



Área de Ingeniería en Computadores

Lenguajes, Compiladores e Intérpretes

Tarea Programada III

spaCEInvaders

Profesor

Marco Rivera Menénes

Estudiantes

Gabriel Chacón Alfaro - 2021049454 gchacon8@estudiantec.cr

Emanuel Marín Gutiérrez - 2019067500 emarin702740530@estudiantec.cr

Jose Andres Rodriguez Rojas - 2019279722 joseandres216@estudiantec.cr

Fecha

19 de abril, 2023

Contenido

- 1.1 Descripción de las estructuras de datos utilizadas
- 1.2 Algoritmos desarrollados
- 1.3 Patrones de diseño utilizados
- 1.4 Problemas sin solución
- 1.5 Problemas encontrados
- 1.6 Plan de actividades
- 1.7 Conclusiones
- 1.8 Recomendaciones
- 1.9 Bibliografía
- 1.10 Bitácora

1.1 Descripción de las estructuras de datos utilizadas

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
- `length`: Indica cuántos 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

Hashmap

Un hashmap es una estructura de datos en la cual se mapean llaves a ciertos 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.

- `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 del 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: hashmap de sprites del juego
- entities: hashmap de entidades del juego

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 creó 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

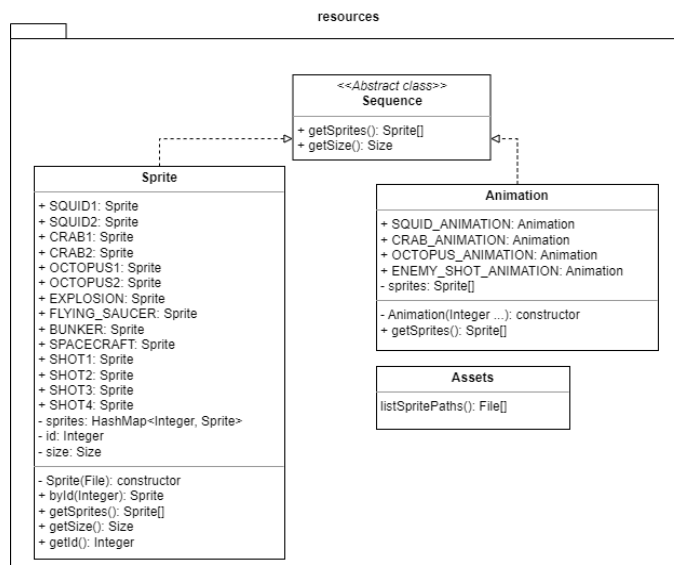
Dentro del cliente se hace uso del formato JSON para comunicación con el servidor. Esto hacía necesario el desarrollo de una estructura relativamente simple que sirviese para la descomposición de objetos JSON. Para esto se agregó un struct key_value el cual contiene los campos:

- 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 de los algoritmos desarrollados

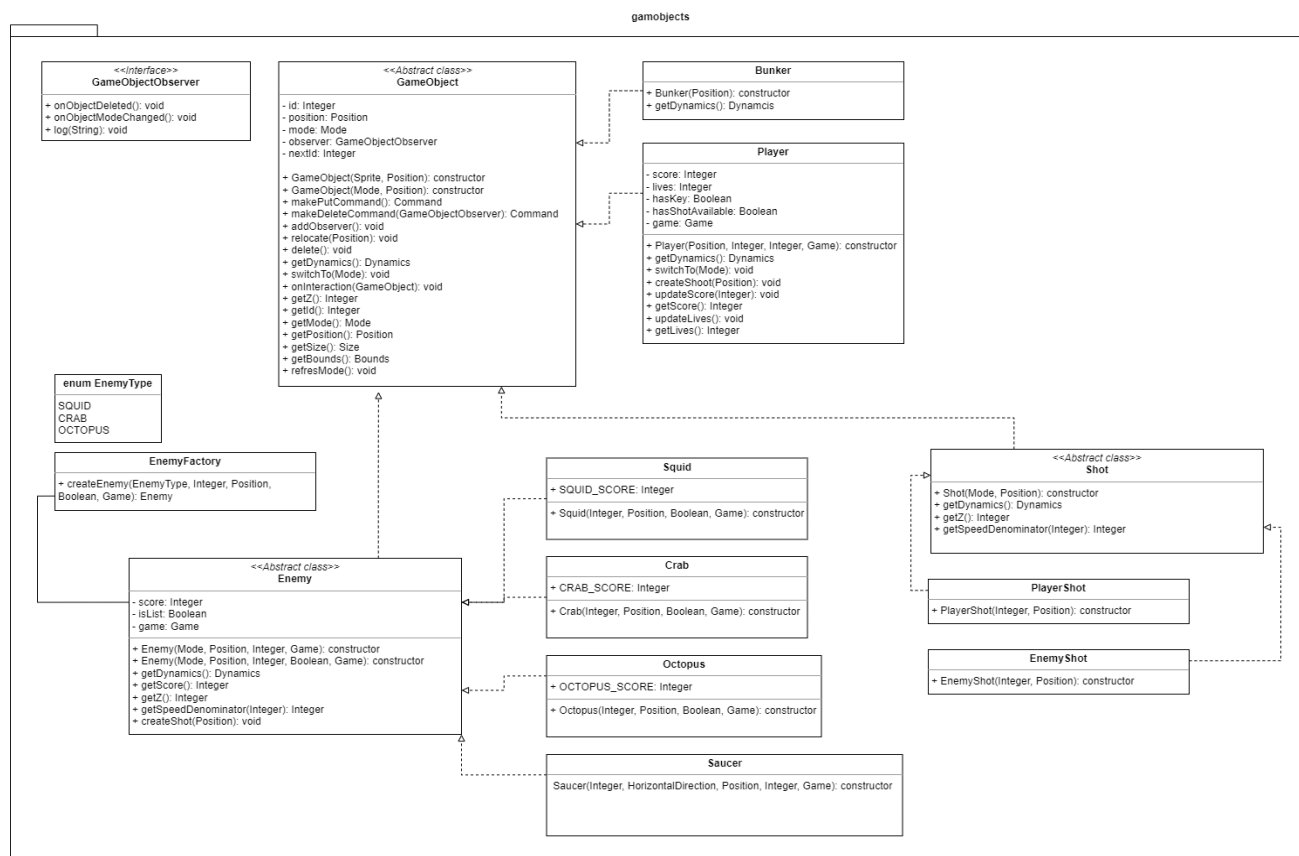
El servidor es un programa en java compuesto por algunas clases base y 6 paquetes que controlan distintos aspectos de la lógica de juego.

