ESCUELA TÉCNICA SUPERIOR DE INGENIERÍA INFORMÁTICA

GRADO EN INGENIERÍA DEL SOFTWARE

SIMULADOR DE ECOSISTEMAS PARA EL ESTUDIO Y ANÁLISIS DE POBLACIONES ARTIFICIALES

ECOSYSTEM SIMULATOR FOR THE ANALYSIS AND STUDY OF ARTIFICIAL POPULATIONS

Realizado por

Antonio Vidal Carrasco

Tutorizado por

Pablo López Olivas

Departamento

Lenguajes y Ciencias de la Computación

UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO 2015

Fecha de Lectura:

El secretario del Tribunal

Resumen:

Cada vez se simulan más situaciones de cualquier índole y se hace de varias maneras. Desde simular cambios cuánticos de las partículas, pasando por programas de entrenamiento para pilotos hasta llegar a las casas en forma de videojuego, la simulación está presente en nuestra vida.

El objetivo de este trabajo es realizar un pequeño simulador del juego de la vida pero no en forma de autómata celular, sino más bien representando especies mayores de animales, que pueden decidir por sí mismas y moverse por el mundo.

Para realizar esta tarea se necesitan un amplio conocimiento de técnicas informáticas y matemáticas, haciendo especial hincapié en la inteligencia artificial.

Palabras claves:

F#, Programación funcional, Simulación, Inteligencia Artificial, Quadtree, Ray casting, Comportamientos, Toma de decisiones, Movimiento cinemático, Multiproceso, .NET, Bases de datos, MySQL.

Abstract:

Simulations of various kinds are getting more and more common.

Simulation is present in our lives in a large number of ways, ranging from simulating quanta change on particles to flight conditions for training pilots, all the way to sophisticated video games for home entertainment.

The goal of this work is to develop a simulator for the game of life. Our simulator is not based on the usual cellular automaton approach; we shall instead represent individual specimens of animal species who can decide and move around the world by themselves.

To accomplish this task, extensive knowledge of Computer Science and mathematical
techniques are needed, with particular emphasis on Artificial Intelligence.

Keywords:

F#, Functional programming, Simulation, Artificial Intelligence, Quadtree, Ray casting, Behaviors, Decision making, Steering Behaviors, Multithread, .NET, Data bases, MySQL.

Índice

Introducción y objetivos

El mundo real se ha sumergido en la computación y cada vez se usan a los ordenadores para tareas más complicadas y exigentes, donde la simulación del propio mundo real está tomando un papel muy importante. Las simulaciones por ordenador se han vuelto muy útiles en campos como la física, química, biología, economía, ingeniería, entrenamiento de personal y un largo etcétera que acaba con los videojuegos. Si buscamos en la mayor plataforma de videojuegos actualmente, Steam[1], juegos de simulación, encontraremos la más variopinta variedad de ellos.

El objetivo de este proyecto es hacer un simulador de ecosistemas donde convivan distintas especies que interactúan entre ellas y su entorno. Se podría simplemente definir especies ligadas a máquinas de estado que cambiarían con las distintas entradas pero que siempre se comportarían igual independientemente del individuo. Sin embargo, en este proyecto se va a dar un paso más y se va a intentar que cada criatura sea única, con sus propias características y objetivos, a pesar de pertenecer a una misma especie. Para lograr este objetivo habrá que estudiar distintas técnicas de inteligencia artificial, como las técnicas de movimiento, evasión de colisiones y decisión de objetivos.

El proyecto se ha realizado en F#, lenguaje de código abierto para la plataforma .NET de Microsoft. Se ha elegido este lenguaje porque es un lenguaje que aúna la programación funcional con la programación imperativa, además de tener orientación a objetos, permitiendo usar la librería de .NET y ser fuertemente tipado ya que utiliza inferencia de tipos.

La memoria está dividida en capítulos, donde se intentará explicar el desarrollo del proyecto desde las capas más bajas del programa a las más altas.

[1] http://store.steampowered.com/search/?snr=1_4_4__12&term=simulator

Capítulo 1: Introducción a F#

En este capítulo introduciremos brevemente el lenguaje F#. Existen guías online y libros muy completos que pueden servir para el aprendizaje del lenguaje. Se encuentran en la bibliografía de este proyecto y han sido utilizados como referencia asiduamente.

Puesta en marcha

F# se puede compilar desde varios entornos de desarrollo, como Visual Studio y Xamarin Studio, pero también se puede compilar directamente desde un terminal de órdenes. Además, incluye un modo interactivo realmente útil con el que probar el código que vamos escribiendo, lo cual acelera el desarrollo y siempre tenemos un entorno para realizar pruebas. Para realizar este proyecto se comenzó utilizando Visual Studio 2012, pero cuando salió Visual Studio 2013 se hizo una importante contribución al editor de texto del lenguaje, detectando mejor los errores de compilación y sugiriendo o incluso auto escribiendo las funciones y métodos necesarios, incluyendo las librerías de .NET.

Así que simplemente con la instalación de Visual Studio 2013 ya se puede empezar a programar en F#. Otros métodos de instalación se pueden encontrar en su página oficial[2] .

F# en 20 minutos

Vamos a introducir F# una manera muy rápida para aquellos que no estén familiarizados con la sintaxis. No va a ser una introducción muy detallada por lo que se aconseja utilizar material de apoyo para aprender este lenguaje.

Una diferencia de F# y lenguajes con estilo C son la ausencia de llaves para delimitar bloques de código. Como en Python, se utilizar la indentación para ello. Otra diferencia principal es el poder usar espacios en vez de comas a la hora de pasar parámetros. Sin embargo esto sólo funciona si no estamos usando clases, sobre todo las provenientes de .NET.

Comentarios

Se pueden escribir comentarios usando // para una línea o (* texto *) para bloques de texto. También se puede usar /// para usar la auto documentación que .NET y Visual Studio interpreta como oficial. Los comentarios pueden ser agrupados y son interpretados como espacio en blanco, por ejemplo “a (*…*) b” es interpretado como “a b” y no como “ab”.

[2] http://fsharp.org/

Declaración de variables

Como lenguaje funcional que es F#, por defecto las variables no se pueden cambiar. Es decir, si declaramos una variable como "1", luego no se le puede asignar otro valor. Esto es así para garantizar la completitud del programa por parte del compilador de F#. Sin embargo, F# es un lenguaje moldeable y nos permite tratar con tipos mutables, sobre todo por su interoperabilidad con otros lenguajes y bibliotecas de .NET. Para declarar una variable como mutable sólo tenemos que usar la palabra reservada "mutable". Además también existe la posibilidad de crear punteros como en otros lenguajes de programación, lo cual nos vuelve a ayudar la hora de utilizar los lenguajes de .NET o para optimizar en memoria.

```
let a = 1 // Variable no mutable

a <- 2 // Error

let mutable b = 1 // Variable mutable

b <- 2 // Bien

let cell1 = ref 1

cell1 <- 2 // Error

cell1.Contents <- 2 // Bien

b <- b + cell1 // Error

b <- b + !cell1 // Bien
```

Tipos básicos

F# proporciona un gran número de tipos, como los tipos unit, int, float, char, string y bool. También incluye otros tipos como las tuplas y variantes, que veremos más adelante. A continuación se puede ver una tabla con distintos tipos.

Tipo	Ejemplo
unit	()
int	2
float	2.0
float32	2.0f
char	'h'
string	"hola mundo"
type → type (funciones)	funName param = param * 2

Como podemos ver en esta breve tabla, las funciones también son tipos. Es decir, si usamos como parámetro funName, veremos que tiene el tipo (int -> int)= <fun:it>, es decir, es una función que recibe un entero y devuelve un número entero. Más adelante veremos ejemplos de composición de funciones.

Arrays, listas, tuplas y estructuras de datos

F# nos aporta una rica variedad de estructuras integradas con la programación funcional. Entre ellas encontramos algunas mutables y otras que no lo son.

Tuplas

Las tuplas son estructuras inmutables que se declaran de la siguiente manera: (expr,…, expr). Si la tupla es de 2 elementos, por ejemplo (a, 1), podemos usar los operadores "fst" y "snd" para obtener el primer y segundo valor respectivamente.

Listas

Al igual que las tuplas, las listas son inmutables. Se pueden declarar de varias maneras:

Declaración	Ejemplo
[expr; .. ; expr]	[1;2;3;4]
[expr..expr]	[1..4]
[comp-exp]	[for x in 1..99 → x * x]
expr :: expr	[1..4] :: [5…9]
expr @ expr	[1;2] @ [3]

El operador :: añade una lista o elemento en la primera posición de la lista. En realidad no lo añade, sino que crea otra lista nueva con este elemento.

El operador @ es análogo al operador :: pero realiza la operación al final de la lista.

Existen otros operadores a los que se pueden acceder usando la palabra clave List. Ejemplos comunes y útiles de estos operadores son List.map que recibe una función y se la aplica a la lista. List.fold, al igual que List.map, aplica una función sobre la lista mientras acumula el resultado en una variable que devuelve al final. Otro función importante es List.filter, que dada una función nos devuelve una lista con todos los elementos que cumplen la función.

```
let list = [1..10] // [1; 2; 3; 4; 5; 6; 7; 8; 9; 10]

list |> List.map (fun n -> n*2) // [2; 4; 6; 8; 10; 12; 14; 16; 18; 20]

list |> List.fold (fun acc elem -> acc + elem) 0 // 55

list |> List.filter (fun elem -> elem % 2 = 0) // [2; 4; 6; 8; 10]
```

El operador |>, llamado “forward pipe”, es quizás uno de los operadores más importantes en F#. Su función es realmente simple:

```
let (|>) x f = f x
```

Como se puede ver, lo único que hace es aplicar la función en orden contrario. Sin embargo tiene claras ventajas:

Claridad: Cuando se usa en conjunción con operadores como List.map, este operador permite transformar e iterar sobre datos en cadena, como si fuese una tubería.

Inferencia de tipos: Permite que la información fluya desde los objetos de entrada hasta las funciones que los manipulan. F# usa la información recogida por la inferencia de tipos para resolver algunas construcciones del lenguaje como las propiedades accedidas y la sobrecarga de métodos. Para hacer esto la información se tiene que propagar de izquierda a derecha en el código.

Arrays

Los arrays son hermanos de las listas, sin embargo sus valores pueden ser cambiados. En la siguiente tabla se recogen como se pueden crear arrays y algunas funciones elementales:

Declaración	Ejemplo
[| expr; ...; expr |]	[|1;2;3;4|]
[| expr..expr |]	[|1..4|]
[| comp – expr |]	[|for x in 1..99 -> x * x|]
Array.create size expr	Array.create 10 “”
Array.init size expr	Array.init 10 (fun index → index * index)
arr.[expr]	arr.[0] // Accede al primer elemento
arr.[expr] ← expr	arr.[1] ← “nueva variable”
arr.[expr..expr]	arr[0..2] // Array con los 3 primeros elementos
arr.[expr..]	arr[2..] // Desde el 3 elemento hasta el final
arr.[..expr]	arr[..2] // Desde el primer hasta el tercero

Ejemplos de uso:

```
[| 1 .. 10 |]
|> Array.filter (fun elem -> elem % 2 = 0)
|> Array.choose (fun elem -> if (elem <> 8) then Some(elem*elem) else None)
|> Array.rev
|> printfn "%A"


printfn "%A" (Array.collect (fun elem -> [| 0 .. elem |]) [| 1; 5; 10|])
```

Estructuras de datos

F# incluye una colección completa de estructuras de datos que ya estamos acostumbrados a usar en otros lenguajes. Son los siguientes:

Estructuras de datos incluidos en .NET
System.Collections.Generic.Dictionary
System.Collections.Generic.List
System.Collections.Generic.SortedList
System.Collections.Generic.SortedDictionary
System.Collections.Generic.Stack
System.Collections.Generic.Queue
Microsoft.FSharp.Collections.Set
Microsoft.FSharp.Collections.Map

Correspondencia de patrones

Unas de las herramientas más potentes que nos ofrece F# es la búsqueda de patrones. Es un método mejorado del switch/case que se encuentra en otros lenguajes. Vamos a ver un sencillo ejemplo:

```
let simplePatternMatch =
      let x = "a"
      match x with
       | "a" -> printfn "Es una a"
       | "b" -> printfn "Es una b"
       | _   -> printfn "No es ni una a ni una b"
```

El patrón "_" se elige cuando no se puede elegir ninguna otra opción.

Antes han salido los valores “None” y “Some”, sin embargo no han sido explicados. Ahora que se ha introducido la correspondencia de patrones podemos hacerlo, aunque quizás más adelante cuando se expliquen los valores de unión se entiendan completamente. En realidad “None” y “Some”, llamados tipos opción, simplemente representan a un envoltorio para valores nulos o respuestas de funciones con valor nulo. El valor nulo existe en F# para mantener su interoperabilidad con .NET, pero en las aplicaciones propias de F# no se permite normalmente su uso. Para estos casos se usa el tipo opción. A continuación un ejemplo usando correspondencia de patrones y estos elementos:

```
let validValue = Some 99

let invalidValue = None

let optionPatternMatch input =

   match input with

    | Some i -> printfn "input is an int=%d" i

    | None -> printfn "input is missing"


optionPatternMatch validValue

optionPatternMatch invalidValue
```

Definiciones de tipos y objetos

En este apartado se introduce la definición de nuevos tipos y la creación de objetos a través de las clases.

Tipos registro

Estos tipos son los más básicos que podemos crear, están camino de ser objetos pero permanecen inmutables a lo largo del tiempo.

```
type person = {Name: string; DateOfBirth: System.DateTime; }
```

Para crear un objeto de este tipo simplemente usamos las etiquetas:

```
{Name = "Antonio"; DateOfBirth =
new System.DateTime(1962,09,02) }
```

Tipos Unión

Ya se ha visto un ejemplo de “None” y “Some”. Este tipo está declarado de la siguiente manera:

```
type Option<'T> =
| None
| Some of 'T
```

Los tipos unión nos permiten declarar tipos de datos que son versátiles y aceptan varias interpretaciones, lo cual es realmente útil. En el siguiente ejemplo vemos cómo se puede declarar la temperatura en grados Celsius o Fahrenheit manteniendo una semántica:

```
type Temp =
    | DegreesC of float
    | DegreesF of float
let temp = DegreesF 98.6
```

Objetos y clases

La plataforma .NET es fundamentalmente un entorno de programación orientado a objetos. El compilador de F# en realidad compila todos los constructores que hemos visto hasta ahora en objetos de .NET, por lo que a estos tipos ya introducidos se les puede añadir métodos como si de clases se tratasen. Veamos algunos ejemplos:

```
type vec2 = { mutable x : float; mutable y: float } with
      static member ( + ) (a,b) =
            {x = a.x + b.x ; y = a.y + b.y}
      member r.Length
            with get () =
            sqrt(r.x * r.x + r.y * r.y)
        and set len =
            let s = len / r.Length
            r.x <- s * r.x
            r.y <- s * r.y
```

Por otra parte también podemos definir clases como haríamos en otros lenguajes. Esta técnica es necesaria cuando queremos interoperar con clases de .NET. Tenemos 2 maneras de definir las clases en F#, que vamos a ver a través de ejemplos análogos al anterior.

Constructores Explícitos

```
type vec2 =
      val private x_ : float
      val private y_ : float
      new (x) =
          {x_ = x; y_ = 0.0}
      new(x,y) =
          {x_ = x; y_ = y}
      member r.x = r.x_
      member r.y = r.y_
      member r.Length =
         sqrt(r.x * r.x + r.y * r.y)
```

En este ejemplo hemos definido 2 maneras de crear el objeto vector, en uno sólo definimos la x, mientras que la y será igual a 0. También se puede observar la palabra clave “new” para crear un objeto es opcional.

Constructores implícitos

En muchos casos, las clases sólo presentan un constructor para crear los objetos. En F# existe un atajo para crear este tipo de constructores que hace la definiciones de clases más sencillas. Además, este tipo de constructores admite parámetros opcionales que como su nombre indica, pueden ser asignados al crear el objeto o automáticamente.

```
type vec2 (x : float, y : float, ?paramOpcional : int) =
      let ParametroOpcional : int = defaultArg paramOpcional 0
      member r.x = x
      member r.y = y
      member r.Length =
            sqrt(r.x * r.x + r.y * r.y)
let vector = vec2(3.0, 4.0).Length // 5.0
let vector2 = new vec2(3.0, 4.0, 2)
```

La palabra clave “defaultArg” indica que si el parámetro opcional no está definido se use el valor indicado la asignación.

Excepciones

Como cualquier otro lenguaje de .NET, F# soporta el manejo de excepciones. Es importante conocer el manejo de las excepciones debido a que si usamos elementos de .NET tendremos que usarlas.

Definición

Cuando lanzamos excepciones podemos usar las estándar que nos ofrece .NET, como InvalidOperationException, pero también podemos definir nuestras propias excepciones de la siguiente manera:

```
exception MiExcepcion1 of string

exception MiExcepcion2 of string * int
```

Lanzar excepciones

Tenemos 3 maneras de lanzar excepciones en F#. Podemos usar las funciones incorporadas de F#, usando las excepciones estándares de .NET o usando nuestras propias excepciones.

Usando el primer método nos encontramos con 4 útiles funciones que nos ofrece F#:

- failwith, que lanza una System.Exception, tiene una variante que es failwithf
- invalidArg, que lanza una ArgumentException
- nullArg, que lanza una NullArgumentException
- invalidOp, que lanza InvalidOperationException

Veamos un ejemplo completo con las 3 maneras:

```
if x = “bad” then
      failwithf "Fallo en %A a la fecha %O" x DateTime.Now
else printfn "Todo correcto”
if y then "ok"
else raise (new InvalidOperationException("Descripción"))
if z then "ok"
else raise (MiExcepcion1 "Mensaje")
```

Capturar Excepciones

Para capturar excepciones usamos el bloque try-with en vez de try-catch como en otros lenguajes. Es mucho más cómodo y nos permite usar la correspondencia de patrones para tratar los errores. A veces queremos obligar al programa, tengamos una excepción o no, que cierto bloque de código se ejecute, para ello en otros lenguajes usamos la palabra clave finally.

Sin embargo en F# tenemos que usar el bloque try-finally para obtener esta funcionalidad. Esto hace que no podamos obtener ninguna excepción dentro de ese bloque try y F# nos obliga a encapsular el bloque try-finally en otro try-with para tener las funcionalidades completas de la captura de excepciones. También disponemos de la palabra clave reraise para cuando queramos relanzar la excepción, de manera que siga subiendo niveles en la pila de llamadas.

En el siguiente ejemplo se da una muestra de este comportamiento:

```
try
    try
       if x then "ok" else failwith "fail"
  finally
       printfn "Esto siempre se mostrará"
with
| :? System.Exception as ex -> printfn "%s" ex.Message;
| _ -> printfn "Excepción desconocida"; reraise();
```

Una vez vista esta pequeña introducción a F#, vamos a exponer las debilidades y fortalezas del lenguaje.

Beneficios

Seguridad: El compilador comprueba antes de la ejecución del código que este es seguro, como por ejemplo comprobando que no realiza ninguna violación de acceso.

Funcional: Las funciones están completamente integradas, se pueden anidar, pasar como argumentos e incluso guardar en estructuras de datos.

Fuerte tipado: Los tipos de todos los valores son comprobados durante la compilación para asegurar que están bien definidos y su uso es válido.

Tipado estáticamente: Cualquier error de escritura en el programa se recoge durante la compilación, no durante la ejecución como en otros lenguajes.

Inferencias de tipo: Los tipos de los valores son automáticamente inferidos durante la compilación del contexto donde se encuentran. Esto hace que raramente sea necesario declarar el tipo de las variables, que incluso se muestran en el entorno de desarrollo mientras se programa.

Correspondencia de patrones: Los valores, particularmente el contenido de estructuras de datos, pueden ser comparados con complicados patrones que determinan la solución a seguir.

Interoperabilidad: Los programas escritos en F# pueden llamar y ser llamados desde cualquier otro lenguaje de .NET, desde librerías nativas e incluso desde internet.

Desventajas

Extremadamente lineal: Si algo complica al programador a la hora de trabajar con F# es su extremada linealidad. Al ejecutarse de manera imperativa, es decir de arriba abajo en el orden del código, F# no es capaz de deducir lo que viene después. El ejemplo más claro de esto son los tipos. Si declaramos la clase A que usa a la clase B, pero la clase B todavía no está declarada, nos dará error de compilación. Esto puede ser muy frustrante pero tiene una razón de ser ya que evita las dependencias cíclicas, lo cual es beneficioso para el lenguaje.

Se puede evitar en cierta medida con la palabra clave "and" para vincular tipos, pero esto la mayoría de las veces no es posible, ya que normalmente hay que separar el programa en módulos, espacios de nombres, ficheros, etc…

Organización: Si a las desventajas de arriba le sumamos el no poder crear paquetes o carpetas donde organizar nuestro código, esto puede dar muchos quebraderos de cabeza a un equipo de trabajo, ya que trabajar varias personas sobre el mismo fichero con miles de líneas no es bueno. Sin embargo esto es más culpa del editor que del propio F# y se pueden separar en carpetas usando el precompilador[3], aunque esto no evita la linealidad del lenguaje.

