# Using Synthetic Vision for Autonomous Non-Player Characters in Computer Games

Sebastian Enrique[1], Alan Watt[2], Steve Maddock[2], Fabio Policarpo[3]

[1] Departamento de Computación, Facultad de Ciencias Exactas y Naturales, UBA, Planta Baja, Pabellón 1, Ciudad Universitaria, (1428) Buenos Aires, Argentina
senrique@dc.uba.ar
[2] Computer Graphics Research Group, Department of Computer Science, Regent Court, 211 Portobello Street, University of Sheffield, Sheffield, S1 4DP, United Kingdom
a.watt@dcs.shef.ac.uk, s.maddock@dcs.shef.ac.uk
[3] Paralelo Computaçao Ltda., Rua Otávio Carneiro 100, sala 912, Icaraí, Niterói, RJ, CEP 24230-190, Brazil
fabio@paralelo.com.br

**Abstract.** In this paper, we discuss the role and utility of synthetic vision in computer games. We present an implementation of a synthetic vision module based on two viewports rendered in real-time, one representing static information and the other dynamic, with false colouring being used for object identification, depth information and movement representation. We demonstrate the utility of this synthetic vision module by using it as input to a simple rule-based AI module that controls agent behaviour in a first-person shooter game.

## Introduction

We can regard synthetic vision as a process that supplies an autonomous agent with a 2D view of his environment. The term synthetic vision is used because we bypass the classic computer vision problems. As Thalmann *et al* [1] point out, we skip the problems of distance detection, pattern recognition and noisy images that would be appertain for vision computations for real robots. Instead computer vision issues are addressed in the following ways:

1) Depth perception – we can supply pixel depth as part of the synthetic vision of an autonomous agent's vision. The actual position of objects in the agent's field of view is then available by inverting the modeling and projection transform.
2) Object recognition – we can supply object function or identity as part of the synthetic vision system.
3) Motion determination – we can code the motion of object pixels into the synthetic vision viewport.

Thus the agent AI is supplied with a high-level vision system rather than an unprocessed view of the environment. As an example, instead of just rendering the agent's view into a viewport then having an AI interpret the view, we instead render objects in a colour that reflects their function or identity (although there is nothing to prevent an implementation where the agent AI has to interpret depth – from binocular

vision, say, and also recognize objects). With object identity, depth and velocity presented, the synthetic vision becomes a plan of the world as seen from the agent's viewpoint.

We can also consider synthetic vision in relation to a program that controls an autonomous agent by accessing the game database and the current state of play. Often a game database will be tagged with extra pre-calculated information so that an autonomous agent can give a game player an effective opponent. For instance, areas of the database may be tagged as good hiding places (they may be shadow areas), or pre-calculated journey paths from one database node to another may be stored. In using synthetic vision, we change the way the AI works, from a prepared, programmer-oriented behaviour to the possibility of novel, unpredictable behaviour.

A number of advantages accrue from allowing an autonomous agent to perceive his environment via a synthetic vision module. First, it may enable an AI architecture for an autonomous agent that is more 'realistic' and easier to build. Here we refer to an 'on board' AI for each autonomous agent. Such an AI can interpret what is seen by the character, and only what is seen. Isla and Blumberg [2] refer to this as *sensory honesty* and point out that it "…forces a separation between the actual state of the world and the character's view of the state of the world". Thus the synthetic vision may render an object but not what is behind it.

Second, a number of common games operations can be controlled by synthetic vision. A synthetic vision can be used to implement *local* navigation tasks such as obstacle avoidance. Here the agent's global path through a game level may be controlled by a high level module (such as A* path planning or game logic). The local navigation task may be to attempt to follow this path by taking local deviations where appropriate. Also, synthetic vision can be used to reduce collision checking. In a games engine this is normally carried out every frame by checking the player's bounding box against the polygons of the level or any other dynamic object. Clearly if there is free space ahead you do not need every-frame collision checking.

Third, easy agent-directed control of the synthetic vision module may be possible, for example, look around to resolve a query, or follow the path of a moving object. In the case of the former this is routinely handled as a rendering operation. A synthetic vision can also function as part of a method for implementing inter-agent behaviour.

