Universidad Central de Venezuela
Facultad de Ciencias
Escuela de Computación
Centro de Computación Gráfica

Generación Procedimental de Contenido para videojuegos

Trabajo Especial de Grado presentado ante la Ilustre
Universidad Central de Venezuela por la
Br. Sarah Dresden Fernández
para optar al título de Licenciado en Computación

Tutor
Prof. Hector Navarro

Caracas, Mayo 2018

Universidad Central de Venezuela

Facultad de Ciencias Escuela de Computación

Centro de Computación Gráfica

ACTA DEL VEREDICTO

Quienes suscriben miembros del Jurado, designado por el Consejo de Escuela de Computación, para examinar el Trabajo Especial de Grado presentado por la **Br. Sarah Dresden, C.I. 20872585**, titulado: "**Generación Procedimental de Contenido para videojuegos**" a los fines de optar al título de Licenciado en Computación, dejan constancia lo siguiente:

Leído como fue, dicho trabajo por cada uno de los miembros del Jurado, se fijó el día 24 de mayo del 2018 a las 2:00 p.m., para que su autor lo defendiera en forma pública, en el Centro de Computación Gráfica, Facultad de Ciencias, mediante una presentación oral de su contenido. Finalizada la defensa pública del Trabajo Especial de Grado, el Jurado decidió aprobarlo.

En fé de lo cual se levanta la presente Acta, en Caracas a los veinticuatro días del mes de mayo del año dos mil dieciocho, dejándose también constancia de que actuó como Coordinador del Jurado el Prof. Héctor Navarro.

Prof. Héctor Navarro, Tutor

Prof. Walter Hernández, JuradoPrincipal

Prof. Francisco Sans, Jurado Principal

Resumen

En la actualidad las compañías de videojuegos pueden tener hasta cientos de diseñadores, artistas, y programadores trabajando para un mismo juego, lo que implica un gasto de recursos significativo. Para tener una ventaja competitiva con empresas con grandes capitales se utilizan los métodos de Generación Procedimental de Contenido (*PCG, Procedural content generation*), de forma que se puedan reemplazar algunos de los artistas y diseñadores por algoritmos.

En este trabajo de investigación se crearon algoritmos que, con la utilización de las gramáticas, aplican los métodos PCG, para la creación de videojuegos con aumento de dificultad por nivel, específicamente en la generación de obstáculos.

Para verificar la eficiencia de estos algoritmos, se creó un caso de estudio de un videojuego 2D con tres niveles de dificultad, donde el objetivo es recorrer la mayor distancia posible superando los distintos obstáculos. Posteriormente se realizó una encuesta para obtener resultados de origen cuantitativo y cualitativo, donde los usuarios evaluaron tres versiones distintas del videojuego que fueron implementadas. A partir de estos resultados se logró probar la efectividad del método implantado, y se generó la versión óptima del método de PCG que se desarrolló.

Con estas técnicas de desarrollo se logró tener un juego entretenido, y siempre distinto, sin la necesidad de un diseñador por cada versión que sea creada, ahorrando así espacio de almacenamiento, y garantizando la jugabilidad de este.

Como trabajo a futuro se recomienda convertir estos algoritmos en una biblioteca que pueda ser usada por desarrolladores en la programación de otros juegos.

Palabras clave: Generación Procedimental de Contenido, videojuego, algoritmos, gramática, dificultad, acciones, jugabilidad.

A mis padres por su amor y apoyo incondicional en todo momento, por hacerme ver la importancia de la educación, no como algo obligatorio sino placentero y estimulante, permitiéndome conseguir desde pequeña esa ilusión por el aprendizaje. Agradecida por tenerlos a mi lado y ser mi gran inspiración en cada decisión que tomo, este logro es para ustedes, los quiero muchísimo.

Agradecimientos

En primer lugar le quiero agradecer a mis padres, a mi hermano y a mi OMA quienes con su esfuerzo, amor y apoyo, me han hecho la persona que soy hoy en día. Por motivarme en los días más difíciles, y siempre tener las palabras perfectas en cualquier situación. Porque esta carrera la viví junto a ustedes, emociones, nervios, motivación, aprendizajes, diligencias, decisiones porque cada sentimiento lo hicieron parte de ustedes también. Gracias, muchas gracias por ser mi fuente de inspiración, y mi mayor alegría. Gracias a mi abuelo, General Victor José Fernández Bolívar, porque aún con sus 84 años recuerda y se enorgullece de mis logros y motivado quiere asistir a mi graduación.

A mi tutor Héctor Navarro, por primero haber sido un excelente profesor que me inspiró a mantenerme en la rama de computación gráfica; por ser mi tutor y responder cualquier duda, una y otra vez, por su paciencia y ganas de entender mis inquietude y por su apoyo no solo en lo universitario sino en lo laboral.

A los jurados de mi seminario, Walter Hernández y Francisco Sans, por sus excelentes correcciones y críticas constructivas, por su dedicación y tiempo invertido en leer el trabajo y evaluarme.

A la Facultad de Ciencias de la Universidad Central de Venezuela y todos sus profesores que me inspiraron a enamorarme de la ciencia aún mas, profesores del área de Computación Biología, Química, Física y Matemática, a todos los estudiantes que tuve el placer de tratar que me enseñaron en mi día a día de vida universitaria un poco sobre sus carreras. Un placer haber formado parte de una facultad tan integral y llena de conocimientos esenciales para la vida.

A CECOBIO, Centro Excursionista y Conservacionista de Biología, por permitirme formar parte de este dándome la oportunidad de tener un hobbie dentro de la facultad, que luego se convirtió en una pasión en mi vida, la escalada y la montaña, dónde hice también a los mejores amigos.

Por último a mis amigos más cercanos Marubetsy Alcina, David Hernández, Julio Churio, Napoleón Malpica, Jhonantan Miranda y Rebeca Gutierrez con los que he tenido el placer de disfrutar de la ciencia, de la vida, y contar con su apoyo en toda ocasión.

Índice general

Índice de figuras

Lista de códigos

Índice de cuadros

Introducción

Desde que los videojuegos se inventaron, el número de personas que se dedican al desarrollo de un juego exitoso y comercial, ha ido aumentando considerablemente. Es común para un juego que sea desarrollado por cientos de personas en un período de un año o más. Esto lleva a una situación donde pocos juegos son portables, y pocos desarrolladores pueden sustentar los costos del desarrollo de un juego. Muchos de los empleados más costosos necesarios en este proceso, son los diseñadores y artistas más que los programadores [1].

En el 2005 en La Conferencia de desarrolladores, Will Wright, un diseñador legendario de videojuegos como SimCity y Los Sims, dijo que las compañías de desarrollo, que puedan reemplazar algunos de los artistas y diseñadores por algoritmos, podrían tener una ventaja competitiva, ya que el juego se desarrollaría más rápido y con menos gastos, preservando su calidad. El método *PCG, Procedural content generation* [1], se puede usar como herramienta para aumentar la creatividad de los creadores, ya que los algoritmos crean automáticamente contenido quizás diferente al que un diseñador crearía. Éste método permite generar nuevos tipos de juegos, por ejemplo si se puede generar contenido en tiempo real, no habría razón para que el juego tenga un final. Otra ventaja del desarrollo de PCG, desde el punto de vista del programador, es entender el diseño de los videojuegos al tener que crear un código que diseñe este por sí mismo.

La limitación de memoria en las computadoras en los años ochenta fue unas de las razones que impulsó al desarrollo de las técnicas PCG, forzando a los diseñadores a buscar otro método para crear contenido sin ser almacenado. Uno de los primeros juegos utilizando esta técnica fue Rogue, el que posteriormente marca un estilo, juegos "Rogue-like", genera su propio contenido aleatorio a medida que se juega, y al momento de perder comienza un juego totalmente distinto desde el principio. Este tipo de juegos carecen de contenido visual atractivo.

Un juego reciente y nombrado fue "No Man's Sky", publicado en el año 2016, el cual utilizó la generación procedimental de contenido como método de desarrollo. Cada jugador tiene la posibilidad de descubrir planetas nuevos, con distinta flora y fauna en cada uno de ellos, lo cual da a lugar la posibilidad de explorar un universo prácticamente infinito (más de 18 trillones de planetas). Sin embargo las críticas de

parte de los jugadores no fueron positivas, considerándolo una experiencia repetitiva

El objetivo principal del método PCG es crear contenido automáticamente, a partir de parámetros fijos o variables dados por el usuario o el diseñador del juego. De esta manera, al emular un diseñador humano, se evita depender del mismo. Es conveniente ajustar los parámetros de la PCG para obtener resultados más idóneos en el desarrollo del producto final.

Por lo tanto se plantea en este trabajo el desarrollo de un algoritmo que facilite la creación de un videojuego 2D con niveles no repetidos en distintas instancias, sin la necesidad de un diseñador. Se utilizará la gramática para garantizar que este pueda ser jugado, sin dejarlo al azar, en conjunto con técnicas de ritmo para darle dinamismo.

En el presente documento se aborda la investigación para el diseño e implementación de un algoritmo utilizado para generar niveles en un juego 2D, para esto se describen un conjunto de conocimientos relacionados a los términos, y técnicas empleadas. Posteriormente, se expone la implementación hecha exhibiendo cada componente, como también las convenciones empleadas en éstos. Luego se analizan las pruebas y resultados obtenidos para, de este modo, evaluar la efectividad del método implantado y poder seleccionar los valores óptimos del mismo. Finalmente se exponen las conclusiones de la presente investigación y propuestas de trabajos futuros.

Objetivos

Objetivo general

Diseño e implementación de un método basado en gramáticas para la generación procedimental de niveles de videojuegos.

Objetivos específicos

- Diseñar e implementar una gramática, que sea utilizada para la generación procedimental de contenido, basado en el trabajo de Smith [6]
- Diseñar un mecanismo para hacer correspondencia (“mapear”) entre el resultado de la gramática y el contenido real del juego.
- Establecer los valores óptimos para los parámetros del algoritmo.

- Implementación de un caso de estudio con niveles generados procedimentalmente.
- Realizar pruebas cualitativas y cuantitativas para evaluar la efectividad del método implantado.

1 Marco Teórico

1.1. Generación Procedimental de Contenido (*PCG*)

1.1.1. Definición

La generación procedimental de contenido (*PCG, Procedural content generation*) específicamente en el tema de los videojuegos, se refiere a la creación automática de contenido, utilizando algoritmos que le faciliten este proceso a los involucrados. Esta generación no es una tarea fácil, requiere satisfacer las restricciones del artista, y también retornar instancias interesantes para el jugador, con todo lo que esto implique. El contenido a generar se dice ser "pseudo aleatorio" ya que se tiene que asegurar que este sea apto para ser jugado. A pesar de que cada persona especializada en un área en particular en la creación del videojuego tiende a tener su propia interpretación del PCG, en términos generales coinciden en aplicar un conjunto de reglas o parámetros fijos, para empezar con la creación del contenido pseudo aleatorio, y de este modo preservar los puntos importantes de un juego.

Togelus et al. [5] resalta que el contenido creado directamente por un jugador usando un editor o parte del mismo juego, no se considera PCG. Sin embargo no excluye la posibilidad del uso de información dada por el usuario, ya que esto haría que el contenido generado se adaptara mejor al jugador. Por lo tanto Togelus termina definiendo PCG como "la creación algorítmica de contenido del juego con la entrada del usuario limitada o indirecta". Del mismo modo menciona que las palabras "aleatorio" y "adaptativo" pueden o no formar parte de este método.

Shaker et al. [1] hace el análisis de qué se considera PCG y qué no, enumerando algunos ejemplos:

PCG es:

- Una herramienta de software que crea calabozos para un juego de aventura de acción como *The Legend of Zelda* sin ninguna intervención humana. Cada vez que se ejecute la herramienta, un nuevo nivel es creado;
- Un sistema que crea nuevas armas en el espacio en un juego de disparos en respuesta a lo que el equipo de jugadores suele hacer, así que las armas que se le presentan a un jugador son versiones de las armas que otros jugadores usaron;
- Un programa que genera juegos de mesa completos, reproducibles y equilibrados, quizás utilizando algunos juegos de mesa existentes como punto de partida;
- Un motor de juegos *middleware* (intercambio de información entre aplicaciones) que rápidamente llene un mundo con vegetación;
- Una herramienta de diseño gráfico que permite a un usuario diseñar mapas según las propiedades del juego, evaluando continuamente el mapa diseñado y sugiriendo mejoras para que sea más equilibrado y más interesante.

Ahora se enumeran algunas puntos que no se consideraron *PCG*:

1. Un editor de mapas para un juego de estrategia que simplemente le permite al usuario colocar y eliminar elementos, sin tomar ninguna iniciativa ni realizar ninguna generación de contenido por su propia cuenta;
2. Un reproductor artificial para un juego de mesa;
3. Un motor de juego capaz de integrar la vegetación generada automáticamente.

1.1.2. Distintas Aplicaciones

La necesidad de la creación automatizada de contenido, va más allá de los juegos, por lo tanto se podrían aplicar las mismas técnicas en otras áreas de Interacción Humano Computador (*HCI*). Por ejemplo, en la generación de interfaces, el arte de cualquier aplicación, menús de páginas web que sean generados automáticamente según el uso de la persona, horarios, listas de mercado. Esta generación de contenido personalizado según el usuario, se utiliza constantemente en la actualidad, como en páginas de compras dónde se muestran recomendaciones específicas según tu interacción anterior. Sin embargo, también es interesante que este contenido no se base necesariamente en la información recolectada sobre la interacción del usuario, sino que él solo con distintos algoritmos de optimización, dependiendo del caso,

genere pseudo aleatoriamente, por ejemplo, menús efectivos, o artes interesantes, etc [9].

1.1.3. Posible contenido a generar en videojuegos

El *PCG-G*, *Generación Procedimental de Contenido para Videojuegos*, no es una tarea fácil, ya que la palabra "contenido" trae implícita gran cantidad de detalles. Tomando como referencia la investigación de Hendrix et al. [2], en la cual ven el contenido como una pirámide en donde los niveles inferiores están incluidos en los superiores. Empezando por lo más específico, están los llamados "Bits" en estos se encuentran texturas, sonido, vegetación, edificios, conductas, y objetos como el fuego, el agua, las piedras y las nubes. Subiendo un nivel se halla el "Espacio" dónde se lleva a cabo el juego, este puede tener espacios interiores, y/o exteriores, así como cuerpos de agua. Una vez definido el espacio se crean los "Sistemas", como por ejemplo una red carreteras, un ambiente urbano, un ecosistema con flora y fauna, o una unión de todos estos que simule básicamente nuestro mundo. Luego se tienen que definir los "Escenarios", estos pueden contar una historia completa, pequeñas historias independientes, un escenario de acertijos en el que se tenga que encontrar una solución a un problema determinado, o niveles independientes que aumentan la dificultad progresivamente. Y por último el "Diseño" del juego, el cual puede enfocarse en un diseño de mundos, dónde se debe generar una trama o historias, o un diseño sistemático que crea patrones matemáticos, simétricos, como por ejemplo el ajedrez.

En la *Figura* 1.1 se pueden observar los niveles mencionados anteriormente, incluyendo en el tope de la pirámide "Derivados" que se refiere a cualquier contenido derivado del juego.

Figura 1.1: Tipos de contenido de juegos que se pueden generar procedimentalmente. [2].

Cada uno de estos niveles forman parte del contenido de un juego, la *PCG-G* se aplica a uno o a varios de estos. En la *Tabla* 1.1 se muestra una lista de juegos comerciales que aplicaron PCG para su creación, especificando en qué nivel se utilizó este método.

Cuadro 1.1: Lista de Juegos y contenido en el que se utilizan las técnicas PCG

Juegos (Año de lanzamiento)	Bits	Espacios	Sistemas	Escenarios
Borderlands (2009)	x			
Diablo I (2000)		x		x
Diablo II (2008)		x		x
Dwarf Fortress (2006)		x	x	x
Ellder Scrolls IV: Oblivion(2007)	x			
Elite (1984)		x	x	x
EVE Online (2003)	x	x		x
Facade (2005)				x
FreeCiv and Civilization IV (2004)		x		
Fuel (2009)		x		
Gears of War 2 (2008)	x			
Left4Dead (2008)				x
.kkrieger (2004)	x			
Minecraft (2009)		x		
Noctis (2002)		x		
RoboBlitz (2006)	x			
Realm of the Mad God (2010)	x			
Rogue (1980)		x		x
Spelunky (2008)	x	x		x
Spore (2008)	x	x		
Torchlight (2009)		x		
X-Com: UFO Defense (1994)		x		

Se puede observar que no se consiguió ningún juego comercial que haya implementado el método PCG en los últimos niveles *"Game Design"*, y *"Derived Content"*.

1.1.4. Orígenes

"La comprensión de los fundamentos de los juegos digitales radica, en gran parte, en la comprensión de los juegos no digitales que vinieron antes. Estos juegos han sido drásticamente influyente en los artefactos y procesos de investigación en la comunidad de juegos digitales " [10]

Cannizo et al. [11] menciona en su investigación que los métodos PCG se originan en 1975, cuando Benoit Mandelbrot definió el objeto fractal, el cual describe los patrones matemáticos repetitivos y similares entre ellos. A partir de esta idea empiezan las creaciones de texturas, ciudades, bosques, terrenos, videojuegos y otras áreas. También llamado *Dynamic Content Genaration*. La generación de niveles se empieza a aplicar desde la salida del juego " Rogue" en los años 80, posteriormente Diablo (1996), Spelunky (2008), Minecraft (2009).

Sin embargo Smith et al. [10] comenta sobre los orígenes del diseño modular en juegos análogos, mucho antes de los 80, donde los dados, las fichas y las cartas fueron diseñados para que se puedan jugar muchas partidas sin repetirse. En 1976 se crea el generador aleatorio de Haiku interactivo, que utiliza números aleatorios para tomar, de una lista, palabras y luego unirlas formando poemas. En 1979 se crea también un generador de música aleatoria que es capaz de crear armonías de cuatro partes.

A través de los años se escucha la denominación de "roguelike" para ciertos juegos con características específicas, de las cuales la generación procedimental es una de las que resalta. Johnson [12] explica en su investigación que los roguelikes son un género de juegos cuyo nombre deriva del videojuego que utiliza generación procedimental Rogue, creado en 1980, donde los principios centrales de este juego son niveles aleatorios, basado en turnos, alta complejidad, y " permadeath " (la incapacidad para volver a cargar un personaje que ha perecido). Roguelikes casi universalmente evita los gráficos modernos para un estilo gráfico centrado alrededor del texto. Esto significa que las paredes, pisos, y todos los enemigos y objetos están representados por Código Estándar Americano para el cambio de Información Internacional (ASCII), tales como " # " (a menudo paredes), *"]* " (con frecuencia piezas de la armadura" *)"* o " *$* " (a menudo montones de dinero).

Variedad de investigaciones se han llevado a cabo sobre la comprensión del *PCG* por la comunidad científica especializada en la parte técnica de los juegos. Por ejemplo, Hendrix et al. [2], examinó la gama de contenidos que pueden ser producidos y los algoritmos utilizados para producirlo, totalmente enfocado en juegos digitales. Togelius et al. [13] cubre las muchas maneras en que los algoritmos genéticos y otros enfoques de optimización se han utilizado para generar contenido para los juegos. Esta taxonomía establece distinciones entre el tipo de contenido que se produce, la necesidad de este, y el enfoque técnico utilizado. Es de particular interés en este trabajo, la observación de que PCG no necesita ser estocástico, el contenido determinista se ha utilizado para la creación de universos sin aleatoriedad, como es el caso

de Elite *(1984)*.

A Computer Generated Dungeon era una aventura en un calabozo generada por la computadora para el sistema de Túneles y Trolls. Disponible por sólo un corto período comprendido entre 1977-1978. Los pasillos estaban claramente diseñados por un autor humano, pero reordenados por el algoritmo para crear una variedad de experiencias diferentes, cada uno se vendió por cinco dólares con la promesa a su vez personalizarlo para cada jugador: los jugadores podrían seleccionar el nombre del monstruo y personalizar el texto en el pie de página.

1.1.5. Aplicaciones y Librerías

-JUEGOS QUE UTILIZAN PCG

Shaker et al. [14] hace un resumen de los juegos existentes que utilizan PCG:

Las limitaciones de almacenamiento de las computadoras es uno de los principales impulsores de la evolución de las técnicas de PCG. La reducida capacidad de las computadoras personales en los años ochenta restringiendo el espacio disponible para el contenido del juego, obligaba a los diseñadores a buscar otros métodos para generar y guardar el contenido. El videojuego Elite es uno de los primeros que resolvió este problema mediante el almacenamiento de los números de semillas utilizadas para la generación procedimental de ocho galaxias, cada una con 256 planetas con propiedades únicas. Otro ejemplo clásico del uso temprano de PCG es Rogue, un juego de "Dungeons" en el que los niveles se generan aleatoriamente cada vez que se inicia un nuevo juego. La Generación automática de contenido del juego, a menudo viene con ventajas y desventajas; juegos Roguelike pueden generar automáticamente experiencias atractivas, pero la mayoría de ellos (como Dwarf Fortress) carecen de atractivo visual. Recientemente, la generación de contenidos procedimentales ha sido testigo de una creciente atención en los juegos comerciales. Diablo es un videojuego de roles y de acción que ofrece la generación procedimental para la creación de los mapas, el tipo, número y colocación de artículos y monstruos. Civilization IV, es un juego de estrategia por turnos que permite una experiencia de juego única mediante la generación de mapas aleatorios. Minecraft, es uno de los últimos juegos independientes popular, con un amplio uso de técnicas de PCG para generar todo el mundo y su contenido. Spelunky, es otra plataforma 2D juego roguelike notable que utiliza PCG para generar automáticamente las variaciones de los niveles de juego. Tiny Wings, es otro ejemplo de un juego 2D móvil que ofrece un sistema de generación procedimental de terreno y textura, lo que le da al juego un aspecto diferente con cada repetición. En el año 2016 fue publicado No Man's Sky, dónde hay infinitos mundos por recorrer, creados utilizando PCG.

-LIBRERÍAS

MapMage [3]

MapMage es un generador de calabozos aleatorios para los juegos de rol como d20, Dungeons and Dragons o Pathfinder y está disponible en la App Store de Apple.

MapMage, generará instantáneamente mapas de calabozos para sus jugadores, completos con descripciones de la habitación, artículos, sonidos, olores, trampas y Encuentros. Los mapas generados y detalles del calabozo se pueden editar, enviar por correo electrónico o guardar para más adelante (*Figura 1.2 y Figura 1.3*).

