

SEP

TNM

INSTITUTO TECNÓLOGICO DE TIJUANA

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN



GENERACIÓN PROCEDURAL DE NIVELES PARA EL
VIDEOJUEGO DE ANGRY BIRDS UTILIZANDO
COMPUTO EVOLUTIVO

TRABAJO DE TESIS

PRESENTADO POR

JAIME SALINAS HERNÁNDEZ

PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS
EN COMPUTACIÓN

DIRECTOR DE TESIS
DR. JOSÉ MARIO GARCÍA VALDEZ

Tijuana, B.C., México. Noviembre 2019

Resumen

En el documento de tesis presentado se propone un algoritmo evolutivo que utiliza enfoques de la teoría de open-ended evolution para generar niveles para el videojuego de Angry Birds. Los niveles utilizados en este juego se componen de elementos tales como una cierta cantidad de aves que el usuario puede disparar a manera de resortera, una cantidad determinada de puerquitos que se requieren eliminar como objetivo para avanzar en los niveles, así como de una determinada cantidad de piezas que conforman estructuras que sirven como obstáculos para el usuario. El objetivo del sistema que aquí se propone es el de generar las diferentes estructuras que se presentan en el juego, teniendo en cuenta las características que deben de cumplir para poder ser utilizadas, es decir que sean llamativas y funcionales. Para esto se propone una búsqueda de múltiples capas, la primera parte se trata de construir estructuras utilizando piezas básicas del juego, posteriormente utilizar estas estructuras como base y continuar generando desde ese punto para obtener estructuras más complejas. Se utilizará un enfoque basado en open-ended evolution el cual será utilizado para la evolución de las estructuras con el objetivo de crear otras más complejas y diversas entre sí. La función de aptitud que se utilizara para evaluar los niveles considera la complejidad de los niveles y que tan diferentes son estos del resto de la población de niveles generada. Los experimentos realizados muestran que un enfoque evolutivo permite la generación de niveles que logran ser novedosos y son interesantes para ser jugados.

Abstract

In this thesis, we propose an evolutionary algorithm that follows an open-ended evolution approach to generate levels for the Angry Birds video game. The levels themselves are composed of a set of birds that the player can throw with a slingshot, a certain amount of pigs that they must destroy, and a given number of pieces that conform structures that may or may not protect the pigs from birds thrown at them. The current goal is the generation of diverse structures that are playable in the game, having the additional characteristics of being fun and enjoyable. We propose a multi-layered search, first constructing composite structures from basic blocks, to then build more complex structures building from these composites. We follow an open-ended evolution approach in which the evolution of structures is not guided towards a single objective but is rather free to evolve and generate novelty or diversity. The fitness function we use to evaluate the proposed levels considers how complex levels have become and how different they are from the rest of the population. The experiments conducted show that an evolutionary approach allows the generation of levels that are novel and interesting to play.

Agradecimientos

Agradezco principalmente a mi familia, quien sin duda alguna me brindó todo el apoyo y comprensión necesaria para poder continuar con mi desarrollo académico, su motivación brindada para ayudarme a salir adelante es sin duda lo que me ayudó a llegar tan lejos.

Agradezco a mis amigos y compañeros de maestría Sergio y Eduardo con quienes me embarqué en esta aventura de conocimiento, agradezco su compañía, su tiempo y su apoyo.

Agradezco a los profesores con quienes cursé mis materias, M.C.C. Alejandra Mancilla, Dr. Fevrier Valdez, Dra Denisse Paulette, Dra. Gabriela Martínez, Dra. Claudia Gonzalez, Dr. Jose Soria, Dr Oscar Castillo, Dra. Patricia Melin, gracias al conocimiento adquirido por ustedes es que logré llegar hasta aquí.

Un gran agradecimiento a mi director de tesis el Dr. Mario Garcia Valdez por el apoyo que me brindó, su paciencia con mis errores, el tiempo que dedicó a ayudarme con mis dudas y la comprensión brindada.

Finalmente, le agradezco al Instituto Tecnológico de Tijuana por la educación de alta calidad recibida, de igual manera agradezco a CONACYT por el apoyo otorgado a través del Programa Nacional de Posgrados de Calidad (número de CVU 797248).

Índice general

Índice general	i
Índice de Figuras	iv
List of Tables	vi
1 Introducción	1
1.1 Justificación	4
2 Preliminares	6
2.1 Algoritmos Genéticos	6
2.2 Generación procedural de contenido	9
2.2.1 PCG en el ámbito de videojuegos	10
2.2.2 Áreas de interés de generación procedural	10
2.2.3 Diseño de niveles	11
2.2.4 Gráficos	14
2.2.5 Audio	15
2.2.6 Narrativa	16
2.2.7 Reglas y Mecánicas	17
2.2.8 Juegos	20
2.3 Jugabilidad	21
2.4 Open-Ended Evolution	24
3 Estado del arte	25

ÍNDICE GENERAL

3.1	Competencia IEEE CoG	25
3.2	Algoritmos de búsqueda	26
3.3	Agentes para jugar	28
4	Propuesta del proyecto	30
4.1	Propuesta utilizada	30
4.1.1	Generar compuestos de piezas base	31
4.1.2	Generación de compuestos mediante objetos de clase	32
4.1.3	Generar niveles utilizando algoritmos genéticos	36
4.2	Propuestas anteriores	37
4.2.1	Propuesta orientada a métodos	38
4.2.2	Regla de tercios	39
5	Implementación	43
5.1	Codificando los diccionarios de piezas	43
5.2	Creación de compuestos	45
5.3	Definición de clase auxiliar de individuo	46
5.4	Generación de individuos	47
5.4.1	Integración de miembros elite	49
5.4.2	Operador de selección	51
5.4.3	Operador de cruce	52
5.4.4	Operador de mutación	54
5.4.5	Representacion de individuos	57
5.4.6	Cálculo de fitness	60
5.4.7	Selección de miembros élite	66
6	Experimentos y Resultados	68
6.1	Generación de niveles estables	70
6.1.1	Experimento 1	71
6.1.2	Experimento 2	75

ÍNDICE GENERAL

6.1.3	Experimeto 3	79
6.2	Diversidad del contenido creado	83
7	Conclusiones y trabajo futuro	88
7.1	Conclusiones	88
7.2	Trabajo futuro	90
	Bibliografia	93
A	Anexos	97
A.1	Código del sistema	97
	Código principal	97
	Código para calculo de fitness	125

Índice de Figuras

2.1 Ciclo de vida de un algoritmo genético	8
2.2 Áreas de PCG en videojuegos	11
2.3 Nivel generado en el juego Disgaea	13
2.4 Ejemplo de un juego en progreso en el juego de Yavalath	19
2.5 Puntos de evaluación de jugabilidad	22
4.1 Piezas básicas del juego angry birds	31
4.2 Cálculo de bordes de un conjunto	32
4.3 Diagrama de clase de piezas	33
4.4 Diagrama de clase de Composite	34
4.5 Diagrama de clase de Individuo	35
4.6 Diagrama de flujo del algoritmo	37
4.7 Ejemplo de una estructura antes(izquierda) y después(derecha) de una simulación	38
4.8 Ejemplo del uso de la regla de tercios en una imagen, el punto de interés se encuentra en la parte central	40
4.9 Ejemplo del uso de la regla de tercios en un conjunto de piezas, el cromosoma de un individuo(izquierda) se combina con la máscara(centro) para generar una estructura más compleja(derecha)	42
4.10 Resultado obtenido utilizando la regla de tercios, la máscara(izquierda) utilizada y el resultado generado(derecha)	42

