David Sierra Gonzalez

Alex Canut Perdigón

Grupo 12, 2º A

TABLA DE CONTENIDOS

1. Juego

1.1 Selección de juego

El juego escogido para realizar el proyecto ha sido *snake*. La razón de escoger éste en lugar de las otras dos opciones ha sido que entendíamos de forma más óptima su lógica y tendríamos más facilidad de implementarlo con éxito, dado que uno de los miembros del grupo tenía experiencia con la programación del mismo. Aunque ha sido en un lenguaje totalmente distinto (Java), la lógica es la misma.

1.2 Adaptaciones, cambios y mejoras

A diferencia del simple *snake* original, que solo consistía en comer manzanas para aumentar la puntuación, éste además tiene obstáculos generados de forma aleatoria y niveles. Al pasar de nivel se crea un punto de guardado para que cuando mueras sigas desde ese punto y no desde el principio y también se genera un obstáculo nuevo, se elimina el que ya había y se coloca uno nuevo en una posición aleatoria.

También se empieza con tres vidas en lugar de una única. Estas vidas dan al jugador la posibilidad de alcanzar mayores puntuaciones en caso de morir una vez.

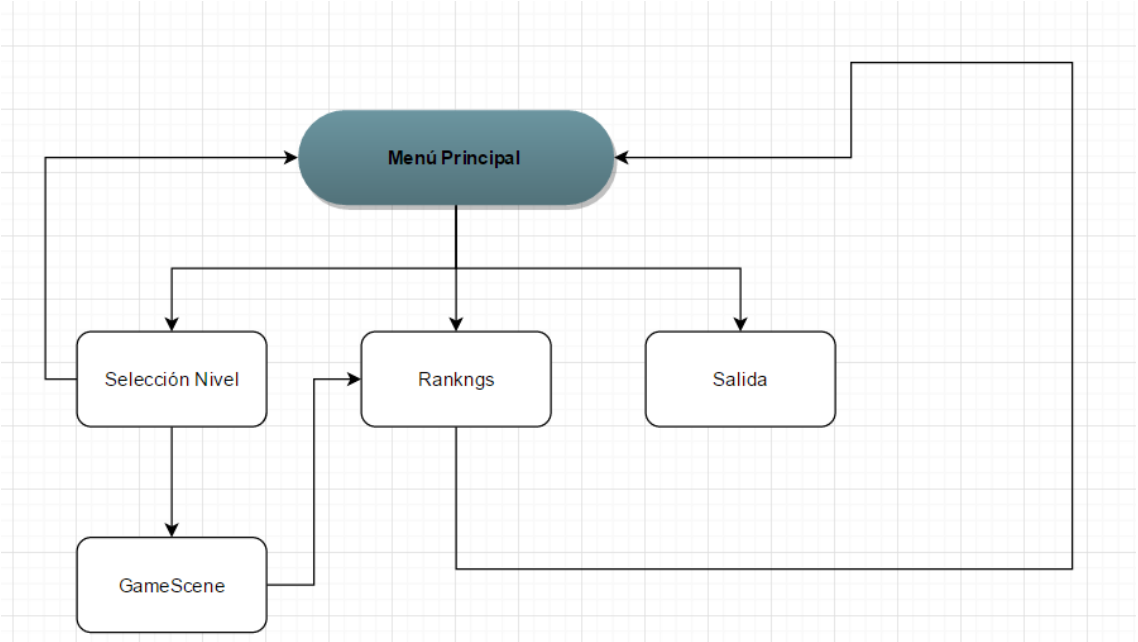
La puntuación aumenta de forma gradual de acuerdo con la puntuación del jugador. Por cada manzana consecutiva que consigue gana mayor puntuación.

Hemos implementado un menú principal dónde puedes comenzar a jugar, ir a la pantalla de *ranking* o salir de la aplicación. Cuando le das a jugar vas a otra pantalla para seleccionar la dificultad. Esto varia la velocidad del *snake*, el tipo de obstáculos, el tamaño del terreno y el tiempo.

El juego también tiene una pantalla de *ranking* donde el jugador puede añadir sus puntuaciones y consultar las de los demás usuarios que hayan jugado anteriormente. Además, se puede acceder a esta pantalla de *ranking* desde el menú principal.