Thus, the provision of synthetic vision reduces to a specialized rendering which means that the same technology developed for fast real-time rendering of complex scenes is exploited in the synthetic vision module. This means that real-time implementation is straightforward.

However, despite the ease of producing a synthetic vision, it seems to be only an occasionally employed model in computer games and virtual reality. Tu and Terzopoulous [3] made an early attempt at synthetic vision for artificial fishes. The emphasis of this work is a physics-based model and reactive behaviour such as obstacle avoidance, escaping and schooling. Fishes are equipped with a "cyclopean" vision system with a 300 degree field of view. In their system an object is "seen" if any part of it enters the view volume and is not fully occluded by another object. Terzopoulos *et al* [4] followed this with a vision system that is *less synthetic* in that the fishes' vision system is initially presented with *retinal* images which are binocular photorealistic renderings. Computer vision algorithms are then used to accomplish, for example, predator recognition. This work thus attempts to model, to some extent, the animal visual processes rather than bypassing these by rendering semantic information into the viewport.

In contrast, a simpler approach is to use false colouring in the rendering to represent semantic information. Blumberg [5] use this approach in a synthetic vision based on image motion energy that is used for obstacle avoidance and low-level navigation. A formula derived from image frames is used to steer the agent, which, in this case, is a virtual dog. Noser *et al* [6] use false colouring to represent object identity, and in addition, introduce a dynamic octree to represent the visual memory of the agent. Kuffner and Latcombe [7] also discuss the role of memory in perception-based navigation, with an agent planning a path based on its learned model of the world.

One of the main aspects that must be addressed for computer games is to make the synthetic vision fast enough. We achieve this, as discussed above, by making use of existing real-time rendering speed-ups as used to provide the game player with a view of the world. We propose that two viewports can be effectively used to provide for two different kinds of semantic information. Both use false colouring to present a rendered view of the autonomous agent's field of view. The first viewport represents static information and the second viewport represents dynamic information. Together the two viewports can be used to control agent behaviour. We discuss only the implementation of simple *memoryless* reactive behaviour; although far more complex behaviour is implementable by exploiting synthetic vision together with the consideration of memory and learning. Our synthetic vision module is used to demonstrate low-level navigation, fast object recognition, fast dynamic object recognition and obstacle avoidance.

## Representations for static and dynamic objects

As we have discussed, a simple synthetic vision scheme simply means exploiting the ability of a games engine to efficiently render a complex scene in real-time. We only need to choose a representation for the existing information that we intend to make available to the agent's AI. We consider two types of synthetic vision: static and dynamic.

### Static Synthetic Vision

The static synthetic vision is mainly useful to identify objects, taking the form of a viewport with false colouring, similar to that described in [7]. Each class of object is assigned a unique colour id. As an example, all health power-up objects are part of the same class of objects and are assigned the colour (1,1,0) in the RGB model. Non-player characters (NPCs) are assigned the colour (0,0.5,0) and spaceships are assigned (0,0.7,0.7). Different colour ids are also used for the level geometry. Polygons that form the floor are assigned colour (0,1,0) wall polygons (0,0,1) and ceiling polygons (1,0,0). In order to identify such polygons, we use the z component of each polygon's face normal, since the z world coordinate points to the sky. Polygons are identified as floor polygons when the (normalized) normal's z component is greater than or equal to 0.8, ceiling polygons when it is less than or equal to -0.8, and wall polygons otherwise (as long as such polygons are not part of a special object, e.g. a power-up). Figure 1 illustrates a rendering of a scene using our false colouring approach.

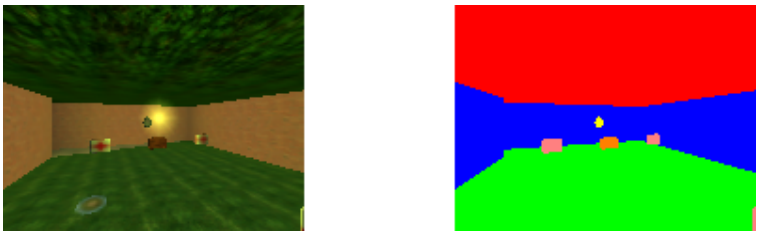

**Fig. 1.** On the left is the actual world rendered from the character's point of view. On the right, the world is rendered as using the synthetic vision module from the same point of view. Pink is used for a health power-up, orange for an ammunition power-up, yellow for a flying ball, green for the floor; blue for the wall and red for the ceiling.