[3] https://web.archive.org/web/20120116085906/http://cultivatingcode.com/2010/02/12/folders-in-f-projects/

Ejemplo completo

He desarrollado este pequeño ejemplo para ver la facilidad y potencia de F#. En este ejemplo se va a leer la página web de la Universidad de Málaga y se va a contar el tiempo que se tarda, la longitud de la página, el número de palabras y los enlaces que contiene.

```
open System
open System.IO
open System.Net

type PageStats =
    { Site : string;
    Time : System.TimeSpan;
    Length : int;
    NumWords : int;
    NumHRefs : int}

let http(url:string) =
       let req = System.Net.WebRequest.Create(url)
       let resp = req.GetResponse()
       let stream = resp.GetResponseStream()
       let reader = new StreamReader(stream)
       let html = reader.ReadToEnd()
       resp.Close()
       html

let time f =
       let start = DateTime.Now
       let res = f()
       let finish = DateTime.Now
       (res, finish - start)

let delimiters = [| ' '; '\n'; '\t'; '<'; '>'; '='|]
let getWords (s:string) = s.Split(delimiters)

let stats site =
       let url = "http://" + site
       let html, t = time (fun () -> http url)
       let hwords = html |> getWords
       let hrefs = hwords |> Array.filter (fun s -> s = "href")
       { Site=site; Time=t; Length=html.Length;
NumWords=hwords.Length; NumHRefs=hrefs.Length}

printfn "%A" (stats "www.uma.es")

(* Devuelve

       {Site = "www.uma.es";

        Time = 00:00:03.8115148;

        Length = 52083;

        NumWords = 17430;

        NumHRefs = 127;}

*)
```

Capítulo 2: Motor gráfico

En primer lugar, con objeto de obtener datos significativos sobre la aplicación, se ha desarrollado un motor gráfico que permita la representación de dichos datos para el usuario. Estos datos no son solamente numéricos, sino también visuales, como son las entidades que componen el sistema.

Para realizarlo, se han usado las librerías disponibles en .NET, así como métodos de dibujo propios.

Dibujo de las entidades

En este apartado se describen cómo se dibujan por pantalla y qué información contienen las figuras que representan a las entidades, imágenes y texto mostrados por pantalla. Para ello, se usa la librería System.Drawing[4] de .NET. A continuación, se describen una a una las clases implementadas:

Line

En .NET disponemos de métodos de dibujado entre dos puntos para representar líneas, sin embargo no hay ninguna estructura que defina formalmente lo que es una línea dentro de estas bibliotecas. Por ello se ha definido un nuevo tipo línea que simplemente recibe 2 puntos como parámetros. Además implemente los métodos de la distancia entre dos puntos, que sería la longitud de la línea, y el punto correspondiente a la mitad de la línea.

Shape

Contiene y maneja la estructura básica de cualquier polígono. Todos los parámetros que pueden constituir esta clase son opcionales, sin embargo, ofrecen el procedimiento básico para manejar un polígono. Además, las demás clases que vamos a ver en este apartado heredan sus métodos y parámetros y los sobrescriben en algunos casos. A continuación, se exponen dichos parámetros, agrupados según/en función de su uso y entre paréntesis se encuentra su nombre real en el código del proyecto:

- **Anchura** (*width*): Establece la anchura del polígono.
- **Altura** (*height*): Establece la altura del polígono.
- **Centro** (c): Señala centro del polígono.
- **Dibujar rectángulo** (*drawRectangle*): Indica si se dibuja el rectángulo asociado a este polígono.

[4] http://msdn.microsoft.com/en-us/library/system.drawing.graphics(v=vs.110).aspx?cs-save-lang=1&cs-lang=fsharp#code-snippet-1

Estos parámetros posibilitarán posteriormente la creación de rectángulos que se usarán para la detección de colisiones y eventos de control por parte del usuario. Esto permitirá detectar, por ejemplo, si el usuario ha hecho clic sobre el polígono.

Los siguientes parámetros constituyen el propio polígono y el cómo se pinta este polígono. Esto lo conseguimos gracias al pincel de dibujado y al establecer si dibujamos una sombra o no bajo el polígono. El dibujado de las sombras es tan sencillo como dibujar de un color gris el mismo polígono con su centro desplazado un pixel en dirección contraria a la fuente de luz.

- **Pen** (pencil): Pincel con su color y modo para dibujar el polígono
- **Sombras** (Shadows): Establece si se dibuja una sombra de este polígono.
- **Points** (points): El polígono se dibuja uniendo estos puntos.

Asimismo, se definen métodos públicos para establecer u obtener dichos métodos desde el objeto de la clase.

Por último, esta clase comprende una serie de métodos abstractos que pueden ser redefinidos por las clases que lo hereden. Estos métodos son:

Draw (dibujar)

Dado el controlador gráfico [5] donde dibujar, este método se encarga del dibujo de los puntos del polígono y del rectángulo de colisión de la entidad.

Move y Spin (Mover y Girar)

Estos dos métodos abstractos están definidos para que sean implementados por las clases que los hereden. Se encargarán de girar y mover la figura dependiendo del ángulo asignado.

Circle

Esta clase dibuja un círculo en pantalla. Recibe como parámetros el radio del círculo, el punto donde se va a pintar, y el pincel de dibujo. Además como parámetro opcional podemos indicar si se dibuja su sombra. El radio mínimo del círculo es de 1, ya que más pequeño que 1 pixel no puede dibujarse por pantalla.

Ya que para dibujar un círculo podemos usar los métodos DrawEllipse[6] y FillEllipse[7] que nos proporciona la interfaz de dibujo de .NET, sólo tenemos que calcular cómo se mueve el centro del círculo para dibujarlo

[5] Un objeto Graphics que nos ofrece System.Forms de .NET
[6] http://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawellipse(v=vs.110).aspx
[7] http://msdn.microsoft.com/en-us/library/system.drawing.graphics.fillellipse(v=vs.110).aspx

ClosedCurve

Esta figura es especial porque representa una curva cerrada entre los puntos proporcionados. Es decir, une cada punto de tal manera que no tiene puntos muertos y encierra un área por completo.

Figura 1. Curvas abiertas y cerradas.

En el programa sólo se usa para dibujar, como se verá más adelante, la cabeza de los individuos, así que esta clase recibe como parámetros el centro de la figura y la distancia a mantener entre los puntos.

Para dibujarlas sólo tenemos que llamar a los métodos DrawClosedCurve [8] y FillClosedCurve[9] que proporciona .NET, pasando como argumento los puntos en los que queremos encerrar la curva. Es importante notar que el dibujo dependerá del orden en el que se pasen los puntos.

CreatureShape

En esta clase se implementa cómo se representan las criaturas en la aplicación, además de mantener algunas estructuras para la detección de colisiones. Para ello se combinan un Circle, una ClosedCurve y 3 líneas. A partir de ahora nos referiremos a estas figuras como el cuerpo, cabeza y bigotes de la criatura, respectivamente.

Recibe como parámetros el centro de la criatura, los colores del cuerpo y la cabeza y además una textura y un color para esta textura. Las texturas simplemente son un bitmap que usará el pincel a la hora de dibujar el cuerpo.

Por último, se debe tener en mente que el centro de esta figura es el centro de su cuerpo, es decir, el centro del círculo.

[8] http://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawclosedcurve(v=vs.110).aspx
[9] http://msdn.microsoft.com/en-us/library/system.drawing.graphics.fillclosedcurve(v=vs.110).aspx

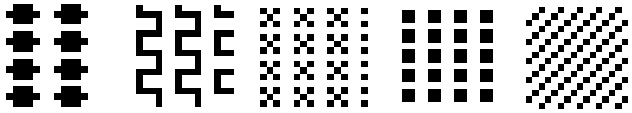


Figura 2. Texturas de las criaturas.

Se aplican texturas a las criaturas para aumentar la variabilidad y vistosidad a la hora de representarlas. Además, usando el modo de dibujado[10] podemos cambiar aún más esta variabilidad, hasta en 4 por cada textura y color. Usando el formato RGB se pueden obtener hasta 2^24 colores que multiplicados por las 5 texturas y sus 4 modos proporcionan una cantidad más que suficiente de variabilidad al sistema, concretamente 335544320 posibles criaturas.

```
let initBitmap =
   for i = 0 to bitmap.Height - 1 do
        for j = 0 to bitmap.Width - 1 do
             if bitmap.GetPixel(i,j) <> Color.FromArgb(255,0,0,0) then
                bitmap.SetPixel(i, j, bodyColor)
             else
                bitmap.SetPixel(i, j, textureColor)
```

Figura 3. Inicialización de textura.

El método más importante de esta clase es sin duda el de girar a la criatura. Cuando una criatura se orienta hacia otra posición, realmente lo único que gira es su cabeza, ya que girar un círculo carece de sentido en el sistema (al pintarlo se va a ver igual se gire o no). Sin embargo girar la cabeza no es trivial, ya que se compone de 4 puntos proporcionados. En la siguiente figura se muestran estos puntos.

Figura 4. Visualización de los puntos y ángulos que componen la cabeza de una criatura.

Así que dado el ángulo total sobre el que queremos girar se hallan los ángulos que distan 30 grados respecto a él y se calcula el seno y el coseno de todos estos ángulos, ya que ahora se usan para calcular las 8 coordenadas necesarias para pintar la cabeza. Para hallar una de las coordenadas sólo hay que multiplicar la proporción que se indicó al crear la criatura por su ángulo y sumárselo al centro de la criatura. En el caso de los dos puntos medios se multiplica por una proporción ya establecida de 2 (el doble de la proporción dada) y 1.3 (30% más largo que la proporción dada).

[10] http://msdn.microsoft.com/en-us/library/system.drawing.drawing2d.wrapmode(v=vs.110).aspx

Para la manutención de los bigotes tratamos exactamente el mismo modelo, sin embargo la proporción (o longitud) de los bigotes será de 12 para los que están a los lados y 17 para el central. Si vemos en la figura 4 las líneas rectas, por allí pasarían las rectas que conforman los bigotes de la criatura. Cabe destacar que los bigotes de la criatura sólo nos van a servir para las colisiones, no son elementos visuales aunque podremos visualizarlos eligiéndolo en las opciones, tal y como se verá más adelante.

Gracias a los métodos que se han implementado en las demás clases, el movimiento de esta figura es tan sencillo como mover cada figura que lo compone. Así que simplemente se asigna el nuevo centro al cuerpo y la cabeza de la criatura y una vez actualizada se gira como se ha indicado en el párrafo anterior.

Image

Por último, esta clase nos permite cargar una imagen y pintarla en pantalla usando el método DrawImage[11]. Es importante saber que siempre se dibuja desde la esquina superior izquierda hacia la esquina inferior derecha, así que el punto desde el que se dibuja es (centro.X - mitad del ancho de la imagen, centro.Y - mitad del alto de la foto).

ResourceManager

Esta clase ofrece el gestor de recursos del sistema, es decir, las texturas e imágenes usadas para representar las entidades. Estos recursos se encuentran embebidos en el programa, en un archivo .dll y en la carpeta resources si vemos el código del proyecto. Son imágenes en formato PNG[12] debido a que usamos transparencias para pintar los objetos y el mundo. Se archivan todas las imágenes en una DLL[13] para usar el método GetManifestResourceStream[14] que ofrece .NET de carga de recursos externos para el programa, de tal manera que el código para obtener las texturas quedaría así:

```
ref (new
Bitmap(Assembly.GetExecutingAssembly().GetManifestResourceStream(imagen.ext)))
```

Las texturas usadas pertenecen la mayoría al juego Age of Empires[15], aunque otras han sido creadas manualmente pero con la misma idea y paleta de colores que en ese juego.

[11] http://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawimage(v=vs.110).aspx
[12] https://en.wikipedia.org/wiki/Portable_Network_Graphics
[13] https://en.wikipedia.org/wiki/Dynamic-link_library
[14] http://msdn.microsoft.com/en-us/library/vstudio/xc4235zt(v=vs.100).aspx
[15] https://en.wikipedia.org/wiki/Age_of_Empires

Césped

Sirve para pintar el bioma continental.

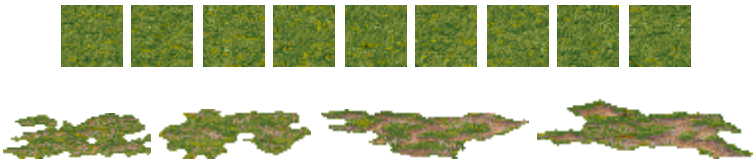


Figura 5. Texturas para el bioma continental.

Desierto

Sirve para pintar el bioma de desierto.

Figura 6. Texturas para el bioma desértico.

Agua

Se usan para pintar el bioma acuático.

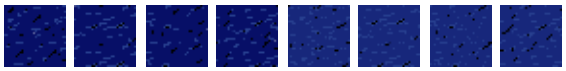


Figura 7. Texturas para el bioma acuático.

Comida

Representa la comida en el sistema que no proviene de una criatura

Figura 8. Textura para la comida vegetal.

Capítulo 3: Gestor de colisiones

En este capítulo se muestra como se ha desarrollado el gestor de colisiones de la aplicación. Las colisiones siempre son un ámbito en desarrollo debido al potencial uso de recursos que hace del ordenador. Esto es debido a que por cada entidad que haya en el sistema, se tiene que comprobar con cada uno de las otras entidades si se está chocando o no, es decir, hecho de una manera inmediata y sin aplicar ninguna técnica estaríamos hablando de algoritmos de complejidad exponencial O(n^n). Para evitar esto se ha implementado la estructura de datos en forma de árbol llamada Quadtree[16]. Estos son usados para dividir recursivamente en cuatro regiones iguales representaciones de 2 dimensiones. Gracias a ello, se elimina gran parte de las comprobaciones que se realizan, reduciendo la complejidad de búsqueda de colisiones a O(d*n) donde d es la profundidad a la que se encuentra el objeto.

A continuación se detallan las clases e implementaciones llevadas a cabo para realizar la estructura.

Boundary

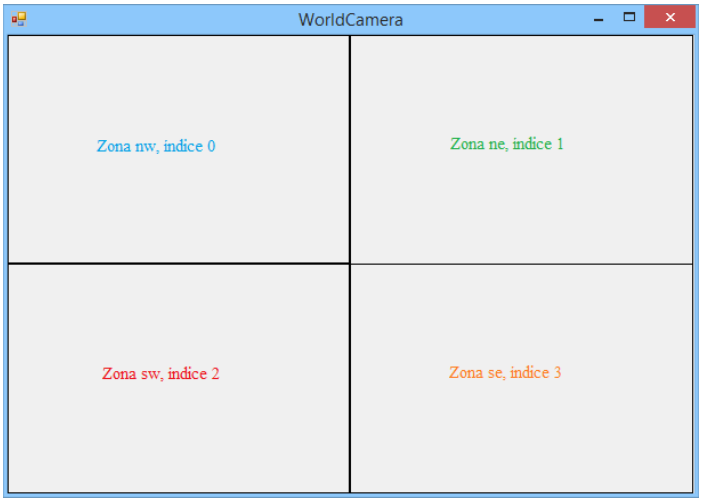
Esta clase representa un objeto del sistema de manera abstracta. Para su creación, recibe como parámetros una identidad y un rectángulo. Este último representa el área que cubre la entidad dentro del mundo simulado. Por otra parte, la identidad será la misma que la de la entidad a imitar, y hacemos esto para distinguir específicamente cada Boundary dentro del árbol, y no sólo por su rectángulo, ya que puede darse el caso de que se solapen perfectamente. Además, en su implementación, se le almacena un color que será usado luego para su representación gráfica.

Quadtree

Esta estructura no sólo contiene Boundaries para separarlos de manera eficiente, sino que ella en sí también es un Boundary que usaremos para comprobar otros rectángulos están contenidos, se intersectan o no con este nivel. Por lo tanto lo único que se necesita para crear un Quadtree es un boundary que contenga el área a cubrir.

Dentro, el Quadtree contiene una array con sus 4 posibles hijos, inicialmente vacío, junto con un booleano que nos indican si están creados o no, también una estructura de datos con los objetos en este nivel y el número de objetos en este nivel. Además, para agilizar cálculos, se guarda la mitad de la anchura y la altura de este área y se precalculan las áreas de los hijos.

[16] https://en.wikipedia.org/wiki/Quadtree



Las áreas de los hijos están divididas en 4, noroeste, noreste, suroeste y sureste, que las se numeran con índices del 0 al 3 respectivamente, tal y como se puede ver en la figura.

Por último se declaran dos funciones para usar en los métodos de la clase, estas funciones son matchBounds y getBounds.

Figura 9. Quadtree y las zonas correspondientes a los subárboles hijos.

La primera recibe como parámetro un rectángulo y devuelve el índice asociado al hijo que lo contiene (de 0 a 3) o -1 en otro caso. Esto lo hace comprobando que el área de cada hijo contenga por completo al rectángulo, es decir, usando el método Contains[17] de la clase Rectángulo que nos ofrece .NET.

La segunda función recibe un número como parámetro y devuelve el Boundary correspondiente dentro de los hijos (de 0 a 3) o este propio Boundary en otro caso.

A continuación se exponen los métodos implementados para esta clase.

Insert

Para insertar en el árbol un elemento se presentan dos casos principales y dentro de cada uno de estos, otros dos casos, haciendo un total de 4 casos posibles. Los casos principales son si tiene los hijos creados o no los tiene. Es decir, si ya se ha subdivido el árbol o no. Vamos a empezar por el segundo, con un árbol vacío.

Si el árbol está vacío primero comprobamos el máximo número de elementos recomendados que puede almacenar el árbol sin tener que subdividirse. Este número suele ser 4 y así está establecido en el programa. Se comprueba pues que la lista que contiene los objetos de este nivel sea menor que 4. Si es así, simplemente añadimos el objeto a la lista de objetos que están en este Quadtree, teniendo cuidado de incrementar el contador de objetos contenidos.

[17] http://msdn.microsoft.com/en-us/library/system.drawing.rectanglef(v=vs.110).aspx

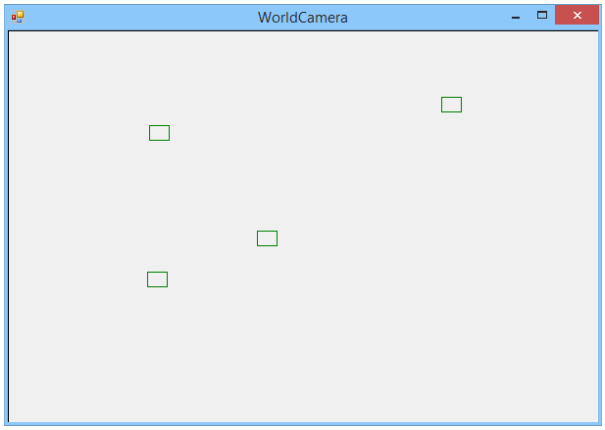


Figura 10. Quadtree con 4 elementos.

En el caso de que existen 4 elementos o más, se deben crear los hijos para dividir el trabajo. Este es el caso computacionalmente más pesado del método Insert. Primero se crean 4 Quadtrees asignándoles el área a cubrir con la ayuda del método getBounds que usará las áreas que precalculamos antes y se establece a verdadero la variable booleana que nos informa de que los hijos ya están creados.

Figura 11. Quadtree con 5 elementos y subárboles.

El siguiente paso es reordenar los elementos que ya estaban en este nivel del árbol. Para ello se hace una copia de los elementos y se reestablece la estructura original para que no contenga elementos. Ahora simplemente se recorre la copia mientras se llama al método Insert por cada elemento, con lo cual quedarán todos los elementos ordenados en su respectivo lugar. Por último se inserta el elemento objetivo en este mismo nivel y el propio algoritmo lo pondrá en su sitio.

Por otro lado, está el caso principal de que los hijos ya estén creados. Este caso es mucho más simple y se usa el método matchBounds descrito más arriba. Si devuelve -1 quiere decir que este objeto no encaja en ningún subárbol y por lo tanto se guarda en los objetos de este nivel. En otro caso siempre se devuelve el índice del subárbol donde tenemos que insertar el objeto, con lo cual simplemente se llama a Insert en ese subárbol.

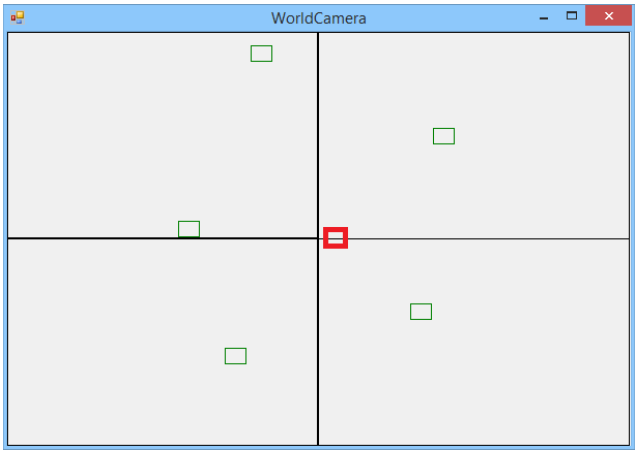


Figura 12. Un elemento que no está incluido en ningún subárbol.

Delete

Este método es muy fácil y rápido de implementar debido a que los Boundaries tienen, como ya se indicó antes, un campo identidad. Se vuelven a tener los dos casos principales que se tienen al insertar.

En el caso de que los subárboles no estén creados simplemente se recorre la estructura que guarda los objetos de este nivel comprobando su identidad. En este caso se ha implementado con una lista, así que usaremos el método RemoveAll[18]:

```
boundariesInThisLevel.RemoveAll(fun e -> e.Id = target.Id)
```

En el otro caso, los subárboles sí están creados, se vuelve a usar la función matchBounds para comprobar dónde está el objeto a eliminar, delegando en el hijo la tarea o eliminando como se ha visto antes si no está en ningún hijo.

Usamos el método RemoveAll porque puede quitar muchos elementos a la vez del árbol. Esto es conveniente porque no siempre tendremos criaturas con identidades distintas dentro del árbol, sino también otro tipo de objetos con los que pueden colisionar y quizás convenga que tengan la misma identidad.

