Tesis de Licenciatura

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

Autor

Enrique, Sebastian
senrique@dc.uba.ar

Director

Watt, Alan
University of Sheffield
United Kingdom
a.watt@dcs.shef.ac.uk

Co-Director

Mejail, Marta
Universidad de Buenos Aires
Argentina
marta@dc.uba.ar

Departamento de Computación
Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Argentina

Septiembre 2002

Abstract

The role and utility of synthetic vision in computer games is discussed. 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 is presented. The utility of this synthetic vision module is demonstrated by using it as input to a simple rule-based AI module that controls agent behaviour in a first-person shooter game.

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.

A mis padres,
quienes sacrificaron todo por mi educación.

Tabla de Contenidos

Agradecimientos

Estaré por siempre agradecido con Alan Watt, quien aceptó guiarme incondicionalmente a través de todo el proyecto. Nunca olvidaré tampoco toda la ayuda que me brindó en mi pequeño viaje a Sheffield, teniendo en cuenta además el esfuerzo que realizó debido a la desafortunada condición en que se encontraba en esos momentos. Espero sinceramente haber cumplido con sus expectativas. Agradezco también a su esposa, Dionéa, quien fue muy amable conmigo en todo momento.

Quisiera agradecer a Manual Sánchez y James Edge, también de Sheffield, por ser tan amigables conmigo y por la buena relación que forjamos en el día a día. Un agradecimiento especial para Steve Maddock, también por la ayuda incondicional brindada y la muy buena charla de puesta en contexto que mantuvimos.

Agradezco especialmente a Fabio Policarpo, por muchas razones: me dio acceso al código fuente completo del motor Fly3D desde sus versiones iniciales y los sucesivos releases; su ayuda incondicional con cada parte del código cuando estaba perdido y no sabía qué hacer fue fundamental; y, además, me dio lugar en sus oficinas en Niterói para trabajar allí por un tiempo. Agradezco también a toda la gente de Paralelo –Gilliard Lopes, Marcos, etc.- y a los amigos de Fabio, todos ellos me aceptaron como uno más del grupo y me hicieron sentir muy bien en el breve tiempo que estuve allí. Puede sentirse la pasión por los juegos en cada rincón de Paralelo.

Estaré por siempre en deuda con Alan Cyment, Germán Batista y Javier Granada, quienes hicieron posible una versión en inglés de esta tesis.

Gracias a todos los profesores del Departamento de Computación que hacen la diferencia, como Gabriel Wainer, de quien no sólo aprendí ciencias de la computación, sino también una filosofía de trabajo.

No debo dejar de mencionar a todos mis compañeros y amigos de la universidad con quienes compartí tantos sufridos y divertidos días y noches. En especial a Ezequiel Glinsky, con quien compartimos cada paso de nuestras carreras.

Gracias Cecilia por haber estado a mi lado todos estos años.

Gracias también a Irene Loiseau, quien inició el contacto con Alan, y a todo el comité de la ECI 2001, quienes aceptaron mi sugerencia de invitar a Alan a la Argentina donde brindó un excelente, interesante y exitoso curso.

Y, finalmente, un agradecimiento especial para Claudio Delrieux, quien reforzó mi pasión por la computación gráfica, y siempre estuvo dispuesto a darme una mano desde antes, durante, y sin ninguna duda también después de este proyecto.

Introducción

Resumen

Hoy en día, usualmente los juegos de computadora en 3D utilizan inteligencia artificial (IA) para controlar los personajes manejados por la computadora (NPC's, Non-Player Characters) tomando información directamente de la base de datos interna del juego. En general, controlan el comportamiento, movimiento y accionar de los NPC's teniendo conocimiento de lo que sucede en el mundo entero, probablemente haciendo "trampa" si el desarrollador o el diseñador no ponen ninguna restricción en ello.

Como una gran cantidad de oponentes, o amigos, controlados por la computadora son como humanos, parecería ser lógico e interesante brindarles sentidos como los que el hombre tiene. Esto significa que esos personajes podrían tener sistemas completos o parciales de sensado visual, auditivo, táctil, olfativo, y degustativo. La información sensada por esos sistemas podría procesarse en un módulo de IA, el cerebro, para actuar, aprender, y reaccionar dependiendo de la personalidad, sentimientos y necesidades de cada personaje. El NPC se convierte entonces en un *synthetic character* que vive en un mundo virtual.

El campo de los synthetic characters o agentes autónomos se encarga de investigar el uso de los sentidos combinados con técnicas de IA con el objetivo de crear personajes con comportamientos realistas. Esas técnicas tienen la intención de forjar una personalidad, usando memoria cognoscitiva, sistemas basados en reglas, y demás para producir agentes que aparentan estar vivos e interactúan en su propio mundo, quizás también interactuando con humanos de la vida real.

Sin embargo, el esfuerzo dado para la investigación del uso de synthetic characters en juegos de computadora 3D en tiempo real ha sido escaso. En esta tesis se propone un sistema de visión para NPC's, una *visión sintética* o *synthetic vision*. Asimismo, se analiza cuán útil y factible podría resultar su uso en la industria de los juegos de computadora.

Creemos que el uso de la visión sintética junto con módulos complejos de IA podrían mejorar notoriamente el gameplay produciendo mejores y más realistas NPC's.

Debemos notar que nuestros esfuerzos se enfocaron en la investigación y propuesta de la visión sintética, y no en la IA que la usa. Por esta sencilla razón, sólo un simple módulo de IA que controla el comportamiento de un NPC usando el sistema de visión propuesto dentro un juego del tipo FPS ha sido desarrollado.

Análisis

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 *computer vision*. 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 computer vision 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 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 rederizar 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* o *sensory honesty* 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 query, 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 and Terzopoulous [TuTe94] la usaron para peces artificiales. El énfasis de su trabajo fue un modelo basado en fisica 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 300º. 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 computer vision 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 computer vision con el uso de información semántica en un viewport.

Por el contrario, un *approach* más simple es usar falso coloreo en el rendering para representar información semántica. Blumberg [Blum97] utiliza ese approach 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* [Nose95] usan falso coloreo para representar identificación de objetos, además de introducir un octree dinámico para representar la memoria visual del agente. Kuffner y Latombe [Kuff99] 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 prever 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.

Trabajo Previo

Más allá de los trabajos de investigación mencionados en la sección anterior, creemos necesario comentar brevemente cada uno de los actores principales y trabajos relevantes en este campo.

Podemos pensar en dos extremos entre los que se encuentra la visión sintética:

- *Visión sintética pura*, también llamada computer vision o *artificial vision* en [Nose95b], que, citando la misma referencia, es "el proceso de reconocimiento de la imagen del ambiente real capturado por una camera (…) es un tópico importante de investigación en robótica e inteligencia artificial."

- *Sin visión*, esto es, no usar ningún sistema de visión para los personajes.

Imaginando una línea recta, con visión sintética pura en un extremo y sin visión en el otro, cada uno de los approaches anteriores caen en algún lado dentro de esta línea.

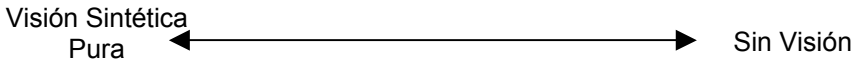

Figura 1. Clasificación gráfica de los approaches.

Bruce Blumberg describe en [Blum97a] una visión sintética basada en energía de movimiento que utiliza para que un agente autónomo evite obstáculos y navegue a bajo nivel. La escena es renderizada con falso coloreo, tomando la información de los píxeles a partir de una formula con pesos que es una combinación de flujo (colores de los píxeles de los frames anteriores) y masa (basada en texturas), dividiendo la imagen en mitades, y tomando diferencias con el objetivo de conducir al agente. Información detallada puede encontrarse en [Blum97b]. Investigación relacionada a synthetic characters puede encontrarse en [Blum01].

James Kuffner presenta en [Kuff99a] un approach con falso coloreo que utiliza para navegación de actores digitales, con detección de colisiones, implementando memoria visual. Información detallada puede encontrarse en [Kuff99b]. Para una rápida introducción a sus trabajos puede navegar por [Kuff01].

Hansrudi Noser *et al* usan en [Nose95a] visión sintética para navegación de un actor digital. La visión es la única conexión entre el ambiente y el actor, y la utilizan para lidiar con los problemas de evasión de obstáculos y representación del conocimiento, aprendizaje y olvido. Para ello implementan una memoria por medio de voxels de la que se valen para resolver, además, las búsquedas de caminos. Su representación de visión hace dificultosa la rápida identificación de los objetos presentes, una de las premisas del modelo que nosotros presentamos, aunque sería interesante integrar sus propuestas de memoria visual a lo hecho en esta tesis. En [Nose98] utilizan las ideas anteriores, más sensores auditivo y táctil, para implementar una simple simulación de juego de tenis.

Sin dudas los trabajos de Kuffner y Noser fueron los que más influenciaron en el desarrollo de esta tesis.

Olivier Renault *et al* [Rena90] utilizan una visión sintética de 30x30 píxeles para animación, donde se valen del front buffer para un rendering normal, del back buffer para identificación de objetos, y del *z*-buffer para las distancias. Proponen comportamientos de alto nivel para atravesar un corredor evitando obstáculos.

Rabie y Terzopoulos desarrollan en [Rabi01] un sistema de visión estereoscópica para navegación y evasión de obstáculos en peces artificiales. Con la misma base trabajan Tu y Terzopoulos en [Tute94] donde desarrollan más profundamente el comportamiento de los peces.

Craig W. Reynolds explica en [Reyn01] una serie de comportamientos para el movimiento de agentes autónomos, como son 'pursuit', 'evasion', 'wall following', 'unaligned collision avoidance', entre otros. Este es un muy interesante trabajo que debería servir como guía para el desarrollo de módulos complejos de IA para NPC's en juegos de computadora.

Damián Isla discute en [Isla02] el potencial de la IA utilizando synthetic characters.

John Laird discute en [Lair00a] inteligencia artificial al nivel de un humano en juegos de computadora. En [Lair01] propone objetivos de diseño para synthetic characters autónomos. En general todos los trabajos están orientados en mejorar notablemente la inteligencia artificial de los oponentes del jugador en los juegos de computadora, un ejemplo de ello es la implementación del Quakebot [Lair00b]. Más trabajos de investigación de Laird puede encontrarse en [Lair02].

El resto de los trabajos se basan aproximadamente en las ideas de la investigación señalada anteriormente.

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 *robot vision* 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 contrar un modelo que se ubique entre medio de ambos extremos, y que sea útil para su uso en juegos de computadora.

Figura 2. Lugar deseado para nuestro modelo.

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.

En la figura 3 se muestra un ejemplo del viewport renderizado en forma normal y el correspondiente viewport estático.

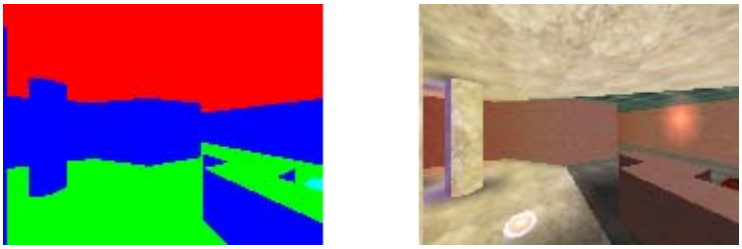

Figura 3. Static viewport (izquierda) obtenido a partir del viewport rederizado en forma normal (derecha).

Static Viewport

El static viewport es principalmente útil para identificación de objetos, tomando la forma de un viewport con falso coloreo similar al descrito en [Kuff99].

Definiciones

Definimos:

Objeto. Un ítem con forma 3D y masa.

Clase de Objetos. 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. Es decir, existe una función *c* que es un mapeo de clases de objetos a colores. Esa función es inyectiva ya que dos clases de objetos cualesquiera nunca tienen el mismo color asociado.

Entonces, dado

$$CO : \text{Conjunto de clases de objetos.}$$

y

$$C : [0, 1] \times [0, 1] \times [0, 1]$$

Conjunto de vectores 3D con componentes reales entre 0 y 1 en cada una de ellas. Cada vector representa un color en el modelo RGB: la primer componente corresponde a la cantidad de color rojo, la segunda a la de verde, y la tercera a la de azul.

$$c : CO \rightarrow C$$

Es la función de mapeo entre clases de objetos y colores.

$$\forall\ co_1, co_2 \in CO : c(co_1) = c(co_2) \Rightarrow co_1 = co_2$$

La función es inyectiva.

Como *CO* es un conjunto finito, y *C* es infinito, no todos los colores posibles son usados, entonces la función no es suryectiva, por lo tanto tampoco es biyectiva.

Sin embargo, 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.

Clase de Objetos	*Color*
Power-Ups de Energía	(1, 1, 0)
Power-Ups de Municiones	(0, 1, 1)
Enemigos	(1, 0, 1)

Tabla 1. Tabla de mapeo entre clases de objetos y colores.

Level Geometry

Además de los objetos definidos en la sección anterior, los juegos de computadora 3D diferencian los objetos normales o ítems, y la geometría del nivel o *level geometry*. Se define:

Ítem. Todo objeto, como fue definido en la sección previa, que no es parte de la level geometry.

Level Geometry. Todos los polígonos que forman parte de la estructura de cada nivel. Por ejemplo: el piso, paredes, etc.

Sin embargo, la level geometry como fue definida podría no ser estrictamente un objeto, ya que usualmente es una construcción formada por una cáscara de polígonos que no necesariamente forman un objeto sólido con masa.

A pesar de ello, dividiremos la level geometry en tres clases, que serán parte del conjunto *CO* de clases 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 adelante (figura 4).

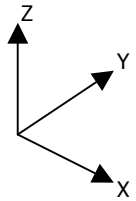

Figura 4. Sistema de coordenadas utilizado.

En la Tabla 2 se muestra una tabla de mapeo extendida con las clases de objetos de la level geometry.

Clase de Objetos	*Color*
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 2. Table de mapeo entre clases de objetos y colores, extendida con las clases de la level geometry.

Profundidad

Aparte de 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:

Camera Angle. (Ángulo de la cámara) Es el ángulo de visión con el cual es renderizada la escena desde el punto de vista del personaje.

Near Plane (NP). El plano más cercano del view frustum volume [Watt01].

Far Plane (FP). El plano más lejano del view frustum volume [Watt01].

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 rederizado, 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.

Sea $d_{x,y}$ = DepthBuffer[x,y] el valor de profundidad del píxel con coordenadas x, y.

$$d_{x,y} \in [0, 1]$$

En coordenadas del mundo, la distancia perpendicular entre el personaje y el objeto al cual pertenece el píxel se calcula como:

$$P = (FP - NP) * d_{x,y} + NP$$

Dynamic Viewport

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. El viewport dinámico tiene el mismo tamaño que el estático.

$$R, G, B \in [0, 1]$$

Los colores rojo, verde, y azul de cada píxel son números reales entre 0 y 1.

Y definimos,

$$V_{max} \in \Re^+$$

Como la velocidad máxima permitida en el sistema, una constante real positiva.