1.3 Pantallas

El flujo de las pantallas es el siguiente:



- **Menú principal:** se muestra el título del juego y tres botones: jugar (selección de nivel), *rankings* o exit. Cada botón cambia la escena o el estado de la aplicación.
- **Niveles:** se muestran cuatro botones. Dificultad fácil, media y difícil, los cuales llevan a la escena *GameScene*, pero se cargan los datos correspondientes a esta dificultad desde un fichero *.xml.* El otro botón te permite volver al menú principal.
- ***GameScene:*** se muestran los elementos de juego (tabla, serpiente, manzana obstáculos, vidas, puntuación y Slider de tiempo) y es donde el jugador puede interactuar con el juego. Cuando se consumen las tres vidas o acaba el tiempo la escena cambia al *ranking*.
- **Rankings:** se muestran las 10 mejores puntuaciones de los jugadores según la dificultad. Si el jugador llega al *ranking* desde una partida se le mostraran las puntuaciones de la dificultad que acabe de jugar. Si se llega a *ranking* usando el botón del menú principal se le preguntará por consola que dificultad quiere que se muestre y se deberá introducir un numero en la consola. Entonces se cargará la escena después de leer el archivo binario.

2. Modelo: *Game Objects* y escenas

2.1 Game Objects y escenas

Para definir la Tabla he usado una matriz creada de forma dinámica (iteramos el primer *array* y en cada posición insertamos otro *array*). Esta matriz contendrá los *Sprites*. La matriz la creamos de forma dinámica para poder definir las dimensiones de cada dificultad cada vez que se empieza una partida. De lo contrario si quisiéramos volver sin reiniciar la aplicación no podríamos cambiar las dimensiones de la tabla. Además, aprovechamos las celdas para colocar los *sprites* que necesitemos y mover a la serpiente por ella utilizando las posiciones de las celdas como referencia.

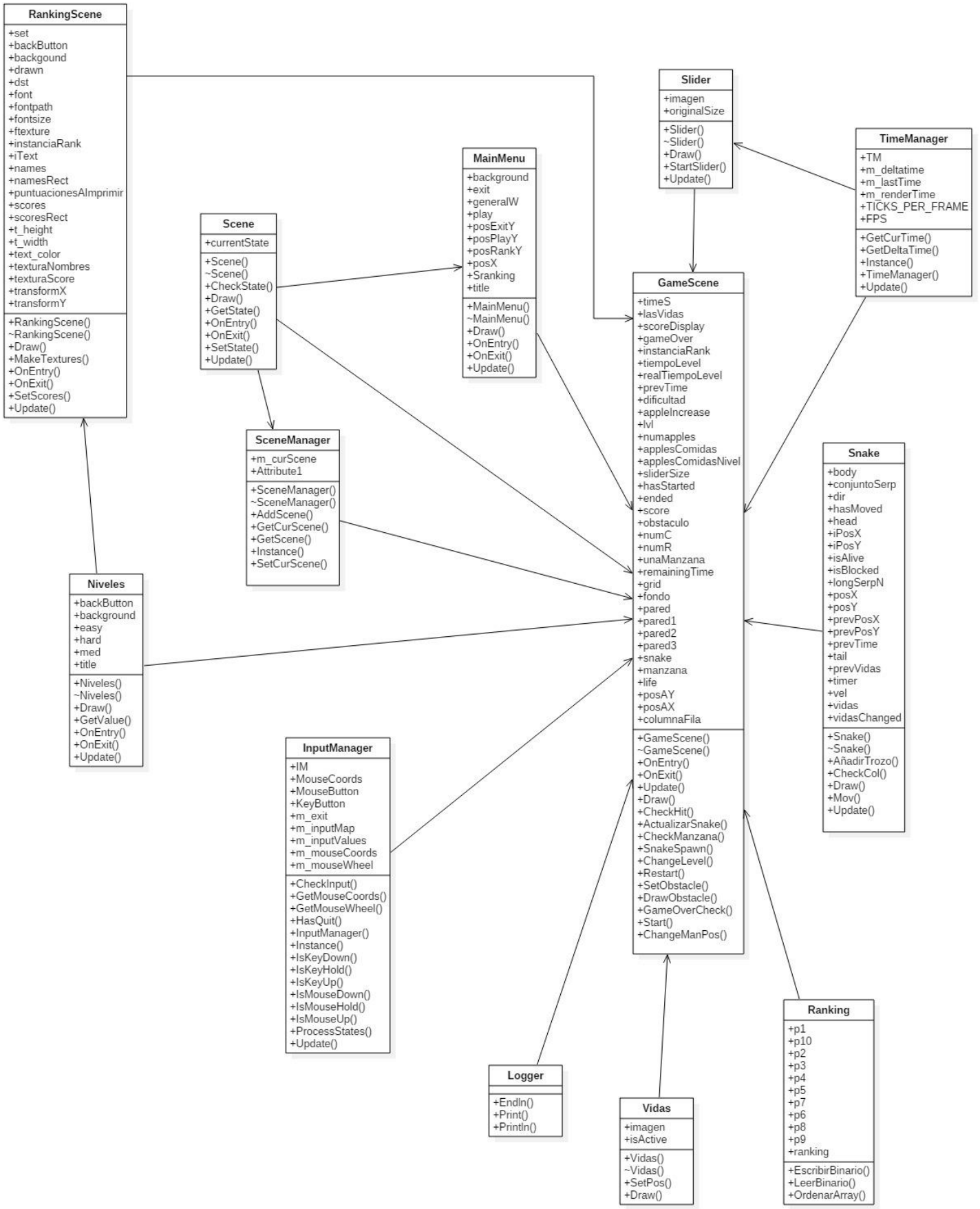
Para el conjunto de *sprites* que representan las vidas se usa un *array* de un *struct* “Vidas”. Este *struct* tiene un booleano que controla si se pinta o no el *sprite*.

