MOVIMIENTOS AUTÓNOMOS EN JUEGOS

Una implementación en Phaser 3

Inteligencia Artificial en Juegos

21 de octubre de 2018

Bonet Peinado, Daiana
FAI - 238
daiana.bonet@fi.uncoma.edu.ar

de la Fuente, Juan
FAI-524
juan.delafuente@fi.uncoma.edu.ar

Universidad Nacional del Comahue
Facultad de Informática

21 de octubre de 2018

Índice general

INTRODUCCIÓN

En el siguiente trabajo, se detalla el diseño e implementación de un Juego capaz de correr en un entorno web, con el objetivo de introducirse en el lenguaje de programación "JavaScript" e implementar movimientos autónomos de distintos agentes del juego.

Para alcanzar los objetivos se utiliza el *framework* Phaser en su ultima versión a la fecha (*Phaser 3*) que brinda facilidades para la programación de juegos web.

El trabajo se compone de una sección principal de Diseño, que detalla en profundidad la historia y las reglas que definen la estructura y funcionalidad del videojuego. Una sección de Implementación que define cada uno de los movimientos utilizados por los agentes autónomos, en su versión teórica y la definición de código que corresponde a su funcionalidad y finaliza con un apartado de posibles mejoras futuras.

OBJETIVO

Este desarrollo se encuadra en la materia optativa *Inteligencia Artificial para Juegos* de la carrera *Licenciatura En Ciencias de la Computación*.

Se espera que el resultado logrado conforme una breve demostración de los movimientos posibles para implementar en diversos Juegos y agentes autónomos y pueda ser reutilizado y analizado para lograr un aprendizaje sobre la temática de la materia.

Todo lo expuesto en este informe y la implementación que acompaña se encuentra bajo licencia *GNU GPL V3* disponible en el siguiente repositorio:

`https://github.com/jmdelafuente22/IAJ`

Es deseo de los autores seguir trabajando sobre este proyecto en los próximos meses.[1]

[1] *Ainulindale* puede ser jugado de forma on-line en el siguiente enlace `jmdelafuente22.github.io/IAJ`

Iluvatar:- ¿y si pudieses ser parte del Todo?
Iluvatar:- ..¿si pudieses ocupar la Nada?..
Iluvatar:- entonces que comience La Canción.

DISEÑO DEL JUEGO

Descripción del Juego

Ainulindale es un juego de tipo *Arcade* en el cual cada jugador controla una voz que canta llamada *Ainur*. El destino de cada *Ainur* es encontrar la melodía que le es propia en la gran canción del dios *Iluvatar* de la cual surge el universo y la tierra *Ëa*.

A continuación se especifica en profundidad cada componente del juego.

Nombre

Ainulindale que significa *La Gran Canción*.

Historia

La historia sucede en el origen del Universo[2], durante la Gran Canción que da origen a todo. Tu deber como *"Ainur"* (voz que canta) es recolectar los tiempos y sonidos que permiten crear la canción sin caer en los tropiezos de los silencios y otros sonidos que no debes cantar.

Objetivo

El objetivo del juego es capturar todas las notas que sean posibles para completar la canción sin que ningún obstáculo te golpee en un mundo de movimiento vertical descendente.

Reglas

A medida que vayas componiendo compases, podrás elegir combinar distintos elementos que hayas encontrado controlando más voces en simultaneo y compitiendo con otros Ainur.

Cada compás termina al alcanzar la cantidad de sonidos que se necesitan para ese nivel.

Podrás moverte dentro de un área restringida del escenario que va pasando de arriba hacia abajo.

[2] Según es narrado en la obra *"El Silmarillion"* de *J.R.R. Tolkien*.