ÍNDICE DE FIGURAS

5.1	Cruce de un punto (arriba) y cruce a dos puntos (abajo) aplicada a la máscara de individuos	53
5.2	Ejemplo del operador de mutación enfocado a remover y agregar compuestos	55
5.3	Ejemplo del operador de mutación enfocado a cambiar el material de los compuestos, pre-mutación (izquierda) y post-mutación (derecha)	56
5.4	Ejemplo del operador de mutación enfocado a cambiar los compuestos asignados, pre-mutación (izquierda) y post-mutación (derecha)	57
5.5	Ejemplo de la combinación de capas de un individuo	58
5.6	Ejemplo de la combinación de un genotipo armado de la Figura 5.5 con la colocación de enemigos	61
6.1	Resultados del experimento, gráfica de la función de aptitud figura a), ejemplos de individuos: figuras b) y c)	73
6.2	Resultados del experimento, primer y segundo mejor nivel generado (sin repetir)	74
6.3	Resultados del experimento, gráfica de la función de aptitud figura a), ejemplos de individuos: figuras b) y c)	77
6.4	Resultados del experimento, primer y segundo mejor nivel generado (sin repetir)	78
6.5	Resultados del experimento, gráfica de la función de aptitud figura a), ejemplos de individuos: figuras b) y c)	81
6.6	Resultados del experimento, primer y segundo mejor nivel generado (sin repetir)	82
6.7	Evolución de los individuos de un experimento	84
6.8	Conjunto 1 de estructuras resultantes en las simulaciones	86
6.9	Conjunto 2 de estructuras resultantes en las simulaciones	87

List of Tables

6.1 Parámetros utilizados en el algoritmo genético	69
--------------------------------------------------------------	----

Capítulo 1

Introducción

El método de generación de contenido procedural ha sido tomado con gran interés en el ámbito de desarrollo de videojuegos debido a los beneficios que brinda a las compañías para poder generar contenido que pueda ser utilizado en juegos. No es posible asociar la generación procedural de contenido con algún aspecto específico de la creación de videojuegos o inclusive únicamente en el enfoque de videojuegos debido a que el tema puede ser utilizado para aspectos diferentes, desde la generación de áreas como en el videojuego Spelunky [1] [2] [2], la generación de historias, que sean inmersivas como en Façade [3], la generación de piezas sonoras como las que se pueden escuchar en Audioverdrive [4], generación de efectos gráficos como el diseño de pétalos en Petalz [5], así como otros en los enfoques de gameplay y videojuegos completos.

Estas ideas han sido de mutuo interés entre la industria y la comunidad científica por lo que en 2005 se originó un simposio de la IEEE en el ámbito de Computación Inteligente y Videojuegos (Computational Intelligence and Games) el cual posteriormente pasaría a ser una conferencia en el año 2009.

Esta conferencia tiene el fin de unir a ambos grupos de interés en un mismo lugar con el fin de conocer sobre los avances de la computación inteligente enfocada en juegos, dentro del marco de la misma conferencia se celebran competencias de diferentes juegos tales como inteligencia artificial en el juego de Hearthstone, jugar niveles de Ms. Pac-Man,

CAPÍTULO 1. INTRODUCCIÓN

competencias de agentes para ganar en juegos de pelea y varios otros.

Dentro de estas mismas competencias se celebra la de generación de niveles para el videojuego de Angry Birds la cual consiste primordialmente en generar niveles que sean interesantes visualmente y sean complejos en el sentido de que sean difíciles de completar por una persona pero que al mismo tiempo no sean imposibles de completar, este juego fue desarrollado por la compañía Rovio Entertainment Corporation[1] en el año 2009, el objetivo dentro del juego consta de utilizar un número dado de aves con diferentes efectos para eliminar puercos color verde que se encuentran protegidos e incluso sobre estructuras de diferentes materiales y formas, en algunos casos al golpear partes de las estructuras se crea un efecto dominó que termina por destruir la mayoría de las misma, el juego cuenta con un sistema de gravedad que permite que las trayectorias de tiro generen diferentes efectos en las estructuras. En este trabajo el interés principal es que mediante el uso del sistema de gravedad del juego se puedan crear estructuras interesantes visualmente y que sean lo suficientemente robustas para soportar golpes de las mismas aves y logre mantenerse en pie el mayor tiempo posible.

El objetivo principal de este proyecto es el de generar un sistema basado en cómputo evolutivo que logre generar las estructuras que conformarán los niveles del juego de Angry Birds y evaluar estas mismas estructuras mediante los aspectos propuestos en la competencia para que sean entretenidos, complejos y con cierto grado de dificultad. Los objetivos que se enmarcaran en el proyecto son los siguientes:

- Adaptar un algoritmo genético para que sea capaz de generar secuencias de estructuras para generar un nivel cumpliendo los requerimientos.
- Modificar y adaptar el software de simulación para poder obtener datos específicos de un nivel requeridos para su evaluación.
- Adaptar el método de evolución para permitir que se puedan generar estructuras complejas.
- Explorar las posibilidades de integrar alguna otra técnica de cómputo evolutivo para

CAPÍTULO 1. INTRODUCCIÓN

optimizar la generación de niveles.

El trabajo propuesto tiene como objetivo la generación de contenido con el cual un jugador podrá interactuar en el videojuego de Angry Birds, cabe remarcar que el uso de métodos de generación procedural han sido utilizados con el propósito de realizar competencias en el marco de la conferencia del Instituto de Ingeniería Eléctrica y Electrónica (IEEE) que lleva por nombre Conferencia de Juegos (CoG), en dicha competencia se utilizan algoritmos capaces de generar los niveles que una persona o agente puede jugar. En nuestro caso la metodología propuesta en este proyecto pretende servir de base para mostrar que es posible la utilización de algoritmos evolutivos libres de tal manera que el contenido generado será único y entretenido para los usuarios. En este mismo capítulo se presentan las motivaciones para la realización del proyecto así como la manera en que la computación evolutiva sirve de apoyo en la industria del entretenimiento (ver la Sección 1.1).

El método propuesto en este documento consta de tres campos diferentes, estos campos son Generación de Contenido Procedural y dos áreas de la computación evolutiva las cuales son algoritmos genéticos y la evolución abierta. El Capítulo 2 está dedicado a explicar los conceptos de estas áreas, mismos que son necesarios para comprender mejor el contenido de los capítulos subsecuentes.

Una vez que los conceptos mostrados en el capítulo anterior han sido vistos en el Capítulo 3 se presentan un conjuntos de trabajos relacionados con lo que se quiere lograr. Estos trabajos permiten tener un mejor entendimiento de que es lo que se puede llegar a lograr con los métodos presentados así como el comprender las técnicas existentes para trabajar en el proyecto.

Posteriormente en el Capítulo 4 se presenta la propuesta del proyecto de tesis, y el enfoque elegido para resolver el problema de generación de contenido para el videojuego de Angry Birds. De igual manera se toca el tema de las propuestas que previamente se habían presentado para resolver el mismo problema y como fue que mediante la modificación y adaptación de conceptos de estas ideas se logró llegar a la propuesta definitiva.

CAPÍTULO 1. INTRODUCCIÓN

La implementación particular del método propuesto se presenta en el Capítulo 5, este Capítulo cubre todo el aspecto técnico del proyecto tal como los lenguajes de programación utilizados, las adaptaciones que realizaron a los archivos de ejecución del software que se utiliza para la simulación así como la manera en la que las ideas propuestas en el Capítulo 4 fueron implementadas en el código, debido a que los conceptos que se presentaron en el Capítulo 4 son adaptadas en este.

Después de realizar las implementaciones pertinentes de las ideas presentadas en el Capítulo 4 se procede a realizar un conjunto de experimentos del sistema funcional. Los experimentos permiten comprender la manera de realizar la generación de contenido con los métodos propuestos, además de que los resultados obtenidos sirven de apoyo para analizar las partes en dónde se pueden realizar mejoras para un mejor desempeño. Estos experimentos se presentan en el Capítulo 6, en este mismo capítulo se analizan los resultados y se dan comparaciones con diferentes variaciones del método y con otras ideas de los trabajos presentados en el Capítulo 3.

Finalmente en el Capítulo 7 se presentan la conclusiones a las que se logró llegar después de concluir el trabajo así como varias propuestas que se pueden agregar o secciones que se pueden optimizar para mejorar el proyecto en un futuro.

