coordinators! Given the outrageous regular workload in jobs of the game industry, this can hardly be appreciated enough.

See you at GDC 2005!

Alexander Nareyek
(Committee Chairman)

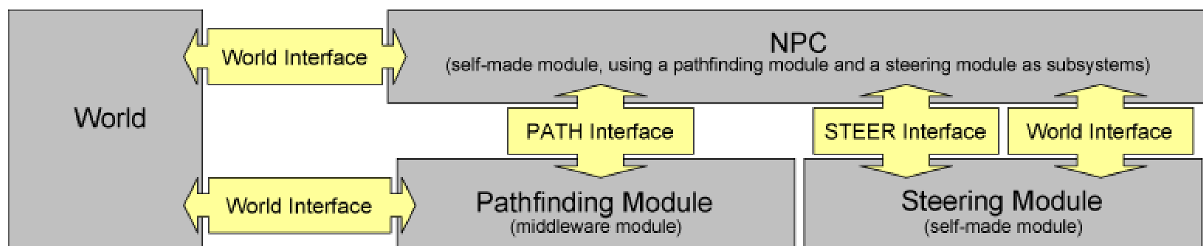
June 17, 2004
Cork, Ireland

2. General Issues

The committee has a general forum to discuss issues that are relevant for all groups. Some discussions are given below.

2.1. Module Interaction

The role of individual modules was further clarified in the discussion, i.e., how they interact with the main game engine and among themselves. The main idea is that every module has a **domain-specific interface** component (the special working groups take care of these) and a general **world interface** (part of the world interfacing group's task). Here is an example - note, however, that this example architecture represents only *one possible way*, mainly to show a variety of possible processing pipelines, and is not how we want everyone to build a system.



In the example, if the NPC wants to move, it uses the PATH Interface to send a movement goal to the Pathfinding Module, which in turn uses its World Interface to query possible waypoints from the World, and returns a potential path by the PATH Interface to the NPC. The NPC then uses the STEER interface to send the first waypoint to the Steering Module, which requests by its World Interface that all information necessary for its steering should be passed to it. The NPC forwards these requests to the World by its World Interface. Every few milliseconds, the Steering Module is called by the NPC via the STEER interface to compute a move, which is returned via the STEER interface. The resulting movement actions are forwarded by the NPC to the World. The requested position/movement updates from the World are passed via the World Interface to the NPC, which forwards this information to the Steering Module by its World Interface. If, however, the waypoint is reached, the NPC uses the STEER interface to push the next waypoint of the path as movement goal to the Steering Module. The NPC has internal logic to compute further behaviour for which it also makes use of its World Interface to the World.

In another scenario, for example, one could have pathfinding and steering on one level, a navigation module on top of them, and a finite-state machine module for the higher-level behaviour on top of the navigation module. All of them could have an own World Interface directly to the World.

There is no "best" way to couple components, and they should be flexible enough to adapt to the architectural requirements of the game at hand.

2.2. State Handling

The discussions on the general design concept continued, and concepts like state-based, object-oriented and data-oriented calls to set module properties were discussed. Issues regarding compatibility, hardware, debugging, and so on were raised in this context. Overall, the discussions showed some strong resentment against the state-based approach, partially based on bad experiences with OpenGL. The exploration of appropriate mechanisms only just started and will surely provide some interesting discussions in the near future.

3. Working Group on World Interfacing

AI Developers and Middleware vendors from the Game and Simulation Industries together comprise the AI Interface Standards Committee's (AIISC) world interfacing group. We are the central hub, coordinating ourselves with all other AIISC groups to create a standard interface between game AI systems, for example the other AIISC group implementations, and the game world representation / game engine.

Ian Wilson took over the coordinator role from Christopher Reed due to his time constraints this year and is now organizing the group's work and moderates its forum discussions.

The world interfacing group has the most active discussions with close to 1000 forum postings to date. In the past year our discussions moved from building up background knowledge about the many issues involved to beginning to put the points agreed on into a concrete form. To this end we are adopting the standard software development process of creating a set of Requirements with associated Specifications which we will use to create our final interface design. At this stage, the Requirements have been completed and we have created a number of sub-groups to create the Specifications for each of these Requirements. This work is currently ongoing.

While comparing this year's report to last years it may seem on the surface that progress has not been fast in fact much progress was made in determining our set of requirements. This process forced us to define exactly what we were going to produce and perhaps just as importantly what we were not going to produce. We now have a clear set of constraints on where we are headed and also a generally much clearer picture of what our interface will look like. This coming year could well see all of our discussions become concrete.

Our overall schedule is to have the Specifications completed by September 2004, the high and low-level Designs completed by December 2004 and an example use case written up by GDC 2005. This may be ambitious given the groups precious time resources, but it is a worthwhile goal for us to aim for.

3.1. Goals (Requirements)

We would like to define an Application Programming Interface for AI Components that:

1. Supports a large variety of Game World data
    1. Provides a set of standard data types
    2. Supports easily adding new data types
2. Supports a large variety of Game Engine architectures and designs

3. Supports data transfer between AI and the Game World
    1. Supports AI receiving event data from the Game World
    2. Supports AI making available event data to the Game World
    3. Allows for the future implementation of AI requesting data from Game World
    4. Allows for the future implementation of AI sending data to Game World
4. Supports the API requirements of all groups in the Committee
    1. Supports the "Rule-based Systems" group
    2. Supports the "Pathfinding" group
    3. Supports the "Steering" group
    4. Supports the "Goal-oriented Action Planning" group
    5. Supports the "Finite State Machines" group
5. Supports all committee wide agreed upon formats
    1. Provides documentation in format readable by a web browser
    2. Provides a High-Level Specification document of the API
    3. Provides a Low-Level Specification document of the API
6. Supports the adoption and use of AI within the games industry
    1. Supports the reuse of API code, allowing developers to avoid "reinventing the wheel" with each product.
    2. Supports reduced development cycles, allowing developers to use existing solutions with greater productivity and familiarity.
    3. Supports the development of more robust Games, allowing heavily tested and well-designed interfaces to be used easily.
    4. Supports the development of 3rd party (or shared) AI authoring tools, allowing developers to focus on game content with an externalized tool.
    5. Provides a set of standard terminology that can be adopted by the Game Industry, allowing better communication and common understanding between development studios for higher productivity.

3.2. Terminology

This list is intended to define those terms used within the Game / Interactive Entertainment industries to describe components of Artificial Intelligence systems and the Game Systems that are connected or related to those components.

Terminology:

```
[AI]
[ACTION]
[ACTUATOR]
[AGENT]
[AI SIMULATION]
[COMPUTER GAME]
[CONTROLLER]
[GAME ENGINE]
[ENGINE SYSTEMS]
```

```
[ENTITY]
[EVENT]
[FINITE STATE MACHINE]
[GAME SYSTEMS]
[GOAL PLANNING]
[INTERFACE]
[OBJECT]
[OBJECTIVE]
[PATH PLANNING]
[PLAYER]
[RULE-BASED SYSTEMS]
[SENSE]
[SENSOR]
[SIMULATED ENVIRONMENT]
[SOFTWARE SYSTEM]
[STEERING]
[STIMULUS]
[TERRAIN]
[QUERY]
[WORLD]
[WORLD INTERFACE]
```

As you can see, we have yet to fill in the actual definitions for these terms and that is something that we should complete soon with the assistance of the AIISC support team who are compiling a master list of terms.

3.3. Concepts

There are a number of potential approaches to designing a World Interface, but bearing in mind the practical realities of game development and that we are designing a general-purpose interface, we are currently moving forward with the following approach:

There is *no* world

Central to our current design initiative is the principle that our interface specifies only the AI side of the interface. No assumptions are made about the game world or engine. There are many different game architectures and it was decided that attempting to impose a standard on the game engine may prove too difficult initially. By defining only the AI side of the interface, essentially how to pass data to and get data from the AI sub-systems, control is placed in the hands of the developer as to how that fits in with their architecture. However, we leave open the possibility of defining both sides of the interface at some future point in time. So, in the words of our final discussion post/proclamation;

1. Whereas, it is critically important that any standard interface be applicable to all GAME ENGINES; and
2. Whereas, GAME ENGINES should have total control over when and how to execute SENSOR and ACTUATOR requests to optimize performance and compatibility with GAME ENGINE architecture; and

3. Whereas, the mechanism for interacting with the GAME ENGINE must be highly generic and expandable; and
4. Whereas, the goals and objectives of this group need to be narrowed and tightened if we wish to make any progress in the short term.

Therefore, be it resolved, that the first version of the World Interface Standard shall be entirely composed of passive SENSOR and ACTUATOR requests, forming a one-sided API where all function calls are defined AI components only. A two-sided API including functions defined on GAME ENGINE components and featuring a QUERY mechanism may be included in a future version of the World Interface Standard.

This makes the interface a passive one, receiving data and allowing data to be taken from it. It does not query the world itself or push data back to the world, which would require knowledge of the world and hence a definition of how the world was structured. While it is understood that many developers use query systems it was also felt that moving to an event-based architecture fulfilled our objectives (and the objectives of game developers) in a much more elegant, simple and powerful design.