1. El juego es continuo, sin turnos ni interrupciones y puede ser tanto de un sólo jugador (o "*Ainur*") o de múltiples jugadores.
2. El mundo está compuesto por todos los elementos observables en la pantalla. Existen elementos que transitan desde el borde superior hacía el borde inferior y elementos que "flotan" libremente por el mundo.
3. El movimiento del personaje principal se encuentra restringido dentro del radio demarcado por el circulo de color amarillo.
4. El personaje principal tiene movimiento indirecto dirigido, es decir, se mueve persiguiendo el puntero amarillo que mueve el Jugador.
5. Cada vez que el personaje principal (*Ainur*) captura una nota, ésta se convierte en la música de fondo y prolonga la melodía que va sonando.
6. Si el personaje principal toca un obstáculo vertical descendente, es eliminado y finaliza el juego.
7. Existen tantas notas cómo duración de la melodía, por lo que si una no puede ser recolectada, la melodía queda incompleta.
8. Cada nota recorre el camino demarcado 2 veces y luego desaparece.
9. El camino de cada nota se genera de forma aleatoria y con una cadencia aleatoria.
10. No se deben recoger las notas rojas ¡Esa melodía no has de cantar!

Finalización:

- El jugador "*Ainur*" completa la melodía, es decir, captura todas las notas.
- Algún obstáculo golpea al jugador.
- Si el jugador principal recoge una nota roja.

Representación

Para la representación del juego se decidió hacer un mundo con gráficos 2D donde los *Ainur* están representados a través de una esfera de movimientos circulares que es la encargada de recolectar todas *las notas* que se generan en distintas partes del mundo con

el fin de crear *la buena tierra de Eä*. Cada nota que se agarre contiene un fragmento de la canción a formar, y de esta manera se simula el canto de los Ainur.

En el mundo existen distracciones que debes evitar para poder crear la tierra. Éstas están representadas a través de barras que caen en dirección a las voces que cantan y por notas diferentes a las que debes cantar. ¡Si te distraes quedas fuera de la canción!

IMPLEMENTACIÓN

A continuación se describe la implementación lograda para el diseño del juego *Ainundale*. Como ya se ha detallado anteriormente, todo el desarrollo ha sido escrito en lenguaje *JavaScript* utilizando el *Framework Phaser* para tecnologías Web.

Representación

Para dotar de funcionalidad básica a un juego desarrollado en *Phaser* es necesario definir la representación básica del tipo de juego, el mundo en que se desarrolla y el motor de físicas principales del mismo.

Todo esto se especifica en un conjunto de parámetros dentro de las *configuraciones* del juego, antes de la *creación de la instancia* propia en que corre el juego.

Dentro de las numerosas configuraciones que permite el *framework*, fueron utilizadas las siguientes:

- Alto y Ancho del juego.
- Color de Fondo.
- Orientación.
- Título.
- Configuración de la física del juego (tipo de juego, gravedad, bordes del mundo, entre otras).
- Personalización y modificación de colores y pixeles.
- Sonido y Música.
- Escenas (diferentes pantallas o secciones diferenciadas).

```
var config = {
    type: Phaser.AUTO,
    width: 800,
    height: 600,
    backgroundColor:'2D2D2D',
    orientation: 'PORTRAIT',
    title: 'Ainulindale: La gran Canción',
    physics: {
      default: 'arcade',
      arcade: {
        debug: false
      }
    },
    pixelArt: true,
    audio: {
        disableWebAudio: true
    },
  };
```

Listing 1: Configuraciones de Phaser para el juego

Todas las opciones detalladas anteriormente forman parte de la implementación del juego, en partícular se ha hecho foco en las opciones de *Physics* para detallar que el juego es de tipo *Arcade*, *PixelArt* que representan la posibilidad de editar imagenes de forma dinámica y *Audio* que aporta todo lo referente a sonido y música del mundo.

En específico, también se buscó crear diversos escenarios entre las configuraciones, tema tratado en la sección *Posibles Mejoras*.

Es necesario detallar que *Phaser* implementa una serie de funciones para *construir* y controlar la temporalidad del juego: *preload*, *create*, *update* y *render*.

Cada una de estas funciones estructurales se ejecuta en un momento determinado del juego para generar los *tiempos* de vida del mismo. Todo lo implementado en la función *preload* es ejecutado como primera instancia, antes de la *creación* del juego y permite precargar recursos necesarios para la ejecución (como lo son: la música, los archivos javascript y las imágenes de gran tamaño). La función *create* es ejecutada como sinónimo de *constructor* del juego y ejecuta todo lo que debe suceder al momento del inicio del mismo (inicialización de estructuras, creación de personajes, recursos, etc.). En la función *update* se define todo lo que genera el *paso a paso* o *step* de continuidad del juego, es decir, todo aquello que sucede *durante toda la ejecución*. Esto permite que existan todos los estados que dan continuidad al mismo (movimientos, captura de pulsaciones y teclas, cambios de estado, entre otros).