Este proyecto de tesis describe un método innovador para la generación de estructuras del juego mediante el uso de técnicas evolutivas como la combinación de un algoritmo genético con un algoritmo de evolución abierta. Este método es robusto y permite modificaciones rápidas a las características evolutivas y de evaluación, además de que es flexible para poder establecer restricciones en el contenido generado. Por ejemplo que no se utilicen ciertas piezas o que se generen niveles con cierta cantidad de objetivos.

1.1 Justificación

La justificación de este proyecto de tesis se presenta en dos puntos diferentes.

- Tomando en cuenta el punto de vista científico, por el cual se tiene interés particular en el proyecto. La utilización de métodos de generación procedural de contenido en

CAPÍTULO 1. INTRODUCCIÓN

este caso de generación de niveles es un caso de estudio muy particular debido a las restricciones que se deben de tomar en cuenta al momento del diseño, debido a que varios aspectos tales como los diferentes terrenos, el posicionamiento y balance de los elementos colocados deben de ser tomados en cuenta de una manera que se asemeje a una manera que podrían ser posicionados en el mundo real con restricciones de gravedad para poder generar estructuras que logren cumplir los aspectos denotados por la jugabilidad que hacen a un nivel viable e interesante.

- Ámbito tecnológico que toma el proyecto para las empresas que utilizan este tipo de recursos en el diseño de videojuegos, esto es debido a que las mismas empresas muchas veces se ven presionados en cumplir fechas de salida para dichos productos de esta manera el uso de este tipo de recursos permite generar el contenido de manera automatizada y que logre cumplir con los estándares establecidos por las compañías.

Se propone, mediante el uso de un algoritmo genético agregarle complejidad a un nivel mediante la evolución de las estructuras que lo componen, esto debido a que es posible realizar operaciones de cruce y mutaciones que lograran producir nuevas estructuras como posibles soluciones ademas de que permite encaminar el sistema a encontrar las mejores mediante funciones objetivo. Esto es interesante debido a las posibilidades de generación del sistema debido a que es posible obtener tanto estructuras como combinaciones de elementos que normalmente una persona no utilizaria de manera manual.

Finalmente, el metodo propuesto provee un sistema que es capaz de crear niveles que seran entretenidos, únicos e interesantes para los jugadores.

En los capítulos siguientes se presentan las bases teóricas que se tomaron en cuenta para el desarrollo de este proyecto, los diferentes métodos que se han utilizado para atacar la problemática descrita, así como la propuesta que definimos para resolver este mismo problema junto son sus resultados y conclusiones.

Capítulo 2

Preliminares

En este capítulo se explican los conceptos básicos para el desarrollo del proyecto, separados en secciones. El tema principal son los algoritmos genéticos, además de esto se explican de manera detallada los conceptos que se tomarán en cuenta en el desarrollo de los métodos que componen el sistema, siendo estos: open-ended evolution, generación procedural de contenido y los conceptos básicos de jugabilidad.

2.1 Algoritmos Genéticos

Los algoritmos genéticos (Genetic Algorithms, GAs por sus siglas en Inglés) son algoritmos inspirados en la evolución Darwiniana y utilizados para la optimización de procesos, fueron propuestos por John Holland en 1975 [6], los GAs son métodos de optimización y búsqueda basados en los principios de la selección natural y genética, la manera de representarlos es mediante un grupo de individuos que representan posibles soluciones a un problema y mediante el uso de operadores genéticos estos individuos se cruzan y evolucionan para acercarse más al objetivo, el cual puede ser minimizar o maximizar una función de utilidad para un sistema.

De acuerdo a una explicación proporcionada por Whitley en uno de sus artículos [7], un algoritmo genético (GA) es un algoritmo de optimización inspirado en el proceso natural de evolución. Por esta razón los GAs son considerados como parte de una subárea de

CAPÍTULO 2. PRELIMINARES

algoritmos de optimización llamada algoritmos evolutivos. La manera en cómo funciona un GA es que se propone tener un conjunto inicial de posibles soluciones al problema, estas soluciones son llamadas población dentro del algoritmo. Cada una de estas soluciones propuestas llevan el nombre de individuos y en ocasiones se definen como cromosomas. Esta población es evaluada mediante el uso de una función de aptitud encargada de determinar el rendimiento de cada uno de los individuos como una de las posibles soluciones para el problema que se está tratando. Este tipo de problemas comúnmente involucran encontrar un conjunto de parámetros que permitan minimizar o maximizar los resultados obtenidos en un sistema.

El proceso que se realiza en un GA se describe de la siguiente manera, primero la población es sometida a un proceso de evolución que involucra la realización de cruces entre los individuos de la población que tienen el mejor rendimiento en la generación, de igual manera aquellas soluciones que tengan un mal desempeño de acuerdo a la función de aptitud son removidos de la lista de la población para las generaciones subsecuentes. El proceso de cruce de individuos se realiza mediante la recombinación de genes en los cromosomas de los individuos seleccionados los cuales son seleccionados en base a sus resultados en la función de aptitud siendo aquellos que llevaran a cabo los cruces los mejores. Este proceso mencionado se repite durante un número determinado de iteraciones las cuales llevan el nombre de generaciones.

El objetivo de los GAs así como en otros algoritmos evolutivos es que, mientras las generaciones van pasando la población comenzará a volverse más apta y se encaminará a cierto punto o valor que se espera sea el indicado para resolver un problema, sin embargo, es común que ninguno de los individuos logre llegar a una solución óptima para el problema presentado. Sin embargo, al ejecutar el algoritmo múltiples veces el algoritmo evolutivo logrará proveer varios resultados diferentes, esto es debido a que la población inicial en cada ciclo o ejecución se genera de manera aleatoria. Por esta razón, los algoritmos evolutivos y por ende los algoritmos genéticos son considerados como algoritmos de búsquedas metaheurísticas, debido a que son capaces de encontrar soluciones casi óptimas para problemas en donde los algoritmos evolutivos no sean involucrados de manera directa, además de

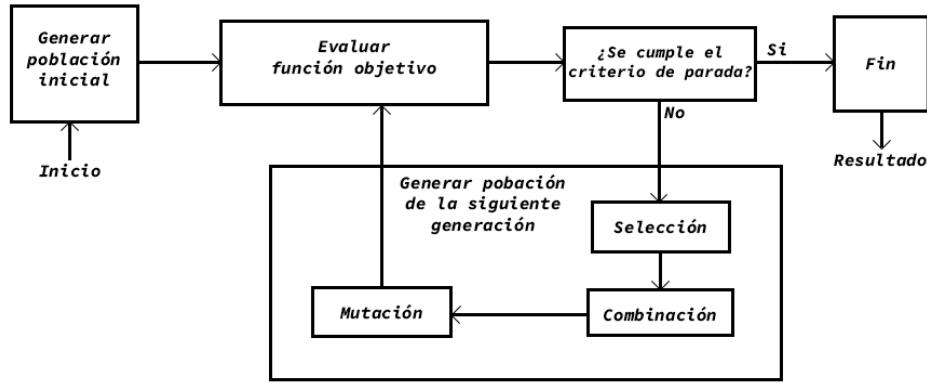


Fig 2.1: Ciclo de vida de un algoritmo genético

estos los algoritmos evolutivos también son considerados algoritmos estocásticos debido a la aleatoriedad que conlleva utilizar este tipo de algoritmos [8].

De esta manera el ciclo de vida de un algoritmo genético se resume de la siguiente manera:

1. Inicializar una población aleatoria de n individuos.
2. Evaluar la aptitud de cada individuo(solución) en la población.
3. Revisar si se ha llegado a una condición de terminación (el valor buscado o un número máximo de iteraciones)
4. En caso de que no:
 - (a) Introducir nuevos individuos a la población mediante los operadores de selección y cruce.
 - (b) Mutar de manera aleatoria a uno de los individuos nuevos.
5. Repetir desde el punto 2 hasta llegar a la solución o se cumpla el límite de generaciones.
6. Seleccionar el mejor individuo de la población como la solución del problema.