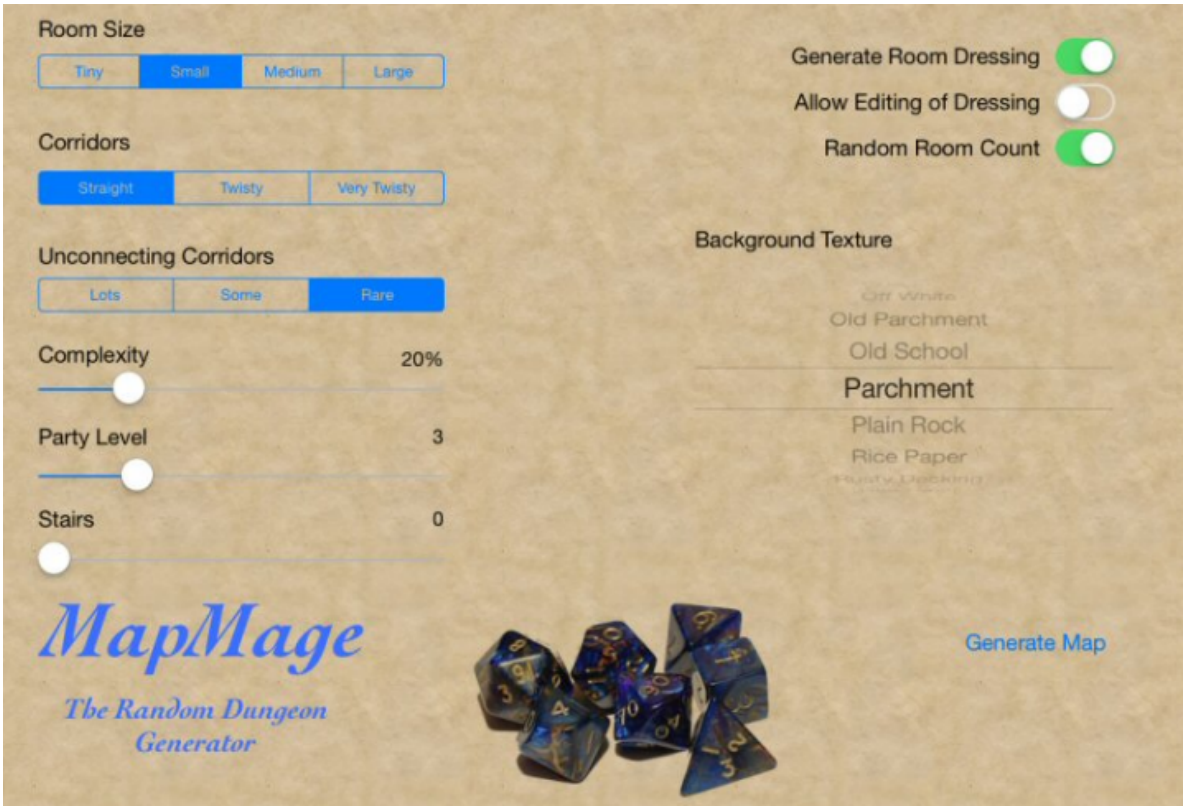

Figura 1.2: Muestra 1 de la interfaz gráfica de MapMage, generador de calabozos aleatorios [3].

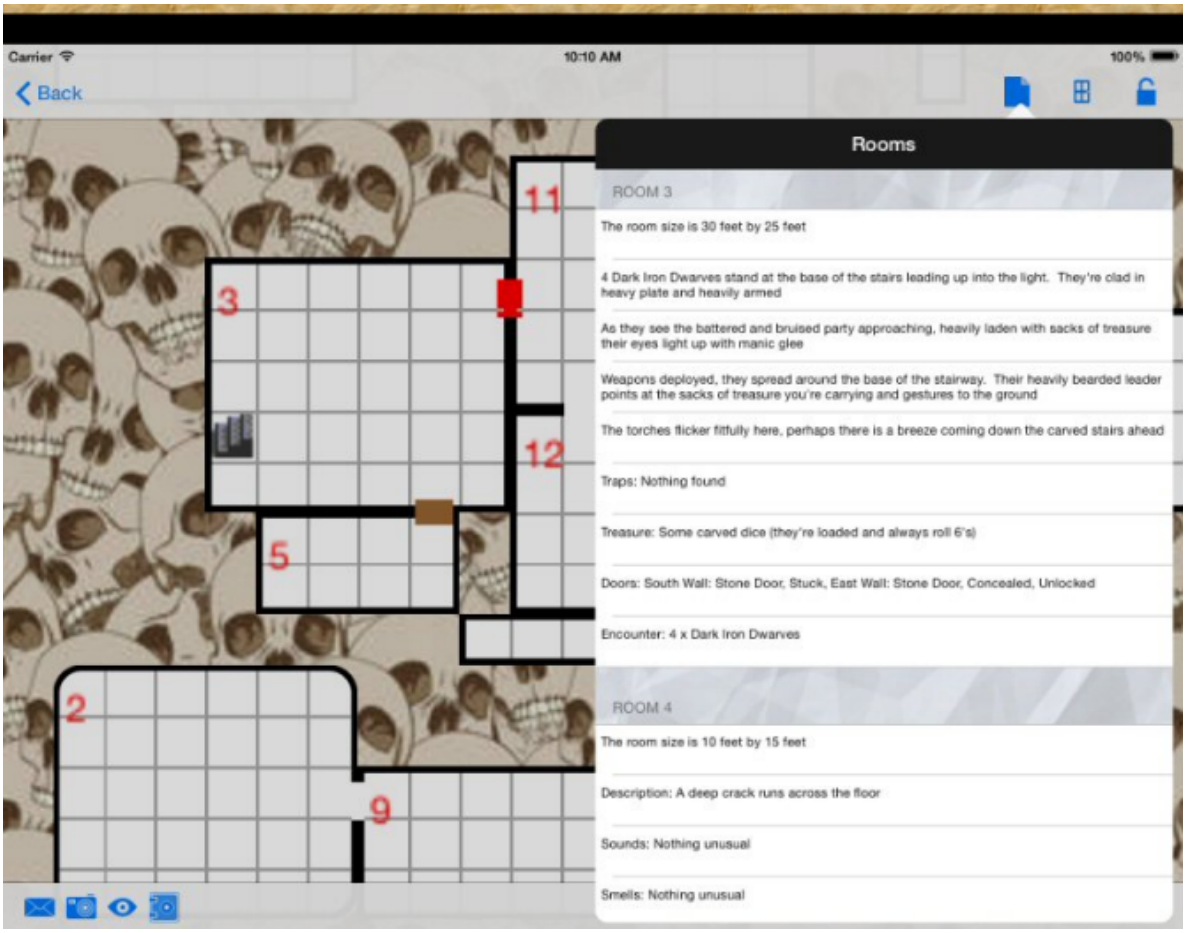

Figura 1.3: Muestra 2 de la interfaz gráfica de MapMage, generador de calabozos aleatorios [3].

LIBPCG [4]

LIBPCG es una pequeña biblioteca para la generación procedimental de contenido, contiene algunos algoritmos útiles que pueden ser utilizados para generar texturas, y demos.

La intención es crear una biblioteca pequeña y portátil que se puede utilizar para generar de manera determinada contenido en tiempo de ejecución para aplicaciones tales como juegos. Debido a que es de licencia MIT, tiene requisitos muy mínimos para el uso de la biblioteca. Mientras que el rendimiento debe ser una meta, en la actualidad se están enfocando principalmente en la implementación de algoritmos útiles. *Figura 1.4*

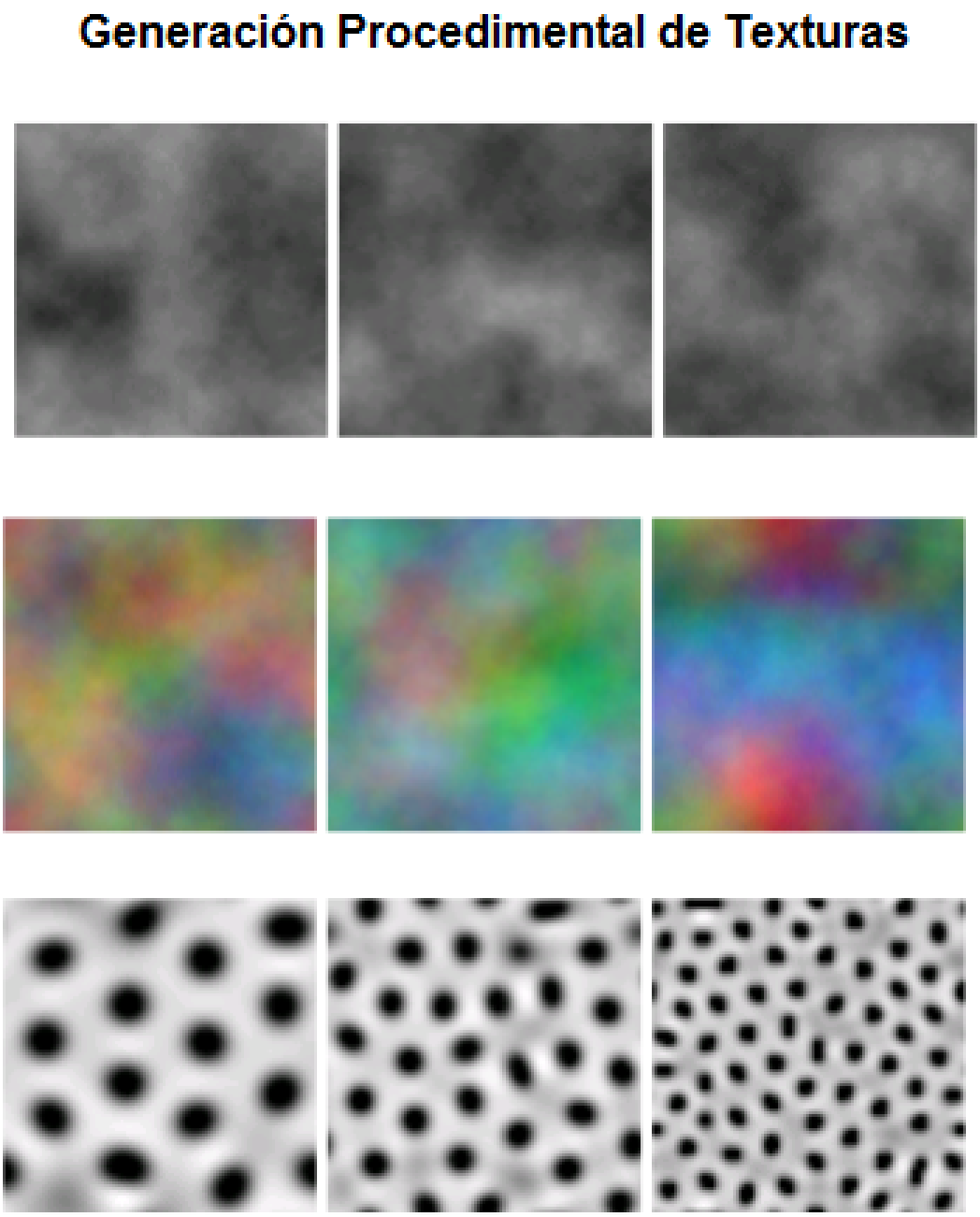

Figura 1.4: Ejemplo de Generación Procedimental de texturas utilizando LIBPCG [4].

Tanagra [15]

Tanagra es un prototipo de herramienta para el diseño de niveles de plataformas 2D, en el que un ser humano y la computadora pueden trabajar juntos para producir un nivel. El diseñador humano puede colocar restricciones en un generador continuo de niveles, de forma que especifique una geometría exacta y la manipulación del ritmo del nivel. La computadora entonces rellena el resto del nivel con geometría que garantiza capacidad de juego, o informa al diseñador que no existe un nivel que satisfaga sus requisitos.

SketchaWorld [16]

SketchaWorld, una investigación que integra técnicas de generación de contenido y métodos de interacción con el usuario para realizar la creación de mundos virtuales accesibles, eficientes y controlables. Utilizando un método intuitivo de interacción llamado "Procedural Sketching" , se puede crear el mundo virtual que se tiene en mente. Se pinta el paisaje con colores que representan escarpadas crestas de las montañas, colinas verdes, desierto estéril, etc. Además de esto, se dibujan las características del terreno, tales como ríos, bosques y ciudades, el uso de líneas y formas sencillas. Mientras tanto, cada elemento que se hizo en el boceto se expande automáticamente a una característica realista del terreno. Además, todas las características que se generan son, en forma automática, integradas con su entorno. Por ejemplo, el terraplén de una carretera está integrado en el paisaje, y cuando se cruza un río se inserta un puente adecuado en su lugar. Mientras se hace el boceto del mundo, se puede evaluar continuamente los resultados, tanto en 2D como en 3D.

1.2. Antecedentes

Existe gran cantidad de documentación, e investigaciones de posibles diseños a utilizar aplicando la Generación Procedimental de Contenido. Véase a continuación las más relevantes extraídas para nuestro proyecto, estructurada dentro de una taxonomía descrita por Shaker et al. [1] .

1.2.1. Taxonomía de la Generación Procedimental de Contenido [1]

Se describe a continuación una taxonomía que ayuda a estructurar los distintos métodos de generación procedimental de contenido.

1.2.1.1. En línea (*online*) versus fuera de línea (*offline*)

Las técnicas PCG pueden ser usadas para generar contenido en línea, mientras el jugador juega, permitiendo variaciones infinitas, con la posibilidad de generar contenido adaptado al jugador, también puede ser generado fuera de línea en el desarrollo de este.

El uso de la generación procedimental de contenido fuera de línea puede ser útil para crear contenido complejo, como mapas, como por ejemplo Left 4 Dead. Algunos juegos generados en línea dan la posibilidad de que otros usuarios puedan utilizar y compartir contenido y de esta forma las técnicas PCG pueden generar contenido aun más variado.

1.2.1.2. Necesario versus opcional

PCG puede generar contenido necesario que es requerido para completar un nivel, y al mismo tiempo podría generar simplemente contenido auxiliar que puede no ser utilizado, o sustituido, a continuación algunos ejemplos.

Contenido necesario:

2006 - Modelo de niveles de juegos de Plataforma [17]

La estructura de esta jerarquía está inspirada en *A Novel Representation for Rhythmic Structure*, un artículo que describe un modelo para la representación de los patrones rítmicos complejos en la música africana y afroamericana por Iyer et al. [18]. Iyer describe una representación jerárquica que captura la repetición rítmica y la combinación de secuencias rítmicas cortas en los pasajes más largos y más complejos. Un nivel en un juego de plataformas puede asociarse a un tramo musical, el diseño de estos se basan en gran medida en el ritmo. Acciones rítmicas ayudan al jugador a llegar a un estado de "flujo" de juego, un estado de concentración elevada (*Csikszentmihalyi, 1990*). Cuando un jugador está "en el flujo" o "en el ritmo" de un juego, haciendo saltos requiere no sólo el cálculo de distancias, sino también de temporización. La colocación rítmica de obstáculos crea una secuencia rítmica de los movimientos del jugador, haciendo que cada individuo salte más fácil en tiempo. Usando una repetición rítmica en la aparición de elementos variados, le puede dar al jugador un nivel más largo, utilizando unos pocos objetos del juego. Por ejemplo, mediante la repetición y la reorganización de algunos elementos básicos, tales como tubos, bloques y plataformas, es posible para los diseñadores construir niveles largos e interesantes en los juegos de *Mario Brothers*.

Contenido opcional:

2012 - Experiencia – Generación procedimental conducida de música para juegos [19]

La música se genera durante la ejecución mediante la frustración, el desafío y la diversión como parámetros generativos. El motor generativo ha sido concebido para ser tan sencillo como sea posible, sin embargo, capaz de producir música tonal simple. Se compone de las partes siguientes, ambas siguen una estructura clásica de algoritmo genético (GA) de Selección- Cruce- Mutación- Evaluación :

- Generador de secuencia armónica;
- Período constructor de la línea melódica.

Cada generador (acorde, secuencia, melodía) engendra nuevos candidatos cada vez que el elemento actual está listo (es decir, el último acorde de la secuencia actual ha terminado). La función de aptitud se utiliza entonces para seleccionar al ganador de entre los candidatos. Tanto el generador de secuencia armónica y el período de trabajo trabajan en tiempo real, creando estructuras musicales que reaccionan a la métrica emoción. La música se genera como una corriente de Interfase Digital de Instrumentos Musicales (*Musical Instrument Digital Interface*), estándar para componer los segmentos de la música que responden a determinados "estados de ánimo", eventos. Los generadores reaccionan en tiempo real a la métrica emoción con las siguientes reglas:

- Latidos por minuto: 120 latidos/minuto para la excitación mínima (relajado) y 135 latidos/minuto para la plena excitación;
- Escala: escala pentatónica es la preferida para la excitación mínima, y escala plena para la excitación;
- Novedad: la repetición de material ya escuchado se prefiere para una excitación mínima, y la introducción de nuevo material se prefiere para la excitación;
- Poca densidad: frases con las notas dispersas son los preferidos para la excitación mínima, y se prefieren frases densas para la excitación;
- Filtro resonante: bajo filtro resonante es el preferido para la excitación mínima, y se prefiere alto filtro resonante para la excitación.

El generador de secuencia armónica trabaja con una idea básica de la armonía: los acordes. Es el responsable de decidir el siguiente grupo de acordes con respecto a una historia de todos los acordes ya jugados. En tiempo de ejecución, cada vez que se necesita una nueva secuencia de acordes, todas las secuencias existentes son evaluados por la función de aptitud: los acordes contenidos en esta se comparan con la historia de los acordes ya jugados. Un índice de novedad se asigna a cada secuencia.

1.2.1.3. Grado y dimensiones de control

La generación de contenido puede ser controlada en distintas formas, con el uso de una semilla aleatoria, donde a partir de esta se genera el espacio, o utilizando parámetros que controlan el contenido a lo largo de sus dimensiones. Si la misma semilla se utiliza en el primer caso, puede generarse un mismo contenido varias veces, en cambio con la utilización de parámetros que varían a lo largo del juego siempre será distinto. Algunos ejemplos a continuación.

Uso de semillas:

2010 - Generación procedimental de niveles, usando la extensión de ocupación regulada [20]

La extensión de ocupación regulada *(ORE)*, es un algoritmo de un conjunto de geometrías que soporta niveles diseñados por personas, basado en escalas arbitrarias. Para generar niveles agradables sin pensar acerca de la mecánica, la *ORE* se basa en trozos prediseñados de niveles como materia prima, que opera de manera similar a la de casos basado en el razonamiento. Esta dependencia impone algunas limitaciones en el sistema, pero también le da algunas ventajas únicas, incluyendo la capacidad de aprovechar la creatividad de un diseñador de niveles humano. ORE funciona a través del ensamblaje de un nivel utilizando trozos extraídos de una biblioteca. El algoritmo utiliza posiciones que el jugador podría ocupar durante la reproducción para anclar cada trozo, y estas posiciones potenciales también se utilizan como los puntos de extensión para un nivel parcial. Este proceso se completa organizando el espacio de los niveles que se pueden producir a partir de los trozos de la biblioteca, más que todo mediante la exclusión de niveles incoherentes y otros que no pueden reproducirse.

Dimensión de control con parámetros:

2011 - Mario Infinito [5]

La versión *offline* del generador de niveles comienza con dejar que el jugador juegue un nivel ordinario (de alrededor de dos a tres minutos de longitud) del juego, que se ha generado al azar utilizando el generador de nivel estándar que se incluye con el *Infinite Mario Bros*. Este nivel está destinado a ser fácil para completar, y se limita a asegurarse de que hay una cierta diversidad entre las acciones del jugador. Inicialmente niveles completamente planos fueron utilizados, pero esto significaba que algunos jugadores sólo caminarían de izquierda a derecha y nunca presionarían el botón de salto o disparar. Todas las acciones tomadas por el jugador durante el juego partidas se registran con una resolución de trama (el juego corre a 24 cuadros por segundo). Tras la finalización del primer nivel, un nuevo nivel se genera para que el jugador juegue. El proceso de generación se inicia con una copia del nivel anterior. El generador de nivel toma los pasos de las acciones registradas del jugador, y modifica el nuevo nivel de acuerdo con las medidas adoptadas. Un simple conjun-

to de reglas determina cómo modificar el nivel en cada posición dependiendo de la acción que el jugador tomó en esa posición. Se puede observar en la *Figura 1.5* un ejemplo.

La versión *online* va creando el nivel a medida que lo vas jugando. Y el tiempo por partida se disminuyó a un minuto aproximadamente.

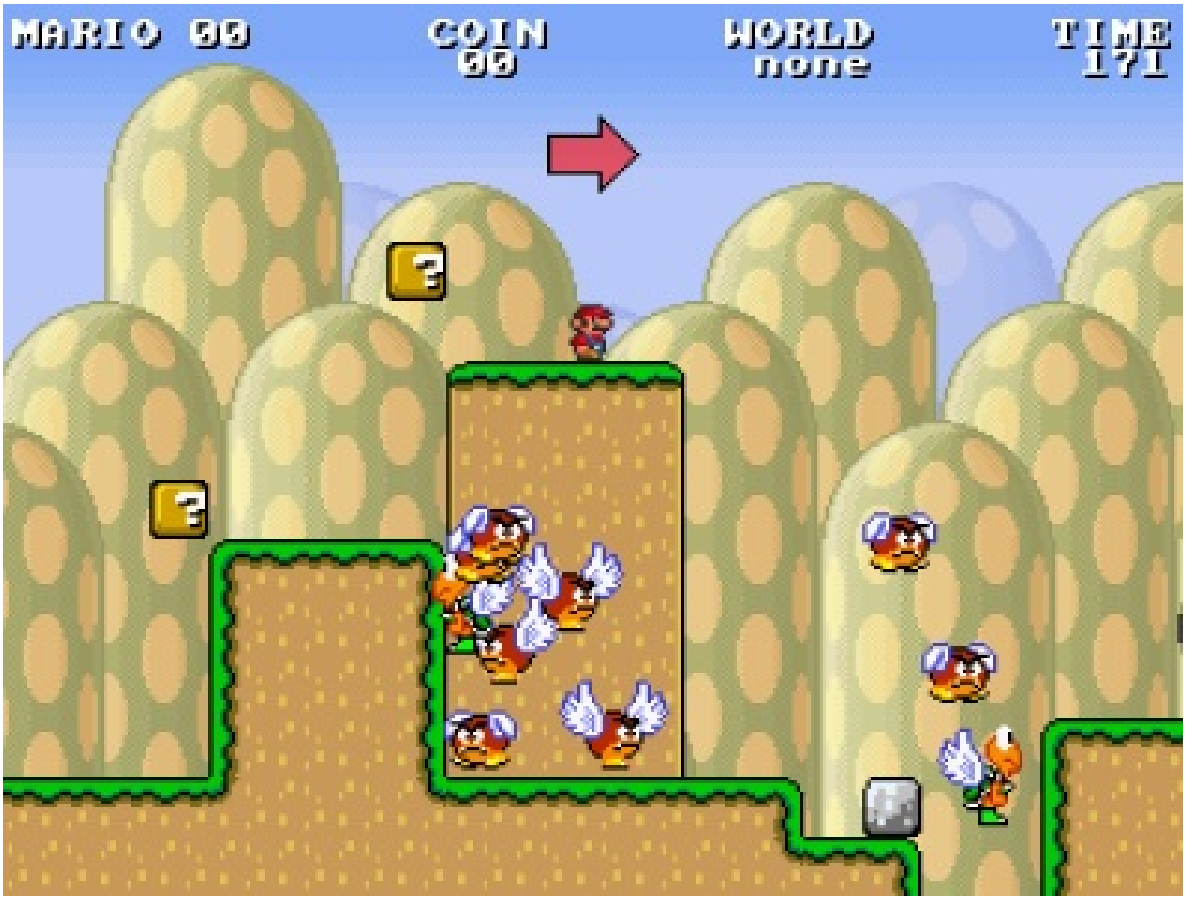

Figura 1.5: Infinite Mario. Enemigos Creados en respuesta a las monedas recolectadas [5].

