Este proyecto no podría haberse completado sin la colaboración de José Luis Álvarez y , especialmente, José María González, a los que desde aquí queremos agradecer su ayuda.

Índice

1. Introducción

1.1. ¿De qué trata el juego?

En este proyecto hemos querido plasmar el comportamiento de un rebaño de ovejas ante el movimiento del pastor y el entorno que las rodea. Nuestra finalidad es hacer que la simulación del comportamiento de estos animales en el videojuego sea lo más cercana posible a los hábitos y reacciones de aquellos.

Tan importante es el comportamiento del ganado como del pastor. Es por eso que, además, se realizaron investigaciones sobre las técnicas y las estrategias de pastoreo. El estudio de estos comportamientos y las investigaciones realizadas al respecto permiten determinar que al ganado se le puede agrupar, induciendo su comportamiento natural de permanecer unidos. Para ello hay varias técnicas:

- Técnica del limpiaparabrisas: El pastor debe moverse en zigzag de un lado a otro de la manada para mantener la línea recta de avance. Ver figura 1 en página 31
- Moverlos por el apretadero : Los animales necesitan tener el suficiente espacio para moverse adecuadamente.
- El pastor debe de tener movimientos lentos y no debe dar vueltas alrededor de los animales.
- Zona de fuga: La zona de fuga de un animal es su zona de seguridad. Los operarios deben mantenerse en el límite de esta zona. Figura 2, página 32
- Movimiento del pastor para que el ganado siga su camino en una manga con laterales: Para obligar al animal a desplazarse hacia adelante, el pastor debe estar por detrás del punto de equilibrio a la altura de los cuartos delanteros.
- Sacar al ganado del corral con un solo controlador: Los movimientos del pastor deben de ser perpendiculares a los del ganado, moviéndose hacia atrás y hacia delante sobre la barra transversal de una gigante T. Figura 3, página 32.
- Sacar al ganado del corral con dos controladores: Cada pastor se moverá hacia detrás y hacia delante en la línea transversal imaginaria que traza una T. Figura 4, página 33.

1.1.1. Comportamientos del Ganado Ovino

Los animales siguen al líder. Si éstos se despistan y se amontonan, el pastor debe concentrarse en mover a los líderes en lugar de empujar al grupo de animales de la parte trasera. Al mover al ganado en un espacio abierto, los animales se mueven en una forma tranquila y ordenada.

Para acelerar el paso de los animales, los pastores penetran la zona de fuga colectiva y se retiran. El campo de visual de las ovejas puede ser de 191 hasta 309 grados, dependiendo de su cantidad de lana.

Los animales de pastoreo tienen un sistema visual que proporciona una excelente visión de lejos, pero los músculos de los ojos relativamente débiles les inhiben de la capacidad de centrarse rápidamente en los objetos cercanos.

La implementación de estos comportamientos en este proyecto implica un lenguaje de programación capaz de reproducir lo explicado anteriormente. De los lenguajes disponibles para realizar aplicaciones, ***lluviaProject*** era el único que disponía de caracteres personales autónomos, los cuales permiten modelar su comportamiento para adaptarse al de un rebaño de ovejas.

2. Por qué Lluvia Project

2.1. ¿Qué es lluviaProject?

Es una API Open Source de Javascript que incorpora parte de las funciones nativas de Ruby. Soporta multihilos a pequeña escala y provee un sistema de mensajes. Las aplicaciones deben crearse dentro del directorio Vendor.

La aplicación debe de contar con un archivo generalmente llamado Dependencies, donde deben incluirse el nombre de los ficheros de los que depende la aplicación. Si existe una función main esta se llama automáticamente después de cargarse ***lluviaProject*** .

Ver ejemplo en código 1 en página 41

2.2. Ventajas

Una de las funcionalidades que ofrece ***lluviaProject*** es el objeto Boid. A este objeto se le puede configurar tanto el comportamiento como las características físicas de aquello que se quiere representar.

En la parte física, se puede modelar tanto el peso, la visión, la velocidad, la posición, la aceleración, capacidad de frenada, la capacidad de giro, etc. Y por otra parte, se pueden modelar comportamientos de huida, de persecución, de cohesión, de alineamiento, de separación e incluso comportamientos hechos a medida. Esta característica de ***lluviaProject*** permite representar los comportamientos de animales, objetos o partículas, entre otros.

Esto, junto con el hecho de que añade librerías y funciones que hacen que la escritura de código en Javascript sea más cómodo, nos hizo decidirnos por ***lluviaProject*** .

3. Parte gráfica

3.1. Front-end (HTML/CSS)

Esta aplicación multiplataforma ha sido diseñada para ser lanzada desde cualquier navegador. La página tendrá como dimensiones 1200 x 800. Está al iniciarse muestra dos controles, botón Play y botón Menú. Ver figura 7 en página 35.

El botón Play inicia el juego mostrando una pantalla en la que aparecen un cerdito y un rebaño de ovejas. El cerdito hace la labor de pastor, teniendo que meter un número determinado de ovejas en el corral para poder pasar al siguiente nivel. En la esquina superior izquierda se encuentra el cronómetro, inicializado en dos minutos y realiza la cuenta atrás hasta llegar a cero. A la derecha de éste se encuentra un contador, que irá aumentando según el número de ovejas que vayan entrando en el redil. En la esquina superior izquierda está situado el menú de la aplicación que está compuesto por las siguientes opciones:

- Instructions: Donde se explica el funcionamiento del juego.
- Restart level: Permite reiniciar el nivel.
- Levels: Muestra los niveles del juego que se irán desbloqueando según se vayan superando.

A su izquierda se sitúa el botón para activar y desactivar la música de fondo. La imagen de éste variará dependiendo de si la música está en reproducción o en silencio.

El diseño de la página principal tuvo varios procesos. La evolución que siguió está representada en las imágenes: figura 5, figura 6 y figura 7.

3.2. Canvas

El elemento Canvas de HTML5 es un contenedor gráfico que proporciona a los scripts un mapa de bits que funciona como lienzo. Lo primero que se debe de hacer es referenciar el elemento Canvas dándole unas dimensiones, 851 x 424 (ancho y largo) y adquirir su contexto. En este caso hemos utilizado un contexto bidimensional (2D).

Dentro del archivo HTML es necesario incluir en la etiqueta body llamar a la función Bring_LLuvia().

El lenguaje de programación Javascript contiene una serie de funciones propias que nos permiten a través de coordenadas dibujar en el Canvas. Estas son algunas de ellas.

- beginPath(): Con esta función se da por comenzado el trazo.
- moveTo(x, y): Coloca el cursor en el punto de inicio y a partir de ahí, se va creando la forma de la figura a través de las distintas funciones.
- lineTo(x, y): Traza líneas rectas desde una coordenada a otra.
- fill(): Dibuja una forma cerrada con el color de relleno actual. Si la forma no está cerrada, la propia función crea una línea recta desde el punto de inicio al punto final para cerrarla.
- arc(x, y, radio, ángulo inicial, ángulo final, sentido de giro): Se utiliza para dibujar circunferencias y arcos.
    1. x, y: Son las coordenadas del centro de la circunferencia.
    2. Radio: Radio de dicha circunferencia.
    3. Ángulo inicial y ángulo final: Marca la amplitud del arco desde el ángulo inicial al ángulo final en el sentido de las agujas del reloj. Los ángulos se miden en radianes. La equivalencia con los grados nos la da esta expresión:
    radianes = (Math.PI/180)*grados
    4. El sentido de giro tiene el valor lógico cierto, el arco irá en sentido contrario a las agujas del reloj. Este último parámetro es opcional y por defecto su valor es falso.
- closePath(): Crea una línea recta desde el último punto al punto inicial. Da por finalizada la figura.

Gracias a HTML5, CSS3 y Javascript, hemos podido realizar tanto el fondo del juego, como las figuras que representan a los personajes. Al igual que en la pantalla principal, los elementos del Canvas han sufrido varios procesos, a medida que se ha ido mejorando la técnica de creación, éstos han ido cambiando. Se puede ver un ejemplo en las figuras 8 (página 35), 9 (página 36), 10 (página 36), 11 (página 36) y 12 (página 37).

4. Clases

4.1. Estructura de una clase

lluviaProject , al igual que otros lenguajes orientados a objetos, está organizado en clases.

Para crear una clase es necesario definir una función constructora.

```
var myClass = function() {;}
```

Ésta puede contener atributos y métodos públicos accesible sólo desde la clase, que no se pueden heredar y que se definen como:

```
myClass.attribute = VALUE
myClass.method = function(){;}
```

También se pueden definir atributos y métodos heredables por aquellos objetos que derivan de la clase. Para esto, es necesario añadir prototype entre el nombre de la clase y el nombre del atributo o método.

```
myClass.prototype.attribute = VALUE
myClass.prototype.method = function(){;}
```

Las variables y métodos privados se definen dentro de la función como

```
var attribute = VALUE
```

Una clase puede devolver una variable o un objeto complejo, llamado cierre, con capacidad para acceder a más de un atributo. De esta manera, una clase puede ser, a su vez, una factoría de objetos.

```
var derived_class = new myClass()
```

Estas clases pueden aceptar parámetros o no y new es opcional.