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

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

La componente de color roja es el mínimo valor entre 1, y la magnitud de la velocidad del objeto ubicado en coordenadas (x, y) del viewport estático sobre la velocidad máxima permitida.

Si D es el vector de dirección/visión del agente, podemos normalizar D y el vector de velocidad $V_{x,y}$:

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

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

Y luego obtener,

$$c = V_{x,y}^N \cdot D^N = V_{x,y}^N{}_1 D^N{}_1 + V_{x,y}^N{}_2 D^N{}_2 + V_{x,y}^N{}_3 D^N{}_3$$

$$c \in [-1, 1]$$

c es el producto interno o el coseno del ángulo formando entre el vector de velocidad $V_{x,y}$ normalizado, correspondiente al objeto ubicado en coordenadas (x, y) del static viewport, y D normalizado, el vector de visión / dirección del NPC. El ángulo entre estos vectores varía entre 0 y 180º.

c es un número real entre –1 y 1.

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

La componente de color verde es un mapeo del coseno c al intervalo [0, 1]. Un valor del coseno de cero producirá una componente de color igual a 0.5.

$$s = \sqrt{(1 - c^2)}$$

$$s \in [0, 1]$$

s es el seno del ángulo formado entre el vector de velocidad $V_{x,y}$ normalizado, correspondiente al objeto ubicado en coordenadas (x,y) del static viewport, y D normalizado, el vector de visión / dirección del NPC. s es calculado utilizando el coseno del mismo ángulo –recordemos que varía entre 0 y 180º-.

s es un número real entre 0 y 1.

$$B(x,y) = s$$

La componente de color azul es un mapeo directo entre del seno s al intervalo [0, 1]. Un valor de coseno de cero producirá una componente de color verde de 0.5, y una componente azul de 1.

Con esas definiciones, un objeto completamente inmóvil tendrá color (0.0, 0.5, 1.0). Objetos en movimiento tendrán distintos colores dependiendo de la orientación y magnitud de su velocidad.

Buffers

La información estática, dinámica, y de profundidad puede mantenerse en buffers en memoria para su uso. Cada elemento de los viewports estático y dinámico contiene tres valores correspondientes a cada componente de color. Cada elemento del depth buffer contiene sólo un valor. El tamaño de los buffers es fijo en ancho por altura de los viewports, siendo por lo tanto cada uno de ellos una matriz bidimensional. Entonces, dada una coordenada (x,y) del viewport, es posible obtener de los buffers la identificación del objeto, su profundidad, y su información de movimiento.

Comentarios Adicionales

Decimos que nuestro sistema de visión es perfecto porque las condiciones de iluminación no son tenidas en cuenta. Un objeto en una zona poco iluminada será perfectamente reconocible como si estuviera en una completamente iluminada.

Lo mismo sucede con la información dinámica, donde más allá de la discretización de los valores, son exactos. Para hacerlo más real (los humanos en general podrían hacer buenas aproximaciones pero nunca estimar la velocidad y orientación en forma precisa), sería posible agregar ruido en cada valor provisto.

Figura 5. De izquierda a derecha, static viewport, viewport normal, y dynamic viewport.

El viewport dinámico esperado es mostrado junto con los viewports estático y normal en la figura 5.

Refiérase al Apéndice B para obtener detalles de la implementación de nuestro modelo de visión sintética sobre Fly3D [Fly01; Watt01; Watt02].

El Cerebro: Módulo de IA

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.

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 FPS.

El módulo es tan simple que realmente no podría ser utilizado en un juego real sin agregarle nuevas características, comportamientos y, por sobre todas las cosas, un refinamiento de los movimientos implementados. Nuestra intención fundamental fue mostrar de qué manera podría aprovecharse el uso de la visión, sin entrar en profundidad en el desarrollo de inteligencia artificial para el uso de la misma.

Definición del FPS

Las características fundamentales de los juegos FPS son las siguientes:

- El desarrollo del juego se lleva a cabo a través de una serie de regiones modeladas en 3D, que pueden ser tanto ambientes de exteriores como de interiores, o combinación de ambos, denominadas niveles.

- El jugador debe cumplir distintos objetivos en cada nivel para poder alcanzar el siguiente. Es decir, cada nivel consta de una meta general: "avanzar al siguiente nivel", que está constituida por un conjunto de metas específicas: "conseguir llave para atravesar puerta", "matar al boss del nivel", etc. También es muy común encontrar submetas que no son necesarias para lograr la meta general, por ejemplo: "si es posible, obtener 100 unidades de oro".

- El jugador puede ir adquiriendo distinto armamento durante su avance en el juego. En general contará con una diversidad de 10 armas, la mayoría de ellas con munición finita.

- El jugador consta con un nivel de vida o energía que, cuando llega a 0, muere.

- El jugador deberá enfrentar constantemente enemigos, NPCs, que intentarán atacarlo y matarlo según las características de cada uno de ellos, con armas o lucha cuerpo a cuerpo.

- Es posible que también existan NPCs que colaboren con el jugador.

- El jugador podrá recoger ítems denominados power-ups, que corresponden a aumentos de energía, munición para las armas, o poderes especiales por tiempo limitado.

- El jugador observa al mundo como si fuera a través de los ojos del personaje que está encarnando.

Se recomienda que realice una recorrida por internet para familiarizarse más con este tipo de juegos, si es que aún no lo está. Wolfenstein 3D [Wolf01] fue uno de los pioneros, mientras que las series de Unreal [Unre01] y Quake [Quak01] son más que demostrativos.

A continuación describiremos las características principales del FPS para la implementación de nuestro modelo de visión sintética.

NPC Principal

Nuestro FPS estará habitado por el personaje principal al que le daremos vida autónoma, 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.

Siempre se cumple que

$$H \in \aleph_0,\ 0 \leq H \leq 100$$

la energía de Bronto varía entre 0 y 100, números naturales. Inicialmente,

$$H_{ini} = 100$$

Y la primer regla con la que se cuenta es que:

$$H = 0 \Rightarrow \text{Bronto muere}$$

Con las municiones sucede algo similar, ya que como invariante durante el juego se cumple que

$$W \in \aleph_0,\ 0 \leq W \leq 100$$

la munición de Bronto varía entre 0 y 100, números naturales. Inicialmente,

$$W_{ini} = 100$$

Pero en este caso si W = 0 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. Es decir, dados:

$t \in \Re^+_0$ como el tiempo.
$t_{w0} \in \Re^+_0$ como el tiempo a partir del cual comenzaron a disminuir las municiones.
$W_0 \in \aleph$ como las municiones al momento t_{w0}, inicialmente $W_0 = W_{ini}$.
$t_{dw} \in \aleph$ el intervalo de tiempo en segundos de disminución de municiones.

$D_w \in \aleph$ la constante de disminución de municiones.

Se define que,

$$t = t_{w0} \Rightarrow W(t) = W_0$$

Cuando se está en el tiempo inicial de conteo t_{w0}, W asume un valor actual W_0.

$$k \in \aleph_0,\ t_{w0} + k\ t_{dw} \leq t < t_{w0} + (k+1)\ t_{dw} \Rightarrow W(t) = máx(W(t_{w0} + k\ t_{dw}) - D_w,\ 0)$$

Cuando se está en un tiempo superior al t_{w0}, W disminuye su valor de acuerdo a intervalos de t_{dw} linealmente, hasta llegar a cero y continuar en ese valor.

Es análogo para el caso de la energía:

$t_{h0} \in \Re^+_0$ como el tiempo a partir del cual comenzó a disminuir la energía.
$H_0 \in \aleph$ como la energía al momento t_{h0}, inicialmente $H_0 = H_{ini}$.
$t_{dh} \in \aleph$ el intervalo de tiempo en segundos de disminución de energía.
$D_h \in \aleph$ la constante de disminución de energía.

$$t = t_{h0} \Rightarrow H(t) = H_0$$

$$k \in \aleph_0,\ t_{h0} + k\ t_{dh} \leq t < t_{h0} + (k+1)\ t_{dh} \Rightarrow H(t) = máx(H(t_{h0} + k\ t_{dh}) - D_h,\ 0)$$

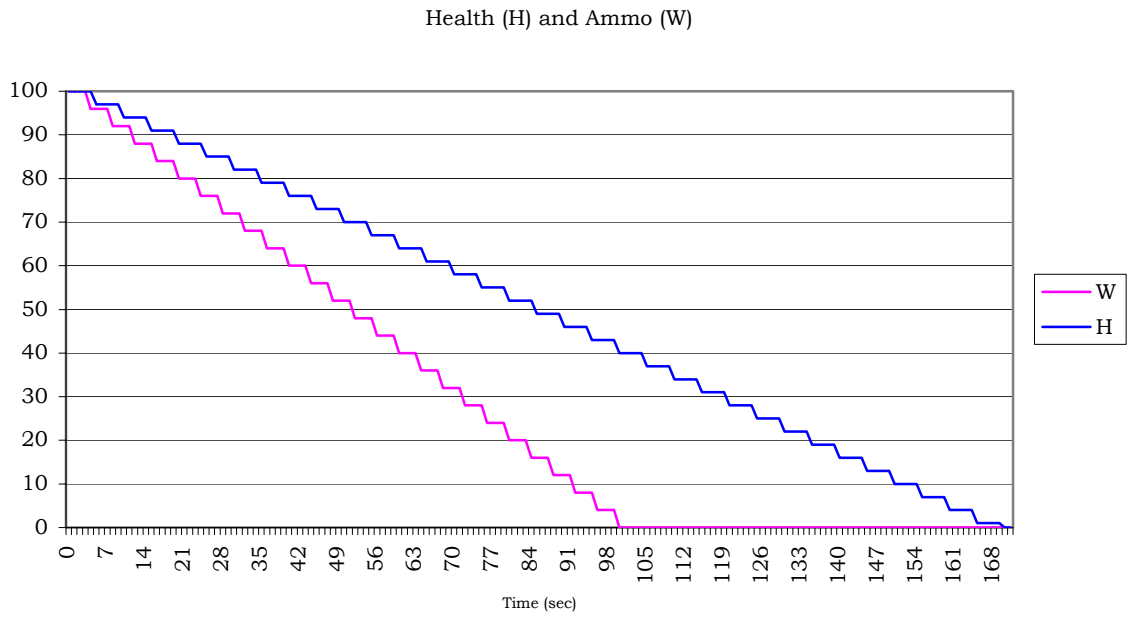

Figura 6. Un ejemplo del sistema de disminución de munición y energía de Bronto, a partir del inicio del juego. Ambos disminuyen gradualmente. Las municiones alcanzan el nivel de 0 a los 100 segundos, mientras que la energía a los 170 segundos, en el momento en que Bronto muere.

Como ejemplo, tomemos:

$t_0 = 0$, inicio del juego.

$H_0 = H_{ini} = 100$.
$t_{dh} = 5$ segundos.
$D_h = 3$.
$W_0 = W_{ini} = 100$.
$t_{dw} = 4$ segundos.
$D_w = 4$.

Se produce la situación descripta en la figura 6, donde el nivel de munición de Bronto va disminuyendo gradualmente hasta alcanzar 0 a los 100 segundos; mientras que la energía recién alcanza 0 a los 170 segundos, que es en el momento que Bronto muere.

Power-Ups

Ya se ha definido la cantidad inicial de energía y municiones, también la forma en que disminuyen, y se ha especificado que si la energía llega a cero, Bronto muere. Falta definir alguna forma de que Bronto aumente ambas propiedades: será mediante 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:

$$H_{after} = H_{before} + A_h$$

$$W_{after} = W_{before} + A_w$$

Siendo,

$A_h \in \aleph$, el valor de incremento de la propiedad Energía del NPC al tomar un power-up de la misma.
$A_w \in \aleph$, el valor de incremento de la propiedad Munición del NPC al tomar un power-up de la misma.

Además de incrementar el valor de la propiedad correspondiente, el tomar un power-up establecerá un nuevo valor de t_{w0} o t_{h0}, según corresponda, que será el tiempo actual; y, además, establecerá $W_0 = W_{after}$ o $H_0 = H_{after}$, según corresponda.

Esto significa que tomar un power-up hace reiniciar el ciclo por el cual se le disminuyen los valores a Bronto de esa propiedad.

Tomemos como ejemplo el visto en la sección anterior, donde:

$t_0 = 0$, inicio del juego.
$H_0 = H_{ini} = 100$.
$t_{dh} = 5$ segundos.
$D_h = 3$.
$W_0 = W_{ini} = 100$.
$t_{dw} = 4$ segundos.
$D_w = 4$.

Agreguemos ahora:

A_w = 10.
A_h = 5.

Y supongamos que Bronto obtiene:

- Un power-up de municiones a los 37 segundos; donde su valor de munición aumentará de 64 a 74, y reestablece el período de tiempo t_{w0} a 37, a partir del cual se le debe descontar municiones cada t_{dh} segundos.
- Otro power-up de municiones a los 43 segundos, donde pasa de 70 a 80 y vuelve a reestablecer el tiempo t_{w0} pero esta vez a 43.
- Un power-up de energía a los 156 segundos, donde su valor de energía aumentará de 7 a 12, y reestablece el período de tiempo t_{h0} a 156.

Las municiones de Bronto recién llegarán a cero a los 123 segundos; y morirá a los 176 segundos, cuando la energía llegue a cero. Observe la situación en la gráfica de la figura 7.

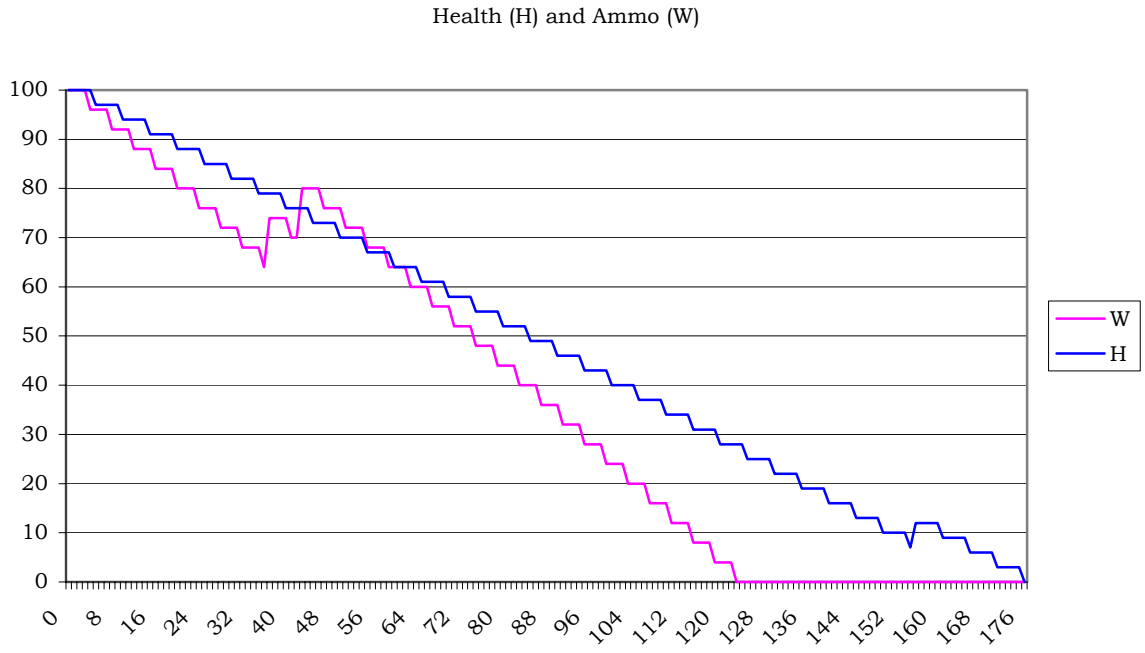

Figura 7. El ejemplo de la figura 6 pero Bronto obteniendo dos power-ups de municiones, a los 37 y 43 segundos, y uno de energía, a los 156 segundos. Las municiones ahora alcanzan el nivel de 0 recién a los 123 segundos, mientras que la energía a los 176 segundos, en el momento en que Bronto muere.