La lista anterior se muestra gráficamente en la Figura 2.1, en donde se puede apreciar más claramente como los operadores de selección, cruce y mutación se engloban en una sola área que es la de la generación e integración de nuevos individuos, éstos también se evalúan para definir si el algoritmo ha logrado alcanzar el punto de terminación, como se explicó anteriormente estos nuevos individuos no se eliminan en caso de no haber sido los ideales, sino que se comparan con los individuos originales y en caso de ser mejores irán reemplazando a los peores con el fin de encaminar al algoritmo a un punto específico que se espera sea la solución idónea.

2.2 Generación procedural de contenido

El término "Generación procedural de contenido" (Procedural Content Generation o PCG por sus siglas en inglés) denota la manera de crear contenido de manera automática mediante algoritmos, en lugar de generar los mismos contenidos de manera manual.

La PCG es un sistema de generación de contenido utilizado desde finales de los 70s, inicialmente se utilizaba para generar los laberintos de algunos juegos utilizando arte ASCII donde se denotaba la distribución de los cuartos y objetos que se podían encontrar en juegos estilo "*rogue*" o juegos que simulan un juego RPG de mesa, actualmente es ámbito de la generación de contenido ah tomado varias variantes y cada vez más empresas toman un interés por el uso de este tipo de herramientas de diseño, cabe remarcar que los diferentes ámbitos no están del todo refinados sin embargo dada la información necesaria y aplicados a aspectos específicos en el ciclo de desarrollo son capaces de crear los contenidos que se requieran.

Mientras que la generación de contenido es un aspecto que se ha utilizado en muchos ámbitos diferentes como en el fotografía, video, anuncios y arte digital, en el área de videojuegos se maneja el uso de generación "procedural" de contenido en donde procedural se define como el proceso computacional de una función particular, en este caso lo que se busca con la generación procedural de contenido es reducir el tiempo que toma generar contenidos de manera manual, la siguiente sección explica más detalladamente como se

utiliza esta mecánica en el área de videojuegos.

2.2.1 PCG en el ámbito de videojuegos

Dentro del ámbito de videojuegos la generación de contenido procedural es un aspecto que ha tenido un gran impulso en tiempos recientes debido a que muchas empresas de preocupan por sacar al mercado juegos de manera continua, estos mismos muchas veces en ciclos de desarrollos muy cortos, por lo mismo, se han buscado diferentes maneras de reducir dichos problemas. Una herramienta útil que ha surgido debido a esto es la generación procedural la cual permite reducir no solo los tiempos de desarrollo, sino que también permite recudir el espacio de memoria total de un juego en particular debido a que ya no es necesario tener todos los archivos englobados en un solo lugar, sino que lo requerido se obtiene de manera automática, de esta misma manera se les permite a las empresas reducir el número de personal necesario y por tal reducir los costos de desarrollo.

La generación procedural de contenido tiene tres objetivos principales:

- Brindar apoyo a los creadores para poder crear contenido mas rápidamente.
- Utilizar la generación de contenido para crear juegos que logran reaccionar en tiempo real a las acciones de los usuarios, cosa que en caso contrario tendría que encaminarse a escenarios específicos.
- Reducir el espacio en memoria tomado por el contenido generado.

Una cosa extra que brinda la generación procedural de contenido es permitir una mayor creatividad al momento de generar.

2.2.2 Áreas de interés de generación procedural

La generación procedural de contenido es un sistema que se enfoca en diferentes aspectos en el área del entretenimiento especialmente en el área de videojuegos, estos aspectos pueden o no estar ligados unos con otros y además cabe mencionar que no se enfocan

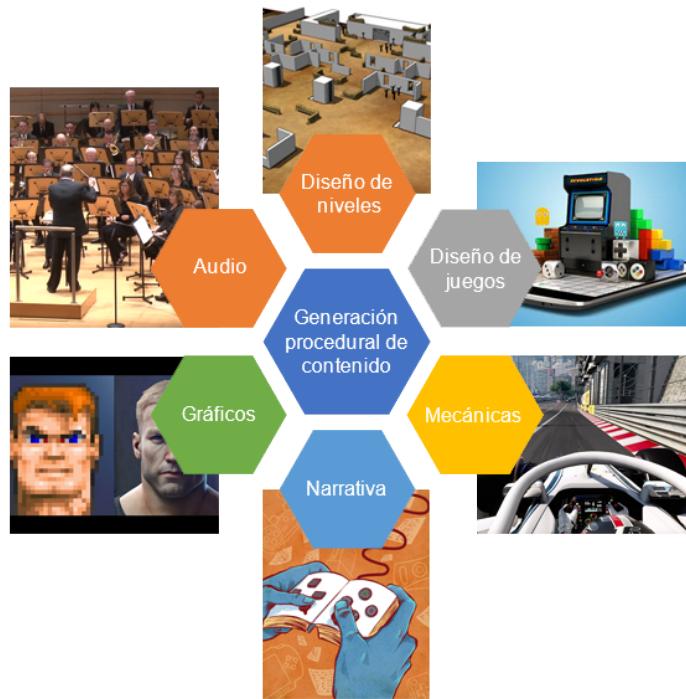


Fig 2.2: Áreas de PCG en videojuegos

primordialmente en la generación de "objetos" sino más bien en la generación de recursos que pueden ser utilizados en el desarrollo de videojuegos, la generación procedural se enfoca en seis puntos mostrados en la Figura 2.2, cada uno se explica en un apartado iniciando con el 'Diseño de niveles' en el capítulo 2.2.3, la generación de gráficos en el capítulo 2.2.4, la creación de audio en el capítulo 2.2.5, creación de narrativa o historias en el capítulo 2.2.6, diseño de reglas y mecánicas en el capítulo 2.2.7 y la generación de juegos completos explicada en el capítulo 2.2.8.

2.2.3 Diseño de niveles

El área del desarrollo de niveles es una de las populares en PCG debido a que los niveles son la parte esencial de un juego, debido a que es el área sobre la cual un jugador puede interactuar, la representación de estas áreas pueden ser desde imágenes en 2D, simplemente con el alto y ancho de los objetos, hasta elementos en 3D que abarquen también el grosor

CAPÍTULO 2. PRELIMINARES

de los objetos.

Debido a que los niveles son esencialmente el área principal de interacción en un videojuego, éstos deben de considerar los aspectos de funcionalidad y estética para que exista una buena interacción y sea llamativo a los usuarios, de esta manera se pueden crear áreas con tonos oscuros y ambiente tétrico para entregar un nivel con temática de terror. La generación de niveles de juego es un tema reciente en el ámbito de investigación debido a los elementos que se deben de tener en cuenta sin embargo dentro del ámbito de videojuegos, es un elemento comúnmente utilizado, algunos ejemplos recientes siendo Spelunky, Minecraft y Disgaea.

Primero tenemos el juego de plataforma Spelunky desarrollado por Derek Yu [2] [9], el juego consta de múltiples niveles alrededor de cinco diferentes áreas, cada área con un estilo diferente (niveles con hielo, lava, etc.) en este juego los niveles que recorre el jugador son desarrollados de manera procedural por diferentes algoritmos dependiendo el área en la que se encuentre el jugador.[10]

El segundo ejemplo es un videojuego desarrollado por Markus Persson llamado urls o referencias Minecraft, [11] [12] este juego consta de un área estilo caja de arena en donde el jugador puede realizar las acciones que quiera, las estructuras están definidas en manera de bloques con diferentes texturas y propiedades que el jugador puede utilizar para construir diferentes cosas, la manera en cómo se utiliza la generación de contenido es mediante la generación de todo un nivel al iniciar el juego de tal manera que se utiliza una semilla de generación y se utiliza el método *Perlin Noise* [13] 3D con interpolación lineal para generar los biomas, elementos y entidades que conformaran el nivel, además de esto el área inicial del juego no es la única área que puede ser recorrida durante la sesión de juego, sino que el sistema utiliza una "semilla"(seed) que define la manera en cómo está conformado el mapa, mientras más se va recorriendo del mapa las áreas se van generando acorde a la semilla de manera infinita. El juego también cuenta con un sistema que genera los biomas que aparecen en el juego, en este caso la manera en como se generan es mediante el apoyo de un diagrama Whittaker [14] el cual es una manera de dividir las áreas generadas y categorizarlas en diferentes biomas con sus propias temperaturas, flora y fauna.

