USING GAMES ENGINES TO IMPLEMENT INTELLIGENT VIRTUAL ENVIRONMENTS

Carlos Calderon and Marc Cavazza
University of Teesside, TS1 3BA Middlesbrough, United Kingdom,
c.p.calderon@tees.ac.uk, m.o.cavazza@tees.ac.uk

KEYWORDS
Games Engines, IVE, and problem solving mechanisms.

ABSTRACT

In this paper we present an Intelligent Virtual Environment (IVE) obtained by incorporating a problem solving mechanism (AI technique) into a Virtual Environment. In particular, in this paper, we will discuss the implementation of the interaction with the problem solving mechanism by using the following metaphor: the visual space provided by the Virtual Environment is seen as the search space.

INTRODUCTION

The emerging area of Intelligent Virtual Environments explores the integration of Artificial Intelligence techniques into Virtual Reality systems. One particular research topic is to couple interactive AI systems (scheduling, planning, etc.) to virtual environments where the VE serves as an interface to the interactive planning system. Because the nature of the tasks to be solved are mainly spatial, Visualisation Engines, in general, and Games Engines in particular are well suited to serve as interface to the planning and/or problem solving mechanisms. That is, by incorporating AI techniques into 3d real-time interactive graphics technologies, we could, for instance, add a problem-solving component to the virtual environment. This would be beneficial for industrial applications where task scheduling or interactive design is relevant.

The objective of this paper is to present an interactive prototype in which we have linked a Virtual Environment (VE) to a constraint solving mechanism (solver). In our example application is possible to directly manipulate the objects from the virtual environment to create an input configuration for the planning system. This new input configuration is analysed by the problem solving mechanism and then, displayed in the Virtual Environment.

The next section briefly introduces the technologies used to create an Intelligent Virtual Environment for our example application. The third section describes how the problem solving mechanism has been incorporated into the Game Engine. Finally, the paper concludes with a brief discussion about the different implemented types of interactivity, further work lines of work and some conclusions.

TECHNOLOGY BASELINE

Interactive Visualization Engine

We decided to use the Unreal Virtual Machine for three main technical reasons: extensibility, an extremely powerful rendering engine and cutting-edge networking capabilities. Unreal has also been successfully used in non-gaming applications [1] and research projects [2].

Rendering engine and subsystems.

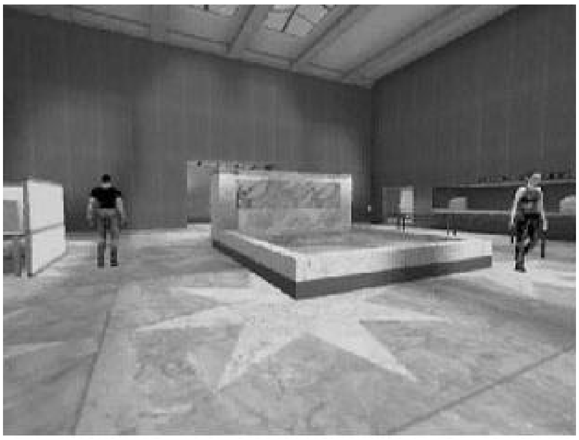

Figure 1. Unreal's scene.

The Unreal rendering engine has the most complete lighting conditions support of all the games engines. A variety of effects: such as surfaces with shadowing, realistic wavy water, animated materials, etc (see figure 1) have been used to add realism to the virtual scene. Unreal also supports digital sound system (software Dolby Surround encoding for full 360-degrees). Real world sounds , e.g water splashing, have been incorporated to the virtual scene to add realism to the exploration of the virtual environment.

Problem Solving Techniques

Our approach is based on a combination of Constraint Logic Programming over Finite Domains (CLP(FD)) and 3D real-time interactive technology.

Our choice of CLP(FD) as a software technology was made based on a series of factors: expressivity, the combination of search and incremental constraint solving capabilities, the short development time while exhibiting an efficiency comparable to imperative languages, and the fact that CLP(FD) is fast enough to react in "real-time" to the user's input configuration

In our concept architecture examples we have used GNU Prolog as a programming environment, which contains an efficient constraint solver over FD. GNU Prolog offers two different propagation techniques to solve arithmetic constraints: full arc-consistency and partial arc-consistency. This coupled with the use of labeling constraints help us to discover inconsistency as soon as possible, thus avoiding futile search through inconsistent alternatives. Hence, efficient and very fast constraint solvers can be programmed.

