Un modelo integrador de máquinas de estados y árboles de comportamiento para videojuegos

Ismael Sagredo-Olivenza,
Marco Antonio Gómez-Martín and Pedro A. González-Calero

Dep. Ingeniería del Software e Inteligencia Artificial
Universidad Complutense de Madrid, Spain
email: isagredo@ucm.es, {marcoa,pedro}@fdi.ucm.es

Abstract. La programación del comportamiento de los personajes de un videojuego, su "inteligencia artificial", es un trabajo en el que colaboran programadores y diseñadores. Aunque idealmente la separación del trabajo de ambos debería estar bien definida, en la práctica existe una separación borrosa entre la parte del comportamiento que es responsabilidad de unos y otros.
Las máquinas de estados son típicamente artefactos que usan los diseñadores para especificar su parte del comportamiento mientras que los programadores se sienten más cómodos con los más expresivos árboles de comportamiento. Para facilitar la colaboración y permitir distintas fronteras entre unos y otros, en este trabajo presentamos un framework que permite integrar máquinas de estados y árboles de comportamiento.

1 Introducción

Tres son los roles fundamentales involucrados en el desarrollo profesional de videojuegos: diseñador, grafista y programador. Los programadores se encargan de construir los diferentes sistemas que integran el videojuego, más allá de los que ya proporcione el motor que se pueda estar utilizando; los grafistas producen los modelos, texturas y animaciones que constituyen el aspecto visual del videojuego; y los diseñadores además de concebir las mecánicas del juego, "a qué se juega", utilizan los bloques de construcción que les proporcionan programadores y grafistas para diseñar los niveles que componen el juego.

Por lo que se refiere al comportamiento de los personajes (NPCs, por las siglas en inglés de *Non-player characters*), la "inteligencia artificial", el trabajo requiere la colaboración estrecha de diseñadores y programadores. Aunque los grafistas también tienen un papel fundamental en esta tarea, ya que en muchas ocasiones lo que más contribuye a transmitir al jugador la percepción de que un NPC se comporta de forma inteligente es su aspecto y las diferentes animaciones que ejecuta, es habitual definir una capa de abstracción que abstrae de las animaciones concretas que se asocian a cada comportamiento de un personaje, de forma que un grafista no tiene que interactuar mucho con sus compañeros para realizar su trabajo.

Programadores de inteligencia artificial (IA) y diseñadores por contra sí deben colaborar estrechamente. En esencia los programadores proporcionan las acciones y los elementos de la percepción de los personajes, parametrizadas de forma que los diseñadores puedan configurarlas utilizando scripts sencillos y máquinas de estados. La dificultad radica en el caracter fundamentalmente explorativo del diseño, lo que lleva a un proceso iterativo donde los diseñadores exploran diferentes alternativas con las herramientas de que disponen, y cuando la solución se vuelve imposible o demasiado compleja piden la ayuda de sus compañeros programadores para generar nuevas acciones o elementos de percepción.

Los árboles de comportamiento (BTs, por las siglas en inglés de *Behavior trees*) son la tecnología utilizada hoy en día en la industria para implementar comportamientos complejos en los personajes. Los BTs también se apoyan en acciones y elementos de percepción básicos programados directamente en el lenguaje del motor (con frecuencia C++), y permiten combinarlos para generar comportamientos complejos. Los BTs se pueden ver como una evolución de las máquinas de estados, aunque su expresividad es equivalente a la de un lenguaje de programación de propósito general, lo que a menudo los convierte en un arma demasiado peligrosa en las manos inexpertas de un diseñador. Por otra parte, los BTs se pueden construir utilizando lenguajes visuales, y si su uso se restringe a especificar aspectos concretos del juego pueden ser más adecuados para los diseñadores que las propias máquinas de estados, que fácilmente pueden caer en situaciones de explosión combinatoria. Sin ir más lejos, en Halo 2 [5,6], el juego que popularizo el uso de BTs, estos se usaban tanto por programadores como por diseñadores, la mayoría de los cuales, dicho sea de paso, también sabían programar.