Using the Z-buffer, the second feature that we can obtain is the depth information of each pixel in the static viewport. Since we know from the 3D engine the near and far rendering planes, and the character position, it is possible to calculate the exact depth of any other object.

**Dynamic Synthetic Vision**

The dynamic synthetic vision supplies instantaneous movement information for the objects that the character can see. A simple AI module could use this to determine if some object is approaching the character or not; a complex one could take the dynamic information on a per-frame basis, and predict future position for dynamic objects with erratic trajectories.

As for the static synthetic vision, the dynamic synthetic vision takes the form of a viewport with false colouring, but in place of object ids the colours represent the velocity vector of each object. Using a RGB model, the colour of each pixel in the viewport is determined as follows:

Red = r(x,y,||V||)

r : [0, ScreenWidth)x[0,ScreenHeight)x[0,MaxVel] → [0,1]

r is a direct mapping into [0,1] using the magnitude of the velocity vector (V) of the object located at screen coordinates (x,y). MaxVel is a 3D engine constant that represents the maximum velocity allowed.

Green = g(x,y,V,D)

g : [0, ScreenWidth)x[0,ScreenHeight)

x{[0,1]x[0,1]x[0,1]}x{[0,1]x[0,1]x[0,1]} → [0,1]

g is a direct mapping into [0,1] using the cosine between the normalized velocity vector (V) of the object located at screen coordinates (x,y) and the normalized character direction vector (D).

Blue = b(x,y,V,D)

b : [0, ScreenWidth)x[0,ScreenHeight)

x{[0,1]x[0,1]x[0,1]}x{[0,1]x[0,1]x[0,1]} → [0,1]

b is a direct mapping into [0,1] using the sine between the normalized velocity vector (V) of the object located at screen coordinates (x,y) and the normalized character direction vector (D).

With this information, the velocity (direction) vector of each object in the static viewport is shown in the dynamic viewport. The two are easily cross-referenced using the (x,y) coordinates. Figure 2 gives an example of the dynamic vision. Both the static and dynamic vision systems are implemented in the Fly3D game engine[1] [8], using a resolution of 160x120 pixels.

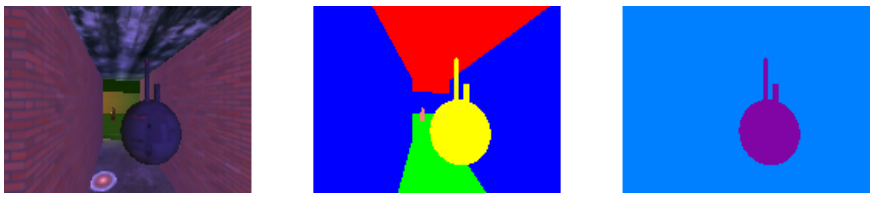

**Fig. 2.** The left image is the world rendered from the character's point of view. The centre image shows the static viewport. The right image shows the dynamic viewport. The unique dynamic object in this screenshot is the flying-ball, with colour (0.5,0.02,0.65), since it has half the maximum velocity, and is going away from the character.

