Generación automática de contenido para un nuevo juego basado en el problema de los tres cuerpos

Alejandro Gutiérrez Alcoba

Máster en Ingeniería del Software e Inteligencia Artificial, Universidad de Málaga

Abstract. A procedural content generation algorithm (PCG), able to develop complete maps for a videogame that simulates a physical phenomenon, is proposed in this paper. The evolutionary algorithm developed tries to improve the difficulty of the maps, but it also uses structured populations, aiming to build maps capable of challenging from the most skilled player to the most inexperienced one. Some of the algorithms developed could be used in other games for PCG purposes.

Keywords. Procedural Content Generation, evolutionary computation, genetic algorithms, gravity simulation

Tutor: Antonio J. Fernández Leiva

1 Introducción

Este trabajo fin de máster está centrado en el ámbito de la computación evolutiva y a su aplicación en videojuegos. En concreto, en el uso de técnicas de generación de contenidos por procedimientos con el uso de algoritmos evolutivos para mejorar la calidad de los niveles de un juego, adaptando su dificultad a la habilidad de cada tipo de jugador. Los objetivos que se persiguen con este trabajo fin de máster son diversos y se enumeran a continuación.
Desde el punto de vista del desarrollo de videojuegos:

1. Se presenta un juego sencillo que permite experimentar con algoritmos PCG y de forma que, tan solo redefiniendo la función de aptitud, se puedan generalizar las técnicas para ser reutilizadas en juegos más complejos, para la generación automática de determinados tipos de elementos de sus mapas.

Desde el punto de vista algorítmico:

1. Se ha desarrollado un algoritmo PCG pensado para crear por completo los niveles del juego presentado, haciendo uso de algoritmos evolutivos e introduciendo la idea de utilizar poblaciones estructuradas, persiguiendo encontrar en una misma ejecución niveles de la población que se adapten a la habilidad de jugadores con distinta experiencia. Los algoritmos propuestos sirven para generar contenido de complejidad diversa y que podrían emplearse en otro tipo de videojuegos con la correspondiente adaptación de los mismos. Esto da lugar a un poderoso mecanismo de generación de contenido automático para videojuegos.
2. Además, para el juego en concreto a tratar se han definido dos posibles medidas de dificultad que se usan como funciones de aptitud para guiar el desarrollo de mapas de distinta complejidad. Estas funciones propuestas son conocidas de evaluación *directa*, ya que consideran atributos propios del contenido para ser utilizadas como parámetros de la función.
3. Al margen de estas funciones de evaluación que servirán para la creación de contenido *online* se ha diseñado otra basada en una simulación del juego, más costosa que las anteriores y por tanto no apta para la generación *online*, pero con la que se podrá comprobar si el contenido generado con las dos primeras se ajusta a lo que se pretendía, es decir, generar contenido de diversa dificultad, apto para todo tipo de jugadores.

Todos estos objetivos se irán desarrollando en detalle conforme vayan apareciendo en la memoria. El resto de este documento se presentará como sigue: En la sección 2 se realizará un estudio previo del estado del arte sobre la generación procedimental de contenidos, que servirá de punto de partida para exponer en la sección 3 las novedades que se presentan en este trabajo, así como explicar el funcionamiento del juego, ya que constituye la base sobre la que se trabajará en este proyecto. Se comenzará explicando desde el sencillo punto de vista del jugador, nombrando el conjunto de reglas establecidas y explicando su objetivo para progresar. Más tarde se expondrá su funcionamiento desde una perspectiva algorítmica, aclarando al mismo tiempo la física que subyace en él.

Posteriormente, en la sección 4 se precisarán los detalles del algoritmo evolutivo que se ha desarrollado para generar los niveles del juego: el algoritmo en sí, sus funciones de cruce y mutación, las diferentes funciones de aptitud diseñadas...

Por último, en la sección 5 se analizarán los resultados obtenidos para un conjunto de niveles obtenidos con las técnicas anteriores, extraídos de una población estructurada cuyos individuos son distintos niveles del juego.

2 Antecedentes

En esta sección se motiva el uso de las técnicas de generación de contenido en los videojuegos y se analiza el estado del arte de este campo de la inteligencia computacional. Además se exponen ejemplos académicos en los que se proponen diversas técnicas PCG.

2.1 Motivación del ámbito de estudio

Con el paso del tiempo los videojuegos se han ido convirtiendo en uno de los sectores más importantes y lucrativos de la industria del entretenimiento. Solo en 2011 en los Estados Unidos el sector experimentó unos ingresos de más de 16 billones de dólares [6]. Por otra parte, los costes económicos para producir un videojuego son muy elevados: el desarrollo es un proceso lento y precisa de un amplio equipo de profesionales heterogéneo y muy cualificado. Por tanto, cualquier mejora que consiga optimizar el tiempo de desarrollo o los recursos disponibles es siempre bienvenida.

El área de la inteligencia computacional sirve de aplicación directa al sector del videojuego [11], [12]. La generación de contenido por procedimientos, o PCG en inglés, Procedural Content Generation, consiste en la generación automática, mediante algoritmos en lugar de forma manual, de contenido para videojuegos. Este contenido generado algorítmicamente puede ser muy diverso, pudiendo referirse a los terrenos del juego, sus mapas, niveles completos, características de armas u otros objetos, incluso ciertas reglas del propio juego,... Puede tratarse desde contenido totalmente necesario para avanzar en el desarrollo del videojuego hasta aquel que es puramente decorativo. Eso sí, los algoritmos utilizados deben garantizar la generación de contenido que cumpla ciertos requisitos de calidad, adaptándose al tipo de contenido que se pretendiese crear manualmente. Por ejemplo, en el caso de tratarse de contenido necesario, un buen fin en mente del equipo desarrollador de un videojuego sería adaptar el contenido a distintos estilos de juego.

Son muchas las ventajas de producir contenido de videojuegos mediante algoritmos PCG. En primer lugar puede permitirnos reducir sustancialmente el consumo de memoria del juego, que si bien en la actualidad puede tener un carácter secundario motivó en el pasado el uso de estas técnicas, pudiendo verse en este caso como una forma de comprimir el contenido del juego hasta el momento que sea necesario. Otro motivo importante para el uso de estos algoritmos es, como ya se ha mencionado, el elevado coste que tiene en muchos casos la generación de cierto contenido de los juegos de forma manual.

Además, si como se ha dicho antes se asegura que el contenido generado por PCG cumple con ciertos criterios, como ajustarse a la habilidad de un jugador, el nuevo contenido puede suponerle un reto constante. Si esta generación de contenido adaptado resulta siempre diversa y se genera de forma *online*, es decir, al mismo tiempo que el juego se ejecuta, pueden conseguirse auténticos juegos infinitos, presentando constantemente a cada tipo de jugador nuevos y distintos retos que ha de superar.

2.2 Técnicas usadas en la generación de contenido

Cuando se trata de generar contenido automático para videojuegos pueden hacerse muchas distinciones globales en lo que respecta a los procedimientos a seguir. Siguiendo la taxonomía propuesta en [18], en primer lugar la generación de contenido puede ser tanto *online*, durante la ejecución del juego (con las ventajas que esto puede tener, ya comentadas previamente) como *offline*, durante la fase de desarrollo del juego.

El contenido generado puede ser considerado como contenido *necesario* para poder progresar en el juego, en cuyo caso se ha de asegurar que el contenido es correcto (en el sentido de no proponer objetivos imposibles al jugador), o bien *opcional*, como pueden ser los elementos decorativos y que por lo general pueden ser mucho menos restrictivos.

Un algoritmo PCG puede generar todo el contenido a partir de *semillas aleatorias*. En el otro extremo, el contenido se genera a partir de un *vector de parámetros*. Otra pregunta que cabe hacerse es en qué grado el algoritmo debe ser *determinista* o por el contrario *estocástico*.

Según los objetivos que se pretendan satisfacer, la generación del contenido puede hacerse de forma *constructiva*, asegurando que el contenido es válido en primera instancia, o bien mediante *generación y test*. En este caso el contenido generado pasa por un proceso de validación y en caso de no superar el test, todo o parte del contenido generado es descartado y vuelto a generar.

Un tipo particular de algoritmos de generación de contenido automático en auge actualmente es el basado en la búsqueda en el espacio de posibles soluciones del contenido a generar, *Search-based procedural content generation* (SBPCG). Es un caso particular de generación de contenido en el que se lleva a cabo un proceso de generación y test del contenido, pero no sencillamente aceptando o rechazando el contenido, sino asignándole un valor real (o varios, en el caso multiobjetivo) que mida el grado de bondad del contenido creado.

En este tipo concreto de PCG es muy común el uso de algoritmos evolutivos [3] o genéticos [13], aunque no necesariamente es este el proceso a seguir y otros tipos de técnicas pueden ser utilizadas con el mismo planeamiento SBPCG.

2.3 Algunos ejemplos de algoritmos PCG

Aunque puede decirse que los algoritmos PCG existen desde el nacimiento de la industria de los videojuegos, ha sido en los últimos años cuando ha cobrado más importancia, convirtiéndose en un área de investigación en auge dentro de la inteligencia computacional.

Son muchos los ejemplos de artículos relacionados con esta materia. M. Hendrikx et al. [9] hace un profundo análisis sobre la diversidad del contenido que puede generarse por estos medios. La investigación en este campo es muy activa en la actualidad y pueden encontrarse multitud de papers relacionados, entre los que podemos citar algunos:

Un ejemplo clásico es el de la generación de niveles para juegos de plataformas. Noor Shaker et al., proponen en [17] un sistema para adaptar al estilo de juego de distintos jugadores los parámetros para el diseño de niveles automáticos para un clon de *Super Mario Bros*. Con respecto a este juego, Pedersen et al. [16] investigan la relación entre los parámetros del diseño de niveles con el estilo de juego y la experiencia de juego causada (diversión, frustración,...).