En resumen, la separación entre las tareas de diseñadores y programadores es a menudo borrosa, como los es decidir qué herramientas son las más adecuadas en cada momento, dependiendo de las habilidades de las personas involucradas o incluso de sus preferencias. Es por ello que hemos desarrollado un framework, Behavior Bricks, para programar la IA de los personajes de un videojuego que permite combinar libremente BTs y máquinas de estados, utilizando el mismo conjunto de acciones básicas y elementos de percepción. Esta integración plantea problemas derivados de los distintos modelos de ejecución que se utilizan en uno y otro caso, y en este artículo presentamos el modelo que hemos desarrollado para unificar ambos modelos. Esta integración es tal que Behavior Bricks permite la creación de comportamientos jerárquicos donde nodos de FSMs son BTs y viceversa.

En el siguiente apartado se describen las ideas fundamentales de máquinas de estados y BTs. En el apartado 3 se detalla el mecanismo que permite unificar las acciones básicas en BTs y FSMs, mientras que en el apartado 4 nos ocupamos de las compuestas. En el apartado 5 nos ocupamos del modelo de percepción unificado y finalmente en el apartado 6 describimos la integración de Behavior Bricks en Unity. La última sección presenta las conclusiones y líneas de trabajo futuro.

2 Construcción de comportamientos en videojuegos

A continuación describimos las dos tecnologías que integra Behavior Bricks para representar el comportamiento de los personajes de un videojuego: máquinas de estados y árboles de comportamiento:

2.1 Máquinas de estados finitos

Las máquinas de estados finitos [1] son un modelo computacional que realiza cómputos automáticamente sobre una entrada para producir una salida. Están fundamentadas en la teoría de autómatas y han sido ampliamente estudiadas y formalizadas.

Para el desarrollo de videojuegos, donde históricamente han sido una de las técnicas más utilizadas para implementar comportamientos, se ven como un conjunto de estados en los que puede encontrarse la entidad y una serie de *transiciones* o condiciones de cambio de estado. Eso permite una representación visual simple e intuitiva en forma de grafo donde los nodos son los estados y las transiciones son las aristas.

Cada nodo/estado se describe mediante la acción o acciones primitivas que la entidad ejecutará cuando se encuentre en ese estado. Cada una de las aristas/transiciones son una descripción de la condición del mundo que *dispara* esa transición.

La ventaja de las máquinas de estados frente a otros mecanismos de creación de comportamientos es su simplicidad, que no requieren conocimientos de programación para entenderlas y crearlas, su determinismo y rapidez de ejecución. Además, como tienen una representación visual simple, se pueden construir editores gráficos.

El tradicional problema de las máquinas de estado estriba en que estas no escalan demasiado bien cuando el número de transiciones es muy grande, ya que comienza a incrementar su número rápidamente cuando la complejidad del problema comienza a crecer, por lo que rápidamente comienza a ser difíciles de mantener, de seguir y controlar por parte del diseñador.

El modelo de las máquinas de estados permite definir estados especiales considerados como *estados finales*. Aunque en juegos no es habitual su uso, Behavior Bricks permite indicarlos para dar por terminado el comportamiento. Como veremos, esto nos permitirá integrarlos fácilmente con la otra técnica que describimos en el apartado siguiente, los árboles de comportamiento.

2.2 Behaviour Trees

Los árboles de comportamiento (Behaviour Trees o BTs) [3] son utilizados en videojuegos para modelar la inteligencia artificial de los NPCs de una forma similar a las máquinas de estado. La diferencia principal radica en que los BTs eliminan las transiciones de las máquinas de estado, lo que permite subsanar los problemas derivados del crecimiento del número de transiciones típicos de las FSMs. De esta forma, los árboles de comportamiento definen un mecanismo

