ISSN 1870-4069

Máquina de estados finita con comportamientos de dirección para manipular agentes de juego RedCell

Abner Quiroz Clemente, Héctor Adrián Díaz Furlong, Verónica González Rivera, Juan Jesús Cuéllar González

Benemérita Universidad Autónoma de Puebla, Escuela de Artes Plásticas y Audiovisuales, Puebla, México
{abner.quiroz, hector.diazfurlong}@correo.buap.mx,
{veronica.gonzalezri, juan.cuellarg}@alumno.buap.mx

Resumen. RedCell es un videojuego surgido del Global Game Jam 2018, creado por integrantes de la Escuela de Artes Plásticas y Audiovisuales (ARPA) de la Benemérita Universidad Autónoma de Puebla, en el cual se lleva a cabo una competencia dentro del cuerpo humano para contaminar células y destruir al oponente. Parte de este juego es la competencia contra un agente controlado por la computadora que busca también contaminar y vencer al jugador. Este trabajo se enfoca en la descripción de cómo se ha ido desarrollando el sistema que controla al agente autónomo, a través del diseño e implementación de una máquina de estados que manipula el flujo de ejecución y, además, permite que el agente utilice distintos comportamientos de dirección para la locomoción del jugador artificial. La combinación de máquinas de estados con comportamientos de dirección permite generar conductas complejas que brindan una sensación de competencia dentro del juego, que es el objetivo del desarrollo de esta inteligencia artificial, procurando siempre mantener un balance en la dificultad para evitar frustración por ser demasiado difícil o aburrimiento por ser demasiado sencilla.

Palabras clave: máquina de estados finita, comportamientos de dirección, agentes de juego, Raycast, RedCell.

Finite State Machine with Steering Behavior for RedCell Game Agent Manipulation

Abstract. RedCell is a videogame developed at the 2018 Global Game Jam by members of the Escuela de Artes Plásticas y Audiovisuales (ARPA) of the BUAP, in which a competition is carried out within the human body to contaminate cells and to destroy the opponent. Part of this game is the competition against a computer-controlled agent that seeks also to contaminate and to destroy the player. This work focuses on the description of how the system that controls the autonomous agent has been developed through the design and implementation of a state machine that manipulates the flow of execution and

that allows the agent to use different steering behaviors for the artificial player's locomotion. The combination of state machines with steering behaviors allows to generate complex behaviors that provide a sense of competition within the game, which is the objective of the development of this artificial intelligence, always trying to maintain a balance in the difficulty to avoid frustration because it is too difficult or boredom because it is too simple.

Keywords: finite state machine, steering behavior, game agents, Raycast, RedCell.

1. Introducción

1.1. Inteligencia Artificial en los videojuegos

La Inteligencia Artificial (IA) en los juegos ha estado presente desde juegos como Pacman, Asteroids y Space Invaders y esto es porque el uso de personajes no jugadores (NPC, por sus siglas en inglés), descritos en [5], brinda una mayor presencia dentro de la experiencia del juego. Los diseñadores de juegos deben tratar de proporcionar un balance dentro de la inteligencia de los NPC ya que el objetivo de la IA dentro de un videojuego no es sobrepasar al humano (jugador) sino brindar la ilusión de que no está solo dentro del mundo virtual, a través de compañeros, guías, enemigos, etcétera.

En este trabajo se usan agentes [1] de juego que percibirán su entorno a través de colisionadores circulares y raycasting, los cuales mandarán información para elegir sus movimientos y elegir el momento de aproximarse a otro jugador.

1.2. Máquinas de estados finitas para dirigir la IA

Las máquinas de estados finitas (FSM) son ampliamente utilizadas dentro del mundo de la IA de los videojuegos debido a que entre otras ventajas que tienen, es que son intuitivas, flexibles y fáciles de programar [2].

Esta parte del agente se encarga de escoger los objetivos o tareas a realizar, es decir que lleva a cabo la planeación de acciones.

La estructura que sigue cada estado de la FSM implementada consiste en las siguientes funciones, que son similares en [2] y [9]:

- Enter. Las acciones que realiza el NPC una sola vez al inicio del cambio de estado;
- Reason. Esta función verifica cada frame si alguna condición se cumple para que la FSM realice un cambio de estado;
- Act. Son las tareas del estado que se llevan a cabo durante el lapso que dure éste;
- Exit. Las tareas de finalización del estado. Se llama una sola vez cuando hay un cambio de estado.