Por último, el método RemoveAll devuelve el número de objetos eliminados dentro de la lista. Este número se usa para reducir el número de objetos que están incluidos en el nivel que se elimina y además, se devuelve como respuesta al final del método. Con lo cual, si el resultado es 0 significa que no se ha eliminado ningún objeto, y si es mayor que 0, significa que sí.

[18] http://msdn.microsoft.com/en-us/library/wdka673a(v=vs.110).aspx?cs-save-lang=1&cs-lang=fsharp#code-snippet-1

CollisionWith

Este es sin duda el método más importante y por el que se implementa el Quadtree. Sin embargo, una vez que ha entendido cómo se introducen y eliminan objetos del árbol, es intuitivo de implementar. Recibe como entrada un Boundary objetivo y devuelve la lista de objetos con la que hace colisión.

Primero se calcula en qué cuadrante encaja el objetivo y creamos una lista vacía de colisiones. Esta vez no tenemos dos casos principales por la sencilla razón de que un objeto puede colisionar con objetos que no están en su nivel solamente, sino también con aquellos que no están en ningún subárbol, es decir, los objetos que están en el nivel del padre, y del padre del padre, etc. Esto se hace por eficiencia computacional. Debido a que el número de objetos que no encajará en ninguno de los árboles suele ser pequeño en niveles superiores del árbol y suele ser grande conforme los subárboles se hacen cada vez más pequeños, el número de comprobaciones innecesarias en niveles superiores es mínimo con la gran ayuda que nos ofrece en niveles inferiores.

Por lo tanto sólo queda el caso de que los hijos estén ya creados. En este caso, si el objetivo encaja en alguno de los hijos, simplemente delegamos en el hijo la búsqueda de colisiones, añadiendo el resultado que devuelva a la lista que ya tenemos en este nivel. Esto se hace con el método AddRange[19] que nos ofrece .NET para las listas.

Por otro lado, si el objetivo no encaja en ninguno de los hijos, debemos comprobar con qué hijos está tocando este objeto para comprobar solamente las colisiones con estos subárboles y no otros, ya que cada subárbol puede contener muchos objetos y este es un punto importante donde optimizar. Para saber con qué subárbol está tocando comprobamos con los 4 hijos la intersección entre ese hijo y el objetivo usando el método IntersectsWith[20].

[19] http://msdn.microsoft.com/en-us/library/z883w3dc(v=vs.110).aspx
[20] http://msdn.microsoft.com/en-us/library/system.drawing.rectangle.intersectswith(v=vs.110).aspx

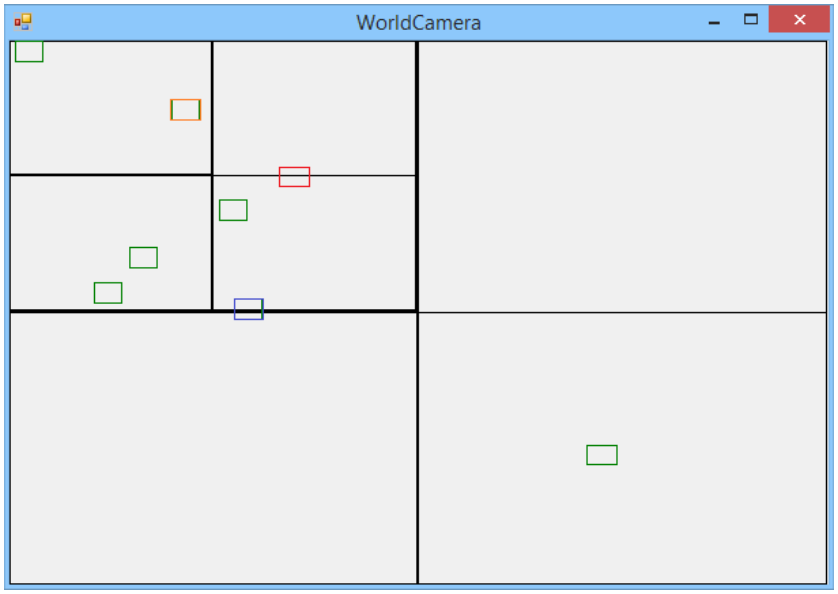


Figura 13. Posibles casos de colisión en un Quadtree.

Las posibles colisiones que se comprueban para el objeto remarcado en naranja son el objeto que está en su nivel, el objeto remarcado en rojo y el objeto remarcado en azul. Para el objeto que está abajo a la derecha, sólo se comprueba la colisión con el objeto azul. Para el objeto azul por su parte se comprueba con el cuadrado inmediatamente encima de él y el objeto rojo.

Como se puede observar, el número de comprobaciones es realmente pequeño y es clave para el rendimiento del programa.

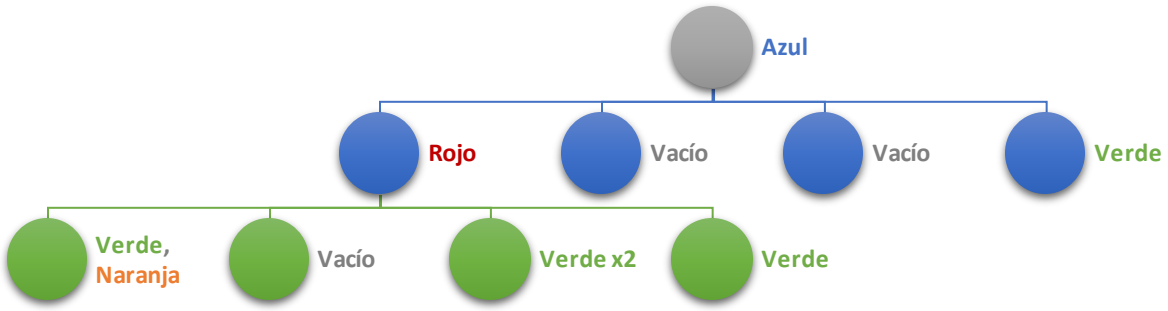


Figura 14. Figura jerárquica del Quadtree de la figura anterior.

Show y Draw

Estos métodos representan por pantalla el Quadtree para que el usuario pueda ver qué es lo que pasa dentro de él y ver cómo interactúan las colisiones. Simplemente llaman al método de dibujado o salida por consola que proporciona .NET por cada elemento de su subárboles y objetos en su propio nivel.

QuadTreeHolder

Por último, se ha implementado el patrón Singleton para mantener el árbol, ya que sólo habrá un árbol por mundo y se puede acceder a él fácilmente.

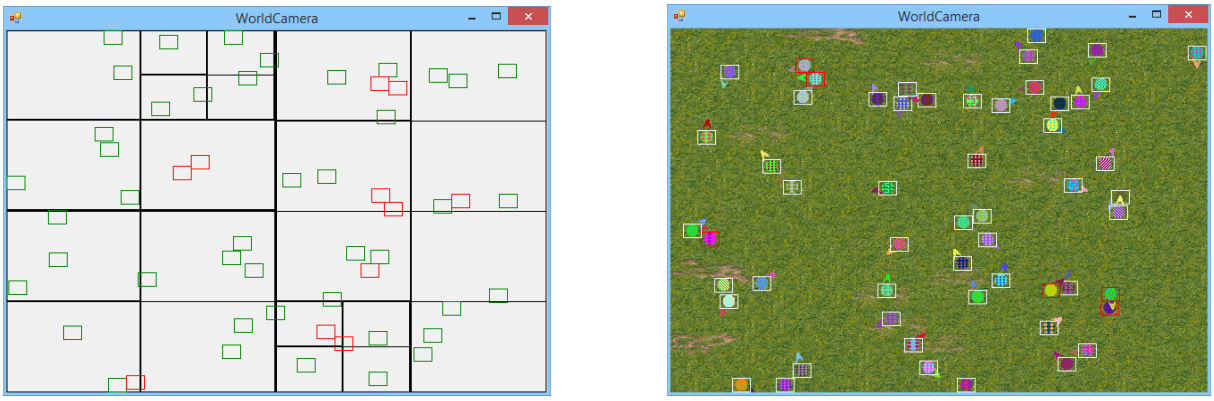


Figura 15. Colisiones vistas desde el Quadtree y sin Quadtree.

Ray casting

Una vez que tenemos definidas las colisiones con el Quadtree surge otro problema. Si nos fijamos en las figuras anteriores, podemos ver que dos entidades y concretamente las criaturas, solo colisionan cuando sus rectángulos de colisión hacen contacto. Sin embargo la cabeza del individuo puede "atravesar" el rectángulo de otra entidad debido a que el rectángulo de colisión de las criaturas es más ancho que alto, y no abarca también a la cabeza. Además, imposibilita casi por completo la detección de colisiones en movimientos oblicuos.

Para arreglar esto vamos a usar los bigotes de las criaturas, que definimos anteriormente en el capítulo 2. La idea es utilizar estas líneas y comprobar si hace intersección con algún rectángulo dado para detectar colisiones. Esto se conoce como "Ray casting"[21] en inglés, aunque debido a que nuestras líneas tienen una longitud determinada, más bien podría llamarse "Segment casting".

La idea para saber cuándo tenemos una intersección con una de las rectas es crear pequeñas rectas con los puntos del otro polígono que queremos probar. Como nuestro sistema de colisiones sólo lo componen rectángulos esto es realmente sencillo de hacer, tan sólo tenemos que obtener las rectas que unen las esquinas de éste.

Dado dos puntos de cada recta[22] podemos averiguar la diferencia en la ordenada x y en la ordenada y usando un conjunto de determinantes que pueden ser resumidos en la siguiente operación:

[21] http://en.wikipedia.org/wiki/Ray_casting
[22] https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection#Given_two_points_on_each_line

$$(P_x, P_y) = \left(\frac{(x_1 y_2 - y_1 x_2)(x_3 - x_4) - (x_1 - x_2)(x_3 y_4 - y_3 x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)}, \frac{(x_1 y_2 - y_1 x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 y_4 - y_3 x_4)}{(x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4)} \right)$$

Figura 16. Fórmula de la intersección entre dos rectas.

Si ambos términos de la tupla resultante esta entre 0 y 1, entonces las líneas se intersectan en algún punto y no lo harán en caso contrario. Si ambas rectas son paralelas o coinciden, entonces el denominador de la operación es 0.

Un vez hemos definido las intersecciones entre 2 rectas, simplemente tenemos que usar este método para comprobar la colisión con las líneas que componen el rectángulo. Si cualquiera de estas pruebas nos devuelve cierto, entonces tenemos una colisión.

Figura 17. Criatura mostrando el rectángulo de colisión y los bigotes.

Capítulo 4: Entidades

En este capítulo se describen las entidades del sistema. Las clases que lo componen harán que el mundo cobre vida, usando lo visto anteriormente y la inteligencia artificial descrita en el siguiente capítulo. Para ello todas las entidades se sincronizan y realizan su acción "casi" al mismo tiempo. En realidad se llaman linealmente a todas las entidades, pero el tiempo que se recibe es tan corto que no se nota esta linealización y parece que todas las entidades responden a la vez. Esto se hace con un temporizador que nos ofrece .NET y que veremos con detalle más adelante.

Entity

Esta entidad base del sistema recibe como parámetros un número natural que será la identidad, una figura del tipo Shape y por último recibe un manejador de inteligencia artificial que veremos más adelante cómo están implementado. También define el método abstracto Tick e Init para que las clases que lo hereden los definan. Estos métodos son los responsables de avisar a la entidad de que es su turno para realizar acciones y de inicializar el manejador de la inteligencia artificial de la entidad, respectivamente. Por último, todas las entidades tienen un campo accesible que indica al sistema si tiene que eliminar esta entidad del mundo, aparte de definir métodos para acceder rápidamente al boundary de la entidad, su identidad, figura etc…

Food

Esta entidad hereda los métodos y parámetros de Entity y representa a la comida para las criaturas. Añade como parámetro la cantidad de comida que tiene inicialmente y además mantiene la cuenta de la cantidad de comida restante. Cada turno, la cantidad de comida disminuye en 1, por lo que aunque no sea comida, tarde o temprano desaparecerá del sistema.

Existen dos tipos de comidas. Una es predefinida y representa a la vegetación que nos encontraríamos en el mundo real. La otra comida son las propias criaturas, que al morir quedan en forma de comida para los demás. De esta manera conseguiremos que las criaturas se atacan entre sí por el posible alimento que puedan proveerles la muerte de la otra. Cuando una criatura muere, copiamos su figura cambiando la cabeza al color negro y creamos un nuevo objeto de la clase Food.

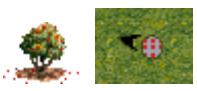


Figura 18. Tipos de comida.

Creature

Las criaturas son las entidades más importantes de este proyecto ya que de ellas dependen los comportamientos que intentaremos identificar. Las criaturas vuelven a heredar los parámetros y métodos de la clase Entity y añaden un parámetro importante, las propiedades de las criaturas. Estas propiedades se encapsulan en la clase CreatureProperties para modelar más fácilmente el sistema y a continuación la veremos en detalle. Definen varios métodos que son de especial utilidad para el manejador de la inteligencia artificial, y que devuelven el rectángulo sensorial y el rectángulo de proximidad de la criatura. Además, mantienen una lista de criaturas aliadas y enemigas que nos serán de utilidad más adelante.

Redefinen el método Tick para realizar sus acciones y actualizar parámetros como el hambre, el tiempo de vida o incluso marcarse como "muertas" para que el sistema las convierta en comida. En este capítulo nos vamos a centrar en las propiedades de las criaturas y más adelante veremos con detalle los parámetros relacionados con la inteligencia artificial.

Especies

Primero vamos a definir las especies de las criaturas. Una criatura sólo puede pertenecer a una especie pero se podrán reproducir con distintas especies si se da el caso. Una especie simplemente es una estructura (tipo) con los siguientes parámetros:

- id: Identidad de la especie
- lifeTimeRate: El ratio de tiempo de vida de esta especie. Las criaturas pueden vivir más o menos tiempo dependiendo de este parámetro.
- aggressivenessRate: El ratio de agresividad de esta especie. A mayor ratio mayor probabilidad de que ataquen a otras criaturas.
- hungryRate: El ratio de hambre de esta especie. Hay especies que necesitan alimentarse más a menudo que otras.
- speedRate y accelerationRate: Son los ratios de velocidad y aceleración de esta especie. La velocidad implica cuán rápido puede moverse por el mundo y la aceleración cuán rápido puede llegar esa velocidad o girar sobre sí misma.
- sensoryRate y closeThresholdRate: Son los ratios sensoriales y de proximidad que hemos introducido antes. Sirven para marcar como de buena es una especie a la hora de saber qué es lo que pasa a su alrededor.
- foodOffer: Es la cantidad base de comida que ofrece esta especie a otras criaturas, es necesario ya que no es lo mismo comerse un conejo que una vaca.
- shape: Las especies tienen su propia figura y tendrá que ser igual para todas las criaturas que son de esta especie. A veces se pueden dar pequeñas

variantes a la hora de la reproducción y son pequeñas mutaciones para hacer el sistema más interesante.

Figura 19. Distintas especies.

Se pueden crear nuevas criaturas en el sistema de 2 maneras distintas, manualmente eligiendo dónde se va a colocar la criatura o automáticamente cuando dos criaturas se juntan para procrear. En el primer caso es bastante sencillo y simplemente tenemos que crear una especie con parámetros totalmente aleatorios, asignándoles la siguiente identidad disponible para las especies. El segundo caso es un poco más complejo pero tampoco abarca mucha dificultad y solo podrá darse si los individuos que se juntan son de distinto sexo.

Primero obtenemos las especies de los dos padres y por cada parámetro elegimos aleatoriamente entre quedarnos el de un padre u el del otro. Lo hacemos con esta sencilla función:

```
let chooseOne a b =
      if randomG.NextDouble() > 0.5 then
            a
      else
            b
```

De manera que tendríamos por cada parámetro un código de la siguiente manera:

```
let mutable parámetro = chooseOne padreA.especie.parámetro
                        padreB.especie.parámetro
```

Una vez obtenido todos los parámetros base, intentamos mutar estos parámetros para aumentar la variabilidad del sistema. Para ello definimos una probabilidad de mutación por la cual se elige si un parámetro muta o no. Si se da el caso de la mutación modificamos ese parámetro por la media de la suma del parámetro de los padres y añadimos o quitamos una pequeña cantidad que está multiplicada por otro parámetro que indica el cambio que se realiza si hay mutación. Además, se mantiene un contador con el número de mutaciones que ha habido, ya que si es muy elevado, en torno a un 95%, daremos a esta especie por una nueva.

```
if randomG.NextDouble() < mutationRate then

      parámetro <- (padreA.especie.parámetro + padreB.especie.parámetro) / 2.0
+ randomG.Next(-1, 2) * mutationChange
```

Una vez que tenemos todas las mutaciones y hemos contado cuantas ha habido, creamos una nueva especie asignando estos parámetros que se han obtenido y la identidad de la especie, una nueva si se han dado el número suficiente de mutaciones.

Propiedades

Una vez definidas las especies en el sistema, podemos exponer las características individuales de cada criatura. Para ello se ha creado una clase que abarca todos estos componentes, la clase CreatureProperties, que recibe como único parámetro una especie tal y como hemos definido antes.

Esta clase mantiene distintos parámetros con distinto propósito y que vemos a continuación.

Propiedades individuales

Estas son las propiedades que marcan cómo es el individuo. Tenemos 4 parámetros:

- health: Representa la vida actual del individuo. Si es menor o igual que 0, esta criatura se da por muerta.
- ticksLived: Es el número de ticks que ha vivido la criatura. Cuando este número es igual al número de ticks que puede vivir una especie, la criatura se da por muerta.
- sex: Es el sexo de la criatura. Dos criaturas sólo podrán procrear si tienen distinto sexo. Sólo hay dos sexos, el tipo A y el tipo B.
- bornChilds: Es el número de hijos que ha tenido esta criatura. Se usará más adelante como inhibidor de las ganas de procrear de la criatura.

Necesidades

Estas dos propiedades marcan las necesidades actuales de la criatura. Actualmente son propiedades muy básicas:

- hungry: Representa el nivel de hambre que tiene esta criatura. Cada turno este nivel va aumentando dependiendo del ratio que vimos antes para cada especie. Tiene un máximo de 255, a partir del cual la criatura verá mermada su salud en cada tick si no es capaz de encontrar comida, pudiendo morir de inacción.
- procreation: Representa las ganas de procrear de la criatura. Esto es así para que una misma criatura no se dedique sólo a procrear y se olvide de lo demás.

En cada turno se ve incrementado en 1 dividido por el número de hijos que haya tenido. También tiene un máximo de 255.

Propiedades físicas

Estas propiedades son las que les da la propia especie:

- lifeTime: Representa cuánto tiempo puede vivir una criatura. Tiene un mínimo de 3530 ticks y un máximo de 7060, que vienen a ser de 2 a 5 minutos de vida.
- aggressiveness: Representa cómo de agresiva es la criatura, tiene un máximo de 255.
- closeWide: Mide el radio por el cual una criatura se sentiría especialmente cerca de otra entidad y le permite responder antes a posibles colisiones. Tiene un mínimo de 20 y un máximo de 30.
- senseWide: Mide el ratio por el cual una criatura puede detectar a otras entidades. Tiene un mínimo del mismo tamaño que closeWide y un máximo de 100.
- maxSpeed y maxAcceleration: Representan la máxima velocidad y aceleración con la que la criatura puede moverse por el mundo. Tienen un mínimo de 0.5 y un máximo de 2.
- hungryRate: Mide el ratio por el cuál a una criatura le entran más o menos ganas de comer, esto hará que haya criaturas que necesiten comer menos que otras. Va de un mínimo de 0.1 hasta un máximo de 2
- foodOffer: Es la cantidad de comida que ofrece esta criatura al morir, tiene rango desde 0 hasta 255.

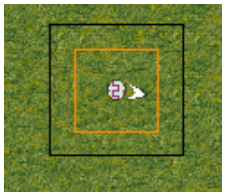


Figura 20. En closeWide (naranja) y senceWide (negro) de una criatura.

En la siguiente figura se muestra el diagrama UML de las entidades, donde podemos observar las relaciones que se dan.

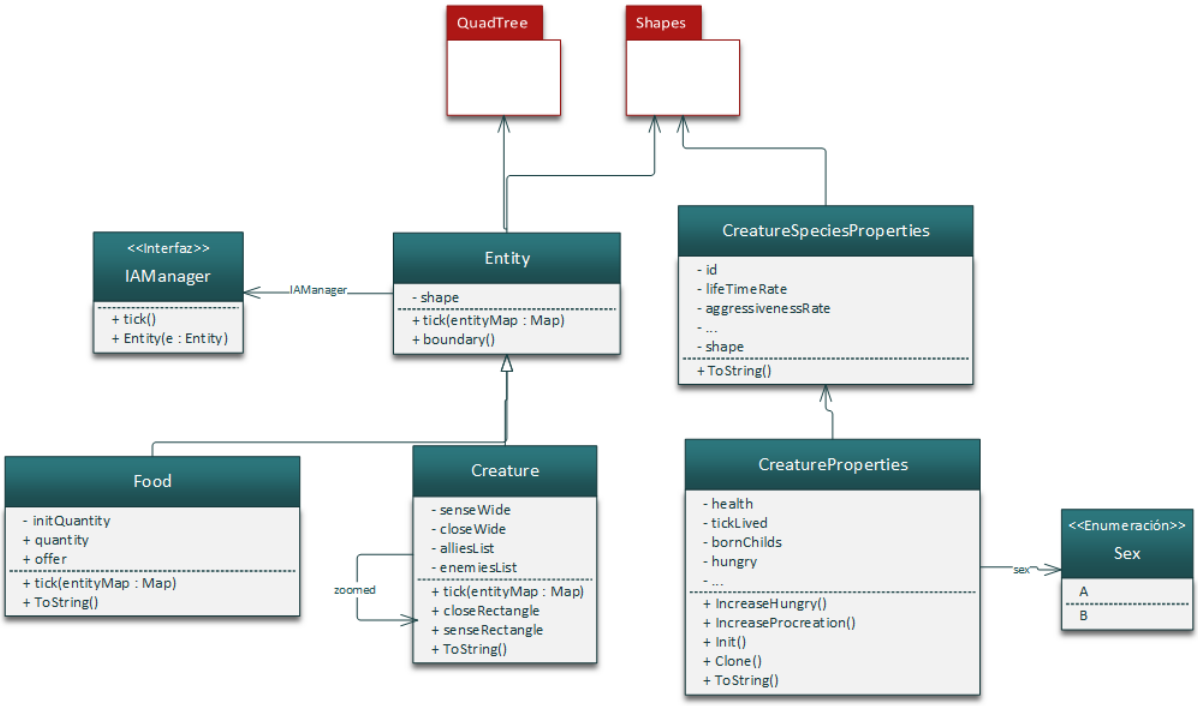


Figura 21. Diagrama UML de las entidades.