para decidir cuál es la siguiente acción a ejecutar. Para conseguirlo, incorporan información adicional de control a las acciones que determine cuál de ellas debe ser ejecutada en un momento dado. Esa información de control es modelada normalmente mediante nodos intermedios que toman decisiones sobre el orden en el que deben ejecutarse las acciones primitivas. Estas decisiones se toman utilizando los datos del entorno mediante nodos especiales que los chequean y utilizando el valor de retorno de las acciones. Y eso porque, al contrario que en una FSM, cuando la ejecución de un nodo de un BT termina éste notifica a su nodo padre si la ejecución de la acción tuvo éxito (SUCCESS) o no (textscFailure). Con esa información de finalización y la información de entorno, los nodos intermedios toman decisiones sobre cuál es la siguiente acción a ejecutar.

Los nodos de los BTs se distinguen entre los nodos *hoja* que contienen las acciones primitivas que alteran el entorno y los nodos internos, o nodos de decisión, con los que se montan los comportamientos complejos agrupando acciones primitivas. Éstos últimos se agrupan en diferentes tipos según sus reglas de funcionamiento y existen distintas colecciones de tipos de nodos que proporcionan expresivada equivalente. Algunos de los más utilizados:

- Secuencias: almacenan una lista de comportamientos que ejecuta en el orden que se han definido. Cuando el comportamiento que está ejecutándose actualmente termina con éxito, se ejecuta el siguiente. Si termina con fallo, él también lo hace. Si todos se han ejecutado con éxito, el nodo devuelve SUCCESS.
- Selector: tiene una lista de comportamientos hijo y los ejecuta en orden hasta encontrar uno que tiene éxito. Si no encuentra ninguno, termina con fallo. El orden de los hijos del selector proporciona el orden en el que se evalúan los comportamientos.
- Parallel: el nodo parallel ejecuta a la vez todos los comportamientos hijo del nodo. No necesariamente esta ejecución debe ser concurrente, puede ser uno detrás de otro, pero dentro de la misma iteración en el bucle de juego. Los nodos parallels pueden tener diferentes políticas de finalización. Pueden finalizar como una secuencia, es decir en el momento que un nodo falle todo el nodo parallel falla, o como un selector, hasta que todos los nodos han fallado no falla el nodo.
- Selector con prioridad: los nodos selector llevan incorporada una cierta prioridad en el orden de ejecución, pero no re-evaluan nodos que ya han terminado. El Priority Selector se diferencia del priority normal en que las acciones con más prioridad siempre intentan ejecutarse primero en cada iteración. Si una acción con mayor prioridad que otra que ya se estaba ejecutando se ejecuta, se interrumpe la acción que se estaba ejecutando anteriormente.
- Decorators: Los decoradores son nodos especiales que sólo tienen un hijo y que permiten modificar el comportamiento o el resultado de ese hijo, inclyendo, por ejemplo, guardas que controlan si el nodo se ejecutará o iteradores que permiten especificar un número de ejecuciones.

Si todo el BT finaliza se pueden dar el comportamiento por terminado o bien reiniciar el comportamiento. Una acción en concreto de un BT puede ser un BT en sí mismo, lo que permite modularizar los comportamientos para poder ser reutilizados en el futuro.

Tanto las máquinas de estado como los BTs utilizan para comunicarse entre ellos un espacio de memoria común que deben gestionar. Este espacio de memoria se denomina *pizarra* (en inglés *blackboard*) y permite compartir información entre las distintas acciones.

3 Acciones en Behavior Bricks

Como comentamos anteriormente, para permitir la combinación de comportamientos independientemente del modo de implementarlos, en Behavior Bricks tanto las acciones primitivas como las FSMs y los BTs son consideradas *acciones*. De esta manera se podrá utilizar indistintamente cualquiera de ellas. Esto automáticamente hace que en los nodos de un BT o en los estados de una FSM pueda colocarse otro BT o FSM, consiguiendo comportamientos jerárquicos. Es por esto que cuando hablemos de acciones debemos entender *comportamientos*, pues estas acciones pueden tener detrás complejas FSMs jerárquicas o inmensos BTs.