1.3. Comportamientos de dirección para manipular el movimiento de un agente

Para controlar la locomoción de un NPC dentro del juego se usan comportamientos de dirección (Steering behavior) que se encargan de calcular las trayectorias de movimiento que debe seguir el agente para satisfacer los objetivos planteados en la parte de planeación realizada en la FSM.

Existen distintos comportamientos para buscar, seguir, huir, perseguir, esconderse, etc. Todos ellos calculan la trayectoria que debe seguir el agente dejando la locomoción, es decir los aspectos mecánicos del movimiento, a otro componente del programa [4].

Los comportamientos se enfocan sólo en su objetivo como el objeto a perseguir o el jugador a evadir, y en el movimiento que deben realizar para cumplir su tarea.

1.4. RedCell

RedCell es un juego de competencia en el cual varios jugadores deben contaminar células a través del contacto o usando un proyectil. En ese juego existen dos etapas:

1. La primera consiste en la contaminación de todas las células en el escenario. El juego inicia con cierto número de células que cada jugador debe tratar de contaminar a través de tocarlas o impactar con ellas con un proyectil.
2. Una vez que todas las células han sido contaminadas entonces la segunda etapa se activa y los jugadores pueden hacerse daño entre ellos. Al disparase si un proyectil enemigo impacta entonces el jugador desaparece y es en este momento que entra en juego la cantidad de células que contaminaron, pues el jugador impactado puede revivir en alguna de las células que le pertenecen si es que tiene. Una vez que el jugador revivió, la célula desaparece.

Este juego surgió en el Global Game Jam 2018, un evento de tipo maratón de 48 horas, donde se forman equipos y se crean videojuegos en ese lapso. Una vez terminado el evento se retomó el juego para su desarrollo completo a partir del prototipo mostrado en la Fig. 1.

Es en el desarrollo posterior del juego que se empezó el diseño y construcción de NPC para que una persona pueda competir contra una IA.

2. Diseño e implementación de la FSM

Las acciones principales que debe desempeñar el NPC en este juego están ilustradas en la Fig. 2 y a continuación se describen:

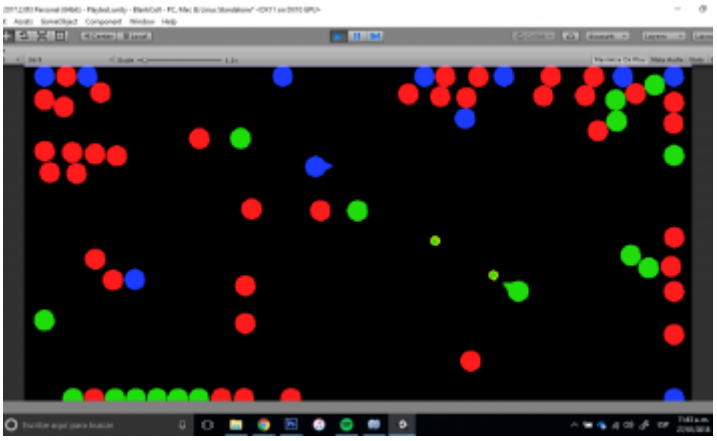

Fig. 1. Primer prototipo de RedCell en el Global Game Jam 2018. Era un juego para dos personas, aún no existía una IA.

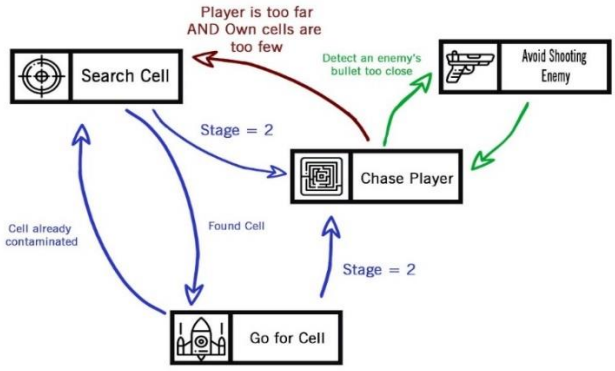

Fig. 2. Diagrama de la FSM que controla la toma de decisiones del NPC en el juego.