Capítulo 5: Controlando el mundo

Las criaturas necesitan un mundo para vivir y desde donde puedan ser controladas. En este capítulo vamos a ver la clase que controla el mundo y mantiene las entidades. También veremos cómo se ha implementado el multiproceso para esta aplicación.

WorldManager

Esta es la clase principal del programa, ya que controla todo lo que pasa en el mundo, desde turnar a las criaturas para que hagan sus acciones hasta pintar todos los elementos en pantalla. Para construir esta clase necesitamos indicarle varios parámetros:

- biome: El bioma representa dónde viven las criaturas. Lo usaremos para decidir cómo será el mundo donde vivan las criaturas. Existen 3 biomas: Continental, Desértico y acuático.
- worldName: Es el nombre de este mundo, lo usaremos para distinguir unos mundos de otros más adelante, cuando podamos guardar el mundo en una base de datos.
- worldWidth y worldHeight: Indican el ancho y alto del mundo, respectivamente. Necesitamos estos parámetros para que las criaturas no se "salgan" del mundo y permanezcan dentro de él.

Además, esta clase mantiene 3 listas: Una para la comida, otra para las criaturas y otra que combina las dos, para las entidades. También mantiene 3 contadores para el número de entidades, criaturas y uno especial que indica el número de la última identidad concedida. Por último, tenemos un mapa que relaciona la identidad de una criatura con ella misma, de manera que podemos acceder rápidamente a una criatura conocida su identidad.

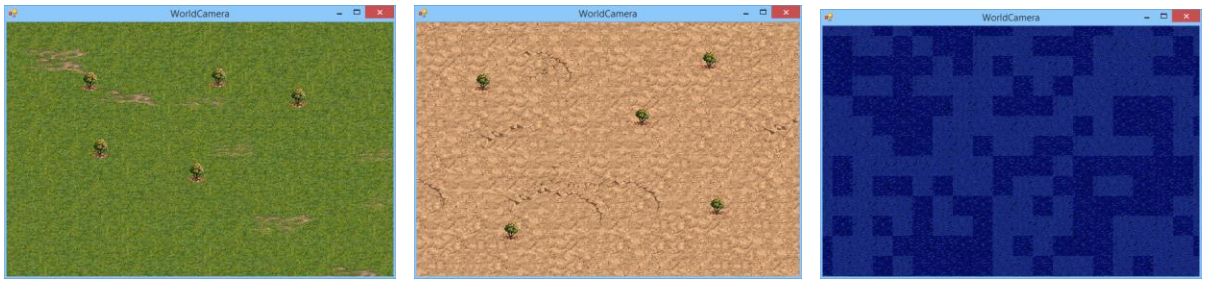


Figura 22. Biomas: Continental, Desértico y Acuático

Esta clase tiene varios métodos útiles que se van a exponer a continuación.

Crear

Al crear el mundo se realiza un pre-trabajo que ayuda luego a la hora de dibujar el mundo. Este trabajo trata de crear el fondo de pantalla para el mundo. Esto lo hacemos dependiendo del bioma elegido y seleccionamos aleatoriamente texturas del ResourceManager que tengan que ver con el bioma.

Tenemos que elegir el tamaño del mundo divido entre 32 más 1 texturas para el fondo de pantalla. Esto es debido a que queremos pintar "por fuera" del propio mundo, ya que sino en las esquinas del mundo se puede pintar un poco de blanco. Además, en el caso de biomas continentales o desérticos añadimos pequeñas texturas para darle detalles al mundo.

Una vez que tenemos elegido el fondo de pantalla, creamos un objeto Graphics[23] y pintamos este fondo en un bitmap. De esta manera tendremos una imagen con el fondo de pantalla del mundo y sólo tendremos que pintar las entidades encima de ella, lo cual mejora drásticamente el dibujado que veremos a continuación.

Draw

Ya hemos visto varias figuras donde se muestra el mundo con sus criaturas, pero no sabemos cómo se dibujan. Para este propósito usamos las herramientas que nos ofrece .NET para el dibujado en formularios. Usando el objeto Graphics que nos envía el formulario cuando queda invalidado, dibujamos el mundo en 2 turnos.

Ya hemos visto como obtenemos el fondo de pantalla, que podemos pintar directamente usando el modo NearestNeighbor como modo de interpolación. Este modo sirve para reducir la carga de dibujado y tarde menos tiempo, ya que el dibujado por pantalla puede llegar a ser la parte más costosa de nuestra aplicación.

Para el dibujado de las entidades hemos creado un objeto Graphics y un bitmap auxiliar que se mantienen durante toda la aplicación para no crear continuamente estos objetos. La idea es pintar en blanco este bitmap auxiliar en cada turno con el método Clear de la clase Graphics y usando el modo de composición SourceCopy[24], que nos permite sobre escribir el fondo. Una vez hecho simplemente iteramos sobre las entidades dibujando sus figuras:

```
entityList |> Seq.iter(fun item -> item.Shape.Draw(ga))
```

También pintamos, si está así elegido, los rectángulos sensoriales y de proximidad que hemos visto anteriormente. Una vez terminado el dibujado sobre esta imagen auxiliar, la pintamos sobre el objeto Graphics que nos mandó al principio.

[23] https://msdn.microsoft.com/en-us/library/system.drawing.graphics(v=vs.110).aspx
[24] https://msdn.microsoft.com/en-us/library/system.drawing.drawing2d.compositingmode(v=vs.110).aspx

InsertCreature

Este método introduce una nueva criatura en el mundo. Esto sólo se hará si el número máximo de criaturas no ha sido sobrepasado. Éste método recibe como parámetros un punto y una creatureProperties que hemos visto antes. Esto es así porque, como veremos luego, crearemos directamente desde la interfaz nuevas especies. Una vez comprobado que podemos crear la criatura actualizamos el centro de la figura que se nos pasa con creatureProperties con el punto dado. Esto lo hacemos porque cuando creamos a la criatura no sabemos dónde la vamos a posicionar, sólo podemos saberlo en este punto. Así pues, simplemente creamos una criatura pasándole como parámetros su identidad, sus propiedades y un nuevo manejador de inteligencia artificial que veremos más adelante. Añadimos esta criatura a las listas de entidades, de criaturas y por supuesto, al Quadtree. Finalmente se incrementa en 1 los parámetros necesarios.

InsertFood

Este método es casi igual que el anterior, sólo difiere en que la figura de la comida puede ser tanto una imagen como el cuerpo muerto de una criatura. Por ello pasamos como parámetros a esta función el punto donde queremos situar la comida y además, usando el tipo opción, una criatura.

Las comidas normales (no cuerpos de criaturas) tienen una capacidad inicial de 5000. En el caso de que pasemos el cuerpo de una criatura, la capacidad inicial será la que ofrecía a las otras criaturas, tal y como vimos en el capítulo anterior en las propiedades de la criatura, más 1000 de base. Luego, al igual que en el método anterior, actualizamos las listas, el Quadtree y el número de objetos del sistema.

DeleteObject

Como su nombre indica, este método borra una entidad del sistema dada una identidad. Esto lo hace simplemente llamando a las listas que mantienen a las criaturas y los individuos utilizando esta información. También elimina a la criatura del Quadtree y decrece los contadores oportunos.

CreateCreature

Este método se llama cuando dos criaturas procrean, por lo que recibe como parámetros a estas dos criaturas. Es parecido al método que vimos anteriormente para insertar una criatura en el sistema, pero esta vez elegimos cuasi manualmente una posición donde colocar esta criatura. Esta posición siempre será al norte, este, oeste o sur del primer padre. Si el número de criaturas no supera al máximo establecido podemos empezar a crear la criatura.

El procedimiento se basa en probar en estas posiciones si la nueva criatura entraría en colisión o no con alguna otra criatura del sistema, por lo que usando el Quadtree es realmente sencillo de comprobar. Si encontramos un hueco en algunas de estas posiciones y el mundo contiene a la nueva posición (no podemos crear criaturas fuera del mundo), entonces podemos crear a esta nueva criatura. El procedimiento ya lo vimos a la hora de crear especies y propiedades de criaturas, así que simplemente llamamos a ese bloque de código e insertamos la criatura en el punto dado como hacíamos en el método anteriormente descrito.

Por último tenemos que añadir a la lista de aliados de los padres a su hijo, y con este último haremos lo mismo con los aliados de sus padres, ya que si da la casualidad de que el hijo o los aliados de los padres son de distinta especie, acabarán atacándose entre ellos casi instantáneamente debido a su proximidad.

Tick

Por último vamos a definir el método más importante del controlador del mundo. Este método se encarga de actualizar la información del sistema y de dar turnos a las entidades para que hagan su trabajo.

Para ello se comienza leyendo la lista de procreación. No hemos hablado todavía de esta lista porque no habíamos introducido el método que usan, que es precisamente el de CreateCreature. Cuando dos criaturas consiguen ponerse de acuerdo para tener un retoño, no se crea inmediatamente, sino que se envía esta petición a una lista del controlador del mundo. Esto lo hacemos para evitar posibles fallos en el Quadtree y, como veremos más tarde, con el multiproceso. Lo que hacemos es leer esta lista y llamar al método CreateCreature con los padres.

Terminada la reproducción de todos los individuos, limpiamos la lista de procreación para dar lugar al turno que nos ocupa. También borramos el Quadtree y todas las entidades que están marcadas para borrar, como vimos en el capítulo 2.

Una vez limpiado el sistema, añadimos todas las entidades al Quadtree y actualizamos el mapa con las nuevas entidades. Por último simplemente por cada entidad llamamos a su método Tick pasándole como parámetro el mapa con las entidades.

Multiprocesamiento

A estas alturas, seguramente el lector estará pensando que si tenemos muchas criaturas el sistema puede sobrecargarse e ir lento ya que solamente estamos usando una CPU[25]. Con las pruebas realizadas llegamos a un 30% de capacidad máxima de la CPU con un solo proceso y notamos cierta ralentización. Sin embargo, al activar el multiprocesamiento el uso aumenta hasta un notable 60%. En algunas pruebas con millares de individuos se ha llegado al 100% del uso de las CPU.

Al contrario de lo que se podría pensar, esto no es malo en absoluto. Debido a que lanzamos la aplicación desde un entorno de ventanas, cuando usamos una sola CPU estamos usando el hilo principal del manejador de ventanas, el cual al realizar tareas de dibujado y también el cálculo de todas las criaturas llega hasta su capacidad máxima rápidamente, que por las pruebas realizadas es de un 30% del uso de la CPU. No he encontrado ninguna verificación en internet que avale estos resultados, aunque sí existe documentación sobre los hilos de Windows Form.[26]

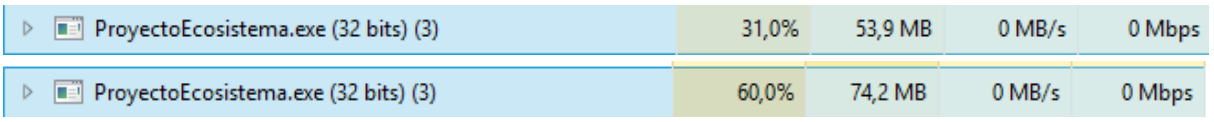


Figura 23. Diferencia uso de CPU entre monohilo y multihilo.

Para conseguir esta mejora primero necesitamos separar el procesamiento del mundo del hilo principal que maneja las ventanas de dibujado.[27] Para ello usaremos un BackgroundWorker[28] a la hora de llamar al manejador del mundo, teniendo cuidado de no llamarlo mientras está trabajando[29].

Aunque esto ya es un poco de multiprocesamiento, vamos a ir aún más lejos haciendo que grupos de entidades utilicen distintos recursos (procesadores) si es posible. F# nos ofrece 2 maneras de hacer esto: Usando el módulo Async de F#[30] o usando las librerías de hilos de .NET.

El módulo Async de F# nos ofrece una manera rápida y sencilla de paralelizar nuestro cómputo, pero en esta aplicación se hace complicada la separación de entidades y la espera a que todas terminen usando este método. Por este motivo he elegido usar la librería de .NET.

La idea a la hora de utilizar varios procesadores es poder asignar a cada uno una cantidad equitativa de entidades que procesar. Además tenemos que esperar a que todas terminen para poder continuar nuestro cómputo, como vimos antes en el

[25] https://en.wikipedia.org/wiki/Central_processing_unit
[26] https://msdn.microsoft.com/en-us/library/ms229730(v=vs.80).aspx
[27] https://msdn.microsoft.com/en-us/library/ms171728(v=vs.80).aspx
[28] https://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker(v=vs.80).aspx
[29] https://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.isbusy(v=vs.110).aspx
[30] https://en.wikibooks.org/wiki/F_Sharp_Programming/Async_Workflows

manejador del mundo. Utilizando solamente los hilos que nos ofrece .NET[31] podemos llegar a conseguirlo, pero tiene un problema fundamental, tenemos que estar creando continuamente nuevos hilos cada vez que queremos realizar el cómputo[32]. Para evitar esto existen las ThreadPool[33].

Las ThreadPool funcionan ofreciendo hilos que podemos usar para realizar nuestras tareas, de manera que simplemente tenemos que encolar el bloque de código que queremos llevar a cabo y el manejador se encargará de avisarnos cuando termine el cómputo. En nuestro caso usamos el método QueueUserWorkItem[34]. Este método recibe como parámetros un WaitCallBack[35], que a su vez recibe como parámetro la tarea a realizar, y un ManualResetEvent[36].

Usando el WaitCallBack, la piscina de procesamiento nos avisará cuando la tarea ha terminado, y una vez que haya terminado usaremos el ManualResetEvent para indicar a nuestro programa que ha terminado. Es importante hacer estos dos pasos ya que necesitamos esperar a que todas las entidades terminen de trabajar de manera asíncrona para realizar la sincronización en el manejador del mundo y no lanzar cada hilo individualmente y esperar a que termine.

Primero debemos crear una tarea por cada unidad lógica de procesamiento que tenga nuestro sistema. Para hallar este número simplemente llamamos a Environment.ProcessorCount. A cada una de estas tareas tenemos que asignarles unas entidades. Se les asigna usando un método de rotación, por el cuál mientras queden entidades se va asignando circularmente una a una a un gestor de tareas diferente. Este gestor de tareas es una clase que he definido y que se encarga de llamar al método Tick de todas las entidades que contiene.

```
type WorldThread(list : List<Entity>) =
      let mre = new ManuaResetEvent(false)
      member this.Task(obj : Object) =
            list.ForEach(fun item -> item.Tick(entityMap))
            (obj :?> ManualResetEvent).Set() |> ignore
      member this.MRE with get () = mre
      member this.ResetMRE () = mre.Reset() |> ignore
      member this.Clear() = list.Clear()
      member this.Add(e) = list.Add(e)
```

[31] https://msdn.microsoft.com/en-us/library/system.threading.thread(v=vs.110).aspx
[32] http://stackoverflow.com/questions/1054889/restarting-a-thread-in-net-using-c
[33] https://msdn.microsoft.com/en-us/library/system.threading.threadpool(v=vs.110).aspx
[34] https://msdn.microsoft.com/en-us/library/4yd16hza(v=vs.110).aspx
[35] https://msdn.microsoft.com/en-us/library/system.threading.waitcallback(v=vs.110).aspx
[36] https://msdn.microsoft.com/en-us/library/system.threading.manualresetevent(v=vs.110).aspx

Una vez declarada esta clase, creamos un array con el número de procesadores lógicos del ordenador que contenga a esta clase.

```
let THA = [| for i in 0 ..(logicProcessors - 1) -> new WorldThread(new
            List<Entity>)|]
```

Ahora simplemente realizamos el trabajo descrito anteriormente. Por cada turno de trabajo limpiamos las listas y volvemos a asignar las entidades, ya que puede haber entidades nuevas o que hayan desaparecido. Luego reinicializamos el MRE y encolamos los trabajos en la ThreadPool.

```
for i = 0 to THA.Length - 1 do
      THA.[i].Clear()
for i = 0 to THA.Length - 1 do
      THA.[i % logicProcessors].Add(entityList.Item(i))
for i = 0 to THA.Length - 1 do
      THA.[i].ResetMRE()
      ThreadPool.QueueUserWorkItem(new WaitCallBack(THA.[i].Task), THA.[i].MRE)
      |> ignore
WaitHandle.WaitAll([| for i in 0.. (THA.Length - 1) -> THA.[i].MRE :>
                    WaitHandle |]) |> ignore
```

Con estos tres bloques de código hemos conseguido una manera simple y efectiva de manejar todos los hilos necesarios para la aplicación, que usando cualquiera de los otros dos métodos no hubiese sido posible.

Sólo nos falta una cosa por ver. Debido a la naturaleza del problema, luego veremos cómo las criaturas interaccionaran entre ellas y podrán mandarse mensajes, añadirse o quitarse de listas. Esto por supuesto da lugar a innumerables situaciones de errores debido al acceso concurrente de estas variables. Para evitar esto F# nos ofrece la palabra clave lock[37].

Lock recibe como parámetros un objeto y una función. Obliga a la exclusión mutua de ese objeto mientras se esté aplicando la función, con lo cual para evitar los problemas simplemente tenemos que hacer un lock al objeto que queramos y realizar la tarea. Sin embargo, normalmente el número de objetos al que se accede de manera concurrente suele ser elevado y lograr una buena exclusión mutua puede ser laborioso.

[37] https://msdn.microsoft.com/en-us/library/ee370413.aspx

```
try
        lock this trabajar
with
| _ -> ()
```

Este método expuesto es lo que al final llama el método Tick del manejador del mundo, donde trabajar son los pasos que hemos descrito antes.

Tenemos que usar lock con el controlador del mundo y capturar excepciones porque el hilo de dibujado de Windows Forms no lo podemos controlar e intentará continuamente usar los recursos, pudiendo lanzar InvalidOperationException si no consigue los recursos cuando él quiere.

Capítulo 6: Inteligencia Artificial

Una vez que hemos definido cómo son las entidades y cómo funcionará el mundo, podemos dar paso a cómo las criaturas van a interaccionar entre ellas y cómo se moverán por el mundo.

El motor de la Inteligencia Artificial

En los últimos años la creación de Inteligencia Artificial (IA a partir de ahora) ha cambiado mucho, influida principalmente por los videojuegos, los cuales explotan activamente este campo de estudio.

Inicialmente la IA era escrita para cada juego y para cada personaje. Por cada nuevo personaje en el juego o simulador debía existir un bloque de código que ejecutase su IA. Este código era controlado por un pequeño programa y no había necesidad de tomar decisiones.

Ahora existe una tendencia cada vez mayor a tener unas rutinas generales para la IA y que permiten a los diseñadores de niveles y artista crear personajes únicos más fácilmente. La estructura del motor está fijada y la IA de cada personaje se crea juntando componentes de una manera adecuada.

La estructura de un motor de Inteligencia Artificial

Existen unas estructuras básicas que debemos identificar en un sistema de IA. Primero necesitamos algún tipo de infraestructura en dos categorías: Un mecanismo general para controlar los comportamientos de la IA (decidir qué comportamiento se ejecuta y cuándo) y una interfaz con el mundo para recopilar la información que necesita la IA. Segundo, necesitamos llevar a la pantalla lo que decida la IA. Esto consiste en pequeñas interfaces para el movimiento y la animación, que convierta peticiones como “anda silenciosamente hasta x,y” en acción. Y por último necesitamos una estructura que junte a estas dos anteriormente descritas.

Por supuesto, todo esto tiene que ser pensado con anterioridad al desarrollo del simulador o juego. Aunque en esta memoria la IA está casi al final, en realidad fue pensada y estudiada antes incluso de saber cómo iban a ser las entidades que componían el mundo.

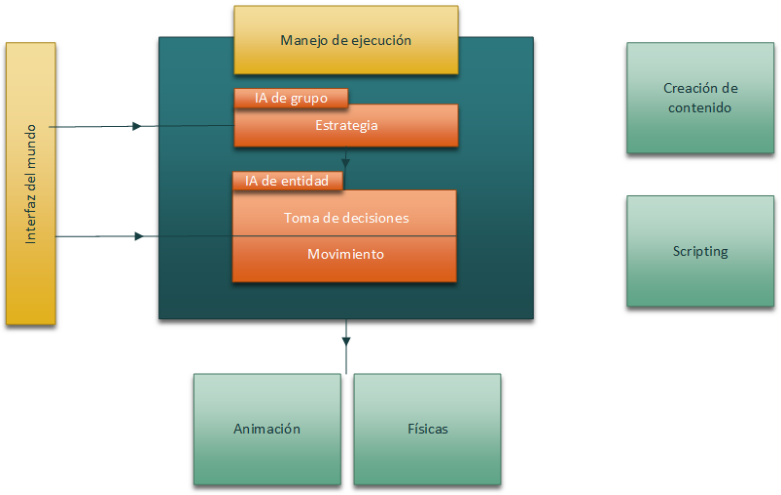


Figura 24. Modelo de la inteligencia artificial.

