Este proyecto no podría haberse completado sin la colaboración de José Luis Álvarez y 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. Figura 1.1.1
- 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 1.1.2
- 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.
- 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.

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.js, 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 figura 2.1.1

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 3.1.3

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. En el centro superior de la pantalla 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.

El diseño de la página principal tuvo varios procesos. La evolución que siguió está representada en las imágenes: figura 3.1.1, figura 3.1.2 y figura 3.1.3

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 3.2.1, 3.2.2, 3.2.3, 3.2.4, 3.2.5

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 figura 4.1.1

lluviaProject permite definir una función dentro de la propia clase. Ver ejemplo en figura 4.1.2

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 4.1.3

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. Figura 4.2.1.1

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 4.2.1.2

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. Todos estos comportamientos conocidos se guardan 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 figura 4.2.3.1
- update_physics(): Calcula la nueva posición, velocidad y aceleración en función del tiempo que ha pasado. Esta función ha sido necesaria redefinirla para poder llamar a la función sheep_limits(). Además, se comprueba si el Boid oveja ha entrado en el corral y, en caso afirmativo, se suma un punto al contador. Ver figura 4.2.3.2
- 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 figura 4.2.3.3
- 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 figura 4.2.3.4

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 comportamiento de persecución. Ver figura 4.2.4.1

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 forsee, 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 4.3.2.1). 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 figura 4.3.2.2
- 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 4.3.3.1

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 Mundo, 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 figura 4.3.4.1

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 podrán 'viajar' a cualquier parte de él, llegando a desaparecer del canvas.

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 figura 4.4.1
- 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 figura 4.4.2
- running_steady(): Actualiza la hora del procesador, comprueba el nivel y si éste ha sido superado, deja de pintar el canvas. Ver figura 4.4.3
- 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 figura 4.5.1
- 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.

Device: Provee un mecanismo asíncrono de comunicación. Usa una cola de mensajes y dispara eventos. No tiene conexión propia con el DOM de HTML, pero ésta se realiza mediante un objeto de la clase Gate.

Gate: Es un 'envoltorio' para elementos HTML.

Están diseñados para responder a eventos HTML manteniendo el campo de aplicación objeto. Recibe como parámetros opcionales el elemento HTML a envolver (por ejemplo un div), el contenedor HTML donde situar el Gate y las acciones de respuesta.

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

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

Escribiríamos:

```
var brush_button = new Gate('brush_btn', 'button_bar')
```

Lo que generaría un div con id: 'brush_btn' y un objeto javascript que responderá a cada evento HTML que se haya definido en el correspondiente handler. Ver figura 5.1.1

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.

- Estado out: Cambia la altura de menú a 250px. Ver figura 5.2.1

- Estado getting_out : Incrementa la altura de menú de 5 en 5px mientras este mida menos de 206 px y mas de 50px. Ver figura figura 5.2.2
- Estado getting_in : Decrementa la altura de menú de 5 en 5px mientras éste mida más de 55px. Ver figura figura 5.2.3

5.2.2. Clase Animation

Recoge los eventos que se producen en el menú a partir de la interacción del usuario. Animation.js deriva de la clase Gate. Recibe el elemento HTML que envuelve el Gate Animation. En su función initialize además de crear como atributo el elemento (HTML) y de llamar al super constructor de Gate, 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. Es decir, el estado 'saliendo' de MenuAutomata.
- do_onmouseout: Añade como estado solicitado el estado getting_in del MenuAutomata. Es decir, el estado 'entrando' de MenuAutomata.

5.2.3. Clase MenuHandler

Se encarga de disparar los eventos que se producen en el menú. Se comunica con la clase OptionHandler. Menu handler deriva de la clase Device, éste tiene sus propios eventos : self_events.

Para el despliegue del menú, creamos un Gate contenido en Animation para el elemento o etiqueta 'menú' del HTML para asociar los efectos del Gate Animation al atributo menu_effects y a partir de ello creamos un MenuAutomata.

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
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()

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 estado actual solicitado.
- attend_get_menu_in: Guarda el estado getting_in de MenuAutomata como estado actual solicitado

5.2.4. Clase OptionHandler

Dispara los eventos que se producen en la clase Levels y se comunica mediante mensajes con la clase MenuHandler.

Al igual que MenuHandler, éste deriva de la clase Device y tiene diferentes posibles estados: 'get_panel_out', 'keep_menu_out', 'get_menu_in'. Éste crea diferentes Gate para cada botón level.

5.2.5. Clase Levels

Recoge los eventos que se generan en el Submenú. Deriva de la clase Gate y tiene como métodos do_onmouseover y do_onmouseout:

- do_onmouseover: Lanza como evento la creación de un nuevo mensaje de mantener el menu desplegado
- do_onmouseout: Oculta el elemento level_option_container, cambiando su display a none y lanza como evento un nuevo mensaje con el estado get_menu_in para ocultar el menu.

5.3. Reloj

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: Segundo que quedan por jugar. Queda inicializado a total_time.
- before: Almacena el en segundos el momento en el que se empezó a jugar.
- running: Hace referencia al estado el marcha del reloj. Inicializado a true.

La función Initialize llama constructor de Device.

Métodos de la clase Clock:

- 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).
- run: Recalcula el tiempo que queda por jugar. Vuelve a llamar 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 a partir del momento actual, a través de la función get_now. Recalcula remaining_time.

  lo que queda = lo que quedaba - lo que ha pasado.

  Establece el valor de running a true.
- 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 esta en marcha(running es falso), realizamos las mismas operaciones 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 y la llamada a la función finishedLoading, la cual se encargará de crear y reproducir los recursos (recogiéndolos del array que contiene las rutas). Ésta será ejecutada después de solicitar la ejecución del método load de bufferLoader.

6.2. Clase BufferLoader

Clase bufferLoader:

7. Conclusiones

Este proyecto nos ha permitido comprobar la importancia y las ventajas de la utilización de los lenguajes orientados a objetos. Éstos últimos permiten una programación más ágil y eficiente que aquellos 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.

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.

Índice de figuras

Índice de cuadros