## Implementing (memoryless) reactive behaviour with the static viewport

The aim of this section is to demonstrate how simple reactive or rule-based behaviour implementation flows easily from a synthetic vision module. We are developing a very simple AI module that uses our synthetic vision approach in order to control a character's behavior. The character starts in a game-like world with 100% of Ammo and Health in the initial state: *Walking Around*. The character can take Ammo or Health power-ups that increase the value in a fixed quantity, but which never surpasses 100%. Over time Ammo and Health fall independently in a linear way. If the Health reaches 0, the character dies. We can identify six different states:

### State 1: Walking around and obstacle avoidance

This is the initial state, when health and ammo are above an upper threshold, say, 80%. In this state the AI chooses a target or destination point for the character in the static viewport and the games engine constructs a path formed from one or more Bezier curves. The character is then animated along this path (Figure 3) with the character's orientation per frame derived from the tangent vector at the current position. (The use of Bezier curves enables continuous paths to be constructed piecewise and gives a natural looking global control for a walk animation. Full details of how the curves are derived in practical situations are given in [9]). The simplest possible case is shown in Figure 3a where a single segment curve is derived from the character's current position and orientation and a destination point and orientation.

[1] Fly3D Engine website: http://www.fly3d.com.br/. Paralelo Computaçao

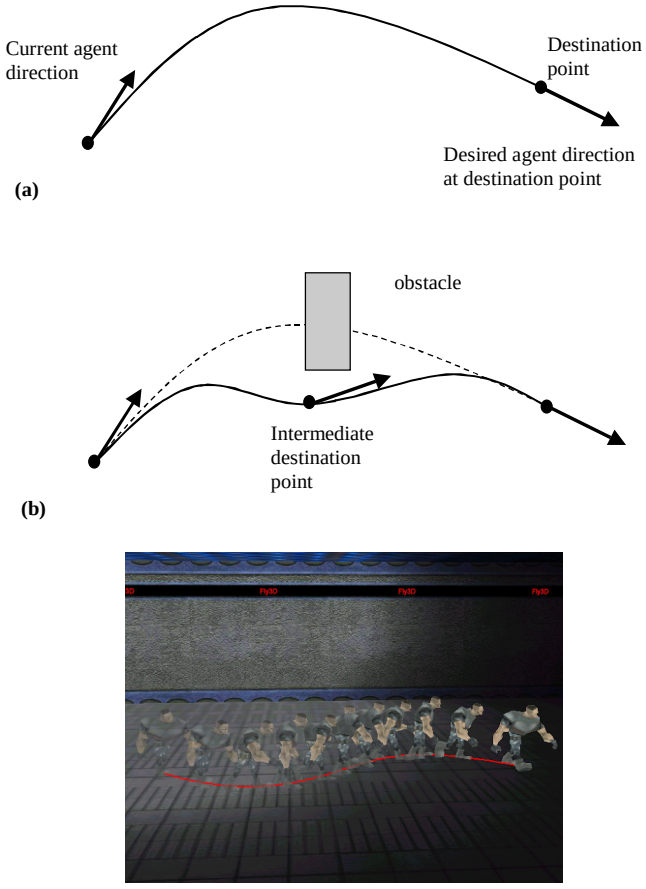


**Fig. 3.** a) Construction of a Bezier path; b) Bezier paths and obstacle avoidance; c) A character walking along a Bezier curve

Obstacle avoidance is easily implemented using information derived from the synthetic vision. If the destination point is obscured by an obstacle in the static viewport, then an intermediate destination point can be ascertained by choosing to go around the obstacle to the right or left and using the bounding boxes of the obstacle and the character.

### State 2 and 3: Looking for a Power-Up

State 2 is when health is below the upper threshold and above the lower threshold, say, 20%, and ammo is above the upper threshold. State 3 is when health is above the upper threshold, and ammo is under the upper threshold, but above the lower threshold.

Like state 1, if the character is idle or has walked a known fraction of the path, he tries to choose a new target point. The target point will be a power-up if one exists on

screen; if not, state 1 is reentered. To see if a power-up is present on screen, all the pixels in the static viewport are scanned to locate the power-up colour id. The process uses a tolerance value, together with depth-values, to estimate if a point of the same colour corresponds to the same object, or to another power-up of the same class. Then, the AI chooses the closest object, and selects a floor point below that object, but with a similar depth value, as the destination point.

**State 4: Looking for Any Power-Up**

In this state, both health and ammo are between the upper and lower thresholds. The AI module processes this state in a similar way to that for states 2 and 3. The difference is that it iterates looking for both ammo and health colour ids, and chooses to go to the item that is closer to the autonomous character. If no item is seen in the viewport, it walks around using the state 1 process.