En la figura 8 se describe el proceso completo mediante un diagrama de estados. El juego comienza (evento Start Game) y se establecen los valores iniciales. Durante el desarrollo del juego (Game Running) pueden surgir varios eventos: aquellos que hacen disminuir la cantidad de municiones o energía que posee Bronto, al llegar a un valor de tiempo determinado; y aquellos que hacen aumentar la cantidad de municiones o energía, que es cuando se toma un power-up. El evento de alcanzar el valor de energía cero hace que Bronto muera.

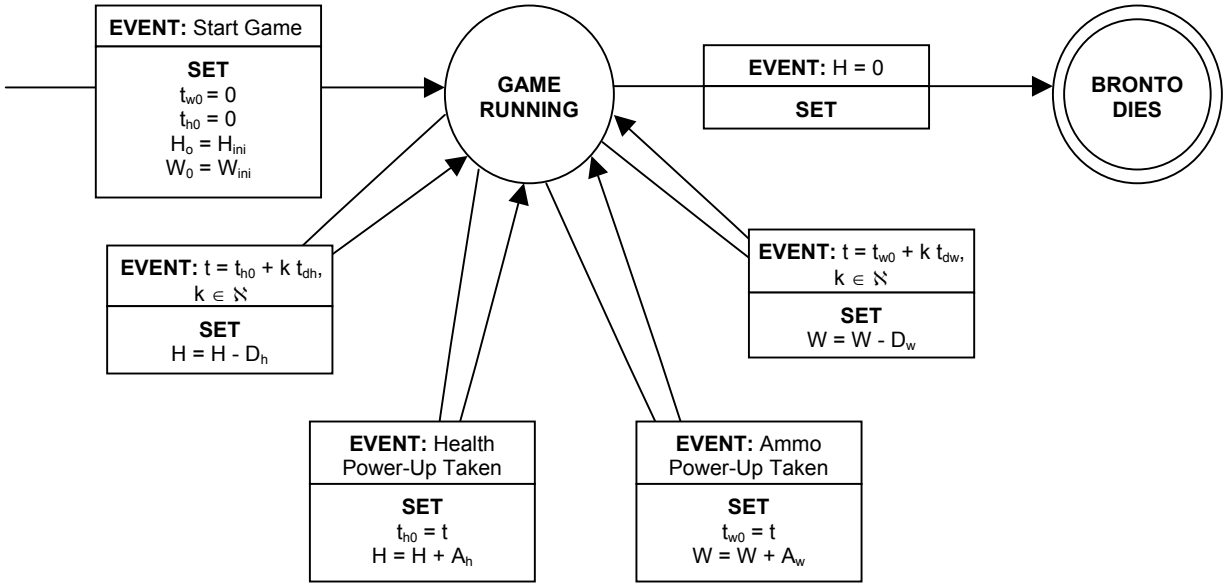

Figura 8. Los eventos que afectan y son afectados por el valor de las propiedades de municiones y energía que posee Bronto durante el desarrollo del juego.

Comportamiento de Bronto

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

Sean:

$H_{ut} \in \aleph$, el umbral superior de energía
$H_{lt} \in \aleph$, el umbral inferior de energía
$W_{ut} \in \aleph$, el umbral superior de municiones
$W_{lt} \in \aleph$, el umbral inferior de municiones
H y W los valores de las propiedades de energía y municiones, respectivamente, como fueron definidas en las secciones anteriores.

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:

1. 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.

2. 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.
3. Looking for Weapon (LW), cuando $H_{ut} \leq H$ y $W_{lt} \leq W < W_{ut}$. Cuando Bronto está estable de energía, por encima de H_{ut}, y empieza a sentir necesidad de municiones, entre W_{lt} y W_{ut}, el objetivo de Bronto será recolectar power-ups de municiones.
4. Looking for Any Power-Up (LHW), cuando $H_{lt} \leq H < H_{ut}$ y $W_{lt} \leq W < W_{ut}$. Cuando Bronto siente necesidad tanto de energía como de municiones, su objetivo será recolectar cualquier power-up.
5. Looking Quickly for Weapon (LQW), cuando $H_{lt} \leq H$ y $W \leq W_{lt}$. Cuando Bronto siente una necesidad extrema de municiones, porque su cantidad cayó del umbral inferior W_{lt}, su objetivo será recolectar lo más rápido posible power-ups de municiones.
6. Looking Quickly for Health (LQH), cuando $H < H_{lt}$. Cuando Bronto siente una necesidad extrema de energía, porque su cantidad cayó del umbral inferior H_{lt}, 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.

En la figura 9 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 $H < H_{lt}$, que con las condiciones planteadas para nuestro FPS junto con valores balanceados de H_{ut} y H_{lt} no debería producirse.

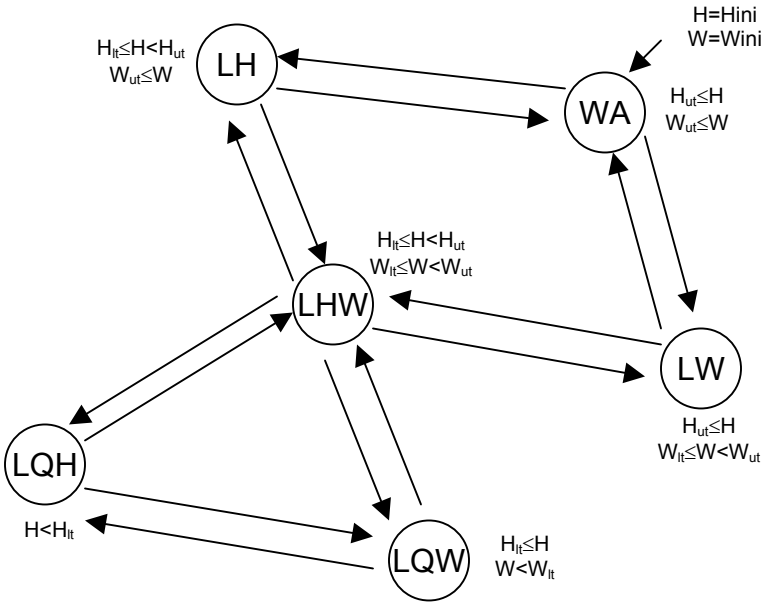

Figura 9. Diagrama de estados del comportamiento de Bronto. Sólo los cambios de estados esperados están representados con flechas, a pesar de que es posible el cambio de cualquier estado a otro.

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.

Cálculo del Destino

Una vez elegido un punto de destino en coordenadas (x, y) del viewport, se desproyecta el mismo para obtener sus coordenadas dentro del mundo. Para ello, además de las coordenadas elegidas, se necesitan la matriz de modelado, M, y la de proyección, P, actuales, que están presentes en todo motor 3D, el ángulo de la cámara utilizado para el render del viewport, y el viewport V en sí.

El proceso consiste en aplicar las matrices en orden inverso al proceso de proyección descrito en [Eber01].

El cálculo es:

$$T = (P.M)^{-1}.\begin{pmatrix} \frac{2(x - V[0])}{V[2]} - 1 \\ \frac{2(y - V[1])}{V[3]} - 1 \\ 1 \\ 1 \end{pmatrix}$$

Las tres primeras coordenadas de T corresponden a las coordenadas (x, y, z) del mundo del punto elegido con coordenadas (x, y) en el viewport.

Si se cuenta con un motor basado en OpenGL [Open01], como es el caso de Fly3D [Fly01] la operación se reduce a utilizar el comando de la extensión glu llamado gluUnProject.

Generación de la Curva de Bezier

La curva de Bezier se genera en el plano paralelo al XY sobre el cual Bronto está parado, por lo tanto se trata de una curva en 2D. No es necesario construir una curva 3D en este caso, porque Bronto posee sólo 4 grados de libertad. Una curva 3D debería trazarse en el caso de objetos que posean 6 grados de libertad, por ejemplo, una nave espacial.

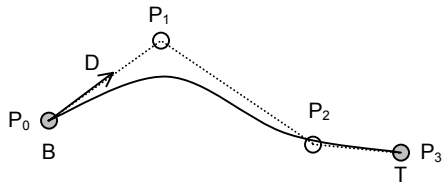

Figura 10. Los puntos de control de la curva de Bezier, junto con B, la posición inicial, T, la posición final, y D, la dirección de visión actual de Bronto.

Para cada curva, se necesitan 4 puntos de control. El primer y el último punto de control corresponden al punto de inicio y final de la curva, los otros dos puntos no tocan la curva. Refiérase a [Watt01] para más información sobre curvas de Bezier.

Si descartamos en todos los casos la coordenada Z, sean:

B : las coordenadas del mundo de la posición actual de Bronto.
T : las coordenadas del mundo de destino.
D : el vector de dirección/visión de Bronto.

PathDistance = || T – B ||, la distancia en línea recta entre la posición actual y la de destino.

PathDirection = (T – B) / PathDistance, el vector entre el destino y la posición origen, normalizado.

Los puntos de control se establecen como:

$P_0 = B$
$P_1 = B + D * (PathDistance / 3.0)$
$P_2 = T – PathDirection * (PathDistance / 3.0)$
$P_3 = T$

Con los puntos de control establecidos de este modo, la tangente inicial de la curva se corresponde con la dirección de visión inicial de Bronto, y la de destino con el vector entre el punto de destino y de llegada.

En la figura 10 se muestran las variables y puntos de control de una curva.

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:

$$C_f \in \Re,\ 0.0 < C_f \leq 100.0$$

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:

$$B_{bbr} \in \aleph, 0 < B_{bbr} < 80$$

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:

$B_{waupd} \in \aleph$, $0 < B_{waupd} < 80$, margen lateral y superior, en píxeles, del viewport de visión sintética; y
$B_{walpd} \in \aleph$, $0 < B_{walpd} < 80$, margen inferior, en píxeles, del viewport de visión sintética.

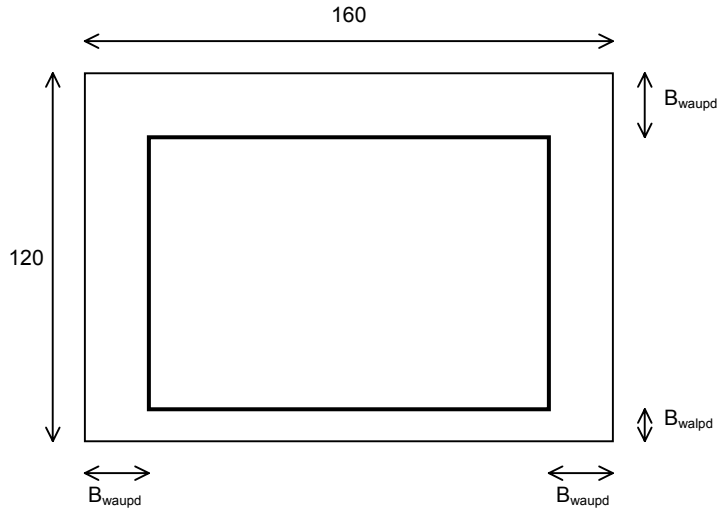

Figura 11. El viewport estático y los márgenes utilizados en Walk Around. Los píxeles fuera del recuadro más grueso no podrán ser elegidos como puntos de destino.

En la figura 11 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 primer fila, fila 0, corresponde a la línea inferior del viewport. La información del viewport será accedida como:

vps[i][j], i, $j \in \aleph_0$, donde $0 \leq i < 160$ y $0 \leq j < 120$, siendo i la columna y j la fila.

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 12.

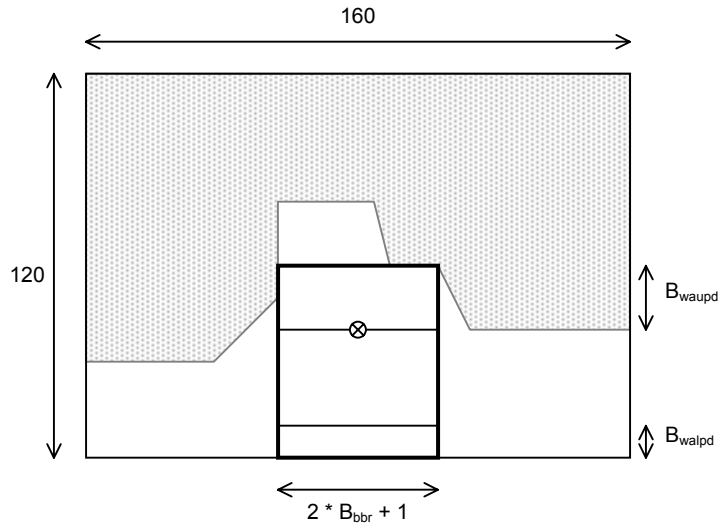

Figura 12. Un camino libre en el viewport estático. La zona en blanco corresponde a piso, la grisada a pared. El rectángulo de línea gruesa marca el camino libre elegido, dentro de él pueden apreciarse las líneas de los márgenes superior e inferior, y el punto que la heurística seleccionará como el nuevo destino.

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 180º hacia la izquierda o la derecha.

Algoritmo Walk Around

En esta sección presentaremos el pseudocódigo de Walk Around junto con la estrategia de búsqueda de caminos libres. En cada paso se explican detalles o se muestran ejemplos según se considere necesario.

