Componente formativo

Diseño de videojuegos y prototipado

Breve descripción:

Se explican conceptos básicos de diseño de videojuegos, programación de comportamientos básicos y construcción de prototipos.

Área ocupacional:

Ciencias Naturales.

Mayo 2023

Grupo de Ejecución de la Formación Virtual

Tabla de contenido

Introducción

A continuación, se explican conceptos básicos para el diseño de videojuegos, programación de comportamientos básicos y construcción de prototipos. Estos conocimientos serán útiles para planear videojuegos, de acuerdo con procedimientos y requisitos técnicos a partir de los requerimientos, permitiendo así, esquematizar el prototipo físico de las mecánicas del videojuego de acuerdo al concepto.

1. La idea y el concepto de videojuego

Los videojuegos son aplicaciones de software que se crean con el fin de entretener. Todos los videojuegos comienzan con una idea que puede variar en el transcurso del tiempo y que se restringe a aspectos como la plataforma de publicación, la plataforma de desarrollo, costos de desarrollo, gustos del grupo objetivo y oportunidades de innovación.

En primer lugar, la idea de videojuego comprende la noción fundamental de jugabilidad, su narrativa, si ha de tenerla, sus personajes y plataforma de publicación. Se trata de establecer aspectos técnicos básicos pero que caracterizan todavía más el carácter del videojuego. A nivel de concepto de videojuego, es posible establecer una idea más profunda de las mecánicas, es decir, la jugabilidad unida con aspectos básicos de motivación y reto. Se puede ser más detallado en la concepción gráfica, el perfil de jugador y de plataforma de publicación.

1.1. Narrativa, personajes y entorno

Algunos aspectos fundamentales que determinan el discurso narrativo que puede llegar a tener un videojuego son la narrativa propiamente dicha, los personajes de la narración y el entorno donde se desarrolla.

La narración.

En primer lugar, es importante tener en cuenta que, en un videojuego, como en cualquier tipo de contenido, se puede encontrar narrativa e historia como elementos que, aunque interdependientes, son diferentes el uno del otro. Todos los videojuegos tienen una historia, que se refiere al universo fantástico de la narración; en tal sentido, se refiere al discurso implícito de carácter ficticio en el que ocurren los eventos que se plantean en el videojuego. Por otro lado, la narrativa tiene relación con la forma como se cuenta dicha historia; en este sentido, involucra al jugador, la experiencia que tiene del juego y su apropiación de la narración, una vez la conoce.

Si bien no todos los videojuegos tienen en sí mismos una historia concreta, pueden discernirse relaciones de eventos y hechos en el transcurrir del juego, donde, al cumplir retos, el jugador construye sus propias narraciones, como es el caso de Pong, el primer videojuego desarrollado en la historia de este tipo de contenidos, publicado en 1972, y que consiste básicamente en una metáfora funcional del ping pong. Conceptualmente, se tienen dos tipos de interacción narrativa: la narrativa embebida, que involucra la construcción en sí misma del universo a través de los mecanismos propios del videojuego (cinemáticas, mecánica de jugabilidad), y la narrativa emergente, que se define en la medida en que el jugador interactúa con el videojuego y se apropia de la historia general del mismo.

En términos generales, se puede decir que existen dos tipos de narrativas presentes en cualquier relato: las narrativas lineales, que, como su nombre indica, conllevan una linealidad en lo que se cuenta, en la interacción y comportamientos, y las narrativas no lineales, que pueden desbordar en diferentes puntos e incluso en diferentes desenlaces. Sea cual sea el caso, estas tipologías están orientadas fundamentalmente a la toma de decisiones que puede llegar a tener el jugador:

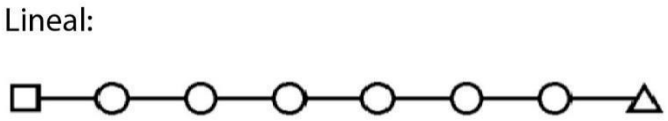
a) Narrativas lineales

Este primer tipo de estructura se caracteriza por ser simple, en lo que se refiere a las decisiones del jugador. Cada una de las etapas de la narración transcurre una después de

la otra, e incluso al punto de discernir su desenlace final. Estas narraciones tienen objetivos específicos y no ofrecen la alternativa de tomar decisiones diferentes que conlleven a eventos diferentes.

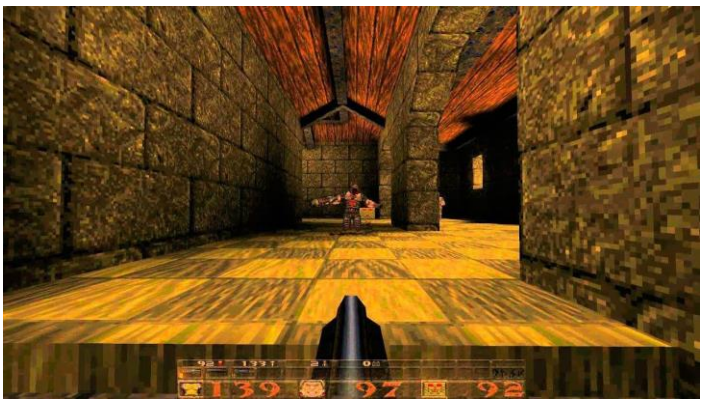
En las narrativas lineales, los eventos y los retos del jugador ocurren en una misma secuencia de tiempo, en un determinado orden cronológico. (Ejemplos: “The Last of Us, The Legend of Zelda: Ocarina of Time”).

Figura 1. Narrativa lineal



En este tipo de narrativas, se tiene un segundo tipo llamado collar de perlas (“String of pearls”), donde la narración lineal es interrumpida cierto tiempo, por periodos cortos, donde el jugador debe tomar decisiones, aparentemente basado en múltiples opciones, que, sin embargo, lo llevarán siempre a una misma línea narrativa que no afecta la narración ni el “gameplay” del videojuego. (Ejemplo: “Call of Duty, Quake”)

Figura 2. Videojuego “Quake”



Tipos de narrativas lineales:

Narrativa ramificada

Esta narrativa es quizás de las más importantes hoy en día y, sobre todo, la que más alternativas ofrece al jugador y más dinámica de jugabilidad. En este tipo de narrativa, tanto el esquema argumental como la dinámica de juegos, se ven afectados directamente por las decisiones que tome el jugador. Su nombre se debe a que el argumento ocupa la base central del tronco de la narración y las ramificaciones las constituyen cada una de las posibles variaciones que tenga la narrativa en función de las decisiones del jugador.

Ramificada binaria

Este tipo de narrativa es limitada y apenas ofrece dos alternativas posibles de decisión, por lo tanto, se encausan directamente a un solo camino. (Ejemplo: *The Secret of Monkey Island, Beyond: Two Souls*).

Ramificada abierta

En esta narrativa se introducen eventos que invitan a decisiones posibles y múltiples, que permiten modificar la historia, de tal manera que el jugador tiene la sensación de estar modificando la historia general del videojuego. Este tipo de narración tiene una gran complejidad, teniendo en cuenta que deben poder abarcarse todos los finales, desenlaces o caminos posibles, para poder tener un control sobre la narración y sobre las decisiones, sin que se vea obvio el posible direccionamiento que debe dársele al jugador. (Ejemplo: *The Walking Dead*).

Narrativa de embudo

En este tipo de narrativa existen ramificaciones en la historia, sin embargo, en algún momento, algunas de estas ramas convergen en puntos específicos de la narración, que en sí mismos constituyen un cuello de botella o puntos de choque. Básicamente, el jugador, aun cuando explore diferentes alternativas, se verá involucrado en una misma línea de narración. (Ejemplo: *Far Cry 2, The Witcher 2*).

Narrativa de caminos críticos

En esta narrativa el videojuego tiene en realidad un único camino, pero el jugador tiene la oportunidad de enfrentarse a retos secundarios, que finalmente conllevan a la línea general. Esta narrativa da la sensación de ofrecer diferentes alternativas y puede tener hasta tres niveles de reto, que le permiten cierta dinámica de juegos sin perder la noción de la línea general. (Ejemplo: “Grand Theft Auto V, The Witcher 3, Fallout 4”).

b) Narrativas no lineales

Como se mencionó anteriormente, las narrativas lineales ocurren en un determinado orden cronológico y dan la oportunidad de tomar ciertas decisiones y cierta capacidad de decisión en los jugadores. Muy diferentes a las narrativas no lineales, donde este tipo de circunstancias no ocurre.

Narración multinivel.

Se encuentra principalmente en los videojuegos de tipo MMO, como “League of Legends y World of Warcraft”. En este tipo de esquema, es más importante el “player agency”, o nivel de control del juego de un jugador, y la posibilidad de incursionar en el mayor número de misiones posible. En este tipo de narrativa, los bloques narrativos no tienen relación alguna entre sí y se hallan distribuidos en regiones del juego, de tal manera que el jugador tiene la posibilidad de unificar todas estas instancias de historia de eventos y de lugares en su cabeza, para construir él mismo el propósito del juego.

Narrativa “parque de diversiones”

En este tipo de narrativa, el carácter de diversidad lo dan los diferentes espacios y lugares en los que el jugador participante puede tener la opción de intervenir. Se llama parque de diversiones porque, precisamente, en un espacio global, se pueden encontrar pequeños lugares específicos, con retos y narraciones específicos. Este esquema de narración se basa en la exploración, teniendo en cuenta que las nuevas líneas narrativas surgen dependiendo de adonde se direccione el jugador. En este sentido, prima la iniciativa de explorar y no necesariamente el perseguir un solo objetivo en específico. En este tipo de videojuegos, existen varias subtramas que se relacionan necesariamente con una trama

general, dando la oportunidad, incluso, de explorar el juego sin siquiera haber abordado alguna parte de la historia principal. Este tipo de narrativa se encuentra en los videojuegos de rol, donde cada uno de los jugadores encarna un personaje, permitiendo identificarse con él y, de esta manera, acercándose más a la narrativa que se plantea, en lugar de simplemente completar retos.

Personajes

Los personajes están constituidos por todas las entidades del videojuego que tienen en sí mismos una identidad, un rol y un papel narrativo y “actoral” dentro del juego; comprende el o los personajes controlados por el o los jugadores y los personajes no jugables o controlados por el sistema.

Roles actanciales

Los roles y las características de los personajes cumplen una función importante en términos narrativos, desde el eje sistemático de lo que realizan dentro de una historia. En este sentido, el análisis será desde el tradicional esquema de los roles actanciales.

Según Greimas, los roles actanciales son realizados por los “actantes” (persona, animal o cosa), por lo tanto, son los que realizan las acciones propias dentro de la narración. Se consideran actantes independientemente de cualquier determinación o de cualquier concepto externo, y su caracterización viene determinada por el rol que desempeñan.

Los tipos de personajes constituyen los modelos de personaje humano o animado que posee rasgos estereotípicos físicos, psicológicos y morales prefijadas por el común de usuario o consumidores de contenido.

A continuación, se puede observar una tabla que reúne los principales roles actanciales desde las propuestas realizadas por tres autores diferentes (Hartt, 2019):

Tabla 1. Roles actanciales

Propp	Souriau	Greimas
Héroe	Fuerza temática orientada	Sujeto
Bien amado o deseado	Representante del bien deseado, del valor orientado	Objeto
Donador o proveedor	Árbitro atribuidor del bien	Destinador
Mandador	Obtenedor virtual del bien	Destinatario
Ayudante	Auxilio, reduplicación de una de las fuerzas	Ayudante
Villano o agresor	Oponente	Oponente
Traidor o falso héroe	***	Oponente

A partir del anterior esquema, se determinarán los siguientes roles actanciales:

a) **Sujeto**: representado esencialmente por el personaje controlado por el jugador o el sistema, de acuerdo con el orden narrativo.
b) **Objeto:** constituye en sí mismo el objetivo o propósito del sujeto (ser amado, dinero, honor, felicidad).
c) **Ayudante**: está representado por la entidad o circunstancia que ayuda al sujeto a realizar su misión. También puede encontrarse un rol ayudante para los oponentes.
d) **Oponente:** es el rol némesis del sujeto. En este sentido, tiene como posible objetivo el mismo del sujeto, pero con intenciones completamente

contradictorias. En el caso de los videojuegos, si el sujeto es controlado por el usuario, el oponente es controlado por el sistema u otro jugador o viceversa.

e) **Destinador:** es el rol que establece, arbitra y orienta los hechos en los que se ven involucrados los personajes. Puede estar representado por una entidad concreta, un fundamento o principio, o una emoción o sentimiento. Es el que motiva al sujeto a cumplir con el objetivo.
f) **Destinatario:** involucra el beneficio directo que logra el sujeto o el oponente al cumplir con las misiones. Puede tener una naturaleza social, ideológica, moral o material. Es quien recibe las acciones o metas del sujeto.