Movimientos básicos

Una de las partes más importantes de la IA es mover los personajes por el mundo de manera correcta. Incluso los primeros personajes controlados por IA (los fantasmas de Pacman por ejemplo) utilizan algoritmos de movimientos que no han sido sustituidos en los videojuegos hasta hace muy poco. El movimiento es la capa más baja en las técnicas que vamos a usar y, aunque lo veamos lo primero, será lo último que apliquemos antes de aplicar el resultado de las decisiones en el mundo.

Personajes en 2 dimensiones (el caso que nos ocupa) poseen un vector que representa la posición dónde se encuentra el personaje y además un parámetro de orientación que indica hacia dónde está mirando el personaje. Además tenemos que guardar un vector de velocidad, que será con la fuerza y dirección que se mueve el personaje actualmente. Estos datos se encapsulan en una clase que llamamos Kinematic.

Además se ha creado un tipo vector con elementos sobrecargados para facilitar la aplicación de las funciones matemáticas y que pudimos ver un poco de él en un ejemplo del capítulo 1.

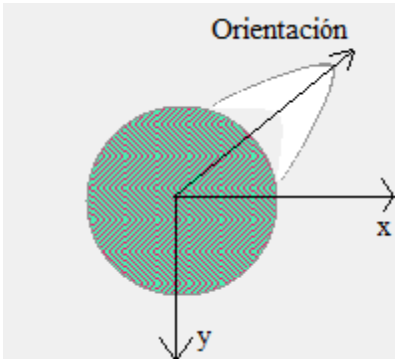


Figura 25. Posición de una criatura respecto al eje de coordenadas.

Cómo podemos ver en la figura anterior, el eje de coordenadas está invertido en nuestra aplicación. Esto es así en todos los dibujados por ordenador, donde se considera la esquina superior izquierda como el punto (0,0) y se incrementa hacia la derecha y hacia abajo. Esto hay que tenerlo en cuenta a la hora de aplicar las funciones matemáticas que veremos, ya que estas funciones están modeladas para un eje de coordenadas normal, al que estamos acostumbrados.

Cuando aplicamos los algoritmos de movimiento tenemos como respuesta la aceleración con la que cambiará la velocidad del personaje para moverlos por el nivel. Por ello creamos el tipo SteeringOutput que contiene un vector lineal y un parámetro angular. Este tipo será el devuelto por todos los algoritmos que vamos a ver de movimiento.

Una vez que tenemos definidas las dos estructuras principales vamos a definir los dos métodos que tiene la clase Kinematic. Uno es el de clonación, que nos hará falta cuando queramos comprobar si cierta decisión nos lleva a un punto correcto del sistema.

El otro método es el de actualización, es decir, aplica el resultado de un algoritmo de movimiento a este Kinematic. Sus entradas son el SteeringOutput del algoritmo, una velocidad máxima y la unidad de tiempo actual del sistema. A continuación podemos ver el código:

```
member this.Update (steering : SteeringOutput, maxSpeed, time : float32) =
        this.Position <- this.Position + (this.Velocity * time)

        this.Velocity <- this.Velocity + (steering.Linear * time)
        this.Orientation <- this.Orientation + steering.Angular

        if (this.Velocity.Length) > maxSpeed then
            this.Velocity.Length <- 1.0f
```

El último paso es la normalización del vector[38]. Tenemos que hacer esto para que las fuerzas aplicadas sean iguales para todos los individuos.

Una vez entendido cómo funciona el posicionamiento y actualización de los personajes con respecto al mundo, podemos empezar a ver los algoritmos de movimientos. Estos algoritmos tienen una estructura muy similar. Siempre toman como entrada la Kinematic del personaje e información adicional, como otra Kinematic objetivo, la máxima aceleración, etc…

Algunas de estas entradas no siempre están disponibles, como por ejemplo la evasión de obstáculos necesita tener acceso a la información de colisiones del nivel. Esto puede ser un proceso muy costo ya que tenemos que anticiparnos al movimiento del personaje y de otros personajes usando ray cast o colisiones con probando el movimiento con el Quadtree. Algunos comportamientos operan en grupo de objetivos,

[38] https://es.wikipedia.org/wiki/Norma_vectorial

como por ejemplo el comportamiento de manada intenta resumir de alguna manera la posición media de toda la manada. Este comportamiento no ha sido implementado por falta de tiempo, pero como veremos en un momento realmente vamos a construir comportamientos apoyándonos en otros.

Buscar y Huir

Seek (buscar) intenta igualar la posición del personaje con la posición del objetivo. Simplemente encuentra la dirección hacia el objetivo y va hacía allí con la máxima velocidad posible. Como estamos usando aceleraciones para calcular el movimiento, esta velocidad crecerá más y más, por eso tenemos definida una máxima aceleración para los personajes. Para implementar Flee (huir), simplemente cambiaremos de orden la dirección en la que nos movemos. A continuación se expone el código:

```
let Seek (character:Kinematic) (target:Kinematic) (maxAcceleration:float32) =
    let steering = new SteeringOutput()
    let direction = (target.Position - character.Position)
    direction.Length <- 1.0f
    steering.Linear <- direction * maxAcceleration
    steering.Angular <- 0.0f

    Some steering
```

Por lo que una llamada del tipo “Seek char target acceleration” buscará al objetivo y una llamada del tipo “Seek target char acceleration” huirá del objetivo. Aunque para que sea más fácil más adelante, yo he duplicado la función y la he llamado Flee.

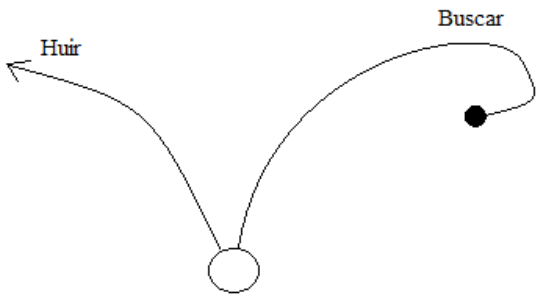


Figura 26. Buscar y huir.

Llegar

Seek siempre intentará moverse hacia el objetivo lo más rápido posible. Esto está bien si ese objetivo está continuamente moviéndose y el personaje necesita perseguirlo. Sin embargo si el objetivo está quieto, Seek “pasará” de largo, volverá, y se quedará dando vueltas alrededor del objetivo continuamente. Si un personaje quiere llegar hasta un objetivo, necesita frenar antes de llegar, así llegará exactamente

al lugar donde quiere llegar. Arrive (llegar) arregla este pequeño fallo, aunque es algo más difícil de implementar.

Utiliza dos radios, uno para pararse cuando creemos que estamos suficientemente cerca del objetivo, y otro radio más grande que hará que frenemos para tener una velocidad admisible para llegar al objetivo.

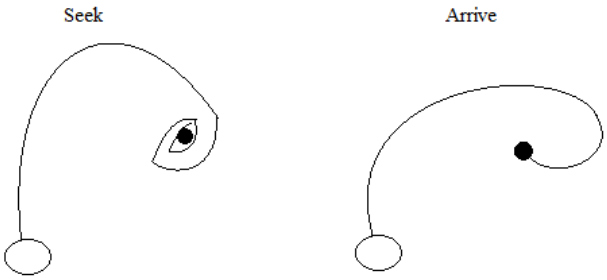


Figura 27. Buscando y llegando.

A continuación se expone el código:

```
let Arrive (character:Kinematic) (target:Kinematic) maxAcceleration maxSpeed
targetRadius slowRadius =
    let steering = new SteeringOutput()
    let timeToTarget = 0.1f
    let direction = target.Position - character.Position
    direction.Length <- 1.0f
    let distance = direction.Length
    let mutable ts = 0.0f
    let mutable targetVelocity = {vector = [|direction.[0]; direction.[1]|]}

    if distance < targetRadius then
        None
    else

        if distance > slowRadius then
            ts <- maxSpeed
        else
            ts <- maxSpeed * distance / slowRadius

        let targetSpeed = ts
        targetVelocity <- targetVelocity * targetSpeed

        steering.Linear <-  (targetVelocity - character.Velocity) /
                             timeToTarget

        if steering.Linear.Length > maxAcceleration then
            steering.Linear.Length <- 1.0f
            steering.Linear <- steering.Linear * maxAcceleration

        Some steering
```

El comportamiento contrario a Arrive sería Leave (dejar) pero no tiene sentido de la implementación de este comportamiento, ya que es raro que "dejemos" un objetivo. Además empezaríamos con muy poca aceleración (posiblemente 0) para ir aumentándola. Lo más seguro es que queramos acelerar lo máximo posible y para esto ya tenemos el comportamiento de huir que hemos visto anteriormente.

Alinear

Alinear intenta igualar la orientación de un personaje con la del objetivo. No presta atención a la posición o velocidad del personaje u objetivo. Debido a que las orientaciones dan una vuelta completa cada 2pi radianes, no podemos simplemente restar las orientaciones entre los personajes para determinar la rotación final, ya que nos encontraríamos con casos en los que para girar un poco el personaje daría una vuelta completa debido a la diferencia en nuestro cálculo.

Para encontrar la rotación correcta, tenemos que restar la orientación del personaje con la del objetivo y cambiar este resultado a un rango de (-pi, pi) radianes. Esto lo hacemos calculando el módulo de la rotación y comprobando si está dentro del rango, si no lo está restamos 2pi a este resultado, y si sí lo está simplemente lo devolvemos.

```
let mapToRange rotation =
    let r = rotation % DPI
    if r > Math.PI then
        r - DPI
    else
        r
```

Este método nos permite tener valores grandes de rotación y convertirlos al rango correcto. A continuación se expone el código del comportamiento Align (alinear).

```
let Align (character:Kinematic) (target:Kinematic) =
    let steering = new SteeringOutput()
    let rotation = float32 (mapToRange (float target.Orientation - float
                                        character.Orientation))
    let rotationSize = abs rotation
    let mutable targetRotation = 0.0f
    let maxAngularAcceleration = 2.0f
    let maxRotation = 0.2f
    let targetRadius = 0.1f
    let slowRadius = 0.5f

    // Ya hemos llegado
    if rotationSize < targetRadius then
        None
    else
        if rotationSize > slowRadius then
            targetRotation <- maxRotation
        else
            targetRotation <- maxRotation * rotationSize / slowRadius

        targetRotation <- targetRotation * rotation / rotationSize
        steering.Angular <- targetRotation

        let angularAcceleration = abs(steering.Angular)

        if angularAcceleration > maxAngularAcceleration then
            steering.Angular <- (steering.Angular / angularAcceleration) *
                                maxAngularAcceleration

        Some steering
```

Movimientos delegados

Hemos visto cómo se han construidos los comportamientos básicos que ahora nos ayudarán a construir otros más complicados e interesantes. Todos los comportamientos que vamos a ver ahora tienen la misma estructura básica: Calculan el objetivo, o suposición u orientación, y entonces delegan el trabajo a otro de los comportamientos básicos para calcular el movimiento. El cálculo del objetivo puede venir dado por varios parámetros. El comportamiento de perseguir por ejemplo busca a un objetivo usando la información de movimiento de otro personaje. Incluso estos comportamientos delegados luego pueden ser usados para crear otros movimientos a su vez. Por ejemplo, el comportamiento de deambular crea objetivos aleatorios e intenta perseguirlos.

Perseguir y Evadir

Hasta ahora sólo hemos visto cómo movernos hacia una posición. Si estamos persiguiendo a un objetivo que se mueve, moverse directamente hacia esa posición no es suficiente. Para cuando lleguemos a donde se encuentra el objetivo, este ya se habrá movido. Si el objetivo está cerca de nosotros esto no es ningún problema, pero si se encuentra lejos, habremos gastado nuestros recursos para nada.

Para predecir hacia dónde se moverá nuestro objetivo podemos usar toda clase de algoritmos, pero seguramente sería una exageración. Simplemente vamos a considerar que el personaje va a seguir moviéndose hacía donde lo está haciendo ahora mismo, lo cual incluso en largas distancias no es ninguna tontería.

El algoritmo calcula la distancia entre el personaje y el objetivo y entonces calcula cuanto tiempo tardaría llegar hasta allí a velocidad máxima. Usa este intervalo de tiempo para predecir hacia dónde se movería él mismo. Calcula entonces la posición del objetivo si sigue moviéndose a su velocidad actual. Esta posición es entonces delegada al comportamiento de búsqueda que hemos visto antes. A continuación se expone el código:

```
let Pursue (character:Kinematic) (target:Kinematic) maxAcceleration
            maxPrediction explicitFunction =

    let mutable prediction = 0.0f
    let distance = (character.Velocity - target.Velocity).Length
    let speed = character.Velocity.Length

    if speed <= distance / maxPrediction then
        prediction <- maxPrediction
    else
        prediction <- distance / speed

    let explicitTarget = target.Clone
    let p = prediction

    explicitTarget.Position <- explicitTarget.Position +
                               (explicitTarget.Velocity * p)

    explicitFunction character explicitTarget maxAcceleration
```

Podemos ver que se le pasa una función con nombre “explicitFunction”. Esto es debido a que, como antes, el contrario a perseguir es evadir. Si queremos perseguir a un personaje pasaremos como argumento la función Seek y si queremos evadirlo usaremos la función Flee.

Mirando hacia donde nos estamos moviendo

Hasta ahora hemos asumido que la dirección hacia donde se mueve el personaje no tiene que ver con la dirección de su movimiento. En muchos casos, en cambio, queremos que nuestro personaje mire hacia donde se está moviendo.

Usando el comportamiento Align esto es realmente simple, simplemente tenemos que calcular la orientación usando la velocidad (que recordamos es también la dirección) hacia donde se mueve el personaje. A continuación se expone el código.

```
let LookWhereYoureGoing (character:Kinematic) =
    let steering = new SteeringOutput()

    if character.Velocity.Length = 0.0f then
        None
    else
        let explicitTarget = new Kinematic()

        explicitTarget.Orientation <- -atan2 -character.Velocity.[1]
                                              character.Velocity.[0]

        Align character explicitTarget
```

Hemos utilizado la función "atan2 y x"[39] que nos devuelve el ángulo entre el eje positivo x con respecto al eje de coordenadas. Debido a lo que comentamos anteriormente sobre el dibujado en las pantallas de ordenadores, tenemos que cambiar el signo tanto del eje y como del propio ángulo resultado de la operación.

Deambular

Este comportamiento es uno de los más importantes, ya que es usado siempre que una entidad no tenga ninguna acción que realizar. La idea es crear un objetivo imaginario lo suficientemente lejos del personaje para aplicar el algoritmo de persecución y que el personaje parezca que se está moviendo aleatoriamente.

```
let makeWander (character:Kinematic) (wanderOrientation : float32 array) =
    let wanderOffset = 50.0f
    let wanderRadius = 100.0f
    let wanderRate = 1.0f
    wanderOrientation.[0] <- (float32 (randomBinomial 1.0)) * wanderRate +
                                                    wanderOrientation.[0]
    let targetOrientation = character.Orientation + wanderOrientation.[0]

    let target = character.Clone

    target.Position <- character.Position + ((vec2.asVector character.Orientation)
                                            * wanderOffset)
    target.Position <- target.Position  + ((vec2.asVector targetOrientation) *
                                            wanderRadius)


    target

let Wander (character:Kinematic) (wanderOrientation : float32 array)
maxAcceleration =
    Pursue character (makeWander character wanderOrientation) maxAcceleration 1.0f

           Seek
```

Hemos divido este método en dos partes porque usaremos la función makeWander para tener más control sobre el objetivo que vamos a perseguir más adelante.

[39] https://en.wikipedia.org/wiki/Atan2

Una vez visto cómo se mueven los personajes podemos avanzar y describir cómo los personajes van a tomar decisiones.

Comportamiento orientado a objetivos

Si preguntamos a alguien de que se trata la inteligencia artificial, normalmente nos contestan que se trata de la habilidad de un ordenador para decidir cosas. Sin embargo, realizar acción siempre se da por hecho.

En la realidad, la toma de decisiones es típicamente una parte pequeña del esfuerzo necesitado para hacer una buena IA. La mayoría de los juegos usan sistemas de decisión muy simples: máquinas de estado y árboles de decisión. Los sistemas basados en reglas son más raros pero siguen siendo importantes.

Últimamente se está mostrando mucho interés en toma de decisiones más sofisticadas, como lógica difusa y redes neuronales. Sin embargo, los desarrolladores todavía no se han adaptado a estas tecnologías porque son difíciles de hacer funcionar bien. En esta aplicación vamos a intentar balancear algunas de estas opciones para conseguir el comportamiento deseado.

Objetivos

Vamos a usar un comportamiento orientado por objetivos. La idea es que cada personaje tenga uno o más objetivos, también llamados motivos. Estos motivos pueden ser desde comer, regenerar salud o matar a un enemigo. Cada uno de estos objetivos tienen un nivel de importancia (o insistencia) representada por un número. Un objetivo con una gran insistencia tenderá a influir en el comportamiento del personaje más que otros.

Los personajes intentarán satisfacer sus objetivos por completo o al menos reducir su insistencia. Un objetivo con insistencia 0 será un objetivo completamente satisfecho

Tenemos 5 objetivos, dos de ellos tienen una identidad asociada, que será la identidad del objetivo del que queremos huir o atacar. Tenemos que saber cuál es porque basándonos en su distancia respecto al personaje aumentaremos o disminuiremos la insistencia de estos objetivos. Luego existen 2 objetivos básicos, que son el hambre y las ganas de procrear. Por último tenemos un objetivo que representa a ningún objetivo, ya que a veces no necesitaremos hacer ninguna acción.

Aparte podemos ver dos métodos. El método get devuelve la insistencia del objetivo. El método getDiscontentment nos da un valor aproximado de cómo de malo o bueno es el objetivo. Por ejemplo, si tenemos como objetivo escapar, tendremos que darle prioridad sobre otros, así que elevamos ese valor al cubo.

```
type Goal =
    | EatFood of float
    | KillG of float * int
    | Escape of float * int
    | Procreation of float
    | NoGoal

    member this.get() =
        match this with
        | EatFood it | Procreation it -> it
        | Escape (it, _) | KillG (it, _) -> it
        | _ -> 1000.0

    member this.getDiscontentment(value) =
        match this with
        | Procreation _ -> abs value
        | EatFood _  | KillG _ -> value * value
        | Escape (_, _) -> abs value * value * value
        | _ -> 1000.0
```

Acciones

Además de tener una lista de objetivos, necesitamos una de acciones que poder realizar. Estas acciones pueden ser generadas en localmente, pero también es común que sean generadas por los objetos del mundo. Un enemigo ofrecerá la acción "atácame" y una puerta ofrecerá la opción de "ciérrame".

Conforme las acciones son añadidas a la lista de opciones a elegir, también se evalúan con respecto a los objetivos que el personaje tiene. Esta evaluación muestra cuánto impacto tiene esta acción para un objetivo concreto. Por ejemplo la acción "jugar con la videoconsola" incrementará la felicidad mucho, pero la energía se verá disminuida.

```
type ActionType =
    | Eat of float * int
    | FleeA of float * int
    | Procreate of float * int
    | WantProcreate of float * int
    | Attack of float * int
    | NoAction

    member this.getGoalChange(goal) =
        match goal with
        | EatFood _ -> match this with
                       | Eat (it, _) -> it
                       | _           -> 0.0
        | KillG (it, id) -> match this with
                            | Attack (it, idd) -> if id = idd then it else 0.0
                            | _ -> 0.0

        | Escape(it, id) -> match this with
                            | FleeA (it, idd) -> if id = idd then it else 0.0
                            | _ -> 0.0

        | _ -> 0.0
```

Eligiendo la acción

Una vez introducidos los objetivos y las acciones podemos pasar a elegir la acción que mejor se adapte a nuestras condiciones. Ya hemos visto que dependiendo de la utilidad de la acción para con el objetivo tendrá un valor u otro. El objetivo de este algoritmo, y por tanto del personaje, será reducir la cantidad de descontento. Esto hace que no se enfoque en un sólo objetivo, sino en todo el conjunto. Mostramos un ejemplo:

Objetivo: Comer = 4 Objetivo: Ir al baño = 3

Acción: Beber refresco (Comer – 2; Ir al baño + 2)

- Después: Comer = 2, Ir al baño = 5: Descontento = 29

Acción: Visitar el baño (Ir al baño – 4)

- Después: Comer = 4, Ir al baño = 0: Descontento = 16

Como podemos ver, se ha elegido correctamente la opción más sensata. Para hacer una decisión, cada posible acción es considerada por turnos. Se hace una predicción el descontento total después de que la acción sea completada. La acción que lleve al menor descontento es la elegida. El algoritmo es realmente sencillo:

```
let calculateDiscontentment (action : ActionType) (goals : Goal array) =
    let discontentment = ref 0.0

    goals |> Array.iter(fun goal -> discontentment.contents <-
                                discontentment.contents +
                                (goal.getDiscontentment((goal.get()) +
                                action.getGoalChange(goal))))

    discontentment.contents

let chooseAction (actions: ActionType array) (goals: Goal array) =
    let bestAction = ref actions.[0]
    let bestValue = ref(calculateDiscontentment actions.[0] goals)
    let thisValue = ref 0.0

       actions.[1..] |> Array.iter (fun action -> thisValue.contents <-
                               (calculateDiscontentment action goals);
                               if thisValue.contents < bestValue.contents then
                                  bestValue.contents <- thisValue.contents;
                                  bestAction.contents <- action)

    bestAction
```

Sistema de eventos

Como se ha explicado, un personaje tendrá una lista de objetivos que cumplir y las demás criaturas y entidades del entorno les ofrecerán acciones que él pueda seguir para cumplirlos. Sin embargo todavía nos queda una cuestión. ¿Cómo sabemos cuándo un personaje nos está atacando? ¿Cómo ponemos de acuerdo a dos criaturas para que se apareen? La respuesta está en el sistema de eventos que he ideado.

