Instituto Tecnológico de Costa Rica

Tarea Corta 3 - DonCEy Kong Jr

José Morales Vargas, carné 2019024270
Alejandro Soto Chacón, carné 2019008164

Area Académica de
Ingeniería en Computadores

Lenguajes, Compiladores
e intérpretes (CE3104)

Profesor Marco Rivera Meneses

Semestre I

Índice

DonCEy Kong Jr

1.1. Descripción de las estructuras de datos desarrolladas

Debe considerarse que entre las estructuras a listar solo se especificaran las del cliente en esta sección. Esto debido a que las estructuras en el servidor son mejor descritas por la relaciones entre clases, puesto que cada clase es en sí una estructura autocontenida. Las relaciones entre clases y qué representan es mejor cubierto en la sección **1.2** de este documento. La razón detrás de esta decisión es que sería redundante colocar los diagramas de clase en dos secciones distintas y repetir la misma explicación de las relaciones entre clases.

Vector

Un vector es un arreglo dinámico en el cual cada uno de sus elementos se encuentran contiguos en memoria. En el cliente se implementa un vector capaz de almacenar elementos genéricos. Este vector es utilizado como pieza fundamental en la construcción de otras estructuras de datos a ser descritas en las secciones posteriores. En la implementación, un vector es dado como un struct llamado `vec`, el cual contiene los siguientes campos:

- `data`: Es un puntero que indica el bloque de memoria en el que comienzan los datos del vector
- `lenght`: Indica cuantos elementos contiene el vector
- `capacity`: Indica la capacidad actual del vector
- `element_size`: Indica el tamaño en memoria que se requiere reservar para cada elemento del vector

hash map

Un hash map es una estructura de datos en la cual se mapean llaves a cierto valores. Un hashmap está compuesto por un vector de buckets, los cuales al mismo tiempo contienen un vector en el cual se almacenan las entradas mapeadas a cada bucket. La ventaja de esta estructura de datos es que el tiempo de acceso a cada elemento es relativamente constante, por lo cual se puede asegurar cierta consistencia de velocidad a la hora de buscar un elemento en la estructura.

En el servidor, se utiliza un hash map para el control de diferentes datos, por ejemplo, las entidades y los sprites. Estos mapas rara vez se encuentran solos, más bien suelen ser uno de los campos de alguna otra estructura superior, entre ellas la estructura con la cual se representa el juego en el cliente.

El hash map es representado pro un struct `hash_map` el cual contiene los campos:

- `buckets`: Es un vector que contiene los distintos buckets que componen el mapa

- `order`: Indica el orden del mapa. Se utiliza para el mapeo de valores
- `value_size` :Indica el tamaño en memoria que se debe reservar para cada valor a almacenar en el mapa

Juego

Un juego en el cliente se representa como un struct el cual contiene los campos:

- `state`: Valor de estado
- `net_fd`: Identificador de archivo descriptor de la conexión
- `x11_fd`: Identificador de archivo descriptor de pantalla
- `timer_fd`: Identificador de archivo descriptor de timer
- `net_file`: Stream del socket
- `window`: Ventana de SDL
- `renderer`: Renderizador de SDL
- `ticks`: unidades de tiempo transcurridas
- `sprites`: hash map de sprites del juego
- `entities`: hash map de entidades del juego
- `fullscreen`: valor que indica si la pantalla se debe dibujar como pantalla completa

Sprites

Un sprite del está conformado por dos elementos:

- `surface`: Conjunto de pixeles(superficie) en los que se dibuja el sprite
- `texture`: Imagen que se dibuja sobre la superficie del sprite.

Razones

Para expresar velocidades y otros valores relativos, se recurre a razones numéricas. Para esto se requiere de una forma de expresar fracciones matemáticas sin perder información. Dado que un tipo de dato de punto flotante era una opción inadecuada, se creo una estructura distinta que registra una razón entre dos números enteros, llamada `ratio`. Cada `ratio` tiene dos elementos:

- `numerator`: Numerador de la fracción
- `denominator`: Denominador de la fracción

Entidades

Para representar a los distintos tipos de elementos gráficos que se dibujan era necesario una estructura que contuviese todo lo necesario para el manejo y manipulación de las distintas

entidades. Para esto se recurrió a crear un struct `entity`, el cual contiene información de posición, velocidad, y algunos otros datos de estado. Los campos de dicho struct son

- `id`: Identificador de la entidad
- `x`: Posición horizontal de la entidad
- `y`: Posición vertical de la entidad
- `sequence`: Vector que contiene secuencia de sprites de animación del estado actual
- `next_sprite`: Siguiente sprite para animar a la entidad
- `speed_x`: velocidad horizontal expresada como una razón entre movimiento/ticks
- `speed_y`: velocidad vertical expresada como una razón entre movimiento/ticks