Desde estos roles, se pueden establecer ejes de relación entre cada uno de los tipos de actantes:

a) El eje destinador – destinatario: involucra el control de los valores y, por lo tanto, de la ideología. Determina la creación de estos y su relación con los personajes.
b) El eje sujeto – objeto: determina la trayectoria de la acción y lo que busca el protagonista o héroe de la historia. Determina el eje del deseo, es decir, el propósito que se busca lograr y los obstáculos que se encuentran.
c) El eje ayudante – opositor: este eje determina la facilidad o la dificultad en la comunicación, produce las circunstancias y las acciones propias dentro de la narración. No necesariamente involucra personajes y puede, desde cierto punto de vista, hacer la proyección del esquema de personalidad del personaje sujeto.

Entorno

Desde el punto de vista narrativo, el entorno constituye el espacio físico en el cual se lleva a cabo la historia y las acciones de cada uno de los personajes. Necesariamente influye en cada uno de los eventos narrativos y cronológicos, y determina en gran medida aspectos de jugabilidad. A este concepto de entorno se añade el concepto de contexto, de

tal manera que, no solamente se define en términos de espacialidad, sino en lo que lo caracteriza, en términos funcionales y la forma como se aborda y aporta en la historia. Las características o puntos de vista comprenden: la cronología, la ambientación, la funcionalidad (jugabilidad) y el estilo visual y narrativo.

Por otro lado, y desde el punto de vista específico de jugabilidad, comprende la relación directa de interacción con el jugador, y se encuentran entornos y contextos basados en la exploración, el paso de obstáculos.

1. **Punto de vista cronológico:** desde esta perspectiva, se encuentran entornos y contextos donde entra a jugar el tiempo y la época a la que se refiere el videojuego. Determinan tanto la narrativa como la ambientación de utilería, arquitectura y vestuario.
2. **Punto de vista ambiental:** desde esta perspectiva, es posible referirse a los aspectos propios de la sensación que causa un espacio como tal y la forma como puede ser simulado a través del videojuego. En este aspecto, es posible involucrar la iluminación, los ambientes naturales y urbanos, la decoración de interiores y la sensación de realismo de los materiales y texturas.
3. **Punto de vista funcional:** en términos funcionales y de jugabilidad, se tiene la clasificación de elementos de acuerdo con su papel dentro de la mecánica de videojuegos: están la utilería, los espacios arquitectónicos, naturales y urbanos, y los vehículos.
4. **Punto de vista de estilo**: este aspecto involucra el entorno y el contexto, en términos de la apropiación de un estilo visual artístico específico. Comprende también el estilo y género narrativo que se haya implementado. Desde esta instancia, se entendería que para un videojuego de terror se implemente una ambientación, quizás una época, y una iluminación diferente, así se trate de un videojuego de aventuras.

1.2. Mecánicas, dinámicas y estéticas

En la actualidad, en el diseño de videojuegos, se especifica una tríada de aspectos que tienen que ver con la totalidad de elementos que deben determinar la experiencia de usuario, y son las mecánicas, las dinámicas y la estética. Las mecánicas se refieren a las reglas y sistemas que determinan la experiencia de juego en sí misma. Las dinámicas hacen referencia al comportamiento de las mecánicas en respuesta a las acciones que realiza un jugador. Y la estética hace referencia a la respuesta emocional que evoca el usuario y, al mismo tiempo, involucra el aspecto visual del videojuego como elemento de jugabilidad.

Dentro de este marco de diseño de videojuegos, se entiende que el jugador y el diseñador de juegos tienen una diferente perspectiva del videojuego. El propósito, en este sentido, es generar un puente entre el diseñador de videojuegos y el jugador.

Entendiendo los videojuegos como productos y servicios de entretenimiento, presentan un comportamiento imprevisible en sus momentos de consumo, y esto se debe, precisamente, a las motivaciones y a los procesos de inmersión e involucramiento emocional de los jugadores. Esto permite establecer los componentes de los videojuegos y descomponerlos en sus aspectos funcionales y su pertinencia alrededor de la acción de jugar; determinando así un esquema de elementos funcionales interactuantes entre sí, que determinan la relación coherente entre el videojuego y el jugador:

a) Mecánica

Las mecánicas determinan los componentes del videojuego, es decir, definen las reglas como acciones y comportamientos permitidos al jugador, y restringen de una u otra manera su trayectoria en el videojuego. Involucra la programación.

Dentro de la construcción de las mecánicas de juegos, y específicamente en este modelo de mecánicas, dinámicas y estéticas, a continuación se presentan las tipologías de mecánicas establecidas por Brenda Brathwaite e Ian Schreiber (Brathwaite y Schreiber, 2009):

1. **Mecánicas de inicio:** comprenden esencialmente las reglas que describen la forma como debe empezar el videojuego.
2. **Mecánicas del triunfo**: comprenden las reglas que describen la manera cómo puede ganar el jugador el videojuego, estableciendo condiciones específicas para lograr la victoria.
3. **Progresión del juego**: comprenden las reglas que determinan cómo se avanza durante el juego, es decir, si existe una progresión por turnos o en tiempo real. En algunas ocasiones, se pueden encontrar videojuegos donde el jugador no tiene una alternancia de turnos, sin embargo, debe corresponder a determinados retos que establecen ciclos cortos de progresión.
4. **Acciones del jugador**: En este sentido, las mecánicas de juegos, dentro del patrón de acciones del jugador, deben establecerse como verbos o acciones propiamente dichas que puede realizar un jugador, como saltar, caminar, correr o disparar. Estas acciones del jugador pueden determinar específicamente las mecánicas de núcleo y, al mismo tiempo, determinar los efectos que pueden dar lugar para cambiar el estado del juego.
5. **Definición de la visión del juego**: son las reglas que determinan qué información será accesible para el jugador y en qué momento.

b) Dinámica

La dinámica comprende la descripción del modo de comportarse de una mecánica en específico durante el tiempo de juego, para definir así las interacciones entre los jugadores. Su propósito es crear las experiencias estéticas, las respuestas emocionales del jugador y, por consiguiente, determinar un direccionamiento comportamental. Involucra la motivación del jugador.

En las dinámicas principales que pueden ser utilizadas dentro de la mecánica núcleo de un videojuego, es importante tener en cuenta que la combinación de una o varias de

estas dinámicas puede facilitar la interacción entre el jugador y el videojuego, de tal manera que resulte verdaderamente atractivo e inmersivo:

1. **Adquisición territorial**: en esta dinámica se describe un tipo de juegos donde el jugador puede moverse a través del entorno virtual y su objetivo es poder controlar determinadas zonas de dicho territorio.
2. **Predicción:** dinámica en la que el jugador puede discernir una serie de eventos que pueden llegar a dar lugar a premios y recompensas. En este sentido, el objetivo es que el jugador, de manera predeterminada, pueda analizar y comprender la forma en la que puede jugar para ganar determinadas misiones o retos.
3. **Razonamiento espacial:** esta dinámica es característica de la mayoría de puzzles, y consiste en la adquisición de determinadas habilidades y, por consiguiente, el logro de determinados retos a partir del razonamiento espacial adecuado.
4. **Supervivencia:** comprende el instinto fundamental del ser humano de sobrevivir y prosperar. Se trata de videojuegos donde el objetivo fundamental es protegerse a sí mismo y sobrevivir durante el videojuego.
5. **Colección:** en esta dinámica se trata de coleccionar o reunir la mayor cantidad de elementos, teniendo en cuenta que estos pueden ser utilizados para cumplir con determinados retos o adquirir ciertas habilidades o puntajes.

c) Estética

La estética describe la respuesta que tiene el jugador, desde el punto de vista emocional, en el momento de interactuar con el sistema de juego. Como propósito, la estética busca hacer que el juego sea divertido, y que sea agradable y excitante la búsqueda del objetivo final. Involucra la inmersión.

1.3. Lineamientos artísticos

Los lineamientos artísticos hacen referencia a los esquemas de direccionamiento de arte que se hace sobre la estética de un videojuego, basándose principalmente en las tendencias de estilo que imperan en el mercado o bien aquellas que se proponen como iniciativa y con el fin de innovar. En términos generales, se espera que los lineamientos artísticos determinen tanto la calidad en la apariencia como los mecanismos de inversión a partir del apartado gráfico.

Estilos artísticos:

Video 1. Estilos artísticos de los videojuegos

Estilos artísticos de los videojuegos

Síntesis del video: Estilos artísticos de los videojuegos

En el arte retro, se utilizan elementos de arte 2D, principalmente, basados en *sprites* y en técnicas que retornan a la estética de los videojuegos antiguos.

En esta estilística, se recurre a perspectivas de cámara basadas en el *scroll* horizontal o vertical, y mecánicas exigentes.

Este estilo visual, arte "Cartoon", se basa en el valor y técnica de la ilustración y dibujo animado. Puede ser tanto 2D como 3D, y su nivel de abstracción formal coincide con lo caricaturesco del "cartoon".

El arte único, como su nombre indica, se trata de un estilo donde prima la originalidad, sobre todo, la intención de generar una corriente estilística propia original y que incluso pueda determinar tendencias.

Por lo general, prima en estos juegos la oportunidad de innovar en su estilo y, cuando logra penetrar en el mercado y tener un impacto importante, se convierte en tendencia y marca la estilística posterior de juegos de otros estudios.

En este tipo de estética, se utilizan fundamentalmente los géneros del arte para dar protagonismo fundamental a la apariencia visual y gráfica del videojuego.

En tal sentido, tiene un valor artístico importante y siempre acercándose al realismo e incluso marcando cierta similitud o relación con las escuelas de artes realistas.

Finalmente, está e**l arte realista.** Este tipo de esquema artístico trata precisamente de resaltar el realismo de las cosas, de representar con la mayor fidelidad entornos, personajes, arquitectura, objetos; de tal manera que el jugador se sienta involucrado en una escena realista y que perfectamente podría sumergirlo en una historia relativamente verosímil.

Estilo artístico según la tecnología gráfica

Un aspecto importante al momento de idear y plantear el esquema visual de un videojuego es la tecnología con la cual se dispone para realizarlo. También es importante

tener en cuenta las características y las posibilidades que tiene el motor del videojuego. Desde esta perspectiva, se establecen las siguientes tendencias de estilo posibles de acuerdo con las tecnologías presentes, teniendo en cuenta las limitaciones tanto de la plataforma de desarrollo como de la plataforma de publicación:

Pixel Art

Es un estilo de arte para videojuegos donde los gráficos se construyen a partir de mosaicos que, a manera de mapas de pixeles, permiten realizar toda la gráfica y toda la apariencia basada principalmente en los aportes estilísticos de los videojuegos antiguos. Estos videojuegos tenían limitaciones técnicas que exigían la creación de los gráficos en bajísima calidad y con características de color bastante limitadas. Como estilo actual, ofrece la oportunidad de crear videojuegos utilizando el píxel como base de apariencia visual, sin las restricciones técnicas que las tecnologías antiguas tenían.

Un aspecto importante de este estilo es que es característico de algunos videojuegos independientes, ya que resulta económico implementarlo y generar toda una estética general para un videojuego, logrando resultados de atractivo y apariencia muy llamativa.