Finalmente todo lo definido en *render* es definido en la creación de los gráficos y renderizado del juego.

Bajo estas estructuras se define la lógica del juego en cuestión y las particularidades del diseño, simplificando y generando un nivel superior de abstracción al permitir trabajar únicamente sobre el propósito del juego y no sobre la estructura del mismo.

Es necesario aclarar que en la implementación lograda existen numerosas funciones referidas a movimientos no inteligentes (desplazamiento del fondo con la funcionalidad de *tweens*, seguimiento de linea para el movimiento vertical de las notas a través de *followers* y movimiento de los obstáculos) y efectos gráficos (cambios de color en notas, personaje principal y lineas de forma dinámica) que escapan de la profundidad del informe pero que representan una gran parte del código del juego.

ALGORITMOS DE MOVIMIENTOS AUTÓNOMOS

En la siguiente sección se detalla la implementación lograda para cada uno de los movimientos particulares que presenta la implementación de *Ainulindale*. En particular el trabajo se baso en los denominados *Steering behaviors*.

Éstos movimientos ayudan a los agentes autónomos a moverse de forma realista, utilizando fuerzas simples que son combinadas para producir una navegación por el mundo que simule ser improvisada y natural. Estos movimientos no son basados en estrategias complejas, cálculos globales ni *path finding*, simplemente utilizando información local, tal como la fuerza de movimiento de los vecinos o de otros personajes. Esto permite que los movimientos sean sencillos de entender e implementar sin perder la capacidad de lograr patrones de movimiento complejos

ARRIVAL

En general, cualquier comportamiento de búsqueda -y en particular el movimiento *Seek*- hace que un personaje se mueva hacia un objetivo. Cuando alcanza el destino, la fuerza de direccionamiento sigue actuando sobre ella basándose en las mismas reglas, haciendo que el personaje “rebote” de un lado a otro alrededor del objetivo, obteniendo como resultado, un movimiento torpe y poco realista.

El comportamiento del movimiento *Arrival* busca evitar que el personaje se mueva a través del destino o que tenga una fuerza desproporcionada al momento de encontrarse con el objetivo, para esto, hace que el personaje se ralentice a medida que se acerca al destino, eventualmente deteniéndose junto al objetivo.

El comportamiento se compone de dos fases. La primera fase es cuando el personaje está muy lejos del objetivo y funciona exactamente de la misma manera que lo hace el movimiento *Seek*, es decir,se mueve a toda velocidad hacia el objetivo.

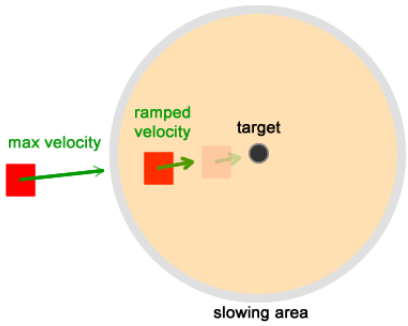

Figura 1: Representación del movimiento *Arrive*

La segunda fase es cuando el personaje está cerca del objetivo, dentro del "área de desaceleración" que es conformada como un círculo concéntrico en la posición del objetivo. Cuando el personaje entra en el círculo, comienza el movimiento *Arrival*, que ralentiza la aceleración del personaje de forma proporcinal a la cercanía al objetivo, de forma tal que al llegar al destino su aceleración sea nula.

```
function calcularVelocidadDeseada(personaje,objetivo) {
     // Calculo el vector deseado normalizado
     //(POSICION TARGET - POSICION PJ) * maximaVelocidad

     var VectorDeseado=new Phaser.Math.Vector2(objetivo.x,objetivo.y);
     VectorDeseado.subtract(personaje);
     //Calculamos la distancia para saber si esta cerca
     distancia=VectorDeseado.length();
     VectorDeseado.normalize();
     VectorDeseado.multiply(new Phaser.Math.Vector2(MAX_SPEED, MAX_SPEED));
     //Si se encuentra dentro del radio definido: Arrive
     if(distancia<MAX_DIST){
         valor=distancia / MAX_DIST;
         VectorDeseado.multiply(new Phaser.Math.Vector2(valor, valor));
     }

    return VectorDeseado;
 }
```