1.2.1.4. Genérico versus adaptativo

El contenido generado puede ser genérico o adaptativo, donde el genérico no toma en cuenta las acciones del jugador, y el adaptativo analiza la interacción de este para crear contenido. A continuación unos ejemplos.

Contenido genérico:

2009 - Generación de nivel basada en ritmo para plataformas 2D [6]

Generación de niveles basada en ritmos para juegos 2D de plataforma: El enfoque basado en el ritmo de los niveles de producción se describe en la *Figura 1.6*. Se inicia basándose en la gramática de dos capas para la generación de pequeños trozos de un nivel, llamados grupos de ritmo . La primera etapa crea un conjunto de acciones que el jugador va a tomar obligado a formar un ritmo. La segunda etapa utiliza una gramática para convertir este conjunto de acciones en la geometría correspondiente de acuerdo con un conjunto de restricciones físicas. Para formar un nivel completo, se unen grupos rítmicos aptos, tendiendo un puente con una pequeña plataforma que actúa como zona de descanso para el jugador. Muchos niveles diferentes se generan, formando un conjunto de niveles de candidatos que se ensayan luego contra un conjunto de críticos para determinar el mejor nivel. Este nivel, entonces se le conoce como un "nivel de base", que puede ser mejorado a través de la decoración con las monedas y atar sus plataformas a un punto común. En todas las etapas de creación de niveles, el estilo juega un papel importante. El estilo es representado como un conjunto de parámetros que un diseñador humano puede ajustar. Los parámetros incluyen por ejemplo la frecuencia de saltos por grupo, y con qué frecuencia un resorte debe ser generado por un salto.

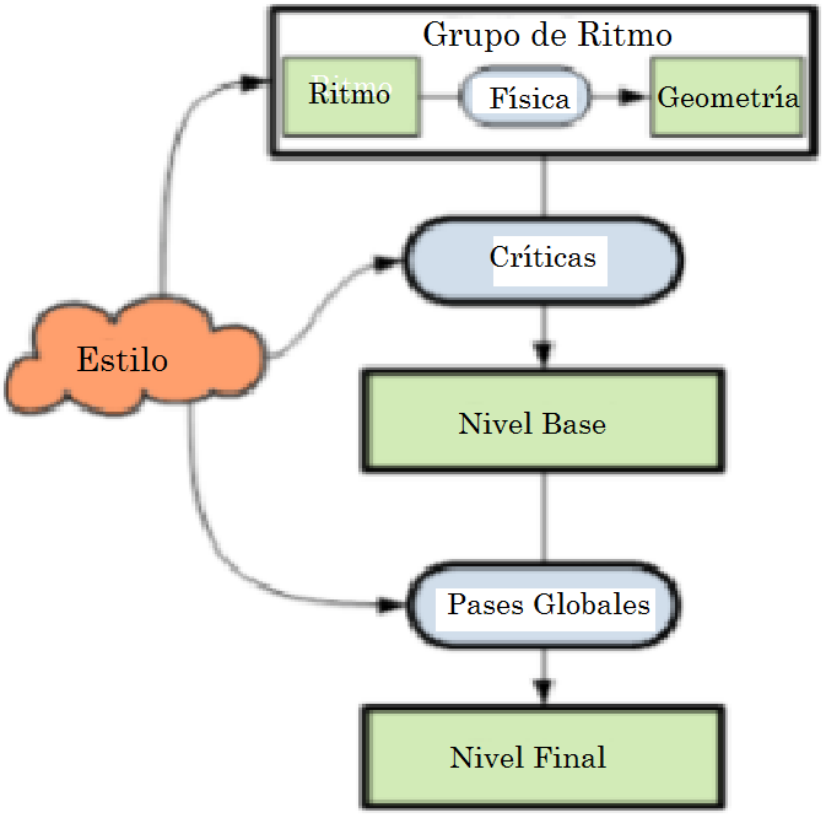

Figura 1.6: Algoritmo de generación de niveles. Los cuadrados verdes indican entidades generadas, mientras que los círculos azules indican limitaciones. Los parámetros de estilo influyen en muchos aspectos de la generación de niveles [6].

```
move  0.00  8.00
jump  2.00  2.25
jump  4.00  4.25
jump  6.00  6.50
move 10.00 12.00
move 14.00 20.00
jump 16.00 16.25
jump 18.00 18.25
```

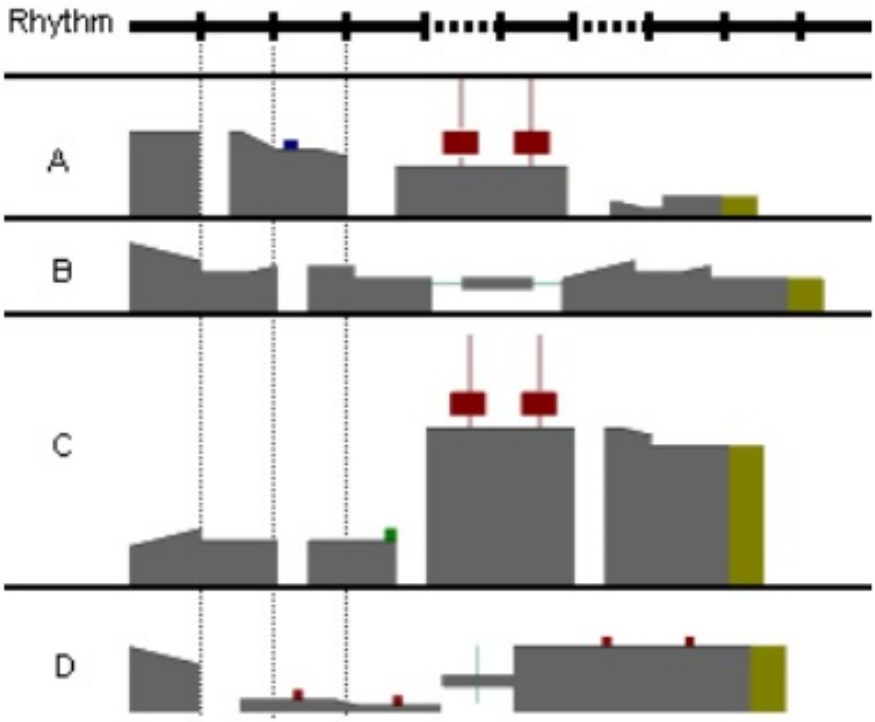

Figura 1.7: Cuatro posibles interpretaciones de la geometría del ritmo proporcionado. Pequeñas cajas rojas denotan enemigos para matar, pequeñas cajas verdes denotan resortes, cajas azules son enemigos a evitar, grandes cajas de color rojo son ascensores que siguen la línea asociada y las plataformas delgadas rectangulares, son móviles. La plataforma verde grande, es el elemento de unión de este grupo al ritmo siguiente [6].

Contenido adaptativo:

2010 - Polimorfo: Ajuste dinámico de dificultad, a través de la generación de niveles [21]

El objetivo de Polimorfo es generar automáticamente los niveles de plataformas 2D durante el juego como un medio de ajuste de dificultad dinámica. Específicamente, en lugar de ser escrito a mano, los niveles de juego serán generados procedimentalmente mientras el jugador se mueve a través del nivel, un trozo a la vez según sea necesario. La generación de estos trozos se puede personalizar para que coincida con el rendimiento del jugador, por lo que cada jugador se le presentará un nivel que ofrece un desafío apropiado a su habilidad. Esto no quiere decir que el jugador nunca va a morir en una sección difícil o resistir en una sección fácil, pero el juego va a corregir esto en la siguiente sección, evitando con suerte todo lo relacionado con las dificultades del jugador frustrantes y el aburrimiento.

Con el fin de generar partes de un nivel para que coincida con el nivel de habilidad de un jugador, se necesita tanto un modelo de dificultad en un dominio previamente obtenido de los niveles de plataformas en 2D y un modelo dinámico del rendimiento actual del jugador. Se crea una herramienta de recolección de datos que le pide a un jugador humano jugar un segmento de nivel corto (aproximadamente 10 segundos), se toman datos sobre el nivel y el comportamiento del jugador en el camino. Después de que el jugador completa el nivel o su personaje muere, se le pide etiquetar el segmento de nivel respondiendo a la pregunta de opción múltiple sobre la dificultad de este segmento de nivel. Las opciones que se presentan al jugador son 1 fácil hasta el 6 difícil. Sólo los datos de los jugadores que completan múltiples niveles se consideran, con el fin de evitar niveles de dificultad subjetivos.

Cuando se genera un primer segmento del nivel, cada dato proporcionado estimará su dificultad en una métrica determinada. Por ejemplo, un crítico examina densidad de la acción del segmento del nivel, mientras que otro simplemente cuenta el número de componentes en el nivel potencialmente mortales presentes, ya que algunos componentes no crean la posibilidad de muerte del jugador por su propia cuenta. Una vez que todos los críticos han examinado el segmento del nivel, se clasifica en varias categorías de dificultad estimadas, las cuales son tomadas por la herramienta de recolección de datos para los segmentos presentados al jugador.

Contenido adaptativo:

2011 - Experiencia - Generación de contenido procedimental impulsado (*EDPG*) [9]

Los componentes de EDPCG son:

- Modelado de la experiencia del jugador.
- La experiencia es modelada como una función del contenido del juego y al es-

tilo de juego del jugador, y sus respuestas cognitivas y afectivas a los estímulos del juego.

- Los modelos de experiencia del jugador pueden ser construidos con diferentes tipos de datos recogidos de parte de estos.

Se pueden identificar tres distintas formas de obtener datos para el modelado de la experiencia del jugador. Se basa en:

1. Los datos expresados por los jugadores.
2. Datos obtenidos a partir de modos alternos a las respuestas de los jugadores.
3. Los datos obtenido a través de la interacción entre el jugador y el juego.

Se evalúa el contenido a generar:

- *La calidad del contenido.* La calidad del contenido generado se evalúa y vincula a la experiencia modelada del jugador.
- *La representación de contenido.* El contenido está representado conforme con maximizar la eficacia, el rendimiento y la robustez del generador.
- *Generador del contenido.* El generador, a través del contenido del espacio, busca contenido que optimice la experiencia del jugador según el modelo adquirido.

Si el contenido es representado a través de un pequeño número de dimensiones (indirectamente), la búsqueda exhaustiva (busca todas las posibles soluciones) debe ser capaz de proporcionar soluciones robustas. En general, cuanto más particular se convierte la representación, mayor será el espacio de búsqueda de contenido. Donde la búsqueda exhaustiva no es factible, otras técnicas se podrían utilizar, variando de heurística y de búsqueda de gradientes (si el gradiente es computable) para las técnicas de optimización estocástica globales tales como los algoritmos evolutivos y optimización por nube de partículas (PSO, simula el comportamiento de las partículas en la naturaleza). Idealmente, el generador de contenido debe ser capaz de identificar, cuánta cantidad y con cuánta frecuencia el contenido debe ser generado para un jugador en particular. Hay jugadores que no les gusta adaptarse a nuevas apariciones, y otros que lo aceptan y en su lugar detestan la idea de tener que repetir cualquier sección de un juego. Se cree que un mecanismo EDPCG exitoso debe ser capaz de reconocer si un jugador no le gusta la noción de adaptación. Esto se suma a la importancia de métodos adecuados para el modelado de la experiencia del jugador durante el juego

1.2.1.5. Estocástico versus determinista

El contenido puede ser estocástico cuando volver a repetirlo se hace casi imposible, es aleatorio, en cambio el contenido determinista está basado en parámetros que

orientan el juego de una forma determinada capaz de ser repetida. A continuación algunos ejemplos de distintos autores.

Este ejemplo podría ser estocástico o determinista dependiendo de su desarrollo, donde tienes la posibilidad de especificar parámetros, o simplemente dejarlo al azar:

2013 - Diseño de niveles generados procedimentalmente [22]

Un diseñador expresa todas las restricciones de diseño, las cuales dan lugar a una instancia de una gramática del juego, capaz de reescribir un sistema de acciones iniciales en un grafo. Los nodos en ese grafo representan grupos de las acciones del jugador y las aristas indican su orden. Con el tiempo, esta información en el grafo determinará un diseño de nivel de juego. Diferentes conjuntos de restricciones especificadas por el diseñador, resultan en gramáticas de juego distintas, que es una primera forma de controlar la generación. El grafo inicial se compone de un conjunto de nodos de acciones iniciales. El algoritmo generativo reescribe las acciones compuestas en un subgrafo de acciones vinculadas.

Se realizan los siguientes pasos mientras todavía exista alguna de las acciones compuestas:

- Seleccione la primera acción compuesta en el grafo
- Seleccione una opción basada en los valores de los parámetros (al azar, si no se establecen condiciones)
- Si es necesario, seleccionar al azar subopciones
- Convertir la opción seleccionada para volver a escribir un grafo de nodos de subacciones (subgrafo).
- Añadir la acción compuesta como el padre de todos los nodos del subgrafo, y así sucesivamente mientras se creen más nodos.
- El siguiente paso consiste en acciones de grupo en el mismo espacio, es decir, resolver la colocalización de las acciones. Los nuevos nodos de grupo se crean a partir de la fusión de los nodos individuales que deben ser yuxtapuestas. Estos nuevos nodos son grupos de acciones que representan un espacio. La agregación de nodos tiene algunas particularidades. Si uno de los nodos ya estaba en un grupo, todos los nodos se fusionan en un nuevo grupo. La fusión debe ocurrir debido a que parte de una secuencia de coubicación se puede cortar por la mitad ya que la ramificación en combinación con la recursividad *depthfirst* (el primeromásprofundo). Si existen los dos nodos para ser fusionadas en el mismo nivel del árbol (comparten un padre o un nodo secundario), más duplicados de uno de ellos podría ocurrir teóricamente en el mismo nivel. El algoritmo inspecciona todas las acciones compuestas originales (en el paso 5) con las que se originó cada nuevo nodo. La fusión se produce sólo dentro

de estas jerarquías de acciones compuestas. Por último, los pares conectados semánticamente están marcados mediante la inspección de todas las acciones y dar marcha atrás a su acción compuesto de padres y jerarquía.

Con este algoritmo generativo, múltiples gramáticas y parámetros pueden generar una variedad de grafos de acción. Estos no sólo indican la secuencia de acciones que deben ocurrir en el juego, sino también otros requisitos como, por ejemplo, su contenido objetivo, los grupos en los que deben producirse algunas acciones en el mismo espacio, así como los pares de acciones semánticamente conectados

Estocástico:

2011 - Diseño de niveles como una transformación de modelos: una estrategia [7]

Un diseñador de niveles genera un conjunto de diferentes modelos, trabajando lentamente hacia el nivel completo. A pesar de que la mayoría de los diseñadores de niveles no piensan en sus productos como modelos, se argumenta que muchos de ellos son: el boceto inicial de un diseño de nivel, o un storyboard son los modelos que se centran en aspectos particulares del nivel completo. Estos modelos están relacionados: el primer modelo de alguna manera afecta o incluso dicta la construcción del segundo modelo. El objetivo de este análisis es doble: se puede utilizar como una estrategia para la automatización de algunas partes del proceso de diseño de los niveles, pero también formaliza el propio proceso de diseño; transformaciones de modelos permiten a los diseñadores razonar acerca del diseño de los niveles de una manera estructurada y abstraída.

El proceso a través del cual se transforma un modelo en otro modelo puede ser capturado utilizando sistemas de reescritura.Estos sistemas consisten en reglas que tienen un lado derecho e izquierdo. Estas reglas especifican un conjunto de símbolos (el lado izquierdo) que se puede sustituir por otro grupo de símbolos (el lado derecho). Esta operación es similar a la utilización de reglas en gramáticas formales. Gramáticas formales o generativas se originan en la lingüística donde se utilizan como modelo para describir conjuntos de frases lingüísticas encontradas en el lenguaje natural Los grafos son más útiles que las cadenas de caracteres para representar estructuras de misiones, y también pueden representar el espacio.

En una gramática de grafos uno o varios nodos y las aristas de interconexión pueden ser reemplazados por una nueva estructura de nodos y aristas. Las *Figuras 1.8* y *1.9* ilustran este proceso. Después que un grupo de nodos se ha seleccionado para sustituirse descrito por una regla en particular (*Figura 1.8*), los nodos seleccionados están numerados de acuerdo con la parte izquierda de la regla (paso 2 en la *Figura 1.9*). Paso siguiente, todos las aristas entre los nodos seleccionados se eliminan (paso 3). Los nodos numerados se reemplazan por sus equivalentes (nodos con el mismo número) en el lado derecho de la regla (paso 4). Luego los nodos en el lado derecho que no tienen un equivalente en el lado izquierdo se añaden al grafo (paso

5). Finalmente, las aristas que conectan los nodos nuevos se introducen en el grafo especificado por el lado derecho de la regla (paso 6) y los números se eliminan (paso 7). Tenga en cuenta que las gramáticas de grafos pueden tener operaciones que permiten a los nodos existentes ser eliminados, estas operaciones no se utilizan en esta técnica.

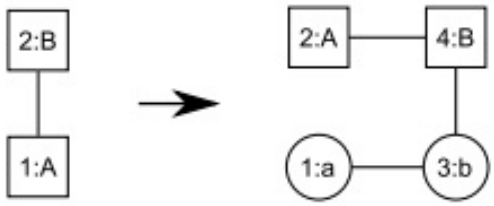

Figura 1.8: Una regla de la gramática de un grafo. Nodos cuadrados denotan símbolos no terminales y nodos circulares denotan símbolos terminales [7].

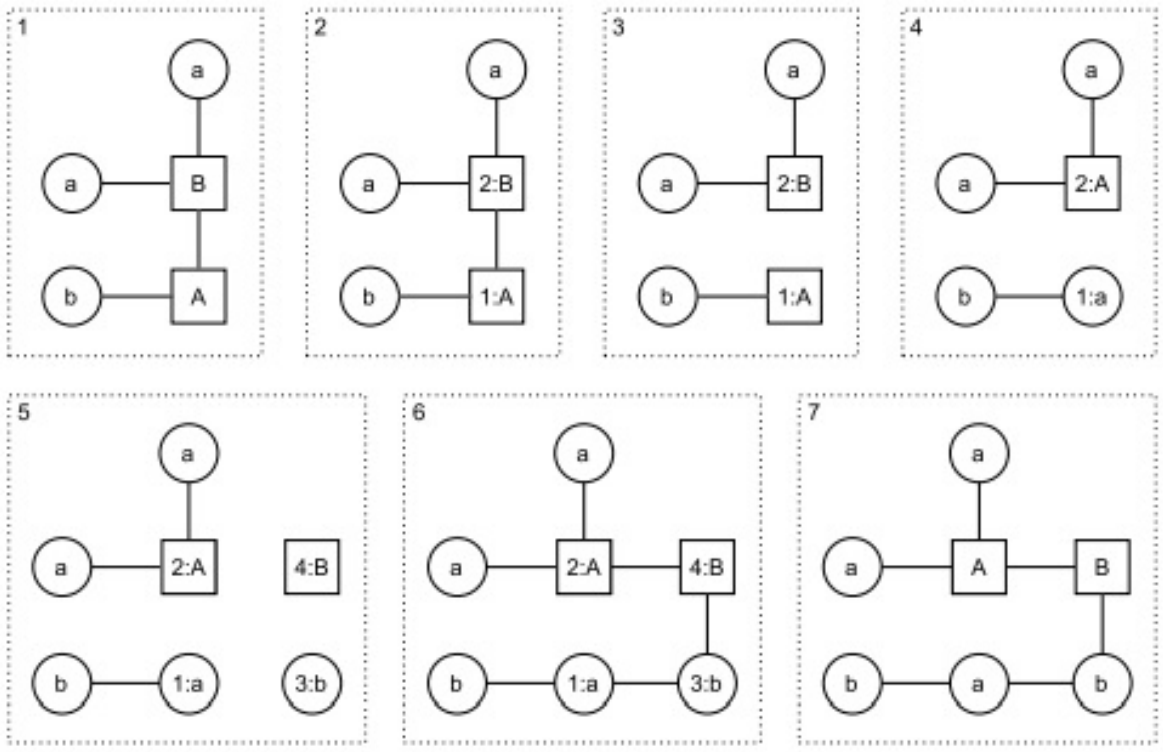

Figura 1.9: El proceso de aplicación de la regla representado en la *Figura 1.8* a un grafo [7].

La diferencia entre las gramáticas formales y sistemas de reescritura de grafos es que los sistemas de reescritura pueden tomar un conjunto de símbolos como su punto de partida, y carece de una clara distinción entre el terminal y los símbolos no terminales. Esto significa que un sistema de reescritura no termina de la misma manera como una gramática formal lo hace. Cualquier transformación conduce a un modelo significativo.

Sistemas de reescritura deben operar en los modelos que pueden ajustarse a una gramática formal. En este caso particular, un sistema de reescritura de grafos podría comenzar a partir de un grafo que representa una misión y transformarlo en un espacio. Las normas de los sistemas de reescritura deben estar diseñadas de tal manera que el modelo de salida no entre en conflicto con la gramática del modelo de destino. La *Figura 1.10* representa los modelos y las gramáticas en relación con la misión y el espacio entre sí y un sistema de reescritura.