Aunque ***lluviaProject*** no define nada al respecto, los cierres se suelen usar a través de la función yield, la cual busca entre los parámetros un objeto función y lo ejecuta.

Las clases aceptan un objeto de configuración, el cual permite modificar la propia clase en caso necesario. Al igual que en javascript, no es necesario definir el tipo de las variables. Ver ejemplo en código 2 (página 41)

lluviaProject permite definir una función dentro de la propia clase. Ver ejemplo en código 3 (página 42)

La palabra reservada this, dentro del ámbito de new, hace referencia al objeto que se está creando, mientras que fuera de él hace referencia al objeto que se está ejecutando.

Object#method_missing:
Cuando una función no existente es llamada, se levanta una excepción y se llama al método method_missing()

4.1.1. Cadena Prototípica

Las clases siguen las convenciones de las cadenas prototípicas de Javascript. Esto quiere decir que si el atributo no está presente en el objeto, entonces se mira en el constructor de la clase padre. Si ésta deriva a su vez de otra, entonces se busca en el constructor del padre de la segunda. Ver figura figura 13 (página 37)

Aquellas funciones definidas sin la palabra reservada prototype no son accesible por ninguna instancia de la clase. Tanto éstas como aquellas funciones que no que no acceden a los atributos de clase se consideran estáticas.

Es posible definir métodos singleton, que se crean en el constructor de la clase y que no pertenecen a la clase padre. Dentro de estos métodos, this pierde su significado y es necesario que esté recogido en una variable local para poder acceder a él.

4.2. Clase Boid

4.2.1. Introducción

Un Boid se define como un carácter personal autónomo que recibe un objeto de configuración y un bloque. Al crear una nueva instancia, se le puede pasar como parámetro un objeto de configuración y/o un bloque de código. Si se le pasa un objeto de configuración, es necesario definir todos los parámetros a incluir en el constructor del nuevo Boid. Si se le pasa un bloque, se pueden definir sólo aquellos parámetros que vayan a cambiar con respecto a la clase padre.

La configuración por defecto incluye un objeto geo_data, el cual contiene los atributos de posición, velocidad y aceleración, color del boid, un cerebro, velocidad máxima, masa, el objeto vision, con radio y ángulo, y el objeto límites dinámicos, que incluye capacidad de empuje, de giro y de frenada. Ejemplo de configuración por defecto. Código 4 (página 42)

Es posible modificar su posición, velocidad o aceleración en tiempo real según sea necesario. También se le puede asignar un comportamiento de los que ya existen o crear nuevos que cubran necesidades más específicas.

Cada Boid tiene acceso directo a la descripción geométrica de la escena, pero los comportamientos de manada requieren que éste reaccione sólo a aquellos boids que se encuentren dentro de un área específica alrededor de él. Este área se define por una distancia, medida desde el centro del Boid, y un ángulo, medido en la dirección de avance del Boid. Aquellos Boids que se encuentran fuera de este área son ignorados. Ver figura 14 (página 38)

Métodos:

- position(): Obtiene o define la posición del Boid
- velocity(): Obtiene o define la velocidad del Boid
- acceleration(): Obtiene o define la aceleración del Boid
- start(): Guarda la hora que marca el procesador.
- delta_at(): Tiempo que ha pasado desde la última vez que la variable 'last time' fue actualizada
- update_physics(): Calcula la nueva posición, velocidad y aceleración en función del tiempo que ha pasado.
- run(): Actualiza el tiempo en las variables del Boid.
- first_draw(): Pinta el Boid y lo guarda como imagen, lo que ahorra tiempo de cálculo.
- draw(): Pinta un Boid en un mundo definido por un contexto.
- heading(): Alinea el vector normal con el último vector de dirección del Boid.
- locale(): Expresa las coordenadas locales del sistema en coordenadas globales.

- globale(): Expresa las coordenadas globales del sistema en coordenadas locales.
- localize(): Cambia las coordenadas globales del sistema en coordenadas del Boid.
- globalize(): Cambia las coordenadas locales del sistema en coordenadas del mundo.
- visible_objects(): Pregunta al mundo si existe un objeto visible dentro de las habilidades de visión y las coordenadas de velocidad, aceleración y posición. del Boid.
- requested_acceleration(): Devuelve al mundo la aceleración requerida (lo que el cerebro desea y lo que el cuerpo permite).
- clip(): Ajusta la aceleración en función de los límites dinámicos del Boid.
- set_target(): Define un target.
- target_data(): Devuelve la información del target.

4.2.2. Clase Brain

Es la clase que crea un cerebro para el Boid. En ella, se guardan los comportamientos disponibles para ser activados. Estos comportamientos conocidos se almacenan en la variable *catalog*. El cerebro, además, se encarga de discriminar entre aquellos que pueden ser activados por un Boid concreto y los que no.

Guarda las aceleraciones que posee el Boid para cada uno de sus comportamientos activos. De esta manera, se puede calcular la aceleración que el Boid necesita para cambiar su posición en función del objetivo.

Métodos:

- can$U(): Comprueba si el Boid puede activar un comportamiento específico.
- can_be_in$U(): Comprueba si un comportamiento puede ser activado para un Boid.
- activate(): Activa un comportamiento determinado en un Boid.

- deactivate(): Desactiva un comportamiento determinado en un Boid.
- is_in$U(): Comprueba si un comportamiento está en la lista de aquellos aceptados por el Boid.
- get_behavior(): Obtiene el comportamiento pasado como parámetro
- $see_accelerations(): Muestra las aceleraciones guardadas por el método desired_accelerations()
- desired_accelerations(): Guarda las aceleraciones de un determinado Boid para todos sus objetivos
- desired_acceleration(): Calcula la aceleración que el Boid necesita para cambiar su posición.

4.2.3. Clase Sheep

La clase oveja deriva de la clase Boid, por lo que hereda la mayoría de su funcionalidad. La diferencia fundamental con la clase padre es la creación de nuevas funciones y la redefinición de métodos ya existentes.

Métodos:

- sheep_limits(): Define los límites entre los que los Boids oveja se pueden mover. Esta función es necesaria para que éstos no se salgan de los límites impuestos por el canvas. Ver código 5 (página 43)
- update_physics(): Calcula la nueva posición, velocidad y aceleración en función del tiempo que ha pasado. Una de las razones para redefinirla es poder llamar a la función sheep_limits(). Esta función, además, comprueba si el Boid oveja ha entrado en el corral y, en caso afirmativo, suma un punto al contador. Ver código 6 (página 43)
- first_draw(): Pinta el Boid y lo guarda como imagen, lo que ahorra tiempo de cálculo. Se redefine para evitar que los Boids se pinten como círculos, ya que en el constructor se determina una imagen para representar a las ovejas. Ver código 7 (página 43)
- draw(): Pinta un Boid en un mundo definido por un contexto. La diferencia con el método original es que ya no se pintan líneas en el canvas. Si la velocidad es mayor que cero, se carga la imagen de la oveja mirando hacia la izquierda y si no, hacia la derecha. Ver código 8 (página 44).

4.2.4. Clase Pig

La característica principal de esta clase es su capacidad para responder a las coordenadas del ratón cuando el usuario dispara el evento onClick(). Éstas son capturadas por el objeto MouseCoordinates, el cual deriva de la clase Gate. A través de su método do_onclick(), transforma las coordenadas globales de la pantalla en locales del canvas y las guarda en sendas variables de clase. Estas variables se pasan a la clase World como parámetros de su función move_shepherd() y se usan para definir el objetivo del Boid, el cual tiene programado un «"‘¡HEAD comportamiento de persecución. Ver figura 15 (página 38)
======= comportamiento de persecución. Ver imagen 9 (página 44)
"’»¿4492f9f5f200a04154461b5f0fcea00593c642ab

Métodos:

Esta clase cuenta con dos métodos, modificados de la clase original. Al igual que en la clase Sheep, ha sido necesario redefinir first_draw() y draw() para poder cargar la imagen del cerdito en lugar de pintar el círculo que se crea por defecto para representar el Boid.

4.3. Clase Behavior

4.3.1. Introducción

La clase Behavior permite modelar el comportamiento de los Boid. Esto se consigue modificando el vector de aceleración que cada uno de los Boids posee como parte de su estructura interna. Un comportamiento se puede definir como un objeto que devuelve una aceleración deseada en un momento concreto.

Al ser una clase abstracta, no permite crear instancias, de manera que para crear un comportamiento nuevo es necesario generar una clase nueva que derive de ésta.

```
Modo incorrecto de crear un comportamiento:

var a = new Behavior()

Modo correcto:

NewBehavior.prototype = new Behavior\\
```

Estos comportamientos puede tener pre y post modificadores. Los primeros modifican la aceleración antes de devolverla y los segundos, después. Un ejemplo de premodificador es foresee, que se encarga de calcular dónde estará el objetivo en la próxima medida de tiempo. Este premodificador unido al comportamiento de búsqueda seek hacen que varíe la aceleración para modificar la trayectoria y perseguir al objetivo. Un ejemplo de premodificador sería arrival, el cual va disminuyendo la aceleración a medida que se acerca al objetivo.