Otro contenido que se puede evolucionar son los mapas. Frade et al. [4], proponen una función de fitness para generar terrenos accesibles. En [5] evolucionan mapas que han sido usados en el videojuego *chapas*. Por su parte, Julian Togelius et al., diseñaron un algoritmo evolutivo multiojbetivo SBPCG para generar mapas para juegos de estrategia [21] y [20]. El mismo autor y otros muestran en [19] un algoritmo que sirve para generar circuitos de un juego de carreras a partir de un vector de parámetros.

En [14], por Hom y Marks, se generan reglas de un juego de mesa para dos personas, persiguiendo además la propiedad de equilibrio entre ambos jugadores. Un ejemplo de esto último puede verse también en [10], por Raúl Lara et al., para el juego de estrategia *Planet Wars*.

También existen numerosos ejemplos para contenido *opcional* del juego, como pueden ser las armas que el jugador puede llevar consigo. Hastings et al. [7], [8] diseñaron un algoritmo SBPCG para el juego *Galactic Arms Race*. El fitness de las armas creadas se calcula teniendo en cuenta el tiempo que los jugadores la

usaban, pudiendo así medir la satisfacción del jugador con el contenido de manera natural y sin requerir la atención explícita de los jugadores en la evaluación.

3 Descripción del problema

Uno de los aspectos más importantes de cualquier videojuego es el diseño de sus niveles, es decir, la forma de sus terrenos y la disposición de los elementos fundamentales para su desarrollo sobre este, como los diversos objetos que pueda recoger el jugador, el lugar donde aparecen ciertos personajes o enemigos,...

En esta memoria se diseñará un algoritmo genético para generar niveles completos del juego presentado, esto es, de forma que se asignen valores a todos los atributos de los objetos que forman parte del juego.

Uno de los principales objetivos en mente cuando se pretende generar contenido con técnicas SBPCG (obviando la motivación inherente de no necesitar la labor humana para producirlo) consiste en tratar de maximizar la diversión que produce en los jugadores. No obstante, con el conocimiento actual sobre este aspecto, las medidas que se suelen proponer suelen ser subjetivas o se dejan caer en presunciones no contrastadas.

En efecto, el algoritmo SBPCG propuesto en esta memoria tratará también de maximizar la diversión y el nivel de satisfacción del jugador, pero para evitar el conflicto mencionado se procurará alcanzar este objetivo tratando directamente la dificultad del juego, adaptándola al nivel de cada jugador. Para ello se propondrán funciones de aptitud destinadas a maximizar la dificultad de los niveles, de forma que estos resulten interesantes a los jugadores más experimentados o hábiles. Al mismo tiempo se busca también el satisfacer a jugadores con menor habilidad en el juego. Por este motivo se propone la idea de usar poblaciones estructuradas en el algoritmo evolutivo. Además, de esta manera se consiguen en la misma ejecución del algoritmo niveles adaptados a jugadores con distinta habilidad.

Podemos clasificar este proyecto dentro de la taxonomía propuesta por Togelius [18] como el de un algoritmo SBPCG para generar los niveles completos (contenido *necesario*) del juego presentado de forma *offline* con funciones de evaluación *directas* (considerando atributos propios del contenido como parámetros de la función), a partir de semillas aleatorias y siguiendo un procedimiento de *generación y test*.

Por las características que definen un nivel del juego (sencillamente la distribución en el mapa de una serie de objetos y sus parámetros) el algoritmo diseñado para la generación de estos podría servir en otros juegos, para definir disposición de algún o algunos tipos de objetos con el que el jugador puede interactuar de alguna manera. En ese caso estaríamos generando contenido tanto *necesario* como *opcional* de ese juego, dependiendo del caso. Además, como también veremos, la definición de un nivel cualquiera del juego aquí presentado podrá interpretarse también de una forma muy distinta: como el diseño de un terreno tridimensional.

Antes de proseguir con los detalles del algoritmo PCG diseñado es necesario explicar el funcionamiento del juego. Si bien este debe considerarse como una herramienta para el resto del contenido de esta memoria (y trabajos futuros), es necesario introducirlo con detalle, sobre todo para comprender como se definen las funciones de aptitud adaptadas al juego.

3.1 El juego

Descripción del juego Dentro del campo de la física y las matemáticas existe un problema clásico llamado *problema de los tres cuerpos*, que puede generalizarse a n cuerpos: este último consiste en determinar, en cualquier instante, las posiciones y velocidades de n partículas regidas por la ley de gravitación universal de Newton, partiendo de cualesquiera condiciones iniciales de posición y velocidad. Dicho problema no tiene solución analítica (para $n \geq 3$) y en general solo puede resolverse con métodos de integración numérica [1]. El juego se inspira en una simulación de este conocido problema, mas con ciertas particularidades que permiten convertir una simulación interactiva de este problema en un entorno jugable.

En primer lugar, la simulación se hace en dos dimensiones, ya que hacerlo en un espacio tridimensional solo lo complicaría de forma innecesaria. Con dos dimensiones, la visualización del juego se hace mucho más sencilla para el jugador: este puede visualizar por completo el área del juego en todo momento de forma clara, con una vista cenital del plano en el que están contenidas las partículas. De esta forma, las trayectorias

que siguen las partículas resultan para el jugador mucho más comprensibles y naturales que si se hicieran en un entorno 3D, teniendo que proyectar una dimensión sobre la pantalla donde se visualiza el juego, en la que es mucho más difícil de observar cambios que las otras dos, naturales de por sí en la pantalla del dispositivo.

En cualquier caso tratar el problema de esta manera no supone perdida de generalidad en muchos juegos 3D, sobre todo aquellos en los que el terreno del juego es una superficie que se puede expresar como el grafo de una única función $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$, en los que aunque, en efecto, el mapa tenga un aspecto tridimensional, la posición de sus elementos se puede referenciar con el uso de dos coordenadas y su topología coincide con la del plano $\mathbb{R} \times \mathbb{R}$. De hecho, y como se verá más adelante, la disposición de los elementos de este juego define de forma natural un terreno con estas características.

Lo segundo, y más importante, es que en la simulación tan solo una de todas las partículas recibirá la fuerza gravitatoria del resto. Esta partícula será la única con la que el jugador podrá interactuar, pudiendo cambiar su vector de velocidad en determinados momentos, para guiarla por la pantalla. El resto de partículas permanecerán siempre en reposo en la misma posición. Otra vez en este caso, el motivo por el que se limita el movimiento a solo una de las partículas se hace por simplificar la información que el jugador ha de procesar, evitando comportamientos extremadamente caóticos y poco predecibles. Las reglas establecidas hasta ahora son las que nos permitirán también definir un terreno tridimensional por cada nivel.

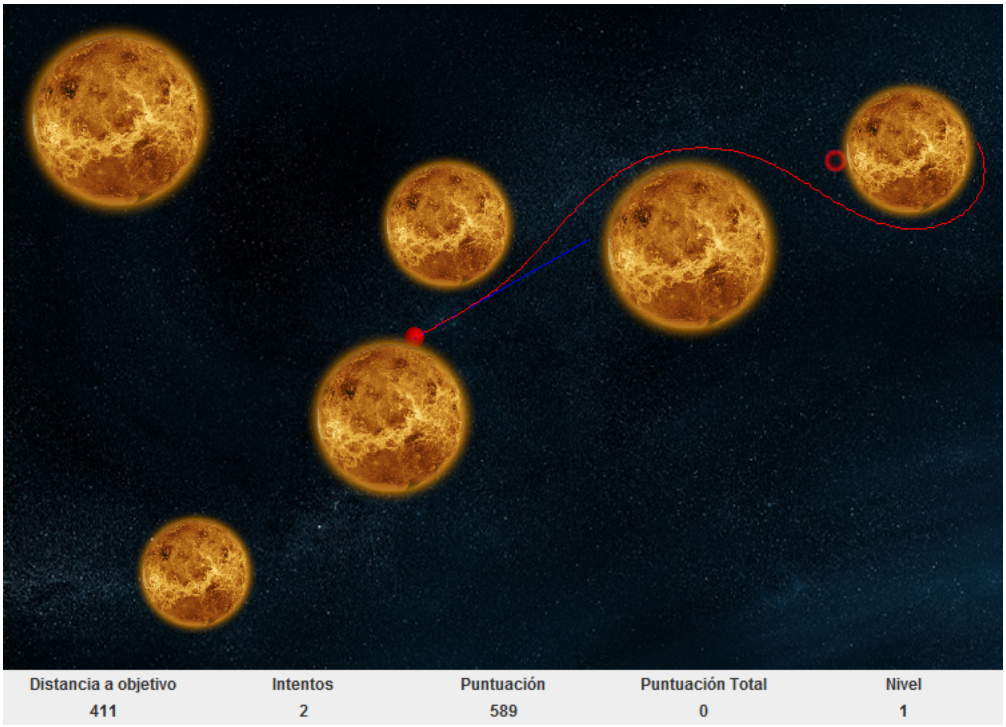

Fig. 1. Ejemplo de un nivel

En la figura 1 se muestra un ejemplo del juego en acción. Como puede observarse, sobre la pantalla aparecen situadas en distintas posiciones n partículas, cada una de ellas con una masa asociada, que se representarán en pantalla como planetas con un radio determinado por la masa de cada una. Estas partículas son las que permanecerán inmóviles durante el juego. Por simplicidad, de aquí en adelante en la memoria, nos referiremos a ellas sencillamente como *planetas*.