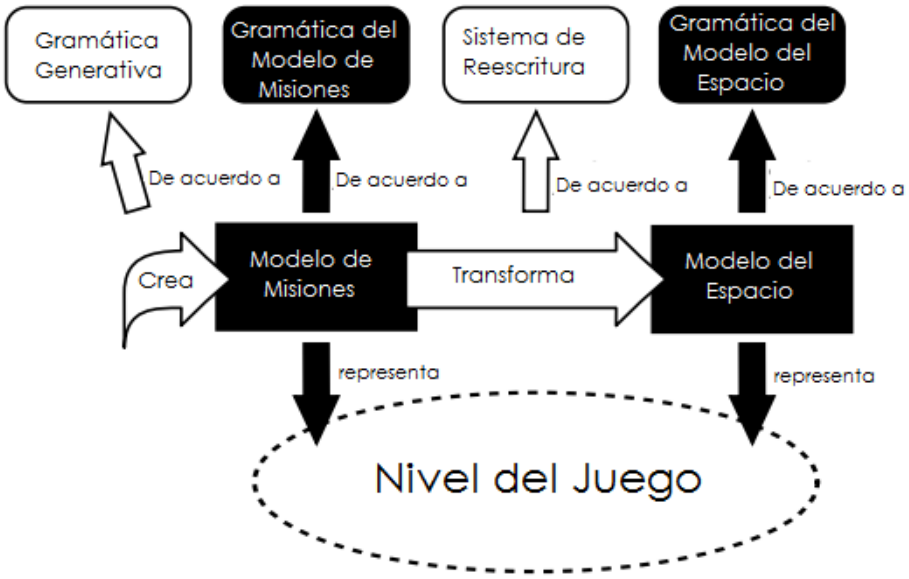

Figura 1.10: Diseño del nivel como una transformación del modelo [7].

Los sistemas de reescritura de grafos son diferentes que las gramáticas formales ya que no definen un lenguaje o un modelo. Ellos pueden, sin embargo, codificar los principios del diseño: los sistemas de reescritura especifican las operaciones que un diseñador podría llevar a cabo en un modelo con el fin de transformar un modelo a otro. Cuando se implementa como una transformación automática, estos sistemas de reescritura muy son estrictos; permiten sólo las operaciones que están representadas por sus reglas y ninguna otra. Diseñadores de la vida real son más flexibles, sin embargo, también siguen ciertas restricciones. Si el objetivo es crear un nivel que tenga solución, ningún diseñador colocaría una llave fundamental detrás de una cerradura

que se abra con la misma llave, ya que esto crearía un punto muerto. La ventaja de utilizar un sistema de reescritura es que tales operaciones se pueden prevenir. Esto requiere que el sistema de reescritura esté construido de acuerdo con ciertas limitaciones, y todas las transformaciones que se apliquen estén conforme a las reglas del sistema de reescritura. Para un diseñador humano esto podría no ser la mejor, o más fácil, forma de trabajar, pero se puede automatizar fácilmente. Una herramienta de software para el diseño de niveles puede ser desarrollada, que implemente todas las operaciones posibles para generar la misión y el espacio basada en modelos de sistemas de reescritura. Una herramienta de este tipo tendría las ventajas adicionales que permitirían a un diseñador producir diferentes niveles y corregir rápido y eficientemente. Además, es perfectamente concebible que para determinados tipos de juegos, todo el proceso de diseño de los niveles se puedan automatizar de esta manera. La *Figura 1.10* sugiere que la aplicación de reglas de reescritura de un grafo que representa una misión siempre resulta en un grafo que representa un espacio. Este no tiene por qué ser el caso. Como ya fue mencionado, la transformación real de una misión en un espacio podría implicar muchos pasos de transformación más pequeños, cada uno gobernado por diferentes sistemas de reescritura. Un sistema de reescritura puede crear un número de tareas, otro sistema podría añadir algunas dependencias o asegurar que las tareas estén en un orden interesante, el siguiente sistema de reescritura podría añadir cerraduras y llaves para crear un no-lineal, una estructura de la misión más de tipo espacio, mientras que otro podría añadir tareas de bonos y recompensas. Esta transformación gradual de la misión y el espacio pone en contraste agudo que la misión y el espacio no son más que perspectivas útiles sobre los niveles de juego; existen muchas perspectivas intermedias. Sin embargo, las estructuras en primer plano de estas perspectivas tienen sus propias características, y los diseñadores experimentados aprovechan de estas características para crear experiencias de juego convincentes.

1.2.1.6. Constructivo versus generar y probar

En la generación procedimental de contenido constructiva el contenido es generado en una sola corrida, como los juegos *roguelike*, por otro lado se puede generar contenido y ser probado y seguir generando cada vez contenido distinto a medida que avanza el juego. Por ejemplo:

Generación constructiva:

2014 - Generación procedimental de calabozos [23]

Uno de los métodos para la generación de laberintos procedimentalmente son con autómatas celulares. Esta estructura de autoorganización consiste en una cuadrícula de célula en cualquier número finito de dimensiones. Cada célula tiene una referencia a un conjunto de celdas que conforman su vecindad, y un estado inicial en el

momento cero. Para calcular el estado de una célula en la siguiente generación, un conjunto de reglas se aplican al estado actual de la célula y al de los vecinos. Después de varias generaciones, los patrones pueden formar una red, que en gran medida depende de las reglas utilizadas y estados celulares. El modelo de representación de un autómata celular es una cuadrícula de células y sus estados. Un ejemplo de conjunto de estados permitidos sería, en cada célula, ya sea un lugar donde un jugador puede ir, o un lugar donde el jugador no puede ir. Se utilizan las capacidades de autoorganización de los autómatas celulares para generar niveles. Ellos definen la vecindad de una celda como sus ocho celdas circundantes. Después de una conversión de celdas al azar inicial (del piso a la roca), el conjunto de reglas se aplica de forma iterativa en múltiples generaciones. Este conjunto de reglas establece que: 1) una célula será un roca si el valor de zona es mayor que o igual a y piso del modo contrario; y 2) una celda de roca que tiene una celda vecina de piso será una celda de pared. Sobre la base de estas reglas, estructuras de niveles de cuevas pueden ser producidas. Este método permite aplicarse en tiempo real y en generación infinitas de mapas.

Gramáticas de grafos se han utilizado previamente por Adams [19] para generar niveles de Cuevas. Aunque la obra del autor se aplica a tiradores en primera persona (*First person shooter, FPS*), nuestra definición de un nivel de cuevas todavía se aplica directamente a los contenidos generados. Esto es claro en el uso exclusivo de Adams del término "niveles de calabozos" a través de su trabajo, aunque sólo teniendo en cuenta los FPS. Reglas de una gramática de grafos se utilizan para generar descripciones topológicas de los niveles. Todos los detalles geométricos adicionales (por ejemplo, el tamaño de la habitación) se excluyen de este método. Lo más interesante es la generación de grafos puede ser controlar mediante el uso de dificultad, diversión, y los parámetros de entrada de tamaño globales. niveles para que coincida con los parámetros de entrada de producción se logra a través de un algoritmo de búsqueda que analiza todos los resultados de una regla de producción en un momento dado, y selecciona la óptima.

Hartsook et al. [22] presentó una técnica para la generación automática de los mundos del juego de rol basado en una historia. La historia puede ser hecha por una persona o generada. Ellos hacen una correspondencia entre la historia del espacio de juego utilizando una metáfora de islas y puentes, y capturan ambas en un árbol de espacio. Las islas son zonas en las que se producen puntos de la trama de la historia. Puentes conectan las islas entre sí (aunque son "ubicaciones" y no "caminos"). Un árbol de espacio representa la conectividad entre las islas y puentes, y también tiene información sobre los tipos de ubicación (también llamados tipos ambientales). Hartsook et al. [22] Utiliza un algoritmo genético para crear árboles de espacio. Cruce y mutación de acuerdo con adición y eliminación de nodos y aristas en el árbol. La función de la fórmula física utilizada por Hartsooketal usa una evaluación de entornos conectados (basado en un modelo que suaviza los ambientes uno al lado del otro) y datos sobre el estilo de juego del jugador (para determinar la longitud correcta y el número de ramas). Los ajustes incluyen el tamaño del mundo,

la linealidad del mundo, la probabilidad de encuentros enemigos, y la probabilidad de tesoros hallados. El árbol del espacio que el algoritmo genético selecciona como mejor opción, después de un número fijo *fi* de iteraciones, se utiliza entonces para generar un mundo del juego, donde los nodos del árbol se asignan a una red utilizando un algoritmo de *Backtracking* (búsqueda en profundidad). Si no hay una solución de correspondencia, a continuación, el árbol del espacio se descarta, y un nuevo árbol de espacio tiene que ser construido.

Generar y probar:

2015 - Generación de contenido basado en la gramática a partir de las curvas de dificultad proporcionadas por el diseñador [24]

La estructura general de este enfoque es la siguiente. El diseñador de juegos proporciona:

1. La curva inicial de dificultad del nivel,
2. la especificación general de la misión,
3. el conjunto de reglas de la misión, y
4. el conjunto de reglas de espacio.

Los cuatro especificaciones de diseño se introducen en un proceso evolutivo que da salida a un grafo de la misión final mediante el uso de gramática generativa. El grafo de la misión final se traduce en el espacio real del nivel, en el que la misión puede ser llevada a cabo por el jugador.

Especificación de la misión. La misión asociada con un nivel describe las tareas que deben ser realizadas por el jugador con el fin de avanzar por el nivel. Se codificó tal especificación de la misión mediante una representación gráfica, en la que los vértices describen las tareas, mientras que las aristas denotan el orden en que las tareas se van a realizar por el jugador.

Especificación curva de dificultad. Una manera para definir una experiencia de juego deseado puede conseguirse mediante la construcción de una curva de dificultad de destino. Aquí, la curva codifica la dificultad anticipada para el jugador mientras se avanza a través del nivel. Por lo tanto, la curva de dificultad se puede definir como una función en el tiempo con valores escalares de un intervalo de dificultad determinado por el diseñador del juego.

Generación de la misión. La generación real de los niveles se realiza en referencia a una curva de dificultad, y dada una especificación de misión prevista, se lleva a cabo con un método evolutivo.

La generación de espacio. Una vez la generación evolutiva de misiones termine, el mejor candidato de la misión se determina y sirve como base para la generación del espacio del nivel. Para cada nodo terminal en el gráfico de la misión se genera un

subespacio y luego todos los subespacios se cosen juntos de acuerdo con la relación espacial entre los nodos.

La adaptación del jugador. El proceso real de generación de nivel se puede realizar durante el proceso de desarrollo del juego, y también en línea, es decir, mientras el juego se está jugando. Esta última posibilidad ofrece una clara oportunidad para personalizar la misión generada procedimentalmente con evaluaciones sobre, por ejemplo, la experiencia del jugador. Tras la entrada en la evaluación de la experiencia del jugador, el conocimiento del dominio del diseñador del juego puede definir que la curva de dificultad que se utilice deberá ser desplazada hacia arriba, hacia abajo, o debe ser sustituida por una curva alternativa por completo; todo ello con el objetivo de proporcionar una experiencia de juego más equilibrada, mientras que el juego se está jugando.

1.2.1.7. Generación automática versus autoría mixta

A parte de la generación de contenido automática mencionada en los ejemplos anteriores, también está la posibilidad de utilizar contenido generado por técnicas PCG y ser aplicado por el diseñador durante la creación del juego, como por ejemplo aplicaciones antes mencionadas como SketchaWorld, que genera mapas, o Tanagra, estas aplicaciones por si solas no generan un juego, pero un diseñador puede utilizarlas. A continuación un ejemplo donde se utiliza un algoritmo para la generación procedimental de edificios para un juego.

Autoría Mixta:

2016 - Generación procedimental de niveles en *Angry Birds* utilizando la gramática constructiva, con modelos de estilo Chino o Japonés [8]

El *2D-BCG* (*2D-Building Constructive Grammar*) está escrito en *Backus-Naur Form* como se muestra a continuación. En lo que sigue, las variables entre corchetes angulares (<>) son símbolos no terminales, mientras que las variables que no están entre corchetes son símbolos terminales. Los símbolos no terminales en el lado izquierdo de ":: =" se pueden convertir en símbolos del lado derecho. El " | " significa selección; un símbolo se puede escoger entre cualquier otro separado por " | " .

```
 ::=
::= wall floor | wall | floor
< main>::= beam < mainlist >beam
< mainlist >::= window | door |

< mainlist> beam< mainlist>
::=  | toproof |
 toproof
```

```
< rooflist>::= roof | roof
```

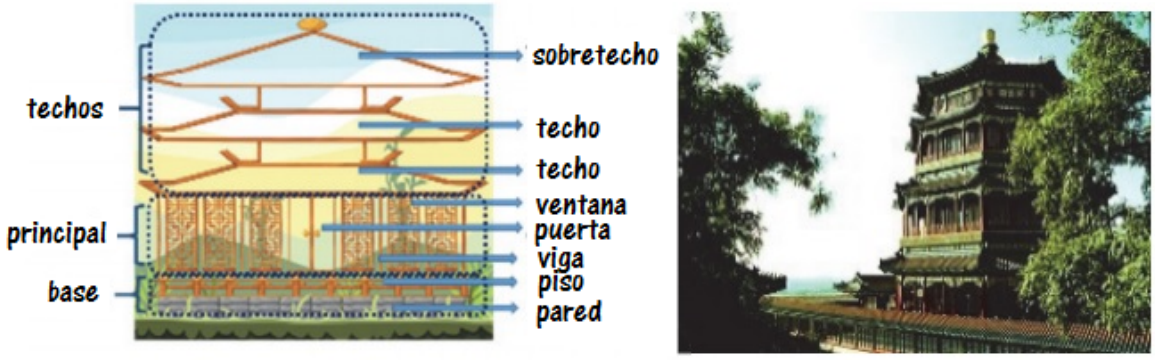

Figura 1.11: Un edificio de estilo chino generada (izquierda) y una edificio similar en el mundo real (derecha) [8].

A continuación se muestra un ejemplo de una regla escrita en 2D-BCG. Un nivel generado por la regla de ejemplo y un edificio verdadero similar a la construcción generados se muestra en la *Figura 1.11*.

```
 ::=
 ::= wall floor
::= beam window window beam window door window
          beam window window beam
::= roof roof toproof
```

Para mantener la coherencia dentro de la estructura del edificio generado, el mismo estilo se utiliza para todas las partes del mismo elemento en la estructura. Por ejemplo, si el edificio tiene dos ventanas, se seleccionará el mismo estilo y se aplica a ambas ventanas.

1.2.2. Método para la Generación Procedimental de Niveles en juegos 2D

El diseñador de juegos Dan Kline ha argumentado que una clave para un buen diseño de juegos con los sistemas de generación procedimental es reemplazar aleatoriedad uniforme con direccionalidad [10].

La utilidad de la investigación depende en parte de la existencia de un conjunto de métodos para PCG-G que se pueden utilizar para generar contenido a través de los diferentes tipos de información recolectada [2]. Estos métodos, que son llamados

fundamentales, podrían a continuación, ser parte de un creador de contenido genérico, y nuestro estudio podría ayudar a aliviar la duplicidad de esfuerzos en esta dirección. Se pueden identificar cinco grupos de métodos para PCG-G. Una visión general de los métodos fundamentales discutidos en este trabajo se representan en la *Figura 1.12*. Los métodos se agrupan en clases tales como *Generador de Números Pseudo-Aleatorios(PRNG), Gramática Generativa (GG), Filtrado de imágenes (SI), Algoritmos Espaciales (SA), Modelado y Simulación de Sistemas Complejos (CS), Inteligencia Artificial (AI)*, etc.

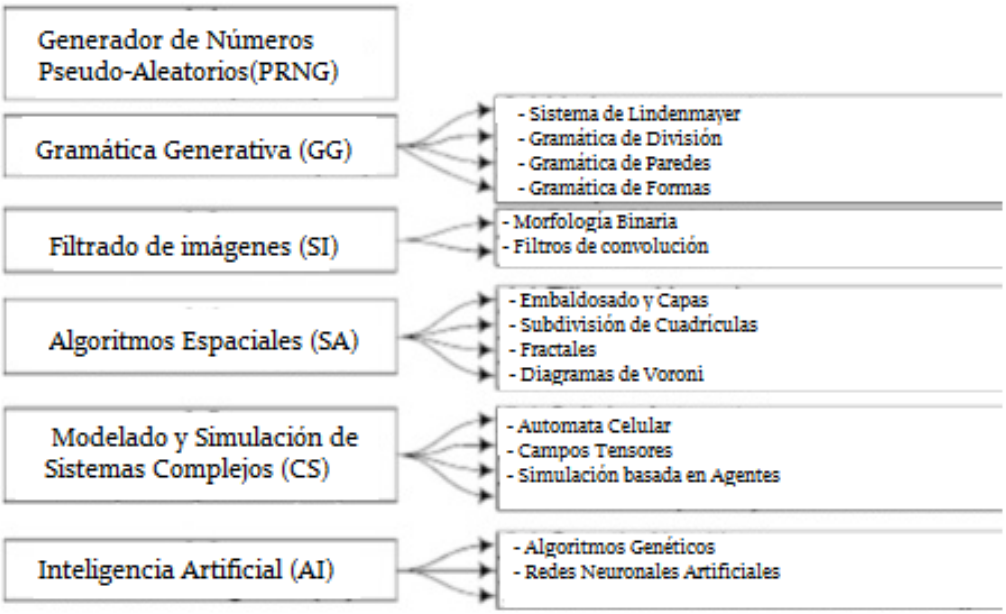

Figura 1.12: Taxonomía de los métodos comunes para la generación de contenido del juego [2].

Como se puede observar en la imagen anterior, cada método tiene asociado los algoritmos más comunes utilizados para aplicar estos métodos.

1. Primero el Generador de Números pseudo-aleatorios (*pseudorandom number generator*, PRNG), el cual da la ilusión de aleatoriedad, más que todo usado para imitar la aleatoriedad de la naturaleza, por ejemplo las formas de una nube, una montaña, fuego, humo, y todo tipo de fenómenos que requieran aleatoriedad sin perder continuidad. El Ruido Perlin, basado en este método, es un generador de ruido utilizado en los efectos especiales, utilizado en imágenes

generadas por computadora para simular variabilidad, acercándose estas así a un aspecto más natural. Esta técnica interpola valores precalculados de vectores construyendo así un valor que varía seudo-aleatoriamente en el tiempo o espacio.

2. En la Gramática Generativa (GG), hay un conjunto de reglas que, a partir de palabras individuales, forman oraciones gramaticalmente correctas. Se puede usar para formar correctamente objetos a partir de elementos codificados como letras o palabras. A continuación algoritmos que se han usado para generación de contenido en el área del entretenimiento
   - Un sistema-L o un sistema de *Lindenmayer* es una gramática formal (un conjunto de reglas y símbolos) principalmente utilizados para modelar el proceso de crecimiento de las plantas. Consiste en un conjunto de símbolos que describen las características de un objeto. Una cadena de caracteres generada por la gramática describirá la estructura y conducta de un objeto.
   - Gramática de División (*Split Grammars*): utilizando reglas de reescritura genera nuevas formas basada en unas ya existentes, como por ejemplo a partir de una puerta puedes sacar una nueva forma de ventana. En esta gramática no importa el contenido propiamente, siempre generará lo mismo sin importar el orden.
   - Gramática de Paredes (*Wall Grammar*): Están específicamente diseñada para crear exteriores de edificios. Manipulas formas del mismo modo que la Gramática de División, pero esta gramática genera figuras más avanzadas, como por ejemplo, generar complejas figuras tridimensionales como balcones.
   - Gramática de Formas: Sensible al contenido, por cada paso de reescritura, el símbolo y sus vecinos en la cadena de caracteres determina que símbolo reemplazará el original. Puede originar estructuras complejas.

3. En el Filtrado de Imágenes (SI), se quiere resaltar características de una imagen. A continuación dos técnicas de procesamiento de imágenes:
   - Morfología binaria (*Binary Morphology*): Son operaciones binarias aplicadas en imágenes convertidas en binarias, dónde los pixeles con poca intensidad se ponen en cero y los demás en uno. Usualmente se sustrae la original del resultado para aplicarle los cambios a la imagen original, en menor tiempo.
   - Filtros de convolución (*Convolution Filters*): La convolución es un operador matemático aplicado en dos funciones, dónde una función modifica a

la otra y crea una nueva, o nueva data. Es utilizada para remover sonidos detectar bordes de objetos, o la dirección del movimiento de los objetos en una imagen. Se puede manipular una textura para crear una completamente nueva, ahorrando almacenamiento.

4. En los Algoritmos Espaciales (SA), se manipula el espacio para generar contenido del juego, la salida es creada a partir de una entrada bien estructurada:
   - Embaldosado y Capas (*Tiling and Layering*): el llamado "tiling" es una técnica usada para crear un espacio del juego descomponiendo el mapa en una cuadrícula. En cambio el "layering" es una técnica que integra en un mismo mapa muchas cuadrículas, llamadas capas. Un "tile" o baldosa es construido sobreponiendo partes de cada capa, así creará un efecto o de agua corriendo, o de un espacio 3D, etc.
   - Subdivisión de cuadrícula (*Grid Subdivision*): Es una técnica iterativa y dinámica para la generación de objetos. Un objeto se divide primero en una cuadrícula uniforme con las texturas apropiadas. Un algoritmo de subdivisión de cuadrícula, se utiliza para añadir detalles iterativamente al objeto. Un ejemplo que utiliza esta técnica es la representación de un terreno generado procedimentalmente: Sólo se detallan (generan) las celdas cercanas al jugador, mientras que el resto del terreno es más grueso, ahorrando así el cómputo.
   - Fractales (*Fractals*): son figuras recursivas que consisten en copias de ellas mismas. Con pocos parámetros se puede controlar un gran rango de posibles resultados. Una ventaja de los fractales es que los objetos con infinitos detalles, se pueden almacenar como una simple función recursiva.
   - Diagramas de Voronoi (*Voronoi Diagrams*) Son descomposiciones de espacios métricos en partes cuyo tamaño y forma está determinado por la posición de puntos de semilla (puntos de interés) en la métrica espacio. Sin embargo, cuando las colecciones de puntos aumentan de tamaño, hay una mejora computacional.

5. Modelado y Simulación de Sistemas Complejos (CS), en algunos casos es poco práctico describir fenómenos naturales con ecuaciones matemáticas. Los modelos y simulaciones pueden ser usadas para solucionar este problema.
   - Autómata Celular (*Cellular Automata*), Un autómata celular es un modelo computacional discreto basado en celdas alineadas en una cuadrícula, donde cada celda tiene un estado y está sujeta a un conjunto común de normas. El modelo computacional se aplica a intervalos de tiempo discretos. Las reglas del tablero determinan cómo el vecindario celular y el

estado influyen en el siguiente estado de la celda. El comportamiento resultante puede ser aleatorio, pero también periódico.

- Campos tensores (*Tensor Fields*) Son generalizaciones bidimensionales de vectores, que se puede utilizar para especificar la forma de un espacio de juego. Los tensores describen la dirección de la elevación del mapa. Debido a que las líneas tensoras pueden ser visualizadas, son adecuadas para el diseño interactivo y la manipulación de las redes de carreteras.
- Simulación basada en agentes (*Agent-based Simulation*, ABS) Se basa en el modelado de una situación compleja utilizando individuos, llamados agentes. Comportamiento emergente, es decir, comportamiento complejo que surge de interacciones de agentes simples, es una característica del ABS que contrasta con el comportamiento promedio observable mediante técnicas de modelización tradicionales. Los agentes se pueden agregar, quitar o reemplazar durante la simulación; Los agentes también pueden aprender con el tiempo.

