Índice

1. Introducción

En el presente proyecto, se busca crear un entorno virtual de pruebas para poder analizar los comportamientos asociados a estados de ánimo en un conjunto de caracteres autónomos. Dichos comportamientos dependerán del nivel de estrés y este, a su vez, cambiará en función de lo que perciban a través de sus sentidos. Para dotar a la simulación de un mayor realismo se crearan fuentes de audio en tiempo real capaces de analizar y reproducir el audio cargado en ellas. La frecuencia resultante del análisis será lo que llegue a los caracteres autónomos mediante su sistema auditivo, permitiendo el estudio de los comportamientos en función de lo que perciban.

Los objetivos que se han marcado son:

- Recrear el sistema de vida artificial presentado por Craig Reynolds en su trabajo "Steering Behaviors For Autonomous Characters"
- Ser capaces de cargar varias fuentes de audio, para posteriormente ser analizadas y después reproducidas.
- Calcular la frecuencia fundamental de una señal de audio.
- Crear un sistema auditivo y de comunicación para caracteres autónomos.
- Diseñar una interface adecuada para poder realizar el mayor número de pruebas.
- Analizar el comportamiento de los caracteres en función de unos parámetros definidos por el usuario a través de la interface.

La memoria consta de cuatro capítulos dedicados cada uno a un aspecto del proyecto. En el primero se comenta de forma detallada los sistemas existentes más conocidos en el área de la aplicación.

En el siguiente, se detalla el proceso de análisis llevado a cabo para la realizacion del proyecto y el cumplimiento de los objetivos marcados al inicio del mismo.

En el capítulo cuatro se describen en profundidad la metodología de trabajo, la tecnología utilizada ,las clases principales del programa, el funcionamiento detallado de la aplicación y una descripción de la interface.

Finalmente, las conclusiones acerca de los resultados obtenidos en este trabajo y las futuras ampliaciones para poder continuar la labor detallada en la introducción.

2. Antecedentes

Vida artificial

Craig W. Reynolds, publica "Steering Behaviors For Autonomous Characters" en este trabajo se explica la forma de crear un modelo computacional para el movimiento coordinado de grupos de caracteres autónomos (llamados boids).

Estas entidades son capaces de moverse por su mundo de una manera improvisada y realista. Para ello el modelo tiene que respetar ciertas reglas que aseguren las condiciones reales del movimiento de caracteres coordinados. Siendo las dos reglas más importantes las siguientes: No puede existir una inteligencia superior al resto. Cada entidad se mueve de modo independiente según las reglas del modelo y su ponderación respectiva.

Gracias a estas reglas, y modificando los valores ponderables de los comportamientos la simulación se asemejara mucho a la conducta real de diferentes especies, desde bancos de peces a bandadas de aves. Existiendo siempre una cohesión para que se desplacen en manada, una alineación para que se dirijan en la misma dirección y una separación para evitar que choquen los unos con los otros.

Además, aunando diferentes comportamientos se pueden crear otros mas complejos que pueden dar lugar a situaciones donde las entidades sean capaces de evitar obstáculos del mundo o seguir un camino prefijado como una manada.

Algunas de las características más importantes del modelo son: la falta de predictibilidad en un lapso de tiempo considerable y la formación de comportamientos emergentes, es decir, que no estan planeados y que pueden sorprender al propio creador.

Audio

Ha habido varios intentos de crear una potente API de audio para la Web y así, hacer frente a algunas de las limitaciones que se han descrito anteriormente. Un ejemplo notable es el Audio Data API, que fue diseñado y prototipo en Mozilla Firefox. El enfoque de Mozilla comenzó con un elemento <audio>, y amplió su API de JavaScript con características adicionales. Esta API tiene un gráfico de audio limitado, y no se ha adoptado más allá de su primera aplicación. Se ha quedado en desuso en la actualidad en Firefox a favor de la Web Audio API [Figura 1].

En contraste con Audio Data API, la Web Audio API es un nuevo modelo de la marca, totalmente independiente de la etiqueta <audio>, aunque hay puntos de integración con otras API de Web. Se trata de una API de alto nivel JavaScript para la transformación y síntesis de audio en aplicaciones

web. El objetivo de esta API, es incluir capacidades que se encuentran en los motores de juego modernos, y algunas de las tareas de mezcla, tratamiento y filtrado que se encuentran en aplicaciones de producción de audio de escritorio modernos. El resultado es una API versátil que se puede utilizar en una variedad de tareas relacionadas con el audio y visualizaciones de síntesis de música muy avanzadas.

3. Análisis

3.1. El mundo

Siendo el objetivo principal del proyecto crear un banco de pruebas para la medición del estrés en manadas de entidades autónomas por medio del sistema auditivo, reproduciremos el modelo propuesto por Craig W. Reynolds en su trabajo "Steering Behaviors For Autonomous Characters" para la creación de dichos agentes autónomos.

Para que las entidades puedan existir es necesario crear previamente un espacio con ciertas características donde puedan cohabitar grupos de dichas entidades.

Para ello, se creará un mundo en dos dimensiones (x,y) con una perspectiva de planta para su representación gráfica. La coordenada (0,0) se situará en la esquina superior izquierda del canvas.

El mundo contará con un tiempo inicial y actual, esta magnitud medible es necesaria para el correcto funcionamiento del sistema. Con ella la velocidad máxima de ciclos de procesamiento dependerá del aumento del tiempo y no de la velocidad propia que pueda alcanzar el procesador principal del sistema. Esto es de suma importancia porque proporciona una independencia a la hora de ejecutar la aplicación en diferentes sistemas con diferentes capacidades de calculo.

Dispondrá de un largo y ancho para su representación gráfica pero en la práctica y a la hora de realizar cálculos no tendrá ninguna limitación de tamaño, esto permite una gran libertad a la hora de realizar diferentes comportamientos por las entidades, la representación gráfica solo es una ventana donde poder observar todo lo que ocurre en el sistema.

Se podrán marcar algunas reglas físicas para adecuar el mundo virtual al real o caracterizarlo de cierta manera. Se podrá acotar la velocidad máxima y la aceleración máxima que pueden alcanzar las entidades, entre otras cosas.

El mundo dispondrá de una lista de todas las entidades y objetos que se encuentren dentro del mismo. Gracias a esto, será más fácil el cálculo y programación de comportamientos, además de, para labores de optimización de la aplicación.

Los objetos u obstáculos se definirán con líneas rectas y se podrán formar figuras geométricas con ellas. Esto da la posibilidad desde crear recintos para acotar el espacio de movimiento a crear obstáculos para definir caminos o simplemente para que sean evitados.

3.2. Nanobot

Cada nanobot sera una entidad que percibe su entorno y que podrá ser capaz de actuar en consecuencia decidiendo que hacer por medio de unas reglas de logica propias del modelo. [Figura 1]

Gracias a esto cada nanobot del grupo poseera cierto nivel de inteligencia, sera capaz de actuar de manera autonoma y tener un nivel complejo de predictibilidad a largo plazo. Al no tratarse de un sistema ideal toda la informacion que le llegue sera acotada para simular un entorno real, es decir, tanto su sistema visual y auditivo seran finitos y acordes a los animales que representen.

3.2.1. Caracterísicas básicas

Las principales características de los Nanobots son:

- Cerebro: En el cerebro se gestionan todos los comportamientos. Se dispondra de una lista de la cual se podra elegir el comportamiento adecuado en cada momento para el correcto funcionamiento del modelo. Tambien se podran activar mas de un comportamiento dando lugar a otros comportamientos mas complejos e impredecibles. Sera capaz de saber cual es la aceleracion adecuada para poder variar la velocidad de manera correcta. Esto se realizara haciendo la media de todas las aceleraciones devueltas por los comportamientos y en funcion de esa aceleracion la velocidad de la entidad se adecuara a la situacion actual.