Los comportamientos disponibles para los Boids en este momento son seek (búsqueda), flee (huida), wander (paseo), containment (contención en el espacio), alignment (alineamiento con otro Boids), cohesion (acercamiento a otros Boids), separation (separación de otros Boids) y obstacle avoidance (sorteo de obstáculos).

Métodos:

- decompose_name(): Dada una cadena de caracteres que representa el nombre del comportamiento y sus modificadores, devuelve un array con la lista de pre modificadores, el nombre del comportamiento y una lista con los post modificadores.
- catalog(): Lista todos los comportamientos y modificadores conocidos.
- new(): Crea un nuevo comportamiento.
- type_of(): Devuelve la clase asociada con el nombre del comportamiento.
- is_a$U(): Comprueba si el nombre pasado como parámetro se corresponde con uno de los comportamientos existentes.
- desired_acceleration(): Devuelve un vector con la aceleración deseada de ese comportamiento.
- is_premodified_by$U(): Devuelve true si el nombre del modificador está en la lista de pre modificadores
- is_postmodified_by$U(): Devuelve true si el nombre del modificador está en la lista de post modificadores.
- is_modified_by$U(): Devuelve true si el nombre del modificador está tanto en la lista de pre como de post modificadores.

- get_modifiers_for(): Devuelve una lista de pre o post modificadores.
- all_modifiers(): Devuelve una lista de todos los modificadores.
- activate_modifier(): Activa un modificador determinado.
- get_modifier(): Busca un modificador determinado y devuelve null si no lo encuentra.
- deactivate_modifier(): Desactiva un modificador determinado.

4.3.2. Clase Sheep Behavior

Es la clase encargada de generar el comportamiento de las ovejas. Deriva de la clase Behavior. Cuando el cerdito oveja entra dentro del radio de visión de la oveja, esto despierta en ella un comportamiento de huida. Es decir, aumenta su aceleración hasta su límite para disminuir ésta gradualmente a medida que se va alejando del Boid cerdito.

Éste es el primero de los cinco comportamientos que se quieren implementar (véase cuadro 1 en la página 40). Los otros cuatro quedan para una fase posterior de desarrollo.

Métodos:

- set_target(): Define el objetivo del Boid, que debe ser otro Boid y el cual se pasa como parámetro.
- target_data(): Define la información sobre la posición del Boid objetivo.
- get_target(): Devuelve la posición del objetivo.
- target_at(): Da la distancia entre el Boid y su objetivo.
- desired_velocity(): Devuelve la velocidad deseada del Boid. Ver código 10 (página 44)
- desired_acceleration(): Devuelve la aceleración deseada del Boid.

4.3.3. Clase Alignment Behavior

Al comportamiento básico de la oveja se le añadió el comportamiento de alineación, con el objetivo de crear una sensación de grupo más marcada. El

comportamiento de Alignment está dentro de los comportamientos de manada del cerebro.

Se calcula el alineamiento medio como la media de las velocidades. Es necesario que aumente la aceleración para eliminar la componente normal al alineamiento en la velocidad del Boid, es decir, suprime toda la velocidad que no esté en la dirección del alineamiento. Ver figura 15 (página 38)
4331

4.3.4. Clase Seek Mouse Behavior

Deriva de la clase Seek Behavior y modifica el comportamiento de la clase Pig a través de uno de sus métodos, al que se le pasan como parámetros las variables x e y, que se corresponden con las coordenadas x e y del ratón, capturadas y transformadas en coordenadas locales por la clase MouseCoordinates.

Este comportamiento se activa en la clase Galactus, en la creación del Boid cerdito. Después, en la función move_shepherd() de la clase World, se consigue el comportamiento de seek mouse, se obtienen los valores de x e y y se escalan en función de la perspectiva del canvas. Estas coordenadas escaladas se pasan como parámetro a la clase Seek Mouse a través de su método set_target_at(). Ver código 11 (página 45)

4.4. Clase World

La clase World deriva de la clase Device y es la que genera el mundo en el que viven los Boids. Necesita un objeto canvas para poder existir, aunque, en principio, sus límites son infinitos. Esto quiere decir que el usuario verá en pantalla tan sólo una parte del mundo, mientras que los Boids pueden representarse en cualquier parte de él, saliendo del canvas en muchas ocasiones.

Las funciones principales del mundo son gestionar el canvas y los boids existentes. A través de diferentes funciones, se puede modificar el tamaño del mundo o del canvas, guardar los Boids creados en una lista y acceder a ésta o actualizar los datos que reciben estos Boids para modificar su estado.

Además, el mundo puede para enviar y recibir mensajes, una de las características de la clase Device. En la variable de clase this.self_events, de tipo Array, están guardados los mensajes que el mundo puede enviar. Puede

recibir cualquier mensaje siempre y cuando haya definida una función attend_nombre_del_mensaje().

Métodos:

- set_dashboard(): Define una nueva interfaz del mundo en el canvas.
- width(): Define el ancho del mundo.
- height(): Define el alto del mundo.
- screen_width(): Define el ancho del canvas.
- assert_screen(): Comprueba si se ha creado un canvas.
- screen_height(): Define el alto del canvas.
- has_born(): Registra la creación de un Boid y se envía un mensaje de notificación a sí mismo.
- get_boids(): Crea un array con todos los Boids.
- each_boid(): Permite acceder a cada boid guardado en el array de Boids
- start(): Inicializa el tiempo de los boids
- is_initalized(): Comprueba si el mundo está inicializado. Si es así, el usuario está reiniciando el juego.
- draw(): Dibuja el canvas en el mundo.
- move_shepherd(): Define el target del Boid en función de las coordenadas del ratón. Ver código 12 (página 45)
- step(): Actualiza la fecha en los cálculos de la física de los Boids.
- is_one_second_from_begining(): Define la hora de comienzo del mundo en un segundo más de la hora en la que la función fue llamada.
- show_boids(): Muestra los Boids que existen actualmente en el mundo.
- check_level(): Comprueba si el nivel ha sido superado por el usuario. Si es así, para el mundo y el cronómetro y llama a la función que cambia la imagen de fondo del canvas. Ver código 13 (página 45)

- running_steady(): Actualiza la hora del procesador, comprueba el nivel y si éste ha sido superado, deja de pintar el canvas. Ver código 14 (página 46)
- visible_for(): Crea un array con los Boids que el Boid referenciado puede ver.
- new_boid(): Crea un nuevo Boid.
- new_seeker(): Crea un nuevo Boid con el comportamiento de búsqueda prefijado.
- start_and_run(): Inicia en mundo y lo pone en marcha.
- attend_focus_boid(): Actualiza la cola de mensajes.
- new_boid_of(): Crea un nuevo Boid de una subclase de Boid.
- method_missing(): Provee el método dinámico new_boid_as_(ClassName)

4.5. Clase Galactus

Clase que se encarga de gestionar la creación de un nuevo mundo al principio de cada juego. Además, se encarga de destruir el mundo anterior cada vez que se reinicia la partida. La función principal de está clase es start_world(), donde se crea el mundo nuevo, se añade un puerto de escucha de mensajes y se llama a las funciones que controlan la cuenta atrás y la música. Además, se crea el Boid cerdito y los Boid oveja y se les pasan los parámetros de configuración y se definen los comportamientos específicos.

La función playSound() se encarga de gestionar el sonido del juego y todo lo relacionado con él. Countdown() es responsable de reloj de cuenta atrás, de iniciarlo cuando se inicia el mundo, pararlo, etc. Destroy_world() y attend_restart_world() son las funciones encargadas de gestionar la destrucción del mundo y creación de uno nuevo cuando el usuario pulsa el botón de reiniciar el juego. Attend_restart_world() es la función que responde al mensaje 'restart_world' enviado por el mundo gracias al puerto añadido en la función start_world().

Métodos:

- start_world(): Inicia el mundo. Ver código 15 (página 46)

- playSound(): Inicia el sonido del juego respondiendo al evento onClick() del botón de play
- countdown(): Inicia la cuenta atrás del temporizador de la pantalla. Si el tiempo se termina, se llama a la función que pinta en el canvas la imagen de final del juego
- destroy_world(): Destruye el mundo actual y deja saber al nuevo mundo que ya había un mundo creado antes.
- attend_restart_game(): Reinicia el mundo. Atiende al mensaje enviado por el gate cuando alguien pulsa el botón restart del juego.

5. Gates y Devices

5.1. Introducción

El servicio de mensajería de ***lluviaProject*** está gestionado por las clases Gate y Device.