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:



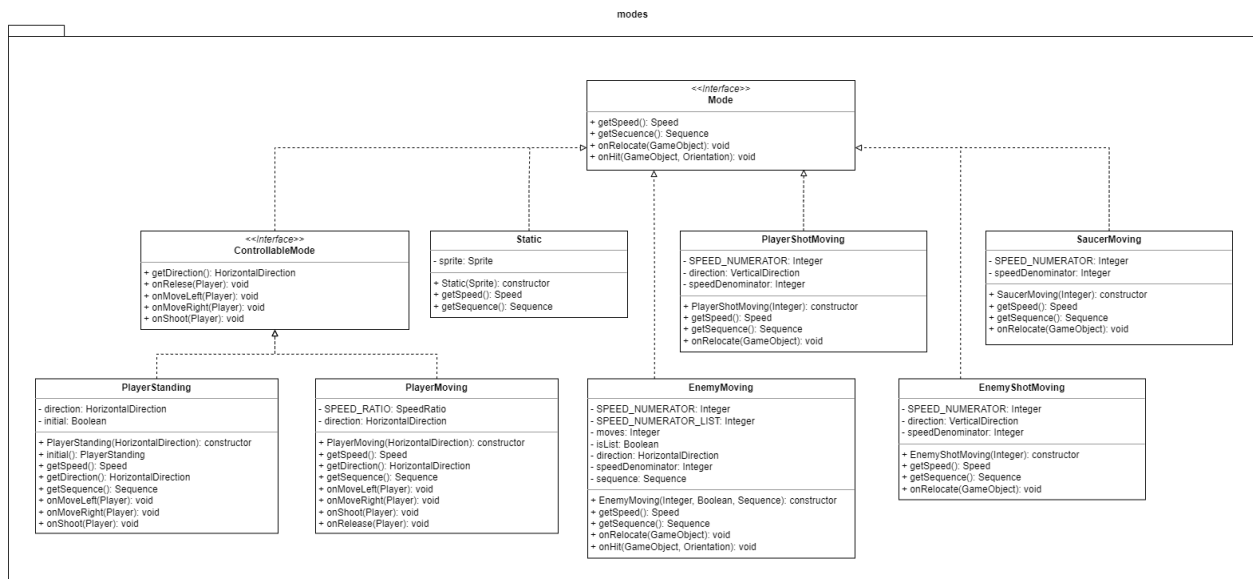
El paquete de gameobjects contiene lo necesario para representar todas las entidades que componen el juego. Para los objetos que pueden ser colocados por el usuario administrador, se implementó un patrón de diseño factory, de manera que la creación de enemigos es relativamente transparente al usuario.

Se implementa además un patrón observer. Este patrón es una forma de permitir 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.

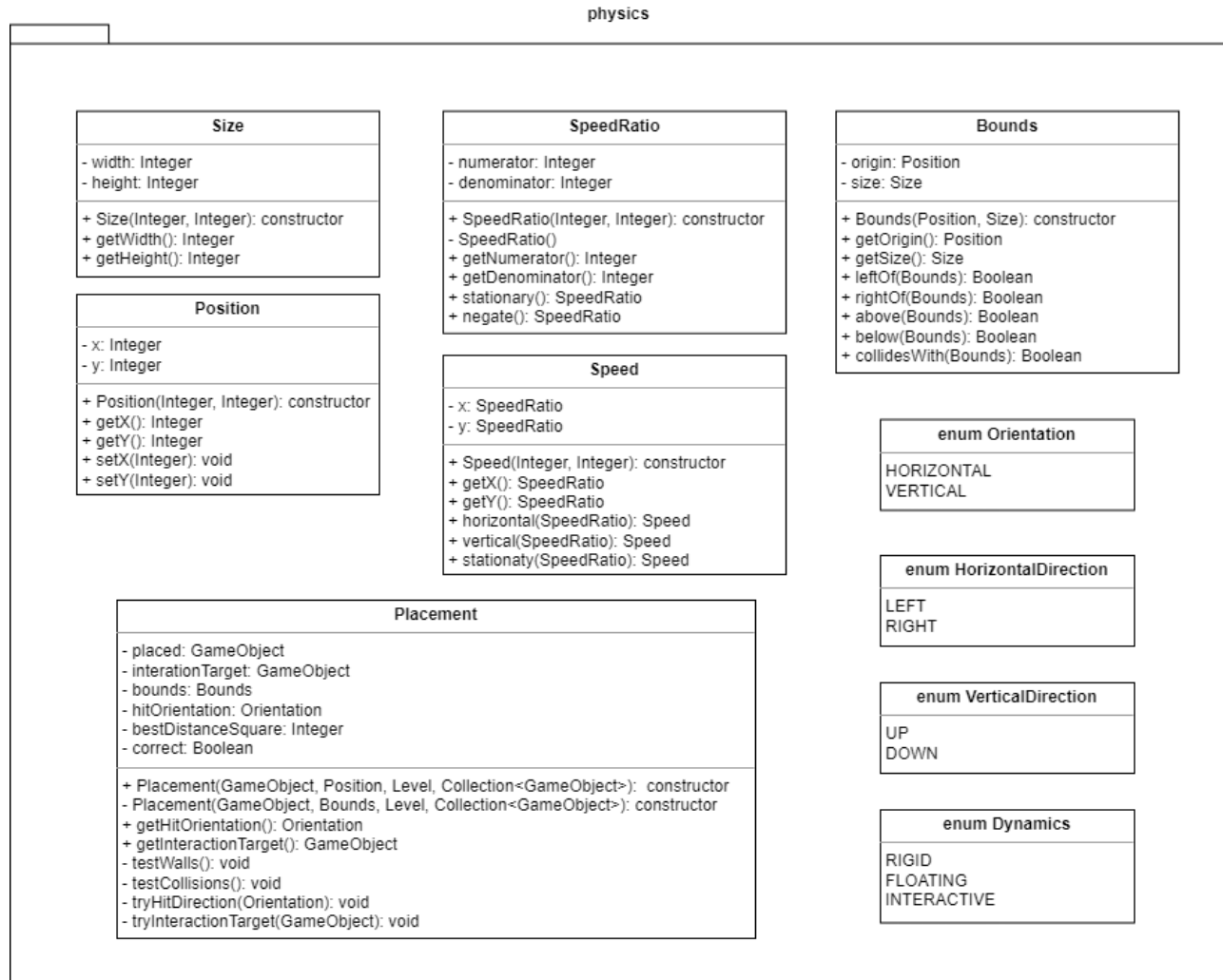
Según el tipo de GameObject se puede tener una observación, ¿Qué sucede con aquellos objetos que tienen estados transitivos, es decir, un objeto que se puede encontrar en varios “modos de operación”?