El cuerpo de la serpiente está implementado con un *vector* de STL de *sprites*. Con esto conseguimos facilidad para borrar y añadir elementos de forma dinámica (necesario para variar el tamaño de la *snake*). Nuestro algoritmo de movimiento de la serpiente consta de mover la cabeza y añadir un nuevo nodo en la segunda posición del *vector* y eliminar el que hay en la segunda posición por lo tanto los métodos que proporciona la librería STL con *vector* nos facilitaban el trabajo.

Éste diseño de algoritmo para mover la *snake* lo hemos hecho pensando en no sobrecargar demasiado la máquina, ya que mover cada nodo independientemente en cada iteración del *update* es costoso y la sobrecarga aumenta de forma directamente proporcional al tamaño del *snake*. Por ello añadir un nuevo nodo y eliminar el último para dar sensación de movimiento creemos que es una solución más factible.

El ranking de jugadores está definido como un *array* de *structs* por conveniencia de la lectura y escritura de archivos binarios, que no permiten leer o escribir estructuras complejas o con memoria dinámica.

El obstáculo que hay en pantalla está guardado en un vector de *sprites*, en cada posición de este vector se añade el *sprite* con la posición en la cual está. Utilizamos vector porque es una estructura dinámica y cuando el obstáculo cambia puede que el nuevo no tenga la misma cantidad de *sprites* y por ello necesitamos variar el tamaño que hay en el *heap* reservado para esta estructura. No queremos que haya espacio reservado pero que no se utilice.


RankingScene
+set
+backButton
+backgound
+drawn
+dst
+font
+fontpath
+fontsize
+ftexture
+instanciaRank
+iText
+names
+namesRect
+puntuacionesAImprimir
+scores
+scoresRect
+t_height
+t_width
+text_color
+texturaNombres
+texturaScore
+transformX
+transformY
+RankingScene()
~RankingScene()
+Draw()
+MakeTextures()
+OnEntry()
+OnExit()
+SetScores()
+Update()
Slider
+imagen
+originalSize
+Slider()
~Slider()
+Draw()
+StartSlider()
+Update()
TimeManager
+TM
+m_deltatime
+m_lastTime
+m_renderTime
+TICKS_PER_FRAME
+FPS
+GetCurTime()
+GetDeltaTime()
+Instance()
+TimeManager()
+Update()
MainMenu
+background
+exit
+generalW
+play
+posExitY
+posPlayY
+posRankY
+posX
+Sranking
+title
+MainMenu()
~MainMenu()
+Draw()
+OnEntry()
+OnExit()
+Update()
Scene
+currentState
+Scene()
~Scene()
+CheckState()
+Draw()
+GetState()
+OnEntry()
+OnExit()
+SetState()
+Update()
GameScene
+timeS
+lasVidas
+scoreDisplay
+gameOver
+instanciaRank
+tiempoLevel
+realTiempoLevel
+prevTime
+dificultad
+appleIncrease
+lvl
+numapples
+applesComidas
+applesComidasNivel
+sliderSize
+hasStarted
+ended
+score
+obstaculo
+numC
+numR
+unaManzana
+remainingTime
+grid
+fondo
+pared
+pared1
+pared2
+pared3
+snake
+manzana
+life
+posAY
+posAX
+columnaFila
+GameScene()
~GameScene()
+OnEntry()
+OnExit()
+Update()
+Draw()
+CheckHit()
+ActualizarSnake()
+CheckManzana()
+SnakeSpawn()
+ChangeLevel()
+Restart()
+SetObstacle()
+DrawObstacle()
+GameOverCheck()
+Start()
+ChangeManPos()
SceneManager
+m_curScene
+Attribute1
+SceneManager()
~SceneManager()
+AddScene()
+GetCurScene()
+GetScene()
+Instance()
+SetCurScene()
Snake
+body
+conjuntoSerp
+dir
+hasMoved
+head
+iPosX
+iPosY
+isAlive
+isBlocked
+longSerpN
+posX
+posY
+prevPosX
+prevPosY
+prevTime
+tail
+prevVidas
+timer
+vel
+vidas
+vidasChanged
+Snake()
~Snake()
+AñadirTrozo()
+CheckCol()
+Draw()
+Mov()
+Update()
Niveles
+backButton
+background
+easy
+hard
+med
+title
+Niveles()
~Niveles()
+Draw()
+GetValue()
+OnEntry()
+OnExit()
+Update()
InputManager
+IM
+MouseCoords
+MouseButton
+KeyButton
+m_exit
+m_inputMap
+m_inputValues
+m_mouseCoords
+m_mouseWheel
+CheckInput()
+GetMouseCoords()
+GetMouseWheel()
+HasQuit()
+InputManager()
+Instance()
+IsKeyDown()
+IsKeyHold()
+IsKeyUp()
+IsMouseDown()
+IsMouseHold()
+IsMouseUp()
+ProcessStates()
+Update()
Logger
+EndIn()
+Print()
+PrintIn()
Vidas
+imagen
+isActive
+Vidas()
~Vidas()
+SetPos()
+Draw()
Ranking
+p1
+p10
+p2
+p3
+p4
+p5
+p7
+p6
+p8
+p9
+ranking
+EscribirBinario()
+LeerBinario()
+OrdenarArray()