- Geo data: Se debe conocer en todo momento la posicion del nanobot, asi como, su velocidad y aceleraccion. Imprescindible para poder realizar todos los calculos de los comportamientos y de su representacion grafica. Estas tres magnitudes seran representadas por vectores de dos dimensiones al igual que el mundo.

- Masa: Es la cantidad de materia que posee cada entidad. Necesaria para calcular la inercia mecánica del movil.

- Visión: Define el ángulo y radio de vision. Con ello se podra calcular todas las entidades que son susceptibles a ser vistas.

- Dirección: Con ella se podrá determinar la dirección en la que mira el nanobot, será calculada como el vector unitario del vector velocidad.

- Límites de la fuerzas: Cada entidad poseerá unos límites físicos para la correcta simulación del modelo, se busca con estos límites dotar de realismo a la simulación [Figura]. Estos limites son:

- Aceleración: Sólo sera posible una aceleración máxima. Superada esa aceleración se volverá inmediatamente a la aceleración máxima.
- Giro: Determina el angulo máximo de giro.
- Frenado: Máxima fuerza de frenado que podrá ejercer el nanobot.

- Velocidad máxima: En todo momento se debe revisar el valor de la velocidad, si el módulo del vector velocidad es mayor que la velocidad máxima impuesta, esta se reducira inmediatamente.

- Nivel de estres: Se dispone de un sistema para medir el estres del sujeto. Con él, se podra definir un nivel de estres en funcion de todo lo que escuche, desde musica a otros nanobots.

- Distancia que recorre la onda de sonido: al poder producir sonido, se define una distancia maxima a la que el sonido producido puede llegar.

3.2.2. Visión

Cada entidad posee un campo de vision que esta determinado por un radio de vision y un angulo[Figura 2]. Estos dos valores podran ser parametrizados para observar diferentes reacciones y comportamientos.

Para la obtencion de las entidades locales que es capaz de visualizar se realizara una resta de los vectores de posicion desde el origen de coordenadas de cada integrante del mundo con respecto al observador, si la distancia es menor al radio de vision es un objeto o entidad susceptible a ser observada , para asegurarlo se realizara ademas una segunda comprobacion con respecto al angulo que forma con la otra entidad, si este es menor al angulo de vision es un objeto que es visualizado por la entidad.

3.2.3. Audición

Cada entidad poseerá un sistema auditivo que hará posible el sentido del oido, es decir, lo faculta para ser sensible a los sonidos. La funcion principal de esto sera transformar las ondas sonoras que se propagan por el aire en informacion que capta la entidad y transmite al cerebro para su procesamiento y posterior reacción.

3.2.4. Sistema de comportamientos

La principal regla en el sistema de comportamientos se basa en la ausencia de una inteligencia que domine a todas las entidades o sobresalga del grupo.

Lo contrario invalida el modelo computacional y lo hace carente de sentido para su estudio. Al tener todos los nanobots un nivel similar de inteligencia ayuda a la hora de calcular, estudiar y medir los patrones de bandadas que pueden llegar a formar con las diferentes ponderaciones que se pueden realizar a sus atributos basicos. Siendo este el motivo principal de estudio del presente proyecto.

Comportamientos de manada

Son las reglas básicas por las que entidades independientes se comportarán de manera similar a manadas de animales reales. Estas reglas son:

- Alineación: Se busca que todas las entidades tengan una direccion común. Para ello, se obtiene la velocidad de todas las entidades vecinas y se calcula su promedio, dicho promedio será la velocidad deseada [Figura 3].
- Cohesión: Las entidades se mantienen unidas, es decir, se busca que los nanobots no se separen con respecto a una distancia maxima. Para ello, se buscan las posiciones de las entidades vecinas y el promedio de esas posiciones sera el punto que seguira cada nanobot para mantener la unidad [Figura 4].
- Separación: Las entidades buscan una separación mínima con su vecinas. Por eso se busca la distancia de separacion de las entidades con respecto al observador. Mientras esa distancia sea grande la fuerza de repulsion sera pequeña y viceversa, a una distancia muy pequeña la fuerza sera muy grande [Figura 5].

Comportamientos individuales Se basan en acciones específicas para cada entidad en particular, son:

- Seguir: El nanobot buscará la última posición en la que se encuentre su objetivo. Se calculará una velocidad deseada que será el resultado de escalar el vector unitario de la distancia entre perseguidor y objetivo por la velocidad máxima. Para posteriormente calcular la aceleración deseada restando a la velocidad deseada la velocidad actual [Figura 6].
- Alejarse: Se trata del comportamiento inverso a seguir. Se realizará el mismo cálculo pero al resultado se le cambiara el signo [Figura 7].
- Perseguir: El nanobot buscará una futura posición en el tiempo en la que se encuentre su objetivo. Se calculará una velocidad deseada que

será el resultado de escalar el vector unitario de la distancia entre perseguidor y la posición futura del objetivo por la velocidad máxima. Para posteriormente calcular la aceleración deseada restando a la velocidad deseada la velocidad actual [Figura 8].

- Huir: Se trata del comportamiento inverso a perseguir. Se realizará el mismo cálculo pero al resultado se le cambiara el signo [Figura 9].
- Evitar obstaculos: Cada nanobot será capaz de evitar obstáculos que se encuentre en su camino. Para ello, se creará una línea imaginaria que es idéntica al vector velocidad que lleve en ese momento. Si esa línea imaginaria intersecta con cualquier línea que esté definida en el mundo para crear obstáculos, la aceleración deseada será el vector normal a la velocidad [Figura 10].
- Vagar: Permite a los nanobots vagar libremente por el mundo sin un rumbo fijo y sin dar cambios bruscos de dirección. Se calculará una velocidad deseada que será el resultado de escalar el vector unitario de la distancia entre el nanobot y un punto aleatorio por la velocidad máxima. Para posteriormente calcular la aceleración deseada restando a la velocidad deseada la velocidad actual [Figura 11].

3.2.5. Toma de decisiones

La aplicación de cada uno de los comportamientos nos indicará hacia donde se debe dirigir el nanobot, al darse la posibilidad de tener activados tantos comportamientos como se deseen o necesiten, la toma de decision que se realizará será la media de la suma de cada uno de los comportamientos, dando como resultado una aceleración vectorial que influirá en la velocidad de la entidad [Figura 12].

3.2.6. Comunicación oral

Cada entidad será capaz de comunicarse con sus semejantes en un determinado radio. Cada nanobot podrá mandar mensajes, siendo así el emisor, susceptibles a ser recibidos por todos los nanobot que se encuentre en el radio de la onda sonora, convirtiéndolos en los receptores del mensaje. Se debe diferenciar el receptor del emisor para que al responder no se produzca un efecto en cadena si otro está escuchando.

Existen dos tipos diferentes de comunicaciones:

- Comunicación simple: Donde solo se intercambian palabras para poder modificar de manera muy rápida el nivel de estrés del receptor. No hay posibilidad de una segunda réplica por parte del emisor.
- Comunicación ligada a una orden: Se trata de comunicar una orden directa a todas los nanobots que esten escuchando. Estas entidades la procesaran y la llevaran a cabo inmediatamente después.

De esta forma estarán capacitados para poder comunicarse de una manera simple.

3.2.7. Nivel de estrés

Cada nanobot poseera un nivel de estrés para reaccionar de acuerdo a el. Según el nivel de estrés ciertos valores basicos del nanobot cambiaran. Los valores susceptibles a ser modificados son: ángulo de visión, capacidad de giro, frenado, aceleración y velocidad máxima.