Diagrama de Gates y Device. Ver figura 16 (página 39)

5.1.1. Clase Device

La clase Device provee un mecanismo asíncrono de comunicación con otros Devices. No tiene conexión propia con el DOM de HTML, pero ésta se realiza mediante un objeto de la clase Gate. Los Devices reciben eventos, que guardan en una cola de mensajes. En el atributo self_ events es donde se almacena la lista de mensajes que éste puede enviar. Por ejemplo, el Device menuHandler atenderá a los siguientes eventos: get_panel_out(), restart_game(), keep_menu_out() y get_menu_in().

```
this.self_events = [ "get_panel_out", "restart_game",
                     "keep_menu_out", "get_menu_in" ].
```

Cuando se realice el envío de alguno de esos mensajes será necesario la utilización del método fireEvent y newMessage indicando el tipo de mensaje, el nombre del mensaje y el evento al que se refiere a través de un gate. El siguiente ejemplo muestra el envío de un mensaje del tipo síncrono "restart_game":

```
that.newGate("restart_option", Gate, { do_onclick:
     function(event, element) {
            this.device.fireEvent(this.device.newMessage("sync",
            "restart_game", this))
        } })
```

Además, para atender a cada uno de esos mensajes habrá que crear una función 'attend':

```
MenuHandler.prototype.attend_keep_menu_out = function(date, msg) {
//Respuesta al mensaje recibido keep_menu_out de la cola de mensajes.
this.menu_effects.menu_automata.currentState.requested =
                              this[this.view].menu_automata.state.out
}
```

Cada vez que se crea un nuevo Gate, se añade un puerto al Device para que sea posible la comunicación.

```
this.handler.addPort("restart_game", this.world)
    countdown(this.world);
```

A su vez los devices pueden enviar mensajes a otros devices lanzando un nuevo evento de envío de mensaje con los fireEvent y newMessage. Por ejemplo:

```
this.device.fireEvent(this.device.newMessage("sync",
                                    "restart_game", this))
```

5.1.2. Clase Gate

La clase Gate se encargan de gestionar eventos. Algunos son onClick(), onMouseOver() o onMouseOut(), que responden a la pulsación del ratón, a su paso por encima de un elemento y a su salida de él, respectivamente. Un objeto Gate está contenido dentro de un objeto Device y en su creación recibe como parámetros opcionales los siguiente elementos:

- Identificador del elemento HTML que va a responder al evento (por ejemplo una etiqueta div)
- Contenedor HTML donde situar el objeto Gate
- Acciones de respuesta.

Por ejemplo, si en el documento HTML tenemos el siguiente elemento:

```
<div id='button_bar'> </div>
```

Escribiríamos:

```
that.newGate("instructions_option", Gate, {do_onclick:
    function(event, element) {
         alert("Move the little pig to place sheeps
               into the barnyard")
}
```

Lo que genera que ante un evento, en este caso on_click, sobre el div con identificador instructions_option, se produzca una respuesta como puede ser un mensaje emergente (alert).

Otra manera posible manera de crear un Gate sería la siguiente, en la que crearíamos una subclase (en el ejemplo, la clase Animation) de la clase Gate.En su creación recibe como parámetro "element" el elemento HTML del cual ha de recibir los eventos y en su función initialize hemos de llamar al supercontructor de Gate, con el método call() para heredar sus métodos y atributos. Posteriormente, crearemos las funciones que se deben ejecutar al producirse los eventos. Ver código 19 (página 48), 17 (página 47) y 17 (página 47)

Por último, dentro del Device que va a contener el Gate, llamaremos al método newGate para crear el objeto de la subclase que hemos creado, indicando el elemento HTML sobre el que actúa y el nombre de nuestra subclase:

```
that.menu_effects = that.newGate("menu", Animation)
```

5.2. Menú y Submenú

5.2.1. Clase MenuAutomata

Controla los estados de entrada y salida del menú. Deriva de la clase threadAutomata y tiene una serie de estados posibles:
'out', 'getting_out', 'getting_in', 'inside'.
Estos estados, a su vez, pueden encontrarse en estado anterior (previous), actual (current) o solicitado (requested). Según cual sea el estado actual (current), se realizarán la funciones correspondientes.

Los estados irán cambiando de currentState desde "requested" a "current" y posteriormente a "previous".

- Estado out: Cambia la altura de menú a 250px, que es la longitud del menú una vez está totalmente desplegado, incluyendo el submenú correspondiente a los niveles. Ver código 19 (página 48)
- Estado getting_out: Incrementa la altura de menú de 5 en 5px mientras este mida menos de 206 px y más de 50px, para que se despliegue poco hasta salir del todo. Ver código 20 (página 48)
- Estado getting_in: Decrementa la altura de menú de 5 en 5px, mientras éste mida más de 55px para que de forma gradual vuelva a ocultarse. La altura mínima es corresponde a la altura del botón menu. Ver código 21 (página 48)

5.2.2. Clase Animation

Recoge los eventos que se producen en el menú a partir de la interacción del usuario. Animation deriva de la clase Gate. Recibe el elemento HTML que envuelve. En su función initialize() además de crear como atributo propio el elemento (HTML) y de llamar al super constructor de Gate para heredar sus metodos y atributos, asocia los efectos de menuAutomata al elemento HTML recibido.

La clase Animation cuenta con los métodos do_onmouseover y do_onmouseout:

- do_onmouseover(): Añade como estado solicitado el estado getting_out del menuAutomata, indicando que al pasar el ratón sobre el elemento HTML, en este caso la etiqueta 'menu' el estado solicitado sea 'saliendo' de menuAutomata..
- do_onmouseout(): Añade como estado solicitado el estado getting_in del menuAutomata, indicando que al pasar el ratón sobre el elemento HTML, en este caso la etiqueta 'menu' el estado solicitado sea 'entrando' de menuAutomata.

5.2.3. Clase MenuHandler

Menu handler deriva de la clase device. Contiene los Gates que gestionan los eventos que se dan en cada uno de los botones del menú, siendo capaz de enviar los mensajes de "restart_game", "pause_clock", y "resume_clock" que se encuentran en la cola de mensajes self_events, para reiniciar el juego, pausar el reloj y reiniciar el reloj respectivamente.

Se comunica con la clase Galactus ya que ésta contiene las funciones attend que hacen referencia a los mensajes que es capaz de enviar MenuHandler. Ver código 24 (página 49)

Para el despliegue del menu, creamos un objeto de la clase Animation, la cual es una subclase de Gate, para el elemento o etiqueta "menu"del HTML para asociar los efectos del Gate Animation al atributo menu_effects y a partir de ello, creamos un menuAutomata.

```
that.menu_effects = that.newGate("menu", Animation)
```

Para realizar las funciones de cada botón del menú:

1. Crea un Gate para etiqueta instructions_options del elemento HTML, sobre la cual si se recibe el evento do_onclick (pulsar sobre el botón) muestra las instrucciones del juego. Ver código 22 (página 48)

2. Crea un Gate para la etiqueta restart_option del elemento HTML, si se realiza un clic sobre esa opción del menú, lanza como evento el envío de mensaje de llamar a la función restart_game(). Ver código 23 (página 48)

3. Crea un Gate para el elemento level_option del HTML, sobre el cual si se realiza un clic, muestra los diferentes niveles, cambiando el display de la etiqueta level_option_container a inline.

La clase MenuHandler tiene dos métodos, attend_keep_menu_out, attend_keep_menu_in.

- attend_keep_menu_out(): Guarda el estado out de menuAutomata como currentState solicitado (requested) para que posteriormente pase a ser current (actual) y se ejecuten las intrucciones correspondientes para hacer permanecer al menú desplegado.

- attend_get_menu_in(): Guarda el estado getting_in de menuAutomata como currentState solicitado (requested) para que posteriormente pase a ser current (actual) y se ejecuten las intrucciones correspondientes para hacer que el menú se vaya ocultando progresivamente.

5.2.4. Clase OptionHandler

Envía mensajes al device MenuHandler para poder gestionar el despliegue del menú principal cuando se muestra el panel de niveles. Al igual que MenuHandler este deriva de la clase Device. A través de Levels, que se explica posteriormente, es capaz de enviar los siguientes mensajes:

- get_panel_out: Despliega el panel de niveles.

- keep_menu_out: Mantiene desplegado el menú principal.

- get_menu_in". Guarda tanto el menú principal como el panel de niveles.

Será necesaria la llamada al método newGate para la creación de un objeto Levels el cual deriva de la clase Gate. Ver código 25 (página 49)

5.2.5. Clase Levels

Levels pertenece a la clase Gate, se encarga de gestionar el envío de mensajes desde OptionHandler a MenuHandler (ambos Device) mediante el uso del método fireEvent() anteriormente explicado.