Ilustración 2D

Del estilo del píxel art se pasa muy fácilmente al del recurso estilístico propio de la ilustración 2D. El propósito en la implementación de este estilo es brindar una apariencia similar a los dibujos animados, pero recurriendo a las actuales tecnologías de desarrollo, bien sea el motor 3D o 2D.

Vectorizados

Si bien los anteriores ejemplos se basan fundamentalmente en la construcción de imágenes basadas en píxeles, en este caso, los gráficos son creados a partir de geometrías en sistemas de coordenadas de tres a dos ejes, y el dibujo resultante, básicamente, es un vector que puede exigir un mayor esfuerzo de cálculo matemático, sin embargo, aporta un significativo valor estético.

Polígonos

Es el principal recurso de construcción formal de los videojuegos 3D y básicamente se trata de la construcción de modelos tridimensionales mediante mallas poligonales. Como en el caso anterior, se trata de gráficos construidos con geometrías cuyo cálculo matemático es exigente, pero compensado con resultados de realismo y de impacto visual significativo.

Voxels

En realidad, se trata de una construcción basada en los gráficos 3D, sin embargo, las formas de los elementos que componen el videojuego, bien sea arquitectura, vehículos, objetos, o personajes, se basa en la implementación de una unidad mínima de construcción que es una simple caja 3D. En la actualidad, más que una técnica de construcción, es un estilo visual.

Gráficos 2.5D, Pseudo 3D o Cámara Fija

En primer lugar, estos videojuegos pueden estar construidos en motores 2D o 3D, mediante modelos tridimensionales o "sprites", y lo que constituye su verdadero valor estético es la manera como se asume la perspectiva de cámara, generalmente perpendicular al plano general del videojuego.

1.4. Monetización

Para establecer una correcta estrategia de monetización es importante tener claras las funcionalidades del videojuego, el grupo objetivo, sus hábitos de juegos y su estilo de vida, en términos de capacidad adquisitiva y gustos.

Desde el punto de vista de jugabilidad y de relación entre el usuario o jugador y el videojuego, la monetización suele considerarse como una mecánica más, que establece también una serie de dinámicas relativas a la interacción comercial, entendiendo el videojuego como un producto o servicio comercial.

En términos generales, se pueden dividir los videojuegos en dos tipos, de acuerdo con la forma como pueden ser adquiridos: aquellos que se ofrecen como free, o gratuitos, y los de pago. A continuación, se relacionan las estrategias de monetización más comunes que pueden manejarse de manera independiente o combinada, de acuerdo con las pretensiones establecidas en el documento de diseño y, por consiguiente, el modelo de negocios.

1. **"Freemium" - Gratis + Oportunidades de monetización adicional:** este es uno de los modelos de monetización más comunes y se trata básicamente de videojuegos de descarga gratuita y, una vez los jugadores se han mantenido bajo cierto nivel de fidelización, se puede optar por la monetización a partir del pago por contenido adicional. Dentro de los modelos "freemium" se puede encontrar:

   a) Uso limitado

En este esquema, el videojuego puede jugarse gratis durante un tiempo específico y ofrece, en el caso de datos, un espacio de almacenamiento limitado. Pasado el tiempo de prueba o el espacio de almacenamiento se ofrece la alternativa de adquirir una cuenta o versión de pago, con mayores tiempos de juegos o espacio de almacenamiento.

b) Prueba gratis

En este modelo, se ofrece el videojuego con todas las funcionalidades durante un tiempo limitado; posteriormente deberá adquirir una licencia que le permita continuar con el videojuego.

c) Funcionalidad

En este modelo, los videojuegos se ofrecen con una limitada funcionalidad, por ejemplo, un número mínimo de niveles o un número mínimo de habilidades de los personajes. El propósito es que, en el momento de avanzar en el juego, el jugador se anime a comprar nuevos elementos de juegos como personajes habilidades o niveles.

d) Experiencia de usuario

En este modelo, aparecen dos tipos de posibilidades de monetización que comprenden, por ejemplo, el pago por parte del usuario para que no aparezca publicidad in-app, y, al mismo tiempo, el pago de empresas anunciantes por la pauta a través de estos videojuegos.

2. **"Paid"** o Pago por descarga: este es un modelo de monetización tradicional, donde el usuario paga por anticipado por un videojuego para que le sea permitida su descarga. Una vez descargado, el jugador tendrá acceso a todas las funcionalidades y el estudio desarrollador no tendrá ningún otro medio de monetización.
3. Suscripción: el modelo de suscripción requiere un pago regular, bien sea mensualmente o anualmente, que permita el completo uso funcional del videojuego. También pueden sugerir posibilidades de actualización periódicas del videojuego, a las cuales se tendrá derecho por haberlo adquirido y mantener constantes los pagos que se sugieren.
4. Publicidad "**In-App"**: en este caso, se trata de un modelo de monetización donde la publicidad se presenta constante, sin posibilidades de ser eliminada por el usuario, y que fácilmente puede fusionarse con el contenido para que forme parte de su diseño de nivel. Ejemplo de ello pueden ser los videojuegos donde se pueden observar edificios con vallas alusivas a empresas existentes.
5. **"Paidmium: Paid"** + Oportunidades de monetización adicionales: en este tipo de modelo, además de pagar por la descarga del videojuego, el usuario tiene la oportunidad de adquirir servicios anexos al videojuego y, obviamente, pagar por ellos. La monetización se logra mediante compras desde el interior de la aplicación y que, básicamente, constituyen mejoras en la experiencia de usuario, por ejemplo,

eliminar publicidad o comprar más contenidos o paquetes que le permitan mejorar su desempeño en el juego.

6. **"Sponsorship – Publisher"**: el primer elemento constituye el modelo más generalizado en el desarrollo general de aplicaciones y consiste básicamente en la búsqueda de inversores que financien el proyecto de desarrollo, ofreciendo un modelo de ganancias compartidas. Para la segunda alternativa, la búsqueda de inversores se concentra fundamentalmente en las compañías de publicación que se especializan en la comercialización y difusión de videojuegos, y donde los esquemas de negociación se basan en ganancias compartidas a partir de cierto número de ventas del videojuego.

2. Algoritmia y lógica de programación

En el presente capítulo, se observarán las nociones fundamentales que tienen que ver con la programación orientada a objetos aplicada en videojuegos y, en específico, al motor de videojuegos Unity 3D.

2.1. Descripción de la API de Unity (C-Sharp)

El concepto API significa interfaz de programación de aplicaciones y reúne un conjunto de protocolos que permiten la interacción entre dos componentes de "software" independientes. Para la documentación que contiene descripciones detalladas de la API de secuencias de comandos que proporciona Unity, es necesario estar acostumbrado o con cierto nivel de conocimiento en programación con Unity.

La referencia de secuencias de comandos se organiza de acuerdo con las clases disponibles para las secuencias de comandos que se describen, junto con sus funciones, métodos, y demás elementos de sintaxis necesarios para programar correctamente.

Las API se agrupan por categorías, dependiendo del tipo de elemento estructural de programación, y se pueden seleccionar en el menú de la izquierda. En términos de programación, la sección de UnityEngine será el principal recurso de búsqueda.

Soporte de entorno de desarrollo integrado (IDE): El entorno de desarrollo integrado (IDE) es una pieza de "software" de computadora que proporciona herramientas e instalaciones para facilitar el desarrollo de otras piezas de software. Unity admite los siguientes IDE:

1. **Visual Studio (IDE predeterminado en Windows y macOS)**

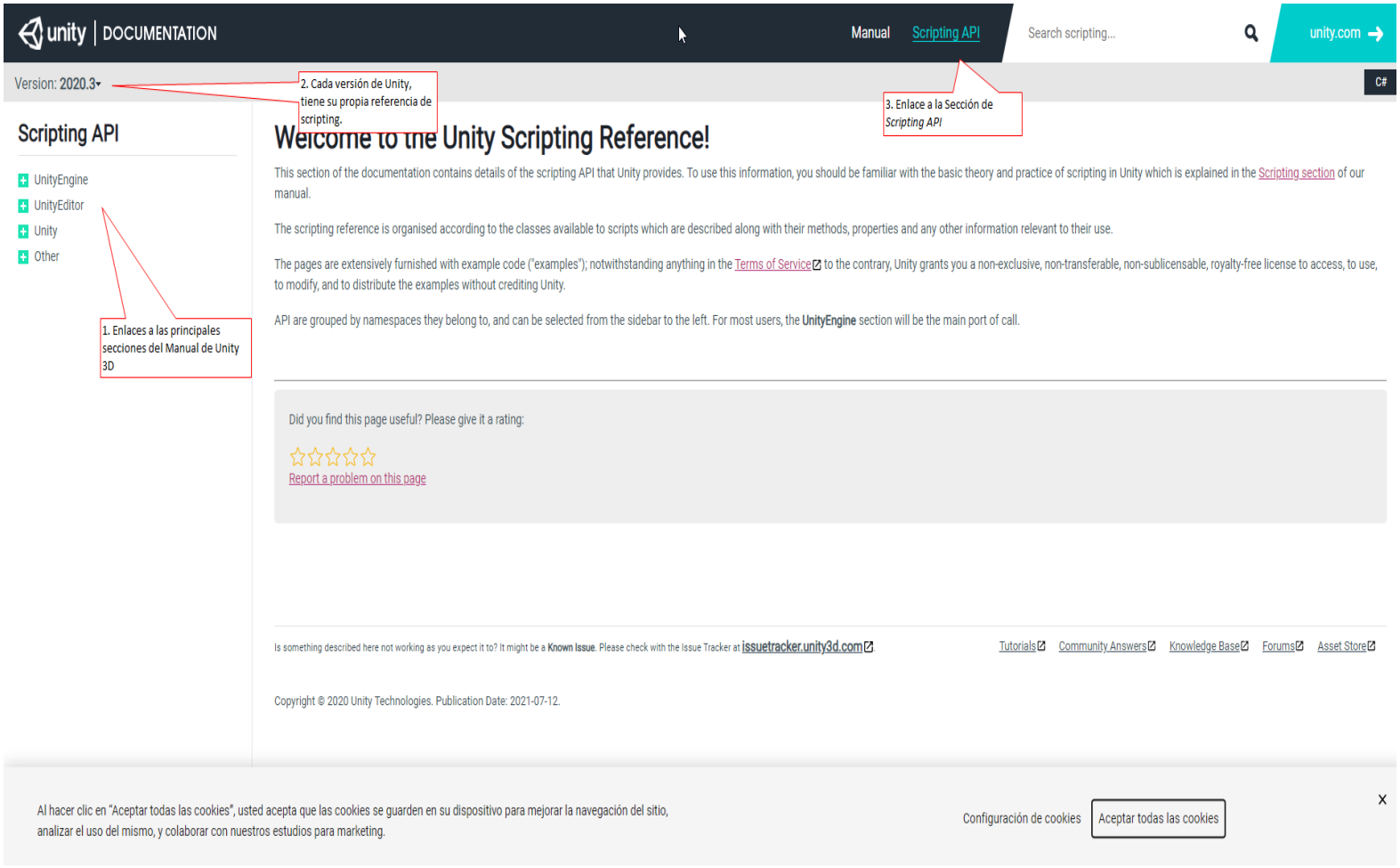
Visual Studio se instala de forma predeterminada cuando se instala Unity en Windows y MacOS. En Windows, puede optar por excluirlo cuando seleccione qué componentes descargar e instalar. Visual Studio está configurado como Editor de scripts externos en Preferencias (Unity > Preferences > External Tools > External Script Editor). Con esta opción habilitada, Unity inicia Visual Studio y lo usa como el editor predeterminado para todos los archivos de secuencia de comandos.

En MacOS, Unity incluye Visual Studio para sistema operativo Mac como el IDE de C#. Visual Studio Tools para Unity (VSTU), proporciona la integración de Unity para Visual Studio para Mac (VS4M).

2. **Visual Studio Code (Windows, macOS, Linux):**

En Unity, la edición de archivos de programación se hace en Visual Studio Code (VS Code). Para abrir scripts en VS Code, selecciónelo como Editor de script externo en las Preferencias del editor (menú: Unity> Preferencias> Herramientas externas> Editor de script externo). Para obtener información sobre el uso de VS Code con Unity, consulte la documentación de Visual Studio sobre Desarrollo de Unity con VS Code.

