Descripción de arquitectura y post-mortem grupal

Este documento detalla el proceso de construcción de una réplica del juego Galaxian (renombrado como Invaders) usando Pygame para el curso Introducción al desarrollo de Videojuegos. Incluye un resumen del proceso de desarrollo, la evolución semanal de este proceso, análisis arquitectónico, patrones usados y una reflexión final.

El enlace al repositorio con el código fuente es el siguiente:
https://github.com/Laurarestrepo03/Ocarina-Galaxian

Resumen del proceso de desarrollo

Para llevar a cabo el desarrollo del proyecto, se tomó la decisión de definir las tareas a realizar según la rúbrica definida. Una vez se establecieron estas tareas, se analizó cuáles debían terminarse primero en términos de dependencias (ejemplo: la colisión entre una bala y el jugador solo puede suceder si el jugador existe). En vista de esto, se repartieron las tareas entre los miembros del grupo de manera equitativa en términos de dificultad, priorizando las obligatorias. Siendo así, el desarrollo como tal involucró la creación de ramas por tarea y, posteriormente, la creación de un pull request para ser revisado por los demás miembros que, una vez haya sido aprobado, se integraba a la rama principal. Tanto los issues (tareas) como los pull requests contaron con una estructura de nombramiento y etiquetación definida para tener un repositorio más organizado. Por último, es importante mencionar que el estado de las tareas fue actualizado en un tablero kanban para informar a los demás miembros de su evolución.

Evolución semanal

Como se mencionó en la sección anterior, el trabajo fue repartido por tareas. Para facilitar, entonces, la explicación de la evolución semanal, se mostrará los issues terminados por semana.

Semana 5

La semana 5 fue dedicada, en su mayoría, a la implementación de features, por lo que las tareas terminadas fueron las siguientes:

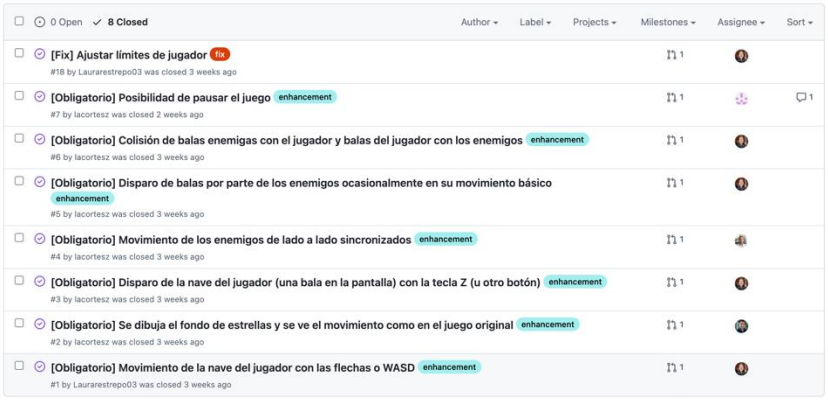

Figura 1. Tareas terminadas en la semana 5

Sin contar la tarea dedicada a publicar el juego, este avance corresponde al 58% de la versión obligatoria juego, lo que representa un porcentaje bastante alto para la primera semana de desarrollo, por lo que se tomó la decisión de enviar un documento de avance a los profesores para recibir retroalimentación.

Semana 6

Una vez obtenida la retroalimentación, se identificaron varias oportunidades de mejora y corrección, por lo que esta semana fue dedicada, en su mayoría, a arreglos:

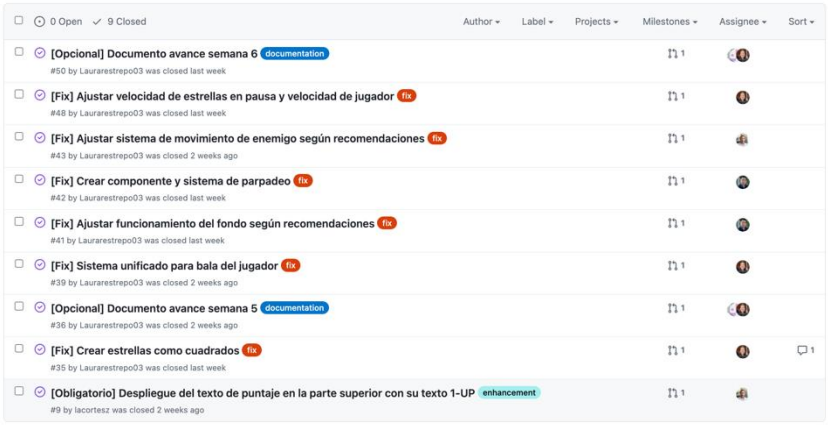

Figura 2. Tareas terminadas semana 6

Como solo se avanzó en una tarea de feature, el avance de esta semana corresponde al 66% del juego. Sin embargo, al haber una cantidad importante de cambios, al igual que la semana anterior, se tomó la decisión de enviar un documento de avance a los profesores para recibir retroalimentación.

Semana 7

Una vez obtenida la retroalimentación, se identificaron oportunidades de mejora y corrección, pero en menor cantidad a comparación de la semana anterior. Siendo así, esta semana fue dedicada a una distribución equitativa de arreglos e implementación de features:

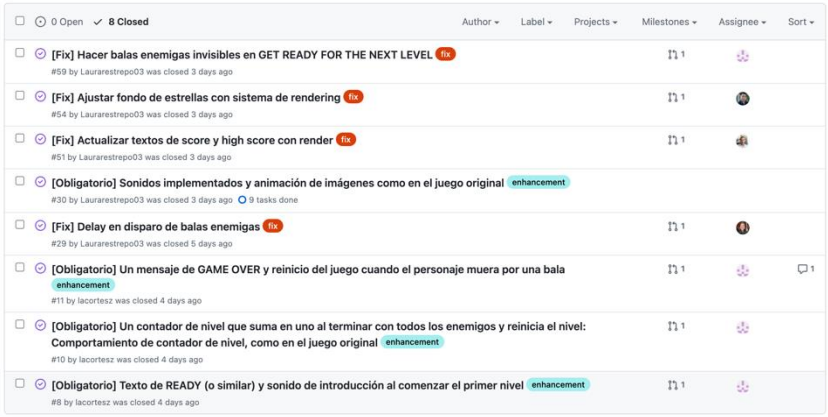

Figura 3. Tareas terminadas semana 7

Con doce tareas obligatorias terminadas, finalmente se llegó a un avance del 100% del juego. Vale la pena mencionar que la tarea #30 era un issue global que contaba con varias subtareas repartidas en otros issues, razón por la cual solo se terminó hasta esta semana.

Otro punto a destacar es el hecho de que algunas tareas obligatorias también contaron features opcionales (bono), de las cuales se encuentra el ataque dinámico por parte de los enemigos, manejo de niveles, sistema de vidas y sistema puntaje máximo conservado al finalizar la sesión (esto solo funciona si se juega localmente, no en Itchi.io).

Análisis arquitectónico

Dando paso al código, el proyecto fue desarrollado haciendo uso del patrón de arquitectura ECS. Teniendo eso en cuenta, se procede a explicar el trabajo en términos de entidades, componentes, demás clases, y archivos de configuración.

Entidades

Por el lado de las entidades, es importante considerar que no todas existen al mismo tiempo en el mundo, por ejemplo, algunas se crean dadas ciertas condiciones. Teniendo esto, a manera general, estas son las que existen en algún punto del juego:

- Bullets: Corresponde a las balas de tanto los enemigos como del jugador.
- Enemy bullet spawner: Corresponde a una entidad de ayuda creada para asociarla a un componente de manejo de balas enemigas (detalles adelante), por lo cual no se puede observar en pantalla.
- Enemy: Corresponde a los enemigos que se observan en la parte superior de la pantalla. Todos, sin importar si tienen sprites diferentes, están asociados a una entidad de enemigo.
- Explosion: Corresponde a las entidades de explosión que aparecen cuando una bala colisiona con un enemigo o con el jugador. Su duración en pantalla se limita a la duración de animación.
- Flag: Corresponde a la bandera que se observa en la esquina de la pantalla, más que todo una entidad visual.
- Game manager: Corresponde a una entidad de ayuda creada para asociarla a un componente de manejo de escenas en el juego (detalles adelante), por lo cual no se puede observar en pantalla.
- Inputs: Corresponde a las entidades de los inputs. Para cada input distinto que puede hacer el usuario se crea una entidad.
- Level: Corresponde a las entidades de nivel que existen al momento de iniciar el juego o aumentar de nivel. Es una entidad de ayuda creada para asociarla a un componente de nivel, por lo cual no se puede observar en pantalla.
- Player: Corresponde al jugador. Existe solamente uno en el juego.
- Star: Corresponde a las estrellas que se observan en el fondo de la pantalla.
- Text: Corresponde a los textos que se observan en pantalla, tanto los permanentes (los de puntaje y nivel), como los temporales (READY, GAME OVER, etc.)
- Life: Corresponde a los indicadores de la cantidad de vida del jugador, representados mediante sprites en la parte superior de la pantalla.
- Score: Corresponde al acumulado de los puntos obtenidos por eliminar un enemigo
- High Score: Corresponde al máximo valor de puntos obtenidos de manera histórica