6. Inteligencia Artificial (*Artificial intelligence*, AI), es un campo largo dentro de la computación, el cual trata la mímica a los animales, o humano inteligente.
    - Algoritmos Genéticos (*Genetic Algorithms*): Se utilizan para resolver problemas de optimización imitando evolución biológica. Las posibles soluciones se codifican como cadenas (cromosomas), y una función de *fitness* se utiliza para evaluar la calidad de una solución. Una mutación y función de cruce se aplican a crear nuevas soluciones. La función de mutación convierte una solución en una nueva. La función de *crossover* especifica el intercambio de partes cromosómicas entre un conjunto de cromosomas progenitores. La tasa de mutación y tasa de crossover determinan la frecuencia que estas operaciones producen.
    - Redes neuronales artificiales (*Artificial Neural Networks*, ANN) Son modelos computacionales con la capacidad para aprender la relación entre una entrada y una salida minimizando el error entre la salida y la salida. Rendimiento esperado. Las ANN pueden usarse para encontrar patrones y clasificar, recordar y estructurar datos. Una ANN consiste en unidades computacionales llamadas neuronas, que están conectadas por bordes ponderados. Una sola neurona puede tener varios bordes entrantes y salientes. Cuando una neurona recibe una entrada, primero combina las entradas de todos los bordes entrantes y las pruebas si es activada por esta entrada. Si la neurona se dispara, envía la señal combinada sobre las líneas de salida. La ANN funciona en un entorno, que proporciona las señales de entrada, procesa las señales de salida, y calcula el error que la ANN puede utilizar para ajustar el peso en los bordes y así aprender.

- Satisfacción y Planificación de Restricciones (*Constraint Satisfaction and Planning*) Implica encontrar un camino desde un estado inicial a un estado final aplicando acciones. Un problema de planificación consiste en un estado inicial, acciones, y una prueba de meta. Planificación de la definición de dominio del lenguaje (*Planning Domain Definition Language*, PDDL) se utiliza comúnmente para expresar problemas de planificación. Una acción se puede ejecutar cuando se satisface la condición inicial. El efecto de una acción puede ser la adición o supresión de variables que resultan en un nuevo estado. Búsqueda avanzada de espacio de estado los algoritmos comienzan a planear desde el estado inicial. En contraste, los algoritmos de búsqueda de espacio de estado hacia atrás comenzar en el estado final. A pesar de ambos tipos de algoritmos, la planificación es NP-complejo en general, lo que explica la importancia de la heurística en la planificación.

La generación de nivel es en la actualidad uno de los tipos más populares de PCG-G. Aunque casi todos los géneros pueden beneficiarse de los niveles generados, juegos de plataformas en 2D y rompecabezas tienen especial atención. Compton y Mateas [17] (2006) y Smith et al. [6] (2009) generan niveles de plataformas en 2D basado en el concepto de ritmo. En relación, Shaker et al. [25] (2010) y Jennings-Teats et al. [21] (2010) demuestran que los niveles personalizados se pueden generar en línea para los juegos de plataformas basadas en un modelo del jugador. Más general que las plataformas, Dormans [26] (2010) utiliza una gramática para crear una estructura de misiones utilizando grafos, que luego se traduce a un nivel 2D mediante el uso de una gramática.

2

Solución propuesta

2.1. Arquitectura

Se plantea en este trabajo de investigación el desarrollo de un algoritmo que genere niveles procedimentalmente en un videojuego, logrando así un juego no repetitivo, donde cada instancia será distinta entre sí, sin tener algún costo en memoria significativo. Este algoritmo clasificado dentro de la taxonomía planteada por Shaker et al. [1], cumplirá las siguientes características:

- *Offline*, fuera de línea.
- El contenido generado con PCG será necesario para el funcionamiento del juego
- Usará semillas aleatorias, el contenido generado podría repetirse
- Será genérico, no se adaptará al usuario
- Contenido determinista, al repetir los parámetros y puntos iniciales se podría repetir una misma instancia de juego
- Contenido constructivo, será generado antes de que empiece el juego.
- Generación de Autoría Mixta, el algoritmo genera por si solo el contenido pero recibe parametros del desarrollador.

Serán creados distintos casos de pruebas que serán evaluados por varios usuarios

para evaluar la efectividad del método implantado, y finalmente se tomarán los datos e información necesaria para seleccionar los valores óptimos y generar así una mejor versión del videojuego.

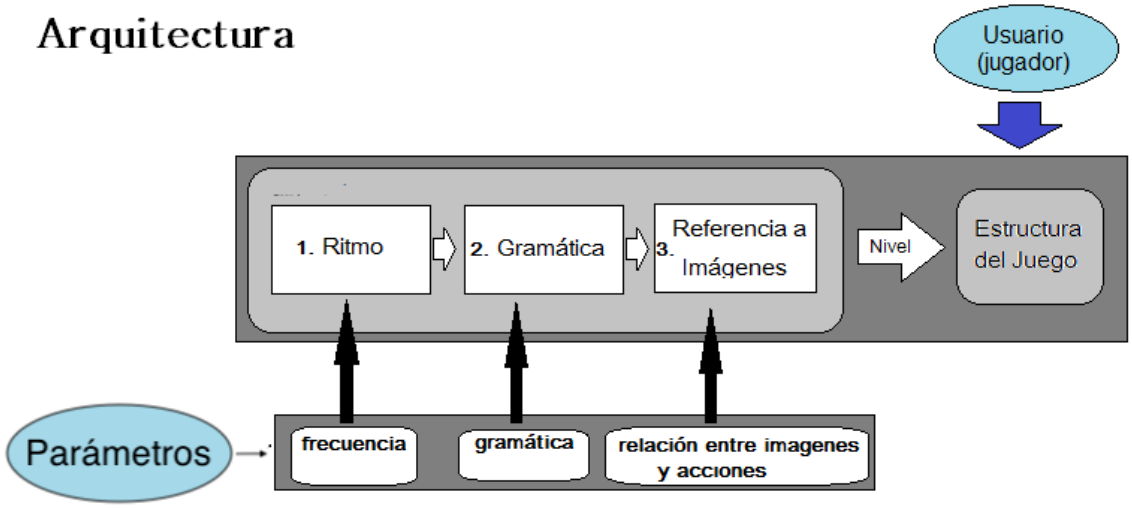

Figura 2.1: Arquitectura de la solución al problema.

En la *Figura 2.1* se puede observar los módulos propuestos para la solución al problema. Se generará un algoritmo con tres módulos, que recibirán distintos parámetros de entrada por el desarrollador. Estos algoritmos proporcionarán la información necesaria para generar un nivel de juego o estructura de juego final, el cual finalmente será jugado por el usuario. A continuación se explicará módulo por módulo.

Módulo 1: Ritmo. Basado en el trabajo de Smith [6], en esta fase inicial se recibirá la frecuencia en la que se desea que ocurran las acciones, controlando así la dificultad del juego, mientras mayor sea la frecuencia más rápido tendrá que reaccionar el jugador.

La frecuencia dirá cada cuántos segundos se requiere que una acción se tenga que ejecutar. Dependiendo de la dificultad del nivel la frecuencia será mayor, o menor. Un nivel podría presentar una acción cada 1 segundo, o en su defecto cada cierta distancia. Como se puede observar en la *Figura 2.2*

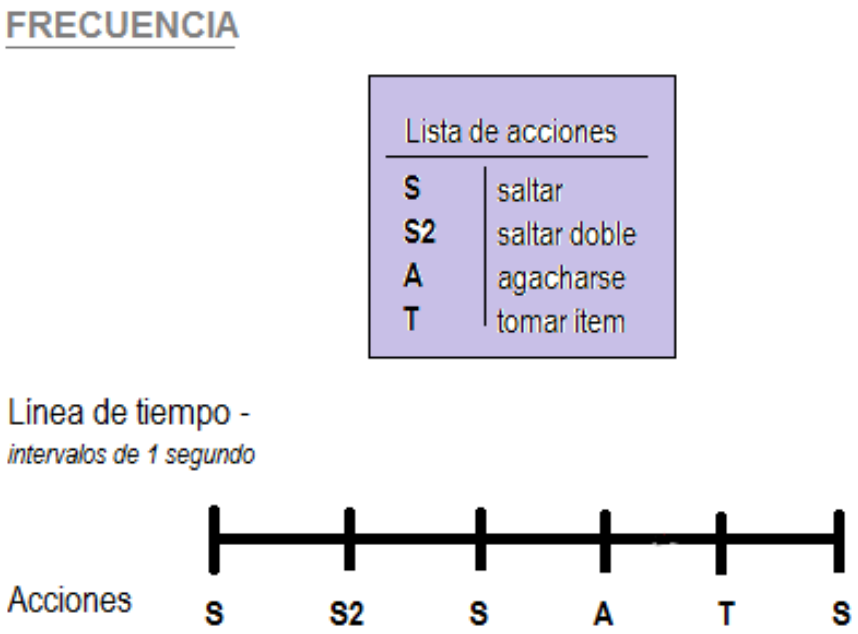

Figura 2.2: Frecuencia. Tiempo de aparición entre acciones

Módulo 2: Gramática. Se crea una gramática, para ser utilizada como restricciones para la generación de múltiples y variadas secuencias de acciones *Figura 2.3*. Cada Secuencia tendrá probabilidad aleatoria de ocurrir y siempre tendrá un símbolo no terminal presente, de manera que se puedan generar infinitas secuencias de acciones.

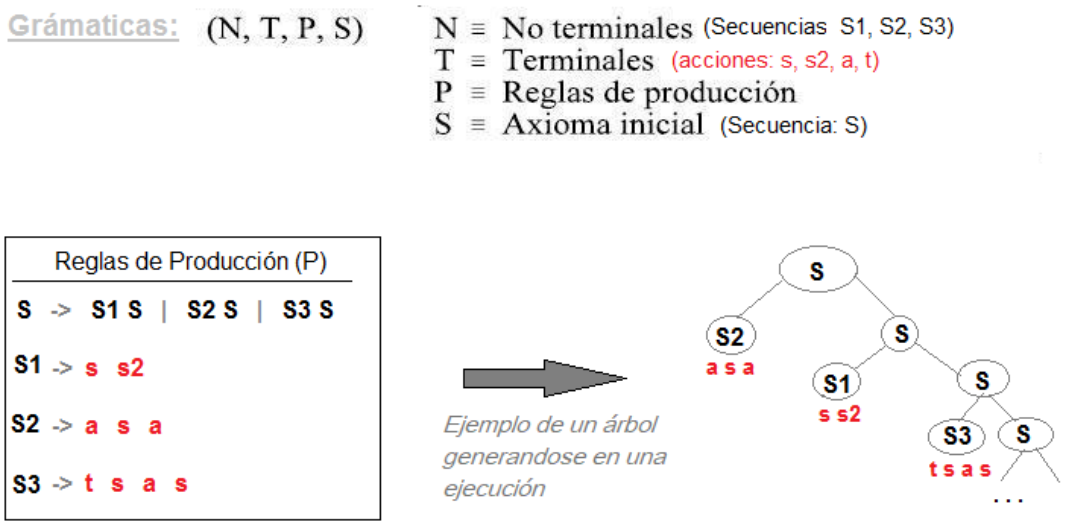

Figura 2.3: Gramática. Usada Para generar secuencias de acciones

Módulo 3: Referencia a Imágenes *Figura 2.4*, En este último módulo ya se tiene una secuencia de acciones dadas por el módulo anterior, ahora cada acción podrá tener distintas imágenes para ser representada, se crea una lista de nombres asociados a imágenes por cada acción a representar, cada imagen tendrá con una probabilidad distinta de aparición.

Referencia a Imágenes

Acción	Identificador de imágen	Probabilidad de Aparición
S (saltar)	Alcantarilla	0.5
	Moneda	0.2
	Arbusto	0.3
S2 (saltar doble)	Andamio	0.4
	Techo	0.6

Figura 2.4: Referencia a Imágenes.

Al terminar todos los módulos, se obtiene una secuencia de imágenes, que siguen un ritmo basado en la gramática desarrollada, que serán procesadas junto con la estructura del juego ya definida.

Por último el jugador experimentará siempre un juego diferente.

La dificultad del juego estará basada en varios parámetros, en el ritmo dado, y también se debe desarrollar distintas gramáticas con dificultades variadas.

3

Implementación

3.1. Generación Procedimental de Contenido

Fue desarrollado un conjunto de algoritmos para permitir la generación procedimental de contenido, específicamente de los elementos con los que interactúa el jugador, utilizando gramáticas y probabilidades, garantizando la creación de un videojuego 2D de niveles orientado, pero siempre variable.

Estos algoritmos fueron desarrollados en su totalidad en el lenguaje de programación C#, en la plataforma de desarrollo de videojuegos Unity (*Figura 3.1*), en conjunto con el entorno de desarrollo integrado Microsoft Visual Studio.

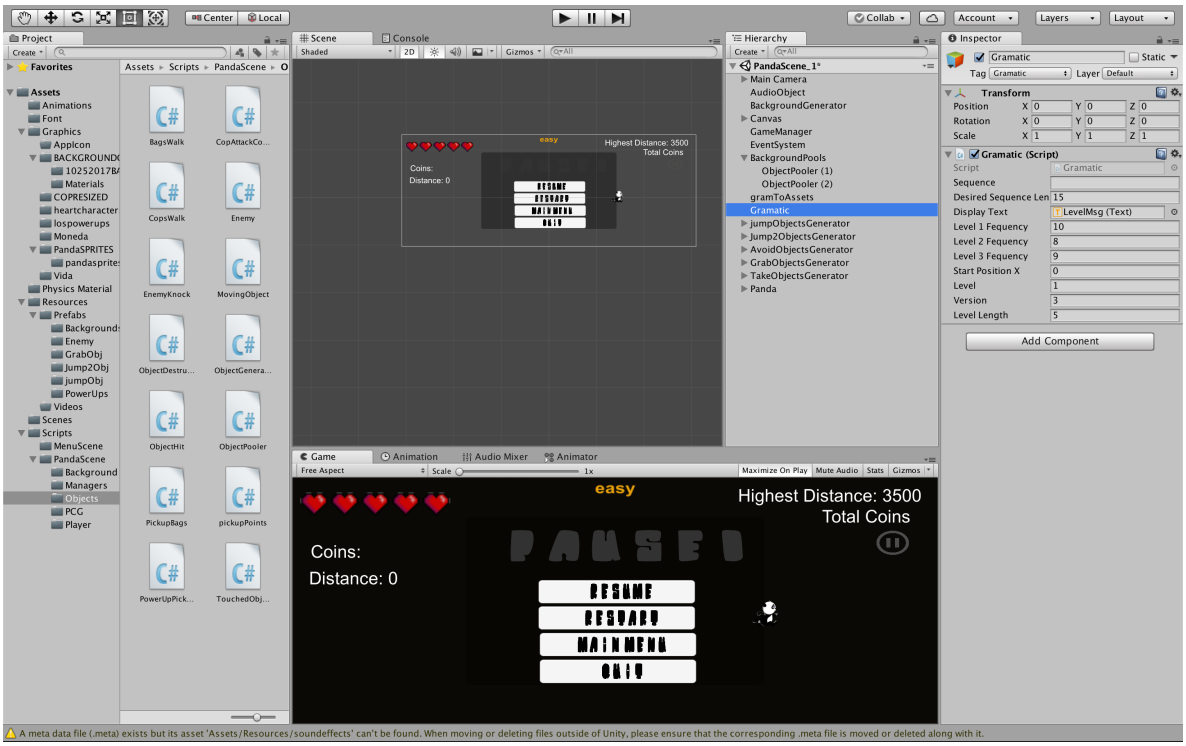

Figura 3.1: Captura de Pantalla de Unity.

El método algorítmico implementado cuenta con tres puntos importantes:

1. **Gramática:** El funcionamiento de una gramática preestablecida de forma que se generen secuencias de acciones siempre que se requiera. Con la existencia de una gramática distinta por cada nivel.

2. **Imágenes:** La asignación de una imagen (objeto) correspondiente a la acción, utilizando probabilidades predefinidas según el nivel.

3. **Reproducción:** Mostrar este contenido visual, eliminándolo después de ser utilizado para la liberación de memoria.

Se tienen parámetros importantes que también fueron utilizados, como la distancia entre imágenes de tipo obstáculo (“frequencyDistance”), para lograr un ritmo variado en el juego según el nivel; y la duración de cada nivel, basada en la cantidad de fondos que se agreguen (“levelLenght”).

3.1.1. Gramática

"Gramatic" es una clase dentro del archivo Gramatic.cs, en el cual se genera una gramática, la cual es utilizada siempre que se desee para generar secuencias de acciones.

La gramática está definida como una variable de tipo *Dictionary* de cadena de caracteres o *string*, y una lista de listas de *string* (*Código* 1).

```
Dictionary < string, List < List < string > > > gram;
gram = new Dictionary < string, List < List < strin g> > > ( );
```

Código 1: *Fracción del contenido del archivo Gramatic.cs, donde se define la variable de la gramática, "gram".*

Se crea la gramática con sus símbolos terminales y no terminales, que serán definidos posteriormente (*Código* 2).

```
// "S" es el axioma inicial que contiene todos los no terminales que seran escogidos al azar

// en el diccionario(gram) la definicion de "S" esta dada por s
AxiomaInicial = "S";

s = new List<List<string>>();
s.Add (new List<string>() { "S1", "S" });
s.Add (new List<string>() { "S2", "S" });
s.Add (new List<string>() { "S3", "S" });
s.Add (new List<string>() { "S4", "S" });

// gram es un diccionario de lista de listas
// "S" esta definido por la lista de listas : s, y asi con "S1", "S2", "S3" y "S4".
// s1 s2 s3 y s4 son listas compuestas por una lista de acciones (string), cada accion es un
    caracter
gram.Add("S", s);
gram.Add("S1", s1);
gram.Add("S2", s2);
gram.Add("S3", s3);
gram.Add("S4", s4);

// Se identifican los no terminales
noTerm.Add("S");
noTerm.Add("S1");
noTerm.Add("S2");
noTerm.Add("S3");
noTerm.Add("S4");
```

Código 2: *Fracción del contenido del archivo Gramatic.cs, donde se inicializan las distintas variables.*

Se llena la gramática con secuencias de acciones diferentes, de la siguiente manera (*Código* 3).

```
s1 = new List<List<string>>();

s1.Add(new List<string>() { "n", "g", "n", "n", "j", "t", "j", "t", "g", "a" });

s2 = new List<List<string>>();

s2.Add(new List<string>() { "J", "t", "n", "j", "g", "a", "n", "t", "n", "g" });


s3 = new List<List<string>>();

s3.Add(new List<string>() { "t", "J", "J", "a", "n", "n", "g", "n", "t", "g" });


s4 = new List<List<string>>();

s4.Add(new List<string>() { "t", "g", "n", "n", "n", "t", "g", "a", "j", "j" });
```

Código 3: *Fracción del contenido del archivo Gramatic.cs, donde se inicializan las distintas variables.*

Para crear esta secuencias, se basa en las acciones que el jugador debe realizar al interactuar con el juego, fueron agrupadas por la reacción que el jugador deba tener ante algún objeto. Se especifican a continuación:

- "n": *nothing*, al jugador no se le presenta ningún objeto
- "g": *grab*, agarra una vida
- "t": *take*, toma un poder positivo
- "j": *jump*, salta
- "J": *double Jump*, salta doble
- "a": *avoid*, evita un enemigo o poder negativo

Cada una de estas acciones requiere una interacción distinta del jugador. Para agarrar una vida en la versión móvil del juego, se toca el lado izquierdo de la pantalla,

para saltar se toca el derecho, para saltar doble se toca dos veces de ese mismo lado. Las otras dos acciones involucra esquivar para evitar el enemigo o en su defecto dirigirse hacia un poder positivo.

Para escoger estas secuencias de acciones, se basa en un porcentaje asignado por nivel, dividido en las acciones positivas ("n", "g" y "t") y las acciones que obstaculizan ("j", "J" y "a"), en ese orden respectivo se presentan a continuación:

- Nivel 1: 70 % y 30 %
  Nivel 2: 50 % y 50 %
  Nivel 3: 30 % y 70 %

Para tener diferencias más detalladas también se le asigna de esa fracción de porcentaje una preestablecida por acción:

- Nivel 1:
  - Positivo 70 %
    n: 30 %
    g: 20 %
    t: 20 %
  - Negativo 30 %
    j v J: 20 %
    a: 10 %
- Nivel 2:
  - Positivo 50 %
    n: 10 %
    g: 20 %
    t: 20 %
  - Negativo 50 %
    j v J: 30 %
    a: 20 %
- Nivel 3:
  - Positivo 30 %
    n: 0 %

g: 10 %
t: 20 %

- Negativo 70 %
  j v J: 40 %
  a: 30 %

En el momento de crear las gramáticas, al escoger las secuencias de elementos terminales, se basa en esos porcentajes para crear varias combinaciones. A cada no terminal se le asignan secuencias de 10 acciones.

La forma de la gramática creada se muestra a continuación

```
S—> s1 S | s2 S | s3 S

s1—> n g n n j t j t g a
s2—> J t n j g a n t n g
s3—> t J J a n n g n t g
```

Cumpliendo con los porcentajes, tomando el ejemplo anterior que corresponde a un nivel 1, se observa como fue aplicado al "s1". De las 10 acciones en la secuencia un 30 % corresponde a *n*, es decir deben aparecer tres *n*; un 20 % de *t* es decir dos veces, al igual que *g*; también un 20 %, o sea dos acciones de *j* o *J* indistintamente; y por último sólo un *a*, que correspondería a un 10 %. De este manera se crean secuencias de orden distinto pero repitiendo las reglas por nivel, para garantizar la dificultad del mismo.