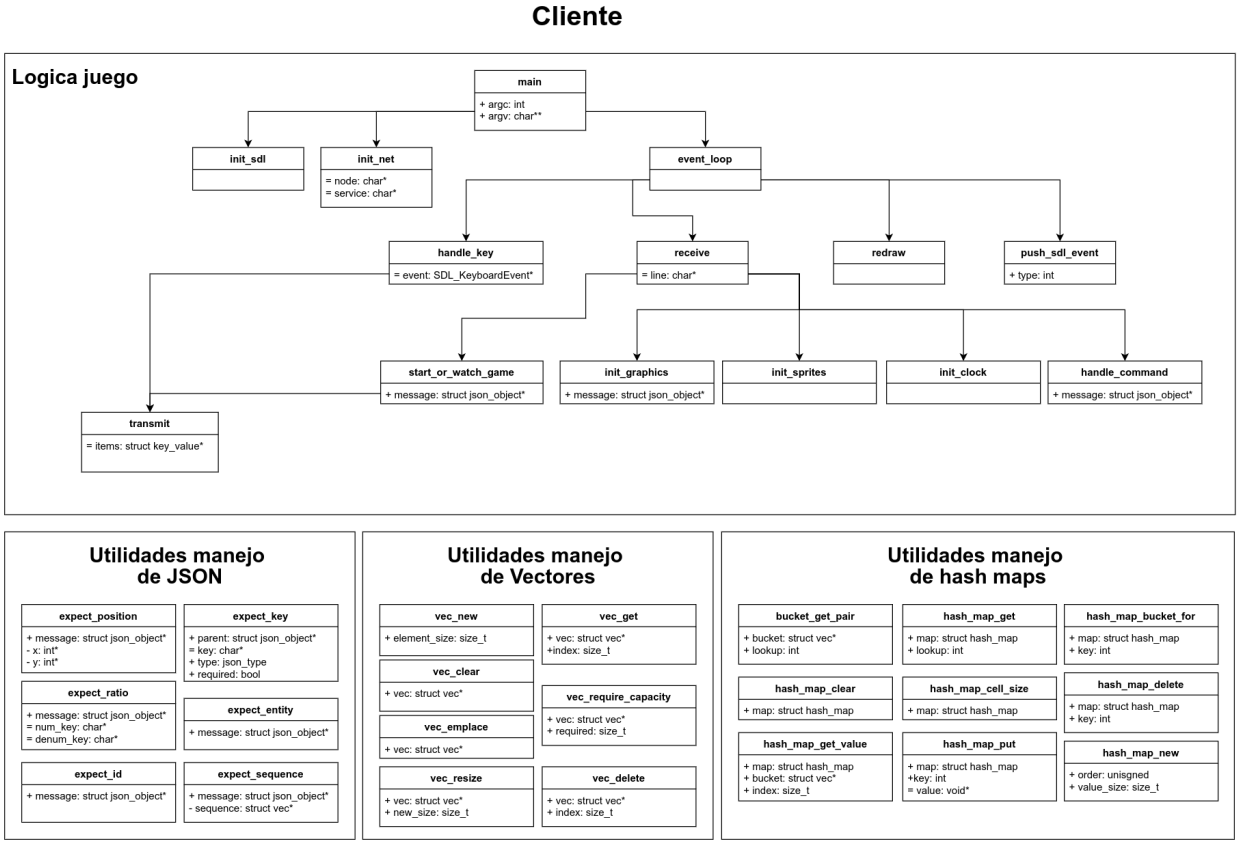
Par llave-valor

Dentro del cliente se hace uso del formato JSON para comunicación con el servidor. Esto hacia necesario el desarrollo de una estructura relativamente simple que sirviese para la descomposición de objetos JSON. Para esto se agregó un struct `key_value` el cual contiene los campos:

- `key`: String que identifica la llave del objeto json
- `value`: Valor que puede ser un valor en sí, o un objeto JSON anidado

1.2. Descripción detallada de algoritmos desarrollados

Cliente



Como se puede observar, el cliente tiene una funcionalidad relativamente simple en cuanto a relaciones entre componentes. Esto se debe a que, tal como se específico, el cliente solo es un intérprete de los comandos enviados por el servidor, ya que el servidor es quien maneja la lógica de juego en sí. El cliente no tiene noción ni siquiera de las colisiones, sino que su trabajo se centra en dibujar y captar acciones del usuario.

La rutina de inicio consiste en inicializar la biblioteca de gráficos y la conexión con el servidor, y terminadas estas dos tareas, comenzar con el ciclo de juego.

Cada ciclo de juego se dan los mismos pasos:

- Si se registró un evento de tecla, se envía un mensaje al servidor con la información de dicho evento.
- Se lee el stream del socket y según la entrada se sigue uno de tres flujos:
  - Iniciar como jugador o espectador: Envía un mensaje de handshake al servidor para comunicarle como el jugador quiere comenzar su instancia cliente.
  - Post handshake: Crea la pantalla, inicializa el timer de juego y carga los recursos gráficos.

    - Manejar comandos: Cuando ya el juego está activo, lee del socket los comandos que debe ejecutar localmente, tal crear entidades, dibujar objetos, entre otras operaciones posibles.
- Si el juego se encuentra en un estado en que debe refrescar la pantalla, la redibuja con `redraw()`

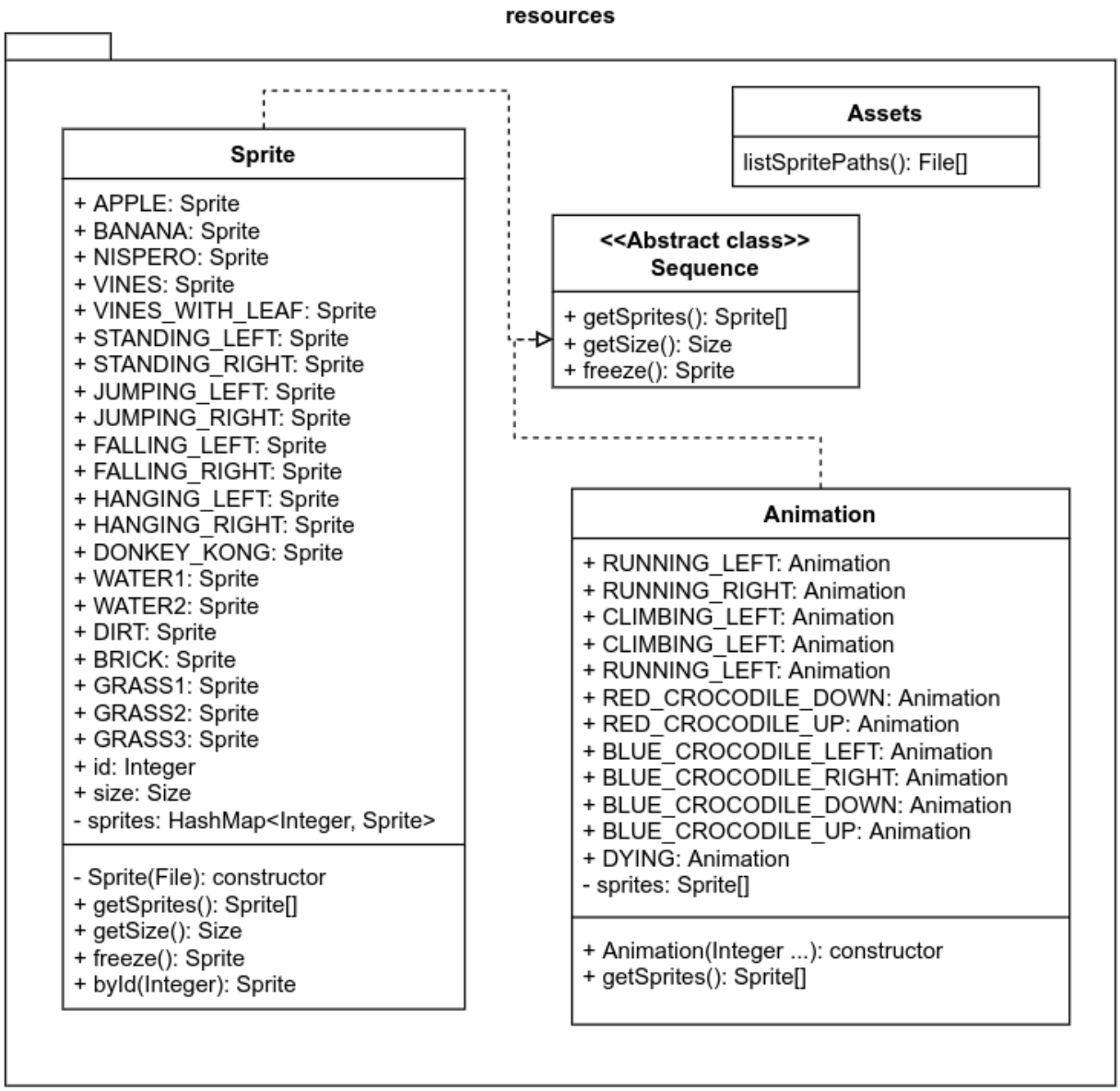
En el manejo de comandos, descifrado de instrucciones y control del juego, el cliente hace uso de tres categorías distintas de utilidades.