INTELLIGENT VIRTUAL ENVIRONMENT

This section explains how we have made the constraint solver work jointly with the visualization engine to create an Intelligent Virtual Environment in which the virtual environment serves as an interface to the planning system.

The starting point is to formulate the *problem* in terms of constraints (using GNU Prolog). Then the next step is to link our constraint solver with the visualization engine (Unreal Virtual Machine).

The next sub-section explains in more detail how the constraint solver has been implemented.

Constraint Solver

Problem

To experiment we have developed a test application in which the task is to allocate a Coke machine in a bank hall. This task can be seen as an example of a spatial/resource allocation problem. This can be represented as a Constraint Satisfaction Problem (CSP), which, consequently, can be easily formalised in CLP (FD). There are numerous design issues, especially in the context of building design, that can be easily transformed into constraints such as health and safety regulations, urban and planning regulations, etc.

The *Constraints* imposed on the Coke machines are the following: a) sources of heat (i.e. radiators) should stay a minimum distance away from the machines, b) the coke machines can only be placed at a maximum distance away from a socket (power point) c) the machines must not obstruct ventilation ducts, d) the coke machines must stay clear from exits and thoroughfares e) the minimum distance to a wall is 25 cm f) finally, the allocation of coke machines must take into account the existing furniture and decoration in the hall.

Representation of the Search Space

There are at least three options to choose from to represent the spatial search space: rectangular or hexagonal discretisation, actual polygonal floor and polygonal floor representation. There is no obvious choice, each representation has its trade-offs. This decision has great implications in terms of speed to solve the CSP problem and flexibility to accurately represent a constraint satisfaction problem.

In this prototype we have opted for a Rectangular or Hexagonal Grid: A uniform rectangular or hexagonal grid is overlaid onto the world. The size of each grid space is related to the smallest character. The pros of this approach are: obstacles and characters can be easily marked in the grid; it works well for our 3D world, in our example application the working (spatial search) space is a flat 3D world that is, the z coordinate is constant; save computations on the z coordinate. c

When we translate our CSP problem to a rectangular grid, we are simplifying the 3D space into more familiar 2D terms. The idea is to solve the problem in a pseudo 3D environment (where the Z coordinate is constant) while providing a full 3D representation to the user. To do this, we use a rectangular grid as a mesh to transform the 3D space into a more manageable search space.

We have experimented with different grid size but as far as the search space goes, the maximum number of cells or the finest possible mesh is implementation dependent. This is due to the fact that in GNU Prolog FD variables can only take values in their domains. There are two internal representations for an FD variable: interval and sparse representation. The initial representation for a FD variable is always internal representation and is switched to a sparse when a “hole” appears in the domain (i.e due to an inequality constraint). So to avoid unexpected surprises we have used the following predicate: fd_set_vector_max(512) that sets the environment variable VECTORMAX to the greatest value that any FD variable can take. This means that in our example application the finest possible mesh has a cell size of 37mm by 37 mm in real units, That is, in the real world.

Problem expressed in CLP(FD)

We can easily transform our initial task into a CSP problem. That is, a problem where one is given: a finite set of variables, a function which map each variable to a finite domain, and a finite set of constraints. In our case, the set of variable is (X,Y) where X and Y represent the coke machine’s x and y coordinates in our search space. We map these variables to a finite domain by using the following predicate: fd_domain(Vars,1,36). Finally, we have to impose our constraints.

