

An Investigation Into the Use of Synthetic Vision for NPC's/Agents in Computer Games

Autor:

Sebastian Enrique

Departamento de Computación, Facultad de Ciencias Exactas y Naturales, Universidad de Buenos Aires, Planta Baja, Pabellón 1, Ciudad Universitaria, (1428) Buenos Aires, Argentina
senrique@dc.uba.ar

Director:

Alan Watt

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

Co-Director:

Marta Mejail

Departamento de Computación, Facultad de Ciencias Exactas y Naturales, UBA, Planta Baja, Pabellón 1, Ciudad Universitaria, (1428) Buenos Aires, Argentina
marta@dc.uba.ar

Resumen

Son discutidos en esta tesis la utilidad y el rol de la visión sintética en juegos de computadora. Se presenta una implementación de un módulo de visión sintética basado en dos viewports renderizados en tiempo real, uno representando información estática y el otro dinámica, utilizando colores falsos para identificación de objetos, información de profundidad y representación del movimiento. La utilidad de este módulo de visión sintética es demostrada utilizándolo como entrada a un módulo simple de IA basado en reglas que controla el comportamiento de un agente en un juego de acción en primera persona.

1. Introducción

Podemos considerar la visión sintética como el proceso que provee a un agente autónomo de una vista en 2D de su ambiente. El término visión sintética es utilizado porque son evitados los clásicos problemas de la visión por computadora. Como Thalmann *et al* [Thal96] señalan, salteamos los problemas de detección de distancias, reconocimiento de formas, e imágenes ruidosas que surgirían para cómputos de visión en robots reales. En su lugar, los problemas de la visión por computadora son tratados de las siguientes maneras:

- 1) Percepción de la profundidad – podemos proveer la profundidad del píxel como parte de la visión sintética de un agente autónomo. La posición actual de los objetos en el campo visual del agente es disponible entonces invirtiendo las matrices de transformación de modelado y proyección.
- 2) Reconocimiento de objetos – podemos proveer función o identificación de objetos como parte del sistema de visión sintética.
- 3) Determinación de movimiento – podemos codificar el movimiento de los objetos en los píxeles del viewport de visión sintética.

De este modo, la IA (Inteligencia Artificial) del agente es provista con un sistema de visión de alto nivel en lugar de una vista del ambiente sin procesar. Por ejemplo, en lugar de simplemente renderizar la visión del agente en un viewport y teniendo un módulo de IA que lo interprete, renderizamos los objetos con un color que refleja su función o identificación (a pesar de que no hay nada que impida una implementación donde la IA del agente tenga que interpretar la profundidad a partir de, digamos, visión binocular y reconocer además los objetos). Con identificación de objetos, profundidad, y velocidad, la visión sintética se convierte en un plano del mundo visto desde el punto de vista del agente.

Podemos considerar también la visión sintética en relación a un programa que controla un agente autónomo accediendo a la base de datos del juego, con su estado actual. Usualmente la base de datos del juego estará ‘marcada’ con información extra precalculada de modo tal de que el agente autónomo sea un oponente efectivo para el jugador. Por ejemplo, sectores de la base de datos podrían estar marcados como buenos lugares para ocultarse (podrían ser zonas en penumbras), o caminos precalculados entre un nodo de la base de datos y otro podrían guardarse. Con el uso de la visión sintética, cambiamos el modo en que la IA trabaja, de un comportamiento preparado, orientado al programador, a un comportamiento novel, impredecible.

Ciertas ventajas surgen al permitir a un agente autónomo percibir su ambiente a través de un módulo de visión sintética. Primero, podría orientar el desarrollo de una arquitectura de IA para un agente autónomo más realista y fácil de construir. La referiremos como una IA ‘on board’ para cada agente autónomo. Tal IA puede interpretar qué es lo que, y sólo lo que, está viendo el personaje. Isla and Blumberg [Isla02] llaman a esto *honestidad sensorial* y señalan que “...fuerza una separación entre el estado actual del mundo y la visión del personaje del estado actual del mundo”. La visión sintética puede renderizar un objeto, pero no lo que está detrás de él.

Segundo, cierta cantidad de operaciones comunes en los juegos podrían ser controladas por la visión sintética. Una visión sintética podría ser utilizada para implementar tareas de navegación local como evasión de obstáculos. El camino global del agente a través de un nivel de un juego podría estar controlado por un módulo de más alto nivel (como por ejemplo un algoritmo para plan de caminos A*). La tarea de navegación local podría intentar seguir ese camino tomando desviaciones locales cuando es apropiado. La visión sintética podría ser utilizada a su vez para reducir el chequeo de colisiones. En un motor de juegos la detección de colisiones es llevada a cabo normalmente en cada frame chequeando el bounding box del jugador con los polígonos del nivel o de cualquier otro objeto dinámico. Claramente si existe espacio libre hacia adelante del personaje, no es necesario hacer un chequeo en cada frame para ver si se produjo una colisión.

Tercero, un fácil control de dirección del agente con el módulo de visión sintética es posible; por ejemplo, mirar alrededor para resolver una consulta, o seguir el camino de un objeto en movimiento. El primer caso es resuelto normalmente como una operación de rendering. La visión sintética también podría funcionar como un método para implementar comportamiento mutuo entre agentes.

Por lo tanto, la provisión de una visión sintética se reduce a un rendering especializado, lo que significa que la misma tecnología desarrollada para el renderizado rápido de escenas en tiempo real es explotada en el módulo de visión sintética, generando una directa implementación.

Sin embargo, más allá de la relativa facilidad de producir una visión sintética, aparenta ser un modelo ocasionalmente empleado en realidad virtual o juegos de computadora. Tu y Terzopoulos [TuTe94] la usaron para peces artificiales. El énfasis de su trabajo fue un modelo basado en física y comportamiento reactivo como evasión de obstáculos, *escaping* y *schooling*. Los peces están equipados con un sistema de visión “ciclópeo” con un campo de visión de 300°. En su sistema un objeto es visto si una parte del mismo entra en el volumen de visión y no está completamente oculto por otro objeto. Terzopoulos et al [Terz96] siguieron el trabajo con un sistema de visión que es *menos sintético* en el sentido que el sistema de visión de los peces es presentado con imágenes *retinales* que corresponden a renderings binoculares fotorrealísticos. Algoritmos de visión por computadora son entonces usados para cumplir, por ejemplo, reconocimiento de depredadores. Por lo tanto, este trabajo

intenta modelar, hasta cierto punto, los procesos de la visión animal en lugar de evitar los problemas conocidos de visión por computadora con el uso de información semántica en un viewport.