1. Utilidades manejo de JSON: Utilizadas para obtener información de los mensaje enviados por el servidor.
2. Utilidades de manejo de vectores: En varias secciones del programa se hace uso de vectores para diferentes propósitos. Esta funcionalidad es de utilidad general.
3. Utilidades de manejo de hash maps: Los elementos que componen la pantalla de juego y algunos otros son registrados por medio del uso de hash maps tal como se desarrollo en la sección 1.1. Estas funciones son utilizadas para la consulta y manipulación de estos hash maps.

Servidor

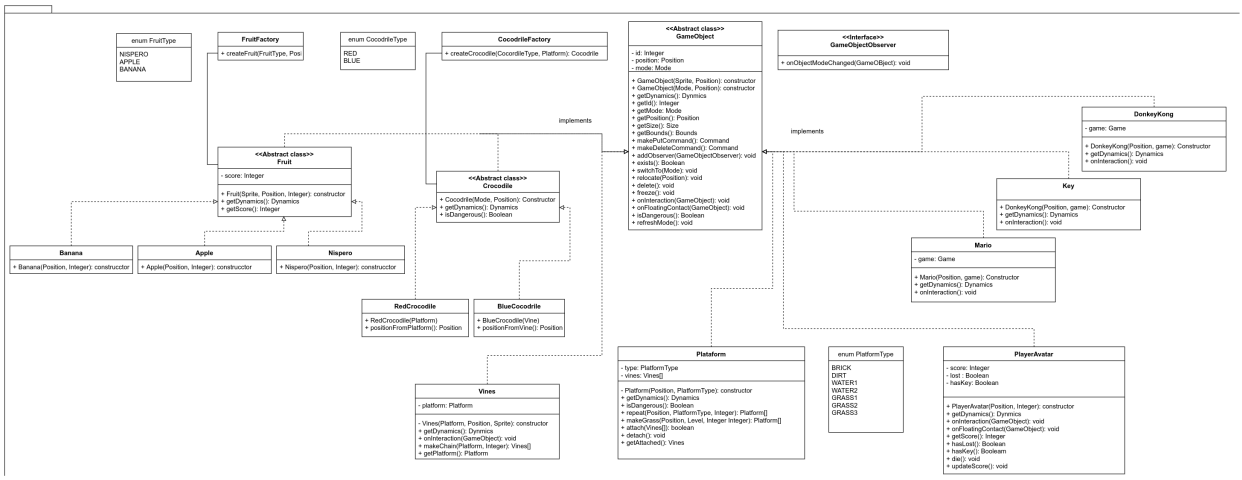
El servidor es un programa en java compuesto por algunas clases base y 6 paquetes que controlan distintos aspectos de la lógica de juego.

Primeramente, véase el diagrama para el paquete `resources`:



EL paquete cumple una funcionalidad relativamente simple. Es un conjunto de utilidades que proveen la noción al servidor del apartado gráfico del juego, o en resumidas cuentas, es el encargado de administrar lo que concierne a sprites y conjuntos de sprite que conforman una animación, pero no la representación de las entidades a las cuales les corresponden dichos sprites.

Las entidades son en cambio, representadas utilizando las diferentes clases del paquete `gameobjects`, visto en el siguiente diagrama



El paquete de `gameobjects` contiene lo necesario para representar todas las entidades que componen el juego, además de ofrecer algunas facilidades para construcción de escenarios. Como puede observarse, desde el jugador hasta las plataformas mismas son representados como instancias de `GameObject`.