We have regarded the constraints *d* and *f* as fixed

```
/*distance to a wall, in this case 25cm*/
walldistance(X,Y) :-
                        X=2,!
                        ;
                        Y=2,!
                        ;
                        X=35.
```

```
/*max distance to, i.e, a plug*/
/*In this case the Manhattan distance is 3 units*/
maxdistance(X,Y) :-
                        D is 3,
                        (Xc is 2,
                        Yc is 1,
                        distance(X,Xc,Dc1),
                        distance(Y,Yc,Dc2),
                        Dc1+Dc2#=<D)).
```

Figure 2. Examples of constraints

topological constraints and we have implemented them as facts in the database. Alternatively, we could have implemented them as atomic constraints such as Y#=<10 what could have been more efficient but for simplicity we chose not to. The constraints a and c have been regarded as conjunctive. The formulation of this type of constraints (conjunctive) is straight forward. See figure 3 where Xo and Yo are the position of the source of heat in the search space. Finally, b is regarded as a disjunctive constraint. The formulation of this type of constraints is also simple. See figure 2 where Xc and Yc represent the location of the socket in the search space. We have used the Manhattan distance as distance criterion.

It is worth noticed that the incremental constraint solving capabilities makes very easy to implement more conjunctive constraints. We just need to add a new constraint but with a different Xi and Yi and the solver takes care of the rest (see figure 3). Figure 4 shows a graphical representation of a fraction of the search space and the constraints implemented in it.

```
/*Where X0 and Yo is a ventilation duct*/
/*The criterion to used to calculate the distance is: */
/*The Manhattan distance*/

mindistV0(X,Y) :-
                    D is 1,
                    X0 is 1,
                    Y0 is 1,
                    distance(X,X0,D1),
                    distance(Y,Y0,D2),
                    D1+D2#>D.

/*A new constraint can be added just by changing */
/*te location of the ventilation duct, Xi Yi*/


mindistVi(X,Y) :-
                    D is 1,
                    Xi is 1,
                    Yi is 1,
                    distance(X,X0,D1),
                    distance(Y,Y0,D2),
                    D1+D2#>D.
```

Figure 3. Incremental Constraints

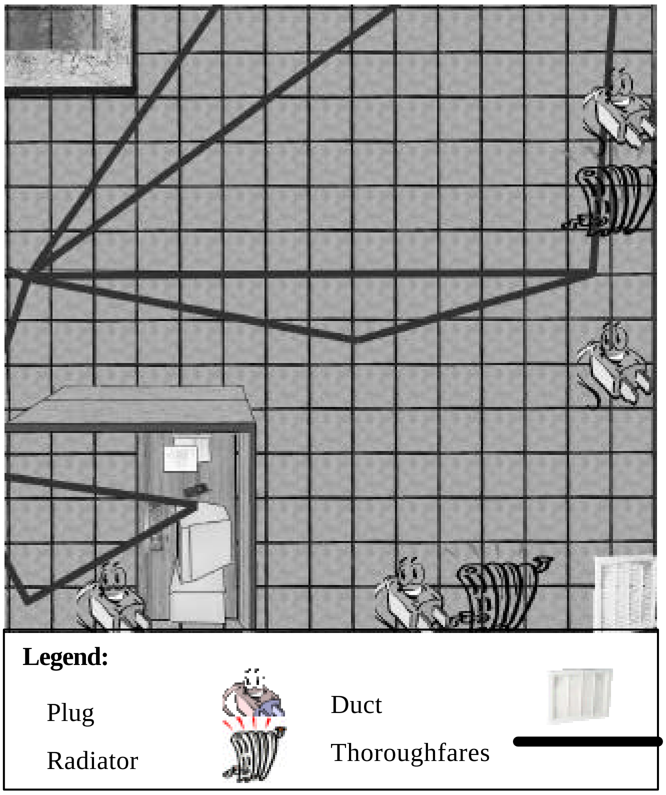

Figure 4. Fraction of the search space with constraints.

System Architecture

Concept Architecture Example

The system is constructed of a number of modules

-LinkServer

This module handles the communication of Unreal with the GNU Prolog solver and it has been implemented using the Berkley socket interface. In short, the aim of this subsystem is to provide a means of inter-process communication that allows bi-directional messages between two processes regardless of whether those processes reside on the same machine or different machines. In our current implementation, the solver is located in a remote location and the communication protocol used to send information across the network is TCP/IP (Transfer Control Protocol and Internet Protocol).

-Interaction Modules (ITModules)

Figure 5. Player explores virtual scene.