3.3.1. API Input

This part of our design then has changed from last year where we had events and queries. We now have only events. In terms of inputs we are looking at the following question (and have separated our group into sub-groups to deal with these items):

```
Requirement #1 [Data Types]
This group will investigate what data types we want to support for
transferring back and forth between AI and the game world.

Requirement #2 [Architecture]
This group will investigate the syntactical specifics of the interface,
examine the pros and cons of a rigid component modelling approach as
opposed to a simple class structure, XML Schema,
GUI-Like approach, etc...

Requirement #3 [Data Transfer]
Working closely with group 2, this group will investigate the specific
implementation of sending and receiving the data types that group 1 is
putting together. How does AI make requests? What kinds of data remain
persistent? How does an event get fired? etc...

Requirement #4 [Other Working Groups]
This group will be responsible for spending some time in each of the
other groups forums, discussing our plans as they progress with those
groups and evaluating how their needs are being met.
```

While this work is in its preliminary stages, we do have some examples of the current, non-final, stages of the specifications (see later).

3.3.2. API Output

Again, this has changed somewhat from last year's report in that our interface is now a passive one that does not push data but instead allows it to be retrieved perhaps by direct calling or through the use of callback type methods. The groups mentioned in the section above will, of course, investigate both the receiving and posting of data (currently often termed subscribing and publishing in Event-based Architectures). While many people familiar with event systems will wonder about an event management system or "broker" it should be noted that to date we have only discussed this in passing and have no current plan to include such a system in the interface.

3.4. High-Level Representations

3.4.1. Event Data

Though still at an early stage of discussion the following is an idea proposed for the high-level structure of events. Central to this theme is that all events have a "header" and a "body" whereby the header contains mainly static information about the event which can be processed or routed quickly. Here the interface is passed events and generates events. Conceptually then the information coming to and going from the interface is in the same general format.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE GameAI SYSTEM "gameAI.dtd">

<EventHead>
    <EventID> Integer </EventID>
    <EventName> String </EventName>
    <EventCatagory> String </EventCatagory>
    <EventType> String </EventType>
    <EventVersion> Integer </EventVersion>
    <EventTime> Time </EventTime>
    <EventSynch> Bool </EventSynch>
    <PublisherID> Integer </PublisherID>
    <PublisherName> String </PublisherName>
    <SubscriberID> Integer </SubscriberID>
    <SubscriberName> String </SubscriberName>
<EventHead>

<EventBody>
    <Element>
        <Attribute>
            <Value> </Value>
        </Attribute>
    </Element>

.... etc
</EventBody>
```

Passing these common headers allows us to construct a common language between components while allowing flexibility in the way the data is actually packaged. The XML view above gives a nice conceptual view of this. Here are descriptions of the header elements:

```
<EventID> An integer ID for the event, each event having a potentially
unique ID
<EventName> A string to name the event, i.e. "EnergyPillConsumed"
<EventCatagory> A string to describe the catagory of the event, i.e.
"Pacman"
<EventType> A string to describe the type of event, i.e "EnergyPill"
<EventVersion> An Integer to note the version number of this type of event
or publishing module
<EventTime> The time the event was created
<EventSynch> If the event is synchronius or asynchronius [[, this needs
more explanation - Tom?]]
<PublisherID> An integer ID for the event publishing (creating) module
<PublisherName> An string ID for the event publishing (creating) module
<SubscriberID> An integer ID for the event subscribing (receiving) modules
<SubscriberName> An string ID for the event subscribing (receiving) modules
```

3.4.2. Standard Data Types

Following from this suggestion we will look at the formats to use for the body of the event / data. This is where we will need to coordinate with the other AIISC groups to determine those formats that are commonly in use in those areas. We will be looking at these themes:

- Decision Tree (DT)
- Finite State Machines (FSM)
- Goal-oriented Action Planning (GOAP)
- Pathfinding (PATH)
- Rule-based System (RBS)
- Steering (STR)

While it will of course be necessary and important to allow the developers to create their own data that conforms to our interface standard we would also like to supply them with a number of ready-made and regularly used data types to ease adoption and speed usage of out interface.

3.5. Low-Level Representations

With respect to the high-level XML representation of data type formats above we have also created a similar example in a C++ format to show how it could look in a low-level language.

3.5.1. Event Data

```
//////////////////////////////////////////////////////////////////////////////
// Event.h
//
// C++ Event data type.
//////////////////////////////////////////////////////////////////////////////
struct SEvent
{
    // EventHead
    int EventID;
    char* EventName;
    char* EventCatagory;
    char* EventType;
    int EventVersion;
    time EventTime;
    bool EventSynch;
    int PublisherID;
    char* PublisherName;
    int SubscriberID;
    char* SubscriberName;
    // EventBody
    ...... Required Data Format
};
```

Using our selected game example "Pacman" here is how a concrete example of the data format described might be used with the header information allowing for routing to specified AI modules behind the interface. Again, this is a preliminary suggestion;

```
Event.EventID = LastEventID + 1;
Event.EventName = "NewLocation";
Event.EventCatagory = "PinkGhost";
Event.EventType = "Movement";
Event.EventVersion = 110023;
Event.EventTime = 2456978; // Perhaps seconds
Event.EventSynch = SYNCH; // SYNCH / ASYNCH
Event.PublisherID = 10001;
Event.PublisherName = "GhostMovement";
Event.SubscriberID = 11001;
Event.SubscriberName= "PacmanPathPlan";
// EventBody
Event.NewLocation = Vector( x, y, z );
Event.Velocity = 102;
.... etc
```

3.5.2. Architecture

We also have a current suggestion for the low-level architecture of the interface using a "COM" like approach to handling non-specific and user-defined data being sent to and taken from the interface. Here is an example of how our interface file might look using a COM syntax:

```
//////////////////////////////////////////////////////////////////////////////
// IController.h
//
// C++ Controller Interface and related data types.
//////////////////////////////////////////////////////////////////////////////
struct SAction
{
    // TO BE FILLED IN BY GROUP #1
    // FOR EXMAPLE:
    float mStartTime;
    float mStopTime;
};

struct SSensor
{
    // TO BE FILLED IN BY GROUP #1
    // FOR EXAMPLE:
    float mStartTime;
    float mStopTime;
};

struct SEvent
{
    // TO BE FILLED IN BY GROUP #1 see above for example
};