Listing 2: Cálculo de velocidad para *Seek-Arrival*

Uso en el juego

En *Ainulindale* podemos encontrar el movimiento *Arrival* en el comportamiento del *Ainur*, es decir, del personaje principal que se mueve dentro del circulo demarcado en amarillo. Este personaje tiene como destino permanente el puntero amarillo que controla el jugador.

WANDER

Con frecuencia podemos encontrar en los juegos personajes que parecen moverse de forma aleatoria por el ambiente o mundo. En general, estos personajes están esperando un acontecimiento o una acción en particular (por ejemplo, esperan un enemigo o la aparición del personaje en su área de acción) o se encuentran revisando o buscando algo.

Cuando el jugador es capaz de ver este comportamiento, se espera que el movimiento sea visualmente placentero y suficientemente realista.

Si el jugador es capaz de identificar un camino o lineas concretas, el movimiento pierde realismo e incluso puede generar frustración ya que puede suceder que los movimientos se vuelvan predecibles.

El comportamiento del *wander steering* busca producir un movimiento "casual" realista, que permita al jugador pensar que el personaje se encuentra realmente vivo.

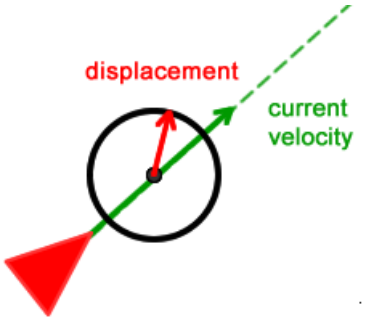

Figura 2: Representación del movimiento *Wander*

La idea básica de la implementación, es producir un pequeño desplazamiento aleatorio y aplicarlo sobre el vector de dirección actual del personaje. Ya que la velocidad del vector define la dirección del movimiento del personaje y cuanto se mueve en cada *frame*, cualquier interferencia, por pequeña que sea, cambia su ruta actual. Esto se puede apreciar en la figura 2.

Si se utiliza un pequeño desplazamiento en cada *frame*, se puede prevenir que el cambio de ruta se perciba de manera abrupta. Por ejemplo, si el personaje se encuentra girando hacia la derecha, en el siguiente *frame* del juego, puede suceder que se encuentre moviéndose hacía la diagonal superior derecha con un pequeño cambio de ángulo.

Este enfoque fue adoptado para la implementación en el juego, mediante el uso de un circulo de acción alrededor del personaje.

La fuerza de desplazamiento que se aplica, tiene como origen el circulo del centro y es restringida por un radio. Mientras mayor sea el radio del circulo (la distancia entre el circulo y el personaje), mayor será la fuerza del "empuje" que reciba el personaje en cada *frame*.

Ésta fuerza de desplazamiento sera utilizada para interferir en la ruta del personaje, y es utilizada para calcular la *wander force* y el ángulo resultante de su aplicación. Facilita la comprensión la visualización de la figura 3.