Las acciones se definen mediante un nombre (único en toda la colección) y una lista (opcional) de parámetros de entrada y de salida. Los parámetros de entrada son los datos que el comportamiento necesita para funcionar, mientras que los de salida son los resultados de su ejecución. Ambos permiten reutilizar los comportamientos creados en contextos distintos y enlazar unos con otros de forma flexible.

Desde el punto de vista de esa ejecución, cuando el ejecutor del comportamiento lanza la acción, se entiende que esta está en el estado RUNNING y en algún momento terminará ya sea de forma satisfactoria (SUCCESS) o con fallo (FAILURE). Las casusas del fallo pueden ser diversas, pero valga como ejemplo la acción que hace que el avatar se mueva a un punto concreto del mapa, cuando no existe ruta posible para llegar.

Las herramientas de edición de FSMs y BTs incorporan el soporte adecuado para el manejo transparente de acciones independientemente de su naturaleza. Para eso:

- En los estados de las FSMs y en los nodos hoja de los BTs se permite colocar no sólo acciones primitivas sino también otros FSMs o BTs que se ejecutarán de forma jerárquica. Ese comportamiento/acción de más bajo nivel terminará con éxito o fracaso que pasará a convertirse en el resultado del propio nodo donde se encuentra (en el caso de un BT) o podrá disparar una transición (en el caso de una FSM).
- Cuando un BT o FSM es añadido a otro comportamiento, el editor permite especificar los parámetros de entrada de esos comportamientos, de igual forma que lo haría con las acciones primitivas o dejarlos como *variables libres*.

– Los BTs y FSMs creados son almacenados y tratados como *acciones*. El editor durante la construcción del comportamiento comprueba si hay alguna variable libre y las que encuentra pasan a ser parámetros de entrada de ese comportamiento. Algo similar ocurre con los parámetros de salida; cualquier salida de las acciones de los nodos/estados son candidatas a convertirse en parámetros de salida del comportamiento en construcción.

Behavior Bricks viene con un pequeño conjunto de acciones independientes de la plataforma, y el módulo de integración con Unity 3D proporciona un buen número de acciones primitivas que permiten interactuar con gran parte de la funcionalidad de Unity, como cambiar animaciones, distintos tipos de movimiento, reproducción de sonidos, etc.

No obstante los usuarios programadores de Behavior Bricks pueden extenderlo añadiendo nuevas acciones primitivas. Para eso Behavior Bricks proporciona un conjunto de clases de soporte siendo `UnityAction` la más importante. Las nuevas acciones heredarán de esta clase y podrán sobreescribir una serie de métodos para implementar la acción que explicaremos más adelante con más detenimiento.

4 Comportamientos compuestos en Behavior Bricks: FSMs y BTs

La forma de definir en Behavior Bricks comportamientos compuestos es, como ya se ha dicho, a través de la definición de máquinas de estados y/o árboles de comportamiento, los mecanismos líderes en el campo de los videojuegos.

Uno de los objetivos de diseño de Behavior Bricks era conseguir que la integración de ambos modelos fuera fácil e intuitiva, además de extensible. Eso ha obligado a *limar* las diferencias que hay entre ellos, de forma que las extensiones personalizadas que los usuarios de Behavior Bricks hagan a éste (añadiendo nuevas acciones primitivas ya descritas en el apartado 3 o extendiendo la parte de percepción de la sección 5) se integren automáticamente en ambos modelos.

4.1 FSMs en Behavior Bricks

El modelo habitual de una FSM está basado en la definición de unos estados en los que se ejecuta una acción determinada hasta que se hace cierta una condición que haga saltar una transición a otro estado. Esto significa que, normalmente, la acción incluída en el nodo se ejecuta indefinidamente hasta que cambia de estado.

Ese funcionamiento encaja muy bien en implementaciones de FSMs *cableadas* en código donde cada estado define un método o función que es invocada por el motor de ejecución siempre y cuando el estado siga activo. Sin embargo en Behavior Bricks toda la ejecución la realizan o bien las acciones primitivas o bien otras FSMs o BTs, es decir en cada estado colocamos un *subcomportamiento*/acción (descritas en la sección 3).