Por el contrario, un enfoque más simple es usar falso coloreo en el rendering para representar información semántica. Blumberg [Blum97a] [Blum97b] utiliza ese enfoque en una visión sintética basada en energía de movimiento de la imagen, para poder evitar obstáculos y navegar a bajo nivel. Para conducir el agente, que en este caso es un perro virtual, se utiliza una fórmula derivada a partir de varios frames. Noser *et al* [Nose95a] [Nose95b] usan falso coloreo para representar identificación de objetos, además de introducir un octree dinámico para representar la memoria visual del agente. Kuffner y Latombe [Kuff99a] [Kuff99b] discuten también el rol de la memoria en una navegación basada en percepciones, con un agente planeando el camino basándose en lo que aprendió del modelo del mundo.

Uno de los principales aspectos que debe ser cumplido para juegos de computadora es hacer la visión sintética lo suficientemente rápida. Esto es logrado, como fue discutido más arriba, haciendo uso de la tecnología de aceleración actual para el rendering en tiempo real del mismo modo que se utiliza para brindarle al jugador una visión del mundo. Proponemos en este trabajo dos viewports que pueden ser usados efectivamente para proveer dos tipos diferentes de información semántica. Ambos usan falso coloreo para representar una vista renderizada del campo de visión del agente autónomo. El primer viewport representa información estática y el segundo información dinámica. Ambos viewports pueden ser usados en conjunto para controlar el comportamiento del agente. Realizamos únicamente la implementación de un simple comportamiento reactivo sin memoria; a pesar de que comportamiento mucho más complejo es implementable explotando el concepto de la visión sintética junto con el uso de memoria y aprendizaje. Nuestro módulo de visión sintética es mostrado en este trabajo con su uso para la navegación a bajo nivel, rápido reconocimiento de objetos, tanto estáticos como dinámicos, y evasión de obstáculos.

2. Descripción del Problema

Definiremos nuestro concepto de Visión Sintética como el sistema visual de un personaje virtual que vive en un mundo virtual en 3D. Esta visión representa lo que el personaje ve del mundo, la parte del mundo vista a través de sus ojos. Técnicamente hablando, toma la forma de la escena renderizada desde su punto de vista.

Sin embargo, para tener un sistema de visión útil en juegos de computadora, es necesario encontrar una representación de la visión de la cual pueda obtenerse la suficiente información como para tomar decisiones realistas velozmente en forma autónoma.

La visión sintética pura no es útil para los juegos de hoy en día, ya que la cantidad de información útil que puede obtenerse en tiempo real está prácticamente limitada al reconocimiento de algunas formas y a la evasión de obstáculos. Este es un campo de investigación por sí mismo. Refiérase a la literatura de visión por computadora para tener una mejor idea de los problemas con que debe lidiarse en su uso.

Un sistema sin visión es lo que no queremos.

Entonces, nuestra tarea consiste en encontrar un modelo que se ubique entremedio de ambos extremos, y que sea útil para su uso en juegos de computadora.

3. Modelo de Visión Sintética

Apuntamos a crear un rendering especial desde el punto de vista del personaje de modo tal de generar una representación del mundo en viewports con el objetivo de proveer a la IA facultad de producir:

- Evasión de obstáculos.
- Navegación a bajo nivel.
- Reconocimiento veloz de objetos.
- Reconocimiento veloz de objetos dinámicos.

Debemos entender ‘veloz’ en una forma subjetiva, dependiente de resultados, como ‘suficientemente rápida’ para producir un trabajo aceptable en juegos de computadora en tiempo real en 3D.

Para alcanzar tales metas, proponemos un sistema de visión sintética con dos viewports: el primero para representar información estática, llamado static viewport o viewport estático, y el segundo para representar información dinámica, denominado dynamic viewport o viewport dinámico.

Asumimos un modelo RGB de 24 bits de representación del color de cada píxel.

3.1 Viewport Estático

El static viewport es principalmente útil para identificación de objetos, tomando la forma de un viewport con falso coloreo similar al descrito en [Kuff99a] [Kuff99b].

Un objeto es un ítem con forma 3D y masa. Un clase de objetos es un agrupamiento de objetos con misma naturaleza y propiedades. En este contexto, por ejemplo, el power-up de energía ubicado cerca de la fuente en un nivel imaginario de un juego es un objeto, mientras que todos los power-ups de energía del mismo nivel forman una clase de objetos. Cada clase de objetos tiene un único color id asociado. Dos clases de objetos cualesquiera nunca tienen el mismo color asociado.

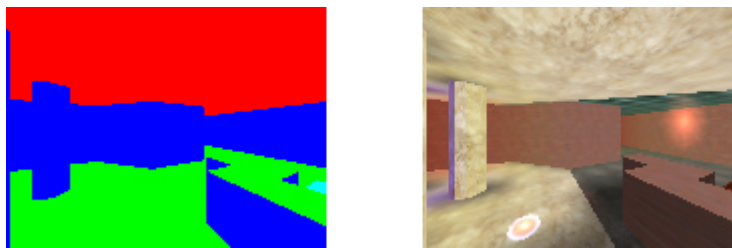


Figura 1. Static viewport (izquierda) obtenido a partir del viewport rederizado en forma normal (derecha).

Además de los objetos definidos anteriormente, los juegos de computadora 3D diferencian los objetos normales o ítems, y la geometría del nivel o *level geometry*, que son los polígonos que forman parte de la estructura del nivel en sí. Por ejemplo: el piso, paredes, etc. Dividiremos a esos polígonos en tres clases, y cada una de ellas llevará un color identificador único –distinto a los de cada clase de objetos–:

- Piso : Todo polígono de la level geometry con componente Z de la normal normalizada mayor o igual a 0.8.
- Techo : Todo polígono de la level geometry con componente Z de la normal normalizada menor o igual a -0.8.
- Pared : Todo polígono de la level geometry que no cumple ninguno de los dos casos anteriores.

Asumimos para ello un sistema de coordenadas tal que la coordenada Z apunta hacia arriba, X hacia la derecha, e Y hacia delante.

<i>Clase de Objetos</i>	<i>Color (R,G,B)</i>
Health Power-Ups de Energía	(1, 1, 0)
Power-Ups de Municiones	(0, 1, 1)
Enemigos	(1, 0, 1)
Pared	(0, 0, 1)
Piso	(0, 1, 0)
Techo	(1, 0, 0)

Tabla 1. Tabla de mapeo entre clases de objetos y colores, extendida con las clases de la level geometry.

Finalmente, podemos crear una tabla de mapeo con el objetivo de conocer a qué color corresponde cada clase de objetos, y viceversa. Vea un ejemplo en la tabla 1.