Figura 3. Ventana de referencia de script



2.2. Estructura y sintaxis básica de C-Sharp

Las líneas de instrucción de los archivos de programación les definen a los "GameObjects" cómo comportarse. En este sentido, la programación en el motor de videojuegos difiere de los métodos de programación pura en que en el motor se definen archivos de programación que serán interpretados y compilados para la plataforma específica de publicación.

Una cualidad muy importante del motor es que permite realizar multitud de tareas de manera simultánea y es el desarrollador el encargado de establecer todas estas relaciones funcionales, además de la participación de cada uno de los miembros del equipo de desarrollo, permitiendo utilizar la enorme versatilidad de Unity.

Un archivo "script" debe estar necesariamente asignado a un "GameObject" en la escena para que Unity pueda utilizarlo en escena, es decir, invocarlo. Los archivos de programación deben estar escritos en un lenguaje comprensible por el motor.

El lenguaje de programación que utiliza Unity se llama C# o C Sharp, que es un lenguaje orientado a objetos. Como cualquier lenguaje de programación, tiene una estructura de sintaxis, es decir, una manera de escribirse y con unas características semánticas específicas. A continuación, se podrá observar un ejemplo de estructura de código en C#.

```
using UnityEngine;
using System.Collections;

public class MainPlayer : MonoBehaviour {

    // Use this for initialization
    void Start () {

    }

    // Update is called once per frame
    void Update () {

    }
}
```

La anatomía básica de un script en C Sharp (C#) está constituida fundamentalmente de variables, funciones y clases.

Las variables son elementos en la sintaxis del programa que tienen la capacidad de contener valores e información referente a determinados objetos y tienen una utilidad importante en función de esta información o de estos valores. Los nombres de las variables deben escribirse en minúscula.

Las funciones son pequeñas estructuras de código que realizan tareas específicas y permiten comparar y manipular variables. Los nombres de las funciones comienzan con la primera letra en mayúscula. El código de programación se organiza en funciones para que sea fácilmente utilizable por el programa.

Las clases son estructuras de código que permiten organizar conjuntos de variables y funciones, que funcionan como plantillas, definiendo propiedades de un objeto que pueden ser utilizadas de diferentes formas. Escribir códigos de programación es, en esencia, una labor de comparación entre objetos, sus comportamientos, estados y valores actuales; determinando resultados lógicos y secuencias lógicas.

Variables.

En Unity, los archivos de programación se deben comenzar, por lo general, con la mención o declaración de las variables, estableciendo si serán visibles o no, es decir públicas o privadas, al mismo tiempo que el tipo y el nombre de la variable, en este orden de secuencia.

```
public class DemoScript : MonoBehaviour {

     public Light myLight;
private Light myOtherLight;
```

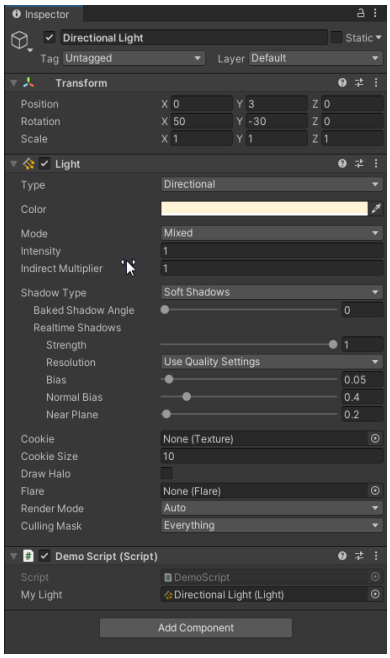
```
}
```

Al declarar variables visibles o de forma pública, pueden ser observables a través del inspector del motor y es una manera muy sencilla de poder controlarlas, para así modificar sus valores y por lo tanto su comportamiento. Cuando la variable es declarada como privada, no puede verse a través del inspector y sólo puede cambiarse su valor a través de la intervención del código de programación.

Además, cuando se hacen públicas las variables, se puede acceder a ellas a través de otros "`scripts`" u otras clases, permitiendo la interacción entre diferentes "`scripts`" y haciendo más dinámica la jugabilidad.

Además de este carácter público o privado, también está el tipo de variable, es decir, la especificación del tipo de información que puede almacenar y, por lo tanto, el tipo de valor que podría mantener en memoria. En términos generales, es necesario especificar el tipo de variable que se está declarando para poder indicarle al motor cómo debe manejar la información o valor que contenga.

Figura 4. Panel del inspector

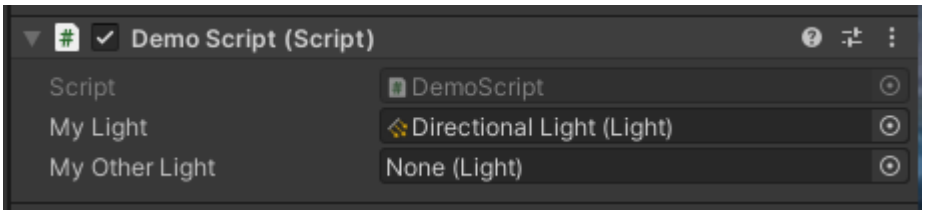


Control de variables en el Inspector de Unity: es importante que las variables tengan un nombre específico, de acuerdo con la información a contener o almacenar, y que al mismo tiempo pueda ser diferenciable de las demás variables que pueda llegar a tener el "script". Al mismo tiempo, tener en cuenta que la variable no se puede nombrar con un número al inicio, ni contener espacios vacíos.

En la denominación de una variable se suele utilizar un estilo o convención de nomenclatura que es camelCase. Esta forma de nombrar la variable comienza con una letra en minúscula y, si se trata de parejas de palabras, se escriben unidas, y la primera letra de la segunda palabra se escribe en mayúsculas, por ejemplo: "myLight".

Tal cómo puede observarse en la siguiente imagen, Unity permite, una vez compilado el código, ver los nombres de las variables separados en el inspector.

Figura 5. Panel de inspector – componente "script"



Funciones

Los “scripts” permiten controlar variables a través de funciones. En términos generales, las funciones más utilizadas se ejecutan de manera automática en el motor, sin embargo, es posible también indicar en qué momento se pueden ejecutar estas funciones de manera controlada.

```
using UnityEngine;
using System.Collections;

public class DemoScript : MonoBehaviour {

public Light myLight;
private Light myOtherLight;

      void Awake () {}

void Start () {}

      void Update () {}

      void FixedUpdate () {}

      void LateUpdate () {}
}
```

Las principales funciones en Unity son:

a) "Awake": se ejecuta sólo una vez en el momento de iniciarse el objeto del juego que contenga el "script". Hay que tener en cuenta que esta función se ejecuta siempre y cuando el objeto de juegos se encuentre inactivo, aún cuando no esté habilitado a través del editor.
b) "Start": de la misma manera que la función anterior, la función "Start" se ejecutará si el "GameObject" que tiene asignado el script está activo, siempre y cuando el componente de script esté habilitado.
c) "Update": esta función se ejecuta una vez cada fotograma y es la función donde se escribe la generalidad de la lógica que se programa y establece para ese "script" específico. Vale decir que, al ejecutarse una vez cada fotograma, contendría instrucciones que siempre estarán operando, siempre y cuando así se establezca o suceda algo en específico.
d) "FixedUpdate": esta función se utiliza sobre todo en efectos de física o efectos de animación, y tiene como fin poder ejecutar relativas acciones independientes de los fotogramas por segundo, de tal manera que el efecto o la animación sea más fluido y eficiente.
e) "LateUpdate": esta función se asemeja a "Update", pero "LateUpdate" se ejecuta al final del "frame" o cuadro. Básicamente, el motor utilizará esta función" LateUpdate" para ejecutar acciones actualizadas hasta el final del "frame".

Al escribir una función, es importante recordar que debe empezar por el tipo de función seguido del nombre de la función, escribiendo la primera letra del nombre en mayúscula, seguido de los parámetros entre paréntesis (si corresponde). El cuerpo de la función, es decir, las acciones que se realizan, se escriben entre llaves o corchetes:

```
public class DemoScript : MonoBehaviour {
```

```
void MyFunction () {

     }
}
```

¿Cómo se llama a esta función?

```
public class DemoScript : MonoBehaviour {

     void MyFunction () {
          myLight.enabled = !myLight.enabled;
     }
}
```

Las funciones pueden realizar cálculos y posteriormente presentar un valor, sin embargo, pueden procesar información y no obtener de la función ninguna respuesta. Estas funciones son de tipo "void".

Clases

Son paquetes o conjuntos de variables y funciones. En términos generales, un archivo de script es en sí mismo una clase y la clase debe tener el mismo nombre del “script”:

```
using UnityEngine;
using System.Collections;

public class DemoScript : MonoBehaviour {
```

```
    public Light myLight;

     void Awake () {
          int myVar = AddTwo(9,2);
          Debug.Log(myVar);
     }

     void Update () {
          if(Input.GetDown("space")){
               MyFunction();
          }
     }

     void MyFunction () {
          myLight.enabled = !myLight.enabled;
     }

     int AddTwo (int var1, int var2) {
          int returnValue = var1 + var2;
          return returnValue;
     }
}
```

Por otro lado, una vez el script se integra a un “GameObject”, debe derivarse de la clase de base, llamada “MonoBehaviour”. Las clases también pueden ser públicas o privadas.

En Unity, se pueden crear clases personalizadas, como en el siguiente ejemplo, y deben solicitarse de manera serializada, lo que significa que se convertirán en datos

digeribles y comprensibles por el motor, de tal manera que puedan también observarse y hacerse visibles en el inspector.

```
using UnityEngine;
using System.Collections;

[System.Serializable]
public class DataClass {
      public int myInt;
      public float myFloat;
}

public class DataScript : MonoBehaviour {

    public Light myLight;

      void Awake () {
            int myVar = AddTwo(9,2);
            Debug.Log(myVar);
      }

      void Update () {
            if(Input.GetKeyDown("space")){
                  MyFunction();
            }
    }

      void MyFunction () {
            myLight.enabled = !myLight.enabled;
```