2.2 Diseño del fichero XML de configuración del juego

El fichero XML está configurado de forma que tiene un nodo raíz “Atributos” y este tiene otros tres nodos, uno por dificultad. Cada nodo contiene los atributos que consideramos necesarios para generar la tabla de juego y configurar la dificultad en función de la que se ha seleccionado, en este caso *Columns*, *Rows*, *Time* (tiempo por nivel), V (Variable velocidad), *Food* (comida inicial), *IncrementoFood* (Aumento de comida por nivel), dificultad (por razones de *ranking*) y *SliderSize* (por razones de espacio en función del tamaño de la tabla).

El fichero XML es accedido utilizando una función estática. Esta función itera el fichero hasta llegar al nodo de la dificultad cuyos parámetros queremos cargar. Una vez situado en el nodo de la dificultad que queremos se iteran todos sus hijos y se guardan en un vector estático. A los valores de este vector podemos acceder mediante una función llamada *GetValue.* Esta función busca que valor es el que queremos i los devuelve.

2.3 Uso y/o modificación de la clase *SceneManager*

La clase *SceneManager* cumple con las mismas funciones que la clase *SceneManager* del modelo del *CandyCrush*.

Esta clase controla cual era la escena que estaba actualmente en pantalla (*SceneState RUNNING*) en función de la situación y ponía a “dormir” las demás escenas con SceneState SLEEP.

Se hacían llamadas a sus métodos *SetCurScene* en diversos casos:

- Haber perdido las 3 vidas e introducido el nombre por consola en el ranking.
- Estar en el menú principal y pulsar algún botón
- Estar en la escena Niveles y pulsar algún botón

En el caso de los botones todo salvo Salir (que llamaba al método SetState<SceneState::Exit>()) llamaban al método SceneManager.SetCurScene<UNAESCENA>().

3. Controlador: Game

3.1 Game. Atributos y métodos

GameEngine

- **AddScenes(void):** función a la que se llama una vez durante la ejecución que añade todas las escenas al *unorderer_map* de *SceneManager* que las contiene.
- **LoadMedia(void):** función a la que se llama una vez durante la ejecución y carga todos los Sprites deseados con sus respectivos IDs.

InputManager

- **Struct MouseCoords**: contiene las coordenadas en pantalla del puntero del ratón
- **Enum MouseButon:** contiene 3 elementos, uno por cada botón del ratón.

SceneManager

- **AddScene():** añade una nueva escena.
- **SetCurScene<>():** cambia la escena que hay cargada por otra.
- **m_scenes:** grupo donde se guardan todas las escenas.
- ***m_curScene:** puntero a la escena que hay cargada.

TimeManager

- **Update():** actualiza el tiempo y las variables necesarias para controlarlo usando la función GetCurTime

3.2 Eventos

Los eventos de juego a nivel de controles son el pulsado de flechas para cambiar la variable que gestiona la dirección del movimiento de la *snake* y los clics en los menús para pulsar los botones:

En el menú principal:

- Pulsar Jugar cambia la escena a la escena Niveles.
- Pulsar Ranking cambia la escena a la escena *RankingScene*.
- Pulsar Salir cierra la aplicación.