Para generar secuencias de acciones utilizando la gramática fue realizado un algoritmo recursivo, el cual se muestra en el código 4.

```
public void GetSequenceRec(ref List<string> strgSequence)
 {
     //Almacenas en "strg" el primer string o caracter (terminal o no terminal)
     string strg = strgSequence[0];

     //Si "strg" es un NoTerminal entra en el condicional
     if (noTerm.Contains(strg)) //gramlist[i])) // == "S1" || gramlist[i] == "S2" ||
         gramlist[i] == "S3" || gramlist[i] == "S")
     {
         //Numero aleatorio entre la cantidad de elementos que tiene el NO terminal "strg"
         int random = rnd.Next(0, gram[strg].Count);

         //"gramlist": elemento aleatorio dentro del NO terminal "strg"
         gramlist = gram[strg][random];
```

```
            //Si "desiredSequenceLenght" (la longitud deseada de la secuencia) es mayor que el
                tamano de las veces que se ha llenado la secuencia (tamano de la secuencia)
            if (desiredSequenceLenght > seqSize)
            {
                //Elimina la posicion ya chequeada de "strgSequence" (la 0 )
                strgSequence.RemoveAt(0);

                //Almacena en el auxiliar "production" lo obtenido dentro del No terminal
                    chequeado en esta iteracion
                //Se le agrega la secuencia que aun no se ha revisado de "strSequence"
                List<string> production = new List<string>();
                production.AddRange(gramlist);
                production.AddRange(strgSequence);

                //Se llena "strgSequence" con lo acumulado hasta ahora (que se lleno en la
                    variable auxiliar "production")
                strgSequence = production;

                //Se llama recursivamente a la funcion nuevamente para analizar su primera
                    posicion otra vez, hasta que se cumpla la condicion de la longitud de la
                    secuencia
                GetSequenceRec(ref strgSequence);
            }
            //------------CONDICION DE PARADA RECURSION
            else
        }
        //Si no se consigue con cadena de terminales
        else
        {
            string t = strgSequence[0];

            //Borras el no terminal que quedo expresado
            strgSequence.RemoveAt(0);

            //"seqSize" es la cantidad de veces que se consiguio terminales en toda la gramatica
            seqSize++;

            GetSequenceRec(ref strgSequence);

            strgSequence.Insert(0, t);
        }
    }
}
```

Código 4: *Fracción del contenido del archivo Gramatic.cs donde se muestra el algoritmo recursivo para generar gramáticas.*

Esta acción recursiva es llamada desde la acción “GenerateSequence()”, donde se determina el nivel en el que se está para escoger la gramática adecuada con la función “ PickLevelGramatic(int x)”. La variable “backCounter” es de tipo de dato entero, esta lleva la cuenta de cuantos fondos se han creado, para determinar que tan largo será el nivel, de forma que “levelLength” es un parámetro de entrada que permite definir al desarrollador dicha longitud.

```
public void GenerateSequence()
{
    seqIsReady = false;
    //————————————————Cambio de nivel————————————
    //backCounter: contador de cuantos fondos se han creado para ver si se hace cambio de
        nivel
    if (backCounter == levelLength)

    {
        //if first time
        if (firstTime)
        {
            level = 2;
            frequencyDistance = level2Fequency;
            displayText.text = "MEDIUM";
            displayText.color = new Color(238f/255f,164f/255f,11f/255f);

            firstTime = false;
            secondTime = true;
            PickLevelGramatic(2);
        }
        //if second time
        else if (secondTime)

    }
        //reinicia los fondos porque empieza un nuevo nivel
        backCounter = 0;
     //————————————————————————————————————


    }

    //strSeq lista que almacenara todas las acciones mientras se recorre la gramatica
    List<string> strSeq = new List<string>();
    strSeq.Add(AxiomaInicial);

    seqSize = 0;

    //————————————Recursion para generar secuencia————————————
    GetSequenceRec(ref strSeq);
    //————————————————————————————————————

    /* Al salir de la recursion, strSeq esta lleno de la secuencia de acciones*/
```

```
        //Se le elimina el caracter no terminal que queda planteado, antes de utilizar la cadena
            de acciones luego en GramaticToAssets

        strSeq.RemoveAll(item => item == "S");

        //Lenar Sequence para usarla luego en "GramatictoAssets"
        foreach (string s in strSeq)
            sequence += s;

        //contador de los fondos por nivel
        backCounter += 1;

        //booleano para ejecutar codigo en Update()
        seqIsReady = true;

        //duplicar desiredSequenceLenght cada vez que se agregue un fondo nuevo para
            aumentar la cantidad de imagenes en escena
        //menos la primera corrida
        if (firstBG)
            firstBG = false;
        else
            totalDesiredSequenceLenght += desiredSequenceLenght;

    }
```

Código 5: *Fracción del contenido de la clase Gramatic.cs donde ya al tener completa la secuencia, se pasa carácter por carácter a una función que los asociará a la imagen correspondiente.*

3.1.2. Imágenes: Gramática a Imágenes

Posteriormente cuando la secuencia está lista se pasa carácter por carácter a la clase GramtoAssets del archivo GramtoAssets.cs que se usarán en una función para asociar cada carácter a una imagen, como se muestra en el código 6. Se toma en cuenta la frecuencia entre objetos predefinida, “frequencyDistance” tomando una distancia inicial, y luego por cada carácter tomado se va sumando esta distancia frecuencial.

La variable “frequencyDistance” tendrá un valor distinto por cada nivel, logrando así utilizar la frecuencia de aparición de obstáculos como un parámetro de dificultad que le cambia el ritmo al juego.

```
    void Update()
    {
        if (seqIsReady)
        {

            //Es pasada letra por letra a GramaticToAssets.cs
            // Se cumple la condicion de que k no se pase del tamano de la sequencia pero se
                pasara nada mas el tamano deseado

            if (k < sequence.Length && k < totalDesiredSequenceLenght)
            {

             //se pasa el caracter de la secuencia para convertirlo en imagen y la posicion
                almacenada donde quedo la ultima imagen generada

                gramToAssets.SecuenceLoop(sequence[k], startPositionX);
                k++;
                startPositionX += frequencyDistance;
            }
        }
    }
}
```

Código 6: *Fracción del contenido la clase Gramatic.cs donde ya al tener completa la secuencia, se pasa carácter por carácter a una función que los asociará a la imagen correspondiente.*

En la clase GramaticToAssets está la función llamada desde Gramatic.cs donde se llama a la generación del objeto, asociándolo a su imagen correspondiente, como se muestra en el código 7.

```
public void SecuenceLoop(char letter, float xPosition)
    {
        //action
        if (gramatic)
        {
            if ( letter == 'j' )
                thejumpObjGenerator.GenerateAsset(xPosition);

            if ( letter == 'J' )
                theJump2ObjGenerator.GenerateAsset(xPosition);

            if ( letter == 'g')
                theGrabObjGenerator.GenerateAsset(xPosition);

            if ( letter == 'a' )
                theAvoidObjGenerator.GenerateAsset(xPosition);
```

```
            if (letter == 't')
                theTakeObjGenerator.GenerateAsset(xPosition);
        }
    }
}
```

Código 7: Fracción del contenido del archivo GramaticToAssets.cs donde se llama a la generación del objeto, asociandolo a su imagen correspondiente.

La variable "xPosition" es para mantener la cronología de las posiciones de cada objeto, inicialmente empieza con un valor inicial dado en la clase Gramatic, y posteriormente se le va sumando la frecuencia por cada carácter de acción utilizado. Como se mencionó anteriormente este parámetro de frecuencia ("frequencyDistance"), es la distancia entre obstáculos, y este será diferente cada nivel, para garantizar un ritmo que vaya acorde a la dificultad.

"GenerateAsset()" es una función definida en la clase ObjectGenerator, del archivo ObjectGenerator.cs, por lo tanto como se puede observar en el código 8, objetos como "thejumpObjGenerator" son de tipo ObjectGenerator.

```
ObjectGenerator thejumpObjGenerator;
ObjectGenerator theJump2ObjGenerator;
ObjectGenerator theGrabObjGenerator;
ObjectGenerator theDuckObjGenerator;

thejumpObjGenerator =
    GameObject.FindGameObjectWithTag("jumpObjs").GetComponent<ObjectGenerator>();
theJump2ObjGenerator =
    GameObject.FindGameObjectWithTag("Jump2Objs").GetComponent<ObjectGenerator>();
theGrabObjGenerator =
    GameObject.FindGameObjectWithTag("GrabObjs").GetComponent<ObjectGenerator>();
theDuckObjGenerator =
    GameObject.FindGameObjectWithTag("DuckObjs").GetComponent<ObjectGenerator>();
```

Código 8: Fracción del contenido del archivo GramatiToAssets.cs donde se definen e inicializan las diferentes variables.

3.1.3. Reproducción: Generación de Objetos

Por último se tiene la clase que muestra en la escena los objetos de tipo "ObjectGenerator" en el archivo "ObjectGenerator.cs". En el código 9 se muestra como la selección del objeto esta basada en las probabilidades que fueron previamente definidas según el nivel.

```
public void GenerateAsset(float xPosition)
{
    //————————Generar numero aleatorio basado enprobabilidades——————————

        double rand = Random.Range(0.0f, 1.0f);
        double cumulative = 0.0;

        for (int i = 0; i < theObjectPools.Length; i++)
        {
            cumulative += objsProbabilities[i];

            if (rand < cumulative)
            {
                obstacleSelector = i;
                break;
            }
        }
    }

    gramToAssets.objPosition = gramToAssets.objPosition + (obstacleWidths[obstacleSelector] /
        2) + frequencedistance;
    transform.position = new Vector3(xPosition, transform.position.y, transform.position.z);

    GameObject newObject = theObjectPools[obstacleSelector].GetPooledObject();

    newObject.transform.position = transform.position;
    newObject.transform.rotation = transform.rotation;
    newObject.SetActive(true);
    }
```

Código 9: Fracción del contenido del archivo ObjectGenerator.cs donde seleccionan las imagenes según sus probabilidades de aparición, y posteriormente se ubican en el espacio.

Por último, se elimina el contenido que se va generando, a medida que va saliendo de la escena. En el código 10, se muestra la clase ObjectDestructor, donde ocurre este proceso.

```
public class ObjectDestructor : MonoBehaviour
{
    private GameObject backgroundDestructionPoint;

    void Start ()
    {
        backgroundDestructionPoint = GameObject.Find("ObjectDestructionPoint");
    }

    void Update()
    {
        if (transform.position.x < backgroundDestructionPoint.transform.position.x)
        {
            gameObject.SetActive(false);
        }
    }
}
```

Código 10: *Fracción del contenido del archivo ObjectDestructor.cs donde se elimina el contenido que se va generando, a medida que va saliendo de la escena.*

4

Pruebas y resultados

Se realizaron 3 modelos de juego, o versiones , con distintas gramáticas, distancia entre obstáculos, distancia entre niveles y probabilidades de aparición. A parte se creó un cuestionario con una serie de preguntas, asociadas al funcionamiento de nuestra gramática, de este modo se pudo hacer un análisis cuantitativo y cualitativo sobre las técnicas implementadas, para finalmente optimizar y crear la versión final. En la *Figura 4.1*, *Figura 4.2* y *Figura 4.3*, se muestran algunas capturas de pantalla del juego en ejecución.

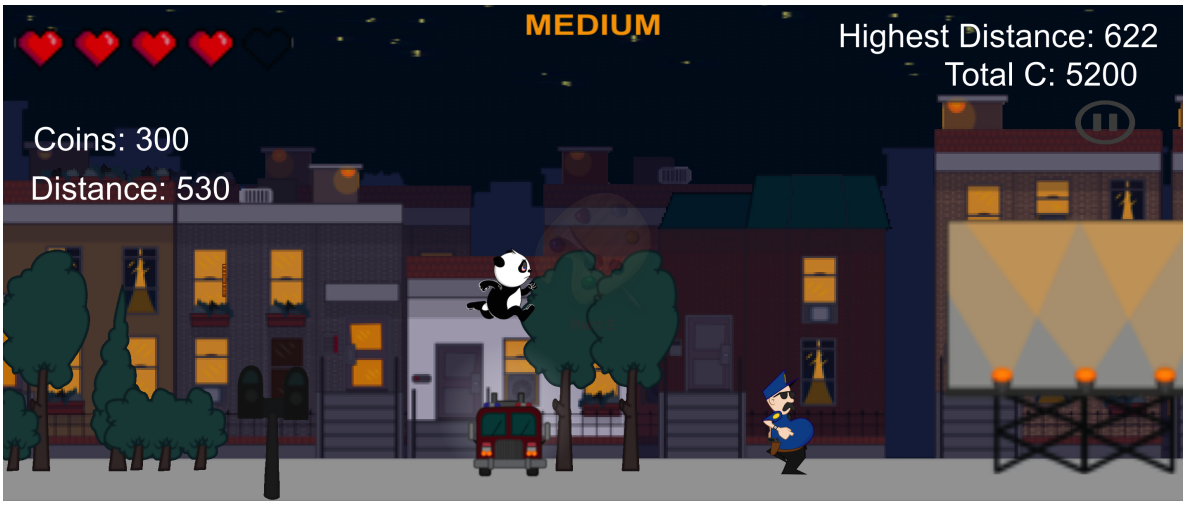

Figura 4.1: Captura de pantalla del juego en ejecución 1

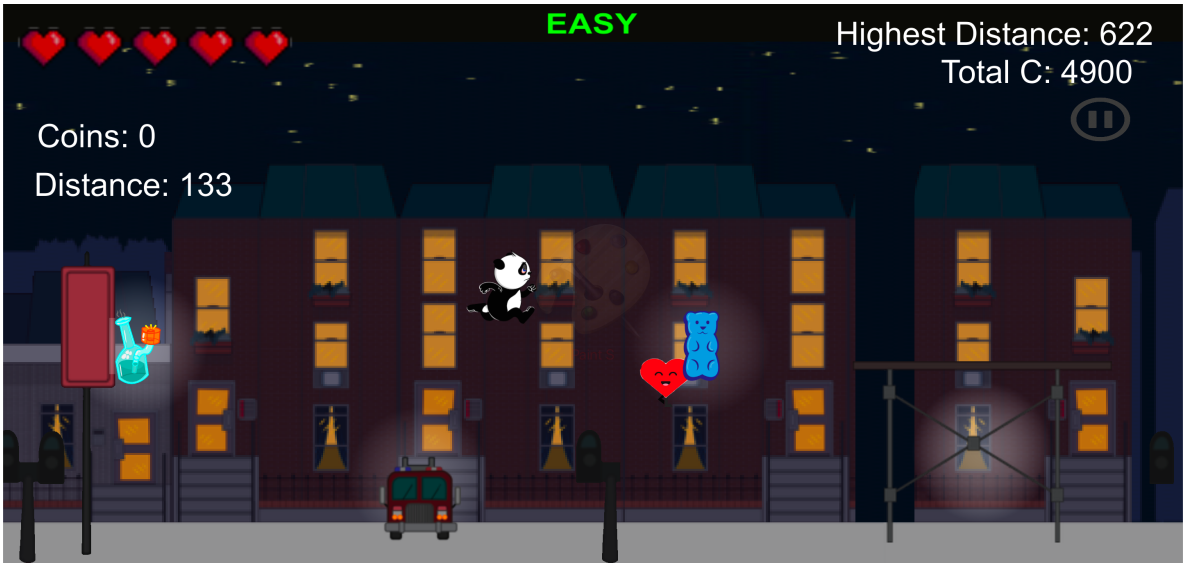

Figura 4.2: Captura de pantalla del juego en ejecución 2

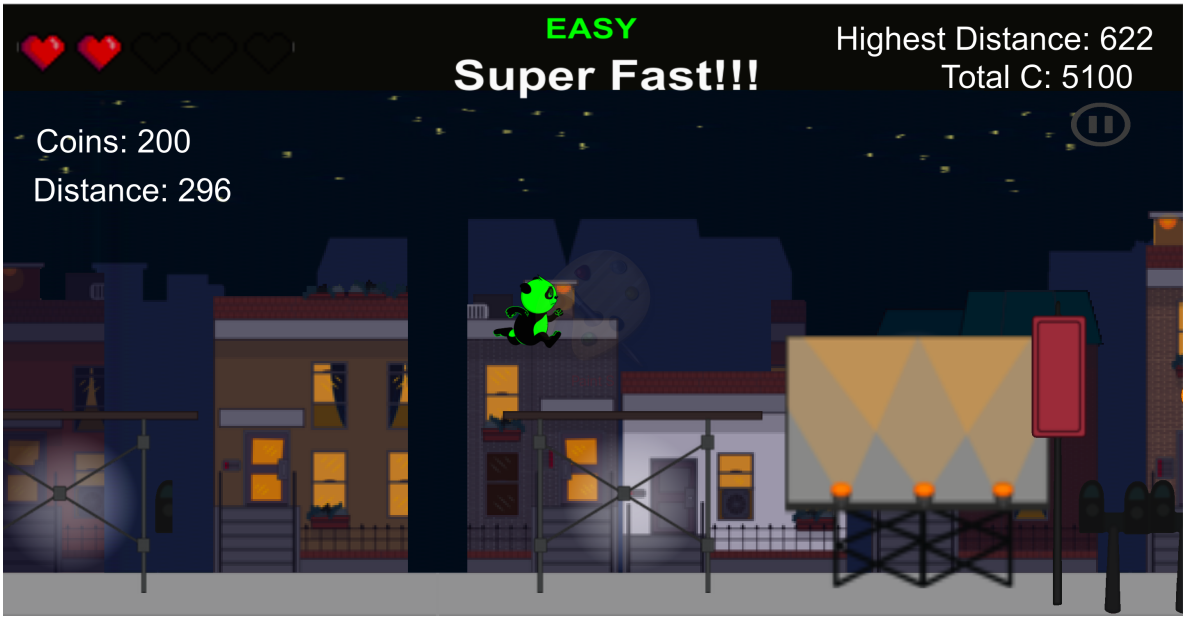

Figura 4.3: Captura de pantalla del juego en ejecución 3

La gramática es la base de las diferencias entre versiones del juego, se muestran a continuación en las siguientes figuras (*Figura 4.4*, *Figura 4.5* y *Figura 4.6*). Estas fueron elegidas manualmente, haciendo combinaciones coherentes, basada en las reglas explicadas en el capítulo anterior.

```
71      //-------------------------Juego 1---------------------------------//
72      /*
73       ** ACTIONS **
74       * n: nothing (just walk)
75       * g: grab (life,touch screen)
76       * t: take (a power up or a coin)
77       * j: jump (an obstacle)
78       * J: double jump (an obstacle)
79       * a: avoid (an enemy or bad Power Up)
80
81       */
82
83      //-------------nivel 1-1
84      if (version == 1)
85      {
86          s1 = new List<List<string>>();
87
88          s1.Add(new List<string>() { "n", "g", "n", "n", "j", "t", "j", "t", "g", "a" });
89
90          s2 = new List<List<string>>();
91
92          s2.Add(new List<string>() { "J", "t", "n", "j", "g", "a", "n", "t", "n", "g" });
93
94
95          s3 = new List<List<string>>();
96
97          s3.Add(new List<string>() { "t", "J", "J", "a", "n", "n", "g", "n", "t", "g" });
98
99
100         s4 = new List<List<string>>();
101
102         s4.Add(new List<string>() { "t", "g", "n", "n", "n", "t", "g", "a", "j", "j" });
103     }
104
```

Figura 4.4: Gramática del juego para la versión 1

```
105
106     //-------------------------Juego 2-----------------------------//
107     //-------------nivel 1-2
108     else if (version == 2)
109     {
110         s1 = new List<List<string>>();
111
112         s1.Add(new List<string>() { "n", "n", "n", "g", "t", "a", "t", "J", "J", "g" });
113
114         s2 = new List<List<string>>();
115
116         s2.Add(new List<string>() { "g", "n", "j", "n", "j", "n", "g", "t", "a", "t" });
117
118
119         s3 = new List<List<string>>();
120
121         s3.Add(new List<string>() { "j", "n", "n", "a", "j", "g", "g", "t", "n", "t" });
122
123
124         s4 = new List<List<string>>();
125
126         s4.Add(new List<string>() { "g", "t", "g", "n", "n", "t", "a", "n", "j", "j" });
127     }
128
129
```

Figura 4.5: Gramática del juego para la versión 2

```
130     //-------------------------Juego 3-----------------------------//
131     //-------------nivel 1-3
132     else if (version == 3)
133     {
134         s1 = new List<List<string>>();
135
136         s1.Add(new List<string>() { "n", "g", "t", "j", "a", "J", "t", "g", "n", "n" });
137
138         s2 = new List<List<string>>();
139
140         s2.Add(new List<string>() { "J", "J", "t", "g", "t", "g", "n", "n", "n", "a" });
141
142
143         s3 = new List<List<string>>();
144
145         s3.Add(new List<string>() { "n", "a", "n", "t", "t", "g", "j", "n", "J", "g" });
146
147
148         s4 = new List<List<string>>();
149
150         s4.Add(new List<string>() { "n", "g", "t", "j", "a", "n", "g", "t", "J", "n" });
151
152     }
153
154
```

Figura 4.6: Gramática del juego para la versión 3

Se modificaron las probabilidades de aparición de obstáculos por versión. En la *Tabla* 4.1 se muestra la selección de probabilidades por nivel (N1,N2 y N3) de cada versión del juego, y grupo de imágenes por acción.

Cuadro 4.1: Tabla de probabilidades de aparición por

Acción	Imagen	Versión 1			Versión 2			Versión 3		
		N1	N2	N3	N1	N2	N3	N1	N2	N3
j	5 Objetos	1	1	1	1	1	1	1	1	1
t	Moneda	0.5	0.4	0.4	0.6	0.3	0.3	0.7	0.5	0.4
	Poder Rápido	0.3	0.3	0.2	0.2	0.2	0.5	0.1	0.3	0.3
	Poder Saltos	0.2	0.3	0.4	0.2	0.5	0.2	0.2	0.2	0.3
J	2 Objetos	1	1	1	1	1	1	1	1	1
g	1 vida	1	1	1	1	1	1	1	1	1
a	Policía	0	0.4	0.6	0.3	0.5	0.6	0.1	0.6	0.9
	Poder Lento	1	0.6	0.4	0.7	0.5	0.4	0.9	0.4	0.1

Estas fueron seleccionadas manualmente, considerando el aumento de dificultad, y el tipo de objeto.

Por último, en cada versión se escogieron distintas duraciones de los niveles, y frecuencia de aparición de obstáculos, o en otras palabras la distancia entre los objetos que aparecen.

El juego se realizó con un solo fondo que se repite infinitas veces mientras va avanzando, utilizamos estos de base para definir la duración ideal de un nivel. Cada fondo es recorrido aproximadamente de 8 a 10 segundos, dependiendo del jugador y los objetos que se presenten.

Duración de cada nivel, basada en cantidad de fondos

```
Versión 1: Cada nivel dura 4 fondos.
Versión 2: Cada nivel dura 5 fondos.
Versión 3: Cada nivel dura 3 fondos.
```

Las distancias, están basadas en la cuadrícula de división que utiliza la plataforma de desarrollo *Unity*, siendo cada unidad un cuadrado.