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 12.
freeway = $\exists\ x_0, y_G \in \aleph_0, 0 \leq x_o < x_1 < 160 \wedge (x_1 - x_0) = (2 * B_{bbr}) \wedge 0 \leq y_G < (120 - b_{waupd} - b_{walpd}) \wedge (\forall\ i, j \in \aleph_0, 0 \leq i < (2 * B_{bbr} + 1) \wedge 0 \leq j < (y_G + b_{waupd} + b_{walpd}), vps[x_0 + i, j] =$ '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

Si (freeway = true) entonces

// Estrategia 1: Elijo, si existe, el camino libre central. En la figura 12 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 13. La estrategia falla cuando luego de darse la precondición, no encuentra ningún camino libre en la parte izquierda del viewport. Esto último se ejemplifica en la figura 14.
Si ($\exists\ i, j, k \in \aleph_0, 0 < i < j \leq B_{bbr} \wedge 0 \leq k < (b_{waupd} + b_{walpd}) \wedge (\forall\ w, s \in \aleph_0, (79 - B_{bbr}) \leq w < (79 + B_{bbr}) \wedge 0 \leq s < k \wedge vps[w, s] =$ 'piso') $\wedge (\forall\ t, u \in \aleph_0, (79 - B_{bbr}) \leq t \leq (79 - B_{bbr} + i) \wedge vps[t, k] =$ 'piso' $\wedge (79 + B_{bbr} - j) < u \leq (79 + B_{bbr}) \wedge vps[u, k] =$ 'piso') $\wedge vps[79 + B_{bbr} - j, k] \neq$ '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 precondición, no encuentra ningún camino libre en la parte derecha del viewport.
Si ($\exists\ i, j, k \in \aleph_0, 0 < j < i \leq B_{bbr} \wedge 0 \leq k < (b_{waupd} + b_{walpd}) \wedge (\forall\ w, s \in \aleph_0, (79 - B_{bbr}) \leq w < (79 + B_{bbr}) \wedge 0 \leq s < k \wedge vps[w, s] =$ 'piso') $\wedge (\forall\ t, u \in \aleph_0, (79 - B_{bbr}) \leq t < (79 - B_{bbr} + i) \wedge vps[t, k] =$ 'piso' $\wedge (79 + B_{bbr} - j) \leq u \leq (79 + B_{bbr}) \wedge vps[u, k] =$ 'piso') $\wedge vps[79 - B_{bbr} + i, k] \neq$ 'piso') entonces Elegir el mínimo x_0, $x_0 > 79$, tal que hace verdadero a freeway.

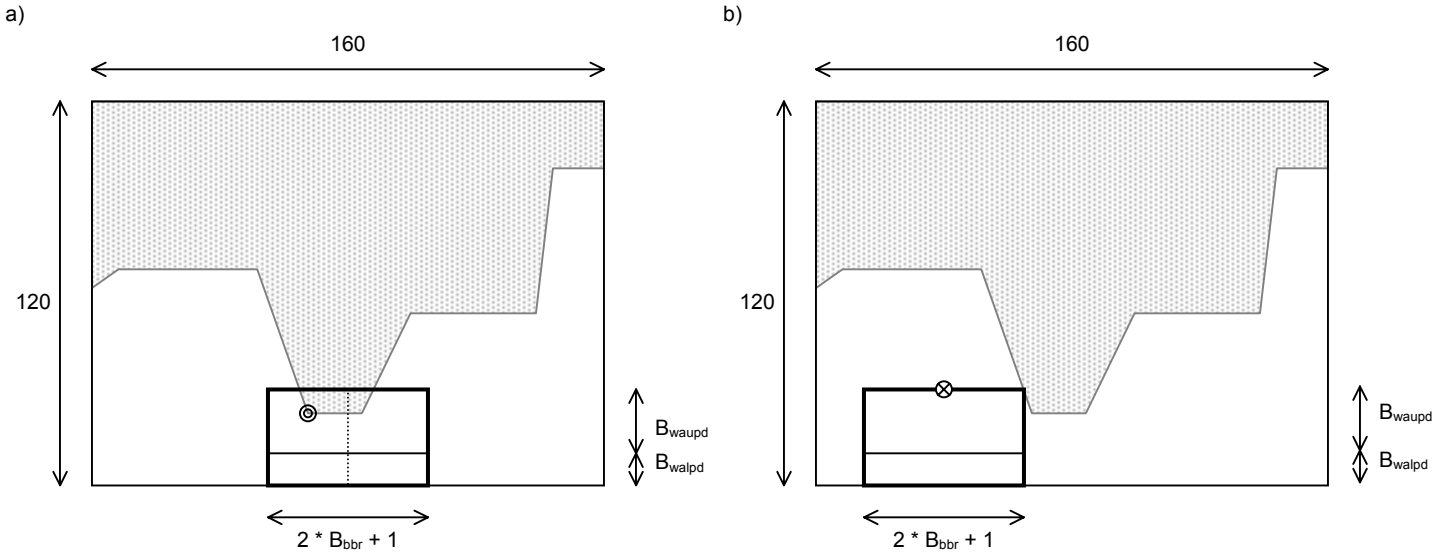

Figura 13. La estrategia 2 en acción. En **a)** puede observarse cómo se da la precondición para que se emplee esta estrategia: no existe un camino central libre y la línea más cercana a la parte inferior del viewport dentro del recuadro de búsqueda que contiene un píxel distinto de piso, contiene un 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.

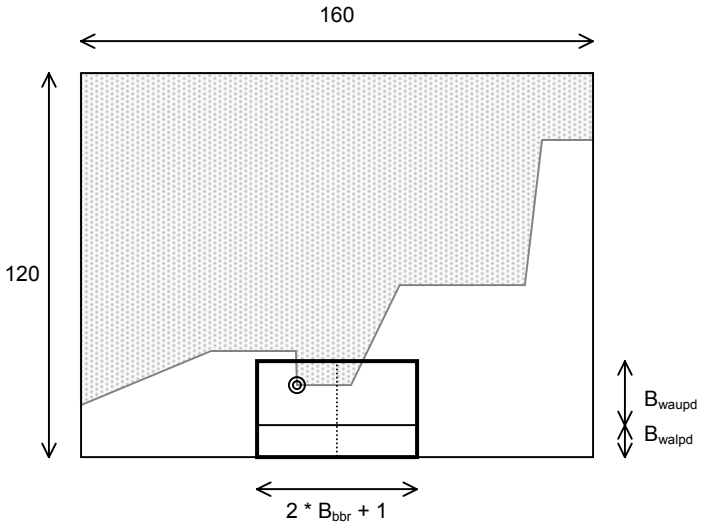

Figura 14. La estrategia 2 cuando falla. Puede observarse cómo se da la precondición para que se emplee esta estrategia. Sin embargo, al buscar un camino libre hacia la izquierda del punto marcado con un doble círculo, no encontrará ninguno.

// 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. Vea los ejemplos en la figura 15.

Si ($\exists$ i, k $\in \aleph_0$, 0 < i $\leq B_{bbr}$ ^ 0 $\leq$ k < (b_{waupd} + b_{walpd}) ^ ($\forall$ w, s $\in \aleph_0$, (79 – B_{bbr}) $\leq$ w < (79 + B_{bbr}) ^ 0 $\leq$ s < k ^ vps[w, s] = 'piso') ^ ($\forall$ t, u $\in \aleph_0$, (79 – B_{bbr}) $\leq$ t < (79 – B_{bbr} + i) ^

vps[t, k] = 'piso' ^ (79 + B_{bbr} - i) < u ≤ (79 + B_{bbr}) ^ vps[u, k] = 'piso') ^ vps[79 + B_{bbr} - i, k] ≠ 'piso') ^ vps[79 - B_{bbr} + i, k] ≠ 'piso') entonces fail = true

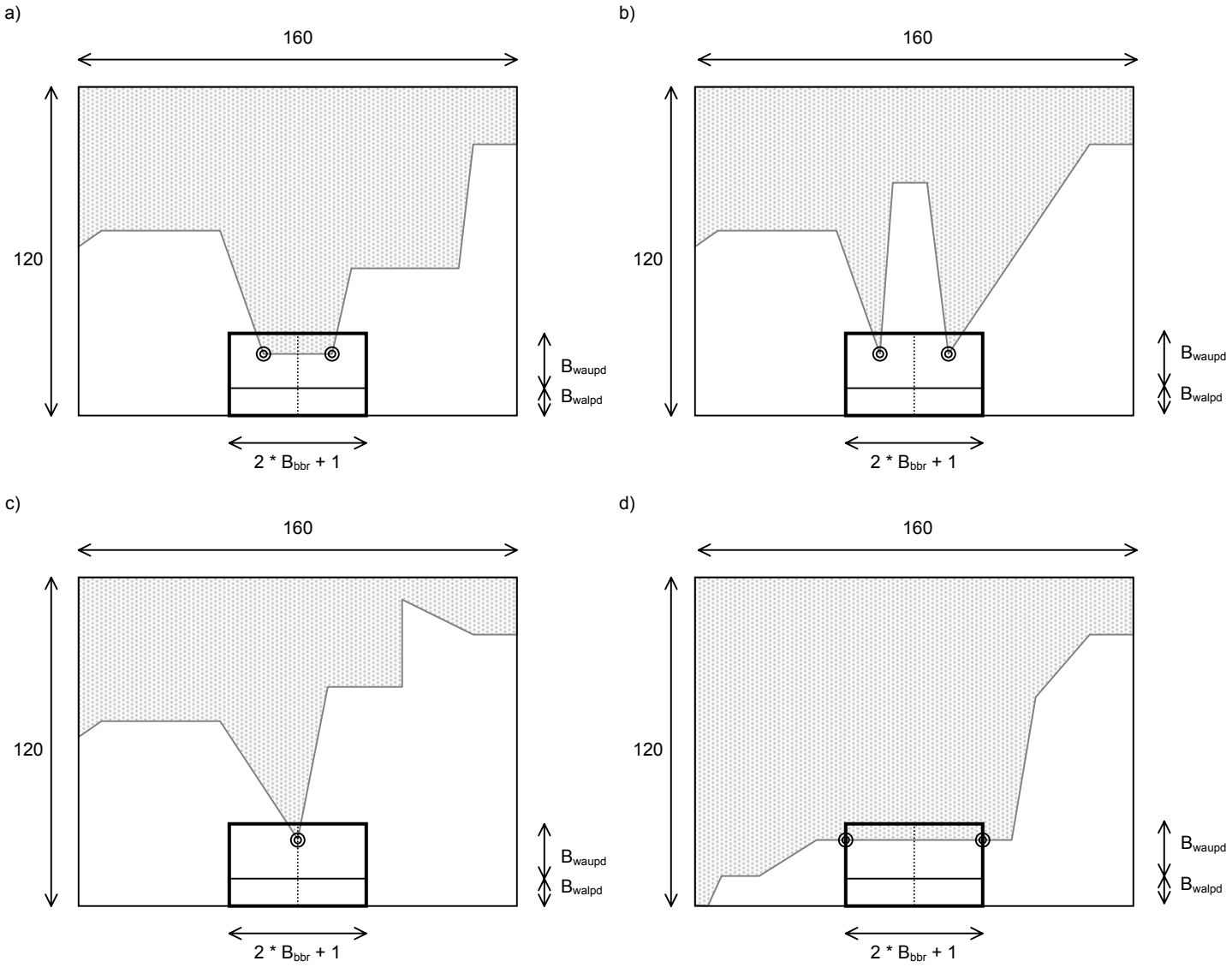

Figura 15. Ejemplos de la estrategia 4. En todos los casos, por más que exista algún camino libre, falla si el recuadro central cumple con las precondiciones de la estrategia. En **a)** encuentra dos píxeles distintos de piso a la misma distancia del lateral izquierda que del derecho. En **b)** se da el mismo caso cuando hacia adelante hay un pasaje muy estrecho. En **c)** se encuentra un único píxel distinto de piso en el centro del recuadro. En **d)** directamente hay una pared de frente.

// 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, intento girar. Más abajo se explican los algoritmos para el giro; ambos reciben por referencia las variables de coordenadas x e y

para poder setearlas si se pudo girar, y retornan verdadero en ese caso o false si no pudieron hacerlo.

```
Si fail = true entonces
        // Elijo hacia dónde girar al azar
        turnTo = random( left, right )

        // Si salió hacia la izquierda, intento hacerlo.
        Si turnTo = left entonces
                Si no turnLeft(&x, &y) entonces
                        // Si falló el intento de giro a la izquierda, intento a la derecha
                        Si no turnRight(&x, &y) entonces
                                // Si también falló, no pude encontrar un nuevo destino
                                newDestination = false

        // Si salió hacia la derecha, intento hacerlo
        Si turnTo = right
                Si no turnRight(&x, &y) entonces
                        // Si falló el intento de giro a la derecha, intento a la izquierda
                        Si no turnLeft(&x, &y) entonces
                                // Si también falló, no pude encontrar un nuevo destino
                                newDestination = false

// 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 newDstination = false y Bronto está quieto entonces rotar hacia derecha o izquierda, al azar, 180ª.
```

Algoritmo TurnLeft

TurnLeft encuentra coordenadas satisfactorias para girar a la izquierda si a la altura b_{walpd} del viewport existe un píxel que sea piso tal que todos los que están a su derecha hasta el centro de la pantalla corresponden a piso, los b_{waulpd} a su izquierda son piso, y el píxel que está inmediatamente a la izquierda de esos b_{waulpd} es distinto de piso o excede los límites del viewport (observe figura 16.a). Es lo que plantea el predicado success del algoritmo mostrado debajo.

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

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

Salida:

Retorna True o False si pudo realizar o no el giro satisfactoriamente.

(x, y), las coordenadas por referencia del viewport para el destino del giro hacia la izquierda satisfactorio encontrado.

// Verifico si existe algún destino satisfactorio para girar, utilizando el predicado success

success = $(\exists\ k \in \aleph, B_{waupd} < k < 79\ \wedge\ (\forall\ w \in \aleph_0, k \leq w < 79\ \wedge\ vps[w, b_{walpd}] = \text{'piso'})\ \wedge\ vps[k - 1, b_{walpd}] \neq \text{'piso'})$

// Si existe lo establezco
Si success = true entonces x = k; y = b_{walpd}.