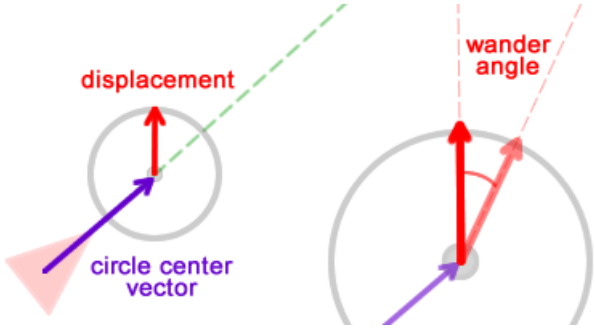

Figura 3: Representación de la aplicación del desplazamiento

```
function wander(personaje){
    //Calculo del circulo concentrico en el PJ
    var circleCenter = personaje.body.velocity.clone();
    circleCenter.normalize();
    circleCenter.scale(CIRCLE_DISTANCE)

    //Desplazamiento aleatorio
    var displacement = new Phaser.Math.Vector2(Math.random(),Math.random());
    displacement.scale(CIRCLE_RADIUS)


    // Cambia aleatoriamente la direccion del vector al cambiar su angulo
    setAngle(displacement, wanderAngle);

    // FIX: cambia muy lieramente el angulo para el sgte frame
    wanderAngle += Math.random() * ANGLE_CHANGE - ANGLE_CHANGE * .5;

    // Calcula la fuerza de wander
    var wanderForce;
    wanderForce = circleCenter.add(displacement);
    // Aplicamos la fuerza calculada
    personaje.body.velocity = wanderForce;
}
```

Listing 3: Implementación de *Wanderer*

Uso en el juego

En *Ainulindale* se implemento el movimiento *wander* (ver código 3) para las notas especiales, las melodías que no has de cantar. Éstas son notas diferentes a las encargadas de crear la canción y por ello se encuentran *'divagando'* alrededor del universo.

Se decidió que estos agentes (notas rojas) lleven el movimiento mencionado para que su paso por el mundo se vea fluido y armonioso. De esta manera, además, permite identificar fácilmente que estas notas son ajenas al *Ainur* que esta manejando el jugador.

FLOCKING

En un el mundo natural, los organismos exhiben ciertos comportamientos cuando viajan o se mueven en grupos. Este fenómeno que se denomina *flocking*, ocurre tanto a niveles microscópicos (en el caso de las bacterias) como en escalas macroscópicas (el comportamiento de los peces). Estos patrones pueden ser simulados creando reglas simples y combinandolas y permite simular movimientos naturales o caóticos reales de grandes grupos.

En particular, se implementaron tres movimientos que relacionados conforman el comportamiento *flocking*: *Alignment, Cohesion* y *Separation.*

Alignment

Alignment es el comportamiento que permite a un agente en particular mantenerse en linea con los agentes que se encuentran alrededor suyo. Es decir, evalúa el movimiento de los agentes que lo rodean e intenta rotar un *determinado ángulo* de forma que la dirección de su movimiento sea lo más perpendicular posible al vector de movimiento de su entorno cercano.

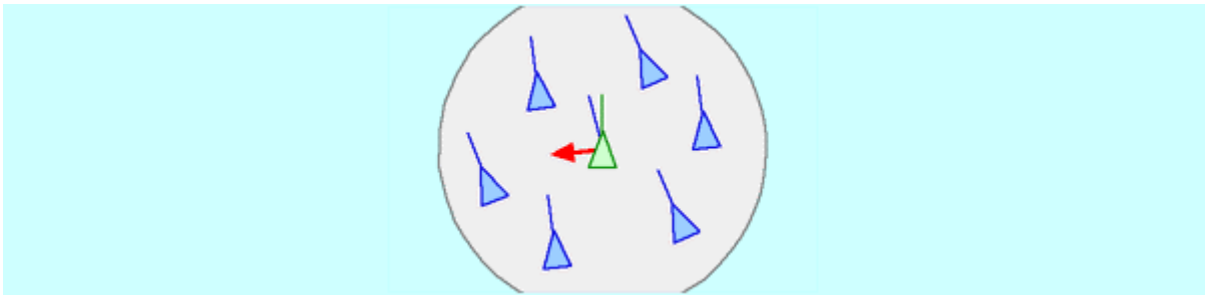

Figura 4: Movimiento de alineación

Para ello,se define un área o *radio de vecinos* que determine que agentes se consideran *cercanos*. Luego, se *itera* sobre cada uno de los agentes que se encuentren dentro del radio y

se suman los componentes del vector de velocidad de cada uno con la finalidad de sacar un promedio y normalizarlo. De esta forma se logra un movimiento homogéneo de alineación *"real"*.

```
for each (var agent:Agent in agentArray)
{
    if (agent != myAgent)
    {
        if (myAgent.distanceFrom(agent) < NEIGHBOR_RADIUS)
        {
            v.x += agent.velocity.x;
            v.y += agent.velocity.y;
            neighborCount++;
        }
    }
}
v.x /= neighborCount;
v.y /= neighborCount;
v.normalize(1);
return v;
```

Listing 4: Pseudocódigo del cálculo para *Alignment*

Cohesion

El comportamiento de cohesión causa que los agentes se muevan hacia el *"Centro de la masa"*, es decir, la posición promedio de la *nube* de agentes que se encuentran dentro de determinado radio.