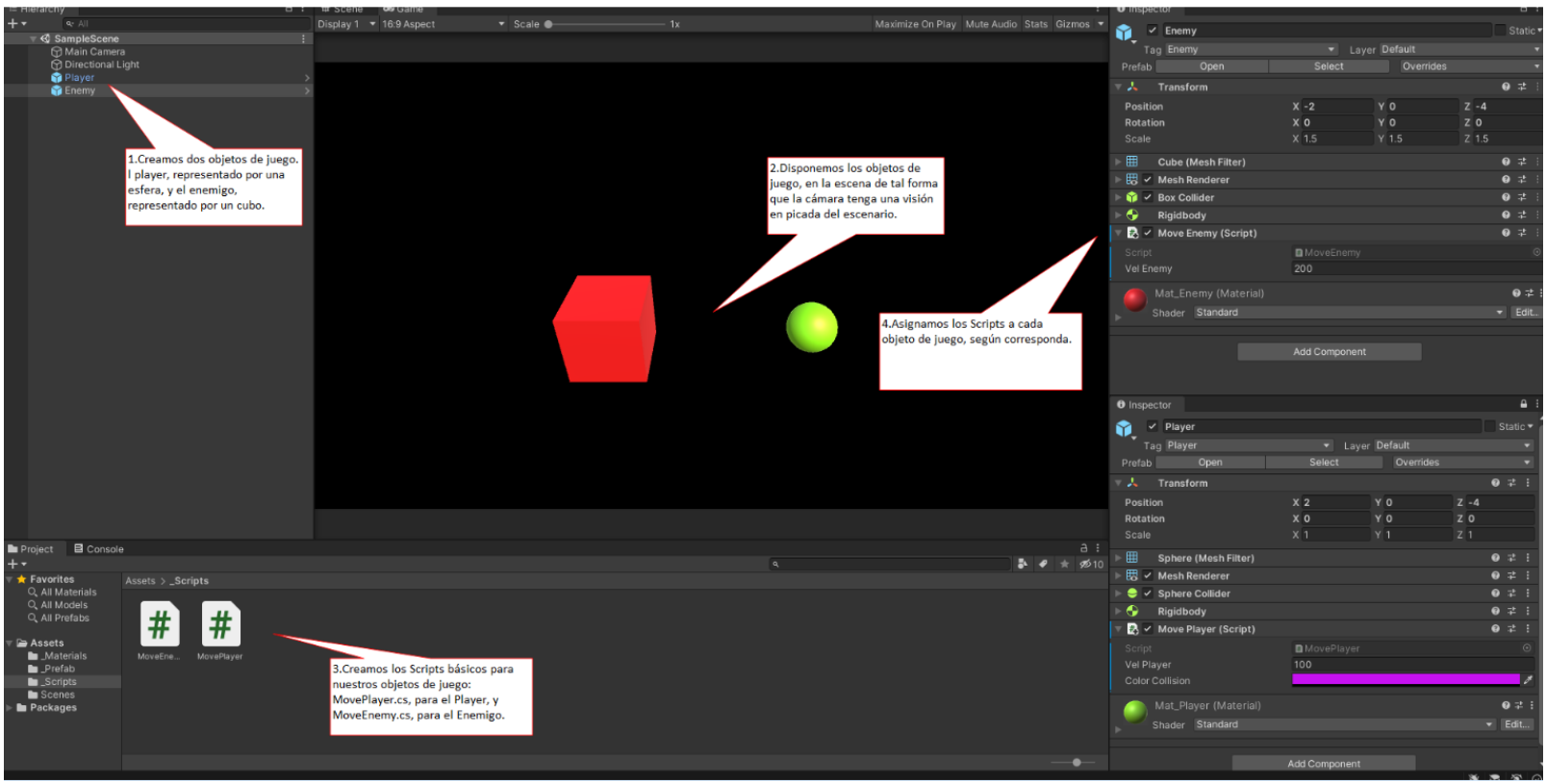
```
        }

        int AddTwo (int var1, int var2) {
              int returnValue = var1 + var2;
              return returnValue;
        }
}
```

Las funciones, sus variables y las estructuras de clases son fundamentales para comprender la manera en la que se escribe código de programación en el motor de Unity.

2.3. Programación de comportamientos básicos de juego

Los videojuegos, en su jugabilidad, y específicamente las mecánicas de videojuego, se relacionan directamente con los procesos de codificación o programación de comportamientos. Desde un ejemplo práctico, se verá cómo se plantea una lógica de programación basada en tres mecánicas básicas: movimiento, colisión y premio / castigo.

Figura 6. Ventana de gameplay



Para mayor ilustración de cómo comenzar a crear la programación de un videojuego, observe los siguientes videos:

Crear carpeta de proyecto del videojuego

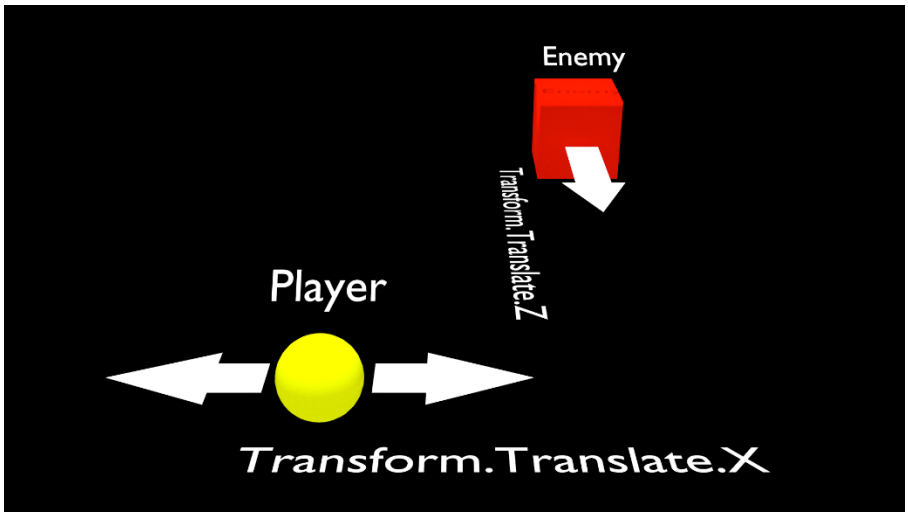
Configuración de escena

Mecánica de movimientos

Como idea inicial, se tiene un mini juego que consiste en mover de izquierda a derecha y viceversa, un objeto de juego que representará a un personaje, para tratar de

esquivar objetos de juego, controlados por el sistema, que se dirigen hacia él y que actuarán como obstáculos.

Figura 7. Algoritmo visual – Transformación



a) El primer comportamiento por definir es el movimiento del personaje `Player`, utilizando la clase `Transform`, que permite realizar cambios de transformación en posición, movimiento, rotación y escala:
   1. Para el personaje `Player`, se define una variable `velPlayer`, que determinará la velocidad.
   2. En la función `Start`, se dará una posición específica en la que aparecerá, utilizando la transformación de posición mediante un nuevo vector de posición 3D.
   3. En la función Update, se especificará un movimiento de traslación en el eje X, de izquierda a derecha, (transform.Translate(x,y,z)), mediante un método de entrada de teclado, (Input.GetAxis(“Horizontal”)), en función del tiempo medido en fotogramas por segundo (Time.deltaTime) y la velocidad en la variable `velPlayer`.

Observe el video de programación de movimiento de Player

```
public class MovePlayer : MonoBehaviour
{
    public float velPlayer;

    void Start()
    {
        float movHorizontal = (Input.GetAxis("Horizontal") *
velPlayer * Time.deltaTime);
        transform.position = new Vector3(0.0f, 0.0f, -6.0f);
    }
        void Update()
    {
        transform.Translate(movHorizontal * Time.deltaTime, 0.0f,
0.0f);
    }
}
```

b) El segundo comportamiento por definir es el movimiento del personaje Enemy, utilizando la clase Transform, que permite realizar cambios de transformación en posición, movimiento, rotación y escala.

1. Para el personaje Enemy, se define una variable velEnemy, que determinará la velocidad.
2. En la función Start, se dará una posición específica en la que aparecerá, utilizando la transformación de posición mediante un nuevo vector de posición 3D, y de forma aleatoria en el eje X, mediante el método Rando.Range(a,b)
3. En la función Update, se especificará un movimiento de traslación mediante el eje Z, de adelante hacia atrás, (transform.Translate(x,y,z)), en función del

tiempo medido en fotogramas por segundo (Time.deltaTime) y la velocidad en la variable velEnemy.

4. Para dar un mayor nivel de impredecibilidad a la mecánica de movimiento del enemigo, si utiliza una condicional if(), permitirá randomizar (Random.Range(a,b)) el reposicionamiento del personaje Enemy, al momento de llegar a una posición límite del escenario if(posActual <= -10.0f). Este método permite utilizar el mismo objeto y ahorrar recursos de proceso y memoria.

Observe el video de programación de movimiento de Enemy

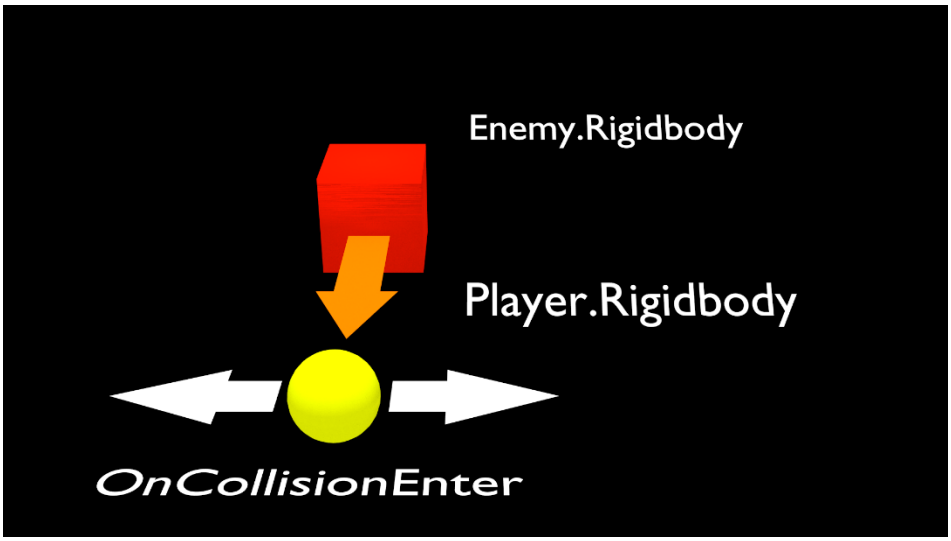
```
public class MoveEnemy : MonoBehaviour
{
     public float velEnemy;

    void Start()
    {
        transform.position = new Vector3(Random.Range(-10.0f,
10.0f), 0.0f, 100.0f);
    }

    // Update is called once per frame
    void Update()
    {
        float posActual = transform.position.z;
        transform.Translate(0.0f, 0.0f, -velEnemy * Time.deltaTime);
        if(posActual <= -10.0f){
            transform.position = new Vector3(Random.Range(-10.0f,
10.0f), 0.0f, 100.0f);
        }
```

```
    }

}
```

Mecánica de colisiones

Los obstáculos, al acercarse, determinarán la posibilidad de colisionar con el personaje, acercándose desde el frente hacia atrás del personaje. El personaje, controlado por el jugador, debe evitar colisionar con los obstáculos.

Figura 8. Algoritmo visual – Función de colisión



Observe cómo programar colisiones.

c) El tercer comportamiento a programar es la colisión de Player con Enemy. Para ello, se interviene el script del Player (MovePlayer.cs) y se utiliza la función OnCollisionEnter, que mediante el parámetro collision, define qué respuesta debe darse a partir de colisionar con determinado objeto. En este caso,

(collision.tag==”Enemy”) se utiliza un condicional para delimitar la respuesta a la colisión, donde, si se colisiona con un objeto con determinada etiqueta (tag), deberá suceder el cambio de color del material de Player (se específica más adelante).

```
void OnCollisionEnter (Collision collision) {
            if(collision.gameObject.tag=="Enemy"){
                /* efecto */
            }
        }
```

d) El cuarto comportamiento a programar es la colisión de Enemy con Player. Para ello, se interviene el script de Enemy (MoveEnemy.cs) y se utiliza la función OnCollisionEnter, que mediante el parámetro collision, define qué respuesta debe darse a partir de colisionar con determinado objeto. En este caso, (collision.tag==”Player”) se utiliza un condicional para delimitar la respuesta a la colisión, donde, si se colisiona con un objeto con determinada etiqueta (tag), deberá generar un reseteo de la posición de Enemy, fuera del escenario y para que reinicie su movimiento hacia el personaje.

```
void OnCollisionEnter (Collision collision) {
            if(collision.gameObject.tag=="Player"){
               transform.position = new Vector3(Random.Range(-10.0f,
10.0f), 0.0f, 100.0f);
            }
```

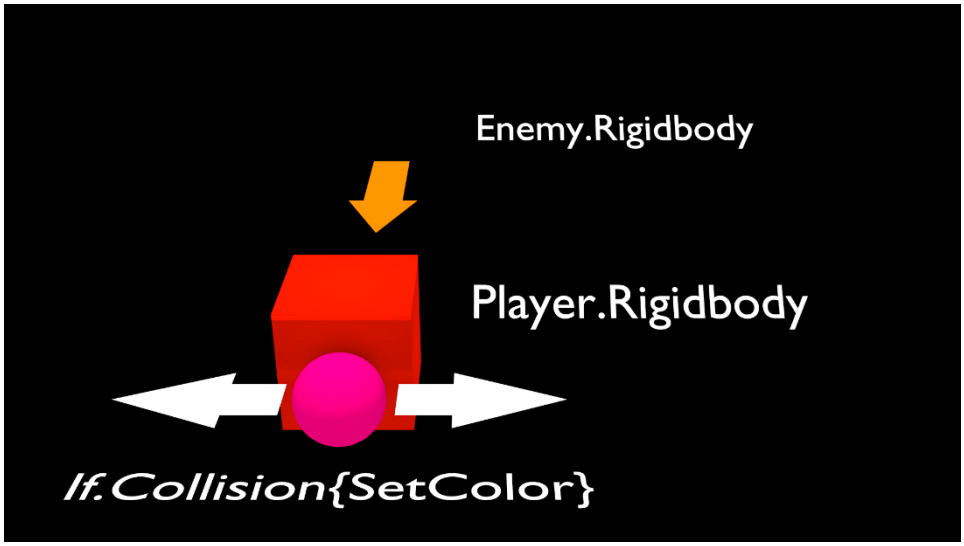
}

Mecánica de premio / castigo

En esta mecánica, si los obstáculos colisionan con el obstáculo, se tendrá un efecto de cambio de color en el personaje, indicando que ha colisionado

.