3.3. Speaker

Un speaker, o altavoz, es la simulación de un transductor electroacústico que se utiliza para la reproducción de sonido. Cada speaker es capaz de reproducir canciones de una lista de reproducción, estas canciones serán analizadas a la hora de ser reproducidas. Todas las entidades que escuchen al speaker recibirán la frecuencia del sonido y la interpretaran según sea conveniente.

Un speaker será capaz de estar, o no, encendido. Al encenderlo se reproducirá la canción que este actualmente seleccionada, existiendo además una opción para poder pasar a la siguiente canción.

También se podrá aumentar, o disminuir, el volumen general del speaker. Con ello variará, no sólo, el volumen de la canción, sino también el radio de alcance de la onda sonora. Con esto se podrá crear diferentes zonas de influencia de la música, con todas las posibilidades de experimentación que ello conlleva.

3.4. AudioContext

La idea de esta API de alto nivel, para poder procesar y sintetizar audio en aplicaciones Web se basa en el concepto de nodos de audio (AudioNode), que serán los elementos que se interconectarán entre sí para formar una cadena, cuyo resultado será el audio renderizado. Lo primero que deberemos definir es un contexto para estos nodos, donde se producirán todos los sonidos. Puede

soportar varias fuentes de audio, con lo que sólo necesitaremos un contexto por cada aplicación de audio.

La forma de crear el contexto sería así:

```
var context;
window.addEventListener('load', init, false);

function init() {
  try {
    window.AudioContext = window.AudioContext||
                          window.webkitAudioContext||
                          window.mozAudioContext||
                          window.oAudioContext||
                          window.msAudioContext;
    context = new AudioContext();
  } catch(e) {
    alert('Este navegador no soporta la API de audio');
  }
}
```

Esta instancia del contexto permite acceder a un montón de métodos para crear nodos de audio, o manejar las preferencias globales de audio. Se centrará la atención en los nodos de audio, ya que son elementos necesarios en la cadena anteriormente citada.

Web Audio API cuenta con ruteo modular: Se puede ir ruteando las señales, a través de arquitecturas simples o complejas, que pasen por envíos y retornos, por ejemplo a efectos, mezclas y submezclas, ...

Existen nodos de varios tipos:

- Nodos de origen: fuentes de sonido, buffers de audio, entradas de sonido en vivo, <audio>tags, osciladores, y procesadores JS.
- Nodos de modificación: como su propio nombre indica sirven para modificar el sonido: filtros, paneadores, procesadores JS, convoluciones,...
- Nodos de destino: las salidas de audio.

Por tanto, para generar una cadena de audio, se debe generar los orígenes, los modificadores, los destinos, y se conectarán todos a través del método connect(). De la misma forma se podrán desconectar con el método disconect (numeroPuertoSalida), y volverlos a conectar. En resumen: Se pueden crear las cadenas de forma dinámica.

```
// Creamos un nodo origen
var origen = context.createBufferSource();
// Creamos un nodo modificador en este caso un Gain node
var gain = context.createGain();
// Conectamos el origen con el modificador
origen.connect(gain);
// Conectamos el modificador con el destino por defecto del contexto
gain.connect(context.destination);

// desconectamos el origen
origen.disconnect(0);
// Desconectamos el modificador
gain.disconnect(0);
// Conectamos directamente el origen al destino por defecto del contexto
origen.connect(context.destination);
```

3.5. Cómo cargar sonidos

La Web Audio API utiliza un AudioBuffer para sonidos de duración media-corta. El enfoque básico es usar una llamada XMLHttpRequest para ir a buscar los archivos de sonido. El API es compatible con la carga de datos de los archivos de audio en múltiples formatos, como WAV, MP3, AAC, OGG y otros.

El siguiente fragmento muestra la carga una muestra de sonido:

```
var dogBarkingBuffer = null;
window.AudioContext = window.AudioContext || window.webkitAudioContext;
var context = new AudioContext();

function loadDogSound(url) {
 var request = new XMLHttpRequest();
 request.open('GET', url, true);
 request.responseType = 'arraybuffer';

 request.onload = function() {
   context.decodeAudioData(request.response, function(buffer) {
     dogBarkingBuffer = buffer;
   }, onError);
 }
 request.send();
```

```
}
```

Un buffer no son más que datos en memoria. En este caso, datos que representan un sonido. Los datos del archivo de audio son binarios (no de texto), por lo que establece la responseType de la solicitud de 'ArrayBuffer'.

Para obtener ese buffer
Una vez que los datos de archivo de audio han sido recibidos (sin decodificar), se puede decidir decodificarlos más adelante o inmediatamente, utilizando el método decodeAudioData. Este método toma el ArrayBuffer de datos de archivos de audio almacenados en request.response y los decodifica de forma asíncrona**. Cuando decodeAudioData () termina, proporciona los datos de audio PCM decodificados como un buffer de audio.

** El objeto XMLHttprequest, es prácticamente admitido por todos los navegadores modernos. Su objetivo es permitir a Javascript formular peticiones Http y enviarlas al servidor, de modo tradicional en las aplicaciones web, estas peticiones se hacen de modo sincrónico junto con un evento que inicia el usuario. El resultado es una página nueva o la actualización de la página que es servida desde el navegador. Pero usando XMLHttprequest, podemos hacer que la página haga tales llamadas asincrónicamente, permitiendo continuar usando la página sin la interrupción de un navegador que se está actualizando, o la carga de una página nueva o revisitada.

JavaScript es un lenguaje de subproceso único. Esto significa que, si se invoca un proceso de larga duración, se bloquea toda la ejecución hasta que el proceso se complete. Los elementos de la interfaz de usuario no responden, las animaciones se pausan y no puede ejecutarse ningún otro código en la aplicación. La solución a este problema es evitar la ejecución sincrónica todo lo que sea posible.

Una vez que se han cargado una o más AudioBuffers, entonces estamos listos para reproducir sonidos, creando lógicamente la fuente de sonido, y asignándole el buffer que se le haya cargado:

```
function playSound(buffer) {
  var source = context.createBufferSource();
  source.buffer = buffer;
  source.connect(context.destination);
  source.start(0);
}
```

Esta función playSound() podría ser llamada cada vez que alguien pulsa una tecla o se hace click con el ratón.

La función start(tiempo) hace que sea fácil programar la reproducción precisa del sonido. Sin embargo, hay que asegurase de que sus el buffer de sonido está pre-cargado.

3.6. Analizar el sonido

Para la resolución del análisis de audio y la obtención de la frecuencia fundamental de un audio en concreto, es necesario conocer algunos aspectos técnicos.

¿Qué es la Frecuencia de Muestreo?
Número de muestras por unidad de tiempo que se toman de una señal continua para producir una señal discreta, durante el proceso necesario para convertirla de analógica en digital[Figura 16].

Según el Teorema de Nyquist, para poder replicar con exactitud la forma de una onda es necesario que la frecuencia de muestreo sea superior al doble de la máxima frecuencia a muestrear. El aliasing se produce cuando la frecuencia de muestreo es inferior a la frecuencia Nyquist y por lo tanto insuficiente para hacer el muestreo correctamente con lo cual inventa frecuencias fantasmas que no tiene nada que ver con la original.

¿Qué es la Profundidad de bit?
Resolución de captura de una señal de audio en relación a la amplitud (volumen). La Profundidad de Bits determina el rango dinámico de una señal de audio, es decir, determina el máximo y mínimo de decibelios que una señal puede tener al ser grabada.

¿Qué es la Calidad/Fidelidad del sonido?
Para los sistemas de audio, esta medida va ha estar determinada por el número de bits por muestra y la tasa de muestreo[Figura 17].

¿Qué es la FFT o transformada de Fourier?
La transformada de Fourier se utiliza para conocer las características frecuenciales de las señales y el comportamiento de los sistemas lineales ante estas señales [Figura 18].