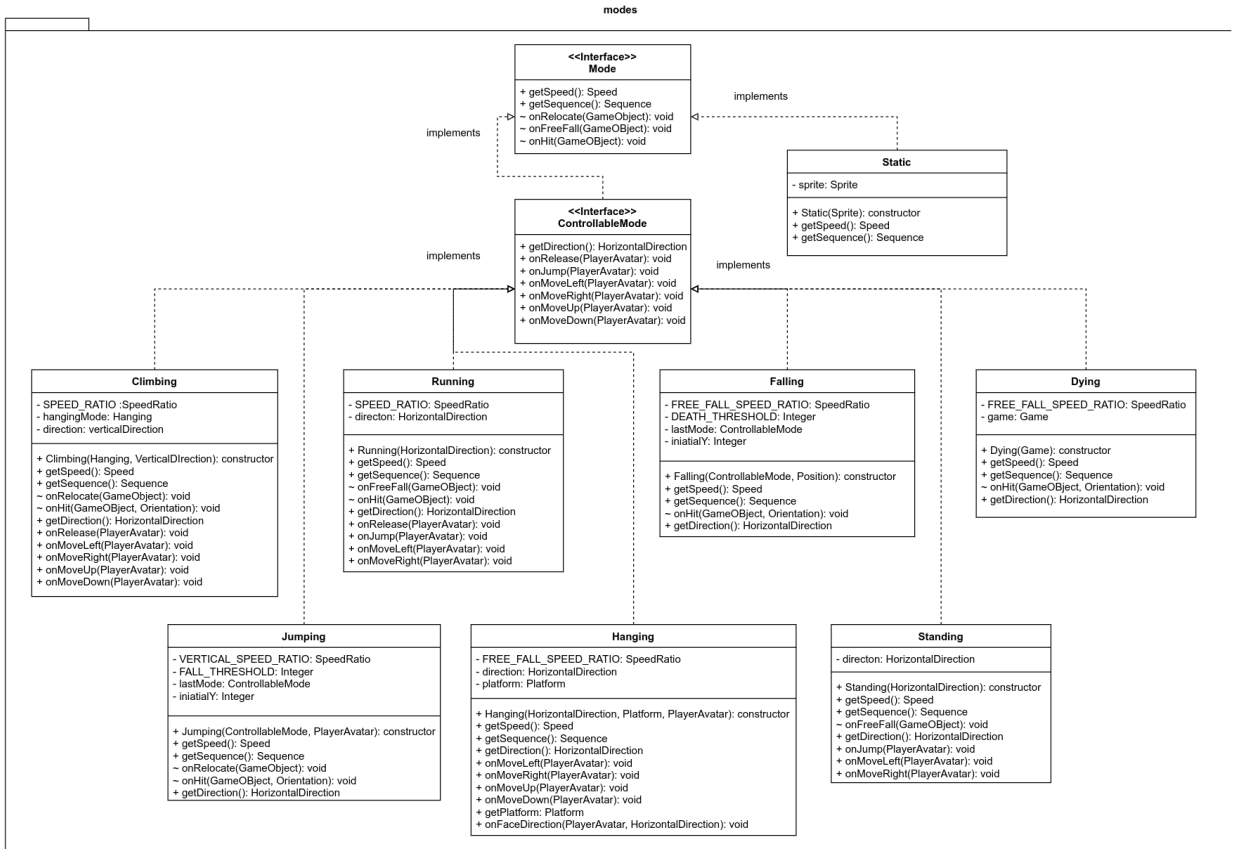
Cada subclase de `GameObject` contiene funcionalidad específica a sí misma, por ejemplo, puede observarse en Platform métodos que facilitan la creación tanto de plataformas de ladrillos como plataformas de pasto, facilitando considerablemente la creación de una escena de juego.

Para los objetos que puede ser colocados por el usuario administrador, se implementó un patrón de diseño factory, de manera que la creación de frutas y cocodrilos es relativamente transparente al usuario.

Se implementa además un patrón observer. Este patrón es una forma de permitirle a una clase ser notificada en caso de cambios en el estado interno de una entidad. Esto es principalmente utilizado para el objeto de juego en sí, el cual debe realizar ciertas operaciones si se da un cambio en el estado interno en alguna de todas las entidades que administra.

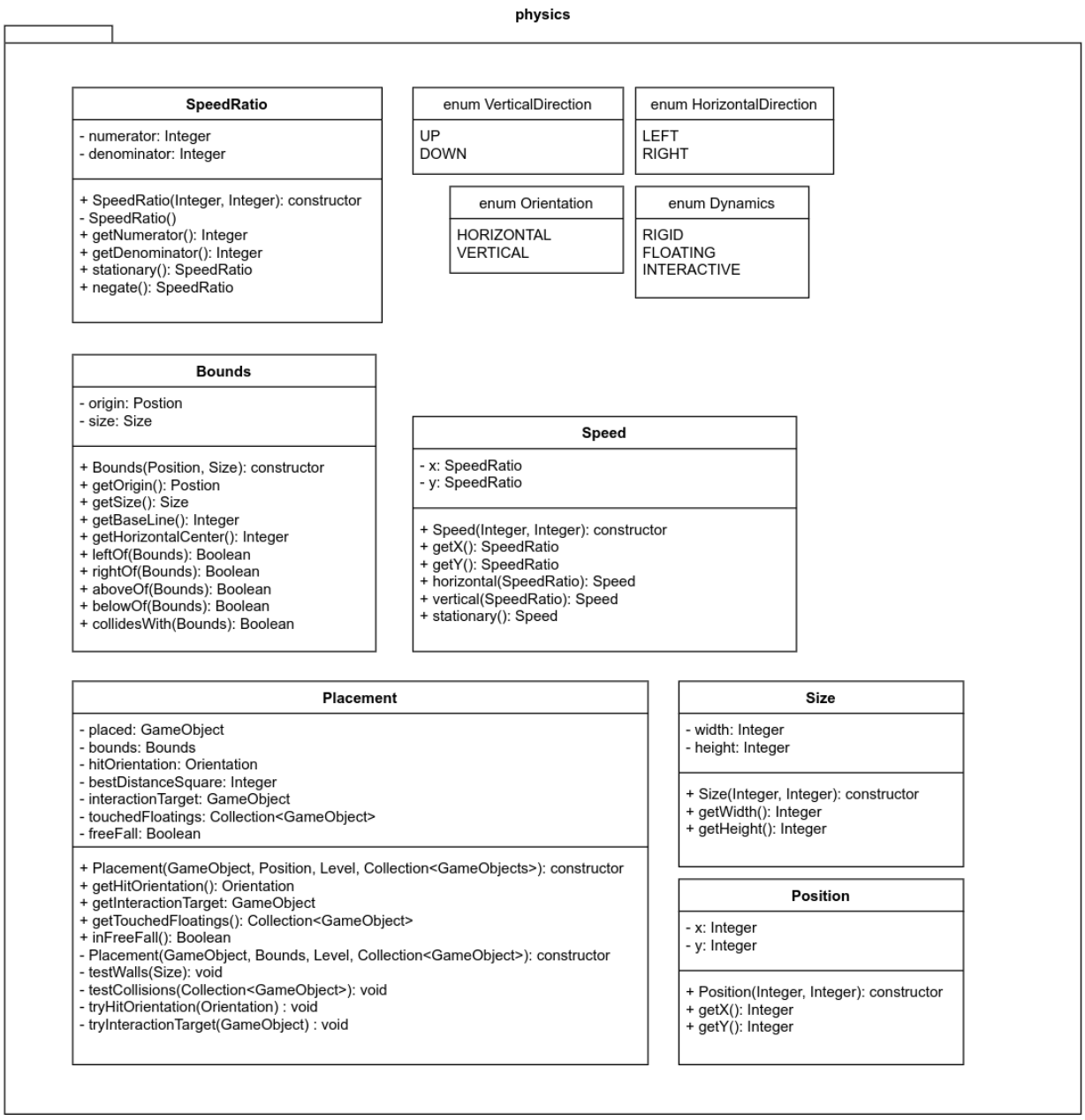
Según el tipo de `GameObject` se puede tener una observación, ¿Qué sucede con aquellos objetos que tienen estados transitivos, es decir, un objeto que se puede encontrar en varios "modos de operación"?