Al comenzar el juego, sobre uno de los planetas del nivel, se encontrará otra partícula, de dimensiones mucho más reducidas que el resto, en estado de reposo(la bola roja de la figura 1). Esta partícula es aquella

con la que el usuario podrá interactuar y que recibe la fuerza gravitatoria de todos los planetas del nivel. El jugador puede establecer un vector de velocidad sencillamente haciendo click dentro de la partícula y arrastrando el ratón. El jugador debe procurar (en la mayoría de los casos) establecer un vector de velocidad cuyo módulo sea suficiente para escapar de la atracción gravitatoria del planeta en el que se encuentre y orbitar por el nivel, regido por la ley de gravitación universal de Newton con las salvedades expuestas anteriormente.

Además, cada vez que la partícula colisione con alguno de los demás planetas perderá parte de su energía (cinética), por lo que finalmente acabará siendo atrapada por la fuerza gravitatoria de alguno de los planetas.

Estas consideraciones iniciales definen el comportamiento básico del juego, sobre el que se pueden proponer multitud de variantes, nuevos objetos que actuen sobre el juego y diferentes objetivos para el jugador que le den más sentido al juego que el de una simple simulación.

En concreto, en este trabajo, se propone un objetivo simple para el jugador: hacer que la partícula quede en estado de reposo lo más cerca posible a una determinada posición del plano, marcada con un círculo (como puede verse en la figura 1), sobre otro de los planetas del nivel distinto a aquel donde se posicionaba inicialmente la partícula, realizando lanzamientos como se ha descrito anteriormente.

Para conseguir alcanzar el objetivo, el jugador dispondrá además de una pequeña ayuda: cada vez que este defina un nuevo vector inicial de velocidad haciendo *drag* con el ratón, vector que estará representado por una linea azul en la pantalla, se dibujará también una curva en el color de la partícula que representaría el primer segundo y medio de la trayectoria que seguiría, a menos que golpee antes a otro planeta, en cuyo caso la visión será recortada.

Por cada nivel el jugador dispone de tres intentos para acercarse lo máximo posible a la posición objetivo. La puntuación que obtenga será mayor cuanto más cerca quede parada la partícula y del número de intentos que haya requerido para dejar en reposo a la partícula sobre el planeta objetivo.

Con el objetivo propuesto y las reglas de puntuación descritas, definir un nivel consistiría en dar un número n de cuerpos con distintas masas y distribuirlos por la pantalla indicando sus vectores de posición, así como indicar los dos planetas y la posición sobre los que se posicionarán la partícula y el objetivo alcanzar.

La física del juego Como ya sabemos, la dinámica del juego se basa en la ley de gravitación universal de Newton, que determina que la fuerza ejercida entre dos cuerpos distintos de masas m_1 y m_2 y separados sus centros de masa por una distancia r, es proporcional al producto de sus masas e inversamente proporcional al cuadrado de la distancia. En concreto,

$$F = G\frac{m_1 m_2}{r}$$

donde G es una constante conocida como *constante de la Gravitación Universal*.

Lo primero que hacemos notar es que si nuestro interés es hacer una simulación tomando las magnitudes que nosotros mismos definamos, dicha constante resulta del todo irrelevante y podemos prescindir de ella sin alterar el correcto comportamiento que debe tener el sistema, como satisfacer las leyes de Kepler.

Lo que nos interesa ahora es conocer no la fuerza, sino la aceleración que produce un cuerpo sobre otro. Es sencillo: la segunda ley de Newton determina que la relación entre la aceleración a que sufre un cuerpo de masa m_1 y la fuerza que se le está aplicando es

$$F = m_1 a$$

Si consideramos como esta fuerza F la causada por la ley de gravitación universal de otro cuerpo de masa m_2 y separado a una distancia r, la aceleración que sufre el primero es precisamente

$$a = G\frac{m_2}{r^2}$$

es decir, no depende de la masa del primer cuerpo.

Este valor obtenido de la aceleración solo nos da una idea de su magnitud, el módulo de su vector, pero como para hacer la simulación debemos dar las coordenadas de posición de las partículas necesitamos conocer

las coordenadas del vector de aceleración. Con un simple cálculo trigonométrico (solo es necesario conocer la definición trigonométrica del seno y coseno de un ángulo) se llega al resultado. Notando p_1 y p_2 a los vectores de posición de los dos cuerpos, la aceleración que sufre el primero como consecuencia del segundo es:

$$a = \frac{m_2}{r^3}(p_2 - p_1)$$

Nótese que en un abuso de la notación se ha prescindido ya de la constante G en la última expresión.

Aunque más adelante se hablará de la clase *Planeta*, conviene ahora citar el método que se encarga de calcular la aceleración que sufre un planeta por la acción de otro en una posición dada:

```
1  public void addAcelBy (Planet p){
2     double cubeDistance= (double) Math.pow(this.distanceTo(p), 3);
3     this.acel[0]+=(p.pos[0]-pos[0])* p.masa / (cubeDistance);
4     this.acel[1]+=(p.pos[1]-pos[1])* p.masa / (cubeDistance);
5  }
```

Como se ve, lo que hace es *sumar* al atributo aceleración del planeta objeto la aceleración causada por el planeta que se pasa como argumento del método. Es la clase *Nivel*, que tiene conocimiento de los objetos tipo *Planeta* que contiene, la que se encarga de aplicar dicho método al planeta al que se le quiera calcular la aceleración, pasando como argumento cada uno de los demás planetas del nivel salvo el propio, obteniendo así el vector de aceleración total en las coordenadas donde se encuentre dicho planeta.

Una forma sencilla de realizar la simulación con el vector de aceleración es discretizar el tiempo y en cada iteración pasar el valor de la aceleración a la partícula correspondiente a la posición que ocupa en ese "instante" y actualizar la velocidad y con ello la posición. Magnitudes como la discretización del tiempo que se haga, la masa media de los planetas de un nivel y las dimensiones de ancho y alto de la porción del plano en la que se juega la partida están estrechamente relacionadas unas con otras y determinan la precisión y la velocidad de la simulación. En concreto, para este juego se toman 200 iteraciones por segundo, las dimensiones del nivel son 900x600 (por supuesto, estas coordenadas son generales y totalmente independientes a las dimensiones en píxeles con las que juegue cada usuario, para asegurar que el comportamiento es exactamente el mismo) y las masas de los planetas se sitúan entre los valores 100 y 900. Con esto se consigue una precisión más que aceptable en la simulación de un juego de estas características.

Un nivel como un terreno tridimensional En [2] se propone un método procedural basado en Programación Genética, que permite la generación automática de terrenos, por ejemplo por medio de combinación de funciones elementales, dando lugar a complejos terrenos.

En este apartado veremos como la disposición de los planetas de un nivel puede interpretarse de otra manera mucho más original a lo que sugiere el juego explícitamente: cualquier nivel del juego genera un terreno tridimensional.

En efecto, en cada punto $\mathbb{R} \times \mathbb{R}$ (excluyendo el punto central de los planetas) tenemos definido un vector bidimensional de aceleración, al que si le calculamos su módulo en cada punto, conseguimos definir una función $f : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$. El grafo de la función f define una superficie que puede usarse como el terreno tridimensional de un juego.

No obstante, si lo que buscamos es una superficie que diseñada en el mundo real simule por el efecto de la gravedad de nuestro propio planeta Tierra (al lanzar una moneda o una bola metálica) la misma situación que el nivel del juego asociado, hay que hacer una precisión en la función. Una primera intuición puede hacer pensar que f como la hemos definido simula correctamente la situación, pero a poco más que se piense se observa que no es tan sencillo. Tal y como lo hemos definido, un nivel con un solo planeta representaría un terreno como la superficie de revolución de una función del tipo $f(x) = \frac{1}{x^2}$. Sin embargo, en un artículo por Keith J. Mirenberg [15], se comprueba que dicha variedad debe ser en realidad un hiperboloide.

Haciendo este cambio en el valor real de f que se asigna a cada par de $\mathbb{R}\times\mathbb{R}$ se obtiene la superficie correcta. En la imagen 2 puede verse un ejemplo de la superficie que genera un nivel cualquiera con 5 planetas. Para

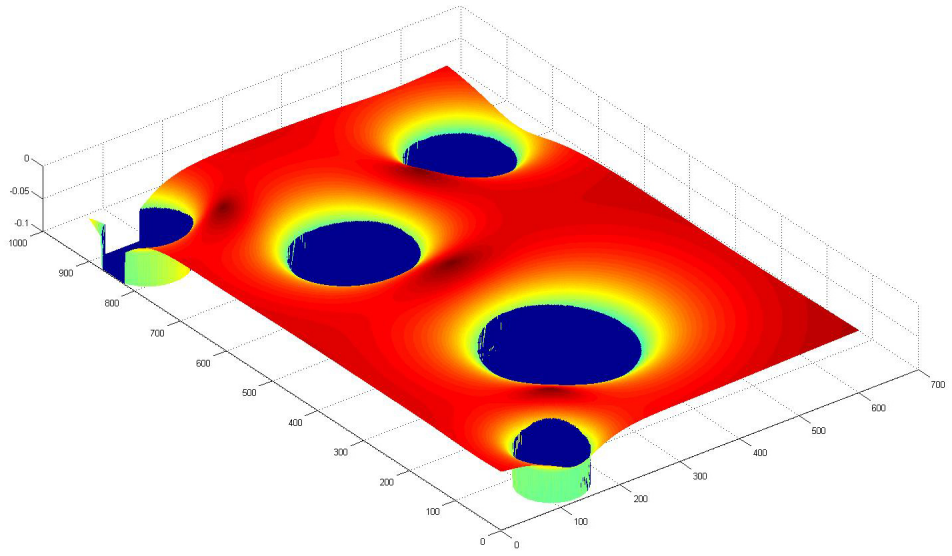

Fig. 2. Mapa generado por 5 planetas

dibujarla se ha hecho un mallado fino de la superficie del juego, calculándose en cada punto de la partición el módulo de la aceleración *corregido* y guardado sus valores en una matriz representada posteriormente en Matlab. Se han truncado los valores del interior de los planetas (incluyendo los puntos centrales, que no están definidos) a valores inferiores para una mejor visualización. Para que el mapa, diseñado en la realidad, simulase las colisiones deberían situarse cilindros del radio de los planetas en su lugar.