Figura 9. Algoritmo visual – Condicional de colisión



e) El quinto comportamiento a definir es lo que sucederá a Player una vez colisiona con el objeto “Enemy”.
   1. Se intervendrá el script de Player (MovePlayer.cs) y se creará una variable colorCollision, de tipo color, donde se almacenará información del color a cambiar (magenta) y una variable colorRend de tipo Renderer, que almacenará y obtendrá los atributos de material del objeto Player.
   2. En la función “Start”, se inicializa la variable colorRend, especificando el método GetComponent, para que, al iniciar el “script”, se tenga esta información en memoria.
   3. Se especifica el valor de asignación de la variable colorRend al color asignado en la variable colorCollision (colorRend.material.color=colorCollision).

4. Por último, se deja esta instrucción, como resultado en la condicional dispuesta en la función OnCollisionEnter.

Este video muestra cómo se programa la mecánica de premio y castigo.

```
void OnCollisionEnter (Collision collision) {
            if(collision.gameObject.tag=="Enemy"){
                colorRend.material.color = colorCollision;
            }
        }
```

Los “scripts” desarrollados deberán tener esta apariencia dentro del IDE:

Script MovePlayer.cs

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;

public class MovePlayer : MonoBehaviour
{
    public float velPlayer;
    public Color colorCollision;
    Renderer colorRend;
    void Start()
    {
```

```
            transform.position = new Vector3(0.0f, 0.0f, -6.0f);
            colorRend = GetComponent<Renderer> ();
        }
            void Update()
        {
            float movHorizontal = (Input.GetAxis("Horizontal") *
velPlayer * Time.deltaTime);

            transform.Translate(movHorizontal * Time.deltaTime, 0.0f,
0.0f);
        }

            void OnCollisionEnter (Collision collision) {
                if(collision.gameObject.tag=="Enemy"){
                    colorRend.material.color = colorCollision;
                }
            }
    }
```

Script MoveEnemy.cs

```
    using System.Collections;
    using System.Collections.Generic;
    using UnityEngine;

    public class MoveEnemy : MonoBehaviour
    {
         public float velEnemy;
        // Start is called before the first frame update
```

```
        void Start()
        {
            transform.position = new Vector3(Random.Range(-10.0f,
10.0f), 0.0f, 100.0f);
        }

        // Update is called once per frame
        void Update()
        {
            float posActual = transform.position.z;
            transform.Translate(0.0f, 0.0f, -velEnemy * Time.deltaTime);
            if(posActual <= -10.0f){
                transform.position = new Vector3(Random.Range(-10.0f,
10.0f), 0.0f, 100.0f);
            }
        }