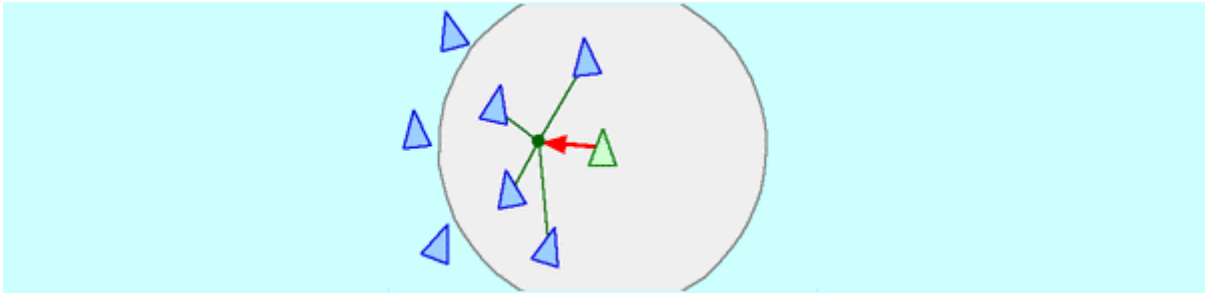

Figura 5: Movimiento de cohesión

La implementación es casi idéntica al movimiento *alignment*, pero con algunas diferencias. Primero, en vez de sumar las componentes del vector de velocidad, se busca la posición

de cada agente.

Luego, cómo se computa el vector promediando la suma de las posiciones sobre la cantidad de vecinos, teniendo como resultado, la posición que corresponde al centro de la masa. Una vez identificado el *centroide*, se calcula la dirección que toma cada agente para dirigirse hacia el centro de la masa, se normaliza el vector y se aplica iterativamente.

Phaser 3 cuenta con herramientas que permiten facilitar el cálculo del *centroide* al aplicar funciones matemáticas considerando el radio de acción como una figura geométrica y cada agente como una partícula de una nube de puntos.

Separation

El comportamiento *Separation* permite que los agentes mantengan una distancia natural con sus vecinos (entendiendo que la superposición de agentes es visualmente incómoda y poco natural).

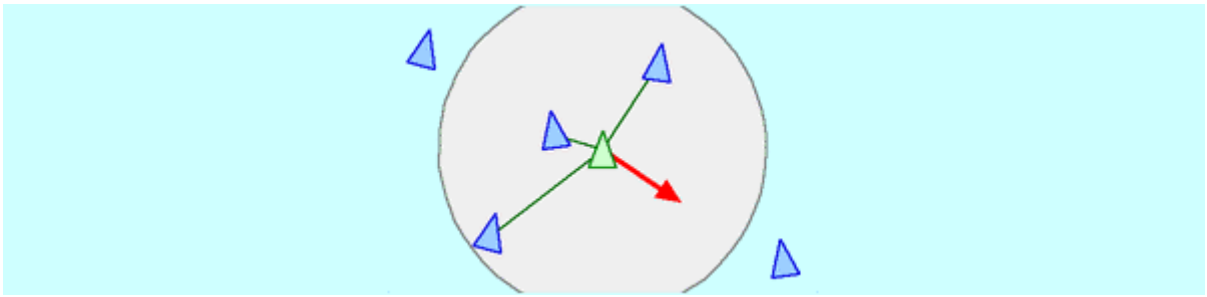

Figura 6: Movimiento de *separation*

La implementación de *separation* es muy similar a las explicadas anteriormente con la única diferencia que cuando un agente es encontrado dentro del radio de influencia, la distancia del mismo es añadida al cómputo del vector.

Para el cálculo final del vector, se promedian las sumas de las distancias con la cantidad de vecinos, se normaliza pero antes de ser aplicado, existe un único cuidado crucial: el vector debe ser negado para que los agentes mantengan una distancia con sus vecinos.