Componentes

Por el lado de los componentes, vale la pena recordar que estos son clases con atributos atados a entidades. Siendo así, estos son los que se crearon para el juego:

- CTagBullet: Para diferenciar las entidades de balas. Tiene un atributo que indica si la bala pertenece a un enemigo o al jugador.
- CTagEnemy: Para diferenciar las entidades de los enemigos. Tiene un atributo que indica su tipo ( ,  ,  , o ). Cuántas balas ha disparado, en qué estado de disparo se encuentra, y un temporizador que ayuda a controlar el intervalo entre las balas disparadas. Estos tres últimos atributos se tienen en cuenta para el movimiento básico, no cuando están persiguiendo al jugador.
- CTagHighScore: Para diferenciar la entidad correspondiente al texto del valor de High Score.
- CTagLevel: Identifica la entidad del texto que indica el nivel actual del jugador.
- CTagLife: Identifica la entidad que representa las vidas del usuario, permite eliminar las vidas de la pantalla.
- CTagPause: Para diferenciar la entidad del texto pause al momento de ser eliminado.
- CTagPlayer: Para diferenciar la entidad del jugador. Tiene dos atributos que indican cuántas teclas de la derecha y la izquierda se están presionando, lo cual es útil para que no se aumente la velocidad n veces por n teclas presionadas.
- CTagReady: Identifica los textos que serán eliminados en algún momento de la ejecución del juego, por ejemplo, el Game Over, o el Ready al inicio de la ejecución.
- CTagScore: Para diferenciar la entidad correspondiente al texto del valor score.
- CTagStar: Para diferenciar la entidad correspondiente a las estrellas.
- CAnimation: Gestiona la información relacionada a la animación de una entidad, que incluye su número de frames, animación actual, tiempo de la animación actual, frame de la animación actual, y una lista con el nombre, inicio, fin y framerate de todas las animaciones que tiene.
- CBlink: Gestiona el parpadeo para todas las funciones, componentes o sistemas que lo puedan requerir.
- CEnemyAttack: Se utiliza para guardar el tiempo actual en el intervalo de tiempo para generar el ataque de un enemigo
- CEnemyBulletSpawnerHunting: Gestiona la información del disparo de un enemigo cuando se encuentra persiguiendo al jugador. Sus atributos incluyen un contador y un tiempo máximo para controlar el intervalo de tiempo entre cada bala.
- CEnemyBulletSpawner: Gestiona la información del disparo de los enemigos en su movimiento básico. Este componente está atado solamente a una entidad, porque se usa para controlar el disparo de manera global. Sus atributos incluyen un contador y un tiempo máximo aleatorio que determinan el intervalo de tiempo entre los disparos de los enemigos. El tiempo máximo se define haciendo uso de un método de clase.
- CEnemySpawner: Se utiliza para contener la información extraída del archivo json de level de los enemigos que se debe crear al comenzar el nivel
- CEnemySteering: Se utiliza para guardar la información relacionada con el desplazamiento de ataque del enemigo, como la entidad del enemigo, la posición de retorno, el estado y el contador del movimiento JUMPING. También contiene una enumeracion de los estados de ataque, JUMPING, HUNTING y RETURNING.