interface IController : public IUnknown
{
    // TO BE FILLED IN BY GROUP #3
    // FOR EXAMPLE:
    virtual void Event (SEvent& event)=0;
    virtual int ActionRequestCount ()=0;
    virtual SAction& ActionRequest (int index)=0;
    virtual int SensorRequestCount ()=0;
    virtual SSensor& SensorRequest (int index)=0;
};
```

Bearing in mind how we have defined an event as containing any type of data, it could be that we class all incoming and outgoing data as events. However, this is essentially a semantic description and depends on what developers are most comfortable with (i.e. Event, Action, Sensor as in the example above).

3.6. Group Members

Current members of the working group on world interfacing:

- *Group coordinator:* Ian Wilson - Neon AI
- Tom Barbalet - Noble Ape
- Axel Buendia - SpirOps
- Greg Paull - MOVES Institute
- John Morrison - MAK Technologies, Inc.
- Doug Poston - Conitec
- Adam Russell - Pariveda
- Duncan Suttles - Magnetar Games

- Borut Pfeifer - Radical Entertainment

3.6.1. Requirements Sub-Group Members

- Requirement #1 [Data Types]: Ian Wilson, Tom Barbalet
- Requirement #2 [Architecture]: Duncan Suttles, Borut Pfeifer
- Requirement #3 [Data Transfer]: John Morrison, Greg Paull
- Requirement #4 [Other AIISC Groups]: Axel Buendia, Doug Poston

4. Working Group on Steering

The steering work group is formed by game developers, game AI middleware producers and academics. All of which contribute with their different backgrounds and points of view to the task of trying to create an interface between a game and a steering system. Much brainstorming has been done and many points of discussion have been started but most currently have not been concluded or reached any agreement. So, this is still a preliminary report on the current work of the Steering group.

4.1. Goals

The goal of the steering group of the AI Interface Standards Committee (AIISC) is to define a standard interface to ease the integration of game AI with a steering system. The interface will allow the expression of the following:

A. Physical state of a navigating agent. This has to be expressive enough to allow arbitrary movement, but it is not required that all of the state components are specified.
B. Steering actions. Again, most forms of movement must be available. Perhaps A and B can ultimately be merged into one?
C. Locomotion properties of the steering agent, e.g., degrees of freedom, max velocity, min turning radius, max rotational acceleration, etc.
D. Steering behaviours, i.e., the driving force motivating the movement. They turn physical state of an agent into steering actions. They will require access to world interface. They may also require access to locomotion properties (another sticking point).
E. Environmental constraints, e.g., obstacle avoidance, agent avoidance, etc. They may be just a type of steering behaviours.
F. (Optional) An arbitration system which, given a set of steering behaviours/constraints, locomotion properties and the physical state, produces the ultimate steering action.

Having defined some abstract concepts and the components of a steering system (as can be seen on last year report), the working group is currently working on a requirements specification and on reaching a conclusion on the following issues:

- A universal representation for A and B. Are a couple of vectors sufficient for most applications or do we need a comprehensive set of primitives, such as linear velocity, linear acceleration, angular velocity, angular acceleration, turning radius (scalar or vector), priority (if multiple goals are supported, see below), time (if goal-oriented planning is supported, see below), destination location (ditto), destination orientation, etc. etc? An open data structure

allowing an arbitrary combination of these components has also been discussed, but there is no agreement on its adoption.

- Separate steering behaviours from the physical substrate and express their workings in more abstract terms. This is most likely to apply to goal B. The most pertinent example is to use values corresponding to signals received from the control pad. Again, the group has not yet able to decide if this will give a sufficient freedom of expression.
- Number of alternative actions that a steering behaviour/constraint is able to produce. The standard approach is to only allow one, but there are situations (e.g., in obstacle avoidance) when multiple outputs enable more informed decision making.

  Support of continuous planning. Should the group strive to provide generic enough a specification for A and B that it allows an arbitrator to construct a detailed space of available trajectories on which a general goal-oriented planner can be run?

4.2. Requirements Specification

While no final interface has yet been decided on, the following requirements specification has been outlined and is currently being discussed:

- 1.1. Supports the steering behaviour *Seek* in 2D and 3D
- 1.2. Supports the steering behaviour *Flee* in 2D and 3D
- 1.3. Supports the steering behaviour *Arrive* in 2D and 3D
- 1.4. Supports the steering behaviour *Wander* in 2D and 3D
- 1.5. Supports the steering behaviour *Pursuit* in 2D and 3D
- 1.6. Supports the steering behaviour *Offset Pursuit* in 2D and 3D
- 1.7. Supports the steering behaviour *Evade* in 2D and 3D
- 1.8. Supports the steering behaviour *Wall Avoidance* in 2D and 3D
- 1.9. Supports the steering behaviour *Obstacle Avoidance* in 2D and 3D
- 1.10. Supports the steering behaviour *Collision Avoidance* in 2D and 3D
- 1.11. Supports the steering behaviour *Path Following* in 2D and 3D
- 1.12. Supports the steering behaviour *Interpose* in 2D and 3D
- 1.13. Supports the steering behaviour *Hide* in 2D and 3D
- 1.14. Supports the steering behaviour *Separation* in 2D and 3D
- 1.15. Supports the steering behaviour *Alignment* in 2D and 3D
- 1.16. Supports the steering behaviour *Cohesion* in 2D and 3D
- 1.17. Supports the steering behaviour *Flow Field Following*
- 1.18. Supports the steering behaviour *Queuing* in 2D and 3D

- 2.1. Supports the truncated sum method of arbitration
- 2.2. Supports the weighted truncated sum method of arbitration
- 2.3. Supports prioritized weighted truncated sum method of arbitration

- 2.4. Supports the prioritized dithering method of arbitration
- 2.5 Supports a custom arbitration policy specified by the client

- 3.1. Supports activating/deactivating individual behaviours
- 3.2. Supports querying if a behaviour is activated or not
- 3.3. Supports querying the combined steering goal produced by 2.x
- 3.4. Supports querying the forward component of the steering goal produced by 2.x
- 3.5. Supports querying the side component of the steering goal produced by 2.x
- 3.6. Supports querying the up component of the steering goal produced by 2.x
- 3.7. Supports some sort of ‘smoothing’ mechanism to avoid jitter
- 3.8. Supports altering the parameters that affect each behaviour
- 3.9. Supports querying the parameters that affect each behaviour

(Optional, as *steering code* could query the client interface for this info)

- 4.1. Supports specifying a 2D/3D goal position
- 4.2. Supports specifying a 2D/3D offset vector
- 4.3. Supports specifying 1..n “target” agents (for pursue/evade/interpose etc)
- 4.4. Supports specifying a “path”

4.3. Group Members

Current members of the working group on steering:

- *Group coordinator:* Marcin Chady - Radical Entertainment
- *Co-coordinator:* Mike Ducker - Lionhead Studios
- Mat Buckland - ai-junkie.com
- Philippe Codognet - University of Paris 6
- Thaddaeus Frogley - Rockstar Vienna
- Leon C. Glover - Entropy Unlimited
- Daniel Kudenko - University of York, UK
- Dave C. Pottinger - Ensemble Studios
- Craig Reynolds - Sony Computer Entertainment America
- Adam Russell - Pariveda

5. Working Group on Pathfinding

5.1. Introduction

The working group on pathfinding currently consists of nine members - after the unfortunate death of Eric Dybsand earlier this year. Syrus Mesdaghi has replaced Noel S. Stephens as the group coordinator. The group has 300 forum postings and more than 30 uploaded files consisting of skeletal interfaces and accompanying sample implementation.

In the past year, the discussions moved from terminology and high-level specification of graphs to more concrete design considerations. During the past year, we have considered different approaches to designing the interface(s).

Specifically, in order to deal with pathfinding, two main decisions have to be made:

1. First, one must decide on the representation of the world. Should the world be represented as a 2D grid, triangle mesh, waypoint graph, volume graph, or another representation?
2. Second, one must decide on the search technique. It is safe to say that most games use A* as their search technique. It is also true that as the size of the world increases and the number of agents increase, even running efficient search techniques such as A* can become impractical. As the result, when possible, many games still use the more trivial "Trial and Error" techniques despite their other drawbacks, and fall-back to A* only when is necessary.

5.2. Goals

We want to define a standard Application Programming Interface that will eliminate the need to reinvent the wheel by every developer that has to approaching pathfinding. Our goal is to:

1. Provide common input structures for world interfacing, or several types of structures for several different representations as detailed below.
2. Provide common output structures for interfacing other AI modules.
3. Allow for different search techniques, including hardware-based search support.

In addition, such API can promote consistency between tools that are used to create data sets on which pathfinding occurs. For example, 3D Studio Max and Maya plug-ins can output data in a format(s) that is beneficial to different games.

5.3. Different World Representations

Designing standard interfaces for pathfinding is not a trivial task. One of the questions is whether there should be a common interface that can meet the pathfinding needs of different games. This is difficult because even games that are from the same genre can have entirely different representations and search techniques. For example, the agents of games that are based on Half-Life or Unreal engines tend use waypoint graphs to navigate around the world. Agents of games that are based on the Quake III and Lithtech engines tend to use of volumes to find their way around the world. There are also plenty of games, including Counter Strike: Condition Zero, that use navigation meshes to help agents navigate the world and understand their surroundings. Games from different genres have even a wider range of needs. For example, an RTS game such has WarCraft III needs to deal with pathfinding for numerous agents. To make the matter worse, the world changes during runtime due to harvesting and destruction of trees. This means that pre-planning is not a viable option.

When designing the interface, there are many practicalities to consider. At one end of the spectrum, it may seem beneficial to abstract the interface enough so that it can deal with pathfinding for a wide variety of games. At the other end of the spectrum, it may make more sense to design multiple distinct interfaces that solve the pathfinding problem for specific scenarios. The latter approach means that there will be redundant functionality across the interfaces. Another point to consider is that if there is a single common interface, it will be easier to use search techniques and algorithms with different representation. However, it is also beneficial for search algorithms to know as much as possible about the specific underlying representation so that they can run efficiently and find better paths.