Buscar una célula. Durante la primera etapa del juego el agente deberá buscar células que contaminar, pues tener estas células significa tener vidas para participar en la segunda etapa. Esto se hace a través de una lista de objetos que almacena las células en el escenario.

Ir por una célula. El estado de Buscar solo se enfoca en localizar alguna célula para contaminar, pero en este estado es cuando se decide una vez ubicada la célula ir tras ella para contaminarla. El estado activará un comportamiento de persecución (Pursuit) que le permite al agente acercarse para contaminar a través de contacto o de un proyectil.

Perseguir al jugador. Cuando el juego entre en la segunda etapa el agente debe tomar como prioridad atacar al jugador para poder destruirlo.

3. Selección y construcción de los comportamientos de dirección

Para lograr que el agente realice las acciones que se planean en la FSM se usan comportamientos de dirección. En los estados de ir tras la célula y perseguir al jugador se utiliza el comportamiento de persecución.

3.1. Persecución

Este comportamiento busca calcular la posición donde se encontrará el objetivo para tratar de acercarse lo más posible a este. No sirve simplemente que el NPC se dirija a la posición actual de una célula o del jugador, pues estos tienen movimiento constantemente y jamás llegaría a alcanzar su destino, ilustrado en la Fig. 3.

Para poder calcular la posición donde se encontrará el objetivo (una célula o el jugador rival) se toma en cuenta la velocidad del agente y la del objeto perseguido y sus posiciones.

Primero calculamos un vector que nos indique la dirección desde el NPC hacia el objetivo, como se describe en [7, 8], usando la ecuación (1).

$$T = O - N, \tag{1}$$

donde T es el vector resultado de 2 o 3 dimensiones que apunta hacia el jugador o célula destino, O es la posición del objetivo y N es la posición del NPC en el escenario.

Una vez encontrada esta dirección podemos calcular el tiempo de anticipación a tomar en cuenta usando las velocidades de los objetos en cuestión.

$$L = ||T|| / (NV+OV), \tag{2}$$

donde L es el tiempo de anticipación que calculamos, NV es la velocidad del NPC y OV es la velocidad del objetivo. Este tiempo encontrado en la ecuación (2) de anticipación es proporcional a la distancia entre el objetivo y el perseguidor; y es inversamente proporcional a la suma de las velocidades de los agentes [3].

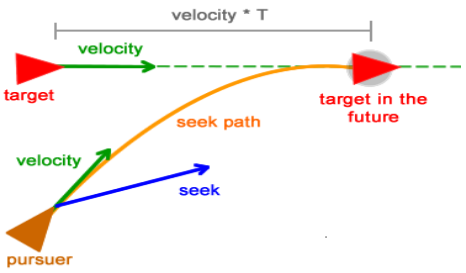

Fig. 3. Descripción gráfica del comportamiento de persecución [3]. En la imagen "pursuer" representa el NPC y "target" es una célula o un jugador rival.

Una vez calculado la cantidad de anticipación el nuevo destino del NPC se calcula como sigue.

$$S = O + OV * L, \tag{3}$$

donde S es la posición futura del objetivo a la cual el agente ahora debe dirigirse. La ecuación (3) se usa en la función completa; en pseudo código queda de la siguiente forma.

```
void Pursuit()
{
    Vector T = O - N;
    float L = T.magnitude / (NV + OV);
// Hay que buscar la posicion calculada del evasor
    Vector S = O + OV * L;
    Vector DesiredVelocity = (S - N).normalized * NV;
// antes de aplicar, multiplicar por el peso
    SteeringForce += (DesiredVelocity - NV) * pursuitWeight;
}
```

Lo anterior es una síntesis de la función de pursuit desarrollada para RedCell. Se omiten algunos detalles.

3.2. Evasión de paredes

Como el escenario de competencia está confinado a un área rectangular para que las células en el juego puedan quedarse dentro de dicha área y estar en movimiento, es común que el NPC al calcular el movimiento que tiene que hacer al dirigirse hacia algún objetivo termine muy cerca de una pared o incluso colisionando con alguna pues los comportamientos de dirección trabajan añadiendo fuerza a los agentes y en ocasiones esto los lleva a situaciones de colisión.