Además de la identificación de objetos, es necesario conocer algo sobre su posición. Algunas de las variables que todo motor de juegos 3D utiliza y para las cuales tenemos un uso específico son: el ángulo de la cámara, y el plano más cercano y el más lejano del view frustum volume (NP y FP). La posición de los objetos puede conocerse combinando las variables anteriores con la información de profundidad de cada píxel renderizado en el static viewport. ¿Cómo conocemos la profundidad? Cuando el static viewport está siendo renderizado, se estará usando un depth buffer o buffer de profundidad (Z-buffer) para saber si se tiene que dibujar o no un píxel dado. Cuando el render termina, cada píxel visto del viewport estático tiene un valor en el depth buffer correspondiente a su profundidad.

En la figura 1 se muestra un ejemplo del viewport renderizado en forma normal y el correspondiente viewport estático.

3.2 Viewport Dinámico

El viewport dinámico provee información de movimiento instantáneo de los objetos que son vistos en el static viewport. Similar a éste último, toma la forma de un viewport con falso coloreo pero, en lugar de identificación, los colores representan el vector velocidad de cada objeto. Como utilizamos el modelo RGB, debemos definir cómo representar esa velocidad según cada componente de color. Definimos:

Si $V_{x,y}$ es el vector velocidad del objeto ubicado en las coordenadas (x, y) del static viewport, y $V_{máx}$ la velocidad máxima permitida,

$$R(x,y) = \text{mín}(\|V_{x,y}\| / V_{máx}, 1)$$

Si D es el vector de dirección/visión del agente,

$$D^N = D / \|D\|$$

$$V_{x,y}^N = V_{x,y} / \|V_{x,y}\|$$

$$c = V_{x,y}^N \cdot D^N = V_{x,y}^N_1 D^N_1 + V_{x,y}^N_2 D^N_2 + V_{x,y}^N_3 D^N_3$$

$$G(x,y) = c * 0.5 + 0.5$$

$$B(x,y) = s = \sqrt{1 - c^2}$$



Figura 2. De izquierda a derecha, static viewport, viewport normal, y dynamic viewport.

De este modo tenemos en la componente roja la magnitud del vector velocidad; en la azul el coseno del ángulo entre los vectores de velocidad del objeto y de visión del agente; y en la roja el seno. En la figura 2 se presenta un ejemplo del viewport dinámico así definido, donde el objeto que parece una esfera está en movimiento, y el resto del ambiente está estático.

El viewport dinámico tiene el mismo tamaño que el estático, 160 píxeles de ancho por 120 de alto.

4. Módulo de Inteligencia Artificial

Tener la información estática y dinámica de lo que se está viendo en un momento dado es útil solo si se tiene un cerebro que sepa cómo interpretar esa información, procesarla, decidir y actuar de acuerdo a un comportamiento. La visión sintética es simplemente una capa que observa el mundo a través de los ojos del personaje y los representa de un modo útil. La capa encargada de tomar como entrada esa información sensada por el módulo de visión sintética, procesarla, y actuar es lo que llamamos el cerebro o el módulo de IA.

Para demostrar un posible uso del módulo de visión sintética hemos desarrollado un pequeño módulo de inteligencia artificial, que intenta dar un comportamiento autónomo a un NPC dentro de un juego del género FPS (First Person Shooter).

4.1. Definición de un FPS

Nuestro FPS estará habitado por el personaje principal al que le daremos vida autónoma, llamado Bronto. Contará con dos propiedades intrínsecas: Energía (H) y Munición o Armas (W). A pesar de que contenga munición, en nuestra implementación no posee la facultad de disparar.

La energía de Bronto varía entre 0 y 100 (su valor inicial). Si alcanza un valor de cero, Bronto muere. Las municiones también varían entre 0 y 100 (su valor inicial). Si llega a cero, lo único que significa es que Bronto no tiene munición.

Además de que Bronto no puede disparar, tampoco podrá recibir disparos enemigos. Dadas estas dos simplificaciones, hemos decidido de que tanto la energía como las municiones disminuyan su valor linealmente en forma discreta, durante el tiempo.

Bronto aumenta ambas propiedades mediante la recolección de objetos especiales, llamados power-ups. Nuestro FPS contará únicamente con dos power-ups: Energía y Munición (o Armas). Cuando Bronto pase 'por encima' de alguno de ellos, se disparará un evento que hará que la propiedad correspondiente de Bronto se vea incrementada por una constante fija.

4.2. Definición del Comportamiento

Describiremos ahora el comportamiento que Bronto asumirá durante el juego basándose en los valores de las propiedades de energía y municiones.

Sean:

$H_{ut} \in \mathbb{N}$, el umbral superior de energía

$H_{lt} \in \mathbb{N}$, el umbral inferior de energía

$W_{ut} \in \mathbb{N}$, el umbral superior de municiones

$W_{lt} \in \mathbb{N}$, el umbral inferior de municiones

H y W, los valores de las propiedades de energía y municiones

tal que:

$$0 < H_{lt} < H_{ut} < 100$$

y

$$0 < W_{lt} < W_{ut} < 100$$

Bronto estará en alguno de los siguientes seis estados posibles:

- Walk Around (WA), cuando $H_{ut} \leq H$ y $W_{ut} \leq W$. H_{ut} y W_{ut} denotan un límite por encima del cual Bronto no tiene ningún objetivo específico y su accionar se reduce a caminar sin rumbo fijo.
- Looking for Health (LH), cuando $H_{lt} \leq H < H_{ut}$ y $W_{ut} \leq W$. Cuando Bronto está estable de municiones, por encima de W_{ut} , y empieza a sentir necesidad de energía, entre H_{lt} y H_{ut} , el objetivo de Bronto será recolectar power-ups de energía.

- Looking for Weapon (LW), cuando $H_{it} \leq H$ y $W_{it} \leq W < W_{ut}$. Cuando Bronto está estable de energía, por encima de H_{it} , y empieza a sentir necesidad de municiones, entre W_{it} y W_{ut} , el objetivo de Bronto será recolectar power-ups de municiones.
- Looking for Any Power-Up (LHW), cuando $H_{it} \leq H < H_{ut}$ y $W_{it} \leq W < W_{ut}$. Cuando Bronto siente necesidad tanto de energía como de municiones, su objetivo será recolectar cualquier power-up.
- Looking Quickly for Weapon (LQW), cuando $H_{it} \leq H$ y $W < W_{it}$. Cuando Bronto siente una necesidad extrema de municiones, porque su cantidad cayó del umbral inferior W_{it} , su objetivo será recolectar lo más rápido posible power-ups de municiones.
- Looking Quickly for Health (LQH), cuando $H < H_{it}$. Cuando Bronto siente una necesidad extrema de energía, porque su cantidad cayó del umbral inferior H_{it} , se impone ante cualquier otro objetivo el de recolectar lo más rápido posible power-ups de energía, porque su caída a cero significaría la muerte.