We now have sample implementations that represent the world as a triangular navigation mesh, waypoint graph, as well as a grid. We are currently considering whether we can try to design an interface that can work for different world representations, or we should commit to a distinct interface for each representation technique. We are generally leaning towards distinct interfaces, but we have not committed to either way.

5.4. Different Search Techniques

We have been debating on the proper way to deal with different search techniques. It is understood that search techniques in the broader sense include more than just different graph searching algorithms. Many games use some kind of algorithms, for refining or smoothing the solutions.

Currently, we consider an API supporting a 3-stage pipeline for searching:

1. *Preprocessing of queries*: find the right nodes in the graph for searching (e.g., the closest way points to the source and destination, the right nav-mesh nodes, move "up" in a tiered approach).
2. *Graph searching* (e.g., A*, "Trial and Error", fast marching).
3. *Post processing* (e.g., string pulling, spline smoothing).

We also hope to design an API so that it can be used for hardware-based search. Efficient hardware implementation requires the interface to support batch queries, i.e., receiving a set of queries for paths between different points and returning all solutions.

5.5. Group Members

Current members of the working group on pathfinding:

- *Group coordinator:* Syrus Mesdaghi - Dynamic Animation Systems
- *Co-coordinator:* Noel Stephens - Atari Games (Paradigm Division)
- Ramon Axelrod - AiSeek
- Mark Brockington - Bioware
- Mike Ducker - Lionhead Studios
- Ian Frank - Future University-Hakodate
- Jason Hutchens - Amristar
- Stephane Maruejouls - MASA Group
- Ian Millington - Mindlathe

6. Working Group on Finite State Machines

Finite state machines are an indispensable tool in every game programmer's toolbox. The finite state machine (or finite state automaton) in its simplest form is defined as a set of states S, an input vocabulary I, and a transition function T(s,i) mapping a state and an input to another state. The machine has a single state designated as the start state, where execution begins, and zero or more accepting states, where execution terminates.

FSMs are often depicted graphically using flowchart-like diagrams in which states and transitions are respectively drawn as rectangles and arrows. Graphical representations of FSMs are common, the Unified Modelling Language (UML) reserves one of its nine diagram types just for state machines.

An FSM is a concise, non-linear description of how the state of an object can change over time, possibly in response to events in its environment. The implementation of FSMs in games always differs from the theoretical definition. Some code is associated with each state so that as the object's state changes, its behaviour changes accordingly. Moreover, the transition function is broken up and its internal logic distributed among the states so that each state "knows" the conditions under which it should transition to a different state.

Game character behaviour can be modelled (in most cases) as a sequence of different character "mental states" - a change in state is driven by the actions of the player or other characters, or possibly some feature of the game world. In an FSM-based behaviour, the states describe how the character will act, and the transitions between states represent the "decisions" that the character makes about what it should do next. This "decision-action" model has the advantage of being straightforward enough to appeal to the non-programmers on the game development team (such as level designers), yet still powerful. FSMs lend themselves to being quickly sketched out during design and prototyping, and even better, they can be easily and efficiently implemented.

The FSM working group seeks to define a standard that captures the simple yet powerful way in which FSMs can be put to use. The working group has members from game development, middleware companies, and academia. Approximately 300 messages transpired last year, both on SourceForge and via direct e-mail.

6.1. Goals for a Finite State Machines Interface Standard

The goals of the FSM Working Group are to:

1. have a unified meta description of finite state machines

2. define an XML standard meta description of FSMs to enable interoperability of vendor packages featuring FSMs.
3. create a standard C++ and Java API for implementing a data driven FSM described in our XML files
4. create a best-practices guide for the many varieties of FSM implementation

6.2. Progress

1. The group has adopted the concepts, nomenclature, and graphical notation of UML to describe finite state machines.
2. The XML description of FSMs is in progress. Several 3rd party standards have been evaluated.
3. We are in the process of creating requirements for a standard API and will next move on to specification.
4. FSM working group member Dan Fu and Ryan Houlette have written a best-practices article for AI Wisdom 2; our own will take inspiration from that work.

6.3. Group Members

Current members of the working group on finite state machines:

- *Group coordinator:* Nick Porcino - LucasArts Entertainment
- Daniele Benegiamo - AI42
- Sam Calis - Universal Interactive
- Scott Davis - Black Cactus Games
- Fred Dorosh - BioGraphic Technologies
- Daniel Fu - Stottler Henke Associates
- Mark Gagner - WMS Gaming
- Ben Geisler - Raven Software / Activision
- Athomas Goldberg - Sun Microsystems
- Dave Kerr - Naturally Intelligent
- Linwood H. Taylor - University of Pittsburgh

7. Working Group on Rule-based Systems

In this report, we introduce the current work carried-out by the Artificial Intelligence Interface Standards Committee (AIISC) working group on Rule-based Systems (RBS). A major topic of discussion during the 2003-2004 period has been *what is the relationship* between Rule-based Systems and Scripting Engines in Game Artificial Intelligence? When are Rule-based Systems and rule programming used in Game Artificial Intelligence? And for what types of games?

We present some of the background concepts and issues discussed on the aspects of specification, design and development of a Rule-based System for game Artificial Intelligence (AI) to implement challenging game agents (i.e., NPCs - "Non-Player Characters"). We are investigating the components and possible architectures of RBS, and the different applications of game AI in general and of RBS in particular, depending on game genres.

7.1. Goals

Rule-based Systems can be used to encode behavioural rules that capture knowledge about a particular game scenario and the agents that inhabit it (e.g., game NPC opponents). In all but the simplest of games, rules that govern the behaviour of entities within the virtual world of a game need to embody complex relationships between large numbers of rapidly changing aspects of the overall game state. Programming these behavioural rules is an acknowledged problem in computer game construction.

We expect a trend in the games industry towards more rule-based styles of AI programming. At recent Game Developer Conferences, speakers (e.g., Peter Molyneux and Will Wright) indicated that game AI will play an increasingly important role as a source of "dynamics" and "emergent behaviour" that leads to new (generated/emergent) content within games. We believe that this trend will drive game development towards more sustainable programming styles based on "rules." While the members of this group may advocate, individually, rule-based and declarative programming techniques for game Artificial Intelligence, that is not the purpose of this group or of this report. Our objective is to establish a standard to encourage this option to game developers. This report introduces the work carried-out by the Artificial Intelligence Interface Standard Committee (AIISC) working group on Rule-Based Systems (RBS) to date.

Rule-based Systems are comprised of a database of associated rules. Rules are conditional program statements with consequent actions that are performed if the specified conditions are satisfied.

The aims of the group are to discuss and develop a set of standards on RBS applications and architectures suitable to games. This is a preliminary report on the current work of the RBS group, and which will be followed-up by recommendation and a proposal for a game-AI RBS interface standard in future reports.

- Our interface will assist game developers using an RBS to separate the representation of knowledge and the behaviour of intelligent entities in games from their implementation in software.
- Our interface will assist game developers using an RBS to easily integrate game-centric interactions involving the game engine and game entity AI.
- Our interface will provide a rule-structure definition language that is compatible with representation of knowledge and decisions in game systems using the RBS.
- With our interface and our reference implementation we will be able to identify general usage patterns applicable to different game genres.

The effort of this working group will also indirectly assist the game developer community in other ways:

- It will identify knowledge and reasoning patterns for building better and faster game AI using RBS.
- It can suggest means for architecting games better using RBS - more modular design, reusable components, and scriptable rules for more easily customized game AI.
- It can suggest means for architecting RBS that are more compatible with games requirements.

7.2. Major Findings Since the Last Report

7.2.1. Increased use of rule-based programming in game Artificial Intelligence expected.

We believe that game AI will rely increasingly upon rule-based and declarative programming techniques in the future. Our expectation is that rules adoption will likely occur first with server-based games (e.g., online multiplayer games), for a couple of reasons. First, server architectures tend to be already largely componentized (e.g., database, login, etc.) - adding a new RBS component on the server is likely more feasible than on highly optimized clients. Second, server-resident games share a number of requirements with commercial server applications, e.g., performance, scalability, and supporting dynamic loading and "hot swapping" of AI logic.

7.2.2. Rule-based Systems can be integrated into games using industry compatible interfaces.

At the level of integration, script interpreters and RBSs pose similar problems. They both have to relate to the game engine. Is their relationship synchronous? Where is the game state? Will they support functional call-backs? Etc. Should the game engine be able to reach-in and tune performance? We have hypothesized how these approaches might ultimately converge behind an interface rooted in an industry standard, **JSR-94** (Java Specification Requests). We believe that a JSR-94-based interface can support a range of RBS middleware (more powerful to less powerful RBS components). We start with the view that an RBS is a rule-engine that (words adapted from JSR-94):