Eso implica dos cosas:

– Las acciones contenidas en los nodos *pueden terminar*. Al tratarse de comportamientos de más bajo nivel, llegará un momento en que terminen, ya sea con éxito o fracaso. Esto lo utilizaremos para permitir al diseñador especificar transiciones que no dependerán de los cambios en el entorno sino generados por el propio nodo, es decir transiciones cuando ocurre el SUCCESS o el FAILURE de las acciones del nodo.
– La ejecución de esos comportamientos de bajo nivel pueden ser *abortadas* desde fuera (desde el ejecutor de la FSM) si se activa una transición. Esto ya lo contemplamos en su momento permitiendo a los programadores de acciones primitivas definir un método específico `OnAbort` para poder realizar acciones de última hora cuando su ejecución va a ser parada sin su "consentimiento" para que pueda liberar recursos, etc.

Dentro de cada estado se puede añadir una lista de acciones que serán ejecutadas cuando el estado se active. El diseñador puede elegir dos políticas de ejecución:

– Tipo de ejecución: permite indicar si las acciones son ejecutadas en secuencia o en paralelo.
– Política ante fallos: si el estado terminará con fallo en el momento en el que una de las acciones falle o sólo si todas las acciones fallan.

Otro punto de diferencia entre las FSMs habituales y las de Behavior Bricks es la relativa a la finalización. Las FSMs en juegos no suelen estar planteadas para poder terminar. Pero dado que en Behavior Bricks éstas pueden aparecer en lugar de las acciones primitivas, el editor permite al diseñador especificar distintos *estados terminales* tanto para finalizar con éxito como con fracaso. Si no se indica ninguno de estos estados, la FSM estará ejecutandose indefinidamente hasta que el comportamiento de nivel superior le "expropie" por transitar a un estado (o nodo de BT) distinto.

Por último, y también como consecuencia de la dualidad FSM-comportamiento, Behavior Bricks permite al diseñador indicar los parámetros de entrada de la FSM así como cuáles son los de salida. Los parámetros de entrada se asociarán directamente a parámetros de entrada de acciones que se hayan colocado en los estados y que se hayan quedado *libres*, mientras que los de salida serán un subconjunto de los parámetros de salida de esas mismas acciones.

4.2 BTs en Behavior Bricks

La integración de los BTs en Behavior Bricks es mucho más fácil, pues encajan perfectamente como acciones. Cada árbol independiente es visto como un comportamiento que puede terminar con éxito o fracaso, exactamente igual que cualquier otra acción primitiva.

En Behavior Bricks están implementados todos los tipos de nodos explicados en el apartado 2.2 y se han añadido, además, algunas acciones simples y guardas concretas adicionales.

Una diferencia significativa entre FSMs y BTs es el modo en el que se incorpora al comportamiento la reactividad, que depende de la percepción del estado del mundo (y cuya implementación detallamos en la sección 5). En el caso de las FSMs ésta reactividad viene determinada por los cambios de estado, y por lo tanto está codificada en las *transiciones* o aristas entre estados. Eso automáticamente hace que la gestión de la percepción esté *fuera* de la ejecución de las acciones propiamente dichas.

En los BTs, sin embargo, esto no ocurre así. La responsabilidad de la reactividad recae en los propios nodos: existen acciones cuya única misión es comprobar si se cumple o no una cierta condición y mantener su ejecución hasta que ésta se mantenga, terminando en FAILURE cuando ésto ocurra, por lo que la acción de ese nodo es la que está contínuamente comprobando la validez de la condición. Existen otros nodos que incorporan condiciones a modo de *guardas* que hacen que la acción del nodo siga ejecutándose siempre y cuando esa guarda no se haga falsa.