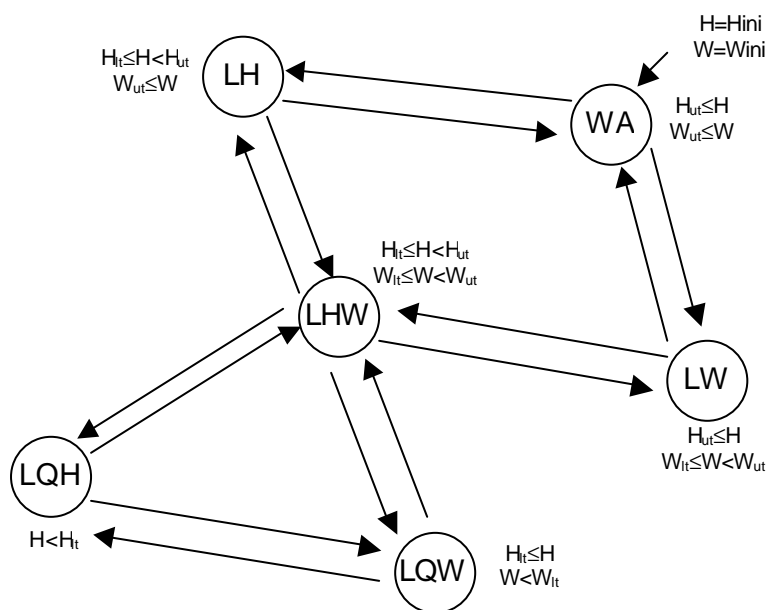


Figura 3. Diagrama de estados del comportamiento de Bronto. Sólo los cambios de estados esperados están representados con flechas, a pesar de que es posible el cambio de cualquier estado a otro.

En la figura 3 se representa mediante un diagrama de estados reducido, el comportamiento de Bronto. A pesar de que es posible ir de cualquier estado a otro, sólo los cambios que se esperan sean más usuales están indicados con flechas; es decir, un cambio de Walk Around a Looking Quickly for Health debería producirse por una disminución abrupta de energía, de $H \geq H_{ut}$ pasar a $H < H_{it}$, que con las condiciones planteadas para nuestro FPS junto con valores balanceados de H_{ut} y H_{it} no debería producirse.

4.3. Solución al Comportamiento

Para poder resolver el comportamiento en todos los estados, utilizaremos sólo el viewport de visión estática, donde podremos obtener la información que nos interesa que es la presencia de power-ups e información para navegabilidad y evasión de obstáculos.

Básicamente el proceso consiste en frame por frame analizar la información brindada por el viewport estático, incluida la profundidad, y seleccionar una coordenada destino del viewport que corresponda a level geometry, específicamente piso, con el cual se hará una desproyección para obtener las coordenadas del mundo del punto elegido, y generar una curva de Bezier entre la posición actual de Bronto y la destino, que será el camino a seguir.

El detalle de cómo se realiza la desproyección y el cálculo de la curva de Bezier puede encontrarse en el texto completo de la tesis, en este resumen explicaremos a continuación sólo las heurísticas para obtener los puntos de destino según el estado en el que se encuentra Bronto.

4.3.1. Walk Around

Para el comportamiento de caminar sin rumbo fijo hemos desarrollado una simple heurística que, si bien no produce resultados extraordinarios, son satisfactorios para esta demostración.

En el estado Walk Around, Bronto elige un nuevo destino –coordenadas (x, y) del viewport de visión sintética- si está quieto o si ya ha alcanzado o superado cierto porcentaje de su camino actual. Ese porcentaje lo denominamos:

$$C_f \in \mathbb{R}, 0.0 < C_f \leq 100.0$$

Otra constante que se utiliza es el ancho medio de Bronto, que en coordenadas del mundo estará dado por el ancho medio de su bounding box. Sin embargo, lo que realmente se utiliza es una estimación del ancho medio de Bronto medido en píxeles del viewport de visión sintética. Este valor lo denominamos:

$$B_{bbr} \in \mathbb{N}, 0 < B_{bbr} < 80$$

Este valor es útil porque Bronto tiene cierto espesor el cual será determinante para que no se quede trabado al intentar pasar, por ejemplo, por pasillos angostos donde él no quepa.

Para brindar mayor flexibilidad, también se utilizan otras dos constantes que servirán para fijar márgenes fuera de los cuales no será factible tomar un punto de destino:

$B_{waupd} \in \mathbb{N}, 0 < B_{waupd} < 80$, margen lateral y superior, en píxeles, del viewport de visión sintética; y

$B_{walpd} \in \mathbb{N}, 0 < B_{walpd} < 80$, margen inferior, en píxeles, del viewport de visión sintética.

En la figura 4 se representan estos márgenes con respecto al viewport.

Por último, se utiliza la información del viewport de visión estática, definida como una matriz de 160 filas por 120 columnas, donde la primera fila, fila 0, corresponde a la línea inferior del viewport. La información del viewport será accedida como:

$vps[i][j]$, $i, j \in \mathbb{N}_0$, donde $0 \leq i < 160$ y $0 \leq j < 120$, siendo i la columna y j la fila.

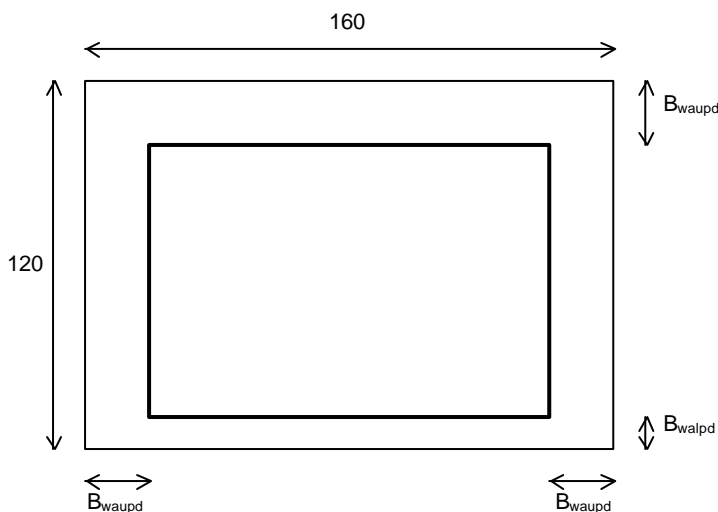


Figura 4. El viewport estático y los márgenes utilizados en Walk Around. Los píxeles fuera del recuadro más grueso no podrán ser elegidos como puntos de destino.

Si corresponde elegir un nuevo destino, la heurística intenta encontrar un rectángulo cuyo lado inferior coincida con la línea inferior del viewport estático, tenga un ancho de $(2 * B_{bbr} + 1)$ y una altura mínima de $(b_{walpd} + b_{waupd})$, y que esté formado en su totalidad –no sólo el perímetro– por el color correspondiente al piso. A cada rectángulo que cumpla la condición planteada lo llamaremos **camino libre**. Vea un ejemplo en la figura 5.