- Acts as an if/then statement interpreter. Statements are rules.
- Promotes declarative programming by externalizing ...game logic.
- Acts upon input objects to produce output objects. Input objects are often referred to as facts and are a representation of the state of the ...game. Output objects can be thought of as conclusions or inferences and are grounded by the game in the... game domain.
- Executes actions directly and affect the game, the input objects, the execution cycle, the rules, or the rule engine.
- Creates output objects or may delegate the interpretation and execution of the output objects to the caller.

From the commercial applications sector we see a trend towards merging rules with scripting (or "rule-based scripting"). This sector has evolved a range of products that uses rule-based scripting to customize middleware. Consider large system architectures servicing business processes. Typically, such architectures integrate a range of products and contain much "glue code" (including scripts) whose behaviour is conditioned on the values in the data as it passes through, e.g., real-time data feeds etc. Such systems often represent metadata using rules: by separating the representation from the implementation (code) it simplifies maintenance.

While rule-scripting for single-player games can be simple, RBS integration with multi-player online game servers will require considerations beyond a JSR-94-based interface. For example, discussion in games technical forums have from time-to-time suggested that RBS optimizations that work well in commercial business domains may need to be modified for use in online games (e.g., RETE-based rule matching). By first agreeing upon the interface, however, the games industry can then let the middleware providers compete on specific solutions.

7.2.3. The JSR-4 RBS interface is a good foundation, but will likely need to be extended to support game Artificial Intelligence.

Within the RBS working group, we discussed what interface extensions to the JSR-94 will we likely need to support in an RBS API proposal. Specific discussions included:

- Should an API be specified to enable caller to throttle engine resource-usage, e.g., with respect to: memory usage, speed, caching etc.?
- Should an API be specified to support object-based scoping of rules? Objects would correspond to game world entities on the engine.
- Should it be an RBS requirement that an RBS support hot-swapping of rules (particularly in server-based games)
- Should the interface specification be in C++ or Java? The answer to this question is related to whether the first audience of the specification lies with single-player or server-based games.

7.2.4. Caveats with using rule-based systems with game Artificial Intelligence.

Within the RBS working group, we identified a number of issues that developers and middleware implementers will need to consider and address long-term in the game Artificial Intelligence domain:

- **Lack of benchmarks.** The rule market is fragmented. There are no current standards or consensus on functionality, and engines vary considerably in ability, performance and polish. There is not currently an established benchmark for comparing different rule engines.
- **Collection handling.** Rule-based systems have traditionally been designed for domains such as blood sample analysis where all variables were known ahead of time and where there was always a single instance. It is not easy and perhaps not even possible for production rule systems to to deal with groups, making it difficult to handle tasks such as selecting an enemy, selecting a tile to build on or evaluating the outcome of a multi-unit battle.
- **Learning curve.** Declarative programming is as different from object-oriented programming as assembler or SQL. It is not necessarily more or less difficult; it is simply different. Rule-based coding, like OO, is easy to do badly, so project teams should have a fair amount of experience to use rules effectively
- **Tool support.** Today, few rule authoring systems offer IDEs and debuggers as polished and functional as those available in modern OO languages. Because of the proprietary nature of most rule-based system tools, there are no general development tools as they run on many platforms.

- **Math support.** Many rule-based systems have no math facilities such as counters or addition.
- **Greater control of the execution of the RBS** - e.g., an additional API so that developers can exert greater control over what or how the rule engine is doing. Memory, thread control etc. are examples.
- **Game-centric RBS optimizations.** For example, the widely used RETE algorithm speeds performance by using substantial amounts of RAM. Because of the way the RETE algorithm is designed, there must be a separate rule base loaded for each agent - memory sharing is not possible. The Soar Quake Bot ran on a dedicated machine and that machine could only support 10 bots. Alternative algorithms or use-patterns may need to be identified for game Artificial Intelligence.

7.3. RBS Introduction

One form of AI that can be used is a *rule-based system*.

Rule-based systems differ from standard procedural or object-oriented programs in that there is no clear order in which code executes. Instead, the knowledge of the expert is captured in a set of *rules*, each of which encodes a small piece of the expert's knowledge.

Each rule has a left-hand side and a ride hand side. The left-hand side contains information about certain facts and *objects* which must be true in order for the rule to potentially fire (that is, execute).

Any rules whose left-hand sides match in this manner at a given time are placed on an *agenda*. One of the rules on the agenda is picked (there is no way of predicting which one), and its right-hand side is executed, and then it is removed from the agenda. The agenda is then updated (generally using a special algorithm called the *RETE algorithm*), and a new rule is picked to execute. This continues until there are no more rules on the agenda.

7.3.1. Rule-based Systems Components

Rule-based systems consist of a set of rules, a working memory and an inference engine. The rules encode domain knowledge as simple condition-action pairs. The working memory initially represents the input to the system, but the actions that occur when rules are fired can cause the state of working memory to change. The inference engine must have a conflict resolution strategy to handle cases where more than one rule is eligible to fire.

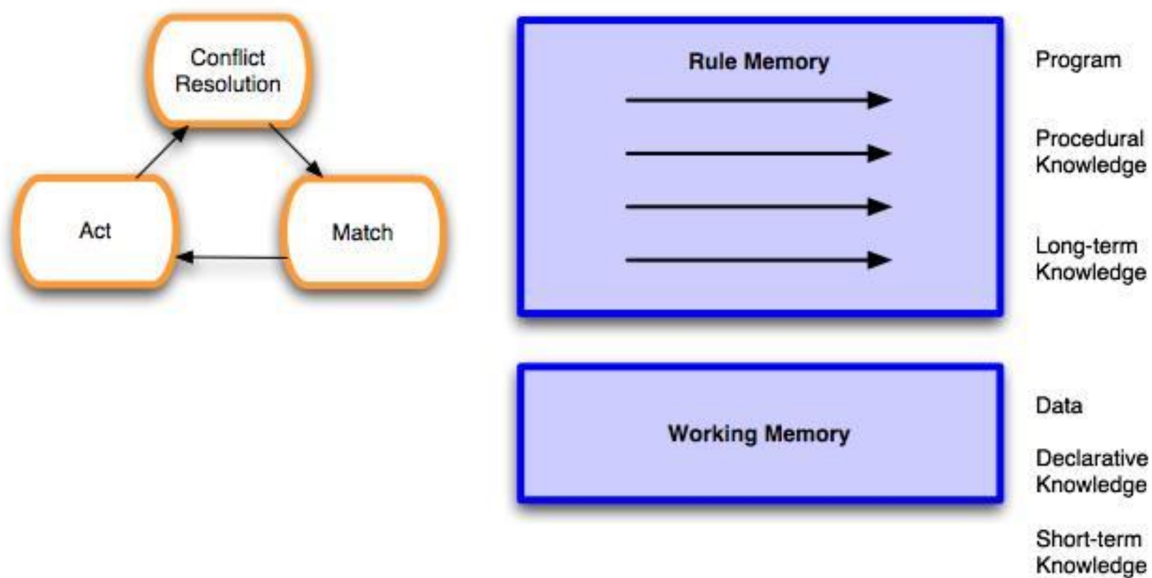
A rule-based system consists of:

- a set of rules,

- working memory that stores temporary data,
- inference engine.

The inference mechanisms that can be used by inference engines are:

- Backward Chaining:
  - To determine if a decision should be made, work backwards looking for justifications for the decision.
  - Eventually, a decision must be justified by facts.
- Forward Chaining
  - Given some facts, work forward through inference net.
  - Discovers what conclusions can be derived from data.



7.4. Discussion: Game-AI Behaviour as Rules or Not?

One of the most significant problems in NPC behaviour control construction is that procedural programming languages such as C, C++ and Java that are typically used in the implementation of computer games are not suitable for implementing large sets of complex behavioural rules. Attempting to construct complex behavioural rules in a language such as C++ tends to result in an impenetrable code involving large numbers of if-then-else constructs, switch, case and loop statements. The programmer is responsible not only for defining the conditions under which a behavioural rule is to be applied, and the action to be taken when it does apply, but must also deal with deciding all of the circumstances under which to check the game state against the behavioural rule's pre-conditions. Many of the behavioural rules required for game play will interact with one another in ways which cannot easily be determined a-priori.

A game continuously changes during development; and having an easily adaptable system for changing/adding/removing rules is of significant benefit.

In general, the result of attempting to construct complex NPC behaviour using procedural programming techniques will be complex code, which is incomprehensible and difficult to maintain, leading to an unavoidable increase in development time and requirements for large memory space and processing power. A Rule-based System enables a more flexible, scalable, robust way to design such behaviour. The reasons are grounded in pragmatics, of scalability, of usability, and of logic expressiveness.

- **Pragmatics**: separates implementation from the logic of the behaviour. This can lead to more maintainable engineering and code.
- **Scalability**: easier to separate logic about type or class from logic pertaining to the instance.
- **Usability**: likely more usable to mod developers - witness this trend with business software. Increasingly games builders are looking to appeal to 4th party developers (see Ben Sawyer) to develop outside content to extend the life of the product as well as to broaden its appeal. Such could lead to a "virtuous cycle" equivalent to one that developed in the applications sector, where we saw tools emerge for use with commercial development of large rule-based systems.
- **Logic Expressiveness**: The expressive need for both declarative and imperative forms is straightforward: sometimes it is just easier to think in rules and compute consequences; sometimes it is vice versa. Consider two different approaches for specifying behaviour: the first approach (imperative) is to describe the consequences or the process first; the second (declarative) is to describe the goals or rules first.

Programming game AI using rules can be more pragmatic than scripting because it separates implementation from the logic of the behaviour. This can lead to more maintainable engineering and code. Furthermore, a rule-based representation can provide a more concise and direct relationship between specification and implementation that would simplify testing. Separating the game engine from the game rules allows independent simulation and testing of each. Separating the logic from the implementation also enables reasoning about the logic by itself:

- Is it complete?
- Are all rules reachable?
- Can we use look-ahead?
- Can learning algorithms be applied?

The earlier sections illustrate some of the possible application for AI in game in general and RBS in particular. These are the creation of interesting opponents, realistic and engaging NPCs and maintaining consistency in dynamic storylines.

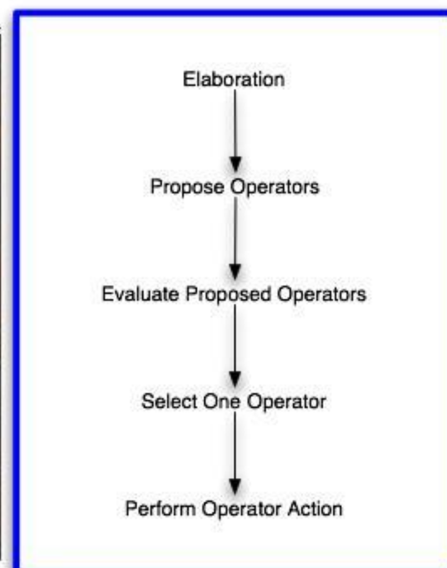
To help in making the game immersive and allow suspension of disbelief, the creation of the NPCs must provide different types of behaviour. Two categories of generic behaviour in NPCs are commonly used: reactionary and spontaneous.

NPCs behave in a reactionary manner whenever they are responding to a change in their environment. If an enemy spots you and starts to run towards you and shoot, then they have acted as a reaction to seeing you.

NPCs behave in a spontaneous manner when they perform an action that is not based on any change in their environment. A NPC that decides to move from his standing guard post to a walking sentry around the base has made a spontaneous action.

7.4.1. SOAR-BOT Example

Example: SOAR-Quake, courtesy of J. Laird.