- CExplosionState: Para gestionar el estado de una explosión. Tiene un atributo que indica qué estado se encuentra, pero solo hay uno: EXPLODE.
- CGameState: Gestiona el estado del juego. Permite almacenar el estado y las variables de estado (current_time, time_dead, number_enemies, etc)
- CInputCommand: Para gestionar la información de un input, como las llaves que usa (sea una o varias), su nombre, y el estado en el que se encuentra (NA, START, END).
- CLevel: Gestiona la información del nivel en general como los scores y el movimiento de la armada
- CPlayerBulletState: Gestiona el estado de la bala del jugador, es decir, si no se ha disparado (NOT_FIRED) o si se encuentra en el aire (FIRED).
- CSurface: Gestiona la información de superficie de una entidad. Cuenta con varios métodos de clase para crear superficies a partir de otras superficies, crear superficies a partir de textos, y obtener el área relativa.
- CTransform: Gestiona la información de posición de una entidad.
- CVelocity: Gestiona la información de velocidad de una entidad.

Sistemas

De igual manera, se crearon ciertos sistemas para asegurar el funcionamiento del juego, explicados a continuación:

- system_animation: Regula la animación de una entidad con el componente CAnimation, a partir del frame actual y su framerate.
- system_blink: Permite controlar el uso del parpadeo en funciones, componentes o sistemas que lo puedan requerir.
- system_bullet_limit: Permite controlar el límite al cual puede llegar una bala con respecto a la pantalla a partir de su área relativa. Si la bala es enemiga, se destruye la entidad, y si la bala es del jugador, se cambia su estado a NOT_FIRED (más detalles sobre qué pasa en este estado adelante).
- system_collision_bullet_player: Este sistema detecta las colisiones entre una bala y un enemigo. Cuando la colisión llama la creación de una explosión y la actualizacion del score.
- system_collision_enemy_player: Este sistema detecta las colisiones entre un enemigo y la nave del jugador. Cuando la colisión llama la creación de una explosión
- system_enemy_attack_fire: Este sistema maneja la generación de los disparos de los enemigos que se encuentran en ataque.
- system_enemy_attack: Este sistema se encarga de seleccionar un enemigo de manera aleatoria dentro del grupo con un tiempo de tiempo aleatorio también para que darle el comportamiento de ataque adicionándole el componente CEnemySteering y de disparo con el componente CEnemyBulletSpawnerHunting.
- system_enemy_bullet_spawn_convoy: Regula el disparo de los enemigos en su movimiento básico. Hace uso del componente CEnemyBulletSpawner para saber si ya se puede disparar o no. Lo que hace es buscar cuáles entidades no han disparado todavía
- system_enemy_movement: Este sistema maneja el movimiento lateral de los enemigos dentro del grupo, validando la ubicación de los componentes con tag CTagEnemy y en caso de que alguno alcance la posición en extrema a la derecha o a la izquierda, cambia la dirección.

- system_enemy_steering: Este sistema controla el desplazamiento de un enemigo hacia la ubicación del jugador o player. Tiene 3 estados, el primero JUMPING, donde se simula un salto del enemigo al salir del grupo, el segundo es HUNTING, que mantiene una velocidad constante en el eje y al fondo de la pantalla y modifica la velocidad en X hacia donde el jugador. Por último, el estado RETURNING se da cuando se supera el límite inferior de la pantalla y direcciona el enemigo hacia su posición actualizada dentro del grupo.
- system_game_manager: Permite controlar los estados del juego en el momento de su ejecución. Entre los estados se encuentra: READY, PLAY, WIN, DEAD, GAME OVER Y PAUSE. A partir de estos estados, es posible controlar partes particulares de la ejecución del videojuego, por ejemplo, la escena inicial, el momento en el que se pausa el juego o cuando hay se pierden todas las vidas. Este sistema también se encarga de actualizar, en el archivo de configuración interface.json, el valor más alto de high score más alto hasta el momento. Esta actualización se hace cuando hay un GAME OVER o cuando se avanza de nivel. Es decir, si el juego se termina repentinamente sin cumplir alguna de estas dos condiciones, no se actualiza el valor *en el archivo*, solo en pantalla.
- system_input_player: Comunica el input del jugador teniendo en cuenta la tecla presionada y si se levantó o presionó con la función _do_action() del motor, que es la encargada de manejar la acción subsecuente.
- system_movement: Controla el movimiento del componente en general a través de la posición y la velocidad por el delta time.
- system_pause: Es un sistema que permite identificar una entidad que representa el texto Pause y la elimina.
- system_player_bullet_state: Controla el estado de la bala del jugador. Lo que hace en estado NOT_FIRED es seguir al jugador, es decir, forzar su posición a que esté encima de la nave. Por otro lado, el estado FIRED verifica si hay alguna colisión con un enemigo. Cuando esto pasa, se crea la explosión del enemigo y se envía la información del tipo de enemigo al componente de nivel para actualizar los puntos en la pantalla.
- system_player_limit: Controla el límite del jugador con respecto a su movimiento. Básicamente, lo que hace es mantener la posición horizontal si el jugador si este se encuentra a 10 pixeles del borde de la pantalla. Nótese que el límite es mayor al de los enemigos, lo que permite al jugador hacerse un poco más a la derecha o a la izquierda del grupo.
- system_rendering: Es el sistema encargo de pintar los componentes en pantalla
- system_star_field: Permite simular el campo de estrellas y controlar que cuando una estrella salga de la pantalla reaparezca en la parte superior con una ubicación aleatoria.
- system_update_score: Este sistema actualiza el texto correspondiente al score, que se incremente con el valor correspondiente al enemigo eliminado
- system_update_high_score: Este sistema actualiza el texto correspondiente a high score, que aumenta cuando el score supera al valor actual de high score.