Si encuentra un camino libre –más adelante, cuando se presente el algoritmo, veremos la estrategia que se emplea– se elige como coordenada de destino x al centro del rectángulo, y como coordenada y a la altura menos el margen superior (b_{waupd}) del camino libre más alto.

Si la heurística falló al encontrar un camino libre, se intenta girar hacia izquierda o derecha al azar; y si esto falla también, no se elige ningún punto de destino.

Walk Around también plantea que si Bronto recorrió el 100% de un camino elegido y no tiene nuevas coordenadas de destino, cosa que podría suceder al no haber encontrado ningún camino libre desde el C_f del recorrido actual hasta que se cumple en su totalidad, gire 180° hacia la izquierda o la derecha.

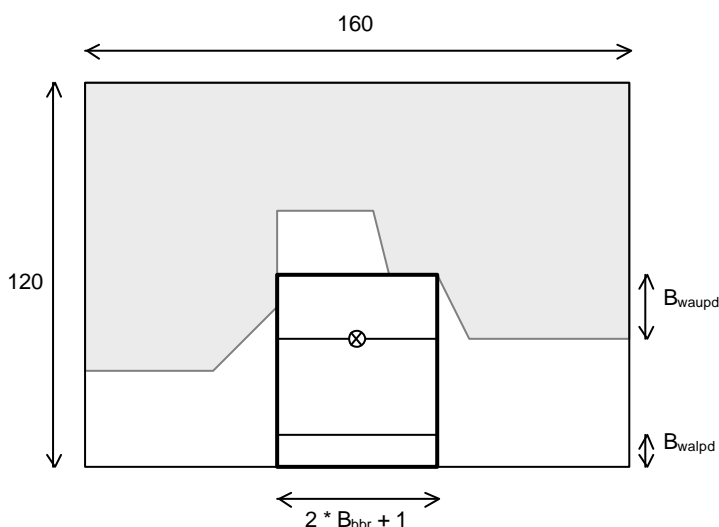


Figura 5. Un camino libre en el viewport estático. La zona en blanco corresponde a piso, la grisada a pared. El rectángulo de línea gruesa marca el camino libre elegido, dentro de él pueden apreciarse las líneas de los márgenes superior e inferior, y el punto que la heurística seleccionará como el nuevo destino.

El pseudocódigo de Walk Around es como sigue:

Entrada:

vps[160][120] : Una matriz con el contenido de cada píxel del viewport estático. La fila 0 corresponde a la línea inferior del viewport.

B_{bbr} : El ancho medio estimado de Bronto en píxeles del viewport.

B_{waupd} : Margen lateral y superior del camino satisfactorio en píxeles del viewport.

B_{walpd} : Margen inferior del camino satisfactorio en píxeles del viewport.

C_f : Porcentaje de camino recorrido a partir del cual se elige un nuevo destino.

Salida:

Un nuevo destino en coordenadas (x, y) del viewport, si se encontró uno satisfactorio.

Si aún no se recorrió al menos el $C_f\%$ del recorrido actual, retornar.

// Inicialización de flag de falla, fail, en la búsqueda de un camino satisfactorio.

fail = false

// Inicialización de flag de nuevo destino encontrado, se asume inicialmente verdadero.
newDestination = true

// freeway es un predicado que indica si hay algún camino libre que potencialmente sea satisfactorio. Un ejemplo de camino libre puede verse en la figura 5.

freeway = $\exists x_0, y_G \in \mathbb{N}_0, 0 \leq x_0 < x_1 < 160 \wedge (x_1 - x_0) = (2 * B_{bbr}) \wedge 0 \leq y_G < (120 - b_{waupd} - b_{walpd}) \wedge (\forall i, j \in \mathbb{N}_0, 0 \leq i < (2 * B_{bbr} + 1) \wedge 0 \leq j < (y_G + b_{waupd} + b_{walpd}), vps[x_0 + i, j] = \text{'piso'})$

// Si no hay ningún camino libre, entonces la búsqueda es insatisfactoria: falla.
Si (freeway = false) entonces fail = true

// Elijo un camino libre si alguno de los que existen me resultan satisfactorios, según la estrategia de búsqueda, caso contrario seteo el flag de falla

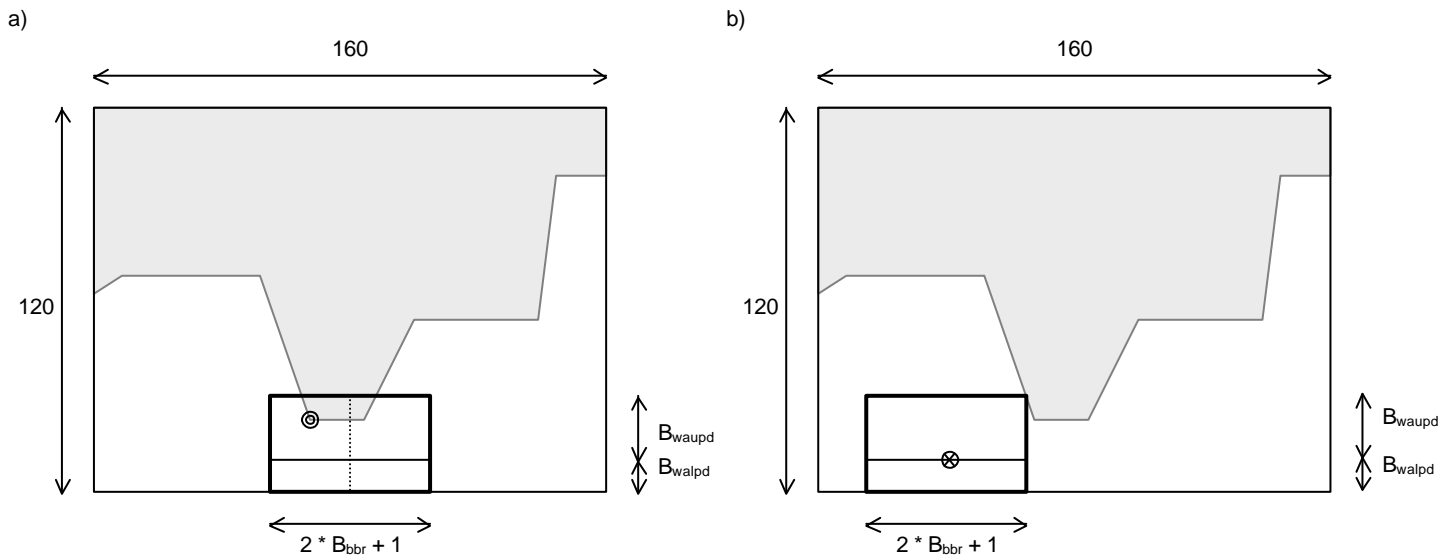


Figura 6. La estrategia 2 en acción. En a) puede observarse cómo se da la precondition para que se emplee esta estrategia: no existe un camino central libre y la línea más cercana a la parte inferior del viewport dentro del recuadro de búsqueda que contiene un píxel distinto de piso, contiene un píxel distinto de piso más cercano al lateral izquierdo que al derecho, marcado con un doble círculo. La línea punteada separa el recuadro de búsqueda en dos mitades. En b) se muestra el camino libre que elegirá finalmente la estrategia, junto con el nuevo destino.