- do_onmouseover: Provoca que menu se mantenga desplegado. Contiene la función fireEvent, esta pasa el mensaje síncrono en este caso 'keep_menu_out' y el evento do_onmouseover. Ver código 26 (página 49)

- do_onmouseout: Provoca que cuando se retire el foco del submenu de niveles, este se oculte. La funcion do_onmouseout recoge el elemento html 'levels_container' y cambia su display a none. También envía mediante el método fireEvent el mensaje asíncrono 'get_menu_in', y el evento do_onmouseout. Ver código 27 (página 49)

5.3. Clase Clock

El principal motivo de su creación fue la necesidad de poder cronometrar el tiempo de juego con una cuenta atrás. Clock deriva de la clase Device y cuenta con los siguientes atributos:

- start_time: A partir de la función get_now (función que obtiene las horas, minutos del sistema, los pasa a segundos y devuelve su suma para obtener el tiempo actual en segundos.) almacena la hora actual en segundos.

- total_time: Almacena el tiempo en segundos del que se dispone para jugar (es recibido como parámetro).

- remaining_time: Segundos que quedan por jugar. Queda inicializado a total_time.

- before: Almacena en segundos el momento en el que se empezó a jugar.

- running: Variable bandera que hace hace referencia al estado el marcha del reloj. Inicializado a true.

Desde initialize llamaremos al contructor de Device para que la clase Clock sea una de sus subclases, heredando sus métodos y atributos.

Métodos:

- reset(): Reinicia el reloj. Llama al método pause, y reinicializa el valor de remaining_time a total_time.
- get_count(): Devuelve los segundos que quedan por jugar (remaining_time). Es decir, devuelve el valor de remaining_time.
- run(): Recalcula el tiempo que queda por jugar. Llama a la función get_now, para conocer el momento actual. Este valor es almacenado en la variable now, recalcula el valor de remaining_time (segundos que quedan por jugar) en función del momento actual:

  tiempo que queda = al tiempo que quedaba - tiempo que ha pasado.

  Si remaining_time es menor o igual que cero, pone con valor falso el atributo running, es decir, el reloj deja de estar en marcha.
- pause(): Pausa el tiempo, cambiando el valor de running a falso.
- resume(): Continua la cuenta atrás desde el momento que fue pausado.Reinicializa start_time al momento actual, a través de la función get_now. A su vez establece el valor de running a true y por último cambia el valor de total_time por el valor de remainig_time, el cual en ese momento guarda los segundos que quedaban por jugar cuando el reloj fue pausado
- get_string(): Devuelve una cadena con el tiempo restante con formato mm:ss. Si running es igual a verdadero, es decir, si el reloj esta en marcha, se llama al método run y obtenemos el valor de los minutos y de los segundos utilizando las funciones Floor() y Round() de la librería Math de javascript. Posteriormente se realiza la concatenación de cadenas para darle formato. Si por el contrario el reloj no está en marcha(running es falso), realizamos que acabamos de indicar sin llamar al método run().

6. Música

6.1. Introducción

Desde la función playSound de Galactus se crea un objeto AudioContext el cual controla la ejecución del procesamiento del audio. En esta misma función se crea un objeto de la clase bufferLoader.

En su creación pasamos como parámetros:

- El objeto AudioContext
- Un array que contiene las rutas de los archivos de audio que se quieren reproducir.
- La llamada a la función finishedLoading.

La función finishedLoading() se encargará de crear y reproducir los recursos, recogiéndolos del array que contiene las rutas. Esta función será ejecutada después de solicitar la ejecución del método load de bufferLoader.

6.2. Clase BufferLoader

Su principal función es la de cargar las pistas de música en un buffer de datos. Para que ello se lleve a cabo, hay que ejecutar su método load(). Este método se encarga de llamar al método loadBuffer para cada elemento del array con las URLs de las pistas de canciones que se envían como parámetro, mandando éstas así como su posición en el array.

Este último método cargará las pistas de audio en el buffer, realizando una petición HTTP al servidor y descodificará el audio como respuesta, añadiendo esa pista recibida al buffer y, posteriormente, ejecutará la llamada a la función finishedLoading (la cual indicamos que tiene que ser llamada en la creación de la clase bufferLoader()), en el caso de no haberse encontrado ningún error.

Por último la función stopPlaying de Galactus el método stop para la pista de música.

7. Conclusiones

Este proyecto nos ha permitido comprobar la importancia y las ventajas de los lenguajes orientados a objetos, que permiten una programación más ágil y eficiente que los lenguajes estructurados. Esto permite expresar aspectos de la vida cotidiana de forma más realista.

Como ***lluviaProject*** es un framework multihilo gestionado por señales, permite añadir a Javascript funciones nuevas, fundamentales para el desarrollo del proyecto. Esto permite enviar y recibir mensajes entre distintos componentes de la aplicación, mejorando la respuesta a los diferentes eventos. Además, la utilización de caracteres personales autónomos (Boids), nos ha permitido comprobar que se puede modelar artificialmente el comportamiento de un animal a partir de operaciones sencillas con vectores.

Modificando la aceleración en función de un estímulo se pueden generar comportamientos de huida, en los que se aumenta la aceleración hasta su máximo durante un espacio para ir reduciéndose de manera gradual a medida que se aleja del estímulo. Si, además, se le añade un radio de visión, compuesto de radio y ángulo, también se pueden generar otro tipo de comportamientos.

Según el ángulo de visión, se generan comportamientos, los cuales pueden variar en función de los hábitos alimenticios de una determinada especie de Boid, por ejemplo. Es decir, si un Boid es carnívoro, tendrán los ojos ubicados en la parte delantera de la cabeza, mientras que si es herbívoro, éstos estarán situados en los lados. Esta variación del ángulo de visión permite ajustar lo observado en la vida real a la simulación artificial.

El radio de visión permite determinar la distancia que ve el Boid. Éste, por ejemplo, puede variar en función de la edad, siendo mayor cuando más joven es el Boids. También permite determinar si una especie tiene comportamientos de manada o no, en función de cuánto se acerquen o alejen de otros animales similares dentro de su radio de visión. De la misma manera, un Boid solo con un comportamiento de manada programado tenderá a acercarse a otros miembros cercanos de la manada.

Tanto para la visión como para los otros atributos del Boid, se ha comprobado que cambios de valor demasiado grandes en las variables generan comportamientos que no son naturales, demasiado ordenados en comparación con aquellos que existen en la naturaleza. Son las variaciones pequeñas

entre valores de un atributo las que devuelven comportamientos reales. Por ejemplo, variando levemente el valor del radio en el comportamiento de separación y el valor de la aceleración en el comportamiento de alienación se obtiene, de un comportamiento de bandada de pájaros, otro de grupo de peces.

Se ha observado también que los Boid oveja responden alejándose en grupo y en línea recta cuando el Boid cerdito se acerca a ellos por la parte trasera moviéndose en zig-zag. Y si el Boid cerdito se sitúa en el punto de equilibrio (cuartos delanteros) del Boid oveja, los Boid oveja retroceden.

Los resultados obtenidos en este proyecto pueden proporcionar una herramienta para la simulación de comportamientos. Lo que en este proyecto se basa en animales puede ser extendido a personas o partículas, generando una plataforma de experimentación.