Flocking

Uniendo todo lo anterior y con las facilidades que provee Phaser 3, se logró el siguiente algoritmo que computa el movimiento *alignment* sobre la velocidad angular, *separation* gracias a la distancia entre puntos y el cálculo del *centroide* con las funciones *Geom* para el comportamiento de *cohesion*.

Uso en el juego

En *Ainulindale* se utilizo el movimiento *flocking* para la creación de las notas, esto es, distintos agentes se unen en un punto común para luego dar origen a la nota que llevara el fragmento de la canción correspondiente.

Se decidió utilizar esta estrategia de movimiento para lograr el efecto de unión. Dado que la melodía a cantar por cada *Ainur* esta compuesta de pensamientos, deseos y tiempo. De esta manera cada agente que compone la nube de *partículas* representa los componentes de cada melodía.

```
function flocking(boids,game){

      // canitdad de agentes (boids objects)
      var boidsAmount = boids.length;
      // velocidad de cada uno de los agentes/boids en px por segundo.
      var boidSpeed = 100;
      // radio de influencia o neighbor radio
      var boidRadius = 500;

      // array temporal para calcular el centroide
      var centroidArray = [];
      // iteramos sobre cada agente
      for(var i = 0; i < boidsAmount; i++){
           // para cada boid, controlamos sus vecinos
           for(var j = 0; j < boidsAmount; j++){
                // si es un agente diferente al personaje y se encuentra dentro del radio
                if(i != j && ((boids[i].x-boids[j].x)+(boids[i].y-boids[j].y)) < boidRadius){
                     // guaramos el agente como un punto de la nube
                     centroidArray.push(boids[j]);
                }
           }
           // Si el array auxiliar tiene algun agente, se computa el centroide
           if(centroidArray.length > 0){
                // calculo del centroide con herramientas geometricas
                var centroid = Phaser.Geom.Point.GetCentroid(centroidArray);
           }
           else{
                // si no existen agentes alrededor, se elige un punto aleatorio
                var randomPoint = new Phaser.Geom.Point(Math.abs(Math.random()*800),
                       Math.abs(Math.random()*600));
                var centroid = new Phaser.Geom.Point(randomPoint.x, randomPoint.y);
           }
           // calculamos la rotacion entre puntos hacia el centroide
           boids[i].body.angle = Phaser.Math.Angle.BetweenPoints(boids[i],centroid);
           // movemos los agentes hacia el centroide aprovechando los movimientos ya calculados
           seek(boids[i],centroid);
       }
    }
```

Listing 5: Implementación en Phaser de *Flocking*

POSIBLES MEJORAS

En el diseño de *Ainulindale* se especificaron múltiples características que por cuestiones de tiempo y propias del transcurso de la materia en que se encuadra este trabajo no pudieron llegar a la implementación y quedan como trabajos a futuro y *posibles mejoras*.

Entre ellas se pueden encontrar (sin un orden específico):

- Opciones *multijugador* con competencia de *ainures* tanto contra jugadores físicos como contra agentes inteligentes.
- Múltiples niveles con diferentes obstáculos y la posibilidad de controlar más de un *Ainur* de acuerdo al progreso del juego.
- Sistema de *Ranking*.
- Menú con las múltiples opciones de jugabilidad, opciones gráficas (nivel de gráficos alto o bajo) y sonido.
- Mejoras en el sonido: actualmente *Phaser 3* se encuentra en versión beta y las funcionalidades de sonidos son inestables y no funcionan adecuadamente, lo que complejiza la creación de la melodía por partes y causa desperfectos de funcionamiento.
- Una pequeña guía *demo* para aprender a jugar que explique la historia del juego.
- Completar la implementación restante para que el juego sea completamente *responsive*.

Bibliografía

[1] MILLINGTON, I., AND FUNGE, J. *Artificial Intelligence for Games, Second Edition*, 2nd ed. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2009.

[2] PHASER. Tutorial, https://phaser.io/learn.

[3] REYNOLDS, C. Steering behaviors for autonomous characters, 1999.

[4] TUTSPLUS. Game developments tutorial, https://gamedevelopment.tutsplus.com/series/understanding-steering-behaviors–gamedev-12732.