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 diferenciar entre estados.



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.

Es relevante discutir también la forma en la que el servidor es capaz de siquiera procesar un estado. 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:

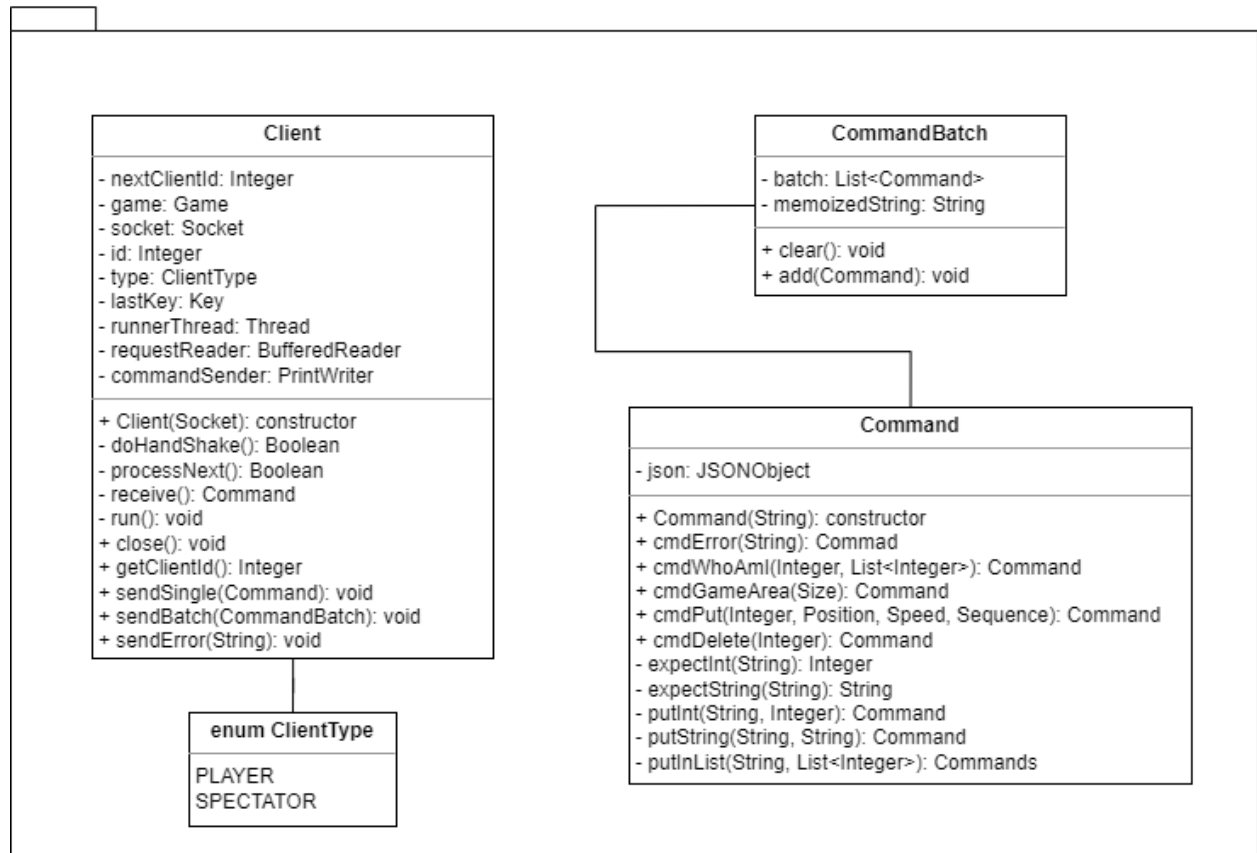


Como puede observarse, el paquete anterior funciona como el administrador de función elemental del juego. Las dos tareas más importantes que administra este paquete son velocidades y colisiones.

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 enemigos 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.

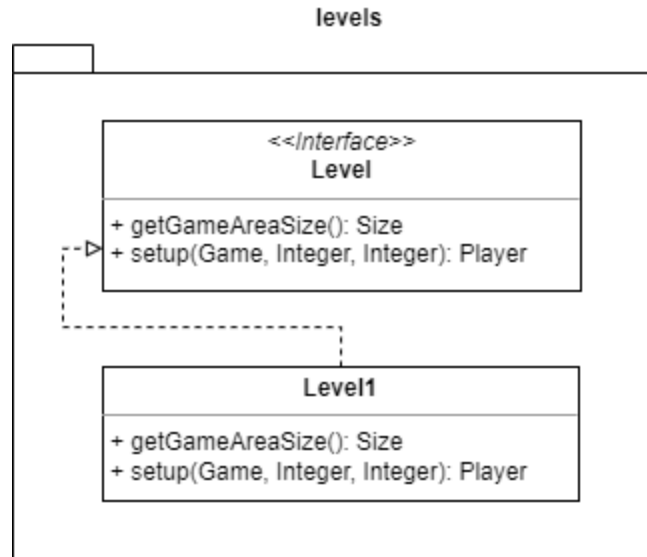
Otra capa fundacional del juego, aunque no relacionada directamente a las entidades del escenario de juego, es el paquete comms.



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, cómo diferenciar entre clientes jugadores y 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.