```
IF
    enemy visible and my health is < very-low-health-value (20%)
OR
    his weapon is much better than mine
THEN
    propose retreat
```

7.5. RBS Programming

Rule-Based programming is based on a simple model of computation in which knowledge is encoded in the form of sets of condition-action pairs known as production rules. The condition part of a production rule represents the pattern which must be matched in order for the rule to be applied, and the action part of the production rule represents the response that is to be made when the rule is applied.

In a game application, the condition part of the production rule might involve interrogating the current state of the game, using virtual *sensors* and the action part of the rule might involve some response to the game state, effected via virtual *actuators*. Rules might also embody intelligent behavioural knowledge at a much higher level, involving planning actions to enable a game agent to achieve high level strategic goals, for example.

Many rule-based programming languages have been implemented, such as OPS5, CLIPS, Jess or RC++, for use in a large number of intelligent systems applications, including Expert Systems and Intelligent Software Agents.

A program written in a rule-based programming language is executed by using a working memory of assertions about the state of the world within which the program is operating, and an interpreter, which matches rules to the working memory, and adds to and removes assertions in response to the actions undertaken by rules that have been executed. A program expressed as a set of production rules is entirely declarative; production rules embody no procedural knowledge. It is the responsibility of the interpreter to control the execution of the rules. This is one of the principal advantages of rule-based programming. The programmer can concentrate on producing a discrete body of rule logic, separated from other aspects of the system, by writing a set of modular rules that are easily understood, easily extended, and easily modified.

7.5.1. Interface with the Game World

An important aspect of the architecture of the RBS is the interface with the game world and how they should be integrated. The world interface is still at a specification stage. However, the components proposed in the current report, such as events, actions and sensors are the main information required from the RBS to enable the NPCs to have the appropriate behaviour and interact with the world in a suitable way. The details of the integration are of course very dependent on what the final world interface will look like.

7.5.2. Research Extensions to Rule-based Systems

Rule-based systems support formalisms with different level of expressiveness. Examples of these include:

- propositional logic,
- first-order logic,
- events and temporal constraints,
- probability associated with rules,
- fuzzy logic,
- etc.

All of these can be used to provide better AIs, e.g., you can imagine a bot that hides when it is being shot at. Then it waits for five seconds before trying to shoot back if there is no other shooting and no incoming noise.

7.5.3. RETE Algorithm

The RETE Algorithm is widely used in commercial RBS implementations - it is regarded as the most efficient algorithm for optimizing mainstream commercial RBS systems. Because of its importance we introduce it here. However, as we have discussed within the working group, the RETE algorithm may not be optimal for game AI applications. We highlight this particular concern by this discussion, as it has been a recurring topic of discussion over the past year.

The efficiency of the RETE algorithm is asymptotically independent of the number of rules. Although a number of algorithms implementing production rules have been considered, based on actual, empirical evidence, the RETE Algorithm is orders of magnitude faster than all published algorithms with the exception of TREAT algorithm. RETE is usually several times faster than TREAT for small numbers of rules with RETE's performance becoming increasingly dominant as the number of rules increases.

The typical RBS has a fixed set of rules while the knowledge base changes continuously. However, it is an empirical fact that, in most RBSs, much of the knowledge base is also fairly fixed from one rule operation to the next. Although new facts arrive and old ones are removed at all times, the percentage of facts that change per unit time is generally fairly small. For this reason, the obvious implementation for an RBS architecture is very inefficient. The obvious implementation would be to keep a list of the rules and continuously cycle through the list, checking each one's left-hand-side (LHS) against the knowledge base and executing the right-hand-side (RHS) of any rules that apply. This is inefficient because most of the tests made on each cycle will have the same results as on the previous iteration. However, since the knowledge base is stable, most of the tests will be repeated. You might call this the *rules finding facts* approach and its computational complexity is exponential.

A very efficient method known is the RETE algorithm. It became the basis for a whole generation of fast expert system shells: OPS5, its descendant ART, RETE++, CLIPS, JESS, and ILOG-Rules.

For more information on RETE see:

- Forgy, C. L., "RETE: A fast algorithm for the many pattern/many object pattern match problem". *Artificial Intelligence*, 19(1) 1982, pp. 17-37.
- Giarratano and Riley, *Expert Systems: Principles and Programming, Second Edition*, PWS Publishing, Boston, 1993.

7.6. Specification

The aim is to propose a general game AI engine organized around the components mentioned in the RBS definitions section. This should make the implementation of NPCs easier by providing a suitable interface with the game-world, a common inference engine and different knowledge base suitable for a large variety of games.

A summary of RBS components is below with possible choices:

7.6.1. Knowledge Representation

- Production Rules
- Frames
- Object-oriented Representation

7.6.2. Inference Algorithm

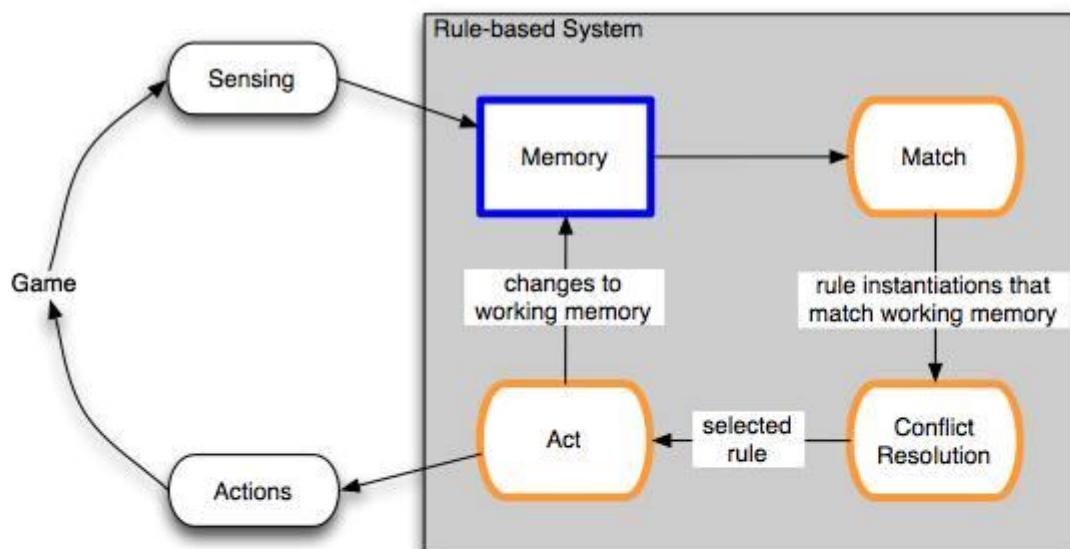
- RETE
- TREAT

7.6.3. System Architecture

- Centralized
- Multi-Agent
- Blackboard

7.6.4. RBS Game AI Cycle

The diagram below shows the "thinking" process.



7.7. Group Members

Current members of the working group on rule-based systems:

- *Group co-coordinator:* Nathan Combs - BBN Technologies
- *Group co-coordinator:* Abdennour El Rhalibi - Liverpool John Moores University
- Jean-Louis Ardoint - ILOG
- Daniele Benegiamo - AI42
- Hannibal Ding - Interserv Information Technique
- Clay Dreslough - Sports Mogul
- Frank Hunter - Adanac Command Studies
- Gerard Lawlor - Kapooki Games
- John Mancine - Human Head Studios
- Miranda Paugh - Magnetar Games
- Baylor Wetzel - GMAC RFC

7.8. WG Appendix A: Concepts and Terminology

Blackboard architecture

A Blackboard Architecture is an AI solution where Knowledge for a domain is shared between numerous KS (Knowledge Sources). Each KS represents an expert bringing its own set of knowledge to the blackboard and uses the knowledge published through the blackboard to build assumptions, make deductions etc.

Condition-action rule

A condition-action rule, also called a production or production rule, is a rule of the form: *if condition then action.*

The condition may be a compound one using connectives like *and*, *or*, and *not*. The action, too, may be compound. The action can affect the value of working memory variables, or take some real-world action, or potentially do other things, including stopping the production system. See also inference engine.

Conflict resolution

Conflict resolution in a forward-chaining inference engine decides which of several rules that could be fired (because their condition part matches the contents of working memory should actually be fired.

Conflict resolution proceeds by sorting the rules into some order, and then using the rule that is first in that particular ordering. There are quite a number of possible orderings that could be used.

Frames

Frames are a knowledge representation technique. They resemble an extended form of record (as in Pascal and Modula-2) or struct (using C terminology) or class (in Java) in that they have a number of slots which are like fields in a record or struct, or variable in a class. Unlike a record/struct/class, it is possible to add slots to a frame dynamically (i.e., while the program is executing) and the contents of the slot need not be a simple value. There may be a demon present to help compute a value for the slot.

Demons in frames differ from methods in a Java class in that a demon is associated with a particular slot, whereas a Java method is not so linked to a particular variable.

Heuristic

A heuristic is a fancy name for a "rule of thumb" - a rule or approach that doesn't always work or doesn't always produce completely optimal results, but which goes some way towards solving a particularly difficult problem for which no optimal or perfect solution is available.

Inference engine

A rule-based system requires some kind of program to manipulate the rules - for example to decide which ones are ready to fire (i.e., which ones have conditions that match the contents of working memory). The program that does this is called an inference engine, because in many rule-based systems, the task of the system is to infer something, e.g., a diagnosis, from the data using the rules. See also match-resolve-act cycle.

Knowledge base

Collection of the data and rules that suitably represent the problem domain.

Match-Resolve-Act cycle

The match-resolve-act cycle is the algorithm performed by a forward-chaining inference engine. It can be expressed as follows:

loop

match all condition parts of condition-action rules against working memory and collect all the rules that match;

if more than one match, resolve which to use;

perform the action for the chosen rule until action is STOP or no conditions match.

Step 2 is called "conflict resolution". There are a number of conflict resolution strategies.

RETE

Algorithm used to optimize forward chaining inference engines by optimizing time involved in recomputing a conflict set once a rule is fired.

Rule-based system

A rule-based system is one based on condition-action rules.

Search

Search is a prevalent metaphor in artificial intelligence. Many types of problems that do not immediately present themselves as requiring search can be transformed into search problems. An example is problem solving, which can be viewed in many cases as search a state space, using operators to move from one state to the next.

Particular kinds of search are breadth-first search, depth-first search, and best-first search.

Working memory

The working memory of a rule-based system is a store of information used by the system to decide which of the condition-action rules is able to be fired. The contents of the working memory when the system was started up would normally include the input data - e.g., the patient's symptoms and signs in the case of a medical diagnosis system. Subsequently, the working memory might be used to store intermediate conclusions and any other information inferred by the system from the data (using the condition-action rules).

7.9. WG Appendix B: References

See References section.

8. Working Group on Goal-oriented Action Planning

The goals of the Goal-orient Action Planning (GOAP) working group are somewhat different than those of the other groups due to the fact that GOAP is not a commonly used AI technique in today's games. Our group believes that the benefits of applying a GOAP architecture to games make it superior to more common techniques such as Finite State Machines (FSM) and Rule-based Systems (RBS). We are attempting to design an interface that will be appealing to game developers who have not previously employed a GOAP architecture, and extensible to others who would like more advanced features.

We think that GOAP addresses important issues that game developers are facing as the scale of game development increases. Specifically, we think the benefits are:

- Modularity: Separating NPC behaviour into atomic Actions and Goals makes a modular system that facilitates maintenance, and sharing of behaviours among different NPCs and/or projects.
- Workflow: The modularity of the GOAP architecture is well suited for the workflow of a game development team. Engineers implement Actions and Goals, and designers specify which Goals and Actions are available to different NPCs. Designers do not have to be concerned with coding any kind of explicit logic; the GOAP system uses preconditions and effects to sequence Actions into valid plans in real-time.
- Gameplay: NPCs employing a GOAP system can understand complex dependencies in the game world. By regressively searching for a satisfying plan in real-time, NPCs can come up with alternate means of solving problems, sometimes even in ways that are unanticipated by designers.

8.1. Goals

We have been focusing on our requirements specification, keeping the following long-term goals in mind:

- Determine how real-time dynamic planning can be used in practice, in mainstream commercial games.
- Define an interface for dynamic planning that suits game developers’ needs.
- Define an interface that can benefit planning in small instances and is easily extensible as planning becomes a more common Game AI technique.

This year, we have taken steps to arrive at a common understanding of what we mean by GOAP and determine the simplest set of interface requirements that would be useful to game developers. The minimal interface will be useful to the maximum

number of developers. Once more developers are using GOAP, the interface can be extended with more advanced features. These are the steps we have taken:

- Agreed on a standard example: The Sims
- Brainstormed requirements wish list.
- Paring down requirements to the simplest form usable in game development (no bells and whistles - e.g. probability, simultaneous satisfaction of multiple Goals).
- Not attempting to provide interface for Goal selection.

8.2. Concepts and Terminology

In order to nail down the requirements of our GOAP interface, we first needed to define some terms. In particular, we needed to define Goal, Action, and Plan. Our definitions are similar to those found in PDDL (a current planning technology standard for modelling planning tasks in academia). We define these terms as follows:

Goal: A goal is anything that an agent needs to satisfy. Satisfaction is defined by variables that must be true, or that should have a given value, for the state of the world in which to consider the planning task complete. An agent may try to satisfy the single most relevant goal, or multiple goals at once.

Action: An action is an atomic behaviour that contributes to the satisfaction of a goal or multiple goals. Actions have a variable number of effects and preconditions. Effects define how executing the action will change the state of the world. Preconditions define aspects of the state of the world that must be true prior to executing this action.

Plan: A plan is a valid sequence of actions that, when executed in the current state of the world, satisfies some goal or multiple goals. A sequence of actions is valid if each action's preconditions are met at the time of execution. The planner attempts to find an optimal plan, according some cost metric per action. The planning process cannot manipulate the actual state of the world. Instead the planner operates on a copy of the word state representation that can be modified as the planner evaluates the validity of all possible sequences of actions.

8.3. Examples

Once we defined our terms, we found it useful to put the pieces together into an example of how planning could truly be used in a current, commercial game. Our first example describes a situation common in any action games. We later decided that using the Sims as a standard example provides us with a wider range of planning opportunities.

Action game example:

Let's say we have an agent that wanders around repairing things when no threat is present and fires a weapon when a threat is in view. The agent needs to holster his weapon while repairing things and draw his weapon to fire.

The agent has 2 Goals:

```
GoalRepairObject
GoalKillEnemy
```

There are 5 available Actions:

```
1) ActionFixObject
PreConditions: kKey_AtObject = Variable
kKey_Armed = FALSE
Effects: kKey_FixedObject = Variable

2) ActionGotoObject
PreConditions: None
Effects: kKey_AtObject = Variable

3) ActionHolsterWeapon
PreConditions: kKey_Armed = TRUE
Effect: kKey_Armed = FALSE

4) ActionDrawWeapon
PreConditions: kKey_Armed = FALSE
Effect: kKey_Armed = TRUE

5) ActionFireWeapon
PreConditions: kKey_Armed = TRUE
Effect: kKey_TargetDead = TRUE
```

Goal: GoalRepairObject

If the agent is aware of an object that needs repair, GoalRepairObject returns some positive value for its current relevance. Otherwise, it returns 0.0

If GoalRepairObject activates, the agent needs to satisfy the WorldState:

```
kKey_FixedObject = ObjectID
```

where ObjectID is the ID of the object that needs repair.

The Goal now tries to formulate a plan that satisfies this WorldState. This is what the Goal WorldState looks like at each step in the planner:

```
Start Planning:
Goal: kKey_FixedObject = ObjectID

Apply: ActionFixObject
Goal: kKey_FixedObject = ObjectID (SOLVED)
kKey_AtObject = ObjectID
```

```
kKey_Armed = FALSE

Apply: ActionHolsterWeapon
Goal: kKey_FixedObject = ObjectID (SOLVED)
kKey_AtObject = ObjectID
kKey_Armed = FALSE (SOLVED)

Apply: ActionGotoObject
Goal: kKey_FixedObject = ObjectID (SOLVED)
kKey_AtObject = ObjectID (SOLVED)
kKey_Armed = FALSE (SOLVED)
```

So, the planner has found a sequence of actions that takes the WorldState from the true current WorldState to the solved Goal WorldState. The sequence of actions to satisfy the Goal is:

```
ActionGotoObject
ActionHolsterWeapon
ActionFixObject
```

Once this Plan is validated, the agent begins to execute it. When an Action activates, it runs a virtual ActivateAction() that could do anything. Usually it sets the agent's state to animate or navigate. When an Action completes, it applies its Effects to the real WorldState of the agent.

When GoalRepairObject is satisfied, it deactivates. If the agent is aware of anything else to fix, it reactivates and plans again.

Goal: GoalKillEnemy

If the agent can see an enemy, GoalKillEnemy returns some positive value for its current relevance. Otherwise, it returns 0.0

If GoalKillEnemy activates, the agent needs to satisfy the WorldState:

```
kKey_TargetDead = TRUE
```

This is what the Goal WorldState looks like at each step in the planner:

```
Start Planning:
Goal: kKey_TargetDead = TRUE

Apply: ActionFireWeapon
Goal: kKey_TargetDead = TRUE (SOLVED)
kKey_Armed = TRUE

Apply: ActionDrawWeapon
Goal: kKey_TargetDead = TRUE (SOLVED)
kKey_Armed = TRUE (SOLVED)
```

The planner has found a sequence of actions that takes the WorldState from the true current WorldState to the solved Goal WorldState. The sequence of actions to satisfy the Goal is:

```
ActionFireWeapon
ActionDrawWeapon
```

The Sims example:

1. The player has a set of numerical characteristics. These may change on their own, and this rate may change over time.
2. There are Objects. Some Objects may be carried (there is a 'carryable' flag). Some Objects may contain other objects (you can ask an object if it will accept another).
3. There are six primitive actions:
    a. Movement actions (move to an object),
    b. Purchase actions (buy an object),
    c. Discard actions (remove an object from the game),
    d. Use actions (each object may have more than one 'use' action available),
    e. Pick-up action (hold a carryable object, removing it from an existing object)
    f. f. Put-down action (place a held object in another object). Each action has a duration (e.g. movement has a distance to move, purchasing is instant in the Sims world), and a target object.
4. Using an object can alter the properties of a character in a well-defined way. It can also alter the object, or the contents of the object (e.g. using a cooker containing a chicken produces a cooked chicken).
5. Objects may appear and disappear at specific times (e.g. the car to take you to work can only be 'used' while it is outside).

Goals in the example are given as a combination of Personal Characteristic ranges to achieve.

Mapping the ontology of the Sims:

- Surfaces are implemented as just another object that can have others put on (in) them.
- Other characters are implemented as objects with different 'use' actions (e.g. Hug, Talk to).
- Character progression (e.g. the Job and level) are implemented as another numerical personal characteristic.
- Placement of purchases is ignored, as are placement interactions on the rate of change of personal characteristics.
- Change in personal characteristics is largely a black box. Effects such as happiness decreasing when the garden becomes overgrown and looks weedy, can only be reacted to, not anticipated, under this model.

This approach allows both a graphical and command-line implementation. It would allow the problem to be scaled from very simple (few objects and use actions) to very complex (playing the whole Sims game). It would be very simple to code up the game interface (a hundred lines of code or so). The Goals are then satisfied by formulating a sequence of Actions (purchase, use, move, etc). If all of an Agents attributes are values between 0 and 100, a Goal might be to get Hunger below 15, or get Relaxation above 75. Everything else is just a means to an end.

For example, if an Agent wants Hunger to fall below 15, he could either:

```
go to the store, buy food, and cook it
- OR -
get another Agent to fall in love with him, and cook for him!
```

Agents could plan for both the short and long term. Short term, buying food and eating works well, but long-term reading books to get a better job to afford a maid will supply the Agent with food more regularly in the long term.

8.4. Requirements Specification

Below is our most recent iteration of the requirements specification.