La interfaz AnalyserNode representa un nodo capaz de proporcionar la frecuencia en tiempo real y análisis de la información de dominio en el tiempo. Es un AudioNode que pasa el flujo de audio sin cambios desde la entrada a la salida. El nodo funciona incluso si la salida no está conectada.

Propiedades nodo AnalyserNode (hereda las propiedades del padre AudioNode):

AnalyserNode.fftSize

Es un número sin signo que representa el tamaño de la transformada rápida de Fourier que se utilizará para determinar el dominio de la frecuencia. Debe ser una potencia distinta de cero, en intervalos de a 2, entre 32 y 2.048. Su valor por defecto es 2048. Si no es una potencia de 2, o se encuentra fuera del rango especificado, saltaría una excepción.

AnalyserNode.frequencyBinCount (sólo lectura)

Contiene la mitad del tamaño de la FFT.

Métodos nodo AnalyserNode (hereda los métodos del padre AudioNode):

Analyser.getByteFrequencyData()

Copia los datos de frecuencia actual en la matriz. Si la matriz tiene menos elementos que el frequencyBinCount, el exceso de elementos se borra. Si tiene más elementos de los necesarios, se ignora el exceso de elementos.

Analyser.getByteTimeDomainData()

Copia la forma de onda actual, o el dominio del tiempo. Si la matriz tiene menos elementos que el fftSize, el exceso de elementos se borra. Si tiene más elementos de los necesarios, se ignora el exceso de elementos.

3.7. Flujo de trabajo de la aplicación

La aplicacion se encontrara dentro de un bucle infinito desde el comienzo de la misma hasta su final. El proceso será el siguiente:

1. Inicio de la iteración.
2. El mundo actualiza su tiempo y llama a cada nanobot.
3. Cada nanobot comprueba si recibe algún estímulo externo, si lo recibe almacena la información de dichos estímulos.
4. Se actualiza las variables según dichos estímulos para cada nanobot.
5. Cada nanobot realiza los cálculos para llevar a cabo los comportamientos activos según las variables actuales, para terminar devolviendo una nueva aceleración.
6. Se calcula la velocidad final para cada nanobot según la aceleración, para poder determinar la nueva posición.
7. Se pinta el mundo y todos los nanobots en pantalla según la posición recién calculada.

8. Termina la iteración y vuelve a empezar el bucle.

4. Diseño

4.1. Metodología de desarrollo

Para el desarrollo de la aplicacion se ha utilizado el desarrollo guiado por pruebas de software, o Test-driven development (TDD)[Figura 14]. Sus dos reglas principales son:

- Escribir las pruebas primero.
- Refactorización.

Para escribir las pruebas se utilizaron pruebas unitarias. En primer lugar, se escribe una prueba y se verifica que las pruebas fallan. A continuación, se implementa el código que hace que la prueba pase satisfactoriamente y seguidamente se refactoriza el código escrito. El propósito del desarrollo guiado por pruebas es lograr un código limpio que funcione. La idea es que los requisitos sean traducidos a pruebas, de este modo, cuando las pruebas pasen se garantizará que el software cumple con los requisitos que se han establecido.

Una ventaja de esta forma de programación es el evitar escribir código innecesario. Se intenta escribir el mínimo código posible, y si el código pasa una prueba aunque sepamos que es incorrecto nos da una idea de que tenemos que modificar nuestra lista de requerimientos agregando uno nuevo.

La generación de pruebas para cada funcionalidad hace que el programador confíe en el código escrito. Esto permite hacer modificaciones profundas del código pues sabemos que si luego logramos hacer pasar todas las pruebas tendremos un código que funcione correctamente.

Otra característica del desarrollo guiado por pruebas de software es que requiere que el programador primero haga fallar los casos de prueba. La idea es asegurarse de que los casos de prueba realmente funcionen y puedan recoger un error.

A pesar de los elevados requisitos iniciales de aplicar esta metodología, el desarrollo guiado por pruebas (TDD) puede proporcionar un gran valor añadido en la creación de software, produciendo aplicaciones de más calidad y en menos tiempo. Ofrece más que una simple validación del cumplimiento de los requisitos, también puede guiar el diseño de un programa. Centrándose en primer lugar en los casos de prueba uno debe imaginarse cómo los clientes utilizarán la funcionalidad (en este caso, los casos de prueba). Por lo tanto, al programador solo le importa la interfaz y no la implementación. Esta ventaja es similar al diseño por convenio pero se parece a él por los casos de prueba más que por las aserciones matemáticas.

4.2. Tecnología utilizada

Para la codificación de la aplicacion se ha utilizado el Framework de desarrollo Lluvia basado en el lenguaje de programación JavaScript.

Lluvia es una API Open Source que incorpora gran parte de las funciones nativas de Ruby. Soporta multihilo a pequeña escala y provee de un sistema de mensajería asíncrono gestionado por señales.

En Lluvia, los Devices son los encargados de proporcionar el mecanismo asincrono de comunicacion. Estan preparados para disparar eventos y recibir mensajes de otros Devices, estos mesajes se almacenan en una cola de mensajes para que sean procesados cuando llege su turno. A efectos practicos, cada Device se comporta como una aplicacion independiente del resto.

Para gestionar los eventos del DOM de HTML se utilizan los Gates de Lluvia que estan capacitados para mantener el campo de visibilidad del objeto.

La conjunción de los Devices y los Gates proporciona un metodo dinamico y agil para la creacion de la aplicación web.

4.3. Clases principales de la aplicación

4.3.1. World

La clase World deriva de Device y es la encargada de generar el mundo donde coexisten los nanobots y speakers. Es necesario indicarle el canvas(elemento HTML incorporado en HMTL5 que permite la generación de gráficos dinámicamente por medio del scripting) donde se realizará la representación gráfica y optativo el ancho y largo del mismo.

En la clase World se almacena un array con todos los nanobot y speakers que se hayan generado para su facil manipulacion. Para ir almacenandolos se llama al metodo has_born() que se encarga de almacenar el nanobot creado y disparar el evento new_boid.

Dispone de un método run() que es llamado cada cierto tiempo para su ejecución. En él, se actualiza el tiempo del procesador y se pinta el mundo.

Se dispone del método new_boid_of() para crear nuevos boids de la clase que se desee. Para la llamada del método es posible pasarle en los argumentos el tipo de entidad que se quiere crear(boid, nanobot o speaker) y un bloque. En el bloque se puede pasar una configuración inicial de la entidad para darle valores iniciales.

Los métodos visible_for() y audible_for() permiten comprobar que se puede ver o escuchar desde determinado lugar del mundo, estos métodos son llamados desde los nanobots o speakers para saber que pueden ver u oír desde su posición. Se consideró que este método era más adecuado integrarlo

en el mundo, en vez de en el nanobot o boid, para tener la posibilidad de saber que se puede ver en algún lugar del mundo sin la necesidad de que allí esté situado un nanobot.

La clase World es capaz de atender al evento focus_boid que se encarga de saber si un nanobot esta resaltado y poder dibujar en pantalla información adicional sobre su velocidad y aceleración.

Por último, dispone de los clásicos métodos para poder acceder de manera segura a distintas variables de la clase.

4.3.2. Boid

La clase Boid es clase padre tanto de Nanobot como de Speaker. Es la encargada de dotarlos de unas propiedades básicas idénticas para ambas clases hijas.

Un boid se puede inicializar con un objeto de configuración capaz de sobrescribir todos los valores por defecto de los atributos.

Se crea un objeto donde se almacena toda la información de la posicion, velocidad y aceleración del boid, este objeto se llama geo_data. Además, se dota al boid de una masa, color, visión y cerebro por defecto.