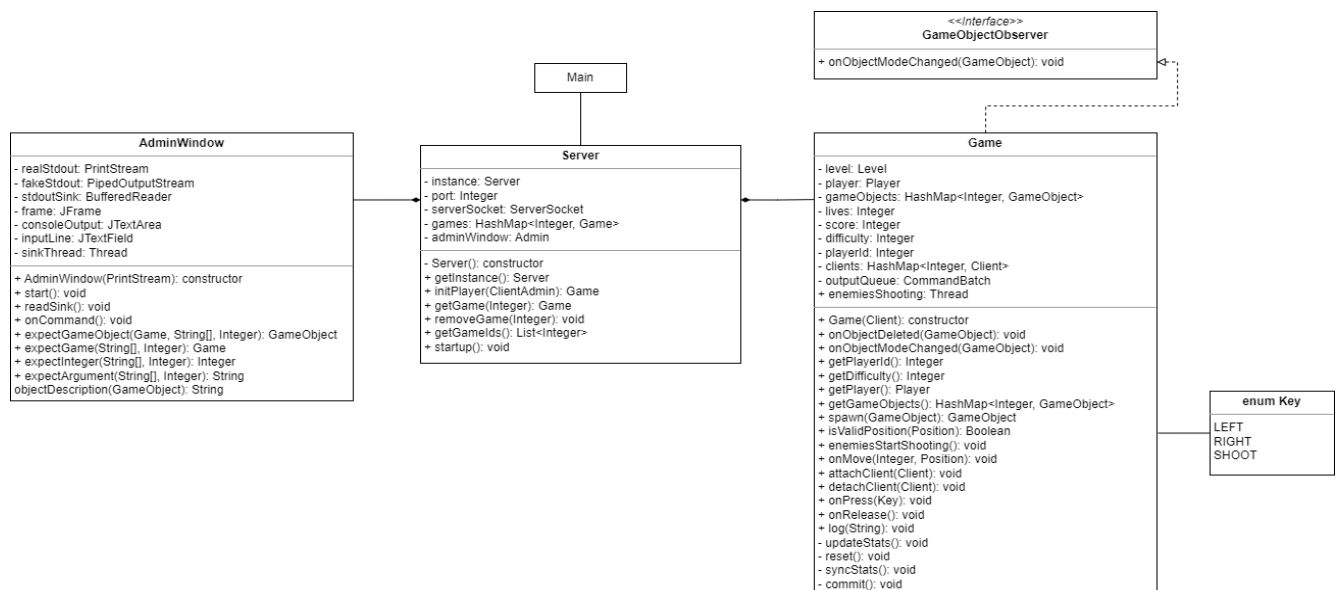
Una vez comprendidos los fundamentos que conforman el juego, se puede proceder a analizar las partes del programa que funcionan sobre estas capas fundacionales.

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



Efectivamente el paquete solo está 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.

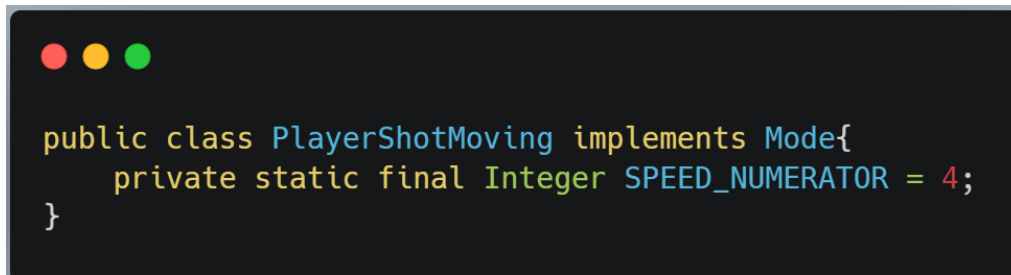
Para evitar posibles conflictos en uso de puertos, entre otros posibles problemas, se recurre a utilizar el patrón de diseño Singleton en la implementación del Servidor. Solo se puede encontrar una instancia de Servidor activa, y no hay forma de crear una instancia adicional que pueda crear problemas, 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 un enemigo, una línea de enemigos y agregar el platillo volador, 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 capas 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.

Movimiento general: Uno de los algoritmos que se desarrollaron durante este proyecto es como se maneja el movimiento general de los distintos “`GameObjects`”

dentro del juego. Primeramente se han establecido dos parámetros principales con los que se logra todo el ciclo, el Speed_Numerator y el Speed_Denominator. El Speed_Numerator hace referencia a la distancia mínima en píxeles que se mueve uno de los objetos cada “tick” o intervalo de tiempo especificado. Por otra parte, el Speed_Denominator hace referencia al intervalo de tiempo pequeño que espera el objeto para volver a moverse o colocarse en otras coordenadas, también llamados “ticks” para efectos del proyecto. De esta forma podemos expresar que un objeto se mueva 10 píxeles cada tick, donde el tick puede ser de 0.25 segundos. En ese caso el Speed_Numerator sería un entero de 10, mientras que el Speed_Denominator no tiene una relación exacta con los segundos del tick, pero entre menor sea este Speed_Denominator más rápidos serán los ticks. Cabe destacar que este Speed_Denominator se ve afectado por otro parámetro llamado difficulty, ya que a mayor dificultad los enemigos y balas se moverán más rápido. Este parámetro difficulty se descartó para su uso en otras partes del juego ya que el requerimiento solicita un único nivel con una misma dificultad. A continuación podemos ver una asignación de un Speed_Numerator para una bala del jugador.

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in Java and defines a class PlayerShotMoving that implements the Mode interface. It includes a private static final Integer variable named SPEED_NUMERATOR with a value of 4.

```
public class PlayerShotMoving implements Mode{  
    private static final Integer SPEED_NUMERATOR = 4;  
}
```

Movimiento de los enemigos: Por otra parte, los enemigos siguen lo propuesto anteriormente respecto al movimiento, pero adicionalmente tiene que tener una lógica para cuando estas colisionan con las paredes de la pantalla. De la misma forma los enemigos tienen que ir bajando en la pantalla y de forma gradual su velocidad en el eje Y va aumentando entre más se acercan a la nave. De esta forma definimos cambiar la dirección cuando choca contra una pared con el método OnHit, de la misma forma se baja al enemigo 10 píxeles hacia abajo y finalmente si rebasa ciertas coordenadas en Y se disminuye el Speed_Denominator, o sea, los ticks del enemigo son más frecuentes.

```

public void onHit(GameObject enemy, Orientation orientation) {
    // If it hits, it changes direction
    this.direction = this.direction.invert();
    this.speedDenominator = 10 - enemy.getPosition().getY()/32;
    moves = 0;
    enemy.relocate(new Position(enemy.getPosition().getX(), enemy.getPosition().getY()+10));
    enemy.switchTo(this);
}

```

1.3 Problemas sin solución

- a) Cuando el usuario pierde la partida (llega a tener 0 vidas), la partida se reinicia, pero si se vuelve a generar uno o varios enemigos, estos poseen un tiempo de disparo menor al previsto.