A series of Unreal modules have been scripted in order to create a satisfactory and appropriate player's interaction with the world. These modules provide the system with a customized player pawn which has embedded the following functionality: the ability to locate the sought object and Grabbing/Dropping objects at will by just pressing the appropriate key. Figure 5 shows a player exploring the

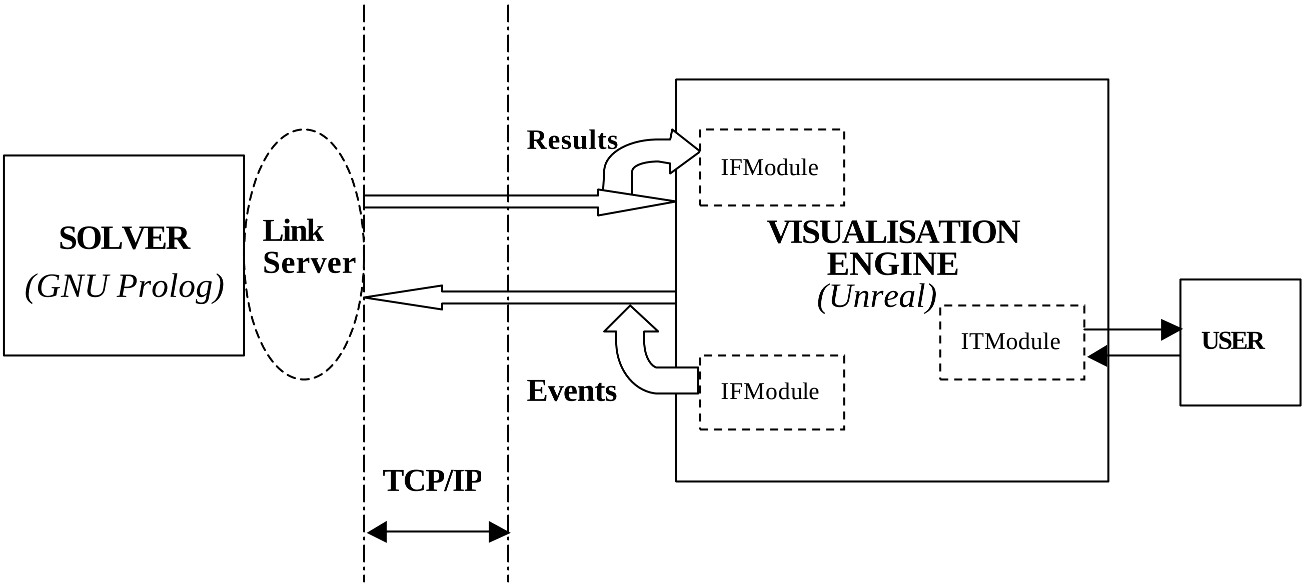

Figure 6 System Architecture.

virtual scene after he/she has “grabbed” a Coke Machine.

-Interface Modules (IFModules)

In order to have a smooth player’s interaction with the virtual environment, an interface with the LinkServer module has been embedded in our Player Pawn class. This approach guarantees that interface is switch on/off at the player’s will, that is, when the player decides to input a new configuration (i.e by dropping an object).

The basic functionality provided by this interface is the following: a) the player’s input configuration is parsed, transformed and send to client-server subsystem b) the new configuration produced by the solver is received from the client-server subsystem, parsed and transformed into UnrealScript commands in order to create the corresponding object configuration which the Unreal Virtual Machine displays in real-time. Figure 6 shows the overall system architecture.

Interaction cycle

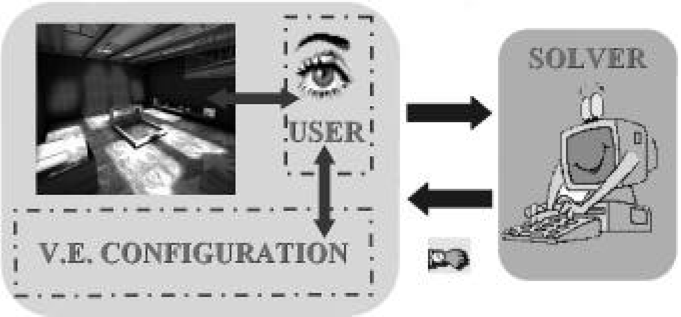

Figure 7. Intercation cycle.