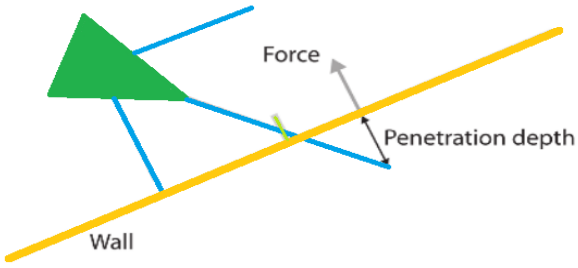

Fig. 4. En azul los sensores de colisión del agente. Para evadir paredes se toma en cuenta qué tan profundo los sensores entran en las paredes y el vector normal de dichas paredes para aplicar una fuerza que aleje al agente del choque. Imagen editada de [3].

Para evitar dichas colisiones se implementó un método de evasión, basado en la idea mostrada en la Fig. 4, de paredes que se encuentran activas todo el tiempo, el cual consiste en 3 pasos:

1. Creación de sensores. Utilizando una técnica de raycasting, se proyectan 3 vectores desde el agente. Uno hacia su frente (0 grados), otro a 45° y un último a -45°. Estos sensores trabajan como bigotes de un felino como se muestra en la Fig. 5, pues se usan para percibir si existe alguna pared cercana con la cual se pueda colisionar.

2. Examinación de cada sensor. Para cada uno de los 3 sensores se verifica si entraron en colisión con algún objeto etiquetado como pared. Si es así entonces se selecciona aquel cuya colisión sea más próxima a ocurrir, o sea que el punto de colisión del sensor sea el más cercano al agente.
3. Cálculo de fuerza. Una vez identificado el punto de colisión más próximo se procede a calcular una fuerza que lo aleje de la pared, esto a partir de que tanto el sensor penetró en la pared y el vector normal de dicha pared:

$$OS = N - C, \tag{4}$$

donde OS es la cantidad que el sensor penetró en la pared. Esta información es importante pues mientras más cercano esté el punto de colisión quiere decir que se debe aplicar una fuerza más grande para alejar al agente lo antes posible de un choque con la pared. N es la posición del NPC y C es el punto de colisión del sensor con la pared.

Después de aplicar la ecuación (4) se procede a crear una fuerza en la dirección de la normal de la pared, con la magnitud de la cantidad de penetración encontrada:

$$F = W * \|OS\|, \tag{5}$$

donde F es el vector de fuerza aplicado al agente para alejarse de la pared, W es el vector normal de la pared y la ecuación (5) se usa para la función de evadir la pared.

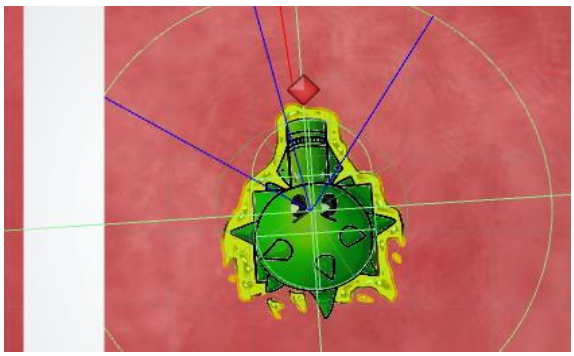

Fig. 5. Captura de RedCell en depuración. En azul, los sensores construidos con raycasting del agente. Estos sensores detectan posibles colisiones con paredes del escenario.

3.3. Evasión de proyectiles

Para darle oportunidad al NPC de competir contra un jugador humano, se diseñó y construyó un comportamiento que le da la oportunidad al agente de esquivar un proyectil lanzado por el rival.

Este comportamiento, que se bautizó como Evasión de proyectiles, tiene la siguiente lógica.

— Se crea un sensor de proyectiles circular alrededor del agente que registra cuando un proyectil del rival se acerca. Al detectar dicho proyectil se agrega a una lista de proyectiles aproximándose al NPC.
— Se busca cuál es el proyectil más cercano al NPC.

— Calculamos una fuerza de dirección que nos aleje del proyectil. Esta fuerza depende de la cercanía del proyectil, ya que mientras más cerca se encuentre, con mayor rapidez debe reaccionar para poder intentar esquivar el disparo.