Podemos imaginarnos este sistema de eventos como un buzón de correo. En cada turno la criatura mira este buzón a ver si le ha llegado algún mensaje. Pueden ser desde un bonito "quiero casarme contigo" hasta un horroroso "voy a matarte". Estos eventos ayudarán a decidir algunos objetivos y a ofrecer acciones.

```
type Event =
    | Collision of List<Boundary> // Colisión con id
    | WantProcreateE of int // Enviamos solicitud para procrear
    | UnderAttack of int // Nos está atacando la id
    | HealthChange of int // La vida que ganamos o perdemos porque nos ataquen
    | FoodEated of int // Si somos comida, nos han comido tanta comida
```

Controlador de la Inteligencia Artificial

Ahora que ya sabemos cómo deciden, cómo se mueven las criaturas por el mundo y cómo se envían mensajes entre ellas, debemos juntar todo y colocarlo de tal manera que podamos observar el trabajo hecho. Sobre esta parte apenas hay nada escrito, ya que depende de los recursos disponibles. Cuando se explicaron las entidades se habló de un manejador que también podíamos ver como interfaz en el diagrama. Ahora vamos a implementar ese manejador de inteligencia artificial. Crearemos uno para las comidas y otro para las criaturas.

Controlador de la comida

Este controlador es muy simple. Simplemente miramos en la lista de eventos si hemos sido comidos. Si lo hemos sido, rebajamos la cantidad disponible de comida en la cantidad indicada por el evento. La comida siempre ofrece como acción ser comida.

Controlador de las criaturas

Este controlador de divide en varias partes debido a su complejidad.

1. *Ofrecer acciones*

   Primero debemos limpiar la lista de ofertas que hacemos, seguido de ofrecernos siempre para ser atacados. Además, si la criatura tiene suficiente ganas de procrear ofreceremos la acción "Quiero procrear".

2. *Detección y limpieza*

Acto seguido limpiamos todas las listas que manejamos, la lista de objetivos, acciones, colisiones, bajo ataque, etc… Una vez hecho esto, guardamos en dos listas las entidades que están al alcance de nuestro sentido de detección y en otra lista las entidades que están demasiado cerca de nosotros.

3. *Repasar los eventos*

Recorremos la lista de eventos tomando acciones para cada uno. Por ejemplo si nos ha llegado un evento de colisión, añadimos la identidad a la lista, si estamos bajo ataque de alguna entidad, la añadimos a nuestra lista negra, si se nos informa de que nos han golpeado, bajaremos nuestra vida en cierta cantidad, etc., etc…

4. *Acción anterior*

Evaluamos la acción anterior. Primero intentamos conseguir el objetivo de la acción anterior, porque puede que este haya muerto o salido de nuestro rango. Si tenemos colisión con esta entidad evaluamos el tipo de acción que estábamos llevando a cabo. Por ejemplo, si estábamos atacando y estamos chocando contra el enemigo, entonces podemos dar el ataque por válido y enviamos a ese enemigo un evento informándole de que debe de bajar su vida en una cantidad. Por otro lado, si queríamos procrear y esta entidad nos mandó en el paso 3 el evento de querer procrear, damos por satisfecho el trato y mandamos, como vimos en el capítulo 5, la petición de procrear al manejador del mundo.

5. *Decidir… o no*

Si estábamos atacando, escapando o buscando una pareja, lo más lógico es que sigamos realizando esta acción durante varios turnos, ya que sino no podrá llevarse a cabo. Si decidimos pasamos la paso 6, sino al paso 7.

6. *Vamos a decidir*

Una vez que decidimos que vamos a decidir (valga la redundancia) añadimos a la lista de pre-acciones todas aquellas que nos ofrezcan las criaturas de nuestro entorno. Además, dependiendo de nuestra agresividad vamos a añadir a la lista de objetivos a las criaturas que estén cerca y si estamos bajo ataque decidiremos entre huir o atacarlas. La diferencia con el anterior es que con las criaturas que nos atacan sí vamos a usar acciones directamente, y con las otras las pondremos como posibles objetivos. Si no estamos bajo ataque pondremos como objetivos el querer comer y querer procrear.

7. *Refinado de acciones*

Una vez que tengamos todas las acciones, lo que vamos a hacer es refinarlas. Lo que hacemos es cambiar los valores de la intensidad que satisfacen dependiendo de lo cerca o lejos que estén. Así pues, una acción de atacar que este al borde del sentido de la criatura tendrá mucho menos influencia que la que pueda tener comer una planta al lado de ella.

8. *Decisión*

Ya hemos visto cómo deciden las criaturas, así que simplemente llamamos a la función con las acciones y objetivos que hemos recopilado y ella nos devolverá la mejor acción para ser ejecutada. Una vez obtenida la acción a llevar a cabo, comprobamos que realmente el objetivo sigue existiendo, y de ser así llevamos a cabo la acción de movimiento. Por ejemplo, si vamos a atacar llamaremos al algoritmo de Pursue que vimos antes para perseguir al objetivo. Si no tenemos objetivo simplemente llamaremos a Wander y la criatura se moverá en una dirección aleatoria. Además, se limpia la lista de eventos en este momento por lo que veremos en un segundo.

9. *Actualización*

Una vez conseguido el SteeringOutput que nos devuelve los algoritmos de movimiento, realizamos una petición de actualización de nuestra cinemática. Primero vamos a probar que realmente podemos movernos a la posición que nos indica el SteeringOutput. Para ello clonamos nuestro Kinematics y aplicamos el SteeringOutput recibido.

Una vez actualizado, necesitamos conocer cuáles son las criaturas cercanas. Para ello enviamos una petición con el rectángulo de cercanía al Quadtree. Estos nos darán una lista de rectángulos que están demasiado cerca de nosotros. Ahora creamos unos bigotes auxiliares con la posición recopilada por el Kinematics clonado y actualizado y comprobamos que no tenemos colisiones y nos encontramos dentro del mundo.

En caso de que no tengamos ninguna colisión podemos realizar la acción, así que actualizamos el Kinematics verdadero de la criatura y actualizamos su Shape y su rectángulo.

En el caso de que sí tengamos colisiones, enviamos como evento de colisión la lista de colisiones que hemos calculado a la propia criatura, para poder tomar las decisiones que hemos visto anteriormente.

También se lleva una cuenta desde la última vez que ha sido posible moverse. Esto lo tenemos que hacer porque las criaturas a veces se quedan “atascadas” y es necesario dar un pequeño giro para liberarse, independientemente de su acción.

10. Final

Por último, siempre, decidamos lo que decidamos, miraremos hacia donde nos indique el SteeringOutput. Esto lo hacemos llamando al método LookWhereAreYouGoing con la cinemática actualizada de la criatura y actualizamos de nuevo la Kinematic de la criatura.

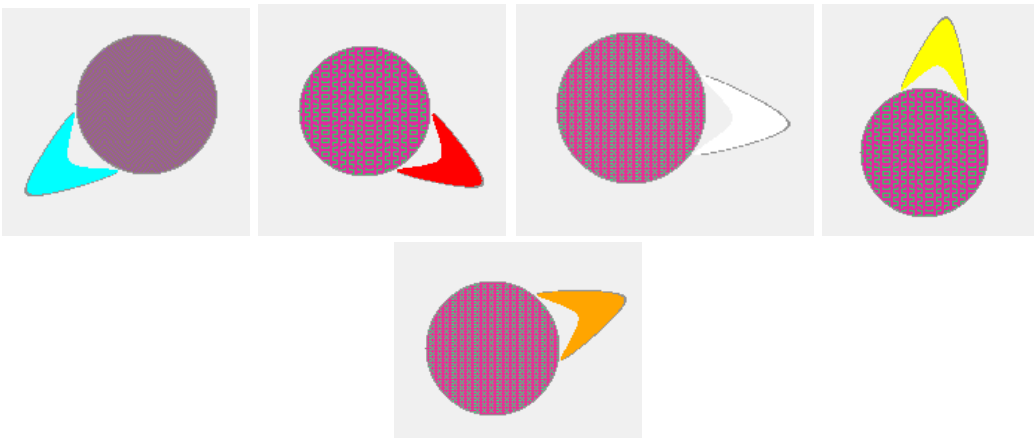


Figura 28. Criaturas expresando sus acciones. De izquierda a derecha: Querer procrear, atacando, buscando, huyendo y comiendo.

Por último, cabe destacar que la criatura expresan su acción cambiando el color de su cabeza, como se puede ver en la figura anterior.

Capítulo 7: Guardando y cargando el mundo

Antes de pasar a las interfaces de usuario que componen el programa vamos a explicar la última funcionalidad del mismo. Es un punto interesante poder guardar el mundo para poder seguir viendo el comportamiento más tarde o cargarlo para retomar la investigación. Para lograr este objetivo se ha diseñado e implementado una base de datos usando el motor MySQL[40].

Creación de la base de datos

Para facilitar las cosas, vamos a usar WAMP[41], un gestor que normalmente se usa para páginas web pero también nos facilita las cosas en caso de que queramos phpMyAdmin[42]. Una vez instalado WAMP podemos acceder a la consola de MySQL como se muestra en la siguiente imagen. La contraseña por defecto es simplemente presionar Enter.

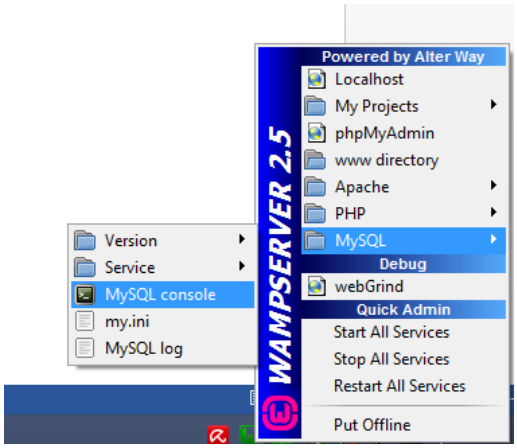


Figura 29. Accediendo a la consola de MySQL

También podemos acceder a muchas utilidades que nos ofrece MySQL accediendo desde la consola de comandos a la siguiente dirección "C:\wamp\bin\mysql\mysql5.6.17\bin".

Ahora debemos crear la base de datos, con sus tablas y atributos. Para ello he creado un script que se adjunta como anexo. Luego, accediendo a la dirección mencionada, ejecutamos la siguiente línea de comando para instalar la base de datos.

[40] https://www.mysql.com/
[41] http://www.wampserver.com/en/
[42] http://www.phpmyadmin.net/home_page/index.php

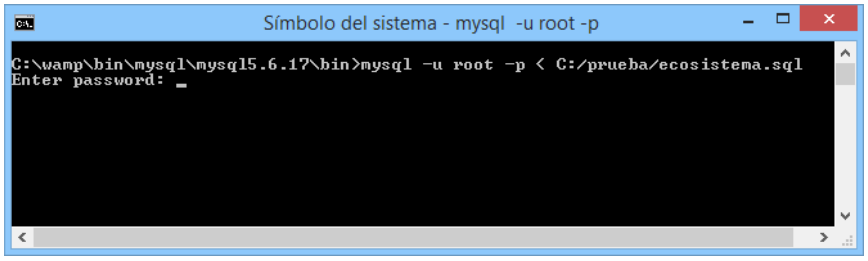


Figura 30. Creando la base de datos.

Ahora, con razón de acceder desde .NET a la recién creada base de datos, tenemos que descargar el conector de .NET[43] y enlazarlo con nuestro proyecto. Para enlazarlo tenemos que hacer botón derecho sobre el proyecto en el Explorador de soluciones y movernos al menú de Agregar y haciendo clic en Referencia.

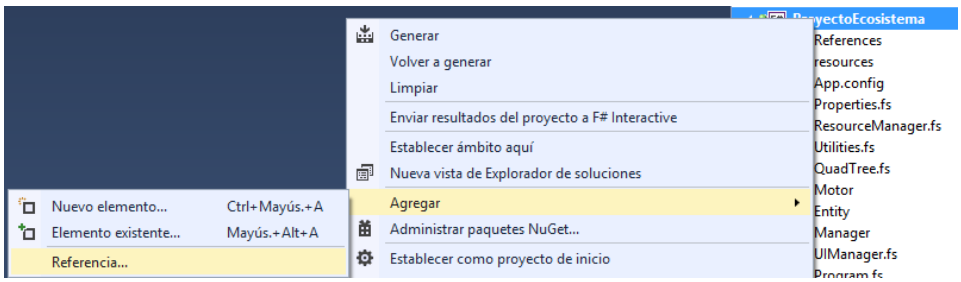


Figura 31. Localización de la opción para agregar referencias.

Ahora nos vamos a la pestaña de examinar y buscamos las .dll que hemos descargado antes, teniendo cuidado de que elegimos las correctas para la versión de .NET que usamos. Estas .dll son:

- MySql.Data.dll
- MySql.Data.Entity.dll
- MySql.Data.Entity.EF6.dll
- MySql.Fabric.Plugin.dll
- MySql.Web.dll

También debemos agregar referencias para System.Configuration, System.Data, System.Transactions y System.Xml, que se pueden encontrar en la pestaña de Ensamblados.

Por último debemos establecer la conexión con la base de datos desde F#, que se hace de la siguiente manera:

```
let connectionSTR = "Data Source=localhost; Port=3306;
                     Database=proyecto_ecosistema; User ID=root; Password=;"
let connection = new MySqlConnection(connectionSTR)
```

[43] http://cdn.mysql.com/Downloads/Connector-Net/mysql-connector-net-6.9.5-noinstall.zip

Antes de empezar a tratar con los datos, debemos tener en cuenta que los booleanos y el sexo de las criaturas lo vamos a guardar como como 0 y 1 en la base de datos, y que las variables del tipo float32 los convertimos a float para que no haya ningún problema.

Primero vamos a definir unos métodos para insertar, seleccionar, actualizar o eliminar datos de la base de datos e intentaremos construir el resto de métodos necesarios a partir de estos, por lo que tratar con los datos será realmente sencillo.

```
let insert query (data : List<string * obj>) =
    try
        try
            connection.Open()
            let cmd = new MySqlCommand()
            cmd.Connection <- connection
            cmd.CommandText <- query
            cmd.Prepare()
            data.ForEach(fun item ->
             cmd.Parameters.AddWithValue(fst item, snd item) |>
                                        ignore)
            cmd.ExecuteNonQuery() |> ignore

        finally
            connection.Close()
            data.Clear()
    with
    | _ as ex -> printfn "%A" ex
```

```
let select query =
    let mutable rdr = None
    let mutable res = None

    try
        try
            connection.Open()
            let cmd = new MySqlCommand(query, connection)
            rdr <- Some (cmd.ExecuteReader())
            let fieldCount = (get rdr).FieldCount;
            if fieldCount > 0 then
                let mutable array = Array.create fieldCount (new
                                        obj())

                if (get rdr).Read() then
                    let a = (get rdr).GetValues(array)
                    res <- Some array

        finally
            if isSome rdr then
                (get rdr).Close()

            connection.Close()
    with
    | _ as ex -> printfn "%A" ex

    res
```

Este método ya cuida por nosotros que nuestra consulta sea válida, devolviéndonos None si no lo era.

Para actualizar tablas usaremos el siguiente método:

```
let update table set where =
    try
        try
            let query = "UPDATE " + table + " SET " + set + "
                         WHERE  " + where
            connection.Open()
            let cmd = new MySqlCommand()
            cmd.Connection <- connection
            cmd.CommandText <- query
            cmd.ExecuteNonQuery() |> ignore

        finally
            connection.Close()
    with
    | _ as ex -> printfn "%A" ex
```

Finalmente, para borrar de la base de datos usaremos el siguiente método:

```
let delete table where =
    try
        try
            let query = "DELETE FROM " + table + " WHERE  " +
where
            connection.Open()
            let cmd = new MySqlCommand()
            cmd.Connection <- connection
            cmd.CommandText <- query
            cmd.ExecuteNonQuery() |> ignore

        finally
            connection.Close()
    with
    | _ as ex -> printfn "%A" ex
```

Guardando el mundo

Ahora ya estamos en disposición de poder guardar los datos. Tenemos que tener en cuenta que existen dos tipos de tablas, las independientes y las dependientes. Las tablas independientes generan su propia identidad automáticamente a la hora de insertar un dato, por lo que tenemos garantizada su unicidad. Las tablas dependientes, por su parte, son aquellas que usan las identidades de las tablas independientes para demostrar su unicidad en el sistema. Veamos un ejemplo para entenderlo mejor.

Queremos guardar la Kinematic de un personaje, que recordemos que se compone de sólo 3 parámetros: 2 vectores indicando su posición y velocidad y un float indicando su orientación en radianes. Para los vectores hemos creado otra tabla que guarda su identidad y su posición (x,y). Por lo tanto necesitamos la identidad de esos vectores para poder enlazar el Kinematic.

Sabemos que cuando se inserta un dato en una tabla independiente, su valor identidad se autoincrementa y por tanto pasa a ser el número mayor de la tabla. La idea es simplemente coger ese valor y devolverlo en cuanto se inserta el dato en una tabla independiente. Para ello se ha definido esta función:

```
let getMaxID table =
    let mutable select_res = None
    select_res <- select ("SELECT max(id) FROM " + table)

    let mutable res = -1

    if isSome(select_res) then
        if (get select_res).Length <> 0 then
            res <- Convert.ToInt32((get select_res).[0])

    res
```

Entonces lo que hacemos es guardar un vector, llamar a esta función, y devolvérsela a la función que guarda el Kinematics, de tal manera que ya tenemos las tablas enlazadas. Veamos cómo se guarda un vector:

```
let saveVector x y (data : List<string * obj>) =
    let query = "INSERT INTO vector
                 (world_id, x, y)
                 VALUES
                 (@world_id, @x, @y)"

    data.Add("@world_id", worldID :> obj)
    data.Add("@x", x :> obj)
    data.Add("@y", y :> obj)

    insert query data

    getMaxID "vector"
```

Como vemos, gracias a los métodos que se han implementado de ante mano, ahora es realmente sencillo usar la base de datos. Simplemente tenemos que preocuparnos de que la consulta que realizamos sea correcta y que los datos que enviamos estén correctamente enlazados también.

Antes de pasar al siguiente apartado, veamos cómo terminaríamos de guardar una cinemática. Podemos ver que la cinemática también devuelve su mayor identidad, ya que enlazaremos esta cinemática con una entidad, y así seguiremos subiendo escalones para guardar la base de datos por completo.

```
let saveKinematic (kinematic : Kinematic) (data : List<string * obj>) =

    // Primero necesitamos 2 vectores para la posición y la velocidad
    let position_id = saveVector (float kinematic.Position.vector.[0]) (float
                                  kinematic.Position.vector.[1]) data
    let velocity_id = saveVector (float kinematic.Velocity.vector.[0]) (float
                                  kinematic.Velocity.vector.[1]) data

    let query = "INSERT INTO kinematic
                 (world_id, position_id, velocity_id, orientation)
                 VALUES
                 (@world_id, @position_id, @velocity_id, @orientation)"

    data.Add("@world_id", worldID :> obj)
    data.Add("@position_id", position_id :> obj)
    data.Add("@velocity_id", velocity_id :> obj)
    data.Add("@orientation", (float kinematic.Orientation) :> obj)

    insert query data

    getMaxID "kinematic"
```

Cargando el mundo

Ahora que ya hemos guardado el mundo, también debemos ofrecer la oportunidad de cargarlo. Para ello vamos a usar la misma técnica que hemos descrito anteriormente, los elementos de niveles superiores pedirán la carga de elementos de niveles inferiores para construirse. Vamos a utilizar los Dataset[44] que recogen todos los datos de una consulta y los ordena en tablas (arrays) con lo cual acceder a los datos es muy sencillo.

```
let select_dataset table =
    let mutable res = None

    try
        try
            connection.Open()

            let ds = new DataSet()
            let da = new MySqlDataAdapter("SELECT * FROM " + table,
                                          connection)
            da.Fill(ds, table) |> ignore

            res <- Some ds.Tables.[table]

        finally
            connection.Close()
    with
    | _ as ex -> printfn "%A" ex

    res
```

[44] https://msdn.microsoft.com/en-us/library/system.data.dataset(v=vs.110).aspx

Con éste método, sólo tenemos que tener cuidado de que nuestra consulta es correcta y además de que al leer los datos lo hacemos de manera correcta, incluyendo la conversión de tipos, ya que los datos que recibimos lo hacen como objetos. Veamos la carga de una cinemática usando este método:

```
// Selecciona un vector
let select_vector id =
    let mutable res = None
    let ds = select_dataset ("vector WHERE id="+ id.ToString() + " AND
world_id=" + worldID.ToString())

    if isSome ds then
        let data = get ds
        let x = float32 (Convert.ToDouble(data.Rows.[0].ItemArray.GetValue(2)))
        let y = float32 (Convert.ToDouble(data.Rows.[0].ItemArray.GetValue(3)))

        res <- Some (x,y)

    res

// Selecciona una kinematic
let select_kinematic id =
    let mutable res = None
    let ds = select_dataset ("kinematic WHERE id="+ id.ToString() + " AND
world_id=" + worldID.ToString())

    if isSome ds then
        let data = get ds
        let position    = select_vector
                  (Convert.ToInt32(data.Rows.[0].ItemArray.GetValue(2)))
        let velocity    = select_vector
                  (Convert.ToInt32(data.Rows.[0].ItemArray.GetValue(3)))
        let orientation = float32
                  (Convert.ToDouble(data.Rows.[0].ItemArray.GetValue(4)))

        if isSome position && isSome velocity then
            let kinematic = new Kinematic()
            kinematic.Position <- {vector=[|float32 (fst (get position));
                                            float32 (snd (get position))|]}
            kinematic.Velocity <- {vector=[|float32 (fst (get velocity));
                                            float32 (snd (get velocity))|]}
            kinematic.Orientation <- orientation

            res <- Some kinematic

    res
```

Siguiendo esta técnica podemos cargar todos los datos del sistema.