En la escena Niveles:

- Pulsar Fácil hace una llamada a la función XML con el valor *“easy”* y cambia la escena a *GameScene*.
- Pulsar Medio hace una llamada a la función XML con el valor *“medium”* y cambia la escena a *GameScene*.
- Pulsar Difícil hace una llamada a la función XML con el valor *“hard”* y cambia la escena a *GameScene*.
- Pulsar Back cambia la escena a la escena del menú principal.

En la escena Ranking:

- Pulsar back cambia la escena a la escena de menú principal.

En cualquier escena:

- El hecho de hacer clic sobre la X hace una llamada al método *Exit()* de la *SceneState* y cierra la aplicación.

A nivel de eventos de juego se estaría hablando de un listado bastante más largo.

El hecho de que *TM. CurrentTime()* sea superior al tiempo contado entre movimientos (influido por la velocidad) significa que la cabeza de la serpiente se moverá, generando una serie de llamadas a otros métodos en respuesta:

- Actualización de la posición del *sprite* de la cabeza (se mueve en función de las posiciones definidas en la tabla).
- Borrado el último *sprite* del vector de *sprites* que la *snake* y la inclusión de un nuevo *sprite* en la posición en la que se encontraba previamente la cabeza.
- Comprobación de colisiones (de la *snake* consigo misma (*CheckCol()*), con la manzana(*CheckManzana()*), con las paredes (*CheckHit()*) y con los obstáculos.
- Actualización de la variable que cuenta el tiempo entre iteraciones del *Update()*.

El hecho de que la comprobación de colisiones de la serpiente con sigo misma, con los obstáculos o con las paredes genera una llamada a una serie de métodos que:

- Reinician el contador de tiempo total del nivel.
- Restan una vida a la serpiente y aplican la puntuación, el tamaño de la *snake* y velocidad que tenía el jugador en el último punto de control, o en su defecto se establecen los valores del principio de la partida.
- Colocan la serpiente en su posición inicial y con dirección cero (*idle*).
- El booleano *vidasChanged* pase a ser true, lo cual llama a la función que hace uno de los objetos de la clase Vidas pase a tener *SetActive.false*, haciendo que no se pinte.

El hecho de que la serpiente colisione con la manzana genera una llamada a una serie de métodos que:

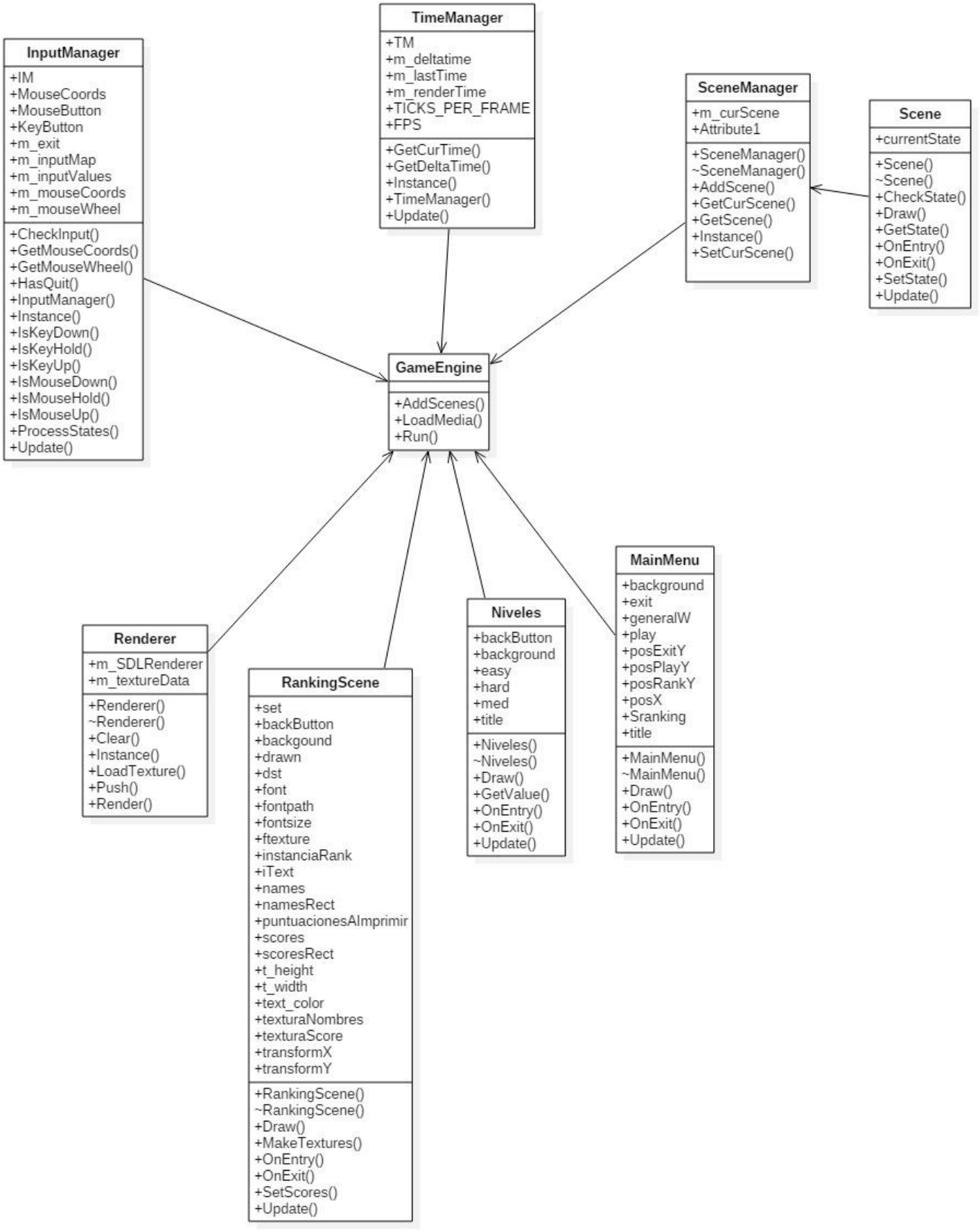
- Aumentan el tamaño en uno del vector de *sprites* que forma el cuerpo de la *snake*.
- Aumenta la puntuación del jugador en función de la cantidad de manzanas conseguidas consecutivamente.
- Comprueba cuántas manzanas lleva (en este nivel) y si el número de manzanas es igual al número de manzanas necesarias para superar el nivel, aumenta el nivel, lo cual implica que se genera un nuevo obstáculo aleatorio en una posición aleatoria.
- Cambia la posición de la manzana de forma aleatoria (En caso de estar en la misma posición que algún trozo de obstáculo vuelve a cambiar la manzana a otra posición aleatoria hasta que ésta consigue una posición disponible).