La respuesta a la pregunta anterior es dada por el paquete `modes`. Este paquete contiene diferentes clases que permiten definir en qué estado se encuentra un `GameObject`. Claramente no todas las entidades tienen estados transitivos, pero las entidades que sí, como el jugador, dependen fundamentalmente de poder diferencias entre estados como caminar, saltar, caer, etc.



Como puede observarse en el diagrama anterior, el paquete `modes` ofrece funcionalidad que le permite a una entidad distinguir entre sus estados, y las implicaciones de las diferencias entre los mismos. Por ejemplo, no todo estado tiene respuestas a un evento en específico. Lo anterior puede inferirse con solo pensar en si debería ser posible saltar mientras ya un jugador se encuentra en el aire. La respuesta a la cuestión anterior es un rotundo no, y por eso mismo puede observarse que a modos como `Falling` y `Jumping` no les concierne controlar que sucede una vez que se llame `onJump()`; simplemente si un jugador se encuentra en alguno de los dos estados anteriores, no puede suceder nada cuando el jugador presiona el botón de saltar.

Es relevante discutir también la forma en la que el servidor es capaz de siquiera procesar un estado como "caer". Para esto, el servidor no solo debe tener una noción de qué hay(`gameobjects`), o cómo luce(`resources`); el servidor requiere de algo que le permita tener una noción de las reglas que rigen el comportamiento y las interacciones de las entidades del juego. Para esto, está el paquete `physics`, el cual se muestra en el siguiente diagrama:



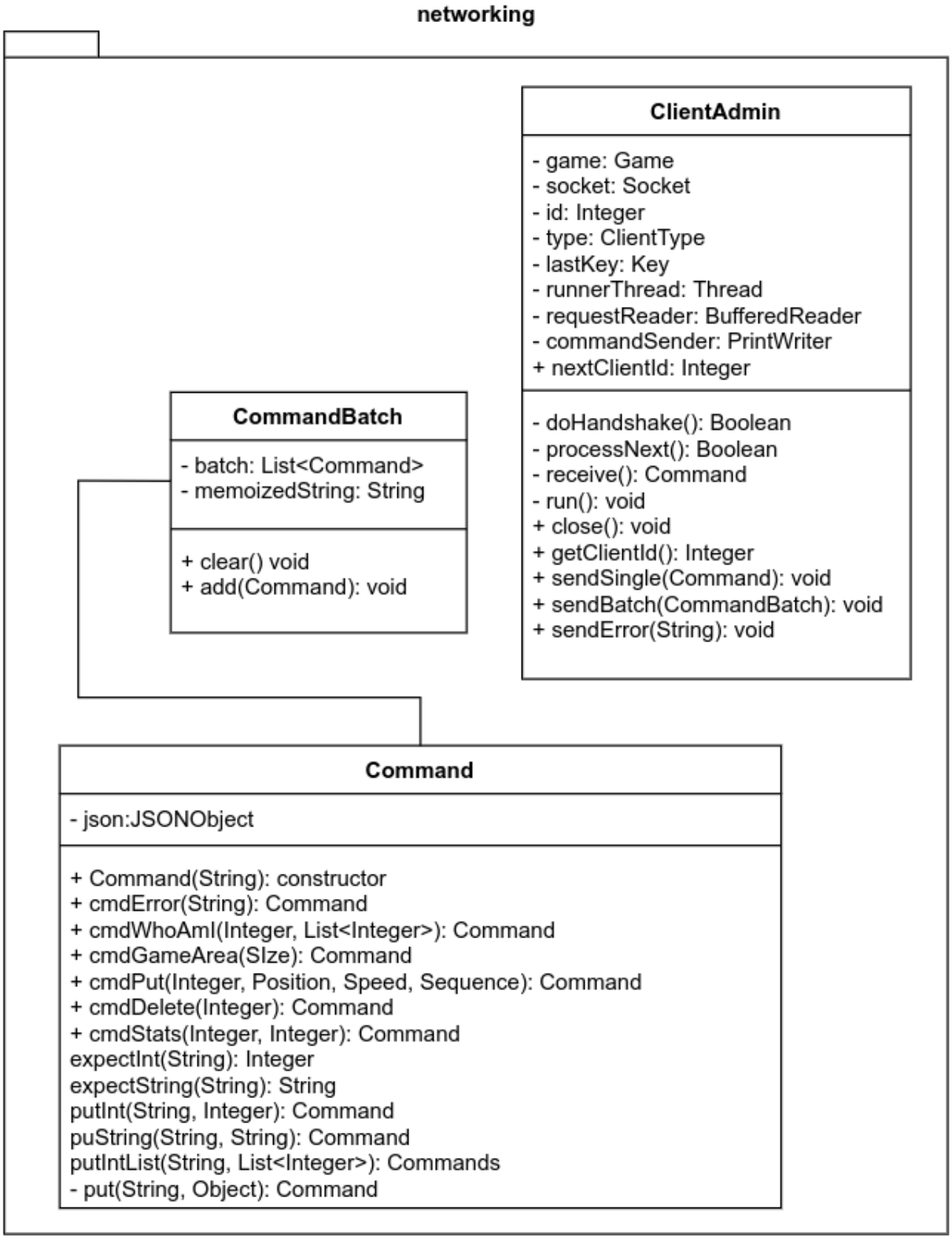
Como puede observarse, el paquete anterior funciona como el administrador de función elemental del juego. Las dos tareas más importantes que administra este paquete son velocidades y colisiones.