Demás clases (servicios)

Además de los componentes, otras clases creadas fueron los servicios, los cuales son los siguientes:

- ConfigsService: Usado para cargar los archivos de configuración. Se tomó la decisión de escribir la lógica que conlleva cargar un archivo haciendo uso de un servicio en vez de hacerlo en la función _load_config_files(), con el fin de tener un servicio global que pueda ser usado en cualquier parte del código. Se creó por buena práctica, porque, de todas maneras, el servicio solo se llama en el motor.
- FontsService: Usado para cargar las fuentes para los textos. Como Pygame no permite, como tal, cambiar el tamaño de los textos, las fuentes deben ser cargadas con un tamaño predefinido, razón por la cual la llave del diccionario para las fuentes es una combinación entre el tamaño y el nombre de la fuente.
- ImagesService: Usado para cargar imágenes, pero en específico para el caso del proyecto, para cargar las imágenes de los sprites. Simplemente las carga una vez, lo que permite su uso posteriormente, a comparación de cargarlas cada vez que se necesitan, pues consume recursos.
- SoundsService: Usado para reproducir los sonidos del juego. A diferencia de los demás servicios, este no permite recuperar archivos, solamente funciona para reproducirlos.

Archivos de configuración

Por último, los archivos de configuración que se usaron fueron los siguientes:

- enemies: Contiene la información relacionada a cada tipo de enemigo. Incluye su imagen, tipo, animaciones, cantidad máxima de balas que puede disparar en su movimiento básico, y puntos que da dispararle.
- enemy_bullet: Contiene la información relacionada a las balas de los enemigos. Incluye su color, tamaño, y velocidad.
- enemy_explosion: Contiene la información relacionada a la explosión de un enemigo cuando colisiona con una bala del jugador. Incluye su sonido, imagen y animaciones.
- interface: Contiene la información relacionada a los textos que se ven en pantalla, Incluye el texto como tal, su posición y tamaño.
- level_01: Contiene la información relacionada al juego en general. Incluye la velocidad de los enemigos, el sonido de los enemigos cuando persiguen al jugador, la información relacionada a las líneas de enemigos (tipo, posición, cantidad y espacio entre ellos), y el mínimo y máximo intervalo de tiempo entre los enemigos que disparan en su movimiento básico.
- player_bullet: Contiene la información relacionada a la bala del jugador. Incluye su sonido, color, tamaño y velocidad.
- player_explosion: Contiene la información relacionada a la explosión del jugador cuando colisiona con una bala enemiga o con un enemigo como tal. Incluye su sonido, imagen y animaciones.
- player: Contiene la información relacionada al jugador. Incluye su imagen, punto de spawn, y velocidad de input (movimiento).
- starfield: Contiene la información relacionada al campo de estrellas del fondo. Inlcuye los colores de las estrellas, cantidad, velocidad, frecuencia de parpadeo y tamaño.
- window: Contiene la información relacionada a la ventana de juego. Incluye el título, tamaño color de fondo y framerate.

