Ibero-American Symposium on Computer Graphics – SIACG (2006)
P. Brunet, N. Correia, and G. Baranoski (Editors)

Implementing Vision in the IViHumans Platform

P. Semião, M. B. Carmo, A. P. Cláudio

Faculdade de Ciências da Universidade de Lisboa, Portugal

Abstract

IViHumans is a platform to build virtual environments inhabited by autonomous virtual humans. The platform has two layers: the Graphical Processing Layer and the Artificial Intelligence Layer. This paper presents a synthetic vision algorithm based on ray casting that has been integrated into this platform.

Categories and Subject Descriptors (according to ACM CCS): I.3.7 [Computer Graphics] Animation

1. Introduction

The construction of 3D Virtual Environments populated by Intelligent Virtual Humans is a research area with applications in multiple areas such as education, medicine, training emergency situations or military applications, cultural heritage, urban planning and entertainment (games, movies).

The main issue is to conceive a synthetic representation of the virtual human with human-like appearance and capable of conveying emotion and personality and also able to interact with other virtual humans and with the environment. A key point to achieve a believable interaction is to endow the virtual human with mechanisms to perceive the surrounding world. Though tactile and auditory sensors have been studied [NT05], vision sensors are the usual approach. In this paper we describe a vision mechanism that has been integrated in a platform designed to build virtual environments inhabited by virtual humans [SPC*05] [CCC*05]. This platform is called IViHumans (**I**ntelligent **Vi**rtual **Humans**).

Several approaches have been proposed to simulate vision; in section 2 we review research work in this area. In section 3, the IViHumans platform is described. In section 4 the vision algorithm that has been implemented is explained. In section 5 work in progress is reported and in section 6 we state our conclusions.

2. Vision mechanisms

There are two major categories of vision mechanisms: sensory omniscience and sensory honesty.

Sensory omniscience methods access directly the scene database and the virtual human becomes aware of all the objects in the scene, even those that are behind its back. Although this type of vision is easier to implement and less time-consuming, it may produce unrealistic behaviours.

Sensory honesty methods try to acquire the information that can actually be seen by the virtual human, as if it was a real human. These methods lead to more realistic interactions because "Allowing a creature to potentially see through walls or through the back of its head – has a devastating effect on the illusion of life!" [BID*01]. As we aim for realistic interactions, sensory honesty was the chosen path. This group of vision mechanisms includes two main approaches: artificial vision and synthetic vision.

Artificial vision is the process of recognizing the image of the real environment captured by a camera [NT95], i. e., all the information obtained by the vision sensor about the world is extracted using 2D image analysis. This process requires time-consuming tasks such as image segmentation, recognition and interpretation [PS02]. It is used in robotics and artificial intelligence research areas.

The concept of synthetic vision, introduced by Renault et al. [RMT90], corresponds to the simulated vision of a virtual human shown by a camera located at the virtual human point of view. This method combines the identification of the visible objects from the scene rendering, through the virtual human point of view, with the query of the scene database to obtain the position and semantic information about those visible objects. Thus it may be the simplest and fastest way to extract useful information from the environment [PS02].

Renault et al. [RMT90] proposed a synthetic vision technique based in false colours. Objects in the scene are not rendered using their usual colours, they are rendered using a unique colour for each object. The scene is rendered from the point of view of the actor and the output is stored in a 2D array containing the pixel value at that

point, the distance from the actor's eye to the pixel and an object identifier of any object at that position. The size of the array is chosen to provide accuracy without consuming too much CPU time. A matrix 30x30 was used to apply this technique to avoid obstacles in a corridor. Noser et al [NT95] used a matrix 50x50 and improved this technique, including a visual memory model implemented with a 3D octree grid. In [KL99] false colour coding is applied with a rendered image of 200x200 pixels. In this case it was assumed that the environment was broken up into a collection of small to medium-sized objects, each assigned a unique ID. Large objects such as walls or floors were subdivided and each piece was assigned an ID. Each character maintains a kind of spatial memory storing a list of object IDs currently visible. Peters and Sullivan [PS02] extended false-coloured rendering providing different vision modes that use different colour palettes: distinct mode and grouped mode. In the distinct vision mode, each object is false-coloured with a unique colour. In the grouped vision mode, groups of objects share the same colour. Objects can be grouped according to different criteria. Group vision mode is less precise as a group of objects can be marked as visible if at least one of its objects is visible. In this case the rendered image is a matrix 128x128.