La primera tarea es esencial, puesto que no puede haber movimiento de una entidad si no se conoce qué distancia se mueve, en qué dirección, y en cuanto tiempo. El movimiento del jugador y cocodrilos depende fundamentalmente de este paquete.

La segunda tarea no debe considerarse menos importante. Las reglas que rigen una interacción de colisión, y los eventos que desencadena una no serían posibles si no fuese por

este paquete. Se sabe que el jugador no debe caer al vacío porque una colisión es detectada con el piso, se sabe que puede subir y bajar por una liana porque colisiona con la misma, sabemos que obtiene puntos al entrar en contacto con una fruta porque colisiona con ella. En síntesis, `physics` provee la lógica base que es el fundamento para establecer las reglas que rigen las interacciones entre entidades del juego.

Otra capa fundacional del juego, aunque no relacionada directamente a las entidades del escenario de juego, es el paquete de `networking`.

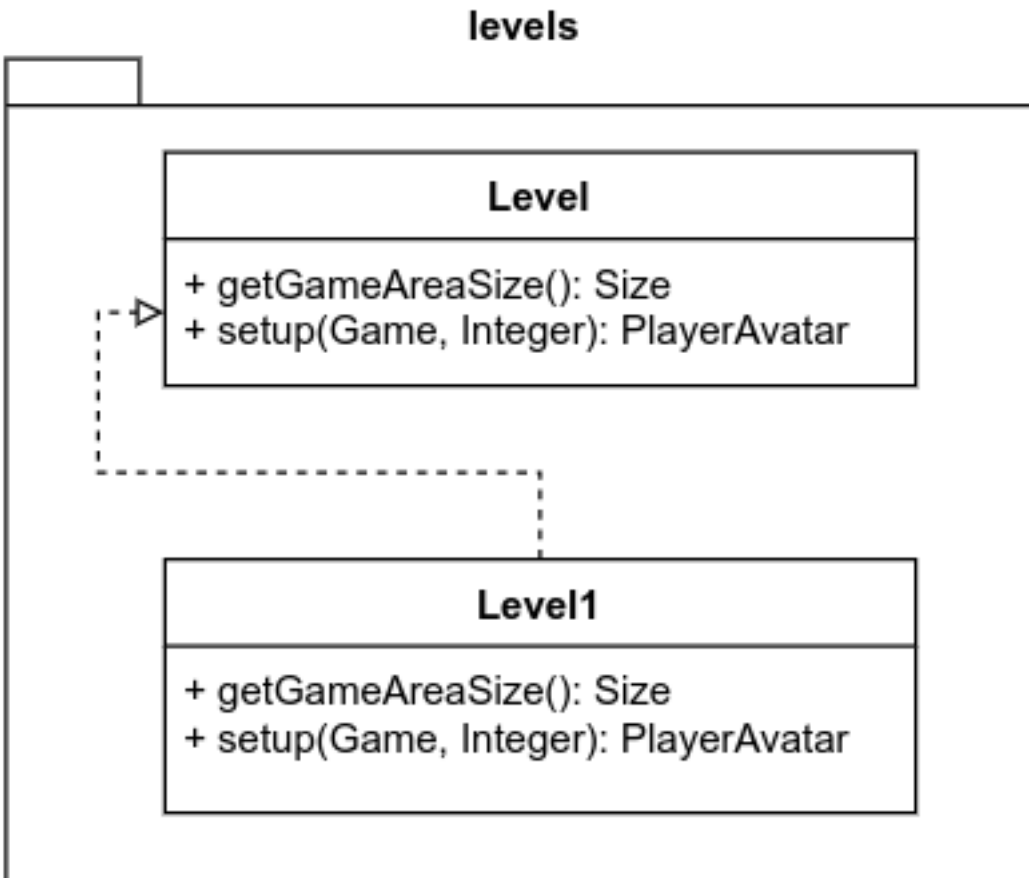


El paquete provee una capa de abstracción sobre el funcionamiento de la conexión con el cliente y lo que respecta a la misma, por ejemplo, como diferenciar entre clientes jugadores y

clientes espectadores. Este paquete también provee la funcionalidad que permite notificarle a un cliente sobre los cambios que deben reproducirse en el escenario de juego, puesto que es aquí donde se forman y envían los mensajes en formato JSON que debe interpretar el cliente al momento de correr el videojuego.

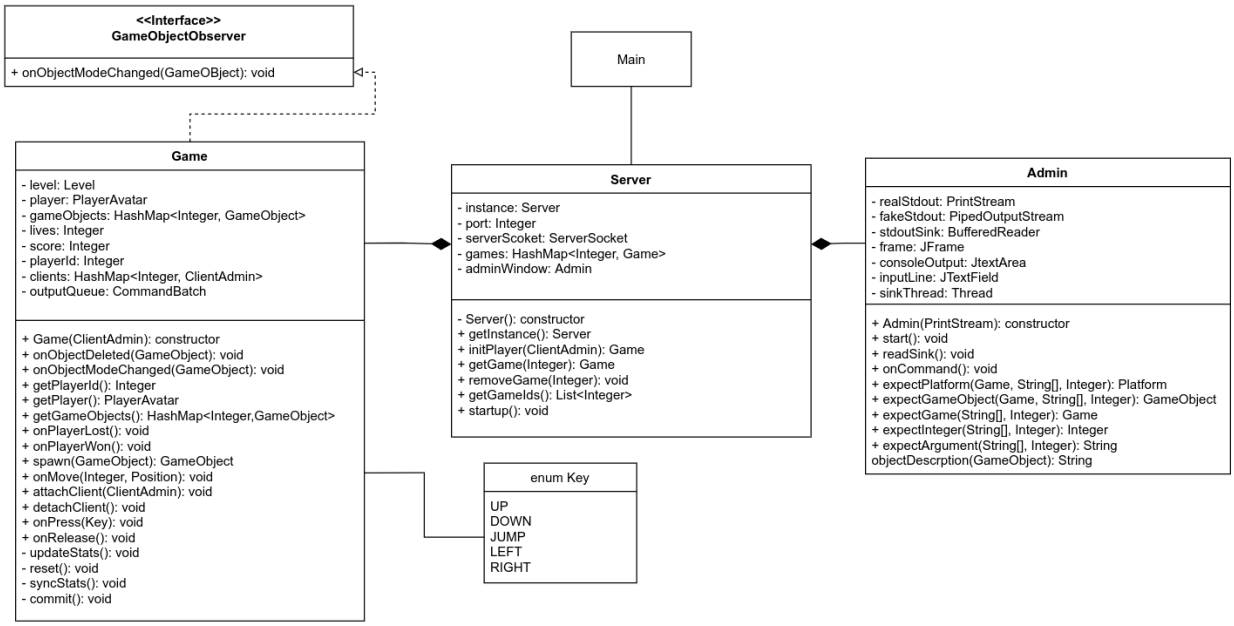
Una vez comprendidos los fundamentos que conforman el juego, se puede proceder a analizar las partes del programa que funcionan sobre estas capas fundacionales.