Si (freeway = true) entonces

// Estrategia 1: Elijo, si existe, el camino libre central. En la figura 5 se representa un camino libre central.

Si ($x_0 = 79 - B_{bbr}$) lo hace verdadero entonces Elegir ese x_0 .

// Estrategia 2: Elijo, si existe, el camino libre más a la derecha de la mitad izquierda del viewport. Esto sucede cuando no existe un camino libre central y en la fila más cercana a la parte inferior del viewport que contiene al menos un píxel distinto de piso, existe un píxel distinto de piso más próximo al lateral izquierdo que al lateral derecho del recuadro de búsqueda. Vea un ejemplo en la figura 6. La estrategia falla cuando luego de darse la precondition, no encuentra ningún camino libre en la parte izquierda del viewport. Esto último se ejemplifica en la figura 7.

Si ($\exists i, j, k \in \mathbb{N}_0, 0 < i < j \leq B_{bbr} \wedge 0 \leq k < (b_{waupd} + b_{walpd}) \wedge (\forall w, s \in \mathbb{N}_0, (79 - B_{bbr}) \leq w < (79 + B_{bbr}) \wedge 0 \leq s < k \wedge vps[w, s] = \text{'piso'}) \wedge (\forall t, u \in \mathbb{N}_0, (79 - B_{bbr}) \leq t \leq (79 - B_{bbr} + i) \wedge vps[t, k] = \text{'piso'} \wedge (79 + B_{bbr} - j) < u \leq (79 + B_{bbr}) \wedge vps[u, k] = \text{'piso'}) \wedge vps[79 + B_{bbr} - j, k] \neq \text{'piso'})$ entonces Elegir el máximo $x_0, x_0 < 79$, tal que hace verdadero a freeway.

// Estrategia 3: Elijo, si existe, el camino libre más a la izquierda de la mitad derecha del viewport. Esta estrategia es una simetría a la estrategia 2. Se utiliza cuando no existe un camino libre central y en la fila más

cercana a la parte inferior del viewport que contiene al menos un píxel distinto de piso, existe un píxel distinto de piso más próximo al lateral derecho que al lateral izquierdo del recuadro de búsqueda. La estrategia falla cuando luego de darse la precondition, no encuentra ningún camino libre en la parte derecha del viewport.

Si $(\exists i, j, k \in \mathbb{N}_0, 0 < j < i \leq B_{bbr} \wedge 0 \leq k < (b_{waupd} + b_{walpd}) \wedge (\forall w, s \in \mathbb{N}_0, (79 - B_{bbr}) \leq w < (79 + B_{bbr}) \wedge 0 \leq s < k \wedge vps[w, s] = \text{'piso'}) \wedge (\forall t, u \in \mathbb{N}_0, (79 - B_{bbr}) \leq t < (79 - B_{bbr} + i) \wedge vps[t, k] = \text{'piso'} \wedge (79 + B_{bbr} - j) \leq u \leq (79 + B_{bbr}) \wedge vps[u, k] = \text{'piso'}) \wedge vps[79 - B_{bbr} + i, k] \neq \text{'piso'})$ entonces Elegir el mínimo x_0 , $x_0 > 79$, tal que hace verdadero a freeway.

// Estrategia 4: Se establece "falla" cuando se cree que no es posible avanzar hacia adelante por ser un pasaje muy angosto o porque hay una pared. Sucede cuando no existe un camino libre central y en la fila más cercana a la parte inferior del viewport que contiene al menos un píxel distinto de piso, existen píxeles, en un caso especial el mismo, distinto de piso a la misma distancia en píxeles del lateral derecho que del lateral izquierdo del recuadro de búsqueda. Pueden encontrarse ejemplos en el texto completo de la tesis.

Si $(\exists i, k \in \mathbb{N}_0, 0 < i \leq B_{bbr} \wedge 0 \leq k < (b_{waupd} + b_{walpd}) \wedge (\forall w, s \in \mathbb{N}_0, (79 - B_{bbr}) \leq w < (79 + B_{bbr}) \wedge 0 \leq s < k \wedge vps[w, s] = \text{'piso'}) \wedge (\forall t, u \in \mathbb{N}_0, (79 - B_{bbr}) \leq t < (79 - B_{bbr} + i) \wedge vps[t, k] = \text{'piso'} \wedge (79 + B_{bbr} - i) < u \leq (79 + B_{bbr}) \wedge vps[u, k] = \text{'piso'}) \wedge vps[79 + B_{bbr} - i, k] \neq \text{'piso'}) \wedge vps[79 - B_{bbr} + i, k] \neq \text{'piso'})$ entonces fail = true

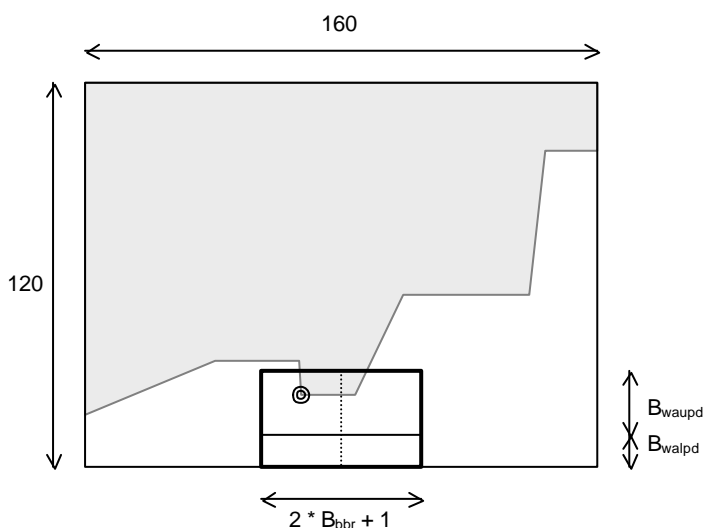


Figura 7. La estrategia 2 cuando falla. Puede observarse cómo se da la precondition para que se emplee esta estrategia. Sin embargo, al buscar un camino libre hacia la izquierda del punto marcado con un doble círculo, no encontrará ninguno.