Con un par de modificaciones sobre el juego se consigue una nueva clase de terrenos para otros videojuegos que además implementa ya de por sí la física necesaria para que el jugador, vehículo,... se desplace por el mapa con mayor o menor dificultad según la inclinación.

- En primer lugar habría que incluir objetos que al contrario que los planetas repelan, en lugar de atraer. Truncando de alguna forma como se hace con los planetas, en este caso para evitar picos infinitos, se consiguen crear mapas con picos y valles.
- En segundo lugar, sería también muy sencillo incluir una fuerza de rozamiento con el terreno para conseguir juegos en los que el jugador pueda estar en estado de reposo en todos o casi todos los puntos (dependiendo esto de la voluntad del programador, con de la fuerza de rozamiento implementada).

Con estas sencillas implementaciones se completa la base para crear juegos con terrenos definidos de forma muy sencilla: indicando la posición en un plano y un solo atributo indicador de la "fuerza", para una pequeña o grande cantidad de objetos atractores o repulsores. Además, se tendría incluso definido de forma implícita un motor de físicas para el juego en cuestión.

Cabe destacar que el caso más sencillo de este tipo de superficies, es decir, un hiperboloide, es fabricado en plástico por diversas empresas y comercializado bajo nombres como *Hyperbolic Funnes* y se usan en grandes superficies, museos y otros lugares públicos para generar beneficios a modo de "máquinas tragaperras" de divulgación científica, persuadiendo a los viandantes a arrojar monedas para ver como describen órbitas, cada vez más pequeñas por efecto del rozamiento, hasta que inexorablemente acaban cayendo por el agujero.

4 Detalles de la propuesta

En la sección anterior ya se ha descrito el juego que va a ser usado en la parte de generación de contenido de esta memoria. Además, se ha visto como el juego, en apariencia simple, puede servir para definir terrenos para otros juegos de forma sencilla. En esta sección nos centraremos en la parte de generación de contenido y nuevas soluciones propuestas.

4.1 Jerarquía de las clases Java del proyecto

Pasando por alto el proceso que se encarga de recalcular la posición y velocidad de la partícula en cada iteración y el que lleva a cabo el aspecto gráfico del juego, que resultan irrelevantes para el diseño del algoritmo PCG, se darán unas pinceladas de las clases Planeta y Nivel, que sí deberán ser utilizadas de forma importante. Los niveles por ejemplo deberán ser accedidos por el algoritmo al ser los individuos de la población del algoritmo evolutivo, así como en el diseño de los algoritmos como el de mutación y cruce. Los planetas en serán necesarios en otros como las funciones de aptitud diseñadas para el algoritmo genético.

La clase Planeta Posee atributos de tipo entero como la masa y el radio. Como ya se ha comentado previamente, el atributo *radio* viene dado a partir de la masa. En todos los constructores definidos para el objeto *Planeta* se define el radio r de un planeta de masa m como el entero que resulta al convertir el valor $4\sqrt{m}$. El escalar 4 tiene un doble sentido: en primer lugar está pensado para que el radio de los planetas cuadre estéticamente con las dimensiones del plano donde se dibujan los planetas. Por otra parte, el valor ha sido ajustado para que la velocidad de escape de un planeta pueda ser mucho menor.

Otros de los atributos de tipo double contienen la información de los vectores de posición, velocidad y aceleración de los planetas.

El tratamiento de las colisiones es de vital importancia en el juego. Por eso el resto de atributos tienen que ver con esto. Los planetas poseen atributos de tipo entero como el número de veces que el objeto ha colisionado con otro planeta *timesCollided* y el número de veces que ha repetido la colisión con el mismo planeta que el anterior, *timesRecollided*, algo que siempre le sucede a la partícula cada vez que se lanza cuando está cerca de llegar a un estado de equilibrio sobre un planeta. Permiten, entre otras cosas, establecer el límite de visualización del lanzamiento o estimar de algún modo la dificultad de llegar en un lanzamiento a un determinado planeta, dependiendo en parte del número de colisiones que se hayan producido en la trayectoria: cuantas más son, más difícil es predecir para un humano el comportamiento de la trayectoria. De esto volverá a hablarse en la definición de la función de evaluación basada en la simulación.

Por último, otros atributos importantes de la clase en relación a las colisiones son el booleano *collided*, que permite controlar el modo en el que debe actualizarse el vector de velocidad de la partícula, y el atributo de tipo Planeta *collidedPlanet*, que permite referirse de forma sencilla al último planeta con el que colisionó el planeta objeto, algo importante durante la ejecución del juego.

Los métodos más importantes de la clase se encargan de modificar los atributos de posición, velocidad y aceleración del planeta objeto, como el ya descrito *addAcelBy (Planet p)*, y otros como *updateVel()* y *updatePos()*, que actualizan la velocidad y la posición controlando también el cambio de dirección correcto en el caso de una colisión.

La clase Nivel Una instancia de Nivel no es más que una colección de planetas: uno de ellos que es la partícula móvil, otro que es el objetivo a alcanzar y por último un array de ellos que son aquellos que reposan en la pantalla y por tanto el contenido que más caracteriza el nivel y sobre el que se hace mayor hincapié a la hora de generar contenido del nivel.

Además, el nivel posee dos atributos enteros y dos tipo double que determinan el vector posición de la partícula y el objetivo de forma indirecta.

Para aclarar como puede representarse un nivel, expongo a continuación un ejemplo de nivel definido en XML. La clase posee un método que permite guardar el propio nivel actual a un fichero con este formato y uno de los constructores de la clase permite instanciar un nivel a partir de un fichero XML.

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<ns2:nivelXML xmlns:ns2="xs">
    <planetList>
        <planet id="0">
            <posX>727</posX>
            <posY>139</posY>
            <masa>594</masa>
        </planet>
        <planet id="1">
            <posX>267</posX>
            <posY>245</posY>
            <masa>504</masa>
        </planet>
        <planet id="2">
            <posX>522</posX>
            <posY>396</posY>
            <masa>784</masa>
        </planet>
    </planetList>
    <argParticle>4.609315317625434</argParticle>
    <argTarget>6.048532768625851</argTarget>
    <idParticle>5</idParticle>
    <idTarget>3</idTarget>
</ns2:nivelXML>
```

Los elementos del fichero son la lista de planetas que ocupan la pantalla, cada uno definido con sus atributos de posición y masa (recordemos que el radio se calculaba en función a la masa), los índices de los planetas sobre los que reposarán la partícula y el objetivo(*idParticle* e *idTarget*) y el ángulo en radianes que ocupa cada uno (*argParticle* y *argTarget*). Cada uno de estos atributos se considera como uno de los *genes* que configura un nivel. Cada uno de ellos será susceptible a ser cambiado en la fase de mutación del algoritmo genético.

Resulta interesante definir así el nivel debido a su simplicidad, en especial el referirse a las posiciones de la partícula y el objetivo, no usando coordenadas cartesianas sino mediante un índice y el argumento ángulo. De esta forma resulta muy sencillo identificar los planetas donde se encuentran la partícula y el objetivo, de vital importancia a la hora de configurar el algoritmo genético.

Es un método propio de la clase el que se encarga de calcular los valores X e Y de la partícula y el objetivo a partir de los atributos dados. Además, la clase contiene métodos para las distintas funciones de aptitud propuestas que serán estudiadas más adelante en la sección 4.3 y otros para calcular (*evaluate()*) y asignar (*setFitnessValue()*) el valor del atributo *fitnessValue* del nivel a partir de la función de aptitud elegida.

```java
    public double evaluate() {

      double fitness = 0;

        fitness = this.intersecciones();
      //fitness = Math.abs(this.accAroundTargetPlanet(50));

        this.setFitnessValue(fitness);
        return fitness;
    }
```

Y por supuesto, para recuperarlo (*getFitnessValue()*) sin tener que volver a evaluar.

Table 1. Descripción de los parámetros

Nombre	Descripción
POP_SIZE	Tamaño de la población inicial
MAX_ITER	Número máximo de iteraciones que realiza el algoritmo genético
MUTATION_RATE	Probabilidad de mutación (por gen)
MUTATION_PARTICLES_RATE	Probabilidad de mutación de los genes de las partículas
CROSSOVER_RATE	Probabilidad de aplicar el cruce entre dos individuos
N_PLANETAS	Número de planetas que tendrán los niveles de la población

La clase Population Un objeto de la clase Population contiene como principal atributo un ArrayList de tipo Nivel que constituye la población inicial del algoritmo genético. Otros atributos importantes, que se recogen en la tabla 1, son las constantes que defienen el funcionamiento del algoritmo.

Los métodos de esta clase (como la selección de individuos, la mutación, el cruce,...) serán estudiados en la sección 4.3

4.2 Constructor aleatorio de niveles

La clase *Population* posee un método que se utiliza para generar la población inicial de individuos (niveles) para el algoritmo evolutivo que asigna los atributos del nivel de forma aleatoria pero que de por sí constituye ya un primer algoritmo PCG debido a que lo hace cumpliendo con unas primeras condiciones consideradas *buenas* que ha de satisfacer cualquier nivel del juego. De esta forma se generan niveles automáticamente que no solo sirven para arrancar el algoritmo genético con una población inicial, sino que los hace plenamente jugables en primera instancia.

En concreto el método se encarga de controlar los límites inferiores y superiores a la masa que pueden tener los planetas, establecer el mínimo espacio que deben dejar con la frontera de la pantalla y evitar no solo que los planetas no se intersequen, sino que se encuentren todos entre sí respetando una cierta distancia. La parte principal de dicho método se muestra a continuación:

```
1  for (int i=0;i<nPlanetas;i++){
2     planeta[i]= new Planet();
3     do{
4        randomMass(planeta[i],100,900,500,200);
5        randomPos(planeta[i], 900, 600);
6     }while(bienDef(planeta[i],planeta) == false);
7  }
```

Uno a uno, se van instanciando los planetas del array que formarán parte del nivel. El método *randomMass* de la línea 4 se encarga de establecer una masa aleatoria al planeta tratado en ese momento que se pasa como primer parámetro y lo hace a partir de una distribución normal de media 500 y varianza 200, truncándose a 100 si el resultado es inferior y a 900 si es superior.

Posteriormente se asigna una posición aleatoria al planeta, en este caso con una distribución uniforme en ambas coordenadas, pero restringiendo el intervalo de forma que no solo el planeta en cuestión se visualice por completo dentro de la pantalla (dependerá de su radio que previamente se ha dado al dar su masa), sino que además permita visualizar por completo la partícula o el objetivo si en algún momento de la partida se encontrasen en el borde. De no tener esto en cuenta podría llegarse en algunos casos a una situación de bloqueo para el jugador.

Por último se comprueba que el último planeta esté "bien definido" respecto al resto de planetas ya hayan sido creados. Como en el método anterior ya se ha asegurado una posición correcta en la pantalla solo hay que comprobar que dicha posición no lo haga intersecar con alguno de los planetas anteriores. En realidad no solo eso, sino respetar un mínimo de distancia entre los planetas, en concreto, al menos respetar una distancia igual al radio del menor de los dos planetas considerados. De no hacerse algo así, aún cuando

la partícula puede pasar entre dos planetas, si la distancia es muy pequeña la jugabilidad puede verse muy afectada en el nivel.

4.3 Diseño del algoritmo genético

Finalmente se ha optado por diseñar por completo el algoritmo evolutivo usado para la generación de contenido, en lugar de hacer uso de alguna de las plataformas existentes para el desarrollo de este tipo de técnicas de optimización.

Se ha programado en Java, para hacer uso directamente de las clases *Planeta* y *Nivel* que se usan en el juego. Es un algoritmo evolutivo de estado estacionario y población estructurada en tres clases de igual tamaño. El primero contendrá aquellos individuos mejor adaptados a la función objetivo utilizada. Como ambas están encaminadas a mejorar la dificultad en esta primera se encontrarían los individuos más difíciles, en la segunda estarían los medios y en la última los más sencillos. De esta forma con una sola ejecución del algoritmo se consigue un amplio espectro de mapas del juego adaptados a todos los niveles.

A continuación se muestra el código del método *main* del proyecto, que es donde se ha insertado el algoritmo

```
1
2  Population pop = new Population();
3  Collections.sort(populationList);
4
5  for (int iter = 0; iter < MAX_ITER; iter++)
6        {
7
8        nivelSel[0]=new Nivel(pop.binaryTournament());
9        do{
10          nivelSel[1]=new Nivel(pop.binaryTournament());
11       }while(nivelSel[1].esIgual(nivelSel[0]));
12
13       int intentos = 0;
14
15         if ( rand.nextDouble() < CROSSOVER_RATE ) {
16
17          do{
18             nivelSel=crossOver(nivelSel[0],nivelSel[1]);
19             intentos++;
20          }while(crossOverFail && (intentos < 100));
21
22         }
23
24         mutar(nivelSel[0]);
25         mutar(nivelSel[1]);
26
27         if(nivelSel[0].inNiveles(populationList)==false){
28          insertarIndividuoEstructurada(nivelSel[0]);
29         }
30         if(nivelSel[1].inNiveles(populationList)==false){
31          insertarIndividuoEstructurada(nivelSel[1]);
32         }
33
34     pop.evaluate();
35 }
36
37 for (int i = 0; i < POP_SIZE; i++) {
38    populationList.get(i).save("nivel"+i+".xml");
```

```
39 }
```

Vemos como en primer lugar se instancia un objeto *Population* que contiene a la población inicial que se ha generado con el método descrito en la sección 4.2.

```
1   public Population(){
2      populationList = new ArrayList<Nivel>();
3      for (int i = 0; i < POP_SIZE; i++) {
4         populationList.add(NivelAleatorio(N_PLANETAS));
5         populationList.get(i).evaluate();
6      }
7      this.evaluate();
8    }
```

A la vez que se van guardando los individuos se evalúan por primera vez llamando al método propio de la clase Nivel, que a su vez calcula el fitness llamando a la función objetivo activa.

El método de mismo nombre, *evaluate()*, pero de la clase *Population*, se encarga sencillamente de recuperar el fitness de todos los niveles de la población en cualquier momento y sumarlos.

Tras generar la población inicial, esta se ordena (en el caso expuesto tratamos de maximizar la función objetivo, por lo que esta ordenación es descendiente).

Ahora ya puede comenzar el bucle principal del algoritmo. En primer lugar se encarga de tomar dos individuos (líneas 8-11) asegurándose de que son distintos. Dicha selección se hace mediante un torneo binario entre dos individuos distintos de la población seleccionados al azar. Los dos individuos seleccionados se copian en un nuevo array de niveles.

Posteriormente, con una probabilidad igual a la constante *CROSSOVER_RATE* se realiza el cruce entre los individuos seleccionados (en caso contrario pasan los mismos individuos seleccionados al siguiente paso del bucle).

El cruce está controlado por un booleano que indica si la descendencia que genera cumple con los mismos requisitos exigidos en el generador aleatorio de niveles de la sección 4.2. Como se verá más adelante en la sección 4.5 el cruce es muy rápido. Esto es debido en parte a que se ha preferido hacer sin llevar a cabo un proceso de corrección en caso de no respetarse la distancia entre los planetas de los niveles descendientes. Tratar de hacerlo podría llevar a un proceso en cadena que hiciese tener que cambiar más de un elemento y desvirtuar demasiado su estructura. Por este motivo si el cruce no se produce correctamente en un determinado número de intentos se prefiere no realizarlo en lugar de tener que aplicar la corrección en el nivel. En cualquier caso este suceso se produce solo en torno al 10% de los casos.

El siguiente paso es realizar la mutación en los individuos descendientes (se explicará en la sección 4.4). Entre las líneas 27-31 se intenta insertar los nuevos individuos en la población (recordemos que es estacionaria). Para ello en primer lugar se comprueba si existían previamente en la población: en algunos casos es posible que el cruce no se realice y que en la mutación no se seleccione ningún gen. En caso de existir se descarta y si se trata de un nuevo individuo el método *insertarIndividuoEstructurada(Nivel nivel)* evalúa al nuevo individuo y lo descarta o lo inserta en el primer, segundo o tercer grupo según proceda (está preparado a priori para una función objetivo que maximice).

Por último se llama al método que actualiza el fitness total de la población, antes de realizar la siguiente iteración. El algoritmo calcula por tanto 2MAX_ITER evaluaciones de la función objetivo, más *POP_SIZE* al inicializar la población con los niveles aleatorios.

Realmente el algoritmo está pensado para ejecutarse de forma *online* en el juego, con la idea posterior de generar un juego infinito ofreciendo siempre niveles adaptados a su habilidad, pero también debemos guardar la información de la población generada para poder usarla de forma *offline*, con la intención de poder hacer pruebas con jugadores reales y ofrecer exactamente los mismos niveles a todos ellos, camino que también se ha iniciado en este trabajo.

Por esto, una vez que el proceso acaba se guarda cada individuo de la población en un fichero XML diferente, para poder recuperarlo en cualquier momento con el constructor de niveles a partir de un fichero XML.

En cuanto a los valores de los parámetros que se han usado en el algoritmo, estos vienen recogidos en la tabla 2.

Table 2. Valores

Nombre	Descripción
POP_SIZE	100
MAX_ITER	varía en las distintas pruebas
MUTATION_RATE	$(3\text{N\_PLANETAS})^{-1}$
MUTATION_PARTICLES_RATE	0.15
CROSSOVER_RATE	0.9
N_PLANETAS	6

4.4 Algoritmo de mutación de los individuos

En un nivel con n planetas el número de genes correspondientes a posición X, posición Y o masa, asciende a $3n$. Por tanto, en la fase de mutación se ha establecido que cada uno de esos genes pueda ser modificado con una probabilidad $p = \frac{1}{3n}$. La elección de este valor no es caprichosa: al hacerlo de esta forma el número de genes de este tipo que son mutados en cada nivel sigue una distribución binomial $X \sim B(3n, \frac{1}{3n})$. Así, la cantidad media de genes los genes correspondientes a los planetas que son mutados resulta ser

$$E(X) = 3n\frac{1}{3n} = 1$$

En el caso de que uno de los genes de posición sea escogido la variación que se le hace es sumarle un valor x que siga una distribución normal de media 0 y varianza 50. Así, si el gen escogido es la posición X esta podrá desplazarse tanto a la izquierda como a la derecha una cantidad igual al valor absoluto de x y si es la posición Y este cambio será hacia arriba o hacia abajo.

Con la masa se ha seguido la misma idea de usar una distribución normal, de forma que el planeta en cuestión pueda encoger o agrandar su masa (y radio) una cierta cantidad. Y aunque por supuesto no son comparables las magnitudes de posición y masa, la varianza escogida para la normal ha sido la misma que en el caso anterior, 50.

En cuanto a los cuatro genes correspondientes a la posición en el nivel de la partícula y el objetivo, *idParticle*, *idTarget*, *argParticle* y *argTarget*, no se sigue el mismo razonamiento. De cambiar cada uno con probabilidad $\frac{1}{4}$ se haría que por término medio uno de ellos siempre cambiaría, es decir, se centraría mucho más la atención de la mutación en este tipo de genes, puesto que en un nivel con 5, 6 o 7 niveles el número $3n$ es significativamente mayor. No queriendo dejar tampoco relegados a este tipo de genes finalmente se ha optado por mutarlos cada uno con probabilidad $0,15$.

El método de la mutación, tras haber pasado uno a uno por cada gen del nivel y cambiando aquellos seleccionados, comprueba si en el proceso se ha producido un efecto no deseado en el nivel, como hacer que un planeta sobresalga de los límites establecidos en la pantalla o que dos planetas queden intersecados o muy cerca.

Al igual que se hace en el cruce, si algo así se produce se vuelve a dejar el nivel como se encontraba inicialmente y se repite el proceso hasta que la mutación se realice correctamente.

4.5 Algoritmo de cruce de los individuos

Para construir un nuevo operador de cruce para los niveles nos centraremos en primer lugar en copiar la información de los planetas, dejando la partícula y el objetivo para más tarde.

Pensemos por un momento en la posibilidad de usar como operador de cruce un algoritmo sencillo, como la recombinación en un punto. Teniendo en cuenta la facilidad con la que pueden producirse malformaciones en los niveles y que no se ha establecido ningún orden para los planetas del nivel contenidos en un array la idea se antoja absurda. Incluso si se estableciese un orden, pongamos el lexicográfico, por ejemplo, teniendo solo en cuenta los atributos de posición X e Y, aún se podrían suceder multitud de errores.

El algoritmo de cruce de dos individuos que se propone resulta original, ya que se inspira en la recombinación en un punto pero de forma geométrica. Lo primero que hace es calcular una recta aleatoria pero que se asegure de cortar el área del nivel en dos partes