Se crea un puntero al mundo donde vivirá, especialmente útil para poder utilizar los metodos visible_for() y audible_for() de la clase World.

También se crean tres variables que almacenan las fuerzas limites de cada boids: thrust, steering, braking (aceleración, ángulo de giro y frenado).

Dispone de un metodo run() que es llamado cada interacción del programa para poder actualizar su aceleración en función de la media de todas las aceleraciones que devuelvan los comportamientos activos, además de, actualizar su tiempo actual.

El método clip() se encarga de adecuar la aceleración y velocidad actual para que se cumplan las fuerzas limite de cada boid. Si se pasa de aceleración o velocidad aqui se vuelve al estado anterior de dicha magnitud.

Por último, dispone de los clásicos métodos para poder acceder de manera segura a distintas variables de la clase.

4.3.3. Nanobot

La clase Nanobot deriva de la clase Boid y es la encargada de crear nuevos nanobot en el mundo.

Para su creación es necesario pasarle un bloque de configuración, que este a su vez pasará a la clase padre para asignarle los valores. Además, se crea un nivel de estrés, un radio para la longitud que alcanzara el habla del nanobot,

un array de frecuencias donde se almacenarán todas las frecuencias que vaya escuchando el nanobot y un array de objetos visibles.

El nanobot es capaz de comunicarse con otros por medio de su método talk(). Para realizar la comunicación se comprueba quien puede escuchar y a esos se les manda un mensaje con set_msg(). Dicho mensaje se trata de un objeto compuesto por la palabra que se quiere comunicar y un puntero al nanobot emisor. Los mensajes según llegan al receptor se guardan en un array de mensajes para poder leerlos de manera asíncrona y no perder información. A la hora de analizar dichos mensajes, en analyze_msg(), se comprueba cual es la palabra que se pasa en el mensaje y en función de su contenido se ejecuta unas instrucciones u otras.

Hay instrucciones que son órdenes que activan comportamientos directamente, para estas es necesario tener un objetivo para el comportamiento, para ello se utiliza el puntero al emisor. A la hora de crear la réplica se elegirá al azar de un array de réplicas que existen para cada caso.

El nanobot es capaz de recibir frecuencias, con el metodo set_frequency(), de una onda de sonido. En el método analyze_sound() se comprobara el valor de la frecuencia y en funcion de su valor aumentara o disminuirá el nivel de estrés del nanobot.

El método listen() encapsula todo lo relacionado con el sistema auditivo del nanobot.

4.3.4. Speaker

La clase Speaker deriva de la clase Boid y es la encargada de crear nuevos speakers en el mundo.

Para su creación es necesario pasarle un bloque de configuración, que este a su vez pasará a la clase padre para asignarle los valores. Ademas, se crea un radio para la longitud que tendrá la música que reproduzca el speaker, variables para saber la frecuencia a la que emite, conocer si esta apagado o encendido, así como, para conocer su volumen.

El speaker se podrá encender o apagar con los métodos on() y off(). Si esta apagado la música parará y no se reproducirá nada. Se podra aumentar y disminuir el volumen con los métodos voluemn_up() y volumen_down().

La frecuencia a la que emite se puede obtener a traves del metodo set_frequency_music() y se podrá mandar a quien escuche con get_frequency_music().

4.3.5. World interface

La clase World interface deriva de la clase Device y es la gestora de crear el dispositivo encargado de poder seleccionar a los nanobots para poder observar

sus características y opciones.

Es capaz de atender el mensaje new_boid que se encarga de añadir un nuevo nanobot o speaker al dispositivo para su selección.

4.3.6. Boid editor

La clase Boid editor deriva de Device y es la encargada de mostrar toda la informacion u opciones para el speaker o nanobot que este selecionado en el World_interface.

Si se trata de un nanobot mostrara la posicion actual, el comportamiento activo en ese momento y su nivel de estrés. Además, mostrará una serie de controles para el manejo independiente de cada uno de los boids.

Si el seleccionado es un speaker mostrará los controles para manipular el speaker, que son: boton de apagado y encendido, subir y bajar volumen, cambiar cancion y la canción que actualmente se está reproduciendo.

4.3.7. Brain

La clase Brain es la encargada de gestionar todos los comportamientos que puedan estar asociados a un nanobot y devolver la aceleración ponderada de todas las aceleraciones individuales devueltas por los comportamientos.

Es capaz de activar y desactivar comportamientos con los métodos activate() y deactivate(). Esto ayuda a crear comportamientos más complejos que los inicialmente propuesto gracias a la combinación de comportamientos simples.

También se puede preguntar al cerebro con can$U() si un comportamiento puede ser activado en un determinado nanobot.

4.3.8. Beahavior

La clase Behavior permite modelar el comportamiento de los boids. 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.

4.3.9. Separation

La clase Separation deriva de Behavior y es la encargada del comportamiento de separación entre nanobots.

Calcula los vectores de distancia entre todos los nanobots visibles y realiza su media para posteriormente dividir cada término del vector por un radio de aproximación mínimo. El vector opuesto a ese resultado será devuelto como aceleración vectorial deseada por el metodo desired_acceleration().

4.3.10. Cohesion

La clase Cohesion deriva de Behavior y es la encargada del comportamiento de cohesión entre nanobots.

Teniendo en cuenta el vector posicion de cada nanobot visible por el nanobot observador calcula su media. El vector resultante de dicha operación será devuelto como la aceleración vectorial deseada por el metodo desired_acceleration().

4.3.11. Alignment

La clase Alignment deriva de Behavior y es la encargada del comportamiento de alineamiento entre nanobots.

Teniendo en cuenta el vector de velocidad de cada nanobot vecino por el nanobot observador calcula su media y devuelve como aceleración deseada la proyección de esa media sobre la velocidad actual, es decir, se realiza una descomposición de fuerzas y se toma solo la que este en dirección a la velocidad actual.

4.3.12. Containment

La clase Containment deriva de Behavior y es la encargada del comportamiento de contención y evitado de obstaculos en el mundo.

Al inicio del programa se crea un array de líneas que serán las encargadas de poner límites al mundo y señalar los obstáculos. Además, cada nanobot posee una línea imaginaria que se corresponde con la velocidad que lleva en ese momento.

Se debe comprobar en cada iteración del programa si hay alguna intersección entre alguna línea del array y la línea de la velocidad. Si es así, se devolverá como aceleración deseada el vector normal a la velocidad escalado cien veces.

4.3.13. Seek

La clase Seek deriva de Behavior y es la encargada del comportamiento de seguimiento entre nanobots. En la llamada se le debe pasar el objetivo a seguir.

Primero se obtendrá la velocidad deseada, para ello se calculara el vector unitario del vector posicion que separa al objetivo del nanobot perseguidor, para que, a ese vector resultante escalarlo por la velocidad máxima.

Finalmente, el metodo desired_acceleration() devolvera como aceleración deseada la resta de la velocidad actual con la velocidad deseada.

4.3.14. Flee

La clase Flee deriva de Behavior y es la encargada del comportamiento de escapar entre nanobots. En la llamada se le debe pasar el objetivo del que se huye. Al tratarse del contrario de seek solo se debe cambiar de signo la velocidad deseada.

Primero se obtendrá la velocidad deseada, para ello se calculara el vector unitario del vector posicion que separa al objetivo del nanobot perseguidor, para que, a ese vector resultante escalarlo por la velocidadmáximaa, que estara con signo negativo.

Finalmente, el metodo desired_acceleration() devolvera como aceleración deseada la resta de la velocidad actual con la velocidad deseada.

4.3.15. Pursue

La clase Separation deriva de Behavior es la encargada del comportamiento de persecución entre nanobots. En la llamada se le debe pasar el objetivo a perseguir.