5 Percepción y pizarras

Hasta ahora hemos explicado cómo crear comportamientos en Behavior Bricks pero hemos dejado de lado como gestionamos la información que el NPC recoge del mundo para tomar las decisiones. La percepción es el sistema que se encarga de extraer la información del entorno de juego y proporcionársela al comportamiento para que este tome las decisiones. El ciclo de vida de la IA en Behavior Bricks sería el que se muestra en la figura 1.

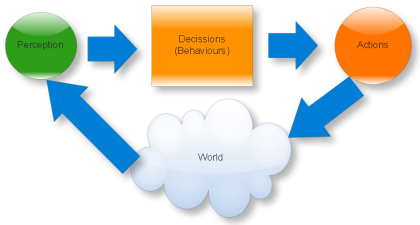

Fig. 1: AI Cicle

En la fase de percepción, el subsistema de percepción recoge la información del mundo, la procesa y se la proporciona al comportamiento para que este pueda decidir que acción debe ejecutar.

Cada una de los sub-comportamientos que conforman el comportamiento final, tienen una pizarra con la que se interconectan las entradas y salidas de sus acciones internas, lo que implica que podremos tener una jerarquía de pizarras en un comportamiento (recordemos que una acción puede estar implementada por una FSM o un BT).

Dichas pizarras tienen visibilidad hacía arriba en la jerarquía, es decir, una pizarra de un comportamiento de nivel inferior puede ver los valores de una pizarra de nivel superior, pero no a la inversa, ya que los datos de la pizarra de un comportamiento son parte de la implementación de dicho comportamiento y no deberían ser accesibles por el principio de encapsulación. Para comunicarse con el exterior, el comportamiento utiliza sus parámetros de entrada y salida, que serían su interfaz. Por tanto, la percepción no puede acceder a las pizarras internas de un subcomportamiento.

Así pues, el mapeo entre salidas de la percepción y el comportamiento, debe realizarse utilizando un pizarra global que maneja la entidad y que permite interconectar la percepción y otros componentes con el comportamiento.

Por lo tanto, para el resto de componentes de una entidad de juego, incluidos los componentes de la percepción, el comportamiento del NPC es una caja negra que sólo muestra sus parámetros de entrada y de salida. Éste sería el escenario ideal para construir comportamientos reutilizables, pero debido a que todas las pizarras inferiores de la jerarquía pueden ver la pizarra de su comportamiento padre y por tanto todos pueden ver la pizarra global, si un diseñador no necesita realizar un comportamiento reutilizable, puede simplificar el diseño del mismo y hacer que todos los datos estén en la pizarra global.

La percepción es realizada por un gestor de percepción que encapsula diferentes gestores que gestionan los diferentes tipos de sentidos que queremos simular. Estos managers permiten registrar *sensores y generadores de señales* [7] para intercomunicarlos. Los sensores reciben las señales emitidas por los emisores del tipo que estos establezcan. Por ejemplo un sensor de visión sólo aceptará señales visuales, un sensor sonoro sólo detectará señales acústicas, etc.

Cada uno de los managers sensoriales puede realizar un procesamiento en su interior de las señales basada en información del mundo que él conoce. Pero antes de entregar la señal le pregunta al sensor si este la puede percibir.

Una señal, dependiendo del tipo, puede tener diferentes atributos, pero al menos uno de ellos será la intensidad. Si la intensidad de la señal es suficiente para ser detectada por el sensor y este está en condiciones de detectarla (por ejemplo el sensor de la vista puede estar orientado hacia un punto donde no ve la señal visual que emite un personaje) esta señal se la entregará al sensor para que la procese.

El sensor avisa al resto de componentes que estén interesados que ha percibido una señal. A parte de esto, por defecto, el sensor tiene unos parámetros de salida donde muestra la información que ha obtenido de dicha señal. Estos parámetros de salida deben de ser mapeados en la pizarra global para nutrir de información al comportamiento del NPC. Con este mecanismo, independizamos la percepción de la ejecución del comportamiento y al tenerla modularizada, podemos sustituir los managers de percepción para poder implementar distintas simulaciones dependiendo de las características de nuestro juego, así como nuevos tipos de señales a tratar.