En el siguiente pseudocódigo se presenta el funcionamiento de este comportamiento:

```
void BulletAvoidance()
{
  //if we have found an intersecting obstacle
  if (EnemyCloseBullets.Count > 0)
  {
    ClosestIntersectingObstacle = null;
    float DistToClosestIP = float.MaxValue;

    foreach (GameObject bullet in EnemyCloseBullets)
    {
      // look for the closest bullet
      float dist =
        Distance(transform.position,
                 bullet.transform.position);
      if (dist < DistToClosestIP)
      {
        ClosestIntersectingObstacle = bullet;
        DistToClosestIP = dist;
      }
    }

    // calculate a steeringforce away from obstacle
    Vector SteeringForceLocal = Vector.zero;
    LocalPosOfClosestObstacle =
        LocalPos(ClosestIntersectingObstacle);
    float ClosestIntersectingObstacleRadius =
        ClosestIntersectingObstacle.radius;

    if (ClosestIntersectingObstacle)
    {

      float FeelerLength = bulletFeelerRadius * 2;
      float multiplier = 1.0f +
        (FeelerLength - LocalPosOfClosestObstacle.X) /
        FeelerLength;

      //calculate the lateral force
      SteeringForceLocal.y =
        (ClosestIntersectingObstacleRadius -
ClosestIntersectingObstacle.y) * multiplier;

    }

    SteeringForce += SteeringForceLocal *
bulletAvoidanceWeight;
  }
}
```

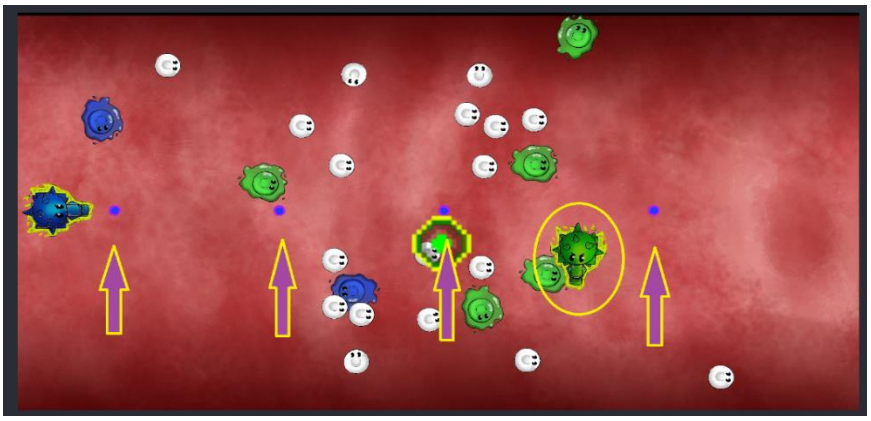

Fig. 6. NPC (verde) en movimiento tratando de esquivar los disparos del rival en azul. El jugador azul disparando proyectiles en dirección del NPC.

Este comportamiento se debe mantener activo todo el tiempo sin importar el estado en que se encuentre el NPC, pues en todo momento el agente debe estar pendiente de los disparos que haga el rival y poder tratar de reaccionar a tiempo para evitar su destrucción. En la Fig. 6 se ilustra el movimiento del agente para esquivar el ataque del rival.

3.4. Uso de raycasting para disparar

Una función importante de los jugadores es la habilidad de disparo, pues con esta es con la que se usan los proyectiles para eliminar al rival y contaminar células. Para que el agente puede disparar en la FSM se establece un estado para buscar al jugador y otro para buscar células, pero en estos estados sólo se manipula el movimiento de la IA, no se manda a disparar.

La función de disparo del NPC está activa en todo momento del juego como el comportamiento de evasión. Cuando el agente se dirige hacia su objetivo se va proyectando un rayo (raycasting) hacia el frente que ayuda a determinar si este se encuentra frente al NPC.

El raycast identifica que objetos toca y discrimina para sólo mandar a llamar la función de disparar cuando una célula se atraviese en su camino y también cuando un jugador rival lo haga durante la segunda etapa. El rayo se ilustra en la Fig. 7.

Para optimizar la función de disparo que generaba instancias de proyectiles se usó un patrón de programación llamado “Object pooling”, el cual permite crear solo las instancias necesarias de un objeto y evita estar creando y destruyendo objetos en tiempo de ejecución [6].