```
GOAP Requirements Document v0.4
===============================
We would like to define an Application Programming Interface that:

1. Supports planning for real-time behaviour of agents in a game.
1.1 Supports specifying a set of Goals per agent.
1.2 Supports specifying a set of Actions per type of agent.
1.3 Provides a means of generating a complete or partial plan given some
symbolic Goal(s) to satisfy.
1.4 Supports specifying an Initial State per agent (what s/he knows about
the world at planning time)
1.5 Supports specifying planning resource constraints (e.g. memory, execution
time).
1.6 Supports specifying domain resource constraints (e.g. money, game world
time).
1.7 Supports specifying a cost metric to optimize (e.g. money, game world
time, enjoyment)
1.8 Supports specifying a partially ordered set/s of Actions to be performed
as a plan.

2. Supports defining a Goal in terms of some state of satisfaction.
2.1 Supports symbolic representation of the satisfaction state.

3. Supports defining an Action in terms of the Action's preconditions and
effects.
3.1 Supports symbolic preconditions for the planner.
3.2 Supports contextual preconditions that depend on real-time game world
requirements.
3.3 Supports symbolic effects for the planner.
3.4 Supports contextual side-effects that affect the game world.
3.5 Supports associating a set of cost metrics with the execution of an
Action.
```

```
3.6 Supports specifying a duration for the execution of an Action.

4. Supports querying the status of the current plan.
4.1 Supports querying to determine if a plan is complete.
4.2 Supports querying to determine is a plan is still valid.
4.3 Supports querying the validation of an Action during execution.
4.4 Supports querying for the completion of an Action's execution.

Future Extensions:
- Specification of heuristics for Action and Goal selection.
- Associating probabilities with selection of Actions.
- Specification of Goal selection interface.
- Querying to determine if a plan is interruptible.
```

8.5. Design Decisions

In the process of fleshing out the requirements for our interface, we determined that we were really looking at three separate systems:

1. A goal selection interface that weighs up relevance, probabilities, and so on, and decides which goal to plan towards.
2. A planner that takes a goal and plans a series of actions to achieve it.
3. An action interface for carrying out, interrupting, and timing actions that characters can carry out.

This separation allows a good deal of modularity, which we feel will be essential to the interface's adoption. However, incorporating all of these systems into our initial interface adds complexity, and forces some design decisions that may not please all developers equally. We decided to focus our first version of the interface on (2) and leave (1) and (3) out for now.

In addition, there are various extensions that we decided to omit initially:

- Multiple goals: We will first concentrate on a useful specification for single goals. Once we have a solid foundation, and the support of game developers, planning for multiple concurrent goals would be a useful extension.
- Any mention of probability: Probability is important, but complicates the basic interface. This can be added in future versions once the basics are ironed out.
- Interruptible plans: Interrupting plans can be handled elsewhere. For example, there can be a wrapping controller that can just dump the current plan and ask the planner for a new plan with an updated game-state if something important changes during plan execution.

8.6. Links and References

See References section.

8.7. Group Members

Current members of the working group on goal-oriented action planning:

- *Group coordinator:* Jeff Orkin - Monolith Productions
- Penny Baillie-de Byl - University of Southern Queensland
- Daniel Borrajo - Universidad Carlos III de Madrid
- John Funge - iKuni Inc.
- Massimiliano Garagnani - The Open University
- Phil Goetz - Intelligent Automation
- John J. Kelly III - Model Software
- Ian Millington - Mindlathe
- Brian Schwab - Sony Computer Entertainment America
- R. Michael Young - North Carolina State University

9. Support Team

The AI Interface Standards Committee (AIISC) support team consists of ten people (mostly students) enthused about game AI from all around the world. There are different characters in the team, some of whom try to take the initiative whenever possible, and others who prefer to be assigned to tasks explicitly. Last year, Bjoern Knafla hold the position of the group coordinator but due to time constraints had to step aside from the group coordination. Börje Karlsson is the current coordinator and builds the main communication link between the committee chairperson Alexander Nareyek and the support team now. Additionally, it is the coordinators' task to try to distribute the workload onto all team members.

Our forum has some postings but most of the work is organized by e-mailing team members or the AIISC working group coordinators. Being far outnumbered by the other committee members - the experts - and all of us studying actively, it is sometimes hard to deliver immediate support when asked for. Nonetheless, we are striving to offer the best help possible.

We are proud to support the world's best game AI experts and to learn from them at the same time. The AIISC is a team, and we are helping to make a difference with our support - and have fun besides, too. However, we aren't sure if all of the experts (apart from some exceptions - mainly working group coordinators) really noticed and utilized our work. But collaborating with the AIISC working group coordinators and experiencing their passion is really great!

9.1. Tasks

Support tasks involve:

- summarizing the discussions of the different AIISC working groups,
- creating activity reports of all AIISC members,
- working into software, APIs or other standards (e.g., XML schemas or OpenGL) seemingly useful for the experts and providing short reviews of them,
- testing and proposing new tools that can be used to enhance the committee's productivity,
- assisting in the creation of presentation slides and reports,
- working on the creation of a glossary to try to uniform terminology usage among the groups,
- collecting questions of experts and answers given in the AIISC forums into so called "Expertly Asked Questions" (EAQ) documents,
- helping with technical problems mostly concerning the usage of SourceForge.net and accessing its CVS repositories,

- and having an open ear to all needs and problems that might occur in the daily committee work, e.g., developing slide templates for conferences like GDC.

Every member monitors at least one committee discussion forum to try to react quickly on support demands and to protocol the posted arguments in summaries. In general, our role is not to participate actively in the discussions, but we do so sometimes (when we think that we could provide some expertise as well).

The current approach to writing the summaries is to try to have more than one member monitoring each forum. One will produce the summary and one of the others will be something like a reviewer, changing this position the next time. With the steadily growing number of postings, more and more group work will surely have to take place, mainly between the supporters assigned to the same AIISC working group.

At the current stage, the support team has also started to work on issues of the support activity itself. We are trying to develop guidelines for making summaries, approaching group coordinators and quickly integrate and evaluate our work. This way we expect to raise our productivity and especially that of new members.

9.2. Group Members

Current members of the support team:

- Stephen D. Byrne - Hiram College
- Alex J. Champandard - University of Edinburgh
- Cengiz Gunay - Emory University
- Börje Karlsson - Pontifícia Universidade Católica do Rio de Janeiro (*current group coordinator*)
- Bjoern Knafla - University of Kassel
- José Lopes - University of Exeter
- Eric Martel - Microids
- Hugo da Silva Sardinha Pinto - Universidade Federal do Rio Grande do Sul
- Samir Pipalia - City University, London
- Jayaraman Ranjith - International Institute of Information Technology

References

[1] Dan Fu and Ryan Houlette. The Ultimate Guide to FSMs in Games. AI Game. Programming Wisdom 2, Charles River Media, 2004.

[2] JCP. JSR 94: Java Rule Engine API, 2004. URL: https://www.jcp.org/en/jsr/detail?id=94

[3] eXpertise2Go. Introduction to Expert Systems, 2001. URL: https://www.expertise2go.com/webesie/tutorials/ESIntro/

[4] Alison Cawsey. Forward Chaining Systems. In Databases and Artificial Intelligence 3 - Artificial Intelligence Segment, 1994. URL: https://www.cee.hw.ac.uk/~alison/ai3notes/subsection2_4_4_1.html

[5] J. E. Laird and M. van Lent. Developing an Artificial Intelligence Engine. In Proceedings of the Game Developers' Conference, 1999.

[6] C. L. Forgy. RETE: A fast algorithm for the many pattern/many object pattern match problem. Artificial Intelligence, 19 (1), 1982.

[7] J. C. Giarratano and G. D. Riley. Expert Systems: Principles and Programming, Second Edition, PWS Publishing, 1993.

[8] Alexander Nareyek, Bjoern Knafla, Daniel Fu, Derek Long, Christopher Reed, Abdennour El Rhalibi, and Noel S. Stephens. The 2003 Report of the IGDA's Artificial Intelligence Interface Standards Committee. International Game Developers Association (IGDA), 2003.

[9] M. Mateas and A. Stern. A Behavior Language for Story-based Believable Agents. IEEE Intelligent Systems, vol. 17, no. 4, 2002. URL: http://www-2.cs.cmu.edu/~michaelm/publications/AI-IE2002.pdf

[10] N. J. Nilsson. STRIPS Planning Systems. Artificial Intelligence: A New Synthesis, Morgan Kaufmann Publishers, 1998.

[11] Jeff Orkin. Applying Goal-Oriented Action Planning to Games. AI Game Programming Wisdom 2, Charles River Media, 2003.

[12] Jörg Hoffmann. Planning Domain Definition Language Description. 2004. URL: https://www.informatik.uni-freiburg.de/~hoffmann/ipc-4/pddl.html