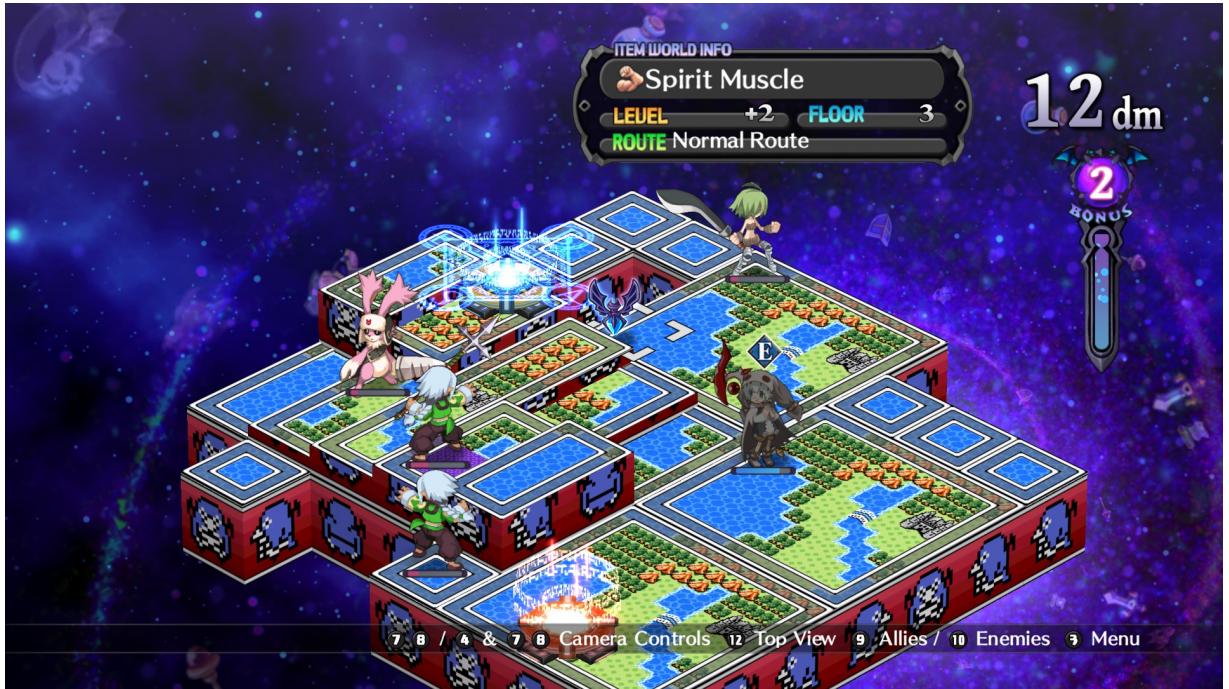


Fig 2.3: Nivel generado en el juego Disgaea

Finalmente tenemos el juego Disgaea desarrollado por Nippon Ichi Software [15], este es un juego de estrategia por turnos en donde el objetivo es eliminar a todos los enemigos en un mapa para continuar al siguiente, la manera en cómo se utiliza la generación de contenido procedural es en una sección extra del juego en donde se puede entrar a un arma para completar niveles gradualmente mas difíciles para darle más poder a dicha arma, en esta parte los niveles que se encuentra el jugador son generados proceduralmente tomando en cuenta las alturas de las partes donde se puede caminar y la colocación de los enemigos en el mapa 2.3, para la generación se toma en cuenta el nivel del arma y su rareza, mientras más altos sean estos valores de igual manera los niveles generados serán más difíciles. Este tipo de generación de niveles también es conocido como generación aleatoria de niveles durante proceso de ejecución (del Inglés runtime random level generation) el cual es un proceso en donde el contenido se va generando gradualmente durante el tiempo de ejecución de un software [16], este es utilizado en diferentes juegos para generar diferentes áreas de los mismos.

2.2.4 Gráficos

El área de desarrollo de gráficos se encarga principalmente de generar las representaciones visuales de los juegos debido a que la mayor parte de los juegos llevan una parte visual a menos que no se requiera, es necesario generar una imagen que denote lo que se quiere dar a entender en un juego, de esta manera se le brinda al jugador un nivel más de inmersión en la situación, mediante el uso de paletas de colores o imágenes en pantalla que puedan representar mejor las situaciones que se presentan.

El ámbito de generación de gráficos ha sido uno de los más explotados debido a que es posible generar gráficos que van desde simples representaciones de 8 bits a representaciones foto realísticas de los eventos u objetos presentes en un juego. Tal es el caso explicado en el paper de Risi S. et al.[5] en donde utilizan una red de producción de patrones compositacionales (CPPN por sus siglas en inglés) la cual es una variante de las redes neuronales artificiales (ANN) regulares, utilizando esta red se buscó modificar un círculo de tal manera que la resultante de tal modificación creara un patrón en forma de una flor, de igual manera la forma de crear diversidad en los tipos de flores generadas fue mediante el uso de la neuro evolución de topologías aumentativas (NEAT) mediante el cual las redes generadas se alteraban mediante la modificación de conexiones entre neuronas y la adición de nuevos nodos de tal manera que se buscaba un nivel adecuado de complejidad para las redes que generaron diferentes tipos de patrones de flores.

Mientras que un segundo artículo escrito por Erin J. et al.[17] en el cual presentan un nuevo algoritmo de generación de contenido llamado neuro evolución de topologías aumentativas para generación de contenido (cgNEAT) el cual se encarga de generar contenido gráfico y contenido del juego de acuerdo a las preferencias de un jugador mientras el juego está en ejecución, para la evaluación del contenido generado se desarrolló un juego multi-jugador en línea llamado *Galactic Arms Race* en el cual la cgNEAT se encarga de generar contenido para todos los jugadores y ellos eran quienes proveían la retroalimentación del funcionamiento del algoritmo.

2.2.5 Audio

El ámbito de generación de audio en videojuegos se puede considerar como un punto opcional dependiendo del juego que se esté desarrollando sin embargo el audio juega un papel importante en la experiencia que se le brinda a un jugador dentro del juego debido a que mediante el uso de componentes sonoros se pueden influir las emociones que se quieren mostrar en lugares específicos, desde cosas simples como el uso de sonidos de objetos como escuchar un teclado siendo utilizado hasta la utilización de pistas de audio en áreas o momentos clave del juego para demostrar momentos dramáticos, tristes o de acción.

Cabe mencionar que es posible utilizar instrumentos musicales o la implementación de orquestas para de igual manera generar estos ambientes sin embargo algunos utilizan este tipo de generación para crear pistas que sean capaces de adaptarse a los sucesos que transcurren en el momento para el concepto de generación procedural de audio puede ser inclusive el uso de elementos en un entorno virtual que generen algún sonido cuando son interactuados [18], algunos ejemplos de adaptabilidad de audio de acuerdo a los eventos en el momento es el caso del juego *Fire emblem Fates*, este es un juego de estrategia por turnos en donde la adaptabilidad de audio se da en transiciones de vista del mapa y combate de unidades, en este caso el audio de la vista del mapa tiene un cierto nivel de *tempo* mientras que en momento que se entra a un combate entre dos unidades el *tempo* del audio cambia mientras se está en esta escena de combate.

Existen también investigaciones académicas de como combinar la generación de niveles con la generación de audio, tal es el caso de *Sonancia* desarrollado por Phil Lopez et al. [19], en este paper los autores proponen una metodología que permitirá a un sistema generador de niveles seleccionar pistas de audio o sonidos que vallan de acuerdo a un tema específico, en este caso los autores colocaron una lista de pistas y mediante un generador crearon un entorno 2D que simulaba una mansión, el propósito fue el de crear un juego de terror/suspense y que las habitaciones tuvieran un audio diferente y que al acercarse a la división entre una y otra el audio de la habitación adyacente fuese subiendo el volumen conforme más se adentraba y el audio de la habitación anterior se dejará de escuchar

lentamente, el sistema es capaz de generar los niveles y asignar los sonidos necesarios sin embargo tiene la limitante de que se debe de proporcionar el listado de pistas de audio debido a que aún no tiene la capacidad de generar audio por cuenta propia sin embargo es otro buen ejemplo de cómo combinar áreas de generación de contenido.