8. Figuras

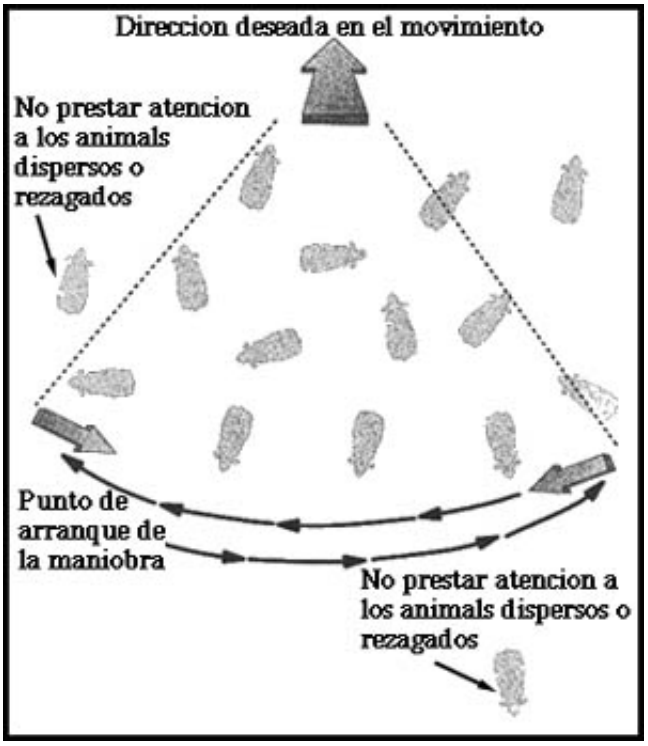

Figura 1: Técnica limpiaparabrisas

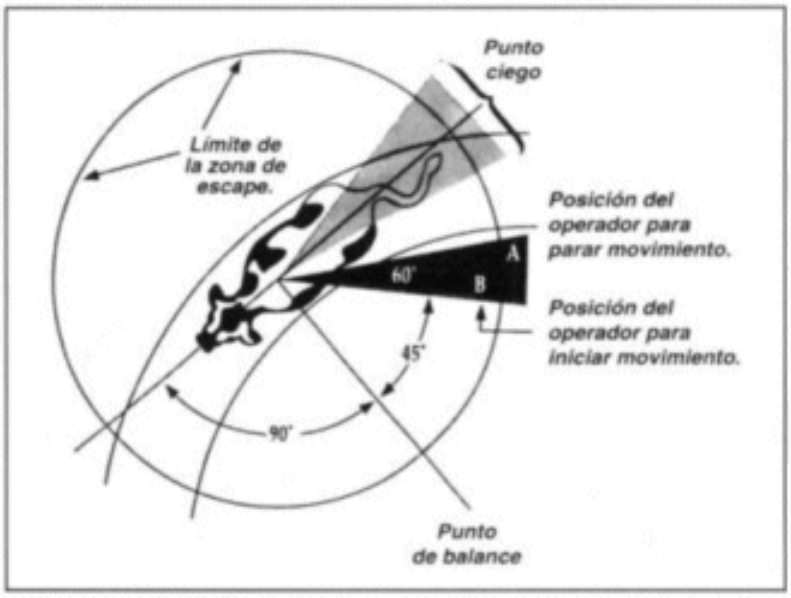

Figura 2: Zona de fuga

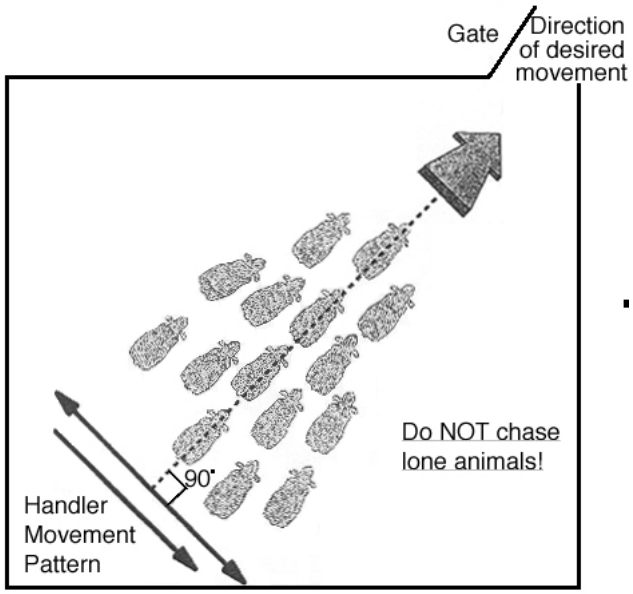

Figura 3: Sacar al ganado: Un controlador

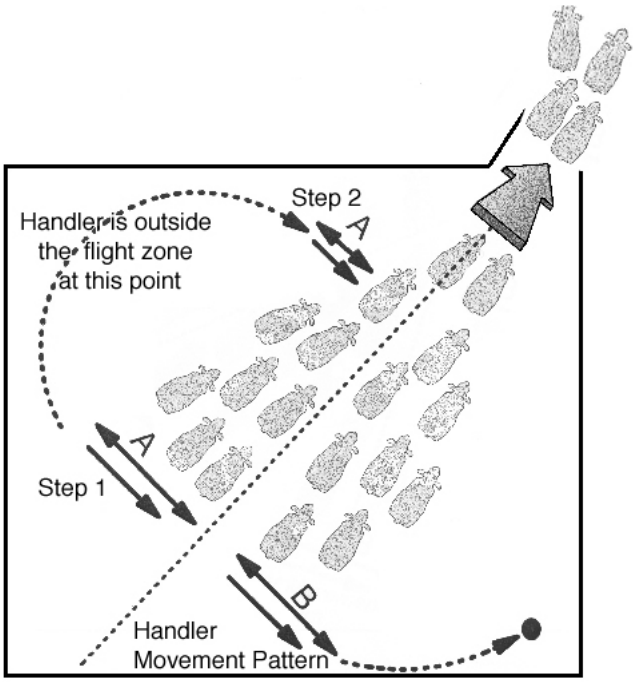

Figura 4: Sacar al ganado: Dos controladores

Figura 5: Primer diseño

Figura 6: Segundo diseño

Figura 7: Diseño final

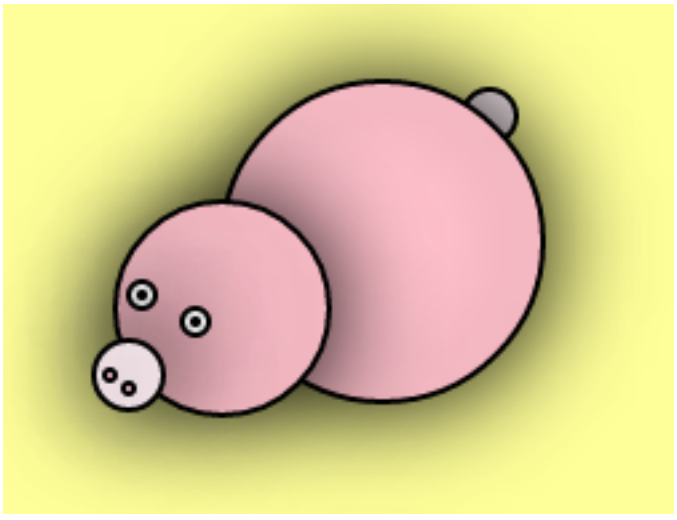

Figura 8: Primer cerdito

Figura 9: Oveja - sólo cabeza

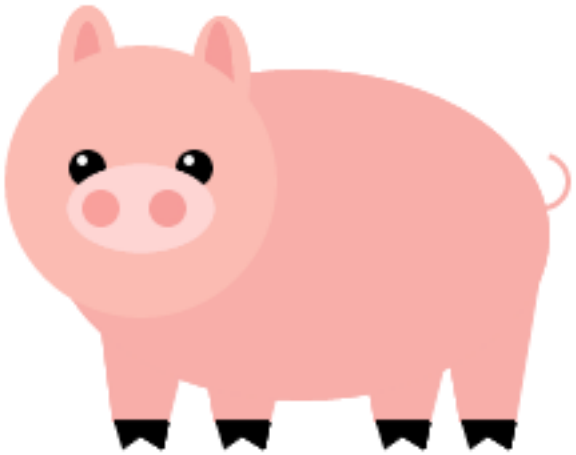

Figura 10: Cerdito definitivo

Figura 11: Oveja definitiva

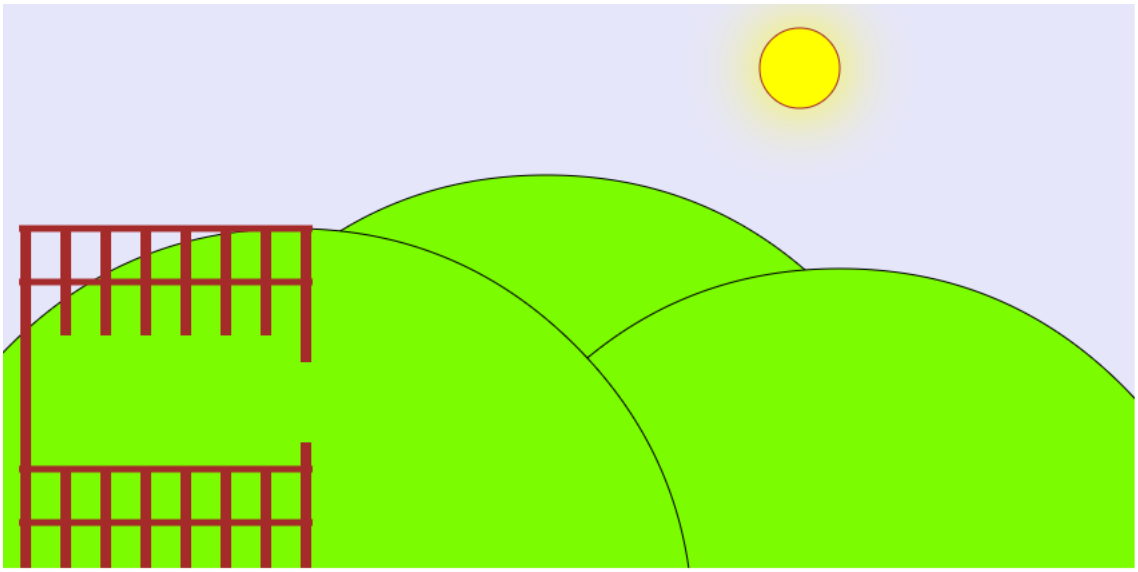

Figura 12: Primer canvas

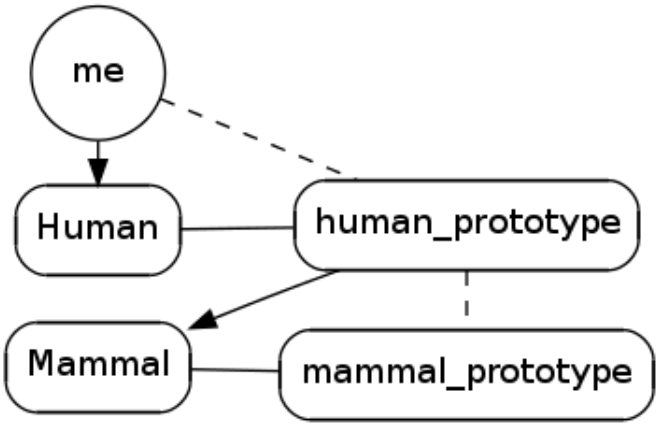

Figura 13: Herencia prototípica

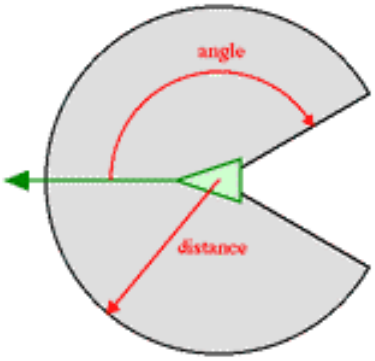

Figura 14: Radio de un Boid

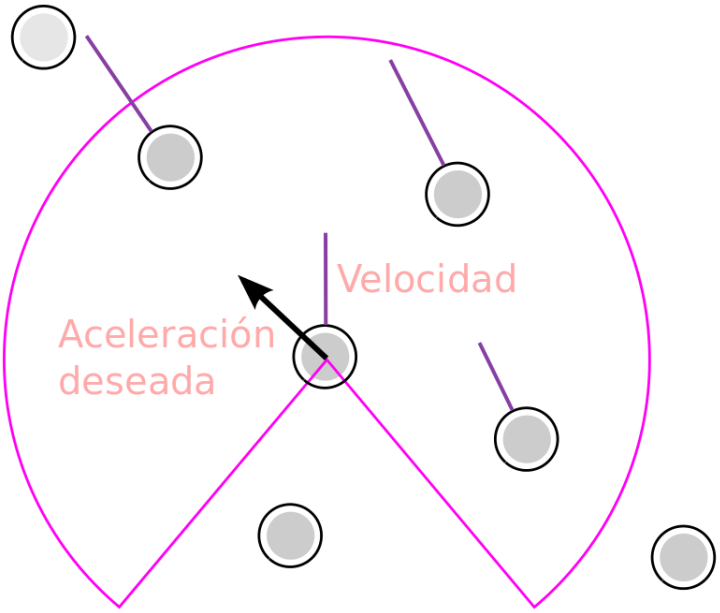

Figura 15: gráfico de velocidades y aceleraciones

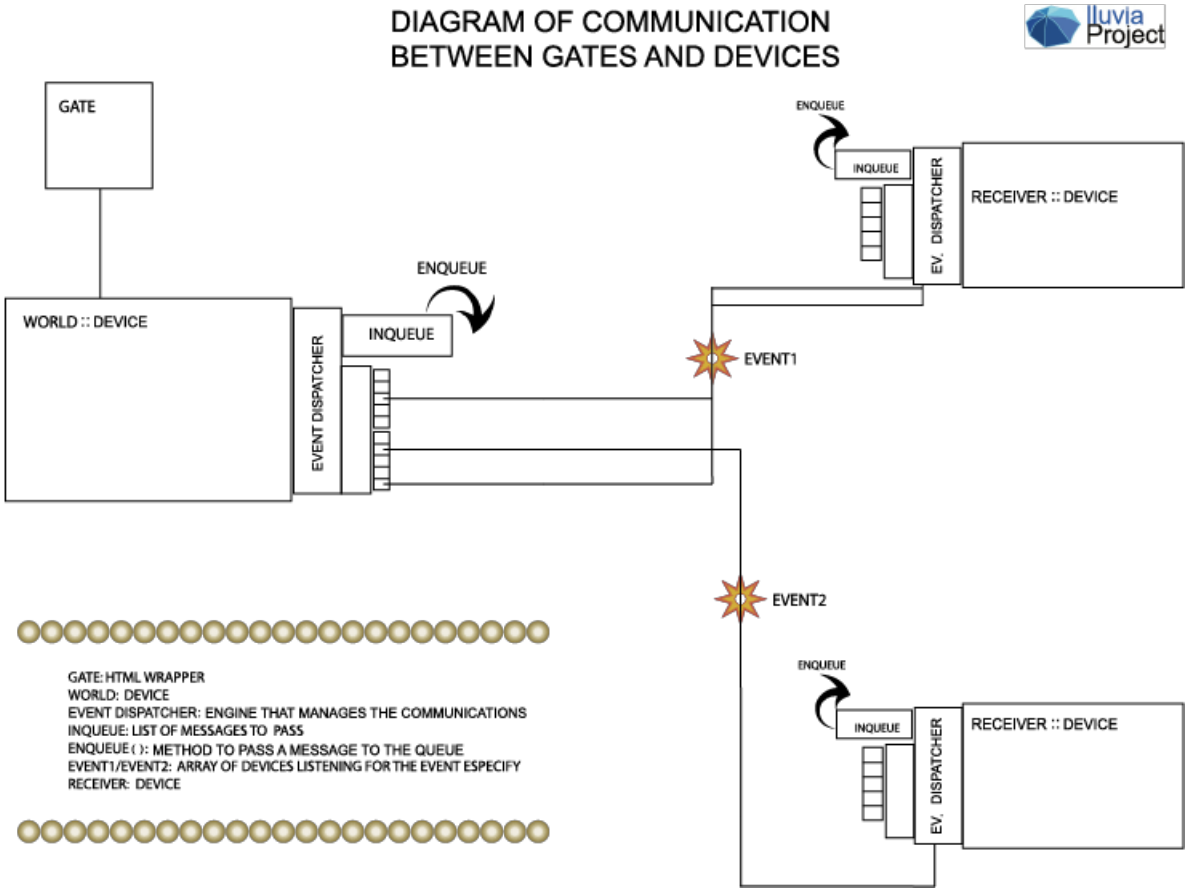

Figura 16: Diagrama Gate y Device

9. Cuadros

Cuadro 1: Comportamientos. Cuadro 4.3.2.1

Nombre de la técnica	Boid cerdito	Boid oveja
Técnica del limpiaparabrisas	El pastor debe moverse en zig-zag detrás de la manada, para mantenerlos en línea recta.	Las ovejas deben de agruparse y moverse en línea recta.
Zona de fuga	El cerdito se encuentra en la zona de fuga de la oveja.	La oveja se agita y se enfrenta a él.
Moverlos en una manga	El cerdo debe de situarse enfrente del punto de equilibrio	Las ovejas avanzan hacia atrás.
Sacar a las ovejas del corral con un controlador	El cerdo se sitúa a 90º detrás del ganado. Los movimientos deben de ser perpendiculares a los del ganado, hacia delante y hacia atrás sobre la barra transversal de una gigante T.	las ovejas salen en manada del corral.

10. Código

```
$K\_app\_dependencies = [
    {   module: "Boids",
        description: "Boids Demo App.",
        path: "",
        files: [
          { name: "brain/behavior\_modifier.js",
            description: "Self protection behaviors." },
          { name: "brain/behavior.js",
            description: "Abstract Behavior." },
          { name: "brain/security_behavior.js",
            description: "Self protection behaviors." },
          { name: "brain/itinerant_behavior.js",