1.4 Problemas encontrados

Argumentos iniciales: Uno de los principales inconvenientes que se habían encontrado durante el desarrollo del programa fueron los parámetros iniciales con los que se ejecutaba la parte del cliente en C. Esto debido a que se le preguntaba al usuario si deseaba entrar como un cliente jugador o como un cliente espectador y estos parámetros se pasan por la consola del IDE. Por lo que en el .exe no se le iban a poder pasar estos parámetros. Finalmente se pudo utilizar la terminal del mismo Ubuntu, ya que al iniciar un segundo cliente el programa automáticamente pregunta si se desea conectar como espectador o como jugador.

Coordinación de los enemigos: Otro de los principales problemas que se encontraron fue poder coordinar el movimiento de los enemigos ya que los enemigos que se crean de forma individual deben bajar a la misma velocidad en Y que los enemigos que se generan en filas, por lo que se le aumentó la velocidad del enemigo individual en el eje X, de esta forma todos los enemigos en pantalla chocan contra la pared a un ritmo similar por lo que todos bajan a un ritmo muy parecido. Además el hecho de que los enemigos más cercanos a la nave tengan una mayor velocidad ayuda a evitar

colisiones. A continuación podemos ver cómo difieren los Speed_Numerators de los enemigos, donde los que dicen “List” son los que se generan en filas.

```
public class EnemyMoving implements Mode{
    private static final Integer SPEED_NUMERATOR = 7;
    private static final Integer SPEED_NUMERATOR_LIST = 2;
}
```

1.5 Plan de actividades

Descripción de la tarea	Tiempo estimado	Responsable	Fecha de entrega
Servidor/Cliente Pensar y crear estructura base con la cual empezar a trabajar en el servidor y cliente, investigar y agregar las bibliotecas a utilizar para la comunicación mediante JSON.	9 horas	-Emanuel Marín	Jueves 30 de marzo
Servidor Desarrollar lógica de las clases Main, Server y AdminWindow. Cliente Trabajar en util.h, vec.c y hash_map.c	6 horas	-Emanuel Marín	Sábado 1 de abril
Servidor Trabajar en el desarrollo de los paquetes gameobjects y comms, para ello crear las clases GameObjects, Player, Client, Command y CommandBatch y la interfaz GameObjectObserver. Cliente Crear el archivo net.c que se encargue de enviar los mensajes en formato JSON al servidor.	6 horas	-Emanuel Marín	Domingo 2 de abril
Servidor	4 horas	-Emanuel Marín	Lunes 3 de abril

<p>Comenzar a agregar clases dentro del paquete physics. Entre ellas Position y Size (para el tamaño y ubicación de la entidades), Speed y SpeedRatio (para determinar la velocidad horizontal y vertical a la que debe moverse la entidad) y el enum Orientation.</p> <p>Agregar en el paquete modes, la interfaz Mode.</p> <p>Cliente Hacer mejoras en los archivos (main.c, constants.h, space_invaders.h, game.c, util.h, vec.c, hash_map.c y net.c).</p>			
<p>Servidor Agregar la interfaz Level y la clase Level1 al paquete leves. Agregar la clase abstracta Sequence al paquete resources.</p> <p>Cliente Agregar los archivos init.c, loop.c y quit.c para inicializar los sprites del juego, la pantalla del juego y los componentes de SDL2, la conexión con el servidor, renderizar texturas y entidades, un event_loop que controle el bucle general del juego una vez iniciado y manejar el cierre de la ventana del juego</p>	7 horas	-Emanuel Marín	Martes 4 de abril
<p>Servidor Agregar al proyecto la carpeta assets que contenga los sprites que se utilizaran en el juego.</p> <p>Ampliar la lógica de la clase AdminWindow para que pueda procesar más comandos.</p> <p>Agregar más clases en el paquete gameobjects para la representación de entidades en el juego, crear una clase factory para crear diferentes tipos de</p>	6 horas	-Emanuel Marín	Miércoles 5 de abril

<p>enemigos.</p> <p>En la clase Level1, escribir código para hacer spawn de los sprites del Bunker, algunos enemigos y el jugador.</p> <p>Agregar clases en el paquete modes que permitan que los enemigos se muevan en sentido horizontal en la pantalla de juego.</p> <p>Desarrollar más los paquetes physics (enum Dynamics y HorizontalDirection) y resources (clases Sprites, Animation y Assets).</p>			
<p>Renombrar algunos sprites del juego.</p> <p>Servidor Actualizar la clase AdminWindow para que procese más comandos.</p> <p>Dibujar los 4 bunkers del juego a partir de sprites de 4x4 píxeles en la clase Level1.</p> <p>Hacer mejoras en las clases del movimiento de los enemigos (paquete modes).</p> <p>Agregar en el paquete physics el enum VerticalDirection para luego desarrollar los disparos del jugador y los enemigos.</p> <p>Crear la clase abstracta Shot y la clase PlayerShot que herede de esta para empezar a desarrollar los disparos del jugador.</p> <p>Cliente Editar la constante STATS_LABEL_FORMAT que se encarga de pintar el score y las vidas del jugador.</p>	4 horas	-Emanuel Marín	Jueves 6 de abril
Cambiar los colores de algunos	1 hora	-Emanuel Marín	Martes 11 de abril