// Retorno True o False dependiendo del predicado
Retornar success

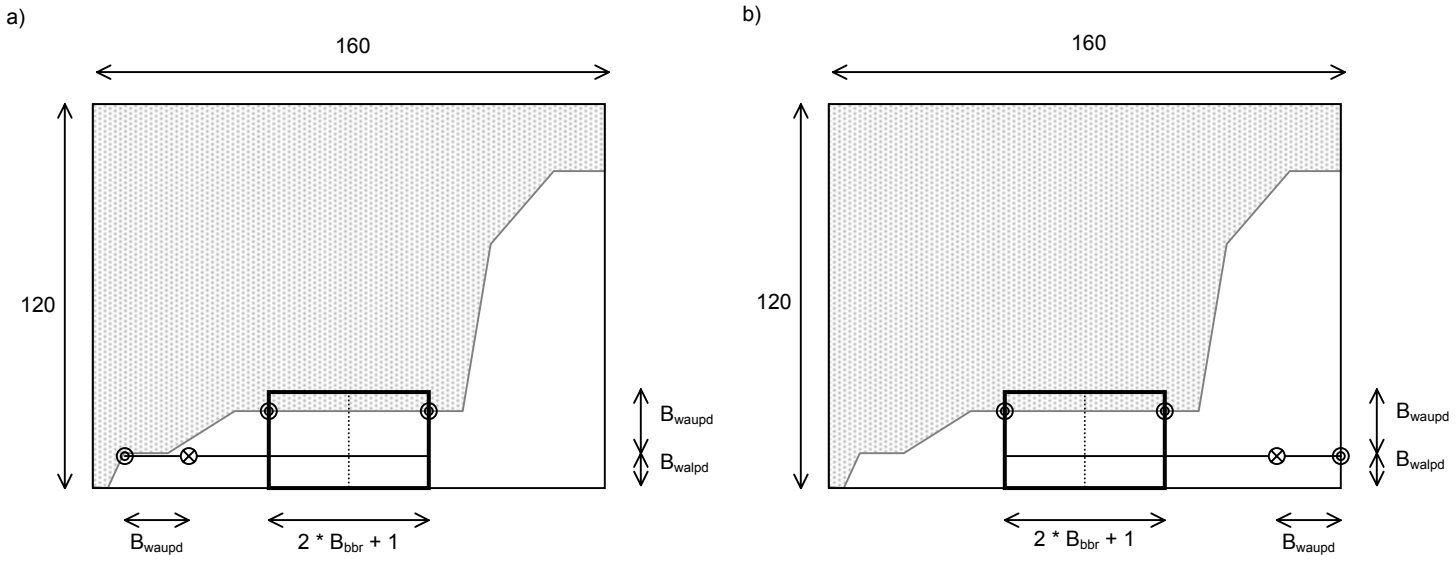

Figura 16. La falla de Walk Around según el caso mostrado en la figura 15.d. En **a)** se optó por girar a la izquierda donde se puede observar el destino elegido y el primer punto en el que la línea desde el centro hacia a la izquierda es distinto de piso. En **b)** se optó por girar a la derecha, todo corresponde a piso así que el margen será tomado desde el extremo derecho del viewport. El círculo con la cruz marca el nuevo destino elegido.

Algoritmo TurnRight

TurnRight es simétrico a TurnLeft; encuentra coordenadas satisfactorias para girar a la derecha si a la altura b_{walpd} del viewport existe un píxel que sea piso tal que todos los que están a su izquierda hasta el centro de la pantalla corresponden a piso, los b_{waulpd} a su derecha son piso, y el píxel que está inmediatamente a la derecha de esos b_{waulpd} es distinto de piso o excede los límites del viewport (observe figura 16.b). Es lo que plantea el predicado success del algoritmo mostrado debajo.

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_{waupd} : Margen lateral y superior del camino satisfactorio en píxeles del viewport.
B_{walpd} : Margen inferior del camino satisfactorio en píxeles del viewport.
Salida:
Retorna True o False si pudo realizar o no el giro satisfactoriamente.
(x, y), las coordenadas por referencia del viewport para el destino del giro hacia la derecha satisfactorio encontrado.

// Verifico si existe algún destino satisfactorio para girar, utilizando el predicado success
success = ($\exists$ k $\in \aleph_0$, 79 < k < (79 - B_{waupd}) ^ ($\forall$ w $\in \aleph_0$, 79 < w $\leq$ k ^ vps[w, b_{walpd}] = 'piso') ^ vps[k + 1, b_{walpd}] $\neq$ 'piso')

// Si existe lo establezco

Si success = true entonces x = k; y = b_{walpd}.

// Retorno True o False dependiendo del predicado
Retornar success

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.

$$C_f \in \Re,\ 0.0 < C_f \leq 100.0$$

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

$\Delta x, \Delta y \in \aleph_0$, cantidad de píxeles de tolerancia en el viewport estático para los ejes x e y.
$\Delta d \in \Re,\ 0 \leq \Delta d \leq 1$, 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:
$o'.x = p.x$; $o'.y = p.y$; $o'.depth = p.depth$;

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,

$$\Delta x = \Delta y = PUr * o.depth$$

$$\Delta d = PUr * (NP / FP)$$

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 psudocódigo 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; y de una tolerancia $\Delta d'$ arbitraria.

Looking for any Power-Up

Buscar por cualquier power-up es como la búsqueda de un power-up específico, pero buscando en el viewport estático cualquier color que se identifique con el de cualquier power-up. En el caso de nuestro FPS se buscarán los colores para energía y municiones. También se elegirá como destino aquel power-up que se encuentre más cerca de la posición de Bronto.

Looking Quickly for a Specific Power-Up

Buscar rápidamente un power-up específico es idéntico al comportamiento descrito anteriormente de 'Looking for a Specific Power-Up', sólo que Bronto aumentará su velocidad, es decir, correrá, en lugar de caminar hacia destino.

Problemas Conocidos

Las heurísticas utilizadas causan tres problemas salientes que describiremos en detalle más abajo. Uno de ellos tiene que ver con los obstáculos formados por los polígonos de la level geometry, y es llamado "The Higher Floor Problem": sucede cuando algo que es visto como 'piso' está demasiado alto como para que Bronto pase sobre él y, entonces, se queda estancado. El segundo problema tiene que ver con la perspectiva utilizada para el render de los viewports y las correcciones que deberían ser aplicadas al recuadro del camino libre para permitir que Bronto elija puntos de destino más distantes; el problema es llamado 'The Perspective Problem'. El último problema tiene que ver con la evasión de obstáculos cuando Bronto está buscando un power-up, ya que elige un punto sin tener en cuenta si existe un camino libre para poder alcanzarlo; el problema es llamado 'The Looking-For Problem'.

Problema "The Higher Floor Problem"

Recordemos que para identificar aquellos polígonos de la Level Geometry que en el static viewport serán representados como piso se utiliza la componente Z de la normal de cada uno de ellos.

Sin embargo, no todos los polígonos que caen en esa clasificación corresponden a 'piso' propiamente dicho –aquellos polígonos en el plano donde el NPC está parado-. Es decir, supongamos que una caja forma parte de la level geometry. Esa caja es lo suficientemente alta como para que Bronto no pueda subirse a ella, pero lo suficientemente baja como para observar la cara superior de la misma. Esa cara superior de la caja se renderiza en el viewport estático como piso.

La heurística Walk Around propuesta tiene inconvenientes cuando se encuentra con este tipo de estructuras.

Figura 17. "The Higher Floor Problem". Bronto frente a un obstáculo en forma de caja que forma parte de la level geometry. A la izquierda se ve el viewport estático donde se puede apreciar el color azul que representa 'pared' de los laterales de la caja; mientras que la cara superior de la misma, en color verde, se confunde con el resto del piso. A la derecha la misma imagen renderizada en forma normal.

Por ejemplo, supongamos que Bronto está relativamente lejos de la caja, observará en el viewport estático los laterales de la misma con el color de 'pared' (Figura 17). Pero a medida que Bronto se acerca a la misma, como el viewpoint está por encima de la cara superior de la caja –es decir, Bronto es más alto que la caja-, se irán perdiendo de vista las caras laterales y eventualmente la cara superior será la única parte visible de la caja, que no podrá diferenciarse del piso propiamente dicho utilizando sólo los colores (Figura 18).

Figura 18. "The Higher Floor Problem". Bronto mucho más cerca de la caja de la figura 17. Walk Around no nota diferencias de alturas de lo que se representa como piso en el static viewport, y en este caso se producirá que Bronto se quede estancado intentado 'atravesar' la caja.

Podrían diferenciarse las superficies si además de los colores se utilizara el depth buffer o directamente desproyectando la imagen. Como Walk Around no hace esto, es decir, sólo utiliza los colores, encontrará siempre un camino libre y elegirá puntos de destino tratando de avanzar hacia delante, produciendo que Bronto se quede estancado en esa posición intentando pasar a través de la caja.

Este problema lo hemos denominado 'The Higher Floor Problem' porque se produce con aquellos polígonos que aparecen en el viewport estático como piso, pero que en realidad están un poco más altos que el plano del piso sobre el que Bronto está parado.

Problema "The Perspective Problem"

Supongamos un pasillo de un ancho constante en toda su extensión que está justo frente al observador. La cantidad de píxeles en el static viewport que representa el piso del pasillo de la parte más cercana es indefectiblemente mayor al de la parte más lejana, efecto causado por la perspectiva (Figura 19). El efecto es análogo al de como una persona real observaría los límites laterales de un campo de tenis si estuviese ubicada en una de las cabeceras del estadio.

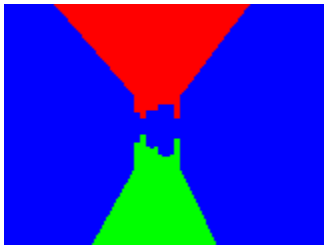

Figura 19. "The Perspective Problem". El static viewport de cuando Bronto está en un pasillo. El ancho del mismo es constante durante toda su extensión; sin embargo, por efectos de la perspectiva, puede observarse que la cantidad de píxeles por cada línea del piso son mayores cuanto más cerca están de Bronto.

Walk Around no toma en cuenta los efectos de la perspectiva, ya que define el camino libre como un rectángulo de base B_{bbr} y altura mínima $b_{waupd} + b_{walpd}$ en el viewport estático, donde todos los píxeles corresponden al color id del piso. Los tres valores mencionados son constantes y definidos por el usuario. Un camino libre más preciso debería definirse como un trapezoide, con base en la línea inferior del viewport y lado superior paralela a ésta, correspondiente al rectángulo de un ancho tal que Bronto quepa exactamente, ubicado en el plano del piso, al cual se le aplicó la perspectiva utilizada. En

la figura 20 se muestra el camino libre actual comparado con el camino libre deseado en una posible situación.

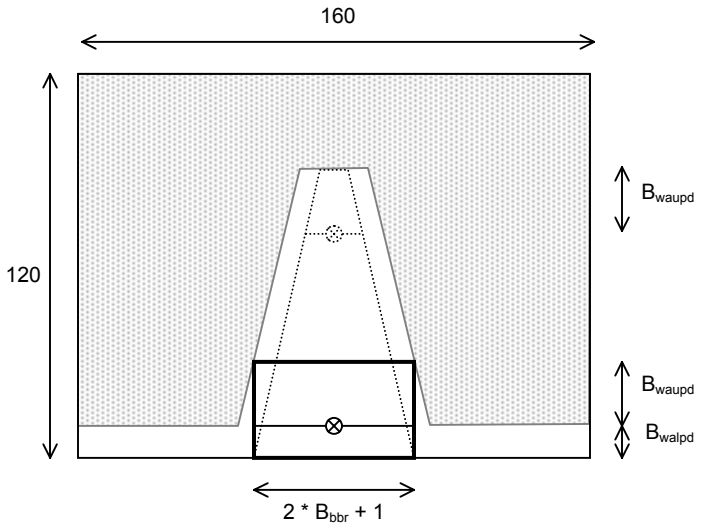

Figura 20. "The Perspective Problem". Posible situación cuando Bronto enfrenta un pasillo. En línea continua se marca el camino libre junto con el punto que elegirá Walk Around como destino. En línea punteada se marca el camino libre con corrección por perspectiva y el punto que elegiría como destino. Nótese la posible enorme distancia entre puntos de destino entre ambas formas de elección.

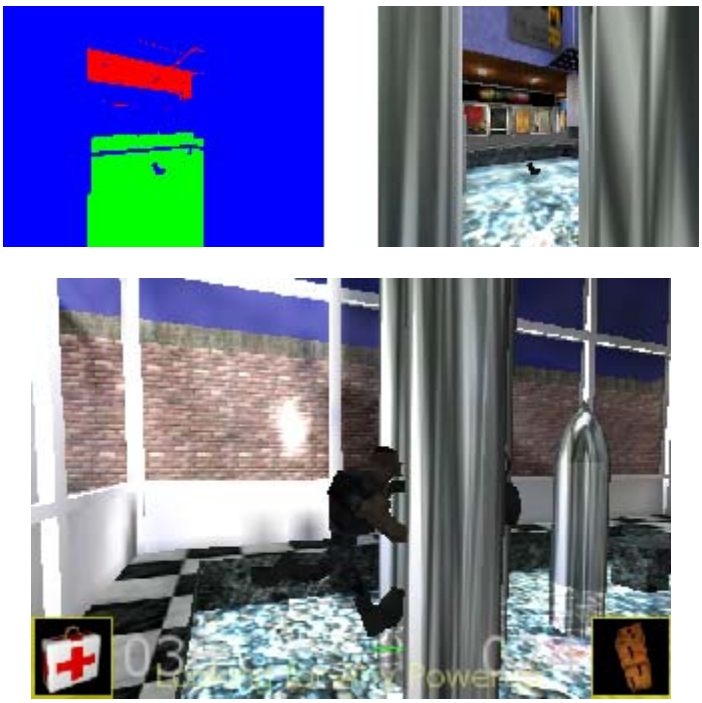

Figura 21. "The Perspective Problem". Bronto estancado intentando pasar por entremedio de dos columnas donde no cabe. En la imagen de arriba a la izquierda se ve el viewport estático, a su derecha la escena renderizada en forma normal desde los ojos de Bronto; la imagen inferior corresponde a una cámara lateral.

Walk Around tal como está produce que Bronto esté 'corto de vista', es decir, siempre optará por puntos de destino relativamente cercanos. Para poder atravesar un pasillo en toda su longitud deberá elegir varios puntos de destino a corta distancia de su posición a medida que avanza. Si se utilizara el camino libre con corrección por perspectiva, podría atravesar el pasillo completamente eligiendo sólo un punto de destino al final del mismo.

Entonces, el valor de B_{bbr} debe ser lo suficientemente pequeño como para que Bronto pueda elegir puntos de destino más lejanos, pero a su vez lo suficientemente grande como para no quedar trabado intentado pasar por un pasillo donde realmente no cabe. La elección de este valor es un trade off que el diseñador debe aplicar por experimentación. En la figura 21 se muestra a Bronto intentando pasar por entremedio de dos columnas donde él no cabe, pero el espacio entre ambas en el static viewport es mayor que B_{bbr}. Bronto quedará estancado allí a menos que se produzca algún cambio de estado que lo libere de la situación.

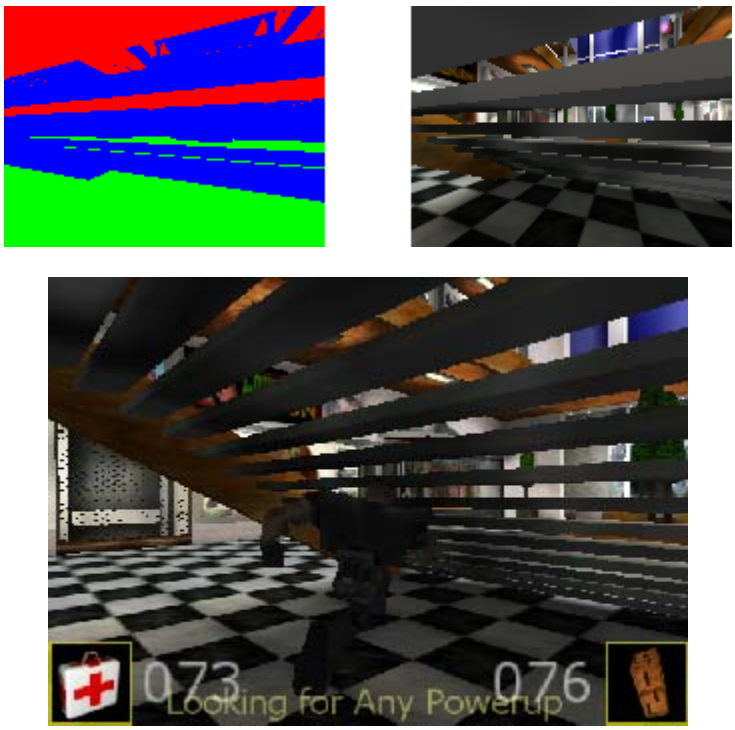

Figura 22. "The Looking-For Problem". Bronto estancado cuando intenta pasar a través de una escalera para llegar a un power-up. En la imagen de arriba a la izquierda se ve el viewport estático, a su derecha la escena renderizada en forma normal desde los ojos de Bronto, y la imagen inferior corresponde a una cámara en tercera persona.