El hecho de que TM.CurrentTime() sea superior al tiempo calculado que tiene el jugador para superar un nivel (indicado mediante una barra que va disminuyendo) implica lo mismo las consecuencias de las colisiones.

El hecho de que el jugador pierda todas las vidas implica una llamada a las funciones que:

- Leen el fichero binario que contiene el ranking de la dificultad en la que se está jugando.
- Modifican los valores de este.
- Actualiza el fichero binario con los nuevos valores.
- Cambia la escena a la escena que muestra el *ranking*.

3.3 Diagrama de clases



3.4 Uso y/o modificación de la clase InputManager

El *InputManager* cumple la misma función que el establecido en el modelo del CandyCrush y se utiliza para detectar los clics de ratón, la posición del ratón al hacer estos clics y el pulsado de las teclas para modificar la dirección de movimiento de la *snake*.

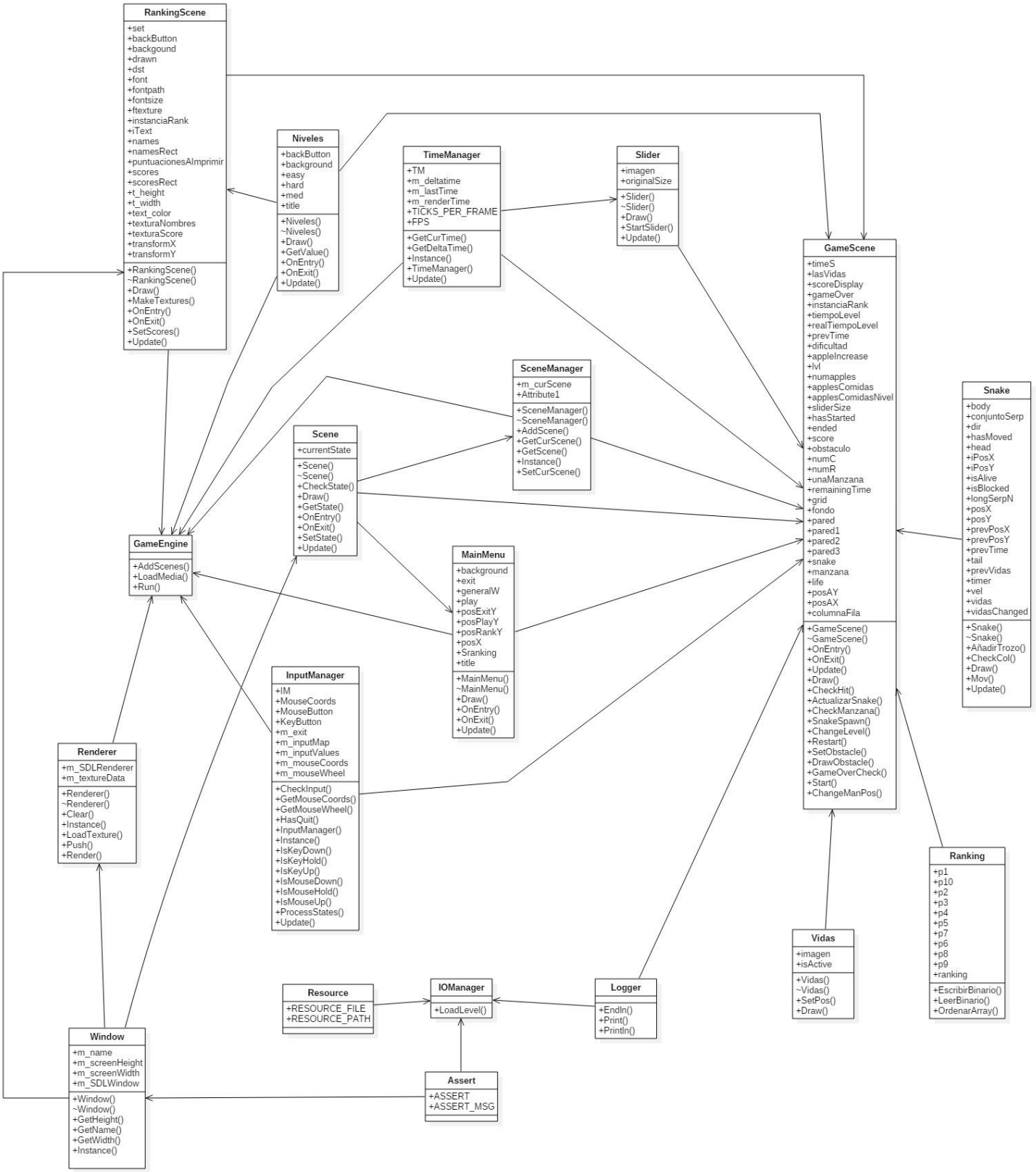
4. Vista: Renderer

4.1 Uso y/o modificación de la clase Renderer

La clase *Renderer* cumple la misma función que el establecido en el modelo del *CandyCrush*. Pintar en pantalla, en función de unos parámetros (posición, tamaño, fuente, ID de imagen) en el orden que se establece mediante las llamadas al método común de todas las escenas *Draw()*.

También carga los *sprites* y los deja listos para ser usados dentro de nuestra aplicación.

5. Diagrama de clases de todo el proyecto



6. Deployment

Para jugar a este juego fuera de Visual Studio se necesita tener en una carpeta los directorios “bin”, “res” y “dep” así como los archivos “rankingEasy.dat”, “rankingMedium.dat”, “rankingHard.dat” y “XMLFile.xml”.