        void OnCollisionEnter (Collision collision) {
             if(collision.gameObject.tag=="Player"){
                transform.position = new Vector3(Random.Range(-10.0f,
10.0f), 0.0f, 100.0f);
            }
        }
    }
```

3. Configuración de compilación y plataforma de publicación

Para la configuración de plataforma y jugador, se verán los aspectos esenciales a tener en cuenta para el proceso de configuración de compilación y plataforma de publicación.

3.1. Configuración de compilación

En el siguiente recuadro, se pueden observar los elementos fundamentales a configurar durante el proceso de compilación del videojuego. Cabe resaltar que es necesario determinar los valores de estas variables antes de iniciar con cualquier gestión dentro del proyecto de videojuego, es decir, en el momento de crear la carpeta del proyecto, se deben determinar de antemano estas características:

Tabla 2. Características

Configuración	Descripción
Target Platform	
Windows	Compilación para Windows
MacOS X (not available on Linux)	Compilación para MacOS
Linux (not available on Mac)	Compilación para Linux
Architecture/ Not available on macOS	
x86	32-bit CPU
x86_64	64-bit CPU

Configuración	Descripción
x86 + x86_64 (Universal)	All CPU devices for Linux
Server Build	Habilite esta configuración para crear el reproductor para uso del servidor y sin elementos visuales (sin cabeza), sin la necesidad de ninguna opción de línea de comando. Cuando habilita esta configuración, Unity crea scripts administrados con la definición de UNITY_SERVER, lo que significa que puede escribir código específico del servidor para su aplicación. También puede compilar la versión de Windows como una aplicación de consola para acceder a stdin y stdout. Los registros de Unity van a la salida estándar de forma predeterminada.
Copy PDB files (Windows only)	Habilite esta configuración para incluir archivos de base de datos de programas de Microsoft (.pdb) en el reproductor independiente integrado. Los archivos .pdb contienen información de depuración de la aplicación que puede utilizar para depurar su aplicación. La copia de archivos .pdb puede aumentar el tamaño de su reproductor, por lo que debe deshabilitar esta configuración para las compilaciones destinadas a la publicación. Esta

Configuración	Descripción
	configuración está deshabilitada de forma predeterminada.
Create Visual Studio Solution (Windows only)	Habilite esta configuración para generar archivos de la solución de Visual Studio para su proyecto, de modo que pueda crear su ejecutable final en Visual Studio.
Create Xcode Project (Mac Only)	Habilite esta configuración para generar un proyecto de Xcode para que pueda construir su paquete de aplicación final en Xcode. Xcode tiene soporte integrado para la firma de código y la carga de la aplicación en la Mac App Store.
Development Build	Habilite esta configuración para incluir símbolos de depuración de secuencias de comandos e incluir Profiler en su construcción. Cuando habilita esta configuración, se establece la definición de scripting Development_Build. Debe utilizar esta opción cuando desee probar su aplicación.
Autoconnect profiler	Requiere que se habilite la opción Development Build. Cuando se habilita esta configuración, Unity Profiler se conecta automáticamente a su compilación.
Deep Profiling Support	Requiere que se habilite la opción Development Build. Deep Profiling Support permite que Unity Profiler registre datos

Configuración	Descripción
	más detallados, instrumentando cada llamada de función. Habilitar la creación de perfiles profundos puede ralentizar la ejecución del script.
Script debugging	Requiere que se habilite la opción Development Build. Cuando se habilita esta configuración, Unity agrega símbolos de depuración a su código de secuencia de comandos.
Scripts Only Build	Requiere que se habilite la opción Development Build. Cuando habilita esta configuración, puede reconstruir solo los scripts para su aplicación, mientras deja intactos los archivos de datos de una compilación que ha ejecutado previamente. La compilación de scripts solo mejora significativamente los tiempos de iteración si solo está cambiando el código en su aplicación. Debe compilar todo el proyecto una vez, antes de poder utilizar esta configuración.

3.2. Configuraciones generales de jugador

En la siguiente tabla, está la información de las configuraciones generales de jugador en términos de las propiedades de pantalla:

Tabla 3. Configuraciones generales

Propiedad	Función
Fullscreen Mode	Elija el modo de pantalla completa. Esto define el modo de ventana predeterminado al inicio.
Fullscreen Window	Configure la ventana de su aplicación a la resolución nativa de pantalla completa de la pantalla. Unity procesa el contenido de la aplicación con la resolución establecida por el script (o por la selección del usuario cuando se inicia la aplicación construida), pero lo escala para llenar la ventana. Al escalar, Unity agrega barras negras a la salida renderizada para que coincida con la relación de aspecto elegido en la configuración del reproductor, para que el contenido no se estire. Este proceso se llama formato de pantalla ancha.
Exclusive Fullscreen (Windows only)	Configure su aplicación para mantener el uso exclusivo de pantalla completa de una pantalla. A diferencia de la ventana de pantalla completa, este modo cambia la resolución del sistema operativo de la pantalla para que coincida con la resolución elegida por la aplicación. Esta opción solo es compatible con Windows; en otras plataformas, la configuración vuelve a la ventana de pantalla completa.
Maximized Window (Mac only)	Configure la ventana de la aplicación con la definición del sistema operativo de "maximizada". En MacOS, esto significa una ventana de pantalla

	completa con una barra de menú y un muelle que se ocultan automáticamente. Esta opción solo es compatible con MacOS; en otras plataformas, la configuración vuelve a la ventana de pantalla completa.
Windowed	Configure su aplicación en una ventana móvil estándar, que no sea de pantalla completa, cuyo tamaño depende de la resolución de la aplicación. En este modo, la ventana se puede cambiar de tamaño de forma predeterminada. Para deshabilitar esto, deshabilite la configuración de Ventana de tamaño variable.
Default Is Native Resolution	Habilite esta opción para que el juego use la resolución predeterminada utilizada en la máquina de destino. Esta opción no está disponible si el Modo de pantalla completa está configurado en Ventana.
Default Screen Width	Establecer el ancho predeterminado de la pantalla del juego en píxeles. Esta opción solo está disponible si el Modo de pantalla completa está configurado en Ventana.
Default Screen Height	Establece la altura predeterminada de la pantalla del juego en píxeles. Esta opción solo está disponible si el Modo de pantalla completa está configurado en Ventana.
Mac Retina Support	Habilite esta opción para habilitar la compatibilidad con pantallas de alta DPI (Retina) en Mac. Unity lo habilita de forma predeterminada. Esto mejora los

	proyectos en una pantalla Retina, pero consume algo de recursos cuando está activo.
Run In background	Habilite esta opción para que el juego siga funcionando (en lugar de pausar) si la aplicación pierde el foco.
Capture Single Screen	Habilite esta opción para asegurarse de que los juegos independientes, en el modo de pantalla completa, no oscurezcan el monitor secundario en configuraciones de varios monitores. Esto no es compatible con Mac OS X.
Use Player Log	Habilite esta opción para escribir un archivo de registro con información de depuración. Predeterminado a habilitado. Advertencia: si planea enviar su solicitud a la Mac App Store, deje esta opción deshabilitada. Para obtener más información, consulte Publicación en Mac App Store.
Resizable Window	Habilite esta opción para permitir cambiar el tamaño de la ventana del reproductor independiente. Nota: si desactiva esta opción, su aplicación no podrá utilizar el modo de pantalla completa con ventana.
Visible in Background	Habilite esta opción para mostrar la aplicación en segundo plano si se utiliza el modo de pantalla completa con ventana (en Windows).
Allow Fullscreen Switch	Habilite esta opción para permitir que las pulsaciones predeterminadas de las teclas de pantalla

	completa del sistema operativo cambien entre los modos de pantalla completa y ventana.
Force Single Instance	Habilite esta opción para restringir los reproductores independientes a una sola instancia en ejecución simultánea.
Supported Aspect Ratios	Habilite cada relación de aspecto que desee que aparezca en el cuadro de diálogo Resolución al inicio (siempre que sean compatibles con el monitor del usuario).

3.3. Características de jugador en términos de renderizado

Tabla 4. Características en renderizado

Propiedad	Función
Color Space	Elija qué espacio de color se debe utilizar para el renderizado: Gamma o Lineal.

Auto Graphics API for Windows	Habilite esta opción para usar la mejor API de gráficos en la máquina con Windows en la que se ejecuta el juego. Deshabilítela para agregar y eliminar API de gráficos compatibles.
Auto Graphics API for Mac	Habilite esta opción para usar la mejor API de gráficos en la Mac en la que se ejecuta el juego. Deshabilítela para agregar y eliminar API de gráficos compatibles.
Auto Graphics API for Linux	Habilite esta opción para usar la mejor API de gráficos en la máquina Linux en la que se ejecuta. Deshabilítela para agregar y eliminar API de gráficos compatibles.
Color Gamut for Mac	Puede agregar o eliminar gamas de colores para que las plataformas Mac independientes las utilicen para la renderización. Haga clic en el icono más (+) para ver una lista de las gamas disponibles. Una gama de colores define una posible gama de colores disponibles para un dispositivo determinado (como un monitor o una pantalla). La gama sRGB es la gama predeterminada (y obligatoria).
Metal Editor Support	Habilite esta opción para usar la API de Metal en Unity Editor y desbloquee una iteración de Shader más rápida para apuntar a la API de Metal.
Metal API Validation	Habilite esta opción cuando necesite depurar problemas de Shader. Nota: La validación aumenta el uso de la CPU, por lo tanto, úsela solo para depurar.

Metal Write-Only Backbuffer	Permitir un rendimiento mejorado en la orientación del dispositivo no predeterminada. Esto establece el indicador frameBufferOnly en el búfer de retroceso, lo que evita la lectura del búfer de retroceso pero habilita cierta optimización del controlador.
Memoryless Depth	Elija cuándo usar texturas de renderizado sin memoria. Las texturas de renderizado sin memoria se almacenan temporalmente en la memoria del mosaico cuando se renderizan, no en la memoria de la CPU o GPU. Esto reduce el uso de memoria de su aplicación, pero no puede leer ni escribir en estas texturas de renderizado. Nota: Las texturas de renderizado sin memoria solo son compatibles con iOS, tvOS 10.0+ Metal y Vulkan. Las texturas de renderizado están protegidas contra lectura/escritura y se almacenan en la memoria de la CPU o GPU en otras plataformas.
Unused	Nunca use la profundidad del búfer de cuadros sin memoria.
Forced	Utilice siempre la profundidad de framebuffer sin memoria.
Automatic	Deje que Unity decida cuándo utilizar la profundidad del búfer de tramas sin memoria.

Static Batching	Habilite esta opción para utilizar el procesamiento por lotes estático.
Dynamic Batching	Habilite esta opción para usar Dynamic Batching en su compilación (habilitada de forma predeterminada). Nota: el procesamiento por lotes dinámico no tiene ningún efecto cuando una canalización de procesamiento de secuencias de comandos está activa, por lo que esta configuración solo es visible cuando no se establece nada en la configuración Gráficos de activos de canalización de procesamiento de secuencias de comandos.
Compute Skinning	Habilite esta opción para habilitar la máscara de cómputo de la GPU DX11 / ES3, liberando recursos de la CPU.
Graphics Jobs	Habilite esta opción para indicar a Unity que descargue tareas de gráficos (bucles de procesamiento) en subprocesos de trabajo que se ejecutan en otros núcleos de CPU. Esto tiene como objetivo reducir el tiempo que se pasa en Camera.Render en el hilo principal, que a menudo es un cuello de botella.
Lightmap Encoding	Elija Calidad normal o Alta calidad para configurar la codificación del mapa de luz. Esta configuración afecta el esquema de codificación y el formato de compresión de los mapas de luz.

Lightmap Streaming	Habilite esta opción para cargar solo los mapas mip de mapa de luz según sea necesario para representar las cámaras del juego actual. Este valor se aplica a las texturas del mapa de luz a medida que se generan. Nota: Para utilizar esta configuración, debe habilitar la configuración de Calidad de transmisión de texturas.
Streaming Priority	Establezca la prioridad de transmisión del mapa mip de mapa de luz para resolver conflictos de recursos. Estos valores se aplican a las texturas del mapa de luz a medida que se generan. Los números positivos dan mayor prioridad. Los valores válidos oscilan entre –128 y 127.
Frame Timing Stats	Habilite esta propiedad para recopilar estadísticas de tiempo de trama de CPU y GPU. Use esto junto con la configuración de la cámara de resolución dinámica para determinar si su aplicación está vinculada a la CPU o GPU.
Use Display In HDR Mode (Windows Only)	Habilite esta casilla de verificación para que el juego cambie automáticamente a la salida del modo HDR cuando se ejecute. Esto solo funciona en pantallas que admitan esta función. Si la pantalla no admite el modo HDR, el juego se ejecuta en modo estándar.
Swap Chain Bit Depth	Seleccione el número de bits en cada canal de color para los búferes de cadena de intercambio. Solo disponible si el modo HDR está habilitado.

Bit Depth 10	Unity utilizará el formato de búfer R10G10B10A2 y primarios Rec2020 con codificación ST2084 PQ.
Bit Depth 16	Unity utilizará el formato de búfer R16G16B16A16 y primarios Rec709 con color lineal (sin codificación).

4. Diseño y construcción de prototipo físico de videojuegos

En este apartado se verá los modelos de mecánica de videojuegos y los paradigmas de la jugabilidad

4.1. Modelos de mecánica de videojuegos

A continuación, se describirán algunos modelos de mecánicas de videojuego, donde se determinan diferentes elementos y, por consiguiente, conceptualmente diferentes formas de allanar grupos o estructuras de mecánica de juegos. Sin embargo, ofrecen una manera amplia de analizar y determinar los aspectos de jugabilidad del videojuego a diseñar.

a) **Modelo de Jesse Schell:** en este modelo se especifican los siguientes elementos o tipologías de mecánicas básicas:
   1. El espacio del juego: comprende fundamentalmente el entorno espacial (arquitectura espacios naturales) donde se desarrolló el videojuego y donde tiene lugar toda la serie de eventos acciones en que el personaje interactúa.
   2. Los objetos que hay en el videojuego: en este esquema, se trata precisamente de todos los objetos con los cuales pueden interactuar los personajes del juego. En esencia, se trata de elementos que pueden determinar el triunfo o fracaso del jugador, que pueden facilitar el logro de

objetivos o metas, o constituirse en obstáculos para la consecución de objetivos.

3. Las acciones del jugador: en este caso se trata de las mecánicas de núcleo, es decir, lo que en esencia debe realizar el jugador para ganar el juego. Generalmente, pueden describirse a manera de verbos como caminar, correr, saltar.
4. Las habilidades del jugador: si bien la habilidad de un jugador no está determinada ni forma parte de manera estricta con el proceso de diseño del videojuego, sí puede establecerse, a partir de determinados esquemas de programación, como una medición del rendimiento del jugador en el juego, o desde una perspectiva más simple, definir pautas de progresión en la dificultad del juego en la medida en que el jugador demuestra cierto nivel de aprendizaje y habilidad.
5. El chance: entendido como elementos, circunstancias que pueden facilitar el logro de determinado reto.

b) **Modelo de David Parlett:** en este modelo, se especifican los siguientes elementos o tipologías de mecánicas básicas:

1. Reglas operacionales: comprende las reglas que determinan la manera como opera y funciona de manera esencial del videojuego.
2. Reglas de comportamiento: especifican la manera como ha de comportarse el jugador, aplicable en el caso de algunos juegos de rol o en su representación el personaje de acuerdo a los objetivos del juego y el rol actancial del personaje.
3. Reglas escritas: se refiere a las reglas que pueden estar definidas de manera arbitraria y que pueden encontrarse en manuales de usuario del videojuego.