Utilizar el camino libre con corrección por perspectiva resolvería estos problemas; sin embargo, obtenerlo quizás resulte prohibitivo por el tiempo de cálculo que lleva en una aplicación como son los juegos de computadoras. Sería bueno buscar aproximaciones que insuman un costo de procesamiento menor y produzcan de todos modos resultados satisfactorios.

Por la influencia de la perspectiva en esta situación, hemos denominado al problema descrito como 'The Perspective Problem'.

Problema "The Looking-For Problem"

Le heurística empleada cuando Bronto está buscando por algún ítem no toma en cuenta obstáculos ni el ancho de Bronto.

Si hay algún obstáculo que le impida llegar hacia el destino seleccionado Bronto no se fijará en él e intentará seguir recorriendo la curva de Bezier trazada. Podrían darse casos en el que continuamente intente llegar al ítem pero delante suyo hay un obstáculo que no le deja avanzar, generando que Bronto se quede estancado. En la figura 22 se muestra un ejemplo donde Bronto busca municiones, en el viewport estático ve un power-up, y traza el camino para llegar hasta él; sin embargo, Bronto está debajo de una plataforma del tipo escalera y ve el power-up entre los espacios vacíos de la misma; intentará avanzar continuamente sin éxito, quedándose estancado debajo de ella.

Este problema podría solucionarse si se observara si la curva de Bezier trazada atraviesa obstáculos que impidan el paso, generando puntos de destino alternativos para evitarlos, si existieran, o incluso generando más de una curva de Bezier, conectadas entre sí en los extremos y con derivada contínua en los mismos, atravesando los puntos necesarios para poder evitar los obstáculos. En la Figura 23 se muestra esta idea para intentar evitar un obstáculo que se encuentra al trazar la curva original.

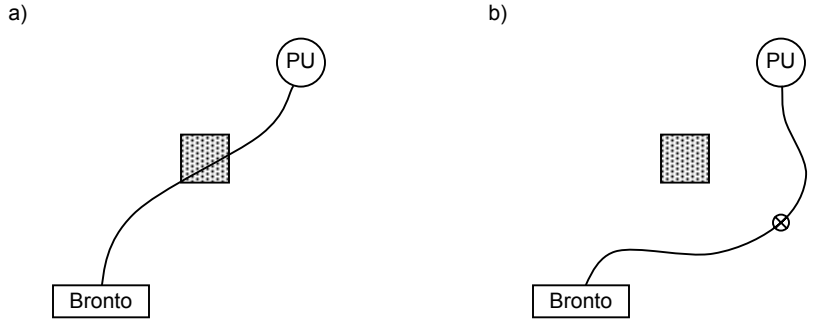

Figura 23. "The Looking-For Problem". En **a)** se ve que el camino seleccionado por Bronto hasta el Power-Up (PU) atraviesa un obstáculo. En **b)** se muestra una posible solución: elegir puntos intermedios que no pasen por el obstáculo y armar de este modo una curva compuesta por más de 1 curva de Bezier hacia el destino final.

Aún no hemos mencionado qué sucede con el ancho. El proceso anterior funcionaría perfectamente, evitando obstáculos, si Bronto fuera un punto. Pero como Bronto tiene un ancho, la curva podría ser trazada y no atravesar ningún obstáculo, pero Bronto podría quedarse estancado si la curva se encuentra entre dos objetos cuya distancia es menor que el ancho de Bronto (Figura 24.a). También puede darse el caso de que la curva pase suficientemente cerca de una pared como para que parte del cuerpo de Bronto intente pasar a través de la misma, produciendo un comportamiento indeseable, hasta que pueda zafar completamente de ella (Figura 24.b). Ambos problemas están presentes en las heurísticas de Looking-for propuestas, es por ello que los hemos incluido bajo el nombre de 'The Looking-For Problem'.

La solución ideal sería, en cada punto de la curva de Bezier o a medida que Bronto avanza sobre la misma, tomar un segmento de recta perpendicular a la tangente de la curva en la posición actual, centrado en la curva y de longitud igual que el ancho de Bronto, y ver si intersecta con algún obstáculo. Si lo hace, aplicar correcciones, como las comentadas anteriormente, generando puntos intermedios para unir varias curvas que eviten los obstáculos y lleguen al destino final.

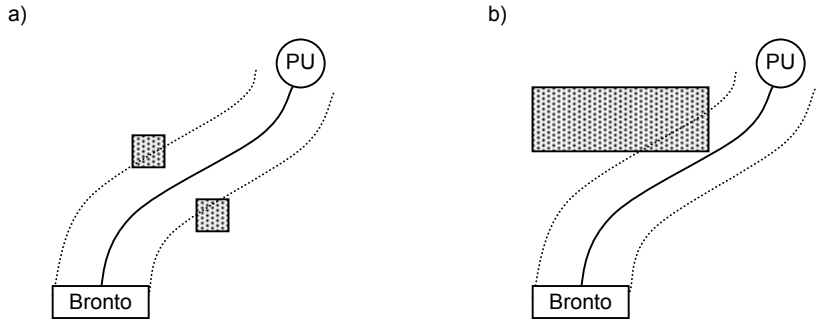

Figura 24. "The Looking-For Problem". En **a)** se ve que el camino seleccionado por Bronto hasta el Power-Up (PU) no atraviesa obstáculos directamente, pero Bronto no pasará por su ancho entre los dos objetos que aparecen en el camino. En **b)** se da un caso similar cuando la curva de Bezier pasa muy cerca de las paredes.

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(θ) 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(θ) < 0, el enemigo viene hacia Bronto; cuando cos(θ) ≥ 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 25.

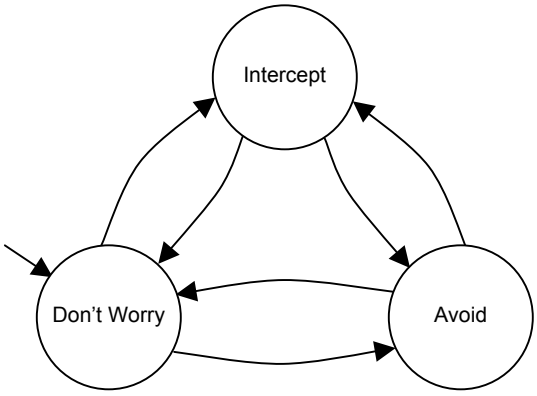

Figura 25. Diagrama de Estados Dinámico. Inicialmente Bronto está en el estado Don't Worry, luego pueden producirse transiciones entre cualquiera de los estados dependiendo de los valores de sus propiedades de energía y municiones, y de su hay o no enemigos presentes en el static viewport (y si los hay, si se acercan o se alejan).

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

Don't Worry

En este estado o bien ningún color id del enemigo ha sido detectado en el viewport estático, o bien todos los enemigos detectados se alejan de Bronto (la componente de color verde es mayor o igual a 0.5) cuando éste tiene los valores de sus propiedades por debajo del umbral superior. Se considera que los enemigos en esta forma están con una actitud inofensiva, entonces Bronto continua con su comportamiento estático correspondiente.

Avoid

En este estado al menos un color id del enemigo ha sido detectado en el viewport estático, ese enemigo se está acercando (la componente de color verde es menor que 0.5), y Bronto tiene alguna de sus dos propiedades por debajo del umbral superior. Se considera que ese enemigo entonces está en una actitud ofensiva, y Bronto no está en condiciones de enfrentarlo, entonces rota 180° para evitarlo. Este comportamiento se impone al comportamiento estático que correspondía ejecutar.

Para buscar un enemigo se utiliza una implementación similar a la del estado Looking-For cuando se busca por un power-up.

Intercept

En este estado al menos un color id del enemigo ha sido detectado en el viewport estático, ese enemigo se está acercando, pero Bronto tiene una cantidad de energía y armas mayor o igual que el umbral superior: está en condiciones de enfrentarlo.

Bronto establece como destino el primer punto que está por debajo del enemigo más cercano, que corresponda a piso y con aproximadamente la misma profundidad, del

mismo modo que cuando Looking-For busca un power-up. Este algoritmo se impone sobre cualquier comportamiento estático. Si no pudo encontrarse un punto de destino que corresponda a piso con las características mencionadas, Bronto continua realizando lo que estaba haciendo según su comportamiento estático

Demostración

El objeto dinámico utilizado como enemigo en la demostración es una bola de metal, llamada 'flying ball', que cambia de dirección al colisionar con las paredes, y viaja con velocidad constante.

Puede verse en el viewport estático con el color amarillo.

El estado *intercept*, al estar implementado del mismo modo que el Looking-For, sufre también el problema llamado "The Looking-For Problem".

Último Comentario sobre IA Dinámica

Como puede observarse, solo estamos usando la componente de color verde de lo provisto por el viewport dinámico.

Las componentes de color rojo y azul podrían ser útiles para mejorar el comportamiento (como el flying ball viaja a velocidad constante no necesitamos utilizar la roja). Por ejemplo, si Bronto tiene que perseguir un objetivo en movimiento, podría estimarse un mejor punto de intercepción, evasión, o de objetivo de disparo usando la magnitud de la velocidad y el seno, demás del coseno, del objeto en movimiento. Además, también sería bueno hacer uso no sólo de la información instantánea, sino de la provista en una serie de frames anteriores, interpoladas o similar, junto a aprendizaje y memoria, para realizar mejores predicciones.

Análisis de Aplicación en Juegos

En esta sección haremos un análisis de cómo podría ser útil el modelo de visión sintética propuesto en los diferentes géneros de juegos. Más allá de que en nuestra implementación hemos optado por un FPS, es posible utilizar el modelo en cualquier otro juego que utilice un motor 3D. Sin embargo, los comportamientos que hemos implementado y explicado en secciones anteriores, serían prácticamente comunes a todos los géneros.

Aventuras (Adventures)

En toda aventura los NPCs cumplen un rol fundamental en el desarrollo de la historia. El jugador debe interactuar constantemente con ellos, comunicarse, interiorizarse en sus vidas. Asimismo, cada NPC tiene características únicas que lo dotan de personalidad. En una aventura completamente 3D, como es el caso de Gabriel Knight III [Gabr01], podría utilizarse la visión sintética en cada uno de ellos, apoyándose en un fuerte módulo de IA, donde se asignarían los atributos necesarios de cada personalidad, y se actuaría de acuerdo a ello. Los agentes, en este caso, además de mantener cierto comportamiento normal, eventualmente podrían asignársele metas, las cuales deben cumplir para poder continuar con la historia según el diseñador ha especificado. Sin embargo, no fijar metas específicas a nivel de diseño abre este tipo de juegos a un realismo nunca antes visto, donde cada personaje podría fijar sus propias metas diariamente ´viviendo su propia vida´. Esto llevaría a que los diseñadores se enfoquen en el desarrollo de personalidades, no se limite a una historia lineal, y se produzcan interacciones impredecibles. Además, el desafío también estará en brindarle al jugador un ambiente atrapante dado lo no determinístico de un juego como el mencionado. Claramente se abrirían nuevos puntos de investigación para construir mejores NPCs, y las posibilidades de diseño se ampliarían.

Acción en Primera Persona (First Person Shooters, FPS)

En un FPS hemos implementado el modelo propuesto de visión sintética. La complejidad del módulo de IA dependerá de la calidad y realismo que se le quiera brindar a cada NPC. En contraste con el género de Aventuras, donde a cada uno de ellos habría que asignarle atributos de personalidad distintos, aquí podrían 'reutilizarse' las personalidades porque, por ejemplo, todos los Skaarj del Unreal [Unre01] estarían dentro del mismo marco. Esto no quiere decir que se comporten todos del mismo modo, pero en líneas generales tendrían el mismo comportamiento por pertenecer a la misma raza, más los atributos extras que podrían llegar a hacer un Skaarj distinto de otro. También sería factible implementar variantes en la representación de la visión; por ejemplo, Predator [Pred01] posee un set de distintas visiones para poder utilizar, entre ellas detección de fuentes de calor. El desafío en esta clase de juegos estará en la capacidad que tengan los NPCs de aprender a combatir cada vez mejor, basándose en la experiencia previa, y en la forma de construir en su memoria el mundo.

Acción en Tercera Persona (Third Person Action Games)

A nivel de la visión sintética no existen diferencias con los FPS, ya que los NPCs seguirían construyendo el mundo a partir de su propia visión independientemente de cómo sea la forma de visualizar el juego para el jugador. Quizás por las características de los niveles de esta clase de juegos, por ejemplo Tomb Raider [Tomb01], donde es común tener que saltar de un lugar a otro para evitar precipicios y caídas, el módulo de IA podría desarrollar una acción de salto más eficiente, siempre teniendo en cuenta lo que el NPC está sensando a través de la visión sintética.

Juegos de Rol (Role Play Games, RPG)

Los juegos de rol pueden aprovechar la riqueza de la visión a un nivel similar a las aventuras, y comportamientos compartidos a los FPS. Se pueden fijar comportamientos autónomos a los avatares o NPC's más importantes, mientras que los monstruos con los que el jugador debe lidiar cada tanto pueden poseer un comportamiento un poco menos realista y más parametrizado, como los que se utilizarían en un FPS. Al igual que en las aventuras, avatares mucho más inteligentes e independientes abren inmensas posibilidades de diseño.

Estrategia en Tiempo Real (Real Time Strategy, RTS)

Los juegos de estrategia 3D en tiempo real podrían aprovechar la visión sintética tanto para unidades 'vivientes', como hombres, animales, alienígenas y monstruos, como para unidades motorizadas. Por ejemplo, un tanque podría tener la visión sintética y utilizarla mediante un módulo de IA que modele el comportamiento del tanque conducido por un humano. Quizás por el aspecto del balanceo del juego se desarrollarían módulos de IA con bastantes restricciones, pero no por eso menos realistas.

Simuladores de Vuelo

Simuladores de vuelo podrían utilizar la parte de visión sintética para cálculos más realistas de trayectorias y disparos. Por ejemplo, podría dotarse de visión sintética a un misil, el cual basándose en ella podría aproximarse al objetivo detectando fuentes de calor a partir de la representación de la imagen. Torretas anti-aviones podrían aprovechar el mismo mecanismo para detectar trayectorias de aviones y poder dispararles haciendo cálculos basándose en lo que ´ven´ en el instante actual y los previos.