Por otro lado, las transiciones en las FSM o las condiciones en los BTs tampoco se definen en los propios comportamientos. En ellos solo se definen

el “mensaje” que los describe, como ocurría con las acciones primitivas. Estos mensajes pueden ser implementados por reglas que gestionará el ejecutor del comportamiento a nivel de la entidad. Cómo sucede con las acciones primitivas, si no se proporciona una implementación de una condición o un evento para una transición, el ejecutor informará del error.

Usando el mismo principio que usamos con las acciones primitivas, estas reglas forman parte de una colección de reglas que podemos reutilizar e reimplementar para ajustar la semántica a las particularidades del juego concreto, manteniendo la definición del comportamiento inalterado. El la figura 2 describimos de forma esquemática el papel de la pizarra como elemento de intercomunicación de las reglas que disparan eventos, la percepción y el propio comportamiento.

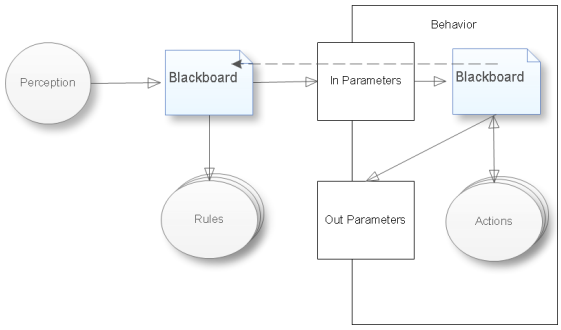

Fig. 2: Perception, Rules and Blackboards

6 Integración de Behavior Bricks en Unity

Behavior Bricks es middleware independiente de plataforma y puede ser portado a cualquier motor de juego. En concreto hemos adaptado Behavior Bricks a Unity 3D, que es uno de los motoros de juego más utilizados en la actualidad, sobre todo por estudios pequeños.

Unity utiliza como lenguaje de scripting una versión de Javascript modificada para la ocasión a la que denominan Unity Scripting, C# y un dialecto de Python denominado Boo.

Para Unity, un juego se divide en escenas. Cada escena está formada por un conjunto de entidades. Dichas entidades no son más que contenedores de *componentes* [9,8,2,4] que les dotan de funcionalidad. Todo en unity es un componente.

Los componentes se comunican entre sí mediante *paso de mensajes*. El paso de mensajes de una entidad a otra o de un componente al resto de componentes, permite desacoplar a los componentes unos de otros. De esta forma cuando envias un mensaje informando de un “evento interesante” para el resto de componentes, se espera que alguno de los componentes esté interesado en dicho

mensaje, pero podría no ser así. El componente simplemente "informa" de sucesos que han ocurrido, pero no asume a quien o quienes les pude interesar. Esto permite reutilizar componentes con mayor facilidad en diferentes entidades, ya que minimiza el acomplamiento entre ellos.

Para ejecutar nuestros comportamientos en Unity, se han creado en C# un componente que ejecuta las máquinas de estado y otro que ejecuta los árboles de comportamiento. A estos componentes los llamamos ejecutores de comportamiento y son los encargados de gestionar los parámetros de entrada del mismo y la pizarra global de la entidad.

Cada una de las acciones primitivas serán implementadas por un componente en Unity que deberá implementar la clase `UnityAction` y escuchar el "mensaje" que lo vincula a la acción primitiva. Para ello debe etiquetarse con dicho mensaje usando un atributo de C# como se muestra en el siguiente fragmento de código par la acción AttackAction.

```
[ActivableAttribute( "AttackAction" )]
public class AttackToBase : UnityAction {}
```

La clase `UnityAction` proporciona acceso tanto a los parámetros de entrada, como a los parámetros de salida y la pizarra del subcomportamiento que en ese momento la esté ejecutando. Además, disponen de acceso a la pizarra global y proporciona cuatro métodos que pueden ser sobrescritos:

- Init() que se llamará cuando se cree la acción.
- OnStart() Que se llamará una vez al principio por cada vez que se vuelva a ejecutarse el comportamiento.
- OnAbort() Que se llamará si el comportamiento ha sido interrumpido desde el ejecutor.
- OnEnd() Que se llamará si el comportamiento ha terminado con Success o Failure.
- OnUpdate() Que se ejecutará una vez por frame si está acción ha sido seleccionada por el comportamiento y que puede devolver Success o Failure si ha terminado.