A continuación podemos ver un esquema de la base de datos:

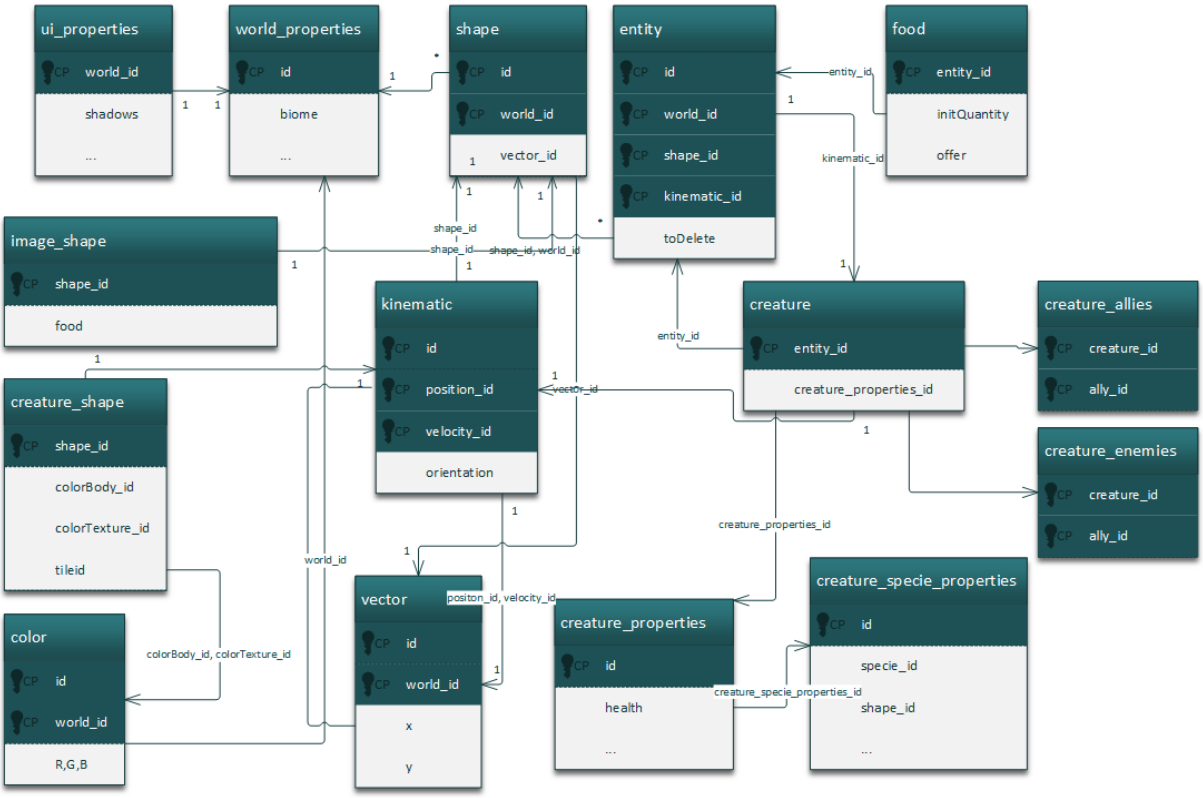


Figura 32. Esquema de la base de datos.

Capítulo 8: Interfaz del programa

Una vez visto todo lo necesario para que funcione el programa, vamos a exponer cómo el usuario interactúa con él. La interfaz se compone de 4 ventanas principales y 2 auxiliares. A continuación podemos ver un diagrama de cómo interactúan entre sí las interfaces.

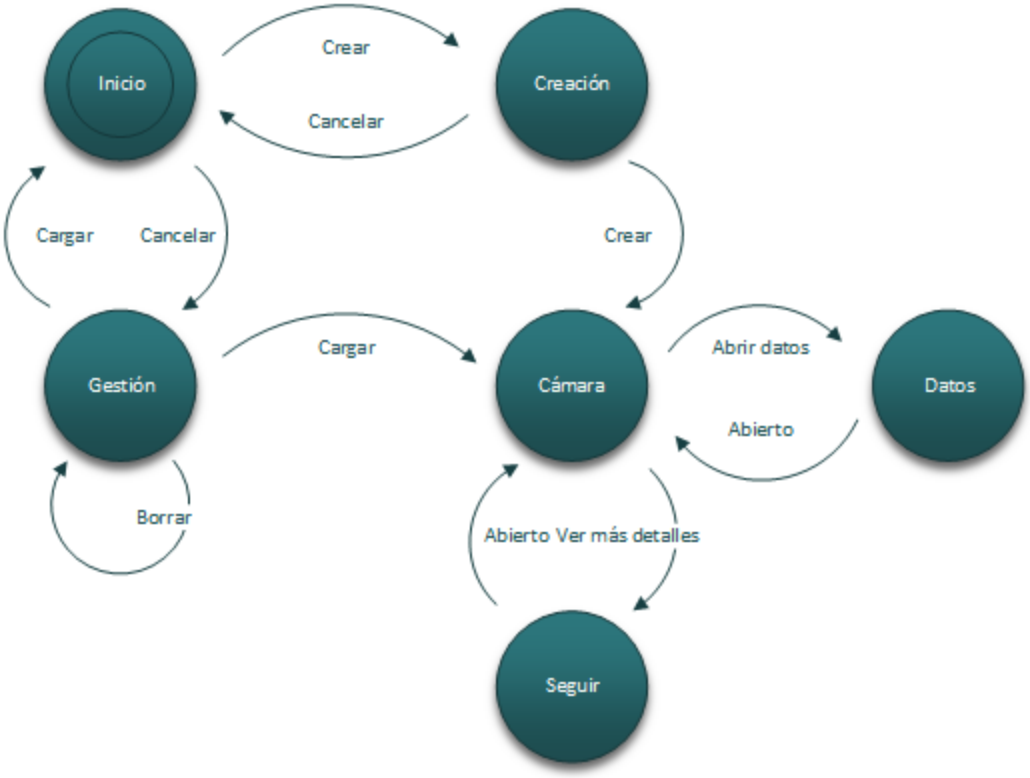


Figura 33. Interacción entre ventanas.

Para realizar las ventanas se ha heredado la clase Form[45], que representa una ventana en donde se pueden poner distintos elementos, como botones y campo de texto, para la interfaz de usuario. Se ha creado una clase básica que inicializa el buffer. Esto debemos hacerlo, sobre todo a la hora de pintar el mundo, para que no parpadee la ventana[46]. A continuación se exponen las ventanas individualmente.

Inicio

Esta ventana es simple. Sólo consta de 3 botones que nos permitirán elegir que acción queremos realizar: Crear un mundo nuevo, gestionar un mundo existente o cerrar la aplicación.

[45] https://msdn.microsoft.com/en-us/library/System.Windows.Forms.Form(v=vs.110).aspx
[46] https://msdn.microsoft.com/en-us/library/3t7htc9c(v=vs.110).aspx

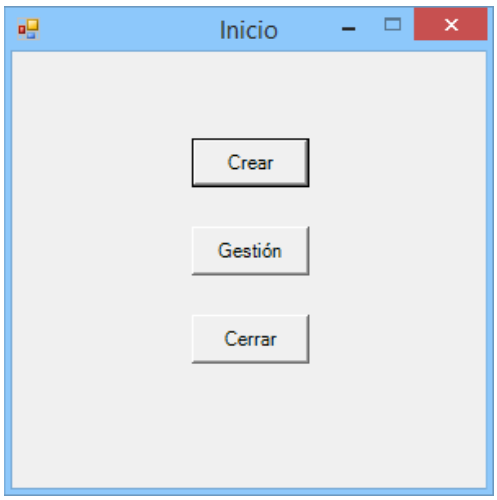


Figura 34. Ventana de inicio.

Creación

En esta ventana podemos crear un mundo nuevo. Nos permite especificar el nombre del mundo, el número máximo de criaturas, el tamaño y el tipo de bioma. Una vez que pulsamos en crear se creará un mundo que delegaremos en la ventana de la cámara del mundo que veremos más adelante.

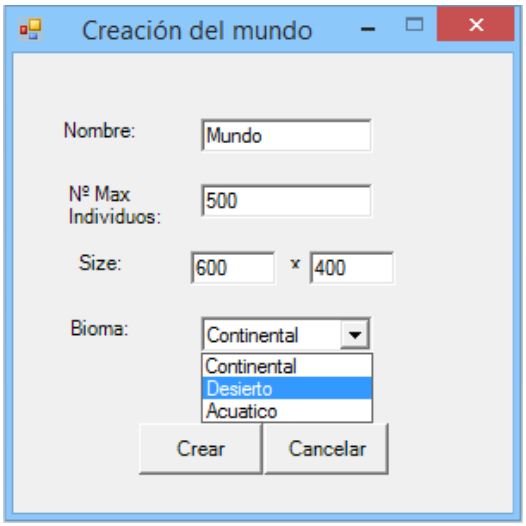


Figura 35. Ventana de creación del mundo.

Gestión de mundos

Desde esta ventana podemos cargar o borrar un mundo existente en la base de datos. Para mostrar los mundos disponibles sólo tenemos que usar la propiedad DataSource[47] de la herramienta Datagrid[48] que nos ofrece .NET. Esta propiedad recibe como parámetros un Dataset. Por este motivo, cuando implementamos los métodos para interactuar con la base de datos, se eligió seleccionar los datos a través de Dataset.

Con el botón de borrar podemos borrar el mundo seleccionado. Esto hará que todos sus datos se borren de la base de datos y no se pueda recuperar.

Con el botón de cargar recorreremos la base de datos con los métodos implementados y recrearemos el mundo con estos datos. Una vez cargado, al igual que al crear, delegaremos este mundo en la ventana de Cámara.

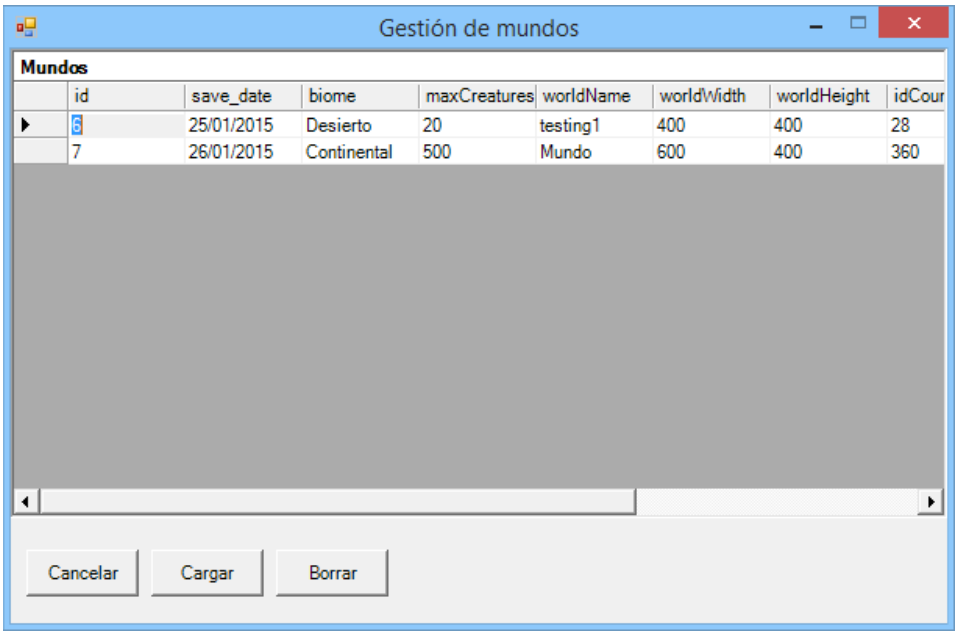


Figura 36. Ventana de gestión de mundos.

Cámara

Esta es la ventana más interesante. Desde esta ventana podemos observar lo que pasa en el mundo. No es el mundo en sí, sino una cámara que se puede mover para ver distintas partes del mundo, por lo que se ha implementado métodos de gestión de posición cuando la cámara se mueve.

[47] https://msdn.microsoft.com/en-us/library/system.windows.forms.datagrid.datasource(v=vs.110).aspx
[48] https://msdn.microsoft.com/en-us/library/system.windows.controls.datagrid(v=vs.110).aspx

Este movimiento es realmente sencillo. Calculamos la diferencia entre el clic inicial y el movimiento que ha hecho el usuario con el ratón y creamos un rectángulo de prueba con estos datos. El rectángulo abarcará desde la posición actual de la cámara más esta diferencia, hasta completar el ancho y alto de la ventana.

Luego probamos si el mundo contiene a esta cámara, y de ser así, simplemente actualizamos los valores desde donde se pinta el mundo con la diferencia con la posición actual de pintado menos la diferencia que calculamos al principio. Es importante que sea una resta porque queremos mover los puntos de dibujado en dirección contraria des donde nos estamos moviendo. El resultado es un movimiento de cámara fluido y exacto.

Por otro lado, también se han implementado distintas opciones que realizar a las que se puede acceder desde el botón derecho del ratón. Si pulsamos sobre una criatura o comida nos mostrará opciones referentes a ellas.

La comida nos dará la opción de eliminarla y de ver qué cantidad de comida le queda disponible. Por su parte, las criaturas nos mostrarán su centro, identidad, poder eliminarlas y ver más detalles. Esta última opción abrirá una ventana que veremos más adelante.

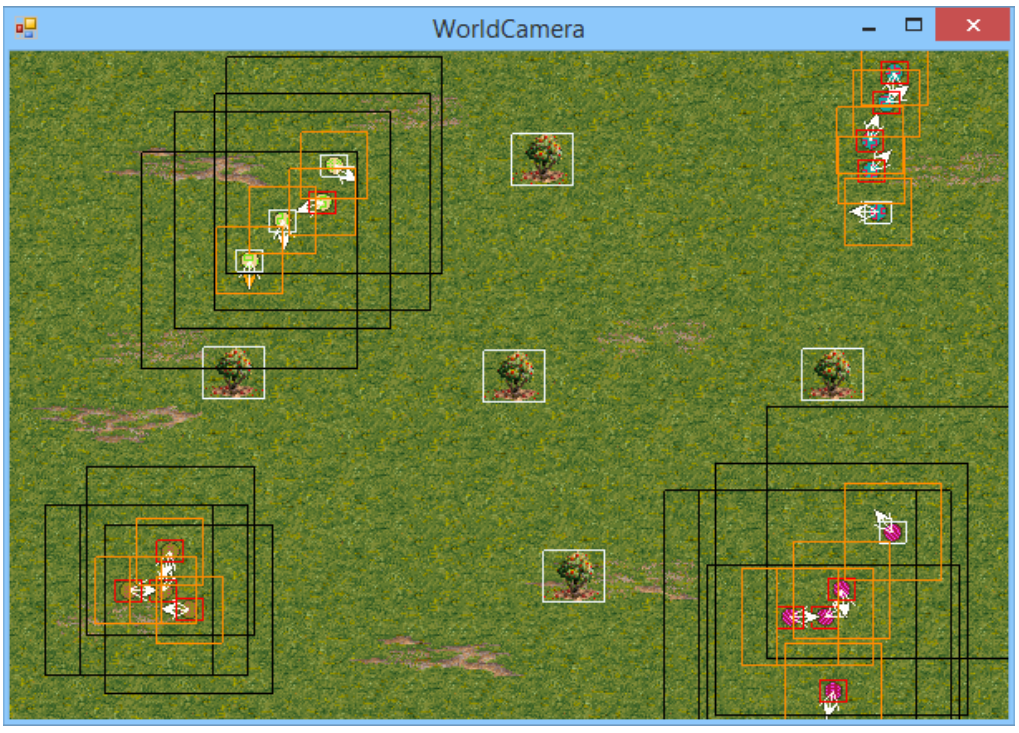


Figura 37. Ventana de cámara. Las opciones de mostrar los rectángulos y las sombras están marcados.

Si hacemos clic con el botón secundario del ratón allí donde no haya ni criatura ni comida, nos abrirá un menú donde podemos elegir las opciones del mundo. Podremos activar las sombras, ver los distintos rectángulos que usan las criaturas para moverse por el mundo, ver el mundo a través del Quadtree, activar o desactivar el multiproceso

y podremos ver una ventana de datos. Este última opción también ser verá más adelante.

En la figura anterior podemos ver los rectángulos y bigotes de las criaturas. El rectángulo blanco representa el cuerpo de la criatura, el naranja representa el rectángulo de cercanía y el negro el sensorial. Además, cuando un bigote hace colisión con el rectángulo blanco, este se marca en rojo para indicar que está habiendo una colisión con otra entidad.

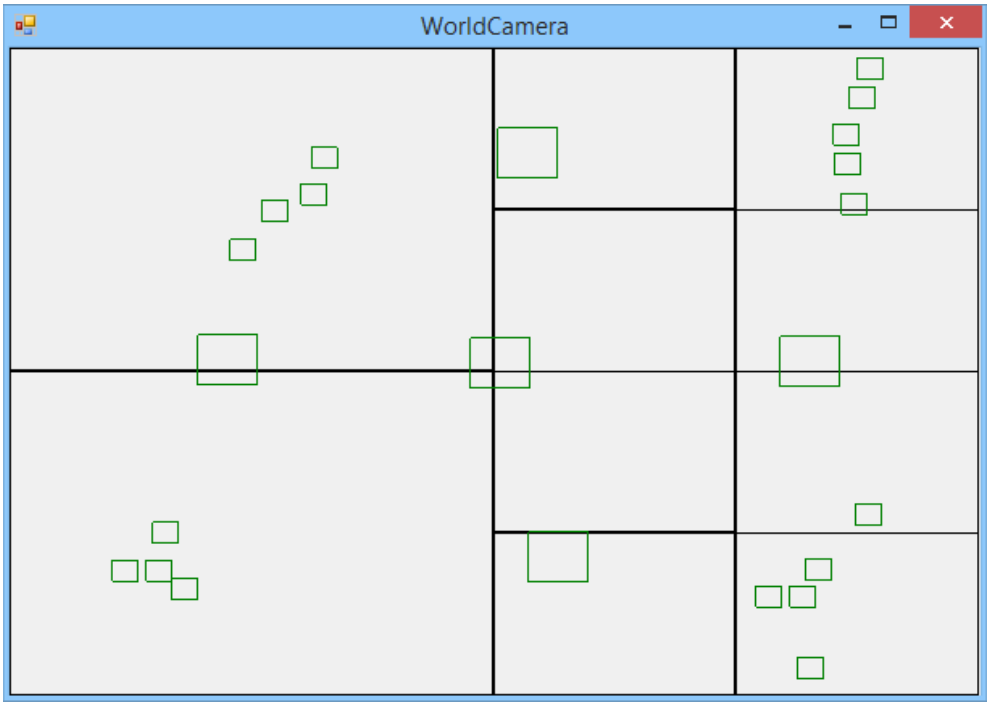


Figura 38. Mismo mundo que la figura anterior pero visto desde el Quadtree.

Para poder crear entidades sólo tenemos que hacer clic con el botón central del ratón allí donde queramos colocarla. En la siguiente ventana se explica cómo se puede cambiar las entidades a poner.

Datos

En esta ventana se muestran datos y otras opciones del mundo. Esta ventana nos informa del número de criaturas y objetos que existen, además del bioma, nombre y tamaño del mundo.

Nos ofrece a través de botones poder pausar el mundo y guardarlo. Es importante notar que para poder guardar el mundo primero tenemos que pausarlo, sino la opción de guardar no se habilitará.

Por otro lado tenemos 3 botones que nos dan la posibilidad de controlar qué entidades vamos a colocar en el mundo como se explicó antes. Estos botones actúan como pulsadores, es decir, si estamos poniendo criaturas, ese botón estará deshabilitado.

Si estamos en este modo podemos cambiar la especie de la criatura que vamos a poner. Este proceso es totalmente aleatorio. También podemos elegir poner comida para las criaturas.