```
1   do{
2      int x1,y1,x2,y2;
3      x1 = rand.nextInt(900);
4      y1 = rand.nextInt(600);
5      do{//evitemos divisiones por cero
6         x2=rand.nextInt(900);
7      }while (x1==x2);
8      y2 = rand.nextInt(600);
9      m = (float)(y2-y1)/(float)(x2-x1);
10     n = -m*x1+y1;
11     divParent1 = division(m,n, parent1);
12     divParent2 = division(m,n, parent2);
13  }while(divParent1[0].length!=divParent2[0].length || divParent1[0].length==0 ||
        divParent1[0].length==N_PLANETAS);
```

Eligiendo cuatro puntos al azar se calcula la ecuación de la recta y posteriormente el método *division* se encarga de guardar por separado los planetas que han quedado en la parte superior de los que han quedado en la inferior. El punto que determina si un planeta queda en la parte superior o inferior es el centro de dicho planeta, es decir, el valor de su posición. Si en cada una de las partes ha quedado al menos un planeta y coinciden el número de planetas que hay en la parte superior y en la inferior de ambos individuos seleccionados, se toma el corte como válido.

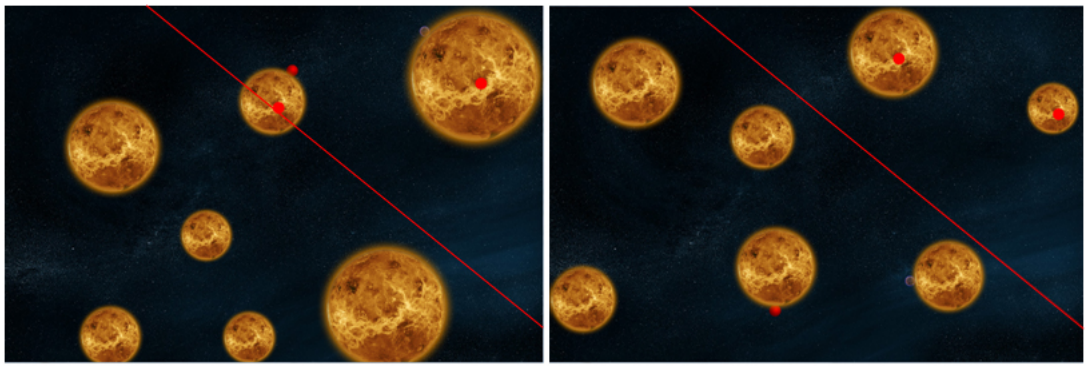

Fig. 3. Ejemplo de dos niveles seleccionados

En principio puede parecer muy restrictivo, pero en realidad, salvando el caso poco probable de que un corte deje una de las partes sin planetas, la probabilidad de que ambas tengan el mismo número es exactamente $\frac{1}{n}$, siendo n el número de planetas del nivel, que no es tan bajo pensando que el número de planetas de un nivel no debería ser superior a 7 u 8 para no hacerlo demasiado impredecible.

El hacer el corte de esta manera generaliza el comportamiento que tendría la clásica recombinación en un punto con la que, si por ejemplo consideramos los planetas ordenados por el orden lexicográfico (uno de los

pocos que tendría sentido para el propósito que nos ocupa) los cortes serían, geométricamente, como líneas prácticamente horizontales o verticales según se considere ordenar primero por la coordenada Y o por la X.

Aún queda por definir como se generan los dos individuos descendientes a partir de los primeros. Es fácil: la parte superior del primer padre con la parte inferior del segundo conforman el primer hijo. Análogamente se genera el segundo.

En las figuras 3 y 4 se muestra un ejemplo de cruce. Se ha dibujado una línea roja con la recta que sirvió para separar los planetas en ese caso. Para diferenciarlos más fácilmente, los planetas de la parte superior aparecen con un círculo rojo pintado en su centro. Como se puede apreciar, en el primer hijo dos planetas casi se intersecan al combinar parte de un padre y la del otro. En un caso como este se repetiría nuevamente el cruce para intentar dar una descendencia correcta. La condición impuesta de la separación mínima entre los planetas es en ocasiones estricta, por lo que si tras un número de intentos el cruce no se produce correctamente se aborta el proceso (como se indicó en 4.3).

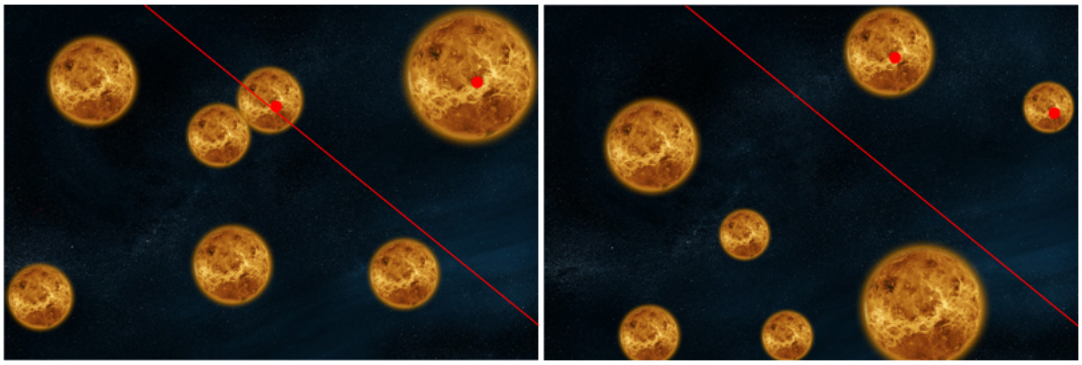

Fig. 4. Ejemplo de la descendencia de la figura 3

Uso del algoritmo de cruce en otros juegos El algoritmo de cruce propuesto puede servir también para ser usado en otros juegos que se desarrollen sobre un terreno, para generar contenido automático de diversos objetos que representen *items* que puedan ser recogidos por los jugadores, obstáculos de distinto tipo o bien elementos decorativos.

Puede parecer una restricción importante el imponer que en ambos padres el nivel se divida en el mismo número de objetos cuando el índice de estos es elevado, y de hecho lo es. Como hemos explicado antes, a mayor índice de objetos a generar menor será la probabilidad de hacer el corte correctamente: con n objetos la probabilidad sería cercana a $\frac{1}{n}$ (cercana porque se está pasando por alto el raro caso de que el corte deje todos los objetos en una sola parte, que tampoco se consideraría correcto) y de este modo el número medio de intentos hasta lograr un corte correcto sería n.

No obstante este problema puede salvarse fácilmente introduciendo una pequeña modificación en el algoritmo: considerar como válido que entre la misma porción del nivel de un padre y de otro pueda haber una diferencia de k objetos ($k << n$ a definir según el problema) y generar en este caso los restantes bien con posiciones al azar o insertando en el padre que tenga menos los que falten del otro.

En este proyecto no ha sido necesario optar por esta idea debido a lo reducido del índice de planetas en un nivel, pero la idea es factible y daría buenos resultados como operador de cruce.

4.6 Funciones de aptitud para el algoritmo genético

Para este trabajo se han propuesto dos funciones de aptitud distintas que representan propiedades del nivel que en un caso al maximizar y en otro al minimizar, hacen más difícil un nivel. Dentro de todo lo expuesto

en esta memoria, estas son las medidas más concretas y particulares al juego tratado y que por tanto son menos extrapolables a otro tipo de juegos.

Método de las intersecciones La primera idea es bien simple: consiste en poner "tierra de por medio" entre la partícula y el planeta en que se encuentra el objetivo.

El método calculará, en primer lugar, la ecuación de la recta que pasa por el punto central de la partícula y del planeta objetivo. Posteriormente comprueba, entre los planetas intermedios entre aquel donde esté la partícula y donde esté el objetivo(así se evita comprobar algunos), si cada uno de ellos interseca con la recta y cual es esa distancia.

Si la recta definida por los puntos mencionados tiene por ecuación $y = mx + n$ y la de la circunferencia del planeta tratado en el momento es $(x-a)^2 + (y-b)^2 = R^2$, los puntos de corte, en caso de existir, son:

$$\begin{cases} x_0 = \frac{\sqrt{R^2m^2+R^2-a^2m^2+2abm-2amn-b^2+2bn-n^2}}{m^2+1} + \frac{a+bm-mn}{m^2+1} \\ y_0 = \frac{a+bm-mn}{m^2+1} \end{cases}$$

Y con el otro signo:

$$\begin{cases} x_1 = -\frac{\sqrt{R^2m^2+R^2-a^2m^2+2abm-2amn-b^2+2bn-n^2}}{m^2+1} + \frac{a+bm-mn}{m^2+1} \\ y_1 = \frac{a+bm-mn}{m^2+1} \end{cases}$$

Si el discriminante es mayor que 0 se realiza el cálculo de los dos pares de puntos y se calcula el módulo del vector definido por los puntos (x_0, y_0) y (x_1, y_1), que dan por resultado la longitud del corte del planeta en cuestión.

El valor de *fitness* de este método será la suma de todas estas intersecciones. Resulta interesante que el uso del algoritmo genético con esta función de aptitud aumenta también de forma indirecta la distancia entre la posición inicial de la partícula y del planeta objetivo.

La evolución típica del fitness de la población total con el algoritmo genético diseñado (solo cambiando *MAX_ITER = 2000*) se presenta en la figura 5.

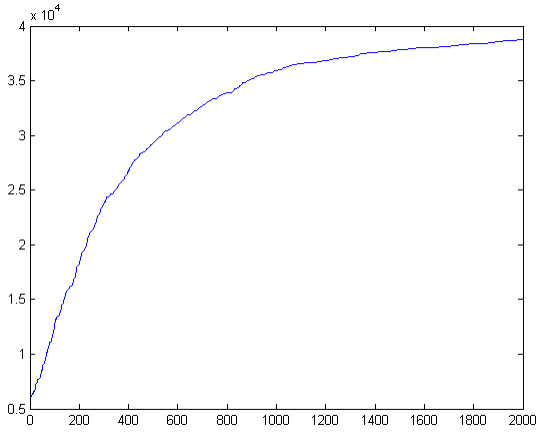

Fig. 5. Evolución del fitnes de la población

Como observación, si sobre el nivel no se impusieran condiciones como respetar una mínima distancia entre los planetas y solo se tuviera en cuenta que no se intersecaran parece claro cual sería la forma típica que tendría un nivel con un fitness máximo global (que además se alcanzaría con mucha más facilidad): los

planetas (no tendrían por qué ser todos) estarían alineados con una de las diagonales de la pantalla, sin un solo hueco entre ellos y la partícula y el objetivo se encontrarían en esquinas de la pantalla opuestas.

Afortunadamente las restricciones impuestas a los niveles impide no solo que algo así suceda sino que permiten mucha más variedad en los niveles generados. Aún así existe cierta tendencia a ver varios planetas en torno a una diagonal , pero podría solucionarse por completo, con un algoritmo multiobjetivo que premiase la poca correlación lineal entre los puntos centrales de los planetas.

Minimizar la atracción en el planeta objetivo Otra forma de añadir dificultad al objetivo de situar la partícula sobre el planeta objetivo es minimizar su fuerza de atracción. Una forma simple de hacerlo sería minimizar la masa del planeta objetivo al mismo tiempo que se aumenta la del resto. Por ejemplo, podría definirse como función objetivo la masa relativa del planeta objetivo, m con respecto a la suma de los demás, M

$$\text{fitness} = \frac{m}{M}$$

No obstante y aunque naturalmente funciona, es evidente que no es necesario utilizar un algoritmo evolutivo para generar niveles automáticamente que satisfagan la minimización de esta medida. Se podría diseñar como algoritmo PCG un método que construyese directamente un nivel con estas características. Y además, esta técnica daría lugar a niveles prácticamente iguales en cuanto a las masas de sus planetas y aún así no se tendrían en cuenta las posiciones de estos para poder controlar la dificultad de los niveles con mayor precisión.

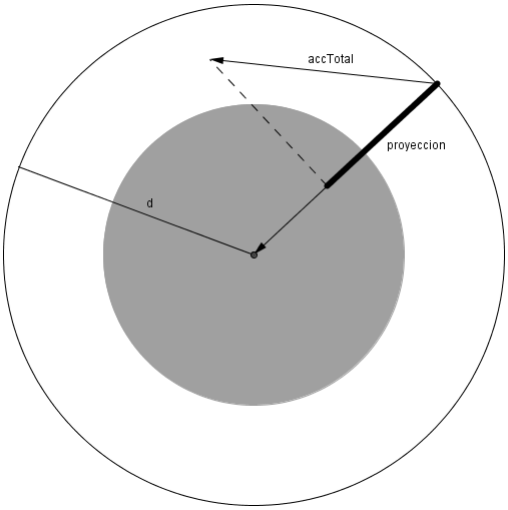

Fig. 6. Proyección de la aceleración

Por ello se ha pensado otra forma de medir la atracción del planeta objetivo con una función que tuviera en cuenta las masas de todos los planetas pero también sus posiciones.

En la figura 6 se muestra un ejemplo del proceso que se realiza en el método. Para hacer la medición se posiciona en primer lugar una partícula auxiliar a cierta distancia d del planeta objetivo (superior al radio de este).

Para ese módulo d se consideran raíces n-ésimas (por defecto se toma $n = 50$) y en cada una de estas posiciones alrededor del planeta a distancia d se calcula el vector de aceleración (etiquetado en la figura 6 como *accTotal*) para finalmente calcular su proyección sobre el vector definido entre el punto considerado en

ese momento y el centro del planeta objetivo. El valor de la proyección indica con su signo si en ese punto la tendencia es de atracción o repulsión y con qué magnitud.

Para el cálculo del fitness se ha tomado como distancia d aquella en la que reposaría la partícula (radio del planeta objetivo más radio de la partícula) y se ha calculado la media de todas las proyecciones para indicar de alguna forma la atracción que produce el planeta objetivo sobre su superficie. Hay que hacer un matiz: si en algún caso la proyección es atractiva y superior a la proyección que causaría el vector de aceleración del planeta objetivo en exclusiva, su módulo, se trunca el valor de dicha proyección a ese valor. Así se evitan casos en los que resulta imposible llegar al planeta objetivo: aquellos en los que el vector *accTotal* es siempre muy potente en casi la misma dirección, haciendo que el valor de la media de las proyecciones se contrarreste a valores que pueden ser muy bajos.

Para esta función de aptitud, dado que lo que se busca es minimizar, se ha configurado el algoritmo de forma distinta: la relación de orden de los niveles definida por el fitness se ha invertido, para que la lista de niveles de la población inicial se ordene en orden creciente y se han invertido también los métodos de los pasos del algoritmo como la selección binaria y la inserción de nuevos individuos en la población estructurada.

Al igual que con la otra función de aptitud, en la figura 7 se muestra la evolución del fitness total de la población tomando el parámetro *MAX_ITER* igual a 2000.

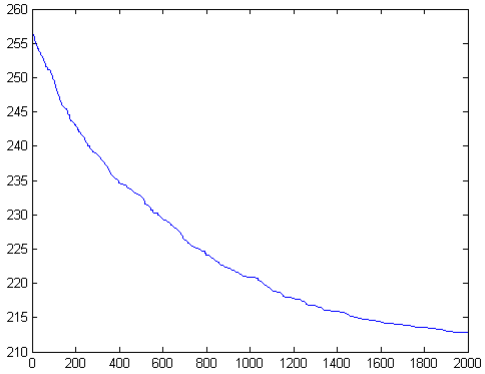

Fig. 7. Evolución del fitnes de la población

Función de evaluación basada en la simulación Se ha diseñado también otra función de aptitud basada en la simulación directa sobre el juego. Por el momento solo está pensada para medir la dificultad de alcanzar el objetivo en el primer lanzamiento, pero aún así da una buena estimación de la dificultad sobre los niveles.

El método, implementado en la clase *Nivel*, simula el lanzamiento de la partícula con multitud de vectores de velocidad diferentes, en módulo y ángulo, eliminando por supuesto la limitación del juego de iteraciones fijas por segundo y su parte gráfica. Al final de cada lanzamiento comprueba si la partícula ha quedado en estado de equilibrio sobre el planeta objetivo, es decir:

```
this.getParticle().getCollidedPlanet()==getPlaneta()[this.getiPlanetTarget()]
```

En caso afirmativo suma 1 a una variable local (para calcular al final la frecuencia relativa de aciertos) y hace una estimación de hasta que punto la forma de llegar al planeta objetivo fue fortuita (a mayor distancia recorrida por la partícula y mayor número de colisiones con otros planetas durante el trayecto se considera el acierto más debido al azar que a la intuición humana). En concreto, la fórmula usada es la siguiente:

$$\text{dificultadAcierto} = \frac{1}{\text{distancia timesCollided}}$$

Recuérdese la definición del atributo *timesCollided* en la definición de la clase *Planeta* de la sección 4.1. Este valor solo aumenta cuando una colisión se produce en un planeta distinto al anterior, por lo que como mínimo tiene valor 1 cuando la partícula solo ha colisionado con el planeta al que ha llegado. El cálculo supone entonces que solo una colisión más con otro planeta reduce a la mitad la capacidad de previsión del resultado (recordemos que la ayuda de la visión preliminar es recortada en el momento se produce la primera colisión).

Al igual que para calcular la frecuencia relativa, estos valores van sumándose y se el resultado final es dividido por el número total de pruebas. Por último, solo con fines de aportar mayor claridad a la magnitud, esta es escalada por 10^6.

5 Conclusiones

Se ha ejecutado el algoritmo con ambas funciones de aptitud y tomando el parámetro $MAX\_ITER = 1000$ en el caso de la función de las intersecciones y $MAX\_ITER = 500$ en el de minimizar la atracción en el planeta objetivo (valores totalmente factibles para la generación del contenido de forma *online*), para generar niveles con la intención de probarlos en jugadores reales. Con cada función de aptitud se ha tomado el mejor individuo de cada una de las tres clases de la población estructurada. El fitness obtenido por los tres niveles del método de las intersecciones se muestra en la tabla 3, mientras que los de la otra función están en el número 4.

Table 3. Intersecciones

Nivel	Fitness de la función	Frecuencia de aciertos	Dificultad simulador
Difícil	418.96	0.065%	0.144
Medio	391.51	0.457%	1.056
Fácil	354.25	3.162%	6.503

Table 4. Atracción Mínima

Nivel	Fitness de la función	Frecuencia de aciertos	Dificultad simulador
Difícil	2.2238	1.697%	3.655
Medio	2.3128	0.443%	2.286
Fácil	2.3664	4.242%	22.003

Se subió un applet en la web https://sites.google.com/site/gravityboules/ con los seis niveles generados y se propuso una encuesta de satisfacción con los niveles jugados (en la que los jugadores podían calificar los niveles del 1 al 5) con la intención de analizar posteriormente los datos y establecer la correlación entre la habilidad de los jugadores (medida por la puntuación en la partida) y los niveles preferidos. Lamentablemente en el tiempo que estuvo expuesta la encuesta no se registró una muestra lo suficientemente grande como para poder sacar conclusiones. Al menos pudo comprobarse con la función de simulación sobre el juego que efectivamente el algoritmo es capaz de generar, con ambas funciones de aptitud para el uso *online*, niveles ajustados a distinta dificultad, como puede observarse en los cuadros 3 y 4. La única excepción se presenta en el nivel medio de la segunda función de aptitud. Sin embargo la mayor diferencia se presenta en la frecuencia relativa (el nivel medio se alcanza alrededor de cuatro veces menos que el difícil), pero la diferencia con el valor que ofrece la función de simulación (cuarta columna) no es tan significativa. No obstante esto es un indicador de que no es suficiente estimar la dificultad solo con el primer lanzamiento y que este puede ser mejorado para ofrecer mayor precisión.

5.1 Resumen de objetivos

En resumen, en esta memoria se ha mostrado un juego sencillo, basado en una idea innovadora. Se ha comprobado que al mismo tiempo que sirve para generar terrenos tridimensionales para otro tipo de juegos y que define de forma implícita (aunque también básica) un motor de físicas para este.

El algoritmo diseñado para la generación de contenido introduce la idea de utilizar poblaciones estructuradas para ofrecer a los jugadores un amplio espectro de niveles de distinta dificultad, adaptado a todo tipo de jugadores, con la idea no resultar aburrido a los más experimentados ni frustrante a aquellos con menor habilidad. Para ello se han definido dos funciones de aptitud que pueden ejecutarse de forma *online*, con lo que se podría generar un juego infinito adaptado a cada tipo de jugador.

La calidad de las soluciones del algoritmo evolutivo con las funciones de aptitud implementadas han sido contrastadas por medio de otra función de evaluación de simulación directa. Esta función, aunque no está pensada para ser usada de forma *online* por su elevado coste, si que podría usarse para evaluar el nivel que vaya a ofrecerse al jugador. Así pueden evitarse situaciones como que se plantee un nivel de imposible resolución al jugador, algo que no resultaría aceptable.

La sencillez del juego permite que este mismo algoritmo pueda ser usado en otros juegos para distribuir por su mapa cualquier tipo de contenido opcional, con solo definir otras funciones de aptitud concretas. Resulta de especial interés el algoritmo de cruce implementado, que resulta más general que el concepto de recombinación en un punto para este juego.

5.2 Trabajos futuros

Como trabajos futuros resultaría especialmente interesante poder realizar el análisis de la correlación de la habilidad de los jugadores con su elección de niveles, algo que no se ha podido realizar debido a no disponer de una muestra lo suficientemente amplia. También podrían introducirse muchas mejoras en el algoritmo evolutivo. Por ejemplo, podría tratarse la optimización multiobjetivo de los niveles. Por último sería también interesante mejorar la función de evaluación por medio de simulación con técnicas de inteligencia artificial, de forma que se pueda crear un verdadero agente capaz de jugar un nivel del juego, realizando varios lanzamientos tratando alcanzar el objetivo.

References

1. S.J. Aarseth. *Gravitational N-Body Simulations: Tools and Algorithms*. Cambridge Monographs on Mathematical Physics. Cambridge University Press, Cambridge; New York, 2003.
2. Miguel Monteiro de Sousa Frade. *Evolving Artificial Terrains with Automated Genetic Terrain Programing*. PhD thesis, Universidad de Extremadura, July 2012.
3. Anna Isabel Esparcia-Alcázar, editor. *Applications of Evolutionary Computation - 16th European Conference, EvoApplications 2013, Vienna, Austria, April 3-5, 2013. Proceedings*, volume 7835 of *Lecture Notes in Computer Science*. Springer, 2013.
4. M. Frade, F.F. de Vega, and C. Cotta. Evolution of artificial terrains for video games based on obstacles edge length. In *Evolutionary Computation (CEC), 2010 IEEE Congress on*, pages 1–8, 2010.
5. Miguel Frade, Francisco Fernandez de Vega, and Carlos Cotta. Evolution of artificial terrains for video games based on accessibility. In *Proceedings of the 2010 international conference on Applications of Evolutionary Computation - Volume Part I*, EvoApplicatons'10, pages 90–99, Berlin, Heidelberg, 2010. Springer-Verlag.
6. NPD Group. Total consumer spent on all games content in the u.s. estimated between 16.3 to 16.6 billion, 2011.
7. Erin J. Hastings, Ratan K. Guha, and Kenneth O. Stanley. Automatic Content Generation in the Galactic Arms Race Video Game. *IEEE Transactions on Computational Intelligence and AI in Games*, 1(4):245–263, 2009.
8. Erin J. Hastings, Ratan K. Guha, and Kenneth O. Stanley. Evolving content in the galactic arms race video game. In *Proceedings of the 5th international conference on Computational Intelligence and Games*, CIG'09, pages 241–248, Piscataway, NJ, USA, 2009. IEEE Press.
9. Mark Hendrikx, Sebastiaan Meijer, Joeri Van Der Velden, and Alexandru Iosup. Procedural content generation for games: A survey. *ACM Trans. Multimedia Comput. Commun. Appl.*, 9(1):1:1–1:22, February 2013.
10. Raúl Lara-Cabrera, Carlos Cotta, and Antonio J. Fernández-Leiva. A procedural balanced map generator with self-adaptive complexity for the real-time strategy game planet wars. In Esparcia-Alcázar [3], pages 274–283.

11. Simon M. Lucas. Computational intelligence and ai in games: A new ieee transactions. *IEEE Trans. Comput. Intellig. and AI in Games*, 1(1):1–3, 2009.
12. Simon M. Lucas, Michael Mateas, Mike Preuss, Pieter Spronck, and Julian Togelius. Artificial and Computational Intelligence in Games (Dagstuhl Seminar 12191). *Dagstuhl Reports*, 2(5):43–70, 2012.
13. K. F. Man, K. S. Tang, and S. Kwong. *Genetic Algorithms: Concepts and Designs with Disk*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2nd edition, 1999.
14. Joe Marks and Vincent Hom. Automatic design of balanced board games. In Jonathan Schaeffer and Michael Mateas, editors, *AIIDE*, pages 25–30. The AAAI Press, 2007.
15. Keith J. Mirenberg. Introduction to gravity-well models of celestial objects.
16. Chris Pedersen, Julian Togelius, and Georgios N. Yannakakis. Modeling player experience in super mario bros. In *Proceedings of the 5th international conference on Computational Intelligence and Games*, CIG'09, pages 132–139, Piscataway, NJ, USA, 2009. IEEE Press.
17. Noor Shaker, Georgios N. Yannakakis, and Julian Togelius. Towards automatic personalized content generation for platform games. In G. Michael Youngblood and Vadim Bulitko, editors, *AIIDE*. The AAAI Press, 2010.
18. J. Togelius, G.N. Yannakakis, K.O. Stanley, and C. Browne. Search-based procedural content generation: A taxonomy and survey. *Computational Intelligence and AI in Games, IEEE Transactions on*, 3(3):172–186, 2011.
19. Julian Togelius, Renzo De Nardi, and Simon M. Lucas. Towards automatic personalised content creation for racing games. In *Computational Intelligence and Games, 2007. CIG 2007. IEEE Symposium on*, pages 252–259. IEEE, April 2007.
20. Julian Togelius, Mike Preuss, Nicola Beume, Simon Wessing, Johan Hagelbäck, and Georgios N. Yannakakis. Multiobjective exploration of the starcraft map space, 2010.
21. Julian Togelius, Rued Langgaards Vej, Mike Preuss, and Georgios N. Yannakakis. Towards multiobjective procedural map generation. In *in Workshop on Procedural Content Generation in Games, co-located with 5th Intl. Conference on Foundations of Digital Games. ACM Digital Library*, 2010.