            description: "Definition of itinerant behaviors."},
          { name: "brain/behavior_group.js",
            description: "Group of related behaviors."},
          { name: "brain/brain.js",
            description: "Boid Brain." },
          { name: "boid.js",
            description: "One Boid." },
          { name: "world_interface.js",
            description: "World Interface." },
          { name: "boid_editor.js",
            description: "Boid panel editor." },
          { name: "world.js",
            description: "The world where all boids live."},
          { name: "main.js",
            description: "main function." },
        ]
    }
]
```

Listing 1: Ejemplo de archivo dependencies

```
 function Vehicle(mass, number_of_wheels){
  this.mass = mass // In kg
  this.number_of_wheels = number_of_wheels
  Vehicle.prototype.yield(this)
}

boeing = new Vehicle(20000, 8,
  function(newly_created_airplane){
    newly_created_airplane.wings = 2
})
```

Listing 2: Ejemplo clase

```
 var animal = function(limbs, name){
  if (limbs<5)
      return function Mammal(main_food){
          this.name  = name
          this.limbs = limbs
          this.main_food = main_food
      }
    else
      return function Spider(number_of_teeth){
          this.name  = name
          this.limbs = limbs
          this.number_of_teeth = number_of_teeth
      }
}

dog = new (Animal("Tim", 4))("meat")
```

Listing 3: Ejemplo clase con funciones

```
 var default_config = {
           geo_data: {
               position: new Vector(Math.floor(Math.random()
                                    * 400),
                                   Math.floor(Math.random()
                                    * 400)),
               velocity: new Vector(Math.floor(Math.random()
                                    * 40),
                                   Math.floor(Math.random()
                                    * 40)),
               acceleration: new Vector(0,0)
           },
           colour: "blue",

           brain: new Brain(that),
           vel_max: 50,
           mass: 2,
           vision: {radius: 100, angle: 130 * Math.PI / 180},

           force_limits: {
               thrust: 20,
               steering: 50,
               braking: 70
           }
}
```