El primero de los paquetes de mayor nivel es precisamente el paquete `levels`.



Efectivamente el paquete solo es compuesto por una superclase Level, y una implementación de dicha clase para representar el nivel 1. El funcionamiento es relativamente simple, una instancia de `Level` permite dibujar un escenario de juego. Ya que el juego actual consta de un nivel único, solo hay una implementación de dicha superclase, sin embargo, planeando de forma anticipada, se provee una forma simple y fácil de agregar e interactuar con niveles adicionales si esto fuese necesario.

Finalmente, las clases en el directorio base del código del servidor son la última capa de lógica en la funcionalidad del mismo:



La noción final del servidor es en sí un compuesto entre dos elementos, una aplicación de administración, y un conjunto de juegos activos. Es gracias a las capas inferiores que es posible expresar la lógica general en una forma tan simple, y relativamente transparente al usuario; El sistema visto solo desde esta capa pareciese administrarse por sí solo.

Para evitar posibles conflictos en uso de puertos, entre otros posibles problemas, se recurre a utilizar el patrón de diseño Singleton en la implementación del Servidor. Solo se puede encontrar una instancia de Servidor activa, y no hay forma de crear una instancia adicional que pueda crear problemas, puesto que el constructor de la clase `Server` es privada.

Esta capa de lógica es efectivamente, la capa administrativa. Es por eso que se ubica alojada aquí la clase `Admin`. Una instancia de esta clase habilita una interfaz gráfica para que un administrador de sistema pueda realizar tareas como crear frutas y cocodrilos en el juego, por medio de los comandos indicados en el manual de usuario.

En el caso de game, se aprecia el flujo de inicio de una partida. El mismo constructor nos indica que es iniciado a través de una instancia de un cliente, dicho cliente es registrado como el dueño de la partida, y demás clientes de dicha partida son registrados como espectadores. Ya la lógica base ha sido administrada en capaz anteriores, por lo que a `Game` solo le conciernen cambios de estados mayores, por ejemplo, lo que sucede cuando un jugador pierde o gana, las vidas y puntaje del jugador, la integración de elementos de juego a la escena y las formas en las que un cliente puede interactuar con un juego. Como se mencionó anteriormente, `Game` funcionalmente es solo un administrador general, y solo le concierne controlar que sucede una vez que un elemento ya se ha auto-administrado. Es por eso que esta clase implementa la interface `GameObjectObserver`, puesto que si bien no le conciernen los detalles sobre los

cambios de estado de un objeto, sí le conciernen las consecuencias macro que implican dichos cambios de estado.