Otro caso de estudio es el del juego *Audio Surf* desarrollada en 2008 por Fitterer [20], la manera en cómo funciona es que mediante el uso de archivos de audio de alguna canción para generar niveles estilo pista de carreras que el usuario recorre al momento de jugar, como este existen varios otros juegos que utilizan mecánicas de PCG para generar los niveles a jugar por los participantes.

Finalmente tenemos *Mezzo* desarrollado por Daniel B. [21] este artículo muestra un programa de computadora del mismo nombre diseñado con el fin de generar música de la era romántica que puede ser utilizada en juegos de computadora, el software fue desarrollado con el propósito de poder darle más expresividad a eventos que ocurren en juegos de computadora, esto es mediante el uso de los elementos presentes en la escena como las partes que conforman la pieza musical, para esto sus acciones son traducidas como diferentes cantidades de tensión armónica lo cual permite que la pieza musical corresponda al estado del juego así como a las acciones que realizan los personajes.

2.2.6 Narrativa

El ámbito de generación de narrativa en videojuegos se encarga de manera básica de generar historias que tengan sentido y sean entretenidas, el uso de PCG para el ámbito de narrativa se ha tomado de diferentes ángulos, el primero es sencillamente crear la historia del lugar en donde se desarrollan las cosas, esto es generar toda la historia desde un punto específico y así crear una "línea" de acción que un jugador deberá seguir, mientras que otro de los aspectos es la generación de historia que se adapte a las acciones del jugador, esto es, que a medida que el jugador progrese en un juego realizando acciones específicas el juego adapte la historia a generar de acuerdo a dichas acciones.

Algunos ejemplos de este ámbito de generación se muestran en el juego *Façade* y el

CAPÍTULO 2. PRELIMINARES

sistema *Versu*, el primero que se explicara es el juego de Facade, Facade es un juego desarrollado por Michael Mateas y Andrew Stern [22] que intenta crear un juego estilo novela visual en donde las acciones del jugador tengan influencia en los eventos que ocurren en el juego, el juego introduce al personaje del jugador como un amigo de una pareja de casados, no se cuenta con un objetivo específico más que el de completar la historia, sin embargo, el juego provee de acciones y respuestas que el personaje puede expresar a la pareja, estas mismas tienen influencia directamente con el cómo se desarrollará la historia, el juego se diseñó con el propósito de que el jugador lo repita múltiples veces con el fin de descubrir como las diferentes decisiones cambiaron el desenlace de la historia.

Finalmente *Versu* es una aplicación desarrollada por Richard Evans y Emily Short [23] es juego se encarga de generar simulaciones en base a líneas de texto que denotan acciones que un conjunto de personajes no jugables (NPC) realizan en determinados escenarios, el juego permite que a una persona se le asigne un personaje del juego y controle las acciones que realizará, mientras que los NPC restantes responderán de manera autónoma mientras el juego continua, este sistema de generación de narrativa inicio como un proyecto académico y después paso a ser utilizado de manera comercial, además de ser la base para el desarrollo de otros juegos de estilo similar.

2.2.7 Reglas y Mecánicas

Las reglas y mecánicas de un juego proveen un framework en el cual el jugador puede realizar acciones y el cómo debe de influenciarlas hacia un objetivo final, las reglas y mecánicas de un juego pueden variar dependiendo del tipo de juego que se trate, por ejemplo, en juegos de estilo plataforma la regla de terminación puede ser simplemente llegar hasta cierto punto del nivel mientras que las mecánicas pueden incluir saltar, correr o agacharse, este conjunto de reglas permite que el jugador no realice acciones no previstas por los programadores y de tal forma limita a un jugador a cierta cantidad de acciones y consecuentes posibles las cuales deberá de aprovechar de diferentes maneras para continuar.

Mientras que el uso de reglas permite tener un juego más balanceado algunas ocasiones

CAPÍTULO 2. PRELIMINARES

el cambiar el conjunto de reglas permite crear diferentes tipos de juegos, tomando el ejemplo anterior es posible que un juego de plataformas se le permita a un jugador volar en un nivel lo cual provocaría que las mecánicas se requieran acomodar a esta nueva habilidad y de igual manera los niveles de un juego se tengan que re-imaginar de tal modo que ahora se pueda utilizar en ciertas o se coloquen lugares que solo se pueden acceder con esa habilidad por ejemplo la transición de reglas y mecánicas entre los juegos Super Mario Bros. (Nintendo, 1985) [24] [25] y Super Mario World (Nintendo, 1990). [26] [27]

Para la evaluación de la generación de mecánicas y reglas de un juego existen diferentes métodos a utilizar, el primero se basa en el balance de las reglas es decir en caso de ser un juego para dos o más jugadores el objetivo de evaluar sería que todos los jugadores tengan las mismas probabilidades de ganar utilizando las mecánicas del mismo juego, mientras que otra manera de evaluar sería mediante la facilidad que tienen las mecánicas de ser aprendidas por los jugadores.

En cuanto a ejemplos de este tipo de generación se encuentran algunas investigaciones académicas tales como la de Ludi [28], Ludi es un sistema desarrollado por Cameron Browne y Frederic Maire, este sistema se utiliza para evolucionar comandos que definen las reglas de un juego en particular a crear, el algoritmo evolutivo se encarga de evolucionar las reglas manteniéndolas dentro de un cierto rango de métricas que establecen un buen patrón para juegos de tablero tales como tales como la complejidad del juego o la simplicidad de las reglas, un juego de mesa diseñado mediante el uso de este sistema se llama Yavalath, una vista rápida del juego se puede apreciar en la Figura 2.4, este juego puede tener un mínimo de 2 y un máximo de 3 jugadores, las reglas son que en base a turnos cada jugador coloca una ficha de su color en cualquier punto no ocupado del tablero, el objetivo es lograr formar una línea de 4 fichas del mismo color, pero la regla es que al formar una línea de 3 fichas se pierde el juego de manera automática, de esta manera el juego se basa en forzar jugadas del oponente que permitan que alguno tenga más "control" de los eventos que ocurrirán.

Un segundo caso fue escrito por Togelius et al. [29], este uno de los primeros ejemplos del uso de computación evolutiva para la generación de reglas para juegos, en este paper se

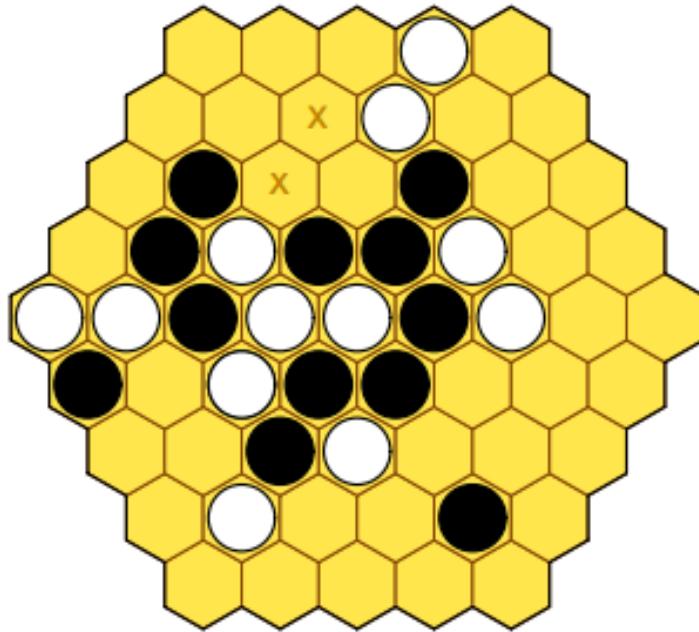


Fig 2.4: Ejemplo de un juego en progreso en el juego de Yavalath

evoluciona un conjunto de reglas de tal manera que un conjunto de agentes que participaran en el juego logren tener un mejor entendimiento de como funciona el juego, la manera de evaluar que tan entendibles son las reglas fue a través de varias sesiones simuladas de los agentes, en este caso se procuró utilizar reglas sencillas que simularan un juego estilo pac-man en donde los agentes deberían de colectar círculos para poder obtener una mejor puntuación.