4. Reglas oficiales: puede entenderse como regla oficial aquella que está integrada directamente en el videojuego y que ha sido aceptada por la comunidad de jugadores.
5. Reglas de advertencia: se refiere específicamente a aquellas reglas que procuran un modelo de conducta o de desenvolvimiento del juego, tratando

de minimizar acciones que limiten o perjudiquen el correcto desarrollo del juego.

4.2. Paradigmas de jugabilidad

Las configuraciones y acciones de jugabilidad son diferentes y específicas a cada videojuego, sin embargo, se pueden encontrar aspectos comunes y que definen comportamientos, al margen de lo que ya establecen las dinámicas y las mecánicas del juego.

a) Toma de decisiones: los jugadores se enfrentan a cada momento del juego a una serie de circunstancias que, de acuerdo con las mecánicas de jugabilidad, definen opciones de acción que, en sí mismas, son parámetros de decisiones. A continuación, se definen las más preponderantes:

1. Intercambio: Esquema de decisión donde el jugador tiene la alternativa de elegir el cambio o sustitución de un elemento de juego.
2. Persistencia: Esquema de decisión donde el jugador debe mantenerse en un evento de juego, a partir de una decisión tomada y como consecuencia.
3. Situacional: Esquema de decisión donde las situaciones definen la pertinencia o conveniencia de una determinada decisión.
4. Personalización: Esquema de decisión donde el jugador tiene la oportunidad de diseñar un estilo propio de juego, basado en un perfil de decisiones.

b) Desafíos y mecánicas: las mecánicas definen el funcionamiento de los desafíos, la forma como se activan, las condiciones que las delimitan. En términos generales, los esquemas de desafío pueden ser activos o pasivos.

1. Desafíos pasivos. Son aquellos que comprenden la interacción con elementos ya presentes en el nivel de juego y, por lo tanto, son objetos de juego físicos del nivel, por ejemplo, una pared de un edificio donde el desafío es escalarla.
2. Desafíos activos. Pueden denominarse desafíos propiamente dichos, y están delimitados por las reglas y mecánicas del juego. El combate y derrota con entidades de AI son un ejemplo de este tipo de desafíos.

c) Acciones y desafíos: Las acciones que los jugadores realizan se deciden con el fin de cumplir con los desafíos que se presentan en juego. Un mismo desafío puede sugerir una serie específica de acciones o la elección variable de multitud de opciones. Existen dos tipos de acciones desde el enfoque de lo que pueden hacer los jugadores:
   1. Acciones operativas: Son las acciones que el jugador puede realizar, y forman parte de su mecánica de núcleo. Por ejemplo, en Mario Bros, el jugador puede caminar, correr y saltar.
   2. Acciones resultantes: Son aquellas acciones que tienen un valor significativo, con relación al cumplimiento de determinados desafíos. De una acción operativa, pueden derivarse diferentes acciones resultantes. Por ejemplo, de la acción saltar, puede derivarse una acción resultante de saltar encima de un enemigo, que puede dar lugar a su neutralización, que sería diferente a una acción resultante de saltar sobre un obstáculo como una roca o un muro. Las acciones resultantes sugieren interacciones más específicas con el contexto de juego y surgen de las particularidades de cada mecánica de videojuego.

5. Tipologías de prototipado

El propósito principal de los esquemas de prototipado es poder evidenciar y probar la funcionalidad del videojuego y cómo esta funcionalidad puede involucrar de manera específica las mecánicas o de manera global todos los aspectos de jugabilidad. Incluso,

puede apenas relacionarse con aspectos visuales, como los escenarios y su apariencia y el recorrido delimitado para el jugador.

5.1. Prototipo físico

En primer lugar, tenemos los prototipos físicos, que tienen como ventaja el poder enfocarse en la jugabilidad, sin concentrarse en la tecnología o el aspecto visual. Son relativamente de bajo costo, teniendo en cuenta que puede recurrirse casi que a cualquier elemento físico para poder utilizarlo como componente del prototipo (juguetes, dados, cartas). Permite hacer ajustes a las mecánicas sin pensar en su programación y obviamente sin utilizar una sola línea de código. Permite evaluar aspectos generales o específicos y solucionarlos de manera pronta. Puede intervenir todo el equipo de trabajo, aportando cada uno desde su área.

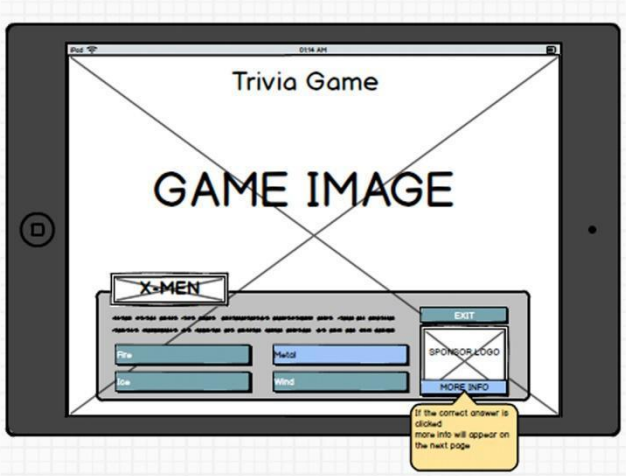
Para la elaboración de un prototipo físico se deben tener en cuenta los siguientes aspectos conceptuales:

1. Fundamentos: se refiere principalmente a la evaluación de las mecánicas de núcleo, y para ello se pueden utilizar cartas, dados, papel, con el fin de replicar las mecánicas básicas del juego.
2. Estructura: en este aspecto es necesario replicar y evaluar la forma como se avanza en el juego, el poder de ataque y todo aquello que permita comprender la manera como se actúa en el juego.
3. Detalles formales: en este punto, se escriben las reglas y procesos que dan lugar a la normativa del juego y el cómo debe ser jugado.
4. Refinamiento: es el objetivo final y fundamental de un prototipo físico, en tanto que permite encontrar errores y determinar qué mecánicas funcionan correctamente y cuáles deben ser modificadas, al igual que mejorar las reglas y procesos generales de jugabilidad.

a) Modelo wireframing

Tiene por objeto mostrar la funcionalidad del flujo de información y pantallas, enfocándose en las propiedades de la interfaz de usuario y los elementos gráficos del videojuego, en términos de comprensión de su información inherente y las condiciones de interacción con el jugador.

Figura 10. Wireframe de interfaz de videojuego Marvel Trivia



b) Modelo maqueta

El prototipado físico basado en maqueta tiene por objeto evaluar las mecánicas de juego, utilizando elementos físicos comunes, como implementos de papelería, y elementos de juego tradicional, como dados o mazos de cartas.

Por otro lado, este tipo de modelo puede aplicarse para la construcción de elementos estructurales del videojuego, como vehículos, escenarios, o utilería. Incluso,

pueden llegar a construirse los escenarios, para, a través de ellos, evaluar con mayor detalle determinadas mecánicas.

Figura 11. Maqueta de prototipo físico Videojuego Ivarok

c) Storyboard experiencial

Si bien un guion gráfico no se considera como un prototipo físico en sentido estricto, se trata de utilizar modelos de Role Play unidos a la metodología de guion gráfico, que puedan representar de manera física, visual y dramatizada la experiencia de usuario y, sobre todo, las connotaciones posibles que pueda llegar a tener el jugador al momento de jugar.

5.2. Prototipo digital

Los prototipos digitales sugieren la construcción de una versión del videojuego de manera digital, mínimamente programada y codificada, donde se hace énfasis en aspectos relativos a las mecánicas de juego, la estética, la kinestesia y la tecnología. Puede desarrollarse con los mínimos elementos y detalles posibles y apenas con el objetivo de verificar y evaluar la funcionalidad.

Figura 12. Prototipo de videojuego realizado con Game Builder de Google

Los aspectos importantes de prototipado a este nivel son:

1. Prototipar las mecánicas de juegos: algunas mecánicas pueden ser más fáciles de comprender y de evaluar cuando se analizan y se desarrollan en un sistema digital que en uno físico.
2. Prototipar la estética del juego: Se enfoca fundamentalmente en la apariencia visual y gráfica del videojuego. En este tipo de prototipado, se trata de representar de manera simple las gráficas del juego. Puede ayudar a comprender el funcionamiento del juego y, entre otros, la manipulación y sentido de la interfaz de usuario.

3. Prototipar la kinestesia del juego: En este punto, se trata de reproducir la forma en la que el juego funciona y determinar cómo es el comportamiento corporal esperado en un jugador, bien sea lo que se refiere a las reacciones o a la interacción con dispositivos de control.
4. Prototipar la tecnología: En este último aspecto, se trata de evaluar la manera como la kinestesia se desenvuelve de acuerdo con un determinado tipo de tecnología, y cómo favorece o perjudica las mecánicas de juegos. Por ejemplo, simular el juego a través de una pantalla táctil o con `joystick`.

Síntesis

Con el diagrama de actividades se termina la descripción de las tres técnicas usadas para la especificación y el análisis de requisitos, como se pudo analizar cada una de las técnicas nos brinda características para lograr representar gráficamente y de una forma clara, los requisitos que un usuario necesita poder cumplir satisfactoriamente con las solicitudes de los clientes y usuarios, por otro lado se pueden usar todas las técnicas en una especificación y análisis de requisitos si se ve la necesidad.

A continuación, se muestra un mapa conceptual con los elementos más importantes desarrollados en este componente.

Glosario

Asset: cualquier recurso susceptible de ser usado en los juegos y que está contenido en la carpeta de proyecto. Puede provenir de un archivo externo creado en un programa diferente de Unity, como un archivo 3D, un archivo de formato de audio o imagen. También se pueden crear algunos tipos de assets en Unity, como un Animator Controller, un Audio Mixer o una Render Texture.

Asset package: es un conjunto de archivos recursos del proyecto de Unity, que se empaquetan de forma comprimida en un archivo similar a los archivos Zip, con la extensión unitypackage. Los paquetes de recursos son una forma práctica de compartir y reutilizar proyectos y colecciones de activos de Unity.

Asset Server Legacy: un sistema de control de activos y versiones, con una interfaz gráfica de usuario integrada en Unity. Permite a los miembros del equipo trabajar juntos en un proyecto en diferentes computadoras.

Asset Store: es una biblioteca donde pueden encontrarse recursos gratuitos o comerciales creados por la comunidad de Unity, organizada por categorías y que abarca diferentes tipos de medios para integrar de manera automática a un proyecto de videojuego.

Asset Store package: colección agrupada de activos disponibles para comprar o descargar en Unity Asset Store, comprimidos y almacenados en un archivo con la extensión. unitypackage, como un paquete de activos. Puede administrar sus paquetes de la tienda de activos en la tienda en línea o a través de la ventana Administrador de paquetes.

Audio Clip: un contenedor para datos de audio en Unity. Unity admite activos de audio mono, estéreo y multicanal (hasta ocho canales). Unity puede importar formatos de archivo de audio .aif, .wav, .mp3 y .ogg, y formatos de módulo de seguimiento .xm, .mod, .it y .s3m.

Category: categoría Profiler identifica los datos de la carga de trabajo para un subsistema de Unity (por ejemplo, las categorías Rendering, Scripting y Animation). Unity aplica códigos de colores a las categorías para ayudar a distinguir visualmente los tipos de datos en la ventana Profiler.

Compression : es un método de almacenamiento de datos que reduce la cantidad de espacio de almacenamiento que requiere. Consulte Compresión de textura, Compresión de animación, Compresión de audio, Compresión de compilación.

Frustum: es la región del espacio 3D que una cámara en perspectiva puede ver y renderizar. En la vista de Escena, el tronco está representado por una pirámide rectangular truncada con su parte superior en el plano de recorte cercano de la cámara, un plano que limita qué tan cerca puede ver una cámara desde su posición actual. El plano es perpendicular a la dirección de avance (Z) de la cámara.

GameObject: es el objeto fundamental en las escenas de Unity, que puede representar personajes, accesorios, escenarios, cámaras, waypoints y más. La funcionalidad de un GameObject está definida por los Componentes adjuntos a él.

Level of Detail: la técnica de nivel de detalle (LOD) es una optimización que reduce la cantidad de triángulos que Unity tiene que representar para un GameObject cuando aumenta su distancia de la cámara.

Model: es una representación de modelo 3D de un objeto, como un personaje, un edificio o un mueble.

Model file: un archivo que contiene datos 3D, que pueden incluir definiciones de mallas, huesos, animación, materiales y texturas.

Package: es un contenedor que almacena varios tipos de características y activos para Unity, incluidas las herramientas y bibliotecas de Editor o Runtime, colecciones de activos y plantillas de proyectos. Los paquetes son unidades independientes que Unity Package Manager puede compartir entre los proyectos de Unity. La mayoría de las veces se denominan paquetes, pero ocasionalmente se denominan paquetes Unity Package Manager (UPM).

Prefab: es un tipo de recurso que le permite contener un GameObject completo, con componentes y propiedades. Este tipo de recurso puede ser utilizado una y otra vez a través de copias o instancias.

Profiler: es la ventana que te ayuda a optimizar tu juego. Muestra cuánto tiempo se pasa en las distintas áreas de su juego. Por ejemplo, puede informar el porcentaje de tiempo dedicado a renderizar, animar o en la lógica de su juego.

Project: en Unity, se usa un proyecto para diseñar y desarrollar un juego. Un proyecto almacena todos los archivos relacionados con un juego, como el activo y los archivos de escena.

Project Settings: amplia colección de configuraciones que le permiten configurar cómo se comportan Física, Audio, Redes, Gráficos, Entrada y muchas otras áreas de su proyecto.

Sprite Atlas: textura que se compone de varias texturas más pequeñas. También se conoce como atlas de texturas, sprite de imagen, hoja de sprite o textura empaquetada.

State Machine: es el conjunto de estados en un Animator Controller en el que puede estar un personaje o GameObject animado, junto con un conjunto de transiciones entre esos estados y una variable para recordar el estado actual. Los estados disponibles dependerán del tipo de juego, pero los estados típicos incluyen cosas como inactivo, caminar, correr y saltar.

State Machine Behaviour: es la secuencia de comandos que se adjunta a un estado dentro de una máquina de estado para controlar lo que sucede cuando la máquina de estado entra, sale o permanece dentro de un estado, como reproducir sonidos cuando se ingresan los estados.

Referencias bibliográficas

Brathwaite, B. y Schreiber, I. (2009). *Challenges for Game Designers*. Charles River Media.

Gibson, J. (2014). *Introduction to Game Design, Prototyping, and Development: From Concept to Playable Game with Unity and C#*. Addison-Wesley Educational Publishers Inc.

Hartt, C. (2019). *Connecting Values to Action: Non-Corporeal Actants and Choice*. Emerald Publishing Limited.

Mernard, M. (2011). *Game Development with Unity*. Cengage Learning PTR.

Okita, A. (2014). *Learning C# Programming with Unity 3D*. Routledge.

Créditos

Nombre	Cargo	Regional y Centro de Formación
Claudia Patricía Aristizábal Gutiérres	Responsable del equipo	Dirección General
Liliana Victoria Morales Gualdrón	Responsable de línea de producción	Centro de Gestión De Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Xiomara Becerra Aldana	Instructora Ambiental	Centro de Gestión Industrial
Lubin Andrés Hernández Sanabria	Instructor	Centro de Gestión Industrial
Jesús Ricardo Arias Munevar	Instructor	Centro de Gestión Industrial
Javier Ricardo Luna Pineda	Diseñador Instruccional	Centro para la Industria de la Comunicación Gráfica
Silvia Milena Sequeda Cárdenas	Evaluador Instruccional	Centro de diseño y metrología
Rafael Neftali Lizcano Reyes	Asesor Pedagógico	Centro Industrial del Diseño y la Manufactura
Martha Isabel Martínez Vargas	Productora audiovisual	Centro Industrial del Diseño y la Manufactura
Gloria Amparo López Escudero	Adecuadora instruccional - 2023	Centro de Gestión de Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Andrés Felipe Velandia Espitia	Metodólogo para la formación virtual - 2023	Centro de Gestión de Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Alix Cecilia Chinchilla Rueda	Asesor metodológico	Centro de gestión de Mercados, Logística y Tecnologías de la Información

Yuly Andrea Rey Quiñonez	Diseñadora web	Centro de Gestión De Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Manuel Felipe Echavarria Orozco	Desarrollador Fullstack	Centro de Gestión De Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Laura Gisselle Murcia Pardo	Animador y Producción audiovisual	Centro de Gestión De Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Carolina Coca Salazar	Evaluadora de contenidos inclusivos y accesibles	Centro de Gestión De Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Lina Marcela Pérez Manchego	Validadora de recursos educativos	Centro de Gestión De Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital
Leyson Fabian Castaño Pérez	Validadora de recursos educativos	Centro de Gestión De Mercados, Logística y Tecnologías de la Información - Regional Distrito Capital