// Si uno de los caminos resultó satisfactorio, establezco las coordenadas (x, y) en el viewport estático del punto destino.

Si fail = false entonces Tomar el máximo y_G que hace verdadero a freeway con el x_0 elegido; $x = x_0 + B_{bbr} + 1$; $y = y_G + b_{walpd}$.

// Si ningún camino resultó satisfactorio, intenta girar. El detalle de los algoritmos para el giro puede encontrarse en el texto completo de la tesis.

Si fail = true entonces Girar(x, y)

// Si pudo elegirse un nuevo destino mediante las estrategias utilizadas, lo establezco

Si newDestination = true

Establecer nuevo destino (x, y)

// Si no hay nuevo destino y ya se recorrió el 100% del último destino tomado, es decir, si se está quieto, darse vuelta

Si newDestination = false y Bronto está quieto entonces rotar hacia derecha o izquierda, al azar, 180°.

4.3.2. Looking for a Specific Power-Up

El concepto empleado cuando Bronto debe buscar un power-up es, al no mantener memoria de lo que ha visto en el pasado, simplemente revisar si hay algún power-up del buscado en el viewport estático actual. Si existe, dirigirse al más cercano, si no, utilizar Walk Around.

Como en Walk Around, utilizaremos el factor C_f que indica el porcentaje del camino recorrido actual a partir del cual se elige un nuevo destino.

$$C_f \in \mathfrak{R}, 0.0 < C_f \leq 100.0$$

El pseudocódigo del algoritmo podría resumirse a los siguientes pasos:

1. Si aún no se recorrió al menos el $C_f\%$ del recorrido actual, retornar.
2. Crear una lista *objectlist* de los power-ups de la clase buscada que se vean en el viewport estático en ese momento.
3. Si no hay power-ups de la clase buscada, entonces utilizar Walk Around.
4. Si hay al menos uno, obtener de *objectlist* el power-up p más cercano a Bronto.
5. Obtener un píxel d que corresponda a piso, esté línea recta hacia abajo de p , y tenga aproximadamente el mismo valor de profundidad.
6. Si no existe ese d , entonces utilizar Walk Around.
7. Si existe, establecer d como las nuevas coordenadas de destino.

Como puede inferirse del paso 5, también es necesario el uso de la información de profundidad brindada por el viewport estático.

Para el paso 2, crear una lista de power-ups presentes en la pantalla no es una tarea sencilla para realizar con precisión utilizando exclusivamente la información brindada por los viewports tal cual fueron definidos. Surgen las siguientes preguntas:

- ¿Si dos o más objetos de la misma clase forman una única área de color en el viewport estático, cómo diferenciar uno de otro?
- ¿Cómo saber si una única área de color corresponde a uno o más objetos?

Considerando la situación de que la identificación de power-ups es útil en nuestro caso sólo para elegir un punto de destino, no es demasiado importante si donde realmente hay sólo un objeto, se estima que hay más. Lo que realmente importa es saber dónde hay un objeto y saber cuál es el más cercano.

La heurística que desarrollamos para ello es la siguiente:

Sean

$\Delta x, \Delta y \in \mathfrak{N}_0$, cantidad de píxeles de tolerancia en el viewport estático para los ejes x e y.

$\Delta d \in \mathfrak{R}, 0 \leq \Delta d \leq 1$, tolerancia en el depth buffer para la profundidad.

Se recorren los píxeles del viewport de izquierda a derecha y de abajo hacia arriba haciendo:

Si el píxel p es un power-up de la clase buscada entonces

Para cada objeto o de la lista *objectlist*

Si $p.x \in [o.x - \Delta x, o.x + \Delta x]$ y

$p.y \in [o.y - \Delta y, o.y + \Delta y]$ y

$p.depth \in [o.depth - \Delta d, o.depth + \Delta d]$ entonces

Se trata de un píxel del objeto o , dejar de comparar objetos y seguir con el próximo píxel del viewport.

Si p no pertenecía a ningún objeto, agregar un objeto o' a *objectlist* con:

$o'.x = p.x; \quad o'.y = p.y; \quad o'.depth = p.depth;$

Los valores de tolerancia que en nuestra implementación resultaron ser razonables fueron:

Siendo PUr el radio del bounding box de cada power-up, NP la distancia del plano más cercano al viewpoint en coordenadas del mundo, y FP la del más lejano,

$$\Delta x = \Delta y = P_{Ur} * o.depth$$

$$\Delta d = P_{Ur} * (NP / FP)$$

Para el paso 4, simplemente se obtiene de la lista de objetos creada en el paso 2, aquél objeto que tenga menor valor de profundidad.

Por último, el pseudocódigo del paso 5 es muy sencillo:

Sea o el objeto elegido como destino, hacer

DestinationFound = False

$y_{coord} = o.y - 1$

Mientras $y_{coord} \geq 0$ y DestinationFound = False

Si $depth[o.x, y_{coord}] \in [o.depth - \Delta d', o.depth + \Delta d']$ entonces

DestinationFound = True

Fin Mientras

Si DestinationFound = True entonces

Establecer nuevas coordenadas de destino $x = o.x$; $y = y_{coord}$.

Nótese que se hace uso de la matriz que contiene el depth buffer (buffer de profundidad); y de una tolerancia $\Delta d'$ arbitraria.

4.4. Comportamiento Extendido con Reacciones Dinámicas

Para utilizar la información provista por el viewport dinámico se desarrolló un modulo de IA muy simple, basado en reglas y reactivo.

Definimos los siguientes tres estados:

$$\text{Intercept} \Leftrightarrow \text{Enemigo Presente} \wedge H \geq H_{ut} \wedge W \geq W_{ut}$$

$$\text{Avoid} \Leftrightarrow \text{Enemigo Presente} \wedge (H < H_{ut} \vee W < W_{ut}) \wedge \cos(\theta) < 0$$

$$\text{Don't Worry} \Leftrightarrow \text{Enemigo No Presente} \vee (\text{Enemigo Presente} \wedge (H < H_{ut} \vee W < W_{ut}) \wedge \cos(\theta) \geq 0)$$

Las definiciones anteriores se extienden a más de un enemigo en forma natural. Revise las secciones previas para encontrar las definiciones de las variables utilizadas.

$\cos(\theta)$ está mapeado en la componente de color verde. Entonces, podemos escribir las condiciones como:

$$\cos(\theta) < 0 \Leftrightarrow \text{Enemigo.Verde} < 0.5$$

$$\cos(\theta) \geq 0 \Leftrightarrow \text{Enemigo.Verde} \geq 0.5$$

Cuando $\cos(\theta) < 0$, el enemigo viene hacia Bronto; cuando $\cos(\theta) \geq 0$ el enemigo se aleja.