Otros Escenarios

Sería interesante experimentar en otros escenarios como en competencias automovilísticas, donde se tomaría la visión del piloto y el módulo de IA modelaría su comportamiento y acciones en base a lo que ve y a las características del conductor. También sería bueno dotar de visión a todos los participantes de un juego colectivo, como el fútbol, y dependiendo de las características y ánimo de cada jugador, tomar decisiones de un modo distribuido, individual y autónomo para llevar adelante objetivos colectivos.

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 robot o computer vision, pero restringiendo los datos que recibe el módulo de inteligencia artificial sólo a lo sensado (caso contrario al no uso de visión donde se tiene acceso a todo).

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.

Finalmente, se analiza y discute brevemente el potencial de la visión sintética y autonomía de NPC's en los distintos géneros de los juegos de computadora hoy en día.

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.

Trabajo a Futuro

En la parte gráfica de la visión sintética podemos nombrar:

- Experimentar con representaciones 'no humanas', para agentes con mayores capacidades de visión, como podría ser visión infrarroja, sensado por calor, o cualquier otro tipo de mejora o distinción en la visión (como podrían ser las distintas visiones de Predator [Pred01] o el visor de Geordi La Forge [Star01] en Star Trek: The Next Generation [Star02]).

- Investigar variantes que permitan representar los efectos de la iluminación. Por ejemplo, que en una zona oscura sea mucho más difícil identificar un objeto que en una zona totalmente iluminada, donde la identificación sería perfecta.

- Investigar cómo es posible agregar algún tipo de ruido en la representación de la visión dado un índice, llamémosle de 'cansancio visual', que degrade la visión a causa (pensando en un juego) del cansancio, salud o algún otro factor que esté afectando al agente. También podría variar el índice de visión según si se pasa de una zona muy iluminada a una con poca iluminación (el ojo tardaría en acostumbrarse al nuevo ambiente) y viceversa.

Para el campo de la inteligencia artificial sin duda lo que se requiere es desarrollar módulos complejos que mejoren y extiendan ampliamente el presentado en esta tesis, y que hagan uso de la visión propuesta. No sólo en el género de las FPS, sino llevar el comportamiento a otros géneros. Algunas de las ideas que surgen:

- Memoria. Que el agente mantenga una memoria para que, por ejemplo, sepa dónde está un power-up cuando lo necesite por haberlo visto previamente. También sería útil para recordar eventos que hayan sucedido, cambios de estados (un objeto que antes estaba y ahora no), etc.

- Aprendizaje. Que aprenda en base a lo que está viendo para mejorar tácticas de ataque/evasión o por qué lugares del nivel lo conviene ir según la experiencia que tuvo anteriormente.

- Interacción. Que pueda interactuar con otros agentes identificándolos por la visión, elemento muy útil en el género aventura o RPG.

- Personalidad. Brindar atributos de personalidad y basándose en la memoria, aprendizaje e interacción vaya forjando su propia vida, convirtiéndose en un agente totalmente autónomo dentro de un mundo virtual como podría ser en un Massive Multiplayer OnLine RPG.

No debemos dejar de mencionar que también está como trabajo a futuro:

- Integrar el sistema de visión en todo género de juego factible.

- Realizar un juego completo con NPC's que utilicen únicamente el módulo de inteligencia artificial más la información sensada por el sistema de visión sintética.

- Ampliar el sistema de sensado con el resto de los sentidos: auditivo, táctil, olfato y gusto.

- Investigar las posibilidades de nuevos géneros de juegos y gameplay que podrían obtenerse en base a un mundo habitado completamente por personajes autónomos.

Referencias

[Blum01] Synthetic Characters Media Group Website: http://www.media.mit.edu/characters/. MIT.

[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.

[Eber01] David H. Eberly. 3D Game Engine Design. Chapter 3: The Graphics Pipeline. Morgan Kaufmann. 2001. ISBN 1-55860-593-2.

[Ente01] PadCenter level by Andreas Ente. E-mail: ENTE.endres@t-online.de. Website: http://padworld.exp-network.de/.

[Fly01] Fly3D Website: http://www.fly3d.com.br/. Paralelo Computaçao.

[Gabr01] Gabriel Knight III Website: http://www.sierrastudios.com/games/gk3/. Sierra Studios.

[Isla02] 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.

[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.

[Kuff01] James J. Kuffner Website: http://robotics.stanford.edu/~kuffner/.

[Lair00a] Human-level AI's Killer Application: Interactive Computer Games. John E. Laird; Michael van Lent. AAAI, August 2000.

[Lair00b] Creating Human-like Synthetic Characters with Multiple Skill Levels: A Case Study using the Soar Quakebot. John E. Laird; John C. Duchi. AAAI 2000 Fall Symposium Series: Simulating Human Agents, November 2000.

[Lair01] Design Goals for Autonomous Synthetic Characters. John E. Laird.

[Lair02] John E. Laird AI & Computer Games Research Website: http://ai.eecs.umich.edu/people/laird/gamesresearch.html/. University of Michigan.

[Micr01] Microsoft DirectX Website: http://www.microsoft.com/directx/. Microsoft Corp.

[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, Nº1, 1995, pp. 7-19.

[Nose95b] Synthetic Vision and Audition for Digital Actors. Hansrudi Noser; Daniel Thalmann. Proc. Eurographics `95, Maastricht.

[Nose98] Sensor Based Synthetic Actors in a Tennis Game Simulation. Hansrudi Noser; Daniel Thalmann. The Visual Computer, Vol.14, No.4, pp.193-205, 1998.

[Open01] OpenGL Website: http://www.opengl.org/.

[Pred01] Predator. Directed by John McTiernan; with Arnold Schwarzenegger and Carl Weathers. Fox Movies, 1987.

[Quak01] Quake Website: http://www.quake.com/. Id Software.

[Rabi01] Motion, Stereo and Color Analysis for Dynamic Virtual Environments. Tamer F. Rabie; Demetri Terzopoulos.

[Rena90] A Vision-Based Approach to Behavioral Animation. Olivier Renault; Nadia Magnenat Thalmann; Daniel Thalmann. Journal of Visualization and Computer Animation, Vol.1, No1, 1990.

[Reyn01] Steering Behaviors For Autonomous Characters. Craig W. Reynolds. Website: http://www.red3d.com/cwr/steer/gdc99/.

[Star01] Geordi La Forge character information at Star Trek Website: http://www.startrek.com/library/individ.asp?ID=112463. Paramount Interactive.

[Star02] Star Trek Website: http://www.startrek.com/. Paramount Interactive.

[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.

[Tomb01] Tomb Raider Website: http://www.tombraider.com/. Eidos Interactive.

[TuTe94] 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, pp. 43-50.

[Unre01] Unreal Website: http://www.unreal.com/. Epic Games, 1998.

[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: Advanced Real-time Rendering and Animation, Addison-Wesley 2002 (in press).

[Wolf01] Wolfenstein 3D Website: http://www.idsoftware.com/games/wolfenstein/wolf3d/. Id Software.

Apéndice A – Contenido del CD

Lo más relevante que podrá encontrar en el CD que acompaña esta tesis es:

- Fly3D 2 Beta 8

  Versión Beta 8 del motor Fly3D, adaptada para soportar el render especial para los viewports estático y dinámico. En subcarpetas podrán encontrarse los niveles y plug-ins modificados/creados.

  La carpeta es: "\flysdk2b8\".

- Fly3D Plug-Ins

  - Walk
    Se ha modificado el walk.dll existente para que soporte el trazado de la curva de Bezier tras la selección del destino por el módulo de IA.
    La carpeta es: "\flysdk2b8\plugin\walk\"

  - Synthetic Vision
    Se ha creado el svision.dll que contiene el módulo de IA junto con la generación y procesamiento de los viewports estático y dinámico.
    La carpeta es: "\flysdk2b8\plugin\svision\"

- DirectX 8.1

  API Microsoft DirectX 8.1.

  La carpeta es: "\directx\".

- Tesis

  Este documento.

  La carpeta es: "\thesis\".

- Papers Referenciados

  Algunos papers de los referenciados en esta tesis.

  La carpeta es: "\papers\".

- ASAI 2002

  Paper y presentación presentados en ASAI 2002, 31° JAIIO, Santa Fe, Argentina.

  La carpeta es: "\asai2002\".

Apéndice B – Implementación

La implementación del módulo de visión sintética y de inteligencia artificial descriptos en esta tesis se realizó sobre el motor gráfico Fly3D [Fly01].

Del Fly3D se han reutilizado algunos de los plug-ins existentes, modificado otros, y creado nuevos. Asimismo, fue necesario también la adaptación de algunos métodos del motor para soportar la funcionalidad requerida por el módulo de visión sintética.

Si bien podría obtenerse una mejor modularización y optimización de lo implementado, creemos que lo realizado fue suficiente como para demostrar la utilidad de la visión sintética.

Comentaremos brevemente los detalles importantes de la implementación en cada componente.

Clases de Fly3D_engine

Las modificaciones que se realizaron en el motor fueron absolutamente necesarias para poder renderizar los viewports estáticos y dinámicos con los mapeos de colores correspondientes. Si bien esto produce una única flyEngine.dll, TODOS los módulos debieron ser recompilados porque están directamente ligados con esta biblioteca.

Clase flyBezierPatch

Se agregaron dos modos de dibujo en el método 'draw', uno correspondiente al viewport estático y otro al dinámico.

Clase flyBspObject

Se agregaron las propiedades 'colorid' y 'dyncolor' que contienen el color correspondiente para el viewport estático, y el calculado para el viewport dinámico, para el objeto bsp.

Se modificó el método 'draw' para asignar a cada cara del mesh el color de 'colorid'.

Clase flyEngine

Se agregó la propiedad 'maxvel' que indica la velocidad máxima que puede alcanzar un objeto, útil para mapear la magnitud de la velocidad en el viewport dinámico.

Se modificó el método 'draw_faces' para que si el modo 'debug' está en 256 se renderice según viewport estático, y si está en 128 según viewport dinámico. Aquí se asignan los colores para la level geometry.

Se modificó el método 'step_objects' para calcular y asignar el color correspondiente en el viewport dinámico a cada objeto.

Clase flyFace

Se agregaron las propiedades 'colorid' y 'dyncolor' que contienen el color correspondiente para el viewport estático, y el calculado para el viewport dinámico, para la cara.

Clases de svision

Este es un nuevo plug-in que contiene la bajada a memoria de los buffers de los viewports y todo el código para controlar el comportamiento del NPC según fue definido en esta tesis.

Las propiedades globales son:
- 'depthbuffer', contiene el depth buffer del último frame.
- 'dynamicbiffer', contiene el buffer del dynamic viewport del último frame.
- 'staticbuffer', contiene el buffer del static viewport del último frame.

A través del método global 'draw_game_status' se pone en pantalla el estado actual del NPC.

Clase ai

Contiene todo el código del comportamiento del NPC.

Las propiedades más importantes son:
- 'bwalpd', margen inferior en píxeles.
- 'bwaupd', margen lateral y superior en píxeles.
- 'cutfactor', porcentaje de corte del camino actual.
- 'hid', color id de power-ups de energía.
- 'hlt', umbral inferior de energía.
- 'hut', umbral superior de energía.
- 'player', referencia al objeto person que se está controlando con esta clase.
- 'radio', B_{bbr}.
- 'state', el estado actual en el que se encuentra el NPC.
- 'wid', color id de power-ups de municiones.
- 'wlt', umbral inferior de municiones.
- 'wut', umbral superior de municiones.

Los métodos:
- 'b_addobject', utilizado para agregar objetos a la lista cuando se necesitan obtener los objetos que hay en el static viewport.
- 'b_getanynextobject', obtiene el objeto más cercano de una lista de objetos, utilizado cuando se buscan ambos power-ups.
- 'b_getnextobject', obtiene el objeto más cercano de un color determinado a partir de una lista de objetos.
- 'b_lookboth', heurística cuando se busca por ambos power-ups.
- 'b_lookfor', heurística cuando se busca por algún power-up específico.
- 'b_turnleft', heurística para girar hacia la izquierda en walk around.
- 'b_turnright', heurística para girar hacia la derecha en walk around.
- 'b_walkaround', heurística de walk around.
- 'step', establece el estado actual en cada frame e invoca a la heurística correspondiente.

El comportamiento dinámico está embebido en los métodos de las heurísticas estáticas.

Clase svobject

Es la estructura que se utiliza para los objetos encontrados en estrategias Looking-for.

Clase vision

Es utilizada para bajar la información de los viewports a memoria. Contiene las siguientes propiedades:

- 'player', referencia al objeto person a controlar por intermedio de visión sintética.
- 'vpstatic', referencia al viewport estático.
- 'vpdynamic', referencia al viewport dinámico.
- 'width', ancho en píxeles de los viewports. Actualmente soporta sólo 160.
- 'height', alto en píxeles de los viewports. Actualmente soporta sólo 120.

El método implementado es el 'step', que por cada frame obtiene del color buffer los píxeles del viewport estático y dinámico para guardarlos en memoria, y lo mismo con el depth buffer del viewport estático.

Clases de viewport

Clase viewport

Se agregaron las propiedades:

- 'coordmode', 0 utiliza las coordenadas porcentuales como en el viejo viewport (xf, yf, wxf, wyf); 1 utiliza las coordenadas en píxeles (xi, yi, sxi, syi), invertidas en y; 2 utiliza el sistema de coordenadas especial para la demostración: centrada el viewport normal, a xi píxeles de la izquierda el estático, y a xi de la derecha el dinámico.
- 'mode', 0 normal; 1 estático, 2 dinámico.
- 'camangle', ángulo para el rendering del viewport.
- 'xi', píxeles a partir de la izquierda en que aparece el viewport.
- 'yi', píxeles a partir de la parte superior en que aparece el viewport.
- 'sxi', ancho del viewport en píxeles.
- 'syi', alto del viewport en píxeles.

Se modificó el método 'draw' para que soporte los distintos modos del viewport. Es quien se encarga de setear el modo debug en 128 o 256, según corresponda.

Clases de walk

El plug-in walk sufrió muchas modificaciones. Se agregaron al mismo las cámaras implementadas y el objeto representado en la demostración como el 'flying ball'. Las modificaciones están básicamente en la clase 'powerup' para poder soportar la energía y las municiones tal como fueron definidas, y por sobre todo en la clase 'person'. En esta clase se agregaron los métodos y propiedades tal que sea posible la interpretación del punto de destino elegido en el static viewport para trasladarlo a un destino en coordenadas del mundo, y luego trazar un camino entre la posición actual y la final.

A través del método global 'draw_game_status' se pone en pantalla los valores de las propiedades Energía y Municiones del NPC.

Clase camera

Nueva clase. Es una cámara sin detección de colisiones que sigue a una determinada distancia a su objetivo. La propiedad 'source' debe contener un flyBspObject que será el objetivo de la cámara. La propiedad 'displacement' indica la distancia a la que permanecerá la cámara del objetivo.