Distancia entre objetos

```
Versión 1
Nivel 1: 9 unidades de separación entre objetos.
Nivel 2: 8 unidades de separación entre objetos.
Nivel 3: 8 unidades de separación entre objetos.

Versión 2
```

```
Nivel 1: 9 unidades de separación entre objetos.
Nivel 2: 8 unidades de separación entre objetos.
Nivel 3: 7 unidades de separación entre objetos.

Versión 3
Nivel 1: 10 unidades de separación entre objetos.
Nivel 2: 8  unidades de separación entre objetos.
Nivel 3: 9  unidades de separación entre objetos.
```

El juego fue probado por once distintos usuarios, la mayoría con experiencia jugando videojuegos. Seguido de probar cada versión del juego, respondieron el cuestionario. El cuestionario cuenta con cinco preguntas, en el inicio con una introducción que explica los algoritmos utilizados en el desarrollo del videojuego y también la finalidad de la realización de esta encuesta (*Figura 4.7*, *Figura 4.8* y *Figura 4.9*).

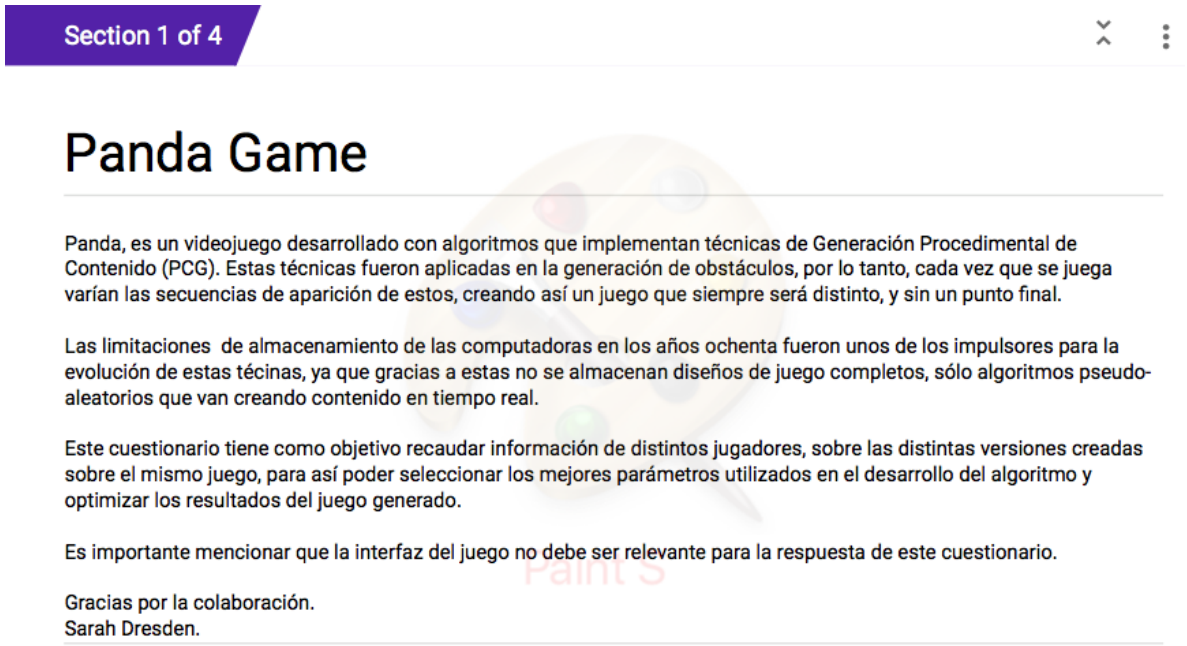

Figura 4.7: Introducción del cuestionario, enviado a los distintos usuarios en conjunto con tres versiones del juego

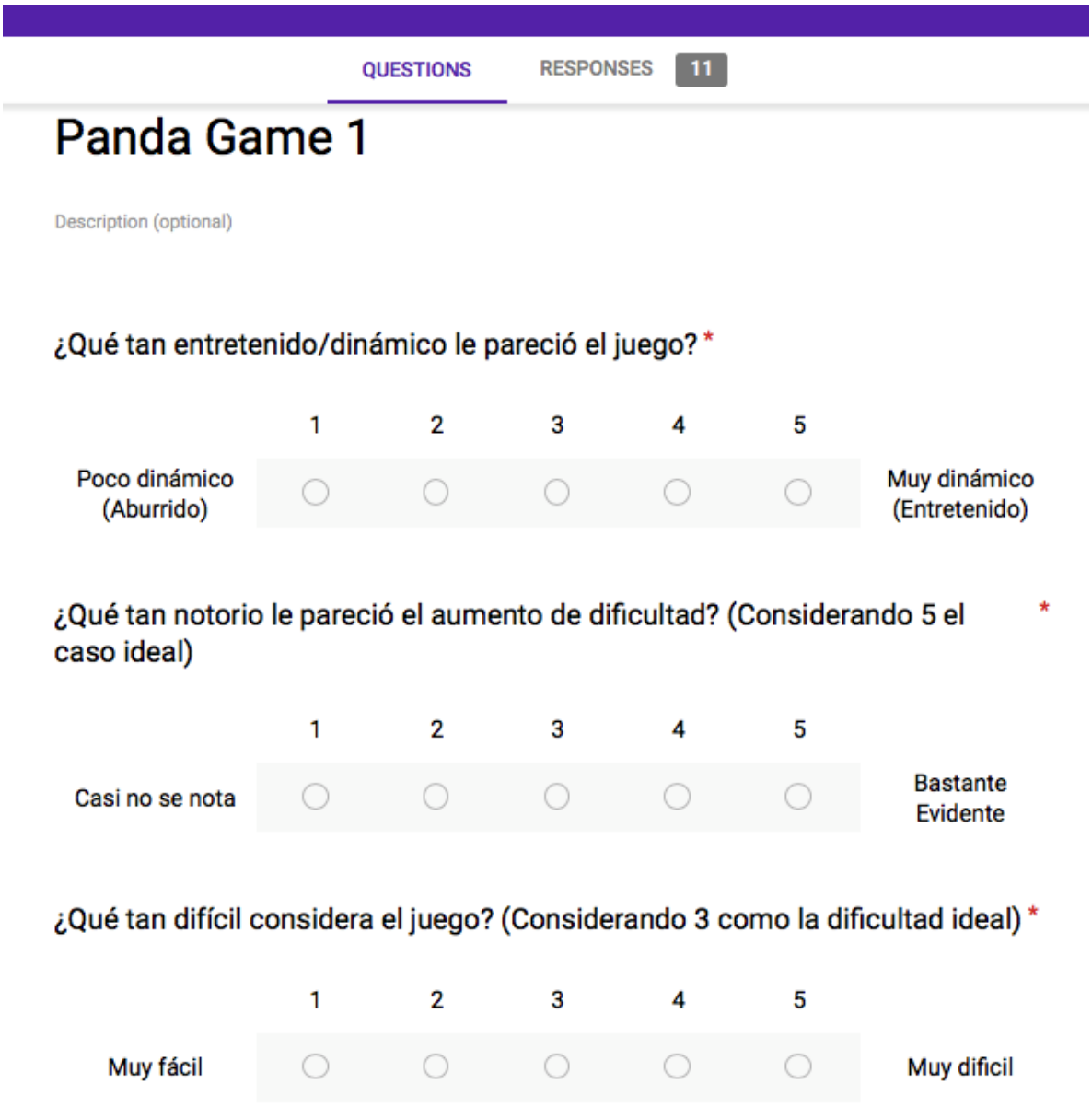

Figura 4.8: Primeras 3 preguntas del cuestionario, enviado a los distintos usuarios en conjunto con tres versiones del juego

Figura 4.9: Últimas dos preguntas del cuestionario, enviado a los distintos usuarios en conjunto con tres versiones del juego

A continuación explicaremos los resultados obtenidos en cada pregunta de la encuesta. A partir de estos se logran obtener los valores óptimos para los parámetros del algoritmo, a partir de las distintas versiones desarrolladas. Estos valores se calificarán como óptimos, basados en el cumplimiento de nuestros objetivos.

En la primera pregunta del cuestionario, sobre lo entretenido del juego, se obtuvieron resultados positivos para las tres versiones. Se observa que ningún usuario calificó por debajo de la media el juego, por lo contrario si se suman las personas que asignaron 4 o 5 puntos a este renglón se puede ver que a más de la mitad de las personas les pareció que estaba por encima de la media. Sin embargo destaca la versión tres del videojuego con un 54.5 % de usuarios que asignaron 4 de puntuación, y en conjunto con los usuarios que les pareció idealmente entretenido, se tiene un 81.8 % de votantes. En las *Figuras 4.10*, 4.11 y 4.12 se ven los resultados.

En la segunda pregunta del cuestionario se busca evaluar cuál versión de video-

juego tuvo una mejor combinación de probabilidades de aparición de obstáculos, distancia entre ellos (frecuencia de aparición) y gramática por nivel, que lograra darle al videojuego un aumento de dificultad ideal para ser jugado. Se puede notar que en las tres versiones, la percepción de aumento de dificultad es bastante variada. En la primera versión, se obtuvo un alto porcentaje de evaluación media baja, por 54,6 % de los usuarios. Sin embargo en la tercera versión un 45,5 % de los usuarios le asignaron 4 puntos, y en conjunto con la respuesta ideal, 5 puntos, llega a un 72,8 % de los usuarios que consideran el aumento de dificultad ideal o casi ideal para el videojuego. En las mismas *Figuras 4.10*, 4.11 y 4.12 se ven los resultados.

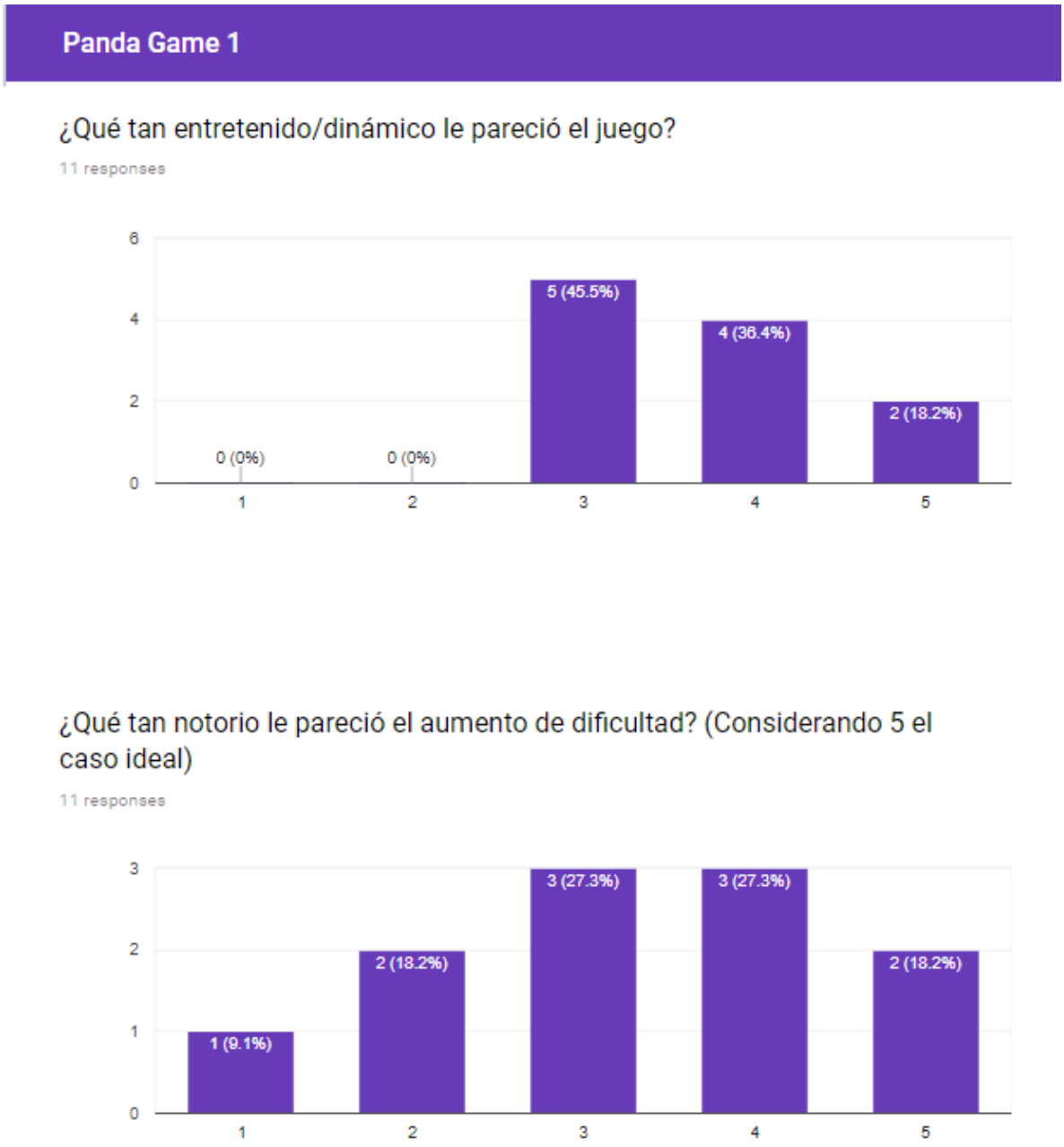

Figura 4.10: Resultados de la primera y segunda pregunta del cuestionario, sobre la primera versión del videojuego.

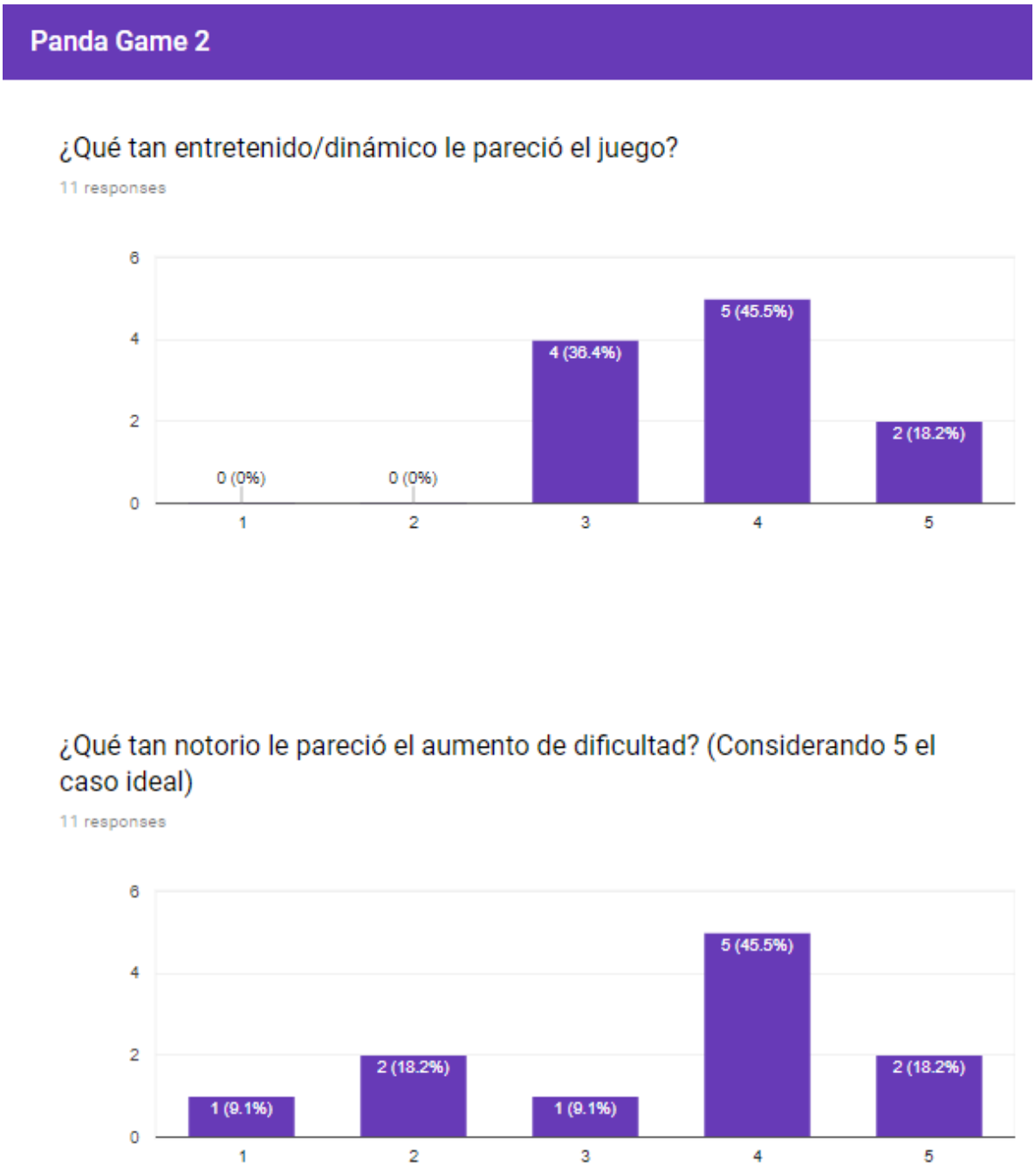

Figura 4.11: Resultados de la primera y segunda pregunta del cuestionario, sobre la segunda versión del videojuego.

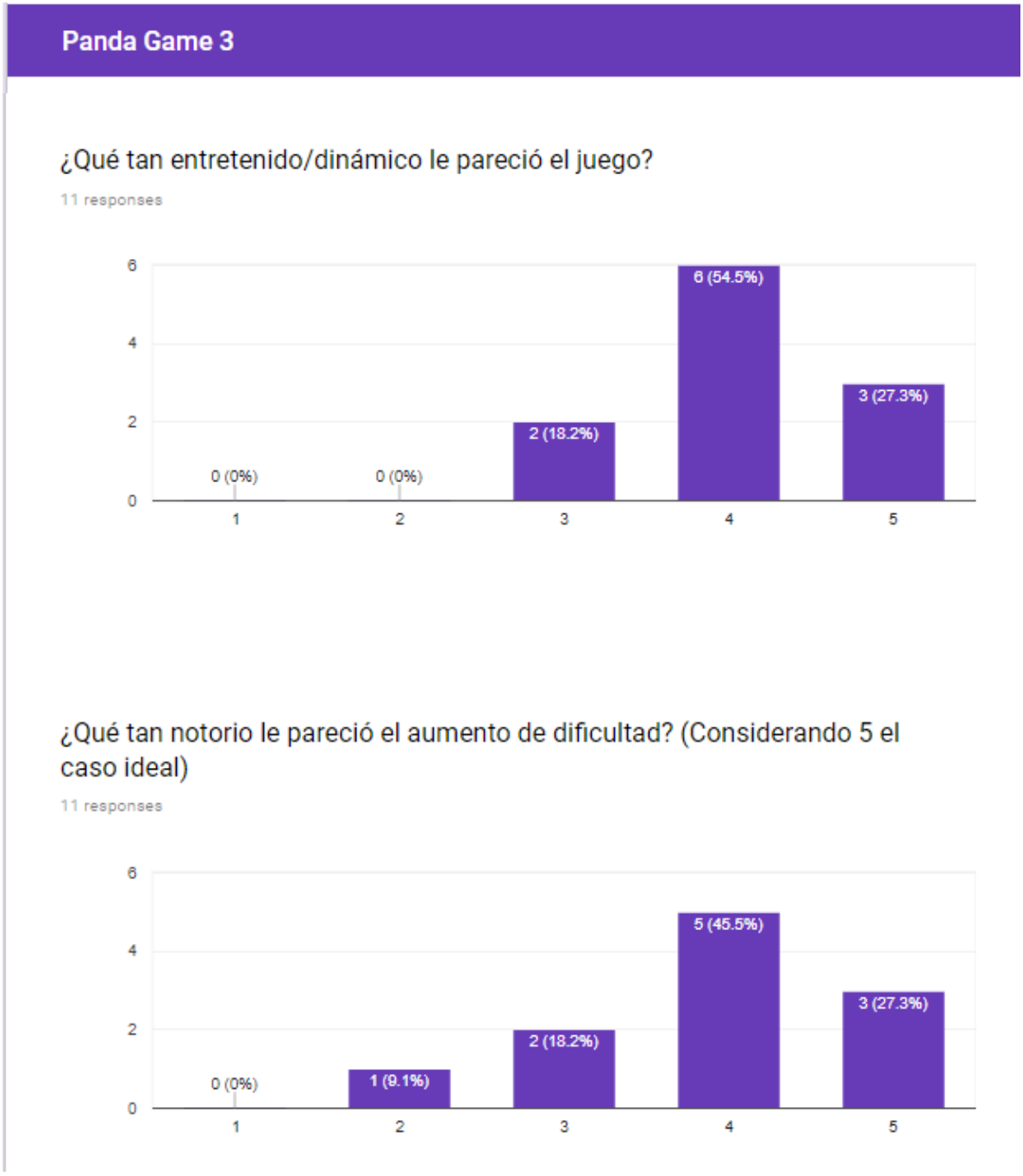

Figura 4.12: Resultados de la primera y segunda pregunta del cuestionario, sobre la tercera versión del videojuego.

Los siguientes tres resultados se discutirán a continuación se pueden observar en las *Figuras 4.13*, 4.14 y 4.15.

La siguiente pregunta consiste en obtener datos sobre si se logró en alguna versión una dificultad ideal para ser jugado, variando los distintos parámetros. En ninguna de las tres versiones se obtuvieron casos extremos, es decir, ningún usuario evaluó los videojuegos con puntuación 1 (muy fácil) ni puntuación 5 (muy difícil). Por otro lado la versión uno tendió a ser más fácil, y la versión tres tendió a ser más difícil, sin embargo esta última tuvo el mayor porcentaje entre versiones de dificultad ideal, con un 38,4 % de los votos.

Próxima pregunta del cuestionario busca evaluar si el tiempo de duración de los niveles, que se escogió por cada versión de juego, fue adecuado, debería durar más o menos, según la percepción de cada usuario, y de este modo escoger el mejor para la versión definitiva. En la versión uno y dos la mayor parte de los usuarios consideró que los niveles tuvieron un buen tiempo de duración, destacando la segunda versión con un 72.7 %. Por otro lado en la primera versión y tercera versión del juego una parte considerable de usuarios consideró que los niveles debían tener un mayor tiempo de duración.

Por último se evalúa la jugabilidad del juego, esto se refiere a la facilidad de uso que un juego ofrece a los usuarios. En las tres versiones se obtuvieron resultados positivos, más del 70 % de los usuarios votaron por encima de la media, es decir asignaron entre 4 y 5 puntos, siendo 5 el nivel de jugabilidad ideal. De las tres versiones se destaca la tercera por un mayor porcentaje de su puntuación ideal, con un 45.5 % de los votos.