Conceptualmente, cuando Bronto está 'lleno' de energía y armas, esto es, sus valores están por encima del umbral superior, se siente en perfectas condiciones como para perseguir el enemigo y, entonces, trata de interceptarlo (*intercept*).

Cuando Bronto tiene algunas de sus propiedades por debajo del umbral superior, y el enemigo viene hacia él, necesita evitarlo (*avoid*).

Si no hay enemigos presentes, o si Bronto tiene alguna de sus propiedades por debajo del umbral superior, pero el enemigo se aleja, no se preocupará (*don't worry*) y continuará con su comportamiento estático normal.

El diagrama de estados dinámico se representa en la figura 8.

Para determinar si un enemigo está presente se busca su color id en el viewport estático.

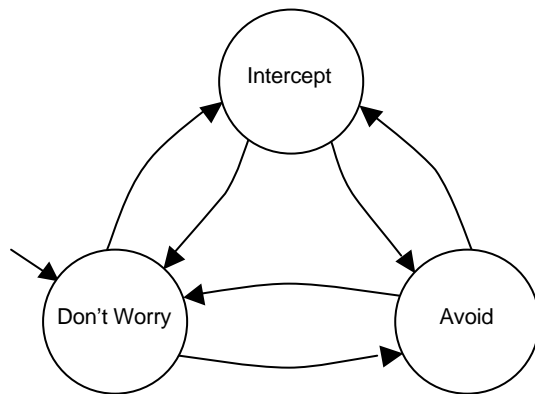


Figura 8. Diagrama de Estados Dinámico. Inicialmente Bronto está en el estado Don't Worry, luego pueden producirse transiciones entre cualquiera de los estados dependiendo de los valores de sus propiedades de energía y municiones, y de si hay o no enemigos presentes en el static viewport (y si los hay, si se acercan o se alejan).

5. Conclusiones

En esta tesis se ha propuesto un modelo de visión sintética para el uso en los juegos de computadora, que cuenta con más información que los modelos puros utilizados en visión por computadora, pero restringiendo los datos que recibe el módulo de inteligencia artificial sólo a lo sentido (caso contrario al no uso de visión donde se tiene acceso a todo).

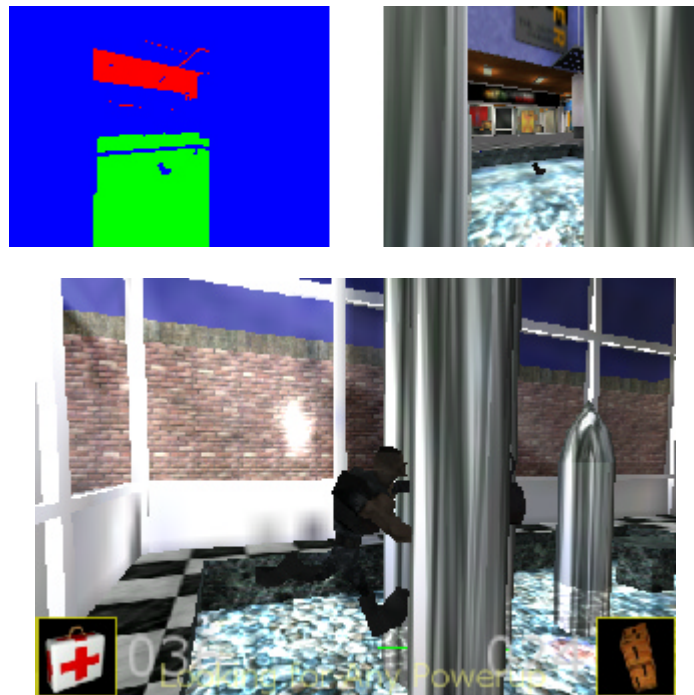


Figura 9. Imágenes de la implementación.

El modelo propuesto consta de dos viewports, estático y dinámico, donde con técnicas de falso coloreo es posible obtener perfecta identificación de objetos e información de profundidad (viewport estático junto al Z-buffer), e información discretizada de la velocidad que tienen los objetos observados en cada frame (viewport dinámico).

Se demuestra que el modelo de visión sintética podría formar parte de agentes o NPC's dentro de los juegos, por intermedio del desarrollo de un simple módulo de inteligencia artificial basado en reglas implementado sobre un NPC dentro de un ambiente que bien podría formar parte de un juego del tipo FPS o Third Person Action Adventure.

No nos hemos enfocado en el desarrollo de técnicas complejas para el módulo de IA, como podría ser memoria, aprendizaje e interacción, sino que la motivación fue sólo demostrar cómo podría ser usado el módulo de visión propuesto. Las heurísticas creadas para los distintos comportamientos que asume el agente, si bien son extremadamente simples, podrían servir como base para construir mejoras que permitan un comportamiento más realista.

El modelo de visión en conjunto con el módulo de IA fue implementado con éxito en el motor gráfico Fly3D [Watt01] [Watt02] (figura 9).

El objetivo deseado a largo plazo es poder contar con personajes completamente autónomos que habiten en los mundos virtuales dentro de los juegos de computadora, con sus propios deseos y necesidades, con su propia vida, con comportamientos realistas, abriendo las fronteras a nuevas posibilidades de géneros y de gameplay.

El sentido de la visión es el primer paso.

Referencias

- [Blum97a] 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.
- [Blum97b] Old Tricks, New Dogs: Ethology and Interactive Creatures. Bruce Mitchell Blumberg. February 1997. Ph.D Thesis. Massachusetts Institute of Technology.
- [Kuff99a] 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.
- [Kuff99b] James J. Kuffner. Autonomous Agents for Real-Time Animation. December 1999. Ph.D Thesis. Stanford University.
- [Nose95a] 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, N°1, 1995, pp. 7-19.
- [Nose95b] Synthetic Vision and Audition for Digital Actors. Hansrudi Noser; Daniel Thalmann. Proc. Eurographics '95, Maastricht.
- [Terz96] 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, pp. 313-320.
- [Thal96] D. Thalmann, H. Noser, Z. Huang. How to Create a Virtual Life?. Interactive Animation, chapter 11, pp. 263-291, Springer-Verlag, 1996.
- [TuTe94] X. Tu, D. Terzopoulos. Artificial Fishes: Physics, Locomotion, Perception, Behavior. Proc. of ACM SIGGRAPH'94, Orlando, FL, July, 1994, in ACM Computer Graphics Proceedings, 1994, pp. 43-50.
- [Watt01] A. Watt, F. Policarpo. 3D Computer Games Technology, Volume I: Real-Time Rendering and Software. Addison Wesley. 2001. ISBN 0-201-61921-0.
- [Watt02] A. Watt, F. Policarpo. 3D Games: Animation and Advanced Real-time Rendering, Addison Wesley. 2003. ISBN 0-201-78706-7.