No obstante, la aplicación continúa quedándose bloqueada cuando se le da la orden de cerrarse.

7. Estimación de tiempo

Alex Canut	**David Sierra**
• Diseño del archivo XML • Implementación y diseño del algoritmo de lectura del archivo XML • Diseño de los diagramas de flujo del proyecto • Diseño y edición de los *sprites* del juego. • Detección y corrección de errores. • Redactado y edición del report.	• Diseño e implementación del algoritmo de escritura y lectura de un archivo binario. • Diseño e implementación de la lógica de clases del proyecto. • Diseño de algoritmos. • Implementación de la lógica del juego. • Detección y corrección de errores. • Edición y simplificación de código. • Redactado del report.
Tiempo estimado: ~30h Vertical Slice: ~7 Alpha: ~5 Beta: ~19	Tiempo estimado: ~100h Vertical Slice: ~4 Alpha: ~5 Beta: ~90

8. Conclusiones

Lecciones aprendidas

Con este trabajo hemos aprendido a hacer uso de la librería SDL para crear juegos 2D que utilicen sprites, a gestionar lecturas de ficheros XML con la librería “rapidXML”, a leer y escribir *arrays* de *structs* en archivos binarios, a organizar un proyecto de Visual Studio con *headers* de forma que siga la estructura Vista-Modelo-Controlador y que la interdependencia de *headers* es un problema no tan fácil de detectar. Para solucionar esto último tienes que utilizar otra clase que haga de intermediaria e incluya estas dos clases que necesitan interdependencia.

Satisfacción

En general diría que estamos satisfechos con el resultado obtenido, salvo por el detalle de que, al cerrar la aplicación, si se ha escrito o leído en binario la aplicación deja de responder, lo cual es algo que no entendemos y no hemos logrado solucionar.