Inicialmente se calculara la posicion del objetivo en un tiempo futuro cercano. Para ello, se calculará la velocidad deseada que será el resultado de escalar el vector unitario de la distancia entre perseguidor y la posición futura del objetivo por la velocidad máxima. Para posteriormente calcular la aceleración deseada restando a la velocidad deseada la velocidad actual. Dicha aceleracion deseada la devolvera el método desired_acceleration().

4.3.16. Wander

La clase Separation deriva de Behavior es la encargada del comportamiento de vagar libremente por el mundo.

El método desired_acceleration() devolverá la aceleración deseada que será el resultado de restar la velocidad deseada a la velocidad actual. Pa-

ra calcular la velocidad deseada se escalará el vector unitario de la distancia entre el nanobot y un punto aleatorio por la velocidad máxima.

4.3.17. Line

La clase Line es la encargada de suministrar métodos para el cálculo entre rectas. Y es el pilar fundamental del comportamiento containment.

El método distance() calcula la distancia entre dos rectas por el método matemático del paralelogramo, intersects$U() comprueba si dos líneas se pueden cortar o son paralelas(al considerarse solo dos coordenadas no se tiene en cuenta si se pueden cruzar en el espacio)

Para averiguar el punto de corte entre dos rectas se hace uso del método get_intersection() que devuelve el punto de corte. Para saber si dos segmentos de una recta se cortan se utiliza intersects_segment$U()

4.3.18. Stringhline

La clase StraightLine deriva de Line y es la encargada de crear líneas rectas que se utilizan para crear obstáculos o barreras. Cada línea recta se define por un punto inicial y un vector director.

Con los métodos get_tanget() y get_normal() se obtiene un vector tangente o normal a la recta respectivamente. Para calcular una linea perpendicular a la recta se utiliza get_perpendicular().

4.3.19. WebAudio.js

Después de conocer cómo funciona el AudioContext, el nodo AnalyserNode, y una breves nociones de física del sonido, se pasa a describir la construcción del objeto WebAudio.js

Se debe tener clara la idea de que Javascript es un lenguaje dinámico interpretado por closures(cierres), que acepta programación orientada a objetos.

Para la creación del objeto WebAudio se usa la nomenclatura:

```
var WebAudio = (new function() { })
```

Se conoce como función anónima (TODO es un objeto, incluidas las funciones). Esto permitirá que no sea necesario instanciar para obtener un objeto WebAudio. Las posibilidad de usar funciones anónimas va de la mano con otros conceptos importantes de Javascript:

- Todo es un objeto: (incluidas las funciones), Permite asignar funciones a variables, y hacer referencia a ellas utilizando la variable, además permite pasar funciones como parámetros a otras funciones, y obtener funciones como resultados de la ejecución de una función.
- Closures: Gracias a los Closures se tiene la ventaja de poder enlazar funciones con variables de otros entornos de ejecución (diferentes ámbitos de variables).
- Objetos dinamicos: A la hora de modificar el comportamiento de un objeto en particular, podemos definir una función y asignarla inmediatamente a un método de un objeto, sobreescribiendo su comportamiento inicial, o extendiendo su interfaz para agregar comportamiento nuevo.

Teniendo todo esto presente y analizando las necesidades del actual proyecto se ideó WebAudio atendiendo a las siguientes consideraciones:

- El contexto de audio siempre va a ser el mismo.
- Sucede exactamente igual para la cadena de conexión, es decir, para la ruta entre nodos audiocontext –> gainNode –> analyser –> destination.
- Sería muy interesante poder contar con un objeto que englobado en un contexto y una cadena de conexión (variables del objeto), tuviese la capacidad de retornar la frecuencia de muestreo, o establecer el tamaño de la transformada de Fourier.
- No se necesitan diferentes objetos de tipo WebAudio en el presente proyecto.

Por todos estos motivos el código realizado es el siguiente:

```
var WebAudio = (new function () {

    var contextClass = (window.AudioContext ||
        window.webkitAudioContext ||
        window.mozAudioContext ||
        window.oAudioContext ||
        window.msAudioContext)

    // Se crea el audio context
```

```
    this.ctx = new contextClass()

/*-------------------------------------------------*/
    // Crear fuente de audio y nodos necesarios
    this.gainNode = this.ctx.createGain()
    this.analyser = this.ctx.createAnalyser()

    this.isPlaying = false
    this.isFinished = false

    this.freqs = new Uint8Array(this.analyser.frequencyBinCount)

/* ================================================================
                        CADENA DE CONEXIÓN
   ================================================================*/
    this.gainNode.connect(this.analyser)
    this.analyser.connect(this.ctx.destination)
/*================================================================*/

    this.gain = function(v){
    this.gainNode.gain.value = v
            }

    this.frequencyData = function () {
                return this.analyser.getByteFrequencyData(this.freqs)
            }

    this.frequencyBinCount = function () {
                return this.analyser.frequencyBinCount
            }
this.sampleFrequency = function () {
                return this.ctx.sampleRate
            }
})
```

Lo primero es asignar una función anónima a la variable WebAudio. De este modo la variable WebAudio es un objeto.

Se hace una instancia de contextClass creando un objeto ctx. Este objeto representa el AUDIOCONTEXT.

Posteriormente se generan los nodos necesarios, para posteriormente conectarlos en una cadena. El origen o la fuente de sonido, NO se creará aquí,

en el script WebAudio.js. Esta tarea se hará y se explicará más adelante, en la clase Songloader.js

Se asigna un array tipado (typed arrays) Uint8Array, representa una matriz de enteros sin signo de 8bits., a las variable freqs (frecuencias). Los arrays tipados de Javascript proporcionan un mecanismo para acceder a los datos binarios sin procesar de manera muy eficiente sin necesidad de conversiones.

Se crea la cadena de conexión: gainNode –> analyser –> destination Es decir, la ganancia de la fuente de sonido, se conecta al analizador de audio, y sale por los altavoces.

Por último, y gracias a los closures de Javascript, podemos acceder a variables que están fuera del ámbito local de las mismas, por tanto, se aprovecha esta ventaja. Se asignan funciones a diferentes variables, todas ellas necesarias para analizar y/o manipular el audio. De este modo si se hace lo siguiente:

```
WebAudio.frequencydata
```

Se accede al método(función) que tiene adherida el objeto WebAudio, en este caso, frequencyData, que retorna los datos referentes a las frecuencias.

La función gain establece la ganancia de la fuente de audio según el parámetro que se le pase. Mediante el método value, se puede acceder al valor de la ganancia. Se asigna este valor al parámetro 'v' que recibe la función gain.

La función frequencyBinCount retorna LA MITAD del valor de la FFT

La función sampleFrequency retorna la frecuencia de muestreo de la fuente de audio.

4.3.20. class_SongLoader.js

Como ya se explicó en el apartado 'Cargando sonidos' , el enfoque básico para tal propósito es usar una llamada XMLHttprequest. La idea de código era poder introducir en una clase esta llamada XMLHttpRequest, cada vez que se quisiera cargar una nueva canción. Para llevar a cabo este fin, era necesario crear la fuente/origen del sonido, una vez se hubiese decodificado el flujo de bytes.

Como atributos, la clase SongLoader posee context (contexto de audio), y callback (petición XMLHttpRequest).

Se sobrecarga el constructor pasándole al constructor como parámetros context y callback.

Mediante herencia prototípica la clase SongLoader posee el método playNewSong, que permitirá decodificar el flujo de bytes en un buffer de audio a fin de que el audioContext lo pueda usar.

Lo primero es instanciar la petición en un objeto que llamaremos request. La instancia provee a este objeto, un método open que permite hacer una petición GET al servidor, pasándole la canción que deseamos decodificar, e indicándole con la palabra 'true' que la petición que se desea hacer es asíncrona (ver apartados anteriores). Mediante el método responseType se indica el tipo de respuesta que se desea obtener, en este caso, un arraybuffer.