Finalmente, un último caso es el de Nielsen et al. [30] esos autores programaron un sistema capaz de representar las reglas de un videojuego utilizando video game description language en donde juegos estilo 2d se representan como mapas en base a caracteres y utilizaron ASP para verificar que los niveles podían ser jugados (se podían cumplir las condiciones de victoria).

2.2.8 Juegos

Mientras que las facetas de generación anteriores se han enfocado en aspectos específicos de juegos mientras que en este aspecto en particular se toma un enfoque global, esto es desde la generación de partes visuales hasta la generación de bandas sonoras, esto demuestra como todos los aspectos de un juego se relacionan unos con otros, un ejemplo de esta relación seria tomar un juego en particular y agregar acciones que originalmente no existían, para poder agregar estas acciones se debe de considerar el aspecto visual, es decir el como deberá de representarse visualmente tal acción, aspectos de sonidos en caso de que una acción requiera de algún efecto de sonido que represente tal acción, utilizando el ejemplo de agregar la mecánica de volar, el aspecto visual se representaría en un cambio visible en el personaje controlado, por ejemplo, dibujar alas y que se vea una animación de movimiento al volar, en el caso del aspecto del sonido el aleteo o movimiento generado por dichas alas al estar volando puede tener un sonido único para identificarlo.

Para este caso existen diferentes maneras de intentar generar juegos nuevos, tal es el caso de Game-O-Matic [31], Game-O-Matic es una herramienta de generación enfocada a la generación de juegos que representan ideas, en este sistema se utiliza un sistema de sustantivos unidos mediante verbos que son ingresados al sistema como datos de entrada, estos datos son procesados y se logra generar un juego simple estilo arcade que logre representar un mapa conceptual generado mediante los valores de entrada.

Otra investigación realizada por Nelson et al. [32], para este sistema los autores proponen un sistema parecido al de Game-O-Matic explicado anteriormente, sin embargo, en este caso el sistema está basado en sprites que representan entidades del juego, lo que se asigna como valor de entrada es restricciones que definen la interacción de los sprites en pantalla, las restricciones son procesadas mediante una red semántica y una base de datos de léxico, este proceso logra generar juegos que cumplan las restricciones estipuladas, en este caso los juegos generados tienen mecánicas estilo WarioWare en donde los juegos tienen una regla de victoria sencilla que hace los juegos relativamente cortos.

Finalmente, uno de los mejores generadores de juegos lleva en nombre de ANGELINA

[33]¹, ANGELINA es un sistema de generación de juegos diseñado originalmente en 2011 y que ha recibido modificaciones desde entonces, la base del generador es el uso de un sistema evolutivo que se encarga de evolucionar las reglas, los terrenos, los personajes y demás conjuntos con respecto unos de otros de una manera orquestada, se encarga no solo de evolucionar la información visual de los componentes sino también de asignar sus locaciones en un campo de trabajo que representa un juego, además de esto es capaz de seleccionar elementos musicales relevantes al tema de los mapas generados o de la faceta emocional presente en el juego además de crear asignarle un nombre a los juegos que logra crear, inicialmente el sistema era capaz de generar niveles de plataforma en 2D, pero actualmente es capaz de generar niveles de temática de aventura en entornos 3D.

Los anteriores generadores a pesar de no ser perfectos han permitido dar un gran paso en la generación de contenido o más bien en la generación de juegos, sin embargo aún falta controlar aspectos que permitan que las generaciones de tal contenido logren involucrar todos los aspectos de la generación de contenido, esto podría darse mediante el uso de múltiples agentes que contantemente evalúen los aspectos generados pero que al mismo tiempo logren comunicarse unos con otros y poder evaluar los juegos generados de manera más amplia.

2.3 Jugabilidad

La jugabilidad es un término utilizado principalmente en el área del análisis y diseño de videojuegos y es utilizado principalmente como medida de calidad para la experiencia de un usuario para con el juego mediante las mecánicas o manera en cómo se tiene una interacción jugador-videojuego y el diseño como tal del mismo.

De igual manera diferentes autores definen el concepto como un conjunto de propiedades que definen la experiencia que logra tener un jugador en un juego determinado según al sistema de juego que se provee.

La Figura 2.5 muestra un diagrama de áreas y conceptos que califican el nivel de

¹De acuerdo al autor: "A Novel Game-Evolving Labrat I've Named ANGELINA"

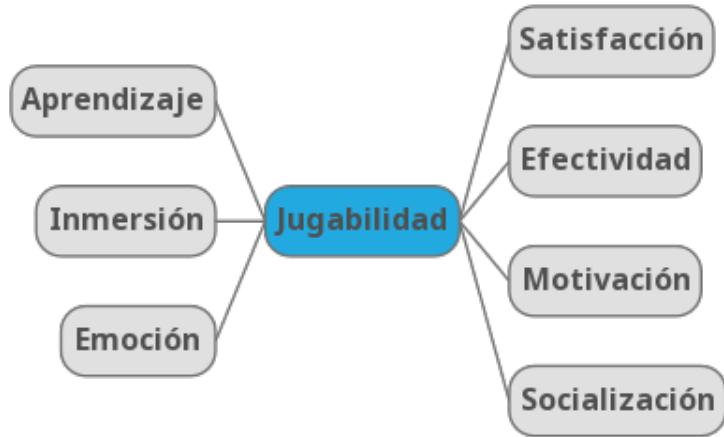


Fig 2.5: Puntos de evaluación de jugabilidad

jugabilidad de una aplicación o juego, estos mismos se pueden definir como sigue:

- Satisfacción: El grado en el que un juego logra agradar o complacer a un usuario.
- Aprendizaje: El grado con el cual un usuario es capaz de comprender las mecánicas de un juego y es capaz de interactuar fácilmente con el mismo.
- Efectividad: Este demuestra la cantidad de recursos y el tiempo utilizados para poder envolver a un usuario y lograr que se divierta.
- Inmersión: El grado con el cual se logra envolver a un usuario con los eventos que transcurren en el juego, es decir que tanto se logra hacer que se integre o muestre empatía por los eventos del juego.
- Motivación: Esta define la capacidad que tiene un juego a motivar a un usuario a continuar con los retos o eventos del juego, esto generalmente se da por mecánicas que dan un sentimiento de recompensa o simplemente por la inmersión lograda.
- Emoción: Es el grado con el cual se logra una inmersión en el usuario de tal manera que es capaz de reaccionar de manera involuntaria a estímulos presentados en

CAPÍTULO 2. PRELIMINARES

el juego, esto puede ser desde una risa por algún evento que ocurre hasta saltos involuntarios por susto en juegos de terror.

- Socialización: Este es el grado con el cual se logra que un usuario establece relaciones con personas que no conoce, en juegos de multijugador masivo en línea ("Massive-Multiplayer Online" - MMO por sus siglas en inglés) existen eventos en los cuales se le pide a los usuarios que construyan equipos para completar niveles en el juego, existen otros juegos de multijugador local en los que varias personas compiten para ganar, y en ambos casos se puede presentar que de los sucesos presentados se puedan crear amistades entre jugadores.[34]

Por otro lado, existe un conjunto de facetas que engloban varios atributos de la jugabilidad que permiten relacionar dichos atributos, para estas facetas se toman los siguientes grupos:

- Jugabilidad intrínseca: Engloba el diseño y mecánicas del juego enfocándose en las reglas y objetivos establecidos, define el cómo se proyectan estos puntos al jugador.
- Jugabilidad mecánica: Se enfoca en parte funcional del juego y en los comportamientos de los personajes, se basa en el diseño del motor del juego.
- Jugabilidad interactiva: Se enfoca más en la relación usuario-juego, tomando en cuenta el cómo se presenta la interfaz de usuario y los controles.
- Jugabilidad artística: En este punto se engloban los puntos de gráficos y sonido, es decir que tan bien es visualmente lo que se presenta y el tipo de ambientación que se crea al combinar efectos visuales, música y gráficos.
- Jugabilidad perceptiva: Este punto trata de cuantificar la manera en como una persona percibe los elementos mostrados en el juego, este es un cálculo subjetivo.
- Jugabilidad interpersonal: Al igual que el punto anterior es un concepto subjetivo, sin embargo, este se enfoca en las percepciones que se generan al jugar en un grupo.