Dificultades de entendimiento y/o implementación

Lo más difícil de entender e implementar fue la lectura del archivo XML, el cual a pesar de mostrar una sencilla aplicación nos dio muchos problemas para leer los datos de forma correcta, y además también hubo problemas para lograr obtener los datos leídos una vez estaban leídos.

Fortalezas/debilidades del software

Fortalezas

- Al utilizar una matriz de *sprites* para generar un *grid* dividido en celdas del tamaño de los *sprites* que más adelante serían introducidos e interactuarían con el entorno, cuadrar las colisiones y el pintado de los *sprites* no representa ninguna formalidad, se podría cambiar el tamaño de todos los *sprites* siempre y cuando no provocarán que la tabla fuera más grande que la pantalla y que se cambie el tamaño de todos los *sprites* para ser el mismo.
- Hemos establecido una diferencia para llegar a la pantalla de ranking. Si llegas desde el botón del menú principal por consola te pedirá que selecciones el nivel de dificultad cuyas puntuaciones quieres mostrar, por otro lado si llegas al *ranking* desde el juego se te mostraran las puntuaciones del nivel de dificultad al cual acabas de jugar.

- Tenemos establecidos unos puntos de control cuando subes de nivel dentro del juego. Estos puntos de control guardan el estado de la partida en ese momento. En el caso de que el jugador muera vuelve a empezar, pero desde ese punto de control. Se carga el tamaño del *snake* de cuando llegó, la puntuación que tenía en ese momento y la velocidad a la que se movía la *snake*.
- El diseño del algoritmo para mover el snake que consta de avanzar la cabeza, añadir un nuevo nodo en ese espacio que queda entre la cabeza y el cuerpo y eliminar el ultimo nodo está optimizado porque no generas la sobrecarga que tendría al iterar toda la *snake* actualizando sus posiciones individualmente sobre todo cuando la *snake* va ganando tamaño.

Debilidades

- Al intentar cerrar la aplicación después de leer o escribir en el fichero binario ésta se queda bloqueada por alguna razón aún desconocida y obliga a cerrar el proceso de forma anormal.
- Al subir de nivel en una partida se genera un obstáculo aleatorio. Como la posición en la que se coloca es aleatoria puede suceder que éste se cree justo en la posición donde está la cabeza de la *snake*; esto provocaría la perdida de una vida inevitablemente.
- Por alguna razón aún desconocida el proceso de nuestra aplicación puede llegar a consumir alrededor 4.5Gb de memoria RAM. Esto hace que el ordenador se ralentice y no responda todo como debe y que incluso la aplicación deje de funcionar (no es muy frecuente este problema).

Referencias

- Tomado como ejemplo el Candy Crush de Jordi.
- Lectura de archivos xml: http://stackoverflow.com/questions/17683231/rapidxml-access-individual-attribute-value-using-the-previous-attribute-value
- Lectura y escritura de archivos binarios: http://campus.enti.cat/pluginfile.php/8475/mod_resource/content/1/IO_C%2B%2B.pdf

Anexos

Requisitos básicos
La serpiente se mueve mediante teclado correctamente.
El jugador tiene "k" vidas y al perderlas termina la partida.
Aparece 1 alimento aleatorio en escena cada vez que la serpiente se come uno.
La serpiente incrementa su tamaño al comer alimentos y se forma una cadena continua.
La serpiente colisiona con los límites de la pantalla y se pierde 1 vida.
La serpiente colisiona con su propio cuerpo y se pierde 1 vida.
Existe una barra de tiempo que disminuye progresivamente y al llegar a 0 el jugador pierde 1 vida.
Se puede pasar de nivel al consumir "x" alimentos.
Cada nivel sigue la fórmula de “x+y*n” alimentos, donde "x" es el número de alimentos, "n" es el número del nivel actual (suponiendo que nivel 1 es n = 0) e “y” es la cantidad de alimentos a añadir según la dificultad del juego.
La puntuación se suma correctamente según la fórmula “q*100” (siendo “q” el número del alimento en el nivel).
Aumenta la velocidad de la serpiente según la puntuación.
La matriz de casillas varía según la dificultad.
El tiempo inicial de juego varía según la dificultad.
La velocidad inicial de la serpiente varía según la dificultad.
El número inicial de alimentos varía según la dificultad.
El número incremental de alimentos varía según la dificultad.
Al perder una vida, se reinicia el nivel con el mismo tamaño que tenía la serpiente al pasar de nivel.
Requisitos adicionales
Cada nivel consta de obstáculos diferentes distribuidos por el mundo de juego.
La serpiente puede colisionar con los obstáculos y se pierde 1 vida.
Cada patrón de obstáculos está creado previamente en código o se carga desde archivo de texto plano.