En la función de carga se añade al objeto loader el contexto de audio y a continuación el método decodeAudiodata, que pasa el flujo de bytes que son números en un buffer de audio reconocible por el audio context. decodeAudioData requiere de tres argumentos:

- resquest.response
- Una función en caso de haya ido todo bien, que recibiría el 'buffer'.
- Una función que nos indique 'error'

La función que recibe el buffer se encarga de crear la fuente de audio. Se asigna esa fuente de audio al buffer que se haya cargado. Lógicamente se conecta al primer nodo de la cadena de conexión que se hizo previamente, en este caso un gainNode. Mediante el método start(), la fuente de sonido comenzaría a sonar.

Por último se envía la petición.

```
 /* ============================================================
                          CLASE SONGLOADER
    ============================================================ */

//  ----------------------- Constructor -----------------------

/* Se sobrecarga el constructor con el audioContext,
   y la llamada XMLHttpRequest */

function SongLoader(context, callback) {
  this.context = context
  this.onload = callback
}

SongLoader.prototype.playNewSong = function(track) {
  var loader  = this
  // crear objeto
  var request = new XMLHttpRequest()
```

```
  /* Petición GET. Al decir "true",
     la carga del buffer se hace de
     forma asíncrona. */
  request.open("GET", track, true)
  /* Indicarle (responseType) el tipo de respuesta
     que queremos obtener.
     'arraybuffer': Conjunto de bytes que no se
                    pueden modificar(solo lectura) */
  request.responseType = "arraybuffer"
  request.onload = function() {
  /* 'decodeAudioData': Pasa del flujo de bytes, que son
      números y no los podemos interpretar, en un buffer
      de audio a fin de que nuestro audioContext lo pueda
      usar. Requiere de tres argumentos:
            - resquest.response
            - Una función en caso de haya ido todo bien,
              que recibiría el 'buffer'.
            - Una función que nos indique 'error' . */
      loader.context.decodeAudioData(request.response,
function(buffer) {
          if (!buffer){
             alert('Error al decodificar los datos del archivo: ' + path)
             return;
          }
var source = WebAudio.ctx.createBufferSource()
WebAudio.source = source
          source.buffer = buffer
source.connect(WebAudio.gainNode)
source.start(0)
WebAudio.isPlaying = true
loader.onload()
      },
      function(error) {
 console.error('Error en la decodificación de datos de audio', error)
      }
    )
  }

  request.onerror = function() {
     alert('Error en la petición al cargador')
  }
```

4.3.21. requestAnimFrame.js

Si se desea construir una visualización para una forma de onda determinada, se necesita consultar periódicamente el analizador, procesar los resultados, y ejecutarlos. Se pude realizar esta tarea mediante la creación de un temporizador de JavaScript como setInterval o setTimeout, pero hay una mejor manera: requestAnimationFrame. Es una gran mejora en el rendimiento, al llamar al fotograma de animación cuando el sistema está preparado para dibujarlo.

En el pasado, para crear temporizadores en general, se utilizaba la función setTimeout. Esta función ha servido desde hace muchos años para toda clase de acciones por temporizador para las páginas web, sin embargo, no fue contemplada para animaciones, que requiere múltiples llamadas por segundo, consumiendo muchos recursos de nuestra computadora, aun si no estamos haciendo uso de la aplicación en cuestión.

Las compañías desarrolladoras de navegadores web han estado consciente de ello, y por tanto han ideado una mejor solución para esta tarea: la función requestAnimationFrame.

Esta función optimiza el uso de información, actualizándose de forma automática tan pronto el CPU le permite (Comúnmente, 60 cuadros por segundo en computadoras de escritorio), mejorando la capacidad del manejo de información en animaciones, consumiendo menos recursos, e incluso mandando a dormir el ciclo cuando la aplicación deja de tener enfoque, dando como resultado, un mejor manejo de las animaciones.

En el tiempo que esta memoria está siendo escrita, requestAnimationFrame es una función relativamente nueva, por lo que los navegadores que no estén actualizados podrían no soportarla, o usar una función experimental no-estándar de ella. Para saber más sobre la versión cuando esta función fue implementada, visitar http://caniuse.com/requestanimationframe , dónde muestran el avance de su soporte.

Para poder usar requestAnimationFrame en estos navegadores antiguos, existen muchas soluciones posibles. La más simple y popular, es agregar esta función al código:

```
var requestAnimFrame = (function(){
   return  window.requestAnimationFrame       ||
       window.webkitRequestAnimationFrame ||
       window.mozRequestAnimationFrame    ||
       window.oRequestAnimationFrame      ||
       window.msRequestAnimationFrame     ||
       function( callback ){
```

```
            window.setTimeout(callback, 1000 / 60);
        };
})();
```

4.3.22. class_Speaker.js

En un principio la clase Speaker.js era en realidad un programa principal llamado main.js que integraba todas las clases y ejecutaba el código en conjunto.

La construcción de la clase Speaker surge de la necesidad de introducir todo el código referente al audio en clases, para facilitar la labor de integración con los boids. La idea es aprovechar el objeto WebAudio introducido en un contexto de audio y que tiene una cadena de conexión determinada, junto con la clase SongLoader que permitirá decodificar el flujo de bytes en un buffer de audio. De esta manera la clase Speaker contiene todo lo necesario, para que se pueda reproducir el sonido decodificado, pintar el analizador de espectros y lógicamente RETORNAR EL VALOR de la frecuencia fundamental del audio que esté sonando en ese momento.

Lo primero es diseñar el constructor Speaker.

Inmediatamente después se introduce en la clase, el primer método llamado 'init_setup' que permitirá ubicar todos los componentes de la aplicación, y elegir correctamente los elementos del DOM a través de su identificador (getElementById).

La variable SPACING es la distancia entre las barras dibujadas en el analizador.

La variable OFFSET, se comporta como un filtro anti-aliasing, es decir, los errores producidos por una frecuencia de muestreo por debajo de Nyquist, generan que las frecuencias muy altas se "dupliquen" en frecuencias bajas en el espectro. Estas frecuencias bajas (que en realidad no existen, son copias periódicas de altas frecuencias), no se desean pintar, por lo que OFFSET definido en 100, equivale a decir que frecuencias menores de 100Hz no sean pintadas. De este modo aseguramos que el analizador no está pintando frecuencias "fantasma" no deseadas.

Canvas (lienzo en inglés) es un elemento HTML incorporado en HTML5 que permite la generación de gráficos dinámicamente por medio del scripting. Permite generar gráficos estáticos y animaciones. Se hará uso de este elemento y se introducirá en su contexto mediante getContext('2d'). También se definirá el alto (HEIGHT) y ancho (WIDTH) de este lienzo o canvas.

Por último, se hace una instancia de la clase SongLoader, y se crea el objeto songLoader

Dentro de la clase Speaker se encuentra la función pintar 'draw'. Es quizás

la función más compleja de entender. Antes de comentar como funciona, es importante recordar que para animar un dibujo es necesario llamar a un temporizador que en este caso es requestAnimFrame al que se le va a pasar como argumento la misma función draw. De este modo se pintará una y otra vez.

Se define el estilo de relleno y el tamaño de las barras analizadoras. Se recorren con un bucle for las frecuencias resultantes, y se pintan las barras.

La frecuencia fundamental es la frecuencia más baja del espectro de frecuencias tal que las frecuencias dominantes pueden expresarse como múltiplos de esta frecuencia fundamental.