La percepción es controlada por un varios gestores o *managers* donde se registran los sensores para ser informados de las señales. Estos sensores tambien son componentes de Unity y deben heredar de la clase `UnitySensor`. El sensor permite sobrescribir el método `bool CanDetect(signal)` que es llamado por el manager de percepción para asegurarse si este puede detectar una señal, una lista de parámetros de salida para almacenarlos en la pizarra global y el método `void Notify(signal)` que es invocado por el manager de percepción avisando de la existencia de una señal perceptible.

Por ultimo el diseñador deberá juntar las piezas para que el comportamiento funcione correctamente. Una vez seleccionado el comportamiento deseado, deberá asignar a cada acción primitiva la implementación que más se ajuste a sus necesidades de la colección de acciones en unity. Mismo procedimiento tendrá que hacer con las Reglas que lanzarán los eventos con los sensores que necesite. Como último paso de configuración, el diseñador utilizando la pizarra global,

pondrá en comunicación las entradas del coportamiento con los sensores y las reglas.

7 Conclusiones

Existen numerosos editores de comportamiento para Unity y para otros motores pero nuestro enfoque con Behavior Bricks está centrado en crear una herramienta que permita cubrir todas las necesidades de implementación de comportamientos para juegos ofreciendo una visión completamente unificada y modular de todo el proceso de ejecución de la IA, desde la percepción, la ejecución de las acciones, pasando por la fase de toma de decisiones. Behavior Bricks es un framework que unifica todas estas etapas y con la integración de un editor visual que estamos desarrollando, pretendemos además que sea fácil de usar por los diseñadores y que puedan colaborar con los programadores fácilmente para desarrollar comportamientos de calidad de forma rápida y fácil. El enfoque hacia la reutilización pensamos que ayudará a que los tiempos de desarrollo de la IA sean más cortos. La comunicación programador / diseñador es esencial para agilizar el proceso de desarrollo de un videojuegos y este framework junto con una herramienta visual que simplifique la autoría para el diseñador, pensamos que permitirá una buena coordinación entre ambos.

Las decisiones tomadas para unificar el concepto de acción primitiva para un BT o una FSM y cómo conectarlo con la percepción hacen de Behavior Bricks una herramienta tremendamente modular que permite modificar cada una de las partes con muy poco acoplamiento entre ellas, permitiendo adaptarlas e intercambiarlas fácilmente a diferentes proyectos y generos de juegos.

References

1. D. M. Bourg and G. Seemann. *AI for Game Developers*. O'Reilly Media, Inc., 2004.
2. W. Buchanan. *Game Programming Gems 5*, chapter A Generic Component Library. Charles River Media, 2005.
3. A. J. Champandard. *Getting Started with Decision Making and Control Systems*, volume 4 of *AI Game Programming Wisdom*, chapter 3.4, pages 257–264. Course Technology, 2008.
4. S. Garcés. *AI Game Programming Wisdom III*, chapter Flexible Object-Composition Architecture. Charles River Media, 2006.
5. D. Isla. Handling complexity in the Halo 2 ai. In *Game Developers Conference*, 2005.
6. D. Isla. Halo 3 - building a better battle. In *Game Developers Conference*, 2008.
7. I. Millington and J. Funge. *Artificial Intelligence for Games*. Morgan Kaufmann, second edition, 2009.
8. B. Rene. *Game Programming Gems 5*, chapter Component Based Object Management. Charles River Media, 2005.
9. M. West. Evolve your hierarchy. *Game Developer*, 13(3):51–54, Mar. 2006.