1.3. Problemas sin solución

1.4. Actividades realizadas por estudiante

Tarea	Descripción	Tiempo estimado	Responsable	Fecha estimada	Fecha de entrega
Reunión de coordinación inicial	Reunión para repartir las tareas iniciales a realizar durante los días iniciales	1 hora	Alejandro Soto José Morales	09/04/2021	09/04/2021
Configuración de repositorio	Crear el repositorio y agregar la estructura de proyecto base	5 min	José Morales	09/04/2021	09/04/2021
Reunión para formalizar plan desarrollo a ejecutar	Reunión para repartir tareas específicas y decidir las herramientas a utilizar para el desarrollo de la tarea	1 hora	Alejandro Soto José Morales	12/04/2021	12/04/2021
Elaboración de archivos base de documentación	Creación de archivos para almacenar la documentación externa del proyecto	20 min	José Morales	12/04/2021	12/04/2021
Creación de diagrama UML inicial	Diseño primitivo de la estructura del programa para tener una guía en el desarrollo de los módulos iniciales	1 hora	José Morales	14/01/1900	14/04/2021
Desarrollo de sockets del lado del cliente	Desarrollo de la lógica necesaria para que un cliente sea capaz de conectarse a un servidor	30 min	Alejandro Soto	18/04/2021	18/04/2021
Desarrollo de lógica de sockets en el servidor	Desarrollo de lógica necesaria para que el servidor pueda administrar las diferentes conexiones de clientes	30 min	Alejandro Soto	18/04/2021	18/04/2021
Búsqueda de los recursos artísticos para el desarrollo de la interfaz	Búsqueda de sprites y sonidos para la renderización del juego en los clientes	1 hora	José Morales	18/04/2021	17/04/2021
Desarrollo de módulos de representación de objetos del lado del servidor	Creación de las clases que permiten representar los objetos del juego en el servidor	1 hora	José Morales	18/04/2021	24/04/2021
Desarrollo de lógica de renderizado en el cliente	Creación de módulos que interactúan con la biblioteca seleccionada para interfaz gráfica para el renderizado de las diferentes imágenes que conforman la pantalla de juego	2 horas	Alejandro Soto	18/04/2021	18/04/2021
Desarrollo de métodos para detección de colisiones	Programación de métodos para la detección y resolución de colisiones del lado del servidor.	30 min	Alejandro Soto	18/04/2021	22/04/2021
Desarrollo de métodos para movimiento de entidades	Programación de métodos que dada una señal de movimiento, cambien las posiciones en la lógica del servidor y envíen los comandos de movimiento al cliente para que los mismo sean interpretados	2 horas	Alejandro Soto José Morales	18/04/2021	18/04/2021
Desarrollo de métodos para interpretación de comandos entre cliente y servidor	Programación de los métodos que manejan el envío de comandos y respuestas entre los clientes y el servidor	2 horas	Alejandro Soto José Morales	18/04/2021	18/04/2021
Desarrollo métodos de lógica de juego	Desarrollo de los métodos que manejan el ciclo de juego en sí	4 horas	Alejandro Soto José Morales	17/04/2021	24/04/2021
Desarrollo de funcionalidad para diferenciar entre tipos de cliente	Funcionalidad en el servidor para discriminar entre clientes jugadores y clientes espectadores	1 hora	Alejandro Soto José Morales	18/04/2021	18/04/2021
Desarrollo de funcionalidad para interpretar comandos del administrador del servidor	Programación de la interfaz de servidor para un administrador que puede agregar o quitar objetos de juego en una partida	1 hora	Alejandro Soto	18/04/2021	18/04/2021
Documentación de clases principales	Escritura de la documentación en estilo de javadoc para las clases utilizadas en la realización de la tarea	1 hora	José Morales	23/04/2021	26/04/2021
Documentación de funcionalidad en C	Escritura de la documentación en estilo de Doxygen para las clases utilizadas en la realización de la tarea	1 hora	José Morales	23/04/2021	24/04/2021
Confección de diagrama de clases final	Tomar los nuevos módulos desarrollados no previstos y cambios en interacción entre clases para poder confeccionar un diagrama que represente el estado final de la solución de la tarea	1 hora	José Morales	26/04/2021	26/04/2021
Confección de Manual de usuario	Escritura del manual que permite a un usuario hacer uso del programa resultante ya sea como jugador, administrador de servido o espectador	30 min	José Morales	26/04/2021	26/04/2021
Arreglos finales, Adapataciones y mejoras en interfaz de usuario	Es posible que algunas secciones necesiten refinamiento al final, este tiempo se considera un tiempo colchón para las tareas relacionadas a este proceso	4 horas	Alejandro Soto	26/04/2021	26/04/2021
Testing y QA	Realizar pruebas del funcionamiento ya fuese para documentación o resolución de posibles errores. Simultáneo a la tarea de arreglos finales, adapatciones y mejoras	4 horas	José Morales	26/04/2021	26/04/2021
Redacción de secciones de estructuras de datos y algoritmos implementados de la documentación externa	Escritura de la documentación externa relacionada a la solución general del algoritmo y las estructuras de datos utilizadas para esta resolución	40 min	José Morales	26/04/2021	26/04/2021
Redacción de secciones de Recomendaciones y conclusiones	Tomando en consideración lo aprendido durante el proyecto y el desarrollo del mismo	40 min	Alejandro Soto José Morales	26/04/2021	26/04/2021
Elaboración de presentación para defensa de código	Elaboración de la presentación a utilizar durante la defensa de la solución de la tarea	20 min	José Morales	26/04/2021	26/04/2021

1.5. Problemas solucionados

1. Pathfinding omite el ubicación inicial
    - *Descripción*:
    - *Intentos de solución*:
    - *Solución encontrada*:
    - *Conclusiones*:
        - x
        - y
    - *Recomendaciones*:
        - x
        - y
    - *Bibliografía*:
        - x
        - y
        - z

1.6. Conclusiones y Recomendaciones del Proyecto

Conclusiones

Recomendaciones

1.8. Bibliografía

America, N. of. (2017). *Donkey Kong jr Instruction Booklet*. https://www.nintendo.co.jp/clv/manuals/en/pdf/CLV-P-NAAFE.pdf.

Gettys, J., Scheifler, R. W., Adams, C., Joloboff, V., Hiura, H., McMahon, B., Newman, R., Tabayoyon, A., Widener, G., & Yamada, S. (2012). *Xlib - C Language X Interface*. https://www.x.org/releases/X11R7.7/doc/libX11/libX11/libX11.html#Obtaining_Window_Information.

Haardt, M., & Brouwer, A. (1996). *Linux Programmer's Manual: perror(3) — Linux manual page*. https://man7.org/linux/man-pages/man3/perror.3.html.

Institute of Electrical and Electronics Engineering Inc., & The Open Group. (2017a). *POSIX Manual: connect(3p) — Linux manual page*. IEEE/The Open Group; https://www.man7.org/linux/man-pages/man3/connect.3p.html.

Institute of Electrical and Electronics Engineering Inc., & The Open Group. (2017b). *POSIX Programmer's Manual: poll(3p) — Linux manual page*. IEEE/The Open Group; https://man7.org/linux/man-pages/man3/poll.3p.html.

Institute of Electrical and Electronics Engineering Inc., & The Open Group. (2017c). *POSIX Programmer's Manual: read(3p) — Linux manual page*. IEEE/The Open Group; https://man7.org/linux/man-pages/man3/read.3p.html.

Kerrisk, M. (2008). *Linux Programmer's Manual: timerfd_create(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/timerfd_create.2.html.

Kerrisk, M., Drepper, U., & Varshavchik, S. (2008). *Linux Programmer's Manual: getaddrinfo(3) — Linux manual page*. https://man7.org/linux/man-pages/man3/getaddrinfo.3.html.

Kleen, A. (1999). *Linux Programmer's Manual: ip(7) — Linux manual page*. https://man7.org/linux/man-pages/man7/ip.7.html.

Koenig, T., & Kerrisk, M. (2008). *Linux Programmer's Manual: getopt(3) — Linux manual page*. https://man7.org/linux/man-pages/man3/getopt.3.html.

Levon, J. (2001). *Linux Programmer's Manual: getline(3) — Linux manual page*. GNU; https://man7.org/linux/man-pages/man3/getdelim.3.html.

Oracle. (2020). *Package javax.swing*. https://docs.oracle.com/javase/7/docs/api/javax/swing/package-summary.html.

University of California, T. R. of the. (1991). *Linux Programmer's Manual: socket(2) — Linux manual page*. https://man7.org/linux/man-pages/man2/socket.2.html.

2. Bitácoras

José Morales

Alejandro Soto