Por tanto si se divide por un lado la celda del array que contiene esta frecuencia máxima, entre la longitud del array de frecuencias(results.length), y todo esto se multiplica por la frecuencia de muestreo/2 (nyquist), se puede hallar la frecuencia máxima de una canción determinada.

Los métodos playSound y stopSound, como su propio nombre indica, se encargan de la reproducción y parada de una canción determinada.

El método playSound es el encargado de llamar a la función draw para que cada vez que se pretenda reproducir una canción, automáticamente se pinte su análisis espectral de frecuencias en el canvas.

Como se comentó en apartados anteriores, en el diseño se tuvo en cuenta la posibilidad de que el usuario pueda elegir una canción diferente en plena reproducción, que esta nueva canción se cargase en el buffer, y pudiese ser reproducida a tiempo real, sin necesidad de recargar el navegador nuevamente.

Es necesario indicar mediante una sentencia condicional, que si el audio esta reproduciéndose, y se desea cargar una nueva canción en plena reproducción, pare la anterior fuente de sonido, y que suene exclusivamente la nueva elección del usuario. Para ello, se llama al método stopSound().

Los métodos playSound y stopSound son 'disparados' mediante el evento onclick al pinchar el usuario sobre los dos botones existentes en el archivo index.html

El método changeVolumen, como su propio nombre indica, se encarga de fijar la ganancia de la fuente de sonido que se esté reproduciendo en ese momento, según el parámetro que se le pase.

Simplemente se llama al objeto WebAudio y a su método obtener los datos de las frecuencias. Una vez teniendo esta información, sólo falta llamar al método gain del objeto WebAudio y pasarle el volumen establecido por el usuario como argumento.

Se construye con la etiqueta ¡input¿un deslizador (type = 'range'), que representará los valores que puede tomar la ganancia que en este caso oscilará entre 0 y 100. Mediante el evento onchange es posible que el usuario

pueda cambiar la ganancia de la fuente, arrastrando con el ratón en el deslizador. Solo hay que asegurarse de que como parámetro changeVolumen recibe este nuevo valor dividido entre 100.

```
 //CONSTRUCTOR

function Speaker() {

}

//MÉTODOS

Speaker.prototype.init_setup = function() {
   this.loader = document.getElementById('loader')
   this.cvs = document.getElementById('canvas')
   this.ff = document.getElementById('ff')
   this.WIDTH = this.cvs.width
   this.HEIGHT = this.cvs.height
   this.SPACING = 5
   this.OFFSET = 100
   this.drawContext = this.cvs.getContext('2d')
   this.tracklist = document.getElementById("lista")
   this.songLoader = new SongLoader(WebAudio.ctx, function(){})

}

draw = function() {
   requestAnimFrame(draw)
   WebAudio.frequencyData()
   var results = WebAudio.freqs
   var cvs = document.getElementById('canvas')

   var drawContext = cvs.getContext('2d')
   var ff = document.getElementById('ff')
   var WIDTH = cvs.width
   var HEIGHT = cvs.height
   var SPACING = 5
   var OFFSET = 100
   var max_cell = OFFSET
   drawContext.clearRect(0, 0, WIDTH, HEIGHT)
   drawContext.fillStyle ='#00FFCC'
```

```
   for(var i = 0; i < results.length-OFFSET; i++){
var magnitude = results[i + OFFSET]
drawContext.fillRect(i * SPACING, HEIGHT,\\
                     SPACING/2, -magnitude)
if ( results[max_cell] < results[ i + OFFSET] )
max_cell = i + OFFSET
   }

   drawContext.fillStyle ='#CCFF00'
   drawContext.fillRect((max_cell - OFFSET) * SPACING,\\
                        HEIGHT, SPACING/2, -results[max_cell])

   // VALOR DE LA FRECUENCIA FUNDAMENTAL
   if(WebAudio.isFinished)
ff.innerHTML = 0 + " Hz"
   else
ff.innerHTML = max_cell / results.length * \\
           (WebAudio.sampleFrequency() / 2) + " Hz"
}

Speaker.prototype.playSound = function() {
   this.songLoader.playNewSong(this.tracklist.value)
   if(WebAudio.isPlaying)
this.stopSound()
   WebAudio.isPlaying = true
   WebAudio.isFinished = false
   draw()
}



Speaker.prototype.stopSound = function() {
   WebAudio.isFinished = true
   WebAudio.isPlaying = false
   WebAudio.source.stop()
}



Speaker.prototype.changeVolumen = function(_volumen) {
```

```
    WebAudio.frequencyData()
    WebAudio.gain(_volumen)
}
```

4.4. Funcionamiento detallado de la aplicación

La aplicación comienza generando un número determinado de nanobots y speakers en el mundo con una velocidad aleatoria y los comportamientos de manada activados.

Al seleccionar un nanobot se despliega en el device boid editor la información más relevante del nanobot: nivel de estrés, comportamiento activado y posicion. Tambien se muestra los controles del nanobot:

- Habla: Listado de palabras para la comunicación entre nanobot.
- Órdenes: Conjunto de órdenes que un nanobot puede dar a otros nanobots.
- Juegos: Repertorio de juegos para los nanobots.

Al seleccionar un speaker se mostrará en el device boid editor los controles para el funcionamiento del speaker:

- Comenzar: Comienza a reproducirse la canción seleccionada.
- Parar: Para la reproducción de la canción en curso.
- Siguiente canción: Cambia la cancion que esta seleccionada por la siguiente en la lista.
- Subir volumen: Sube el volumen de la música y aumenta el radio de emisión del speaker.
- Bajar volumen: Baja el volumen de la música y reduce el radio de emisión del speaker.

Con todos estos controles se podrá realizar el estudio del comportamiento de los nanobots en entornos con tasas de estrés variable.

4.5. Descripción de la interface

El diseño general de la aplicacion se ha basado en una tarjeta de circuito impreso donde cada elemento simula estar conectado por pistas del material conductor de la tarjeta.

Los dispositivos world interface, boid editor y sound simulan pantallas LCD monocromo tipicas de las tarjetas electronicas. El tamaño de la pantalla del world interface se adapta a la cantidad de nanobots que estén en el mundo, la del boid editor es de tamaño fijo pero imita el comportamiento de una pantalla tactil,por ultimo, la pantalla de sound tambien es de tamaño fijo y muestra el ecualizador de sonido.

Los dispositivo zoom y world simulan pantallas LCD a color, de tamaño fijo y no reaccionan si se seleccionan.

La disposición general de los elementos da prioridad al mundo que se coloca en el centro de la aplicacion con el tamaño mas grande de todos los dispositivos. Lo flanquean el resto de los elementos resultando un conjunto proporcionado.

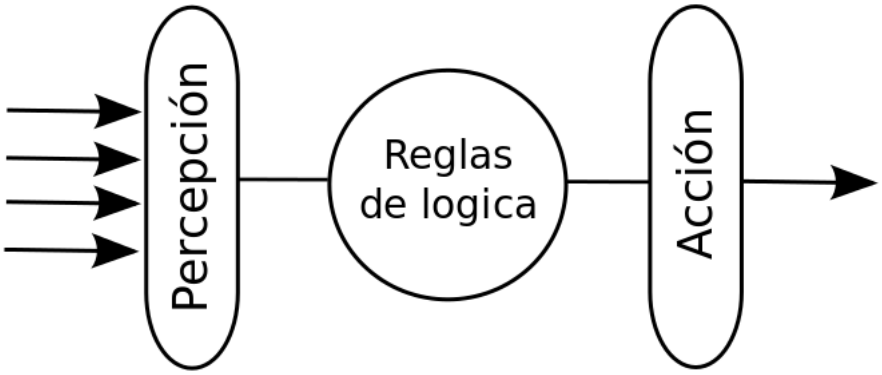

Figura 1: epepep

Manual de Usuario

A. El Interfaz