Patrones usados

Los patrones que se usaron para el desarrollo fueron los siguientes:

- Game Loop: Este patrón se implementó para el ciclo principal del juego, en donde se obtienen eventos dentro del juego, como las acciones del jugador con las teclas o las colisiones entre balas, enemigos y jugador, se procesan los eventos en caso de existir, sin detener la ejecución, actualiza y dibuja inmediatamente.
- Command: Se implementa para el manejo de los eventos por teclado del jugador a través de un componente Command que los identifica con un nombre, y encapsula la entrada y la fase y un sistema input que itera sobre los componentes, compara con sus propiedades y envía la acción identificada a la clase motor
- ECS: Se explica en la parte superior del documento.
- Service Locator: Se implementa para el manejo de los archivos de datos correspondientes a la configuración, los sonidos, las imágenes y las fuentes, dando un punto global de acceso a estos y haciendo una única carga optimizando su uso ya que se utilizan frecuentemente dentro de la ejecución del juego
- State: este patrón funciona como una FSM o Finite State Machine. Básicamente se utiliza una entidad cuyo comportamiento cambia con base en su estado interno. Es decir, se definen estados y transiciones, donde estas transiciones pueden variar de acuerdo con el entorno. Por ejemplo, al presionar un botón (entorno), se pasa de un estado off mediante una transición a un estado on, y viceversa. Ahora bien, dentro del proyecto se utilizó el patrón State en 3 escenarios. En primer lugar, se utilizó para controlar las escenas del juego. Como se mencionó anteriormente, se crearon los estados READY, PLAY, WIN, DEAD, GAME OVER Y PAUSE, y de acuerdo con estos estados, el comportamiento del juego cambia en tiempo de ejecución. Por ejemplo, cuando se eliminan todos los enemigos, el estado del juego cambia a WIN, se muestra el texto y el sonido al ganar el nivel, y finalmente se retorna el estado a PLAY y se empieza un nuevo nivel. En segundo lugar, se utilizó el patrón para controlar el estado de la bala del jugador y evitar crear una bala cada vez que esta colisiona o se sale de la pantalla. Para eso se crearon 2 estados, NOT_FIRED y FIRED, cuando la bala es lanzada pasa su estado a FIRED, una vez colisiona o sale de los límites de la pantalla esta retorna a la posición del jugador y cambia su estado a NOT_FIRED. Finalmente, también se utilizó el patrón para controlar el desplazamiento del enemigo al realizar el ataque al jugador. Se crearon 3 estados JUMPING, HUNTING y RETURNING. El primero simula el salto del enemigo hacia el jugador, una vez salta, cambia su estado a HUNTING donde mantiene una velocidad constante para perseguir al jugador. En caso de no colisionar contra la nave y tocar el límite de la pantalla, retorna a su posición cambiando su estado a RETURNING.

Reflexión

Para finalizar, se hace una reflexión sobre el trabajo realizado, teniendo en cuenta lo que salió bien, lo que salió mal y lo que puede cambiar a futuro.

¿Qué salió bien?

Lo primero a resaltar de lo que salió bien, fue nuestro flujo de trabajo. Crear issues, manejar su estado en un tablero de tareas, crear pull requests, etc., contribuyó mucho a trabajar de manera organizada y ser transparentes con nuestras labores. También hay que mencionar la buena actitud y la responsabilidad de los integrantes del grupo, cada uno intentamos entregar nuestras tareas según lo establecido por el equipo de trabajo en tiempo y forma. Todos

atendimos a las recomendaciones y observaciones de los demás integrantes en el desarrollo de nuestros requerimientos para así finalizar con un proyecto completo y de calidad.

¿Qué salió mal?

Se pudieron realizar más reuniones sincrónicas para apoyar el avance de todos los integrantes del equipo y ayudar si hay dudas o inquietudes con las tareas definidas para cada uno. Asimismo, hubiera sido una buena práctica que más personas se encargaran de revisar y aprobar los pull requests. Tener solo una persona haciendo esto puede llevar a sesgos o que menos miembros puedan dar su aporte y estar al tanto del avance del proyecto.

¿Qué puede cambiar hacia el futuro?

- Definir horarios para reuniones de trabajo sincrónico.
- Estar más involucrados en la revisión de pull requests. Una sugerencia es establecer una regla dentro de GitHub que solo permita integrar los cambios si hay al menos 2 aprobaciones.