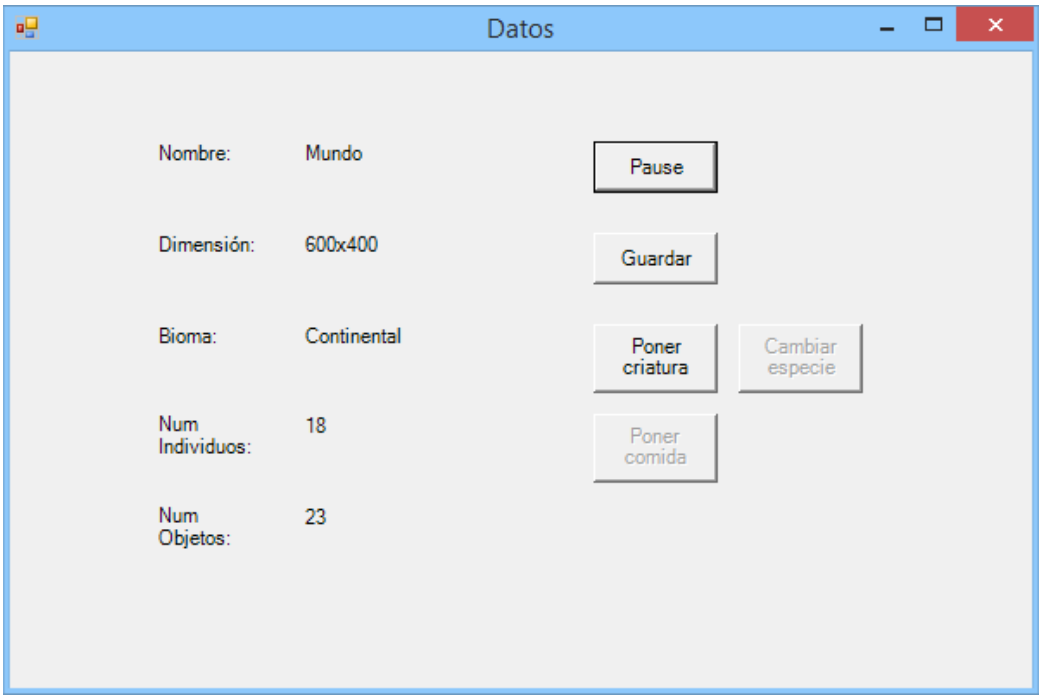


Figura 39.Ventana de datos. La opción de poner comida está seleccionada.

Seguimiento

Por último, podemos acceder a la información de la criatura y ver cómo sus parámetros cambian. Además se ofrece un botón para mostrar más datos, que se imprimirán por la ventana de comandos, y que muestra todos los datos de la criatura, como su especie, etc...

Para llevar esta lista de criaturas en seguimiento, la ventana de cámara lleva una lista con estas ventanas, ya que es necesario establecer una conexión entre estas ventanas y el mundo principal, como por ejemplo a la hora de cerrar el programa.

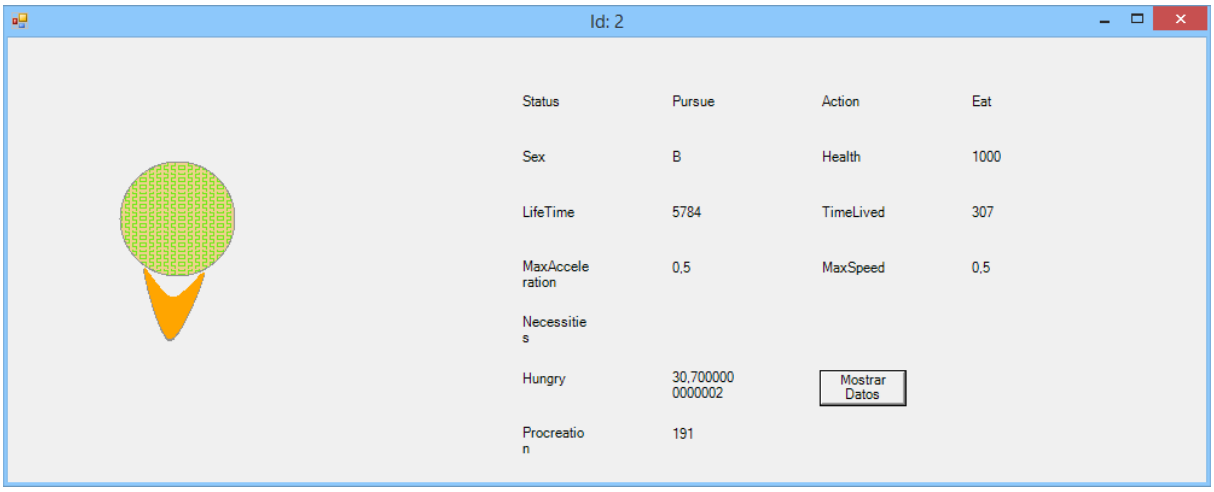


Figura 40. Ventana de seguimiento de una criatura.

Conclusiones

Este trabajo abarca muchos conceptos que se trabajan en la ingeniería y se demuestra su puesta en acción en esta memoria. Se ha aprendido el lenguaje de programación F#, el cual mezcla conceptos de la programación funcional con orientación a objetos y permite el uso de la plataforma .NET.

También se han investigado los sistemas y algoritmos de detección de colisiones más actuales y se han implementado dos de ellos. Se ha llevado a cabo el modelado de una base de datos y el modelado del programa, los cuales se han implementado satisfactoriamente.

Se ha llevado a cabo un extenso estudio en diversidad técnicas de inteligencia artificial, muchas de las cuales no se han mencionado pero han influido en la decisión y creación de los algoritmos presentados.

Un problema que nos hemos encontrado ha sido la extrema linealidad de F#, que obliga continuamente a establecer un nuevo modelado o utilizar "malas" prácticas como variables globales. Lo cual hacía que hubiese cambios drásticos en la estructura del programa.

Sin embargo, una vez que se aprende cómo funciona F#, intuitivamente se encuentran los mejores lugares y las mejores técnicas para hacer funcionar el modelo. A la conclusión que podemos llegar en este caso es que hay que tratar F# como F#, es decir, aunque tenga componentes de la programación imperativa hay que intentar usar su parte funcional.

Por último se exponen algunas limitaciones de la implementación y líneas de trabajo futuro para su mejora.

Uno de los problemas principales que tienen las criaturas al moverse es la carencia de pathfinding[49]. Las criaturas simplemente se mueven hacia o en contra del objetivo que han decidido, pero no son capaces de trazar una ruta para evitar colisiones contra otras criaturas. Esto genera un grave problema ya que las criaturas "se quedan" atascadas continuamente y pueden llevar incluso a la desaparición de una especie.

Se ha estudiado métodos para evitar esto pero se ha llegado a la conclusión de que en este sistema no es viable, ya que necesitamos calcular por cada criatura un sinfín de puntos medios por los que pueda pasar, con lo que el rendimiento del programa decrecía exponencialmente. Esto es debido fundamentalmente a que el mundo cambia realmente rápido.

[49] https://en.wikipedia.org/wiki/Pathfinding

Aunque en el anteproyecto se mencionaba la posibilidad de intentar implementar algún algoritmo de aprendizaje, esto no ha sido posible. Después de estudiar diversas técnicas, entre ellas una parecida al sistema de decisión por eventos, una vez implementadas no lograban encajar bien, ya que las criaturas mueren muchísimo más rápido de lo que pueden aprender.

Otro problema relacionado con el aprendizaje es que, si las criaturas logran aprender, suelen aprender más un comportamiento erróneo que correcto, lo cual no beneficia en nada al proyecto y por eso fue descartado en etapas tempranas del mismo.

Figuras

Bibliografía

[1] Microsoft, «Graphics Class,» [En línea]. Available: http://msdn.microsoft.com/en-us/library/system.drawing.graphics(v=vs.110).aspx?cs-save-lang=1&cs-lang=fsharp#code-snippet-1.

[2] J. Harrop, F# for Scientists, Wiley, 2008.

[3] D. Syme, A. Granicz y A. Cisternino, Expert F#, Apress, 2007.

[4] F# Software Foundation, «F# Software Foundation,» [En línea]. Available: http://fsharp.org/.

[5] «Folders in F#,» [En línea]. Available: https://web.archive.org/web/20120116085906/http://cultivatingcode.com/2010/02/12/folders-in-f-projects/.

[6] Microsoft, «Drawellipse,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawellipse%28v=vs.110%29.aspx.

[7] Microsoft, «Fillellipse,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.graphics.fillellipse%28v=vs.110%29.aspx.

[8] Microsoft, «Draw closedcurve,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawclosedcurve%28v=vs.110%29.aspx.

[9] Microsoft, «Fill ClosedCurve,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.graphics.fillclosedcurve%28v=vs.110%29.aspx.

[10] Microsoft, «Drawing wrapmode,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.drawing2d.wrapmode%28v=vs.110%29.aspx.

[11] Microsoft, «Draw image,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.graphics.drawimage%28v=vs.110%29.aspx.

[12] Wikipedia, «PNG,» [En línea]. Available: https://en.wikipedia.org/wiki/Portable_Network_Graphics.

[13] Wikipedia, «DLL,» [En línea]. Available: https://en.wikipedia.org/wiki/Dynamic-link_library.

[14] Microsoft, «Get Manifiest Resource Stream,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/vstudio/xc4235zt%28v=vs.100%29.aspx.

[15] Wikipedia, «Age of Empire,» [En línea]. Available: https://en.wikipedia.org/wiki/Age_of_Empires.

[16] Wikipedia, «Quadtree,» [En línea]. Available: https://en.wikipedia.org/wiki/Quadtree.

[17] K. Schouviller, «Quadtree design,» [En línea]. Available: http://www.kyleschouviller.com/xna/quadtree-code-design/.

[18] C. W. Reynolds, «Steering Behaviors for Autonomus Characters,» [En línea]. Available: http://www.red3d.com/cwr/steer/gdc99/.

[19] Zet code, «Importing and Exporting data in MySQL,» [En línea]. Available: http://zetcode.com/databases/mysqltutorial/exportimport/.

[20] Zet Code, «MySQL C# Tutorial,» [En línea]. Available: http://zetcode.com/db/mysqlcsharptutorial/.

[21] S. Toub, «.NET Matters, ThreadPoolWait and HandleLeakTracker,» MSDN Magazine, 2004. [En línea]. Available: https://msdn.microsoft.com/en-us/magazine/cc163914.aspx.

[22] Microsoft, «Rectangle,» [En línea]. Available: msdn.microsoft.com/en-us/library/system.drawing.rectanglef(v=vs.110).aspx.

[23] Microsoft, «RemoveAll from List,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/wdka673a%28v=vs.110%29.aspx?cs-save-lang=1&cs-lang=fsharp#code-snippet-1.

[24] Microsoft, «AddRange List,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/z883w3dc%28v=vs.110%29.aspx.

[25] Microsoft, «Rectangle Intersection method,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.rectangle.intersectswith%28v=vs.110%29.aspx.

[26] Wikipedia, «Ray Casting,» [En línea]. Available: http://en.wikipedia.org/wiki/Ray_casting.

[27] Wikipedia, «Intersecting lines,» [En línea]. Available: https://en.m.wikipedia.org/wiki/Line%E2%80%93line_intersection#Given_two_points_on_each_line.

[28] Microsoft, «Graphics class,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.graphics%28v=vs.110%29.aspx.

[29] Microsoft, «Compositing mode,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.drawing.drawing2d.compositingmode%28v=vs.110%29.aspx.

[30] Wikipedia, «CPU,» [En línea]. Available: https://en.wikipedia.org/wiki/Central_processing_unit.

[31] Microsoft, «Multhreading in Windows Form Controls,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/ms229730(v=vs.80).aspx.

[32] Microsoft, «How to: Make Thread-Safe Calls to Windows Form Controls,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/ms171728%28v=vs.80%29.aspx.

[33] Microsoft, «Background Worker class,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker%28v=vs.80%29.aspx.

[34] Microsoft, «Background Worker isBusy property,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.componentmodel.backgroundworker.isbusy(v=vs.110).aspx.

[35] A. Princess, «Async Workflows,» [En línea]. Available: https://en.wikibooks.org/wiki/F_Sharp_Programming/Async_Workflows.

[36] Microsoft, «Thread class,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.threading.thread%28v=vs.110%29.aspx.

[37] Emiswelt, «Restarting a thread in .NET,» [En línea]. Available: http://stackoverflow.com/questions/1054889/restarting-a-thread-in-net-using-c.

[38] Microsoft, «ThreadPool class,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.threading.threadpool(v=vs.110).aspx.

[39] Microsoft, «ThreadPool QueueUserWorkItem Method,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/4yd16hza%28v=vs.110%29.aspx.

[40] Microsoft, «WaitCallback Delegate,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.threading.waitcallback%28v=vs.110%29.aspx.

[41] Microsoft, «Manual Reset Event,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.threading.manualresetevent%28v=vs.110%29.aspx.

[42] Microsoft, «Lock,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/ee370413.aspx.

[43] Wikipedia, «Norma vectorial,» [En línea]. Available: https://es.wikipedia.org/wiki/Norma_vectorial.

[44] Wikipedia, «Atan2,» [En línea]. Available: https://en.wikipedia.org/wiki/Atan2.

[45] Oracle, «MySQL,» [En línea]. Available: https://www.mysql.com/.

[46] Alter way, «Wampserver,» [En línea]. Available: http://www.wampserver.com/en/.

[47] phpMyAdmin contributors, «phpMyAdmin,» [En línea]. Available: http://www.phpmyadmin.net/home_page/index.php.

[48] Microsoft, «Dataset class,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.data.dataset(v=vs.110).aspx.

[49] Microsoft, «Form class,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/System.Windows.Forms.Form(v=vs.110).aspx.

[50] Microsoft, «How to: Reduce Graphics Flicker with Double Buffering for Forms and Controls,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/3t7htc9c%28v=vs.110%29.aspx.

[51] Microsoft, «Datagrid.Datasoure property,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.windows.forms.datagrid.datasource(v=vs.110).aspx.

[52] Microsoft, «Datagrid class,» [En línea]. Available: https://msdn.microsoft.com/en-us/library/system.windows.controls.datagrid(v=vs.110).aspx.

[53] Wikipedia, «Pathfinding,» [En línea]. Available: https://en.wikipedia.org/wiki/Pathfinding.

[54] I. Millington, Artificial Intelligence for Games, Morgan Kaufman, 2006.

Anexo 1: Script para la base de datos

```
-- Este archivo crea una nueva base de datos con todas las tablas
-- necesarias para el proyecto ecosistema, pero no contiene datos
-- Creado por Antonio Vidal Carrasco
-- Usando Wamp y mysql
-- Host: localhost    Database: proyecto_ecosistema
-- ------------------------------------------------
-- Server version    5.6.17
DROP DATABASE IF EXISTS proyecto_ecosistema;
CREATE DATABASE proyecto_ecosistema;
USE proyecto_ecosistema;
DROP TABLE IF EXISTS world_properties;
CREATE TABLE world_properties (
        id int(11) NOT NULL AUTO_INCREMENT,
        save_date datetime DEFAULT CURRENT_TIMESTAMP,
        biome varchar(50) NOT NULL,
        maxCreatures int(11) NOT NULL,
        worldName varchar(200) NOT NULL,
        worldWidth int(11) NOT NULL,
        worldHeight int(11) NOT NULL,
        idCount int(11) NOT NULL,
        objectCount int(11) NOT NULL,
        creatureCount int(11) NOT NULL,
        speciesCount int(11) NOT NULL,
        PRIMARY KEY (id)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS ui_properties;
CREATE TABLE ui_properties (
        world_id int(11) NOT NULL,
        sensorialRectangles int(1) NOT NULL,
        shadows int(1) NOT NULL,
        drawRec int(1) NOT NULL,
        showData int(1) NOT NULL,
```

```
	quadtree int(1) NOT NULL,
	multiThreading int(1) NOT NULL,
	PRIMARY KEY (world_id),
	FOREIGN KEY (world_id)
		REFERENCES world_properties (id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS shape;
CREATE TABLE shape (
	id int(11) NOT NULL AUTO_INCREMENT,
	world_id int(11) NOT NULL,
	PRIMARY KEY (id, world_id),
	FOREIGN KEY (world_id)
		REFERENCES world_properties (id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS vector;
CREATE TABLE vector (
	id int(11) NOT NULL AUTO_INCREMENT,
	world_id int(11) NOT NULL,
	x float NOT NULL,
	y float NOT NULL,
	PRIMARY KEY (id, world_id),
	FOREIGN KEY (world_id)
		REFERENCES world_properties (id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS color;
CREATE TABLE color (
	id int(11) NOT NULL AUTO_INCREMENT,
	world_id int(11) NOT NULL,
	R int(11) NOT NULL,
	G int(11) NOT NULL,
	B int(11) NOT NULL,
```

```
	PRIMARY KEY (id, world_id),
	FOREIGN KEY (world_id)
		REFERENCES world_properties (id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS image_shape;
CREATE TABLE image_shape (
	shape_id int(11) NOT NULL,
	world_id int(11) NOT NULL,
	vector_id int(11) NOT NULL,
	food int(11) DEFAULT 0,
	PRIMARY KEY (shape_id, world_id),
	FOREIGN KEY (shape_id, world_id)
		REFERENCES shape (id, world_id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS creature_shape;
CREATE TABLE creature_shape (
	shape_id int(11) NOT NULL,
	world_id int(11) NOT NULL,
	colorBody_id int(11) NOT NULL,
	colorTexture_id int(11) NOT NULL,
	vector_id int(11) NOT NULL,
	lon int(11) DEFAULT 8,
	tex int(11) NOT NULL,
	tileid int (11) NOT NULL,
	PRIMARY KEY (shape_id, world_id),
	FOREIGN KEY (shape_id, world_id)
		REFERENCES shape (id, world_id)
		ON DELETE CASCADE,
	FOREIGN KEY (colorBody_id)
		REFERENCES color (id)
		ON DELETE CASCADE,
	FOREIGN KEY (colorTexture_id)
```

```
                REFERENCES color (id)
                ON DELETE CASCADE,
        FOREIGN KEY (vector_id)
                REFERENCES vector (id)
                ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS creature_specie_properties;
CREATE TABLE creature_specie_properties (
        id int(11) NOT NULL AUTO_INCREMENT,
        specie_id int(11) NOT NULL,
        shape_id int(11) NOT NULL,
        world_id int(11) NOT NULL,
        lifeTimeRate float NOT NULL,
        aggressivenessRate float NOT NULL,
        hungryRate float NOT NULL,
        speedRate float NOT NULL,
        accelerationRate float NOT NULL,
        sensoryRate float NOT NULL,
        closeThresholdRate float NOT NULL,
        foodOffer float NOT NULL,
        PRIMARY KEY (id, specie_id, shape_id, world_id),
        FOREIGN KEY (shape_id, world_id)
                REFERENCES shape (id, world_id)
                ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS creature_properties;
CREATE TABLE creature_properties (
        id int(11) NOT NULL AUTO_INCREMENT,
        world_id int(11) NOT NULL,
        creature_specie_properties_id int(11) NOT NULL,
        sex int(1) NOT NULL,
        health int(11) NOT NULL,
        lifeTime int(11) NOT NULL,
        aggressiveness float NOT NULL,
```

```
	maxSpeed float NOT NULL,
	maxAcceleration float NOT NULL,
	senseWide float NOT NULL,
	closeWide float NOT NULL,
	hungryRate float NOT NULL,
	foodOffer float NOT NULL,
	hungry float NOT NULL,
	procreation float NOT NULL,
	bornChilds int(11) NOT NULL,
	ticksLived int(11) NOT NULL,
	PRIMARY KEY (id, world_id),
	FOREIGN KEY (creature_specie_properties_id)
		REFERENCES creature_specie_properties(id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS kinematic;
CREATE TABLE kinematic (
	id int(11) NOT NULL AUTO_INCREMENT,
	world_id int(11) NOT NULL,
	position_id int(11) NOT NULL,
	velocity_id int(11) NOT NULL,
	orientation float NOT NULL,
	PRIMARY KEY (id, world_id),
	FOREIGN KEY (world_id)
		REFERENCES world_properties (id)
		ON DELETE CASCADE,
	FOREIGN KEY (position_id)
		REFERENCES vector (id)
		ON DELETE CASCADE,
	FOREIGN KEY (velocity_id)
		REFERENCES vector (id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

```
DROP TABLE IF EXISTS entity;
CREATE TABLE entity (
        id int(11) NOT NULL,
        world_id int(11) NOT NULL,
        shape_id int(11) NOT NULL,
        kinematic_id int(11) NOT NULL,
        toDelete int(11) NOT NULL,
        PRIMARY KEY (id, world_id),
        FOREIGN KEY (shape_id, world_id)
                REFERENCES shape (id, world_id)
                ON DELETE CASCADE,
        FOREIGN KEY (kinematic_id)
                REFERENCES kinematic (id)
                ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS food;
CREATE TABLE food (
        entity_id int(11) NOT NULL,
        world_id int(11) NOT NULL,
        name varchar(50) DEFAULT 'Comida',
        initQuantity int(11) NOT NULL,
        offer int(11) NOT NULL,
        quantity int(11) NOT NULL,
        PRIMARY KEY (entity_id, world_id),
        FOREIGN KEY (entity_id, world_id)
                REFERENCES entity (id, world_id)
                ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS creature;
CREATE TABLE creature (
        entity_id int(11) NOT NULL,
        world_id int(11) NOT NULL,
        creature_properties_id int(11) NOT NULL,
        PRIMARY KEY (entity_id, world_id),
```

```
	FOREIGN KEY (entity_id, world_id)
		REFERENCES entity (id, world_id)
		ON DELETE CASCADE,
	FOREIGN KEY (creature_properties_id)
		REFERENCES creature_properties (id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS creature_allies;
CREATE TABLE creature_allies (
	creature_id int(11) NOT NULL,
	world_id int(11) NOT NULL,
	ally_id int(11) NOT NULL,
	PRIMARY KEY (creature_id, world_id, ally_id),
	FOREIGN KEY (creature_id, world_id)
		REFERENCES creature (entity_id, world_id)
		ON DELETE CASCADE,
	FOREIGN KEY (ally_id)
		REFERENCES creature (entity_id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
DROP TABLE IF EXISTS creature_enemies;
CREATE TABLE creature_enemies (
	creature_id int(11) NOT NULL,
	world_id int(11) NOT NULL,
	enemy_id int(11) NOT NULL,
	PRIMARY KEY (creature_id, world_id, enemy_id),
	FOREIGN KEY (creature_id, world_id)
		REFERENCES creature (entity_id, world_id)
		ON DELETE CASCADE,
	FOREIGN KEY (enemy_id)
		REFERENCES creature (entity_id)
		ON DELETE CASCADE
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```