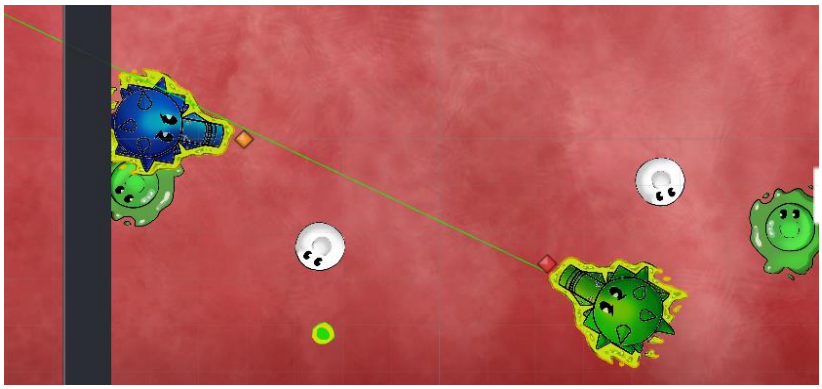

Fig. 7. Con una línea verde se aprecia el rayo (raycast) que detecta que objetos se encuentran delante del NPC. Este rayo solo se muestra para propósitos de depuración, no aparece en el juego final.

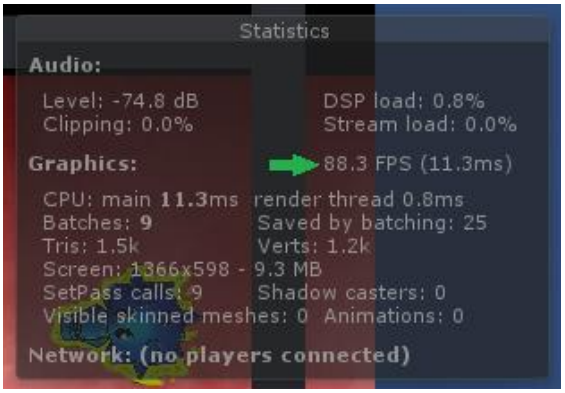

Fig. 8. La cantidad de cuadros por segundo en un juego de alta gamma se espera que sea de 60 para que el jugador no tenga problemas de interacción y sienta las animaciones del juego fluidas. En el caso de RedCell se cumple este número sin problemas.

4. Experimentación y resultados

Para poder probar el juego se realizaron partidas con jugadores humanos contra el NPC que se desarrolló. Vale la pena recordar que, para el caso de videojuegos, no es objetivo que la IA sea mejor que las personas, sino que sea retadora y genere una sensación de diversión y la necesidad de volver a jugar, es decir generar competencia.

El uso de IA no afecta el rendimiento del juego. Algunas estadísticas en la Fig. 8 muestran que el funcionamiento y "frame rate" son adecuados.

En cuanto a la experiencia de juego y el desempeño de la IA se realizaron pruebas de juego de humanos contra el NPC. En los siguientes gráficos se muestran los resultados obtenidos.

Tabla 1. ¿Cuántas veces ganaron los jugadores y cuántas el NPC?

Ganador	Porcentaje de usuarios
Jugador	31.7%
CPU	65.9%
Empate	2.4%

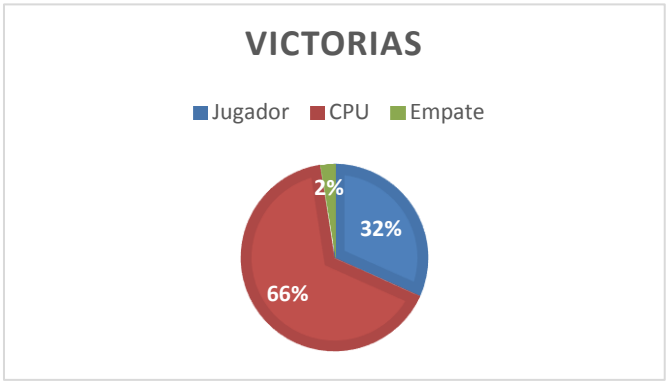

Fig. 9. Victorias de humanos contra CPU. El agente logró vencer en varias ocasiones a los jugadores que hicieron pruebas.

Tabla 2. ¿Qué tan difícil le pareció al jugador la IA?

Dificultad	Porcentaje de usuarios
Muy fácil	0.0%
Fácil	4.9%
Medio	70.7%
Difícil	24.4%
Muy difícil	0.0%

Tabla 3. Se les pidió a los jugadores que describieran en una palabra el desempeño general de la IA.