The interaction cycle works as follows:

-Initial State (Initial Knowledge)

An initial configuration is proposed to the user. In our example, a Coke machine is initially allocated in the Virtual Environment.

-Exploratory State

The user explores the initial proposed configuration in the Virtual Environment.

-Solution State

To reach this state the u ser has previously decided to place the Coke machine in the Virtual Environment in a position different from the one initially given. Once the user has placed the Coke machine, the new input configuration is sent to the solver. This analyzes it and reacts to it. There are two possible scenarios:

a) The new configuration complies with the constraints. Outcome: the system reacts by displaying a confirmation message.

b) The new position does not comply with constraints embedded in the solver. Outcome: the system reacts by returning to the previous valid configuration.

Once the Solution State is completed the user is back to the Exploratory State and consequently, a new interaction cycle commences. Figure 7 depicts the interaction cycle.

FURTHER WORK

We see the results presented on this paper as a step towards the development of reactive environments In practical implementation terms in our example application the following could be an example of a reactive environment:
In our initial scenario we could have two types of constraints: hard constraints (i.e topological) and soft constraints (i.e movable objects). Thus, if player’s configuration complies with all the topological (hard constraints) but not with the soft constraints (movable objects) then the system should react by reallocating the movable objects in the new configuration.

DISCUSSION

One key aspect to successfully incorporate planning systems into Visualisation Engines is that AI techniques are often less interactive that it would be required for a complete integration into the virtual environment. In previous papers [3] we discussed and shown that CLP over FD as software technology it is fast enough to react in "real-time" to user's input configuration. Furthermore, the results shown in table1 underpinned the idea that although constraint programming in itself is not a reactive technique it can be used to emulate reactivity because it can produce a solution quickly enough.

Table1. Solution times.

	To find: First Solution(ms)	To find: All Solutions (ms)
User-time	0	[10-40]
System-time	0	[0-10]
Cpu-time	0	[10-30]
Real-time	[0-2]	[61-63]

Average time in milliseconds over 10 trials

Another key point in order to implement a real-time system is that the sampling rate of object manipulation in the virtual environment must be compatible with the result production granularity of the problem-solving algorithm. In our case, the implementation of a real-time reactive environment will depend not only on using the adequate technology to implement the constraint solver but also on the overall system architecture.

Unfortunately, at this point in time, we do not have enough experimental data to back up this particular point. However, our empirical estimation is that the current implementation produces a satisfactory result in terms of the compatibility of the object manipulation in the virtual environment with the overall result production granularity.

CONCLUSIONS

In this paper we have presented an interactive system in which the virtual environment acts as an interface to the interactive planning system. Because the nature of the tasks to be solved takes place in a 3 dimensional space, Visualisation Engines are suitable tools to serve as an interface to the planning and/or problem solving mechanisms. We have shown that the inclusion of an AI layer adds a problem-solving component to the Virtual Environment. This could also be seen as having a VE where objects have associated behaviours.

ACKNOWLEDGEMENTS

Many thanks to Mr Steve James Mead. for his technical assistance regarding C programming and system architecture.
Many thanks to Mr H. Robert Berry, Jr for his technical assistance regarding UnrealScript.

REFERENCES

[1] De Leon, Victor. (1999), VRND:NOTRE-DAME CATHEDRAL: A Globally Accessible Multi-User Real-Time Virtual Reconstruction, the 5th International *Conference on Virtual System and MultiMedia 1999 (VSMM'99):Next Generation Virtual Reality "Milestones for a New Virtual Milennium"*.Dundee, Scotland, UK, 1-3 September, 1999.

[2] Young, Micheal. (2001), An Overview of the Mimesis Architecture: Integrating Intelligent Narrative Control into an Existing Gaming Environment. In *The Working Notes of the AAAI Spring Symposium on Artificial Intelligence and Interactive Entertainment*, Stanford, CA, March 2001.

[3] Calderon, C and Cavazza, M. (2001), Intelligent Virtual Environments to Interactively Solve Spatial Configuration Tasks. *Proceedings* of *the Seventh International Conference on Virtual System and Multimedia*. Berkeley, USA, 25-27 Oct 2001