<p>sprites del juego.</p> <p>Realizar mejoras a todo el código desarrollado hasta el momento, quitar código que no se usa o que esté comentando. Hacer refactor del nombre de algunas clases, propiedades y métodos.</p>			
Limitar la cantidad de disparos del jugador.	1 hora	-Andres Rodriguez	Jueves 13 de abril
<p>Servidor</p> <p>Crear las clases Bounds y Placement en el paquete physics para poder detectar colisiones de entidades.</p> <p>Actualizar las clases GameObject, Player y Game para detectar colisiones del jugador contra los márgenes de la ventana del juego y los disparos del jugador con los bunkers y enemigos. Actualizar el score</p>	3 horas	-Emanuel Marín	Viernes 14 de abril
<p>Servidor</p> <p>Actualizar las colisiones para que ahora también se puedan detectar las colisiones entre disparos (jugador y enemigo) y disparo del enemigo contra el jugador. Actualizar vidas y reset del juego.</p>	1 hora	-Emanuel Marín	Lunes 17 de abril
Agregar colisión entre enemigo y jugador	1 hora	-Emanuel Marín	Viernes 21 de abril
Implementar la funcionalidad del disparo por parte de los enemigos.	6 horas	-Andres Rodriguez	Domingo 16 de abril
Corregir las rutas referentes a los archivos multimedia del juego (tanto en el servidor como en el cliente).	1 hora	-Andres Rodriguez	Lunes 17 de abril
Corrección de problemas para la generación de ejecutables.	1 hora	-Andres Rodriguez	Martes 18 de abril
Elaboración del manual de usuario.	3 horas	-Andres Rodriguez	Jueves 20 de abril

Generación de ejecutables (tanto para el cliente como para el servidor).	2 horas	-Andres Rodriguez -Gabriel Chacon	Viernes 21 de abril
Se investigan los diferentes componentes que se necesitaran programar a futuro según los roles asignados	2 horas	-Gabriel Chacon	Lunes 3 de abril
Se empieza el desarrollo del movimiento enemigo en la parte del servidor haciendo que bajen hacia el jugador con velocidad ascendente.	3 horas	-Gabriel Chacon	Martes 11 de abril
Se entregan los resultados del movimiento generado y se discuten posibles mejoras al respecto.	2 horas	-Gabriel Chacon	Sabado 15 de abril
Se corrigen y se añaden las diferentes mejoras y correcciones sobre el movimiento.	2 horas	-Gabriel Chacon	Domingo 16 de abril
Se optimizan las clases según lo establecido en la parte del servidor hasta el momento manteniendo la orientación a objetos.	2 horas	-Gabriel Chacon	Lunes 17 de abril
Se realizan los ejecutables de la parte del cliente y del servidor añadiendoles en las carpetas del programa.	1 hora	-Gabriel Chacon	Viernes 21 de abril

1.6 Conclusiones

- Utilizando POO se pueden utilizar clases con distintos atributos y métodos que nos ayudan a modularizar las tareas y el conocimiento haciendo que sea un paradigma ordenado que permite encapsular información y maniobrar datos con facilidad.
- C no es un lenguaje que se caracterice por tener una gran facilidad para formar interfaces gráficas o esta aplicación no es tan popular, por lo que formar una

interfaz de un juego agradable al usuario podría ser mejor en otros lenguajes con características más amigables.

- Los sockets nos permiten hacer una comunicación eficiente entre los distintos lenguajes que los incorporan. En este caso nos permite tener un servidor en Java y un cliente en C para poder realizar las distintas operaciones y procesamiento en el lugar más adecuado.
- Los patrones de diseño nos facilitan soluciones previamente meditadas a problemas comunes para generar una buena comunicación entre clases o funciones dependiendo del caso del programa.
- SDL2 se mostró como una biblioteca de gráficos lo suficientemente capaz como para el desarrollo completo del cliente sin necesidad de recurrir a otra biblioteca de apoyo para tareas gráficas. Esta herramienta es simple, pero increíblemente útil.
- El formato JSON es sustancialmente útil y preferible para la comunicación entre diferentes agentes en una interacción por medio de internet. El hecho de que este formato sea relativamente simple y estandarizado lo hace fácil de parsear y utilizar, razón principal por la cual se escogió este formato para el desarrollo de la tarea.
- Las bibliotecas json-c y json-simple son utilidades simples pero efectivas para el procesamiento y uso de cadenas de caracteres formateadas según un formato JSON.
- La implementación de un patrón Singleton permite evitar problemas que pueden surgir al tener varias instancias de una clase cuyo funcionamiento está diseñado alrededor de ser una clase de instancia única. Es decir, aplicar el patrón Singleton es una buena práctica para declarar de manera explícita que una clase sólo debería tener una instancia en todo un programa.
- El patrón de diseño observer es de gran utilidad para lograr un efecto de autoadministración del código, ya que en vez de tener que diseñar rutinas que se vean obligadas a buscar cambios de estado de cada objeto que es campo de una clase, el patrón observer es capaz de notificar al observador el origen único del cambio, lo cual resulta en economización de recursos.

- El patrón de diseño Factory simplifica la creación de objetos y hace el proceso detrás de la misma transparente al usuario, lo que se podría considerar según el contexto como una ventaja deseable en un subsistema.

1.7 Recomendaciones

- Se recomienda utilizar una estructura distinta ya que asignar una interfaz en el lenguaje C complica mucho la calidad y facilidad con la que se realiza esa interfaz. En dado caso C++ proporciona mejores opciones de interfaz gráfica.
- Se recomienda utilizar la biblioteca de SDL2 para desarrollo de videojuegos o aplicaciones gráficas similares en el lenguaje de programación C. Si bien la curva de aprendizaje puede ser algo pronunciada, la utilidad de la biblioteca hace que valga la pena dedicarle el tiempo necesario para aprender a utilizarla de una forma apropiada.
- Si se quiere establecer un protocolo de comunicación entre un cliente y un servidor simple, sencillo y fácil de utilizar, JSON es un formato recomendado para esta tarea. Una comunicación relativamente compleja es fácil de expresar por medio de JSON.
- Para procesamiento de mensajes en formato JSON: si se trabaja en el lenguaje de programación Java, se recomienda utilizar la biblioteca json-simple. En caso de estar trabajando en el lenguaje de programación C, se recomienda utilizar la biblioteca de json-c.
- Aplicar el patrón de diseño Singleton cuando se considere que una clase debería tener una única instancia durante la ejecución de un programa.
- Aplicar el patrón de diseño Observer cuando se vea que un objeto necesita reaccionar al cambio de estado de algún otro objeto, y buscar formas alternativas de informar al primero sobre el cambio del segundo sea impráctico o desperdicie recursos.
- Utilizar el patrón Factory cuando se requiera esconder detalles de implementación sobre la creación de una colección de objetos, pero igual se le quiere habilitar al usuario una forma simple de crear dichos objetos.