Listing 4: Objeto configurable Boids

```
Sheep.prototype.sheep\_limits = function() {
   var x\_axis = this.geo\_data.position.Coord[0]
   var y\_axis = this.geo\_data.position.Coord[1]

   if(y_axis >= 800 && x_axis >= 170 ||
      y_axis >= 800 && x_axis <= 274){
          if(this.first_time == 0){
                this.my_world.max_score = true
                return
          }
          else
                return
   }
   if(x_axis <= -445 || x_axis >= 395)
          this.geo_data.velocity.Coord[0] = 0
   if (y_axis >= 800 || y_axis <= 0) {
           this.geo_data.velocity.Coord[1] = 0
   }
}
```

Listing 5: Funcion sheep_limits

```
Sheep.prototype.update_physics = function(current_time){
   this.last_time = this.current_time
   this.current_time = current_time
   this.geo_data.acceleration = this.requested_acceleration()
   this.geo_data.velocity = integrate(this.geo_data.velocity,
                this.geo_data.acceleration, this.delta_t() )
   this.sheep_limits()

   if(this.my_world.max_score == true){
           this.my_world.points++
           this.first_time = 1
           this.my_world.max_score = false
   }
   this.geo_data.position = integrate(this.geo_data.position,
                  this.geo_data.velocity, this.delta_t() )
}
```

Listing 6: Funcion update_physics

```
Sheep.prototype.first_draw = function() {
   var canvas = document.createElement('canvas');
   canvas.width = 24;
   canvas.height = 24;

   var ctx = canvas.getContext('2d');
```

```
}
```

Listing 7: Función first_draw

```
Sheep.prototype.draw = function(ctx){
    var p = this.geo_data.position
    var v = this.geo_data.velocity
    var a = this.geo_data.acceleration
    var radius = 10
    var scale = 1 - p.get_coord(1) / 3000
    var x = this.geo_data.velocity.get_coord(0)

    ctx.save()
    ctx.scale( scale, scale / 2 )

    if(x < 0) {
        ctx.drawImage(this.image,
                      p.get_coord(0),
                      p.get_coord(1))
    }
    else
        ctx.drawImage(this.image_left,
                      p.get_coord(0),
                      p.get_coord(1))

    ctx.restore()
}
```

Listing 8: Función draw

```
MouseCoordinates.prototype.do_onclick = function (ev, el) {
  this.X = ev.pageX - screener.offsetLeft
  this.Y = ev.pageY - screener.offsetTop
  this.device.move_shepherd(this.X, this.Y)
}
```

Listing 9: Función do_onclick

```
SheepBehavior.prototype.desired_velocity = function(){
 var arrival_distance
 try{
    arrival_distance = this.target_at().module()
 }catch(err){
    arrival_distance = 0
 }
 var scale = 1
```

```
 if(arrival_distance >= 100){
    scale = 0
 }
 else{
    scale = arrival_distance / 100
 }

 return (new Vector(this.target_at().unit().scale(- scale
                     * this.me.vel_max)))
}
```

Listing 10: Función desired_velocity

```
var behaviour = this.shepherd.brain.get_behavior("seek mouse", null)
var x = screen_x - 425
var y = 500 - screen_y
var scale = 1 - y / 3000

behaviour.set_target_at( x / scale, 2 * y / scale)
%
```

Listing 11: Función set_target_at

```
World.prototype.move_shepherd = function (screen_x, screen_y){
   if (!this.shepherd)
       return
   var behaviour = this.shepherd.brain.get_behavior(
                                        "seek mouse", null)
   var x = screen_x - 425
   var y = 500 - screen_y
   var scale = 1 - y / 3000

   behaviour.set_target_at( x / scale, 2 * y / scale)
}
```

Listing 12: Función move_shepherd

```
World.prototype.check_level = function() {
    if(this.level == 1 && this.points == 5){
            this.is_finished = true
            this.currentState.requested = this.state.suspended
            this.clock.pause()
            this.winner_pig()
    }
}
```

Listing 13: Función check_level

```
World.prototype.running_steady = function(processors_time){
        this.now = processors_time || new Date()
        this.coord_x = this.mouse_coordinates.get_mouse_X()
        this.coord_y = this.mouse_coordinates.get_mouse_Y()

    score_number.style.float = "right"
    score_number.style.fontSize = "24pt"
    score_number.style.marginTop = "5px"
    score_number.style.fontWeight = "bold"
    score_number.innerHTML = ":" + this.points

    this.check_level()
    if(this.is_finished == false)
        this.draw()
}
```

Listing 14: Función running_steady

```
Galactus.prototype.start_world = function() {
   this.world = new World(this.view)
   this.world.level = 1

   this.handler.addPort("restart_game", this)

   this.countdown()
   this.playSound()

   var pig = this.world.new_boid_of(Pig, function(config) {
       config.brain.activate("seek mouse", null)
   })

   this.world.shepherd = pig

   var sheeper = [ ]
   for (var i=0; i<30; i++) {
        sheeper.push( this.world.new_boid_of(Sheep,
           function(config){
             config.geo_data.position = new Vector(
                 Math.floor(Math.random()*400),
                 Math.floor(Math.random()*400) )
             config.geo_data.velocity = new Vector(
                 Math.floor(Math.random()*20),
                 Math.floor(Math.random()*20)  )
             config.geo_data.acceleration = new Vector(0,0)
             config.vel_max = 70
             config.vision = {radius: 100, angle: 80 *
                                        Math.PI / 80}
             config.force_limits = {
```

```
                    thrust: 50,
                    steering: 50,
                    braking: 100
                }
                config.brain.activate("alignment")
                config.brain.activate("sheep", pig)
        }))
    }
     this.world.start()
}
```

Listing 15: Función start_world

```
function Animation(element) {
    var that = this
    function initialize(){
        try {
            if (element) {
                that.element = element
                that[element] = {}
                Gate.call(that, element)
                }
        } catch (e) {
            if ($K_debug_level >= $KC_dl.DEVELOPER)
               alert("No event handlers were
               found.\nException: " + e.toSource())
        }
    }
    if (arguments.length)
        initialize()
}
```

Listing 16: Constructor Animation

```
Animation.prototype.do_onmouseover =
   function(ev, el){
      this[this.element].menu_automata.currentState.requested
         = this[this.element].menu_automata.state.getting_out

}
```

Listing 17: Clase Button

```
Animation.prototype.do_onmouseout =
   function(ev, el){
      this[this.element].menu_automata.currentState.requested
```

```
4           = this[this.element].menu_automata.state.getting_in
5
6 }
```

Listing 18: Clase Button

```
1   [function(){
2       this.gate.panel.style.height = "" + 250 +"px"
3   }; ],
```

Listing 19: Estado out

```
1   [function (){
2       if( this.menu_height >= 50 && this.menu_height<=205)
3          this.menu_height+=5
4       this.gate.panel.style.height = "" + this.menu_height
5                                       + "px"
6   };],
```

Listing 20: Estado getting_out

```
1   [function(){
2       if( this.menu_height >=55 ){
3           this.menu_height-=5
4       this.gate.panel.style.height = "" + this.menu_height
5                                       + "px"
6   },],
```

Listing 21: Estado getting_in

```
1 that.newGate("instructions_option", Gate, {do_onclick:
2                       function(event, element) {
3                          alert("Move the little pig to
4                                 place sheeps into the barnyard")
5         } })
```

Listing 22: Instructions_option Gate

```
1  that.newGate("restart_option", Gate, { do_onclick:
2                       function(event, element) {
3              this.device.fireEvent(
4                        this.device.newMessage("sync",
5                                 "restart_game", this)
```

```
6                           )
7         } })
```

Listing 23: restart_option

```
this.self_events = [ "restart_game",      pause_clock     ,      resume_clock      ]
Gaclactus:

Galactus.prototype.attend_pause_clock = function(date, mssg) {
    this.world.clock.pause()
    this.world.clock.get_string()
    }
```

Listing 24: variable self_events

```
that.newGate("level_option", Gate, {do_onclick: function(event, element){
            var levels_container= document.getElementById('level_option_contair
            levels_container.style.display='inline'
        } })
```

Listing 25: level_option

```
Levels.prototype.do_onmouseover = function(ev, el){
   this.device.fireEvent(this.device.newMessage("sync", "keep_menu_out", this))
}
```

Listing 26: do_onmouseover

```
Levels.prototype.do_onmouseout = function(ev, el){
    var levels_container= document.getElementById('level_option_container')
    levels_container.style.display='none'
    this.device.fireEvent(this.device.newMessage("sync", "get_menu_in", this))
}
```

Listing 27: do_onmouseout

Índice de figuras

Índice de cuadros

Listings