Descripción del NPC	Porcentaje de usuarios
Divertido	29.3%
Frustrante	7.3%
Retador	39.0%
Competitivo	24.4%
Aburrido	0.0%

Aunque muchas veces los jugadores fueron derrotados por la IA como se aprecia en la Tabla 1, los resultados muestran que las personas quedaban entusiasmadas y con ganas de volver a jugar, ilustrado en las Figs. 9, 10 y 11. Consideran que la IA no es imposible de vencer y a pesar de perder contra ella, tenían ganas de volver a competir.

Algo importante que se aprecia en la Fig. 11 es que el elemento de frustración que se busca evitar en videojuegos fue mínimo, e indica que el diseño del juego va por buen camino como se ilustra en las Tablas 2 y 3.

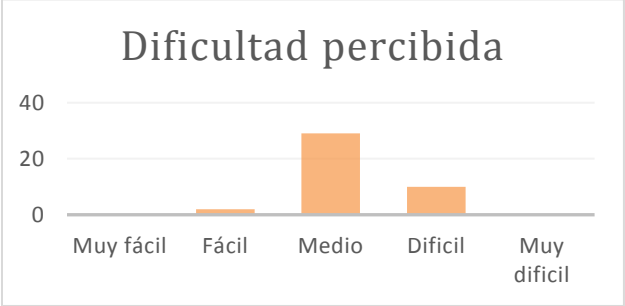

Fig. 10. Que tan difícil es vencer al CPU. Los jugadores describieron que la dificultad de la IA era entre media y difícil sin llegar a ser imposible de vencer.

5. Conclusiones y trabajo actual

Basado en los resultados obtenidos se ha generado una IA competitiva que permite a los jugadores disfrutar de la experiencia y querer volver a jugar, aunque sin duda para las personas el momento en el que se genera mayor presencia en el juego es cuando se compite contra otra persona.

No se puede reemplazar la socialización que existe y es parte de la naturaleza de la competencia humana, sin embargo, el NPC creado es suficiente y cumple su propósito.

¿Cómo alterar la dificultad del NPC? Actualmente se está trabajando en este aspecto. Ideas como aumentar la velocidad de movimiento y reacción de la IA han surgido, pero no son viables porque los jugadores empiezan a sentir frustración al ver que la computadora tiene reacciones más rápidas que las de un humano. Sienten que el NPC está haciendo trampa.

Fig. 11. Gráfica de radar mostrando la descripción que los jugadores le daban al CPU.

La solución que se está explorando es reemplazar el sistema de toma de decisiones actual (FSM) por un sistema de Planeación de Acciones Basadas en Metas (GOAP),

que elimina la restricción que tienen las máquinas de estado de que su comportamiento está limitado y puede ser predicho con mayor facilidad.

También se está trabajando para integrar potenciadores (power-ups) al juego que lo hacen más interesante y desafiante. La IA deberá tomar en cuenta los tipos de potenciadores, cuando tomar alguno, y cuando usarlo. Diseñar al NPC con estas circunstancias presenta un reto interesante.

Referencias

1. Stuart, J., Peter, N.: Artificial Intelligence: A Modern Approach. 3rd ed. Prentice Hall, United States of America (2010)
2. Mat, B.: Programming game AI by example. 1st ed. Jones & Bartlett Learning, United States (2004)
3. Bevilacqua, F.: Understanding Steering Behaviors: Pursuit and Evade (2018)
4. Craig, R.: Steering Behaviors for Autonomous Characters. In: Proceedings of Game Developers Conference 1999, pp. 763–782. Miller Freeman Game Group, San Francisco California (1999)
5. Heri, A., Agung, H.: Kuspriyanto: Design and Implementation of Zoopedia: Behaviour of Non Playable Character (NPC) of Tiger Hunting the Prey. Procedia - Social and Behavioral Sciences 67, pp. 196–202 (2012)
6. Robert, N.: Game Programming Patterns. 1st ed. Genever Benning, United States (2014)
7. Eric, L.: Mathematics for 3D Game Programming and Computer Graphics. 3rd ed. Cengage Learning Ptr, United States (2011)
8. Fletcher, D., Ian, P.: 3D Math Primer for Graphics and Game Development. 2nd ed. A K Peters Ltd, United States (2011)
9. Ray, B., Aung, S., Thet, N.: Unity 2017 Game AI Programming. 3rd ed. Packt Publishing, United States (2018)