1.8 Bibliografía

Oracle. (n.d.). Java documentation. Oracle. Retrieved April 20, 2023, from <https://docs.oracle.com/en/java/>

Baeldung. (n.d.). A guide to Java sockets. Baeldung. Retrieved April 20, 2023, from <https://www.baeldung.com/a-guide-to-java-sockets#:~:text=By%20definition%2C%20a%20socket%20is,destined%20to%20be%20sent%20to.>

GeeksforGeeks. (n.d.). Socket programming in C/C++. GeeksforGeeks. Retrieved April 20, 2023, from <https://www.geeksforgeeks.org/socket-programming-cc/>

RedesPlus. (2019, October 15). Curso de redes - Capítulo 1: Introducción a redes [Video]. YouTube. https://www.youtube.com/watch?v=hLfVO0GwheU&ab_channel=RedesPlus

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>.

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>.

1.10 Bitácora

Fecha	Presentes	Duración	Avance
03/04/2023	-Andrés Rodríguez -Emanuel Marín -Gabriel Chacón	2 horas	Se llevó a cabo una reunión inicial en la que se repasó la especificación de la tarea y se asignaron roles y tareas iniciales.
11/04.2023	-Andrés Rodríguez -Emanuel Marín -Gabriel Chacón	2 horas	Se hizo una segunda reunión estableciendo los siguientes objetivos del grupo y los roles de los participantes.
03/30/2023	-Emanuel Marín	9 horas	Servidor Empecé a crear las clases bases y paquetes que se podrían llegar a utilizar durante el desarrollo del proyecto, entre ellos están la clase Main, Server, Game y AdminWindow. En lo que respecta a paquetes se crearon el

			<p>paquete comms (comunicación con el cliente), gameobjects (creación de entidades del juego), levels (niveles del juego, en este caso solo tendrá el nivel 1 que es el nivel base), modes (modos de operación de la entidades), physics (físicas de las entidades), resources (donde se almacenarán los sprites).</p> <p>También se decidió que los sockets se comuniquen mediante JSON, para ello se investigó qué bibliotecas usar y al final elegí json-simple-1.1.1.jar y json-c</p> <p>Cliente Trabajé en el main.c, la definición de algunas constantes que se llegaran a utilizar en constants.h y desarrollé la lógica inicial del juego en space_invaders.h con game.c</p>
01/04/2023	-Emanuel Marín	6 horas	<p>Servidor Desarrollé la lógica de las clases Main, Server, AdminWindow (consola de administración), Game y agregue el enum Key</p> <p>Cliente Trabajé en util.h donde se define un arreglo dinámico llamado vec (en vec.c) y la implementación de un hash_map (en hash_map.c)</p>
02/04/2023	-Emanuel Marín	6 horas	<p>Servidor Trabajé en el desarrollo del paquete gameobjects creando la clase abstracta GameObject, la interfaz GameObjectObserver y la clase Player que hereda de GameObject (el jugador como entidad del juego)</p> <p>Además de eso, trabajé en el paquete comms para la comunicación con el cliente, desarrollando las clases Client, Command y CommandBatch para administrar la creación de un cliente y la comunicación por medio de mensajes en formato JSON entre el cliente y el servidor.</p> <p>Cliente Creé net.c que se encarga de enviar mensajes en forma de objeto JSON al servidor</p>
03/04/2023	-Emanuel Marín	4 horas	Servidor

			<p>El día de hoy comencé a agregar clases dentro del paquete physics, entre ellas están Position y Size para el tamaño y ubicación de las entidades, Speed y SpeedRatio para determinar la velocidad horizontal y vertical a la que debe moverse la entidad según el enum Orientation</p> <p>Al paquete modes de momento solo le agregué la interfaz Mode.</p> <p>Cliente Hice mejoras a los archivos que he venido trabajando (main.c, constants.h, space_invaders.h, game.c, util.h, vec.c, hash_map.c y net.c)</p>
04/04/2023	-Emanuel Marín	7 horas	<p>Servidor Trabajé en el paquete levels creando una interfaz Level y un Level1 que la implementa, además de eso, al paquete resources le agregue la clase abstracta Sequence.</p> <p>Cliente Agregué los archivos init.c, loop.c y quit.c que se encargan de inicializar los sprites del juego, la pantalla del juego y los componentes de SDL2, inicia un reloj para realizar un seguimiento del tiempo transcurrido, inicializar la conexión con el servidor, renderizar texturas y entidades, un event_loop que controla el bucle general del juego una vez iniciado y manejar el cierre de la ventana del juego.</p> <p>Con todo lo desarrollado hasta el momento, se tiene una base para seguir trabajando y ampliando la lógica principalmente del servidor.</p>
05/04/2023	-Emanuel Marín	6 horas	<p>Servidor Se agregó al proyecto una carpeta llamada assets que contiene los sprites que se utilizarán en el juego y archivo arcade_n.ttf para dar estilo retro o arcade a los labels en la ventana del cliente.</p> <p>Se amplió un poco la lógica de la clase AdminWindow para que ahora pueda procesar los comandos clear y game-list, lo mismo con la clase Game.</p> <p>Al paquete gameobjects, se le agregó la clase</p>

			<p>Bunker y se extendió la lógica de la clase Player. Se definió también la clase abstracta Enemy que igual hereda de GameObject, para con ello poder crear las clases Squid, Crab, Octopus y Saucer. Junto con esto se creó la clase EnemyFactory y el enum EnemyType para más adelante crear los enemigos del juego de forma más sencilla.</p> <p>Con esto en la clase Level1 se pudo agregar la lógica para hacer spawn de los sprites Bunker, algunos enemigos y el jugador.</p> <p>En el paquete modes se agregó la clase Static que hereda de Mode y la interfaz ControllableMode que también hereda de Mode, con ello se pudieron definir la clase PlayerMoving y PlayerStandind, SquidMoving, CrabMoving, OctopusMoving y SaucerMoving.</p> <p>En el paquete physics se agregó el enum Dynamics que indica el tipo de interacción que un objeto puede tener (RIGID, FLOATING, INTERACTIVE). También se agregó el enum HorizontalDirection para indicar si un objeto se mueve hacia la izquierda o hacia la derecha.</p> <p>En el paquete resources se agregaron las clases Sprites y Animation (una secuencia de varios sprites en bucle) que heredan de la clase Sequence, también se creó la clase Assets que se encarga de obtener todas las rutas de archivos de imágenes de sprites en la carpeta assets.</p>
06/04/2023	-Emanuel Marín	4 horas	<p>Hoy empecé renombrando algunos sprites del juego.</p> <p>Servidor Edité la clase AdminWindow para que ahora pueda procesar los comandos help (mostrar comandos de ayuda), clear (borrar lo escrito en la ventana de comandos), game-list (muestra los juegos que se están ejecutando), object-list (mostrar todas las entidades en un juego en ejecución), add-enemy (agrega un enemigo), add-enemy-line (agrega una línea de enemigos) y add-saucer (agrega un platillo volador).</p> <p>En la clase Level1 del paquete levels agregué la</p>