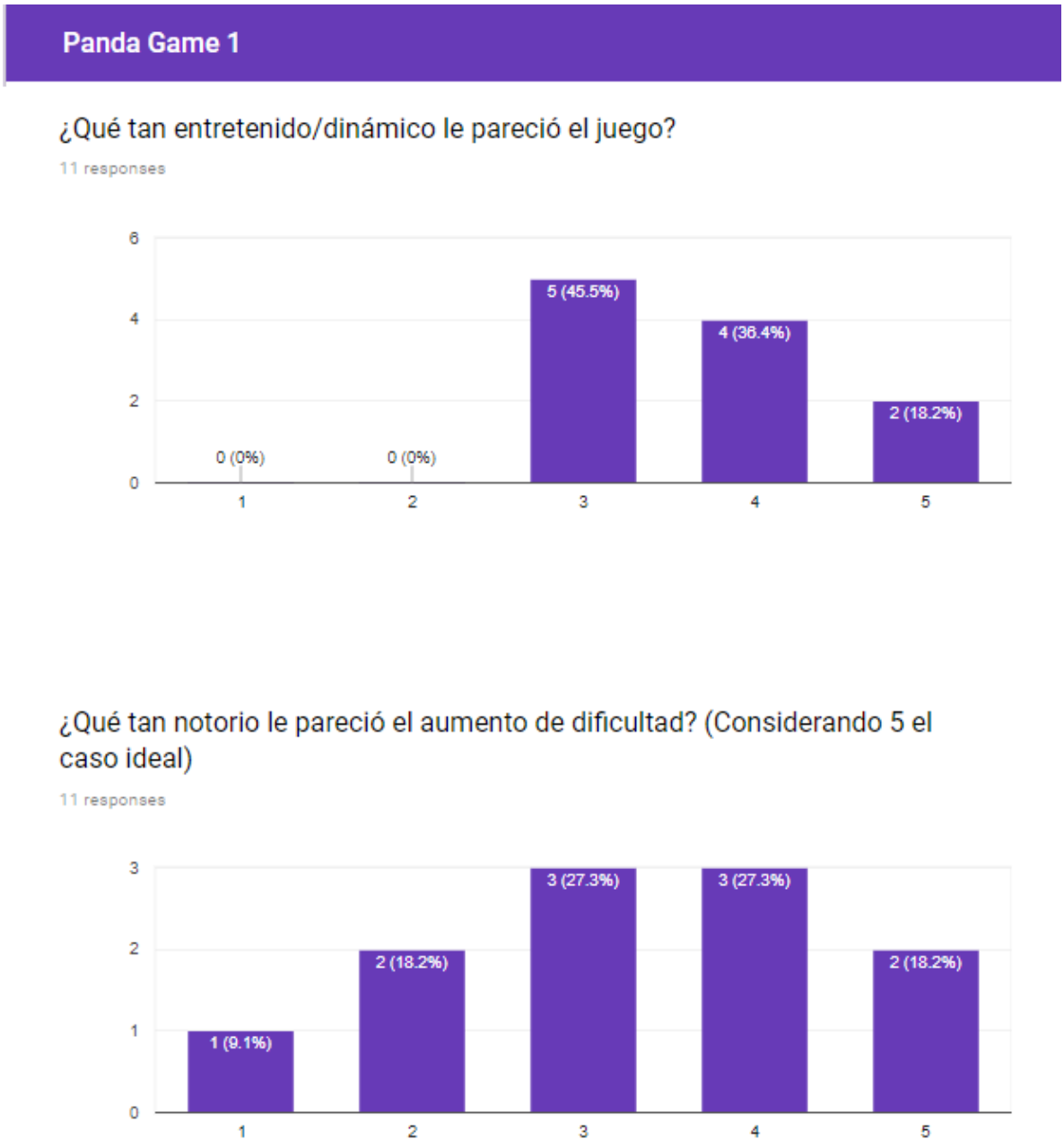

Figura 4.13: Resultados de la segunda, tercera y cuarta pregunta del cuestionario, sobre la primera versión del videojuego.

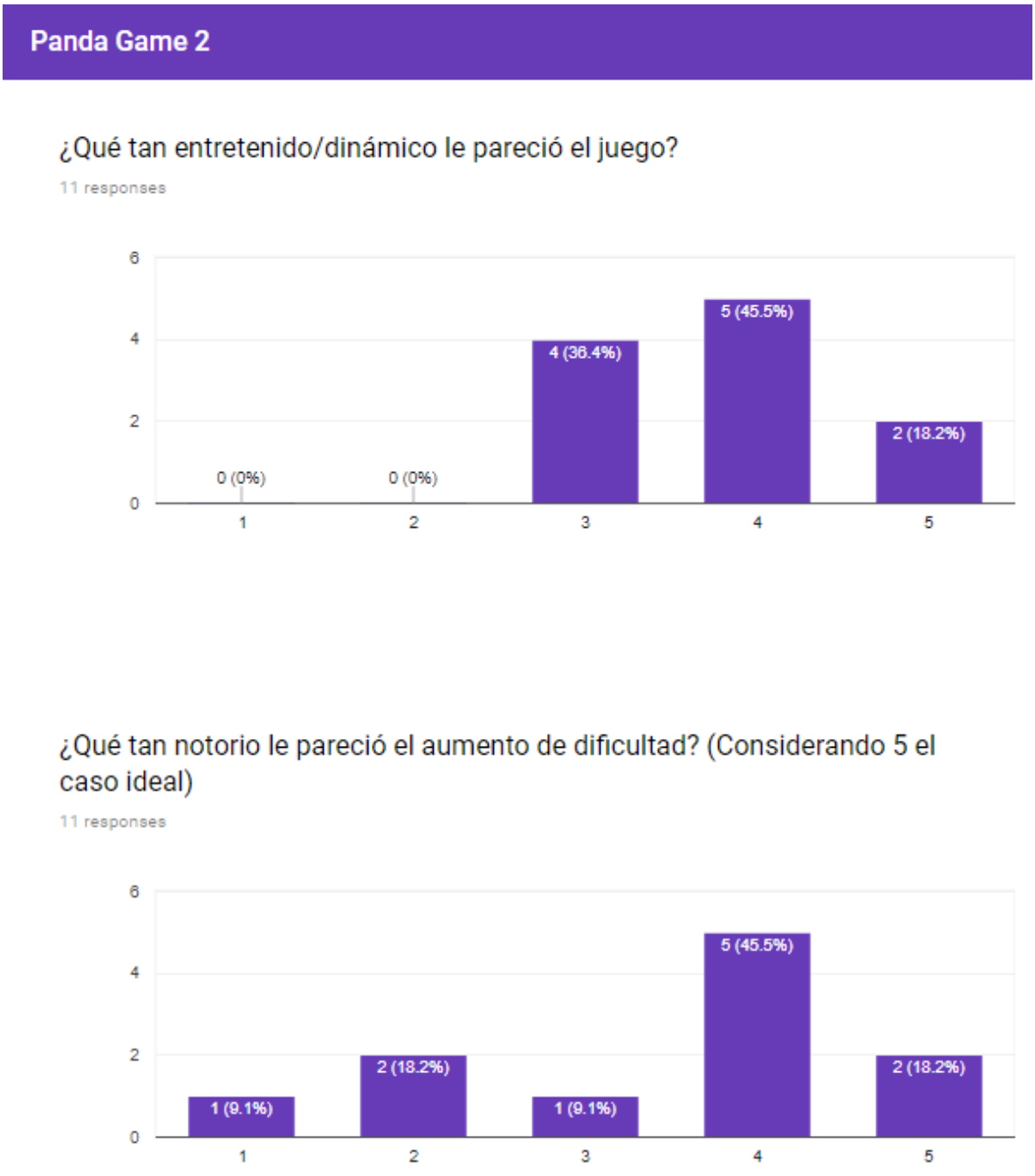

Figura 4.14: Resultados de la segunda, tercera y cuarta pregunta del cuestionario, sobre la segunda versión del videojuego.

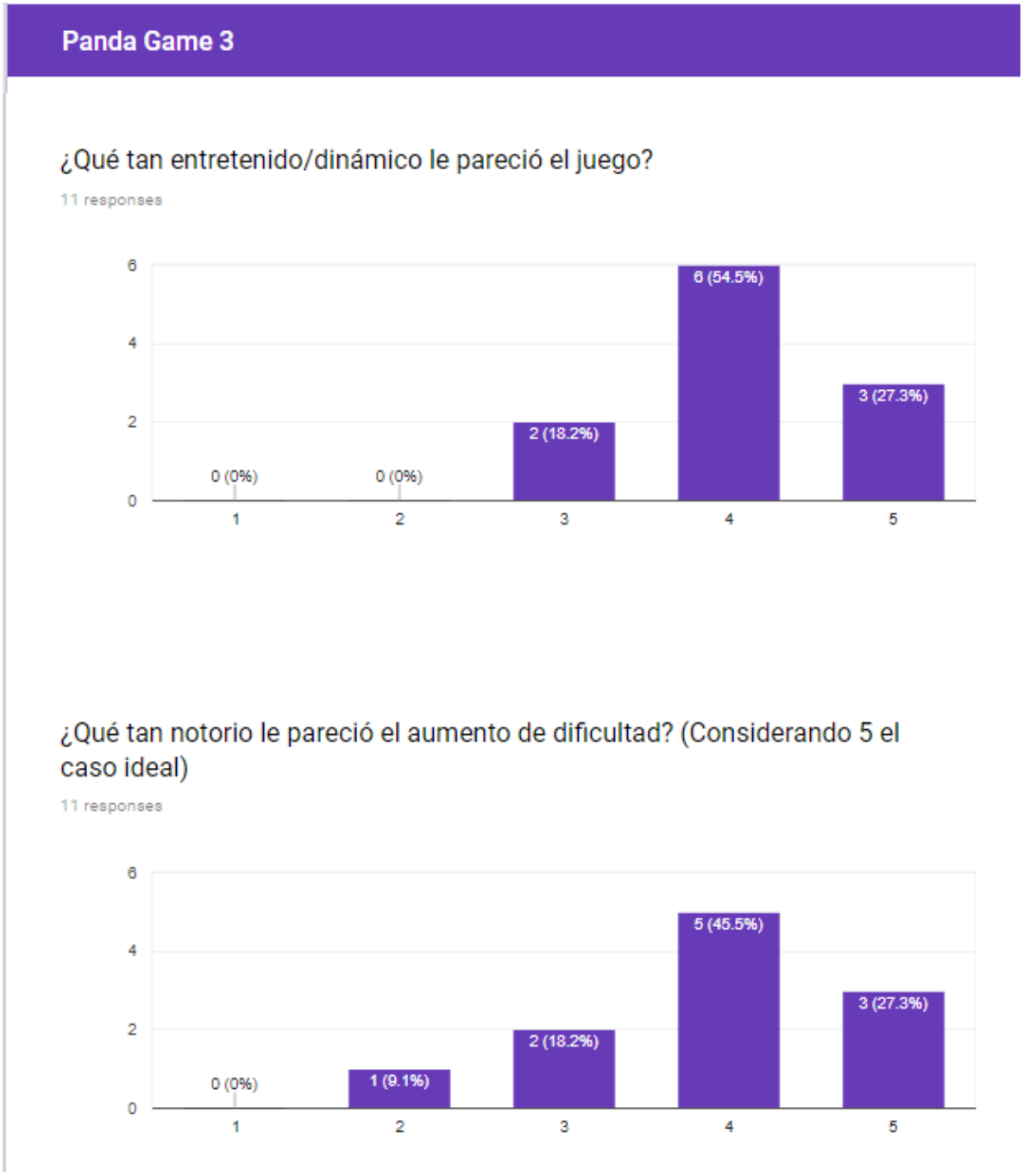

Figura 4.15: Resultados de la segunda, tercera y cuarta pregunta del cuestionario, sobre la tercera versión del videojuego.

Con estos resultados obtenidos en la encuesta, se logra generar una óptima versión del videojuego, utilizando los parámetros que mejor se adaptaron a las necesidades

de los jugadores, por cada versión. A lo largo de la encuesta la versión que obtuvo los mejores resultados fue la tercera, a excepción del tiempo de duración por nivel, que lo obtuvo la segunda. Al combinar estas dos versiones se creó la versión final y óptima del videojuego. Que a pesar de que el algoritmo se mantuvo siempre, se mostró que los parámetros predefinidos, forman parte importante del resultado de un videojuego realmente atractivo, a nivel de entretenimiento y jugabilidad.En la Figuras 4.16 se puede detallar mejor la comparación de resultados.

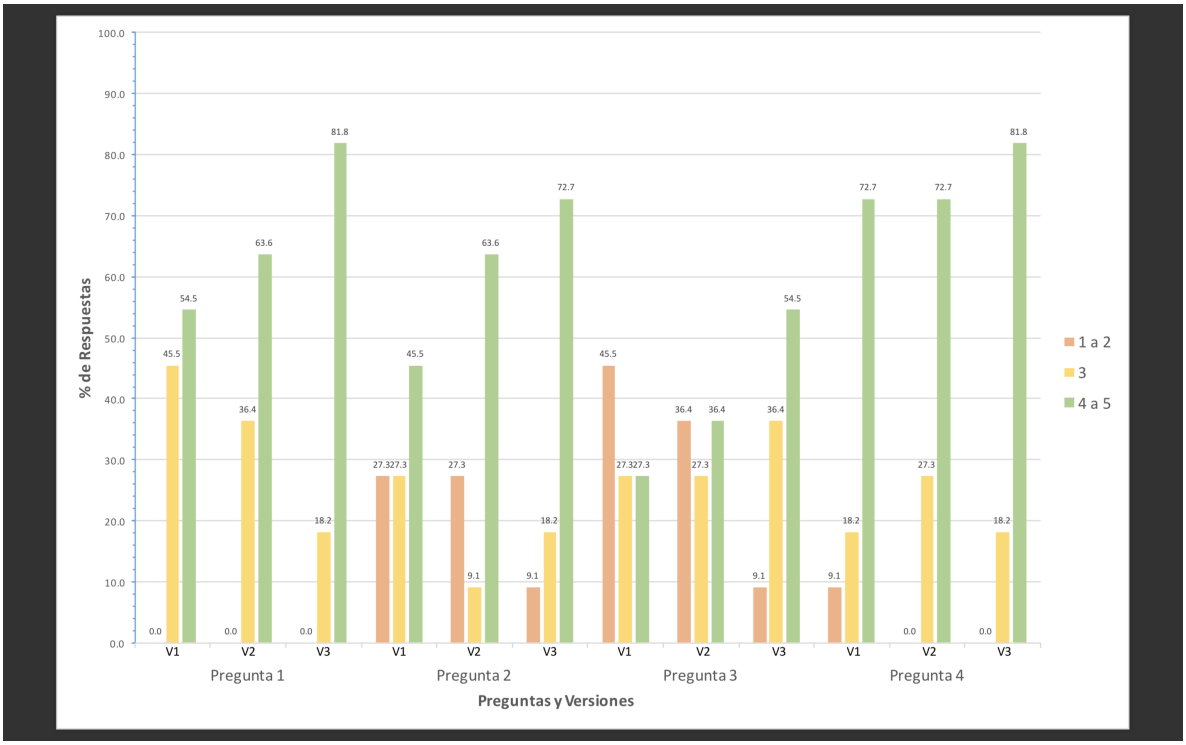

Figura 4.16: Gráfica que muestra todos los resultados unidos, agrupados por número de pregunta separado por versión. Clasificado en grupos de puntuación: 1-2, 3 y 4-5. Las barras crecen segpun el porcentaje de personas que respondieron el valor especificado

5
Conclusiones

El diseño y la implementación del lenguaje basado en gramáticas incidió positivamente en la generación procedimental de niveles de videojuego, ya que este genera secuencias de obstáculos que siempre varían, haciendo así un juego entretenido, siempre diferente.

También se logró el diseño del mecanismo para hacer correspondencia entre el resultado de la gramática y el contenido real del juego, este fue efectivo, pues se obtuvo que los objetos seleccionados estuvieran asociados a la dificultad del nivel, con el uso de probabilidades.

Mediante pruebas de ensayo y error, prácticas personales, y posteriormente un grupo de usuarios que evaluaron el videojuego, se lograron obtener los valores óptimos, dentro de nuestros objetivos, para los parámetros de los algoritmos desarrollados, logrando así instancias entretenidas con una gramática confiable.

Se desarrolló con éxito un caso de estudio utilizando las técnicas implementadas, específicamente un videojuego con niveles generados procedimentalmente. Esto permitió demostrar el correcto funcionamiento de los algoritmos.

Se efectuaron pruebas cuantitativas y cualitativas a partir de tres versiones del caso de estudio desarrollado. La evaluación tuvo éxito, se logró probar la efectividad del método implantado, y se creó la mejor versión del videojuego.

Trabajos futuros

A partir del desarrollo de los algoritmos de generación procedimental de niveles, se recomienda convertir los algoritmos creados en una biblioteca de uso general que le sea de utilidad a cualquier desarrollador, que quiera adaptar a su juego el método de generar obstáculos procedimentalmente con aumento de dificultad por niveles, en un videojuego.

Se recomienda implementar un código que reciba de entrada una gramática, de manera que el algoritmo sea fácilmente adaptable para cualquier proyecto de videojuego del desarrollador.

Se recomienda también crear otros juegos utilizando estos algoritmos ya implementados y así demostrar su aplicabilidad.

Bibliografía

[1] N. Shaker, J. Togelius, y M. J. Nelson, *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*. Springer, 2016.

[2] M. Hendrikx, S. Meijer, J. Van Der Velden, y A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, núm. 1, p. 1, 2013.

[3] MapMage, "The Random Dungeon Generator," http://www.mapmage.com/mapmage.html, 2015, [Online; accessed 20-Agosto-2016].

[4] ANHOLT, "LibPCG," http://www.anholt.net/libpcg/, 2015, [Online; accessed 20-Agosto-2016].

[5] J. Togelius, E. Kastbjerg, D. Schedl, y G. N. Yannakakis, "What is procedural content generation?: Mario on the borderline," p. 3, 2011.

[6] G. Smith, M. Treanor, J. Whitehead, y M. Mateas, "Rhythm-based level generation for 2d platformers," ACM, pp. 175–182, 2009.

[7] J. Dormans, "Level design as model transformation: a strategy for automated content generation," p. 2, 2011.

[8] Y. Jiang, M. Kaidan, C. Y. Chu, T. Harada, y R. Thawonmas, "Procedural generation of angry birds levels using building constructive grammar with chinese-style and/or japanese-style models," *arXiv preprint arXiv:1604.07906*, 2016.

[9] G. N. Yannakakis y J. Togelius, "Experience-driven procedural content generation," *IEEE Transactions on Affective Computing*, vol. 2, núm. 3, pp. 147–161, 2011.

[10] G. Smith, "An analog history of procedural content generation," 2015.

[11] A. Cannizzo y E. Ramírez, "Towards procedural map and character generation for the moba game genre," *Ingeniería y Ciencia*, vol. 11, núm. 22, pp. 95–119, 2015.

[12] M. R. Johnson, "The use of ascii graphics in roguelikes aesthetic nostalgia and semiotic difference," *Games and Culture*, p. 1555412015585884, 2015.

[13] J. Togelius, G. N. Yannakakis, K. O. Stanley, y C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, núm. 3, pp. 172–186, 2011.

[14] N. Shaker, J. Togelius, y M. J. Nelson, "Procedural content generation in games: A textbook and an overview of current research," *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, 2014.

[15] G. Smith, J. Whitehead, y M. Mateas, "Tanagra: A mixed-initiative level design tool," pp. 209–216, 2010.

[16] T. innovation for life, "SKETCHAWORLD: FROM SKETCH TO VIRTUAL WORLD," https://www.tno.nl/en/focus-area/defence-safety-security/missions-operations/sketchaworld-from-sketch-to-virtual-world/, 1932, [Online; accessed 20-Agosto-2016].

[17] K. Compton y M. Mateas, "Procedural level design for platform games," pp. 109–111, Jun 2006.

[18] V. Iyer, J. Bilmes, M. Wright, y D. Wessel, "A novel representation for rhythmic structure," pp. 97–100, 1997.

[19] D. Plans y D. Morelli, "Experience-driven procedural music generation for games," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 4, núm. 3, pp. 192–198, 2012.

[20] P. Mawhorter y M. Mateas, "Procedural level generation using occupancy-regulated extension," IEEE, pp. 351–358, 2010.

[21] M. Jennings-Teats, G. Smith, y N. Wardrip-Fruin, "Polymorph: dynamic difficulty adjustment through level generation," p. 11, 2010.

[22] R. Van der Linden, "Designing procedurally generated levels," 2013.

[23] R. van der Linden, R. Lopes, y R. Bidarra, "Procedural generation of dungeons," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, núm. 1, pp. 78–89, 2014.

[24] M. Traichioiu, S. Bakkes, y D. Roijers, "Grammar-based procedural content generation from designer-provided difficulty curves," 2015.

[25] N. Shaker, G. N. Yannakakis, y J. Togelius, "Towards automatic personalized content generation for platform games." 2010.

[26] J. Dormans, "Adventures in level design: generating missions and spaces for action adventure games," p. 1, 2010.

[27] K. Matsuda y R. Lea, *WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL*, 1ra. ed. Addinson-Wesley, Junio 2013.

[28] K. Hartsook, A. Zook, S. Das, y M. O. Riedl, "Toward supporting stories with procedurally generated game worlds," pp. 297–304, 2011.

[29] M. Kerssemakers, J. Tuxen, J. Togelius, y G. N. Yannakakis, "A procedural procedural level generator generator," pp. 335–341, 2012.

[30] C. McGuinness y D. Ashlock, "Decomposing the level generation problem with tiles," pp. 849–856, 2011.

[31] M. Rauterberg y M. Combetto, "Entertainment computing—icec 2004," pp. 241–247, 2004.

[32] J. Kessing, T. Tutenel, y R. Bidarra, "Designing semantic game worlds," 2012.

[33] J. Roberts y K. Chen, "Learning-based procedural content generation," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, núm. 1, pp. 88–101, 2015.

[34] Y.-S. Lee y S.-B. Cho, "Context-aware petri net for dynamic procedural content generation in role-playing game," *IEEE Computational Intelligence Magazine*, vol. 6, núm. 2, pp. 16–25, 2011.

[35] W. L. Raffe, F. Zambetta, X. Li, y K. O. Stanley, "Integrated approach to personalized procedural map generation using evolutionary algorithms," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 7, núm. 2, pp. 139–155, 2015.

[36] T. ESA, "Essential facts about the computer and video game industry," http://www.theesa.com/article/2016-essential-facts-about-the-computer-and-video-game-industry/, 2016, [Online; accessed 20-Agosto-2016].

[37] Wikipedia, "Historia de los videojuegos," https://es.wikipedia.org/wiki/Historia_de_los_videojuegos/, 2016, [Online; accessed 30-Agosto-2016].

[38] L. Vanguardia, "Diez cosas que te sorprenderán de No Man's Sky, el videojuego del momento," http://www.lavanguardia.com/tecnologia/videojuegos/20160810/403811091804/no-mans-sky-ps4-pc-primeras-impresiones.html/, 2016, [Online; accessed 30-Agosto-2016].