Terzopoulos et al. [TRG96] adopted a vision sensor, covering a 300-degrees spherical angle, to simulate fish vision. Each fish eye is implemented with four virtual cameras. These cameras render images with different resolutions and fields of view. The virtual fish is able to recognize a set of colour models of objects it is interested in, using a colour indexing algorithm.

Blumberg [B96] presents a synthetic vision technique based on motion energy balance inspired by research on bees' navigation. The scene is rendered from the actor point of view and the pixels values in successive frames are used to recover a gross measure of motion energy along with other features of the environment that are used to guide the movement.

A different technique of synthetic vision is presented in [VP05]. In this case the vision mechanism is based on ray casting. A number of rays are cast into the scene inside the agent's field of view. The agent is able to know the position, size and type of the objects that are intersected by the rays.

Our approach is based on ray casting as explained in section 4.

3. IViHumans platform

To generate virtual humans with believable appearance and behaviour, we have to combine a realistic appearance with a virtual brain able to simulate the behaviour of a human. To achieve this goal IViHumans platform has two layers: the Graphical Processing Layer and the Artificial Intelligence Layer. The Graphical Processing Layer supports the modelling and animation of both the virtual environment and the virtual human body. The Artificial Intelligence Layer simulates the "brain" of the virtual human, being responsible for the intelligent behaviour of the agent that is behind the virtual human.

As there are commercial, public domain and open source tools capable of performing the graphical processing, as well as suitable agent development frameworks, we have decided to select and assemble some of these already available tools. For the Graphical Processing Layer we need 3D-modelling and rendering capabilities, as well as the possibility to simulate rigid body dynamics. We adopted the following tools: Blender [hB] to perform 3D modelling, OGRE [hOG] as the rendering engine and the ODE [hOD] library to simulate rigid body dynamics.

In fact, the 3D modelling can be performed using any other tool. It is only required that the modelling tool exports the model in a format that can be read by the rendering engine (OGRE).

To build the Artificial Intelligence Layer the agent development framework JADE [hJ] was chosen.

As the graphical software components (OGRE and ODE) are developed in C++ and JADE is developed in Java, the Graphical Processing Layer was developed using C++ and the Artificial Intelligence Layer using Java. The two layers were connected using sockets.

The modularity of the architecture is an essential feature to support the future evolution of the platform. Moreover, it will be a reusable tool capable of giving human-like appearance to intelligent agents. Another important feature is the capability to be executed in a distributed environment provided by the socket connection between the Graphical Processing Layer and the Artificial Intelligence Layer.

4. Vision algorithm

Our main goal was to implement a vision mechanism that would guarantee sensory honesty. According to this approach, an agent is aware of an object in the world only if this object is indeed visible from the agent's perspective, in other terms, rendered by the agent's camera. It was also important that the agent became aware of the objects that entered its field of vision in a short amount of time. For the time being, we considered less than one second to be an adequate maximum delay. Furthermore, we defined some guidelines to help us evaluate the implementation progress of the vision mechanism.

As general guidelines, we wanted a system that was frame independent but still capable of adapting to low frame rates without loss of perception. These two guidelines are not contradictory although, at a first glance, they may seem so. Frame independence means that frame rate information should not be used to perform any calculations involved in the algorithm itself. As for the adaptation to low frame rates, suppose that the algorithm is designed to scan the agent's field of vision in X frames. Provided that we have a frame rate of more than X frames per second that will not be a problem. But if the frame rate drops bellow that value, we cannot guarantee the less than one second delay, as it will take more than that to scan all areas. At very low values we would not even be able to guarantee that the object would be seen at all.

Consequently, there should be a way to reduce the X value while trying to maintain a good perception.

As the rendering engine was already chosen, we started by doing a survey of the possibilities it offered. There are two: querying the GPU to find if a given object is rendered and sending rays to the scene.