			<p>lógica para dibujar los 4 bunkers del juego a partir de sprites de 4x4 píxeles (esto ayudará la parte de colisiones para representar la destrucción de los bunkers a mano de los disparos del jugador y los enemigos).</p> <p>En el paquete modes hice mejoras en las clases SquidMoving, CrabMoving, OctopusMoving y SaucerMoving para que en el juego ya se puedan mover de izquierda a derecha y viceversa.</p> <p>En el paquete physics solo agregué el enum VerticalDirection que se utilizará para los disparos del jugador y los enemigos.</p> <p>En el paquete gameobjects creé la clase Shot que hereda de la clase GameObject, para que pueda ser utilizada por una nueva clase llamada PlayerShot la cual necesita que se cree (en el paquete modes) la clase PlayerShotMoving para que el jugador pueda realizar disparos en el juego.</p> <p>Cliente En el archivo constants.h edite el STATS_LABEL_FORMAT que es el label en la pantalla del cliente que pinta el score y las vidas del jugador.</p>
11/04/2023	-Emanuel Marín	1 hora	<p>Antes de la reunión grupal, le cambié los colores a algunos sprites del juego</p> <p>Servidor Hice pequeños cambios en algunas clases, principalmente eliminando código comentado y cambiándole el nombre a algunas clases y métodos para que sean más representativos acorde al juego.</p>
14/04/2023	-Emanuel Marín	3 horas	<p>Servidor Para detectar algunas colisiones tuve que agregar en el paquete physics las clases Bounds (cuadro de tamaño de posición que delimita una entidad) y Placement (Crea un objeto que proporciona información sobre la posición de una entidad en el campo de juego).</p> <p>Luego tuve que editar las clases GameObject y Player del paquete gameobjects y la clase</p>

			Game para poder detectar una colisión, por ahora las colisiones que se detectan son si el jugador quiere salir de los márgenes de la ventana del juego y si el disparo del jugador impacta sobre un enemigo o platillo volador, lo que actualiza el score.
14/04/2023	-Andres Rodriguez	2 horas	Limite los disparos del jugador, agregando un tiempo de "recarga" entre disparos, de modo que el jugador solo pueda disparar una vez cada 2 segundos.
17/04/2023	-Emanuel Marín	1 hora	Actualicé las colisiones que se pueden detectar, en este caso son si el disparo del jugador y algún enemigo impactan, en este caso se eliminan ambos disparos. Si el disparo de algún enemigo impacta con el jugador se resta una vida. Si se llega a impactar al jugador tres veces, el juego se reinicia de acuerdo con Level1.
21/04/2023	-Emanuel Marín	1 hora	Agregué colisión entre enemigo y jugador.
16/04/2023	-Andres Rodriguez	6 horas	Codifique las clases y funciones requeridas para habilitar la funcionalidad de disparo por parte de los enemigos, todos los enemigos pueden disparar, lo hacen uno a la vez y tienen un tiempo entre disparos entre 1 y 2 segundos (asociado a la cantidad de enemigos en pantalla).
17/04/2023	-Andres Rodriguez	2 horas	Modifique las rutas de los archivos multimedia del proyecto (fuentes, imágenes, etc.) utilizando rutas relativas, de modo que el código fuera ejecutable en cualquier dispositivo.
18/04/2023	-Andres Rodriguez	1 hora	Solucione el problema asociado con los parámetros de inicio, de modo que se pudieran retirar de IntelliJ y estuvieran implícitos en el código, de esta forma los ejecutables podrían ser generados y funcionarían correctamente.
19/04/2023	-Andres Rodriguez	1 hora	Investigue sobre la forma de generar los archivos ejecutables, tanto para Java (utilizando IntelliJ), como para C (mediante Clion).
20/04/2023	-Andres Rodriguez	3 horas	Elabore una versión inicial del manual de usuario.
21/04/2023	-Andres Rodriguez -Gabriel Chacon	2 horas	Se generaron los ejecutables de ambos programas (servidor y cliente), de modo que se

			<p>pudiera abrir cada programa sin necesidad de utilizar un IDE.</p>
21/04/2023	-Andres Rodriguez	1 hora	<p>Finalice el manual de usuario, incluyendo las rutas de los ejecutables, la forma de ejecutarlos en Ubuntu y realice algunas modificaciones sobre la versión inicial de dicho manual.</p>
11/04/2023	-Gabriel Chacon	3 horas	<p>Se analizó el código realizado anteriormente por el compañero para poder hacer que los enemigos pudieran moverse de forma coordinada y bajar. Solamente se logró la lógica para que bajaran.</p>
15/04/2023	-Gabriel Chacon	2 horas	<p>Se siguió con la solución anteriormente empezada, logrando que los enemigos que se generan en fila y los que se generan de forma individual puedan moverse sin colisionar debido al movimiento.</p>
17/04/2023	-Gabriel Chacon	1 hora	<p>Posteriormente a las soluciones creadas se notaron 3 clases que tenían código muy repetitivo, por lo que se generalizó a una sola clase llamada EnemyMoving el cual tiene un nuevo parámetro que valida el tipo de enemigo que se crea.</p>
20/04/2023	-Gabriel Chacon	2 horas	<p>Se empezó el desarrollo de la documentación técnica desarrollando puntos como los algoritmos implementados y los patrones de diseño que se utilizaron.</p>