**State 5 and 6: Looking Quickly for a Power-Up**

State 5 is when health is above the lower threshold and ammo is under the lower threshold. State 6 is when health is under the lower threshold. The same method used in states 2 and 3 applies here, but the unique difference is that the character runs rather than walks.

## Implementing (memoryless) reactive behaviour with the dynamic viewport

Here, reaction depends on the object class and the dynamic information. As an example we can identify and pursue an opponent. Identification is done using the static viewport and pursuit is achieved by calculating an intersection position to move to from the opponent's velocity information in the dynamic viewport (assuming that the opponent is moving in a straight line). The intersection point can be updated every frame or every n frames. If a Bezier path is to be used for the pursuer (see previous section) then an approximate arc length estimation is required and the pursuer is animated at the appropriate speed along the curve to arrive at the intersection point at the same time as the opponent.

We could integrate this behaviour with the ammo and health considerations in the scenario in the previous section. If the character is in states 1 to 4, he will chase the opponent and fight, using the dynamic information to estimate where the character is going. If the character is in states 5 to 6, the low health or ammo states, the preference would be to run away as fast as possible. In each case the AI module needs to resolve questions such as: Where to run? How far? For how much time?

## Conclusions

Our synthetic vision approach is powerful enough to be used by AI modules to create more realistic NPCs in computer games. Such autonomous NPCs sense the environment rather than have direct access to the game database. In considering how

'sensory honest' our synthetic vision is, a number of issues are worth considering. Our synthetic vision is 'perfect', in that lighting conditions are not considered in the false coloured rendering. Thus an object in shadow, which should be difficult to see, is nonetheless rendered using the same false colour as if the object was not in shadow. The same issue concerns the dynamic viewport. Perhaps noise could be added, so that the synthetic vision would not have exact information. For the dynamic viewport this would mean only an estimation of movement would be available. Having implemented a memoryless agent, the challenge now is to add memory and learning, and possibly other senses such as hearing.

Finally, it may be that building perception modules such as vision and hearing is a powerful approach to autonomous agent AI. After all, in most cases, we want to imitate human or superhuman behaviour.

## References


1. D. Thalmann, H. Noser, Z. Huang. Interactive Animation, chapter 11, pp.263-291, Springer-Verlag, 1996.
2. D. Isla, B. Blumberg. New Challenges for Character-Based AI for Games. AAAI Spring Symposium on AI and Interactive Entertainment, Palo Alto, CA, March 2002.
3. X. Tu and D. Terzopoulos. Artificial Fishes: Physics, Locomotion, Perception, Behavior. Proc. of ACM SIGGRAPH'94, Orlando, FL, July, 1994, in ACM Computer Graphics Proceedings, 1994, p.43-50.
4. D. Terzopoulos, T. Rabie, R. Grzeszczuk. Perception and learning in artificial animals. Artificial Life V: Proc. Fifth International Conference on the Synthesis and Simulation of Living Systems, Nara, Japan, May, 1996, 313-320.
5. B. Blumberg. Go With the Flow: Synthetic Vision for Autonomous Animated Creatures. Proceedings of the First International Conference on Autonomous Agents (Agents'97), ACM Press 1997, pp.538,-539.
6. H. Noser, O. Renault, D. Thalmann, N. Magnenat Thalmann. Navigation for Digital Actors based on Synthetic Vision, Memory and Learning, Computers and Graphics, Pergamon Press, Vol.19, No1, 1995, pp.7-19.
7. J.J. Kuffner, J.C. Latombe. Fast Synthetic Vision, Memory, and Learning Models for Virtual Humans. In Proc. CA'99: IEEE International Conference on Computer Animation, pp. 118-127, Geneva, Switzerland, May 1999.
8. A. Watt, F. Policarpo. 3D Computer Games Technology, Volume I: Real-Time Rendering and Software. Addison Wesley. 2001. ISBN 0-201-61921-0.
9. A. Watt, F. Policarpo. 3D Games: Advanced Real-time Rendering and Animation, Addison-Wesley 2002 (in press).