In the first approach, the information about the visibility is given by the hardware responsible for the rendering. This is a very effective technique. However, it is hardware dependent, not all the GPUs are capable of handling such queries and, even among those that can, these methods have different implementation. We considered unacceptable to impose these restrictions. In the second approach, rays are thrown into the scene, through the viewport, and a query about the intersections that occur is performed. Information about the type of objects intersected can also be obtained. This has been the chosen approach because it is general and independent from the hardware.

To test the validity of the ray system we started with a simple goal: walls detection. A successful implementation would enable the agent to move around the scene with autonomy.

We prodded the agent on a perpetual forward motion, always on the same height level, and set about the task of orienting him through the scene. To accomplish this task three rays are thrown, one that points forward, and one to each side, reaching the limit of the agent's field of view. All the rays are cast in the same plane where the agent is moving in. After performing one query for each ray, we use the results to perform simple orientation. We try to maintain equidistance to both sides by slowly turning to the side further away, and, if the centre ray is too close to a wall then turn sharply to one side. This approach provides adequate visual information to the agent, enabling him to move without problems through the scene.

Then, we proceeded to a more complex sub-goal, the identification of objects in the scene, their nature and their location.

Throwing one ray at each point in the rendered image (pixel) would achieve instant identification of all objects. However, this is not feasible in real time with today's hardware. Taking into account that changes from frame to frame are very little, we have devised an approach that divides up the rays we wish to send among a number of frames. This number of frames should never be greater than the current frames per second (FPS) displayed and the number of rays thrown in each frame should not be so high as to significantly diminish the frame rate. To uniformly distribute the points of the rendered image by successive frames, we constructed a matrix, M, containing in each line a set of points at which rays are thrown in one frame. This matrix defines the original points used in successive frames, one line per frame. We also define an Array, AΔ, which contains the displacement vectors that are applied to the initial points of M along several iterations. The first vector in AΔ is (0,0), that is, no change is applied in the first iteration along the matrix. Further displacements should be chosen in different directions and lengths to avoid generating points hitting the same locations when the agent is moving.

In a more formal manner, let M [L, C] be a matrix of 2D points with L lines and C columns and AΔ(S) an array with S displacement vectors.

In each frame, C rays are thrown at the scene. Combining points in matrix M with displacements in array A, there will be L*C*S individual points targeted by rays along L*S frames.

The set of points targeted at frame F is the same set of points that will be used in frame F+(L*S*K), where K is any positive integer value.

The set of points used in frame F is

$$\{M [H, i] + A\Delta[D], i=1,2,...,C\}$$

where

$$H= ((F-1) \bmod L) + 1,$$

$$D= (((F-1) /L) \bmod S) + 1$$ and

/ represents integer division.

Let's give a simple example for clarity. Consider the following matrix and array

M [2,2] = { { (0.4, 0.4) , (0.6, 0.6) } ,
{ (0.4, 0.6) , (0.6, 0.4) } }

AΔ = { (0, 0) , (0.01, 0.01) , (-0.02, 0.02) }

The set of points at which rays are thrown in each frame is shown in table 1. When the last one is reached, it starts again from the beginning.

Frame id	Set of points
1	(0.4, 0.4), (0.6, 0.6)
2	(0.4, 0.6), (0.6, 0.4)
3	(0.41, 0.41), (0.61, 0.61)
4	(0.41, 0.61), (0.61, 0.41)
5	(0.38, 0.42), (0.58, 0.62)
6	(0.38, 0.62), (0.58, 0.42)

Table 1

5. Ongoing work

After performing several tests, we came to the conclusion that Ogre does not have the capability to provide accurate ray-to-object intersection. An object is represented by its bounding box. This volume is usually much larger than the object itself and therefore is prone to false intersections. To obtain more accurate intersections we chose OpCode [hOC] Optimized Collision Detection, a simple and fast collision library that meets our needs perfectly. We use the OgreOpCode [hOGC] wrapper to communicate with the library. After implementation, we performed some empirical tests that gave a first glimpse into the efficiency of the algorithm and proved that it performed well.