Clase camera2

Nueva clase. Es una cámara con detección de colisiones que intenta seguir a una determinada distancia constante a su objetivo. Posee las siguientes propiedades:

- 'source', debe contener un flyBspObject que será el objetivo de la cámara.
- 'dist', indica la distancia mínima a la que debe permanecer la cámara del objetivo.
- 'radius', es el radio del bounding box.
- 'maxvel', es la velocidad máxima que puede alcanzar la cámara cuando intenta acercarse a su objetivo hasta cumplir la distancia 'dist'.

Clase object

Nueva clase. Pone un objeto con movimiento que cambia de dirección al colisionar. Utiliza la velocidad constante indicada por la propiedad heredada 'vel'.

Clase person

Esta clase es la que contiene al NPC. Es la que se encarga de trazar la curva de Bezier una vez elegido el destino, y quien hace recorrer al agente a través de ella.

Las propiedades agregadas son:

- 'mode', 0 si se desea usar el modo walk normal; 1 si se desea usar la interfase de selección de puntos de destino interactiva, similar a la del shop.dll provista con Fly3D; 2 si corresponde a un comportamiento por visión sintética.
- 'svhealth', valor actual de energía.
- 'svinithealth', valor inicial de energía.
- 'svweapon', valor actual de municiones.
- 'svinitweapon', valor inicial de municiones.
- 'maxhealth', valor máximo que puede alcanzar la energía.
- 'maxweapon', valor máximo que puede alcanzar las municiones.
- 'healthtime', intervalo para hacer disminuir la energía.
- 'weapontime', intervalo para hacer disminuir las municiones.
- 'healthfactor', cantidad en que se debe disminuir la energía una vez cumplido el tiempo especificado.
- 'weaponfactor', cantidad en que se debe disminuir las municiones una vez cumplido el tiempo especificado.
- 'healthpowerup', cantidad en que debe incrementarse la energía al tomar un power-up del mismo.
- 'weaponpowerup', cantidad en que deben incrementarse las municiones al tomar un power-up del mismo.
- 'healthcurtime', tiempo desde que se tomó el último power-up de energía o se disminuyó su cantidad.
- 'weaponcurtime', tiempo desde que se tomó el último power-up de municiones o se disminuyó su cantidad.
- 'movesprite', referencia a un spritelight para marcar el cursor del mouse en modo 1 y 2.
- 'targetsprite', referencia a un spritelight para marcar el destino seleccionado en modo 1 y 2.
- 'spriteflag', indica si el movesprite debe ser dibujado.
- 'targetflag', indica si el targetsprite debe ser dibujado.
- 'characterflag', indica si el NPC está quieto o en movimiento.
- 'walkpath', curva de Bezier para el camino a seguir por el NPC.
- 'path_dist', distancia estimada entre los puntos de control de la curva de Bezier.
- 'path_len', longitud de la curva de Bezier.
- 'path_factor', porcentaje de la curva de Bezier ya recorrida.

- 'svx', coordenada x en el viewport estático del punto de destino elegido, -1 si no hay ninguno.
- 'svy', coordenada y en el viewport estático del punto de destino elegido, -1 si no hay ninguno.
- 'vpstatic', referencia al viewport estático.
- 'lastpos', contiene la posición anterior del NPC, utilizado para saber si quedó estancado o no.
- 'stucktime', tiempo para determinar si el NPC quedó estancado o no en una misma posición.
- 'signo', utilizado para determinar hacia dónde gira una vez que sabe que quedó estancado.
- 'drawpath', flag para dibujar o no la curva de Bezier.
- 'svwalkvel', velocidad del NPC mientras camina en modo 1 y 2.
- 'svrunvel', velocidad del NPC cuando corre en modo 2.

Las métodos modificados y agregados son:
- 'die', simplemente se modificó el mensaje de cuando el personaje muere.
- 'draw', se modificó para dibujar los sprites del cursor 3D y el objetivo, cuando corresponda, y para trazar la curva de Bezier.
- 'init', se agregaron las inicializaciones necesarias para el trabajo con la visión sintética.
- 'move2', cuando mode es 1, procesa los puntos de destino del NPC según lo ingresado mediante el mouse por el usuario. Es una interfase similar al shop.dll de Fly3D.
- 'move3', cuando mode es 2, procesa los puntos de destino según lo elegido por el módulo de IA para generar la curva de Bezier, cuando no debe generarla hace que el NPC camine por la curva ya generada, o si se queda 'estancado' hace que gire 180°.
- 'step', por cada frame invoca al move correspondiente dependiendo del mode; además, controla los valores de energía y municiones para hacerlos decaer si se cumplió el tiempo determinado.

Clase powerup

Se agregó el override del método 'get_mesh' para que devuelva el mesh del power-up sólo cuando es visible (un power-up es invisible cuando fue tomado y todavía no se regeneró).

Se modificó el método 'powerup_get' para que incremente las propiedades, Energía y Armas, según lo indicado en el objeto person si se está trabajando con el modo de visión sintética.

Apéndice C – Guía del Usuario

Requerimientos

El motor gráfico Fly3D versión 2.0 requiere Windows 9x/Me/2k/NT4 para poder correr. También se necesita una buena placa de aceleración 3D (se recomienda con procesador nVidia), que soporte OpenGL. Asegúrese de estar utilizando los últimos drivers disponibles para la misma, además de tener instalados los últimos Service Packs de Windows y la última versión de DirectX.
Para poder ejecutar el motor en NT4 es necesario tener instalado Service Pack 5 o superior.

Para más información refiérase a [Fly01].

Instalando DirectX

Si no tiene instalado DirectX en su sistema, refiérase al Apéndice A para ver en qué carpeta del CD se encuentra el software (DirectX 8.1) o visite por Internet [Micr01].

Ejecute el archive correspondiente a su sistema operativo y siga las instrucciones que aparezcan en pantalla.

Configurando Fly3D

Luego de haber instalado todo, lo primero que debe hacer para poder usar Fly3D sin problemas es configurarlo. Para ellos, siga los siguientes pasos:

1. Ejecute flyConfig.exe, ubicado en el path de Fly3D. Una ventana se abrirá (Figura C-1).
2. Si está chequeada, deschequee la opción 'Customized data and plugin folders'.
3. Seleccione la resolución de pantalla completa deseada en el combo 'Fullscreen Video Mode'.
4. Seleccione el modo de rendering deseado en el combo 'Rendering Modes'. La primera vez que ejecute Fly3D en su máquina, el mayor modo de rendering será automáticamente seleccionado por defecto. Siempre es preferible un modo acelerado por hardware que uno de rendering por software.
5. Presione el botón 'Test', luego 'Save', y por último 'Exit'.

Luego, mientras ejecuta Fly3D, si las imágenes se ven muy oscuras o brillantes deberá volver a ejecutar la configuración para ajustar el valor de brillo (brightness), una vez que lo haga repita el paso 5.

Deberá hacer lo mismo si desea cambiar la resolución de pantalla completa o el modo de rendering.

Ejecutando Fly3D

Antes de ejecutar Fly3D por primera vez, asegúrese de cumplir con los requerimientos del sistema y de haberlo configurado.

Para correr Fly3D, ejecute el archive flyFrontEnd.exe ubicado en el path de Fly3D.

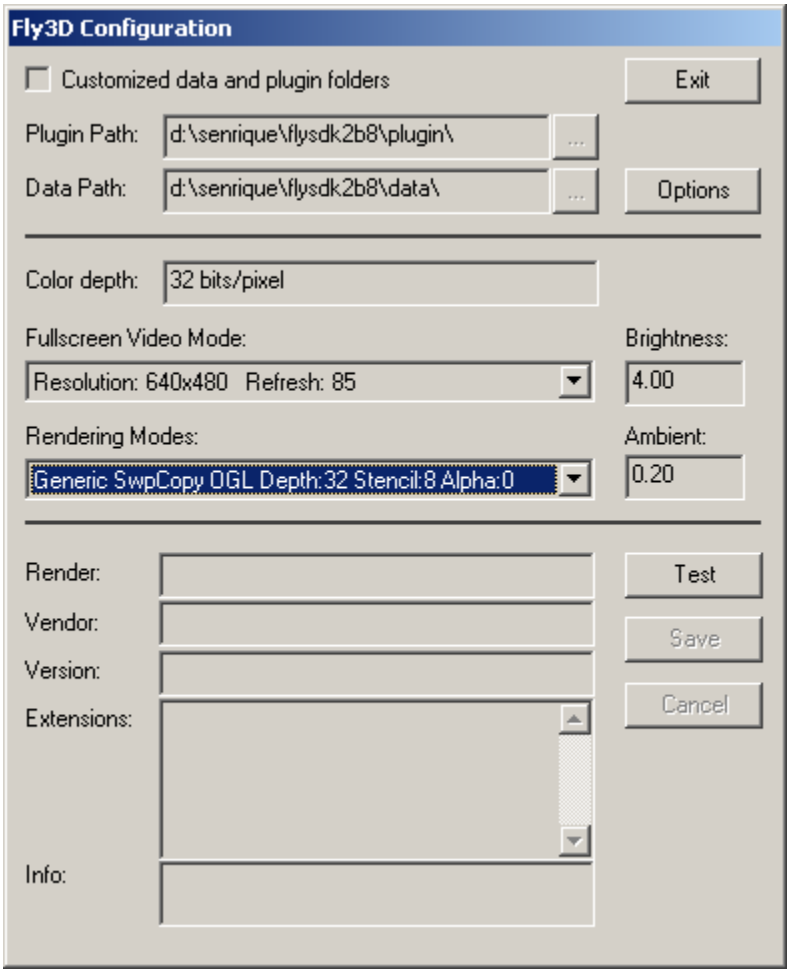

Figura C-1. Ventana de configuración de Fly3D.

Ejecutando Niveles de Visión Sintética

Luego de ejecutar Fly3D aparecerá un menú (figura C-2). Elija 'Single Player' presionando 'Enter'.

El menú de selección de niveles aparecerá (figura C-3). Presione la flecha hacia abajo del cursor, y luego derecha o izquierda varias veces hasta encontrar el nivel que desee. Luego presione 'Enter' para cargarlo.

Cuatro niveles que usan la visión sintética son provistos en el CD:

- Synthetic Vision 1: Un nivel propio con una cámara sin detección de colisiones (la cámara atravesará paredes) ubicada detrás de Bronto.

- Synthetic Vision 2: El mismo nivel propio pero esta vez utilizando una cámara con detección de colisiones.
- Synthetic Vision 3: Un nivel del Quake III Arena [Quak01] convertido al formato aceptado por Fly3D, con el agregado de power-ups y cielo propio. La cámara colisiona.
- Synthetic Vision 4: El nivel PadCenter [Ente01] con los power-ups propios. La cámara colisiona.

Figura C-2. Menú principal de Fly3D FrontEnd.

Una vez que el nivel fue cargado, presione primero la tecla 'B' para activar el Viewport Dinámico y luego la tecla 'C' para activar el Estático. Luego de que ambos viewports fueron activados (en ese orden) el módulo de IA comenzará a controlar el comportamiento de Bronto.

En cualquier momento durante la simulación podrá presionar la tecla 'V' para activar o desactivar el viewport que muestra la escena renderizada normalmente desde el punto de vista de Bronto.

Para salir del nivel presione 'ESC'.

Luego de ello, podrá repetir los pasos descritos desde el principio de esta sección para cargar otro nivel.

Si desea salir de Fly3D, presione la flecha hacia abajo del cursor hasta que la opción 'Quit' aparezca en el menú principal de Fly3D FrontEnd, luego presione 'Enter'.

Figura C-3. Menú de selección de niveles de Fly3D FrontEnd.

Modificando Propiedades de los Niveles de Visión Sintética

Para editar las propiedades de los niveles de visión sintética ejecute flyEditor.exe desde el path de Fly3D y cargue en él el nivel deseado:

- Synthetic Vision 1: sv08.fly.
- Synthetic Vision 2: sv09.fly.
- Synthetic Vision 3: sv10.fly.
- Synthetic Vision 4: sv12.fly.

Refiérase a la literature de Fly3D [Watt01; Watt02] o al website [Fly01] para conocer más sobre flyEditor.

Algunos de los parámetros interesantes para cambiar y *jugar* son:

Parámetros Globales

- Nearplane (NP)
- FarPlane (FP)
- Camangle (Ángulo de la Cámara)
- Camera, valores posibles:
    - camera2: Cámara en tercera persona con detección de colisiones.
    - 3rd_Person: Cámara en tercera persona sin detección de colisiones
    - player: Cámara en primera persona.

SVision plug-in; propiedades de los objetos de la clase "ai"

Los parámetros 'Color Id' en esta clase sirven para poder identificar qué es munición y qué es energía.

- Health Upper Threshold (H_{ut})
- Health Lower Threshold (H_{lt})
- Weapon Upper Threshold (W_{ut})
- Weapon Lower Threshold (W_{lt})
- Health Color Id (Color de power-ups de energía)
- Weapon Color Id (Color de power-ups de municiones)
- WA Upper Pixel Dist (b_{waupd})
- WA Lower Pixel Dist (b_{walpd})
- Cutfactor (C_f)
- Radio (B_{bbr})

Walk plug-in; propiedades de los objetos de la clase "person"

- svhealth (H_{ini})
- svweapon (W_{ini})
- healthtime (t_{dh})
- weapontime (t_{dw})
- healthfactor (D_h)
- weaponfactor (D_w)
- healthpowerup (A_h)
- weaponpowerup (A_w)
- drawpath (1 o 0: Dibuja o no la curva de Bezier)

Walk plug-in; propiedades de los objetos de la clase "powerup"

Cada objeto debe tener el tipo y color id correspondientes seteado.

- ColorId
- Type (1: Municiones; -1: Energía)
- Spawntime (Tiempo de regeneración del power-up luego de que fue tomado)

Es posible agregar tanto power-ups como desee.

Walk plug-in; propiedades de los objetos de la clase "object"

Este objeto es el 'flying ball'. El único color id soportado por ahora es (1.0, 1.0, 0.0).

- ColorId

Es posible agregar tantos 'flying balls' como desee.

Notas

- Todas las variables que indican tiempo están en milisegundos.
- Todas las variables de porcentaje reciben valores entre 0 y 1.
- Color Ids son vectores de punto flotante. Actualmente no todos los colores disponibles funcionan, ya que en ciertas partes del código se usa 'unsigned int' y en otros 'float', produciendo errores de precisión al convertirlos entre ellos. Esto será solucionado en futuros releases.

Apéndice D – Glosario

2D : Dos dimensiones.

3D : Tres dimensiones. Tridimensional.

IA : Inteligencia Artificial.

ECI : Escuela de Ciencias Informáticas.

FPS : First Person Shooter. Género de juegos de disparos en primera persona.

NPC : Non Player Character. Personaje manejado por la computadora.

RGB : Red-Green-Blue. Modelo de representación del color.

RPG : Role Player Game. Juego de rol.

Apéndice E - Lista de Figuras

Apéndice F - Lista de Tablas