In spite of the good results, we are aware of some specific scenarios in which this kind of algorithm can either

fail or have poor results. Although we have a rather large amount of rays canvassing the screen, it is still possible for a small object to remain undetected, mainly if the agent does not move. Aiming for an algorithm capable of adapting on-the-fly we consider the following aspects: increasing or decreasing the number of rays used in a single frame, so that it can maintain a reasonable amount of FPS; assuring that every point in the screen will be the target of a ray, avoiding lack of detection of very small objects when the agent is stationary. We are currently performing extensive tests to study the algorithm performance by varying the number of objects in the scene and the number of rays thrown by frame. For illustrative purposes table 2 shows initial results of the number of FPS obtained in a few cases studied. The machine used for the tests is a Pentium 4 3.40GHz with 1GB of DDR2 RAM and a NVIDIA GeForce 7800 GTX (Gainward Golden Seal). We used Direct3D9 with a resolution of 800x600 in windowed mode. The scene has an average of 10000 triangles rendered at any single instance. The values given are expressed in average FPS.

		Number of Rays	
		30	50
Number of Objects	6	284	264
	12	279	261
	18	267	249

Table 2 –number of FPS

These numbers are closely tied to OGRE and OpCode. Since there are no available numbers on other algorithm's performances we can only assume that our strategy of "divide and conquer" would allow a greater number of points to be visited across several frames.

6. Conclusions

We have presented a synthetic vision algorithm based on ray casting that optimizes object detection, varying the target points in successive frames. Our preliminary tests indicate that this mechanism allows good coverage of all rendered area, maintaining high performance.

References

[B96] BLUMBERG, B.: Old Tricks, New Dogs: Ethology And Interactive Creatures. PhD Dissertation, MIT Media Lab, 1996

[BID*01] BURKE, R., ISLA, D., DOWNIE, M., IVANOV, Y., BLUMBERG, B.: Creature Smarts: The Art and Architecture of a Virtual Brain, The Game Developers Conference, 2001

[CCC*05] CARMO, M. B., CLÁUDIO, A. P., CUNHA, J. D., COELHO, H., SILVESTRE, M., PINTO-ALBUQUERQUE, M.: Plataforma de Suporte à Geração de Cenas Animadas com Agentes Inteligentes. In Actas do 13º Encontro Português de Computação Gráfica, pp. 79-84, 2005 (in portuguese)

[hB] http://www.blender3d.org/About/?sub=Features

[hJ] http://jade.tilab.com/

[hOC] http://www.codercorner.com/Opcode.htm

[hOD] http://ODE.org/

[hOG] http://www.ogre3d.org

[hOGC]http://conglomerate.berlios.de/wiki/index.php/Ogre Opcode_Introduction

[KL99] KUFFNER, J. J., LATOMBE, J.-C.: Perception-Based Navigation for Animated Characters in Real-Time Virtual Environments. The Visual Computer: Real-time Virtual Worlds, 1999

[NT95] NOSER, H., THALMANN, D.: Synthetic Vision and Audition for Digital Actors. Proc. Eurographics '95, Maastricht, pp 325-336, 1995

[PS02] PETERS, C., AND O'SULLIVAN, C.: Synthetic Vision and Memory for Autonomous Virtual Humans, Computer Graphics Forum, vol. 21, issue 4, pp 743, Dec. 2002

[RMT90] RENAULT, O., MAGNENAT-THALMANN, N., THALMANN, D.: A Vision-based Approach to Behavioural Animation. Journal of Visualization and Computer Animation, vol. 1, nº 1, pp.18-21, 1990

[SPC*05] SILVESTRE, M., PINTO-ALBUQUERQUE, M., CARMO, M. B., CLÁUDIO, A. P., CUNHA, J. D., COELHO, H: Platform for the Generation of Virtual Environments Inhabited by Intelligent Virtual Humans. In Proc. ACM ITiCSE'05, *student poster*s, pp 402, 2005.

[TRG96] TERZOPOULOS, D., RABIE, T., GRZESZCZUK, R.: Perception and Learning in Artificial Animals. Artificial Life V: Proc. Fifth International Conference on the Synthesis and Simulation of Living Systems, Nara, Japan, May, pp 313-320, 1996

[VP05] VOSINAKIS, S., PANAYIOTOPOULOS, T.: A Tool for Constructing 3D Environments with Virtual Agents. Multimedia Tools and Applications, 25, pp 253-279, 2005