2.4 Open-Ended Evolution

El término Open-Ended Evolution (OEE) es el nombre que se le da a una variante del algoritmo de evolución genética, este algoritmo difiere de los objetivos predefinidos de un algoritmo genético en el cual el objetivo principal de un GA es el de llegar o aproximarse a un valor o conjunto de valores que logren solucionar un problema determinado, en el caso de OEE este no es el caso sino que lo que hace el algoritmo es seguir evolucionando y creando diversidad en la población cada vez más compleja no solo dentro de los límites de evolución de un elemento sino comenzar a crear nuevas "*especies*" de soluciones.

Una explicación más clara es presentada en un paper por T. Taylor et al.[35], en este paper se define OEE como un sistema capaz de novedad después de un punto en la evolución en donde el estado del grupo o población no cambia en lugar de converger hacia un estado casi estable.

La manera en cómo se definirá el uso de OEE que se utilizara en este proyecto es simplemente un proceso de evolución genética capaz de encapsular procesos de evolución similares capaces de incrementar en número según la complejidad a la que las entidades generadas logren llegar.

Capítulo 3

Estado del arte

Este capítulo se enfoca en mostrar los diferentes trabajos que permitirán comprender mejor la problemática que se quiere resolver, así como las diferentes propuestas para poder resolverlo.

3.1 Competencia IEEE CoG

Como se demuestra en el artículo presentado por J. Renz et al.[36], la competencia de Angry Birds utilizando inteligencia artificial se realiza de manera anual desde el año 2012 iniciando con la conferencia dentro del marco de la AAAI (Association for the Advancement of Artificial Intelligence), en este artículo se explica la primera línea de competencia que consta de desarrollar agentes que sean capaces de solucionar niveles del juego de tal manera que se asemeje a la manera en cómo jugaría una persona real, se explica el estado del arte el cual consta de previos competidores y de cómo han logrado realizar sus agentes para el juego, de igual manera se explica la nueva línea de competencia que consta de utilizar el mismo conjunto de niveles y permitir que los agentes compitan para ver cual logra resolver más rápido el conjunto de niveles.

Posteriormente en un artículo elaborado por M. Stephenson et al.[37] se explica detalladamente como es que el juego se conecta en una arquitectura cliente-servidor y como es que se adapta la arquitectura para que los agentes utilizados puedan obtener datos

del juego para poder realizar los cálculos pertinentes, en este mismo artículo se presenta información de la segunda línea de competencia llamada competencia de generación de niveles de AIBIRDS la cual se comenzó a realizar desde el año 2016, en esta versión de la competencia los generadores deben de cumplir con el propósito de crear niveles que sean creativos, divertidos, estables en cuanto a la gravedad del juego y posibles de ser solucionados, además de esto se debe de tener en consideración realizar niveles que sean desafiantes a los jugadores, como el juego no es open-source los niveles se generan en una versión clon del juego que cuenta con las mismas mecánicas.

Un segundo artículo escrito por M. Stephenson et al.[38] se presenta una versión más actualizada de las reglas de la competencia, en este artículo se establece el uso de un archivo proporcionado por los organizadores, este archivo cuenta con 4 líneas en las cuales se establece las condiciones que los niveles generados deberán de cumplir, la información proporciona en el archivo es la cantidad de niveles que se deberán de generar, las piezas no permitidas en un nivel, así como las combinaciones que se deberán de evitar, la cantidad de puercos a colocar en un nivel y por último el tiempo límite para terminar los niveles, en el caso de los tipos de materiales prohibidos se entrega una lista que especifica que piezas con que material no se puede colocar, en el caso de los puercos se entrega un rango numérico en el cual se especifica la cantidad mínima y máxima a colocar, en cuanto a los valores de niveles y tiempo límite se entregan como valores numéricos enteros, sin embargo los valores que se entregan generalmente son 5 para la cantidad de niveles y 30 en la cantidad de minutos que puede durar el generador.

3.2 Algoritmos de búsqueda

L. Ferreira et al.[39] presentaron un sistema de generación de niveles mediante el uso de algoritmos genéticos, en el sistema se propone la definición de un individuo como una lista de elementos que se acomodaran en forma de torre, el sistema propuesto utiliza las piezas base del juego y se agregaron 4 elementos compuestos, los niveles se definen como un área de trabajo con tres posiciones donde las torres podrán ser generadas a base de las piezas

CAPÍTULO 3. ESTADO DEL ARTE

en la lista principal, las piezas irregulares tales como los dos diferentes tipos de triángulos son tomados en cuenta únicamente en la última posición de las torres.

Una herramienta de generación de niveles basado en el uso de patrones de generación es propuesto por Y. Jiang et al.[40], en este trabajo los autores proponen el uso de patrones predefinidos de generación de estructuras basadas en el alfabeto americano, valores numéricos, símbolos, así como el ordenamiento de estos en base a patrones de frases o palabras predefinidas que pueden ser utilizadas, utilizando estos patrones se generan niveles en donde se muestran frases divertidas o motivacionales que serían jugados por personas para evaluarlos, debido a la manera de acomodo de bloques utilizada en este paper el problema de que los niveles sean estables se eliminaba y simplemente se utiliza el sistema para cumplir los dos objetivos restantes los cuales son que los niveles sean jugables y entretenidos, mientras que la manera final de evaluar la funcionalidad del generador se basaba la re-jugabilidad, la legibilidad del texto así como de la dificultad de los niveles, para esto los autores se apoyaron de 10 personas con cierto grado de comprensión del idioma inglés, de esta manera en caso de que los textos fueran difíciles de comprender, la dificultad fuera demasiado alta o generara un interés por repetir el nivel los participantes serían los encargados de proporcionar estos resultados.

Para la adaptabilidad de los niveles basados en los resultados de la función de aptitud M. Kaidan et al.[41] proponen una medición basada en la habilidad de un jugador en un nivel, la evaluación de la habilidad está dada por el puntaje obtenido durante el juego, en el juego de Angry Birds el puntaje de un jugador está ligado a la cantidad de destrucción de estructuras lograda y la cantidad de aves que no se utilizaron para completar el nivel, en este caso el paper se apoyaba de la función de aptitud de L. Ferreira et al.[39] para la distribución de los puercos en un nivel y se adaptaba la función para modificar la cantidad de puercos colocados de acuerdo a la facilidad con la que se lograba resolver el nivel, de esta manera el sistema generaba un conjunto de niveles para que una persona los solucionara y basándose en los resultados de los niveles se generaba el siguiente conjunto para resolver, de tal manera que el sistema requería que una persona evaluara los niveles cada generación para que el sistema generara niveles más desafiantes para esa persona.

Otro generador propone el uso de un selector de dificultad como parámetro de entrada de un algoritmo genético para la generación de niveles con elementos predefinidos, en este paper M. Kaidan et al.[42] utilizan una versión extendida del Algebra de Intervalo Temporal de Allen (Allen's Temporal Interval Algebra)[43] para calcular área mínima delimitadora (Minimum Bounding Rectangle) de cada elemento del juego para realizar cálculos de posición más exactos al momento de obtener resultados, utilizando estas mecánicas proponen evaluar la relación de movimiento después de que un ave a impactado en alguna de las piezas del nivel.

Otros investigadores proponen utilizar sistemas de generación que no incluyan limitantes en las estructuras a fin de permitir una evolución general mas "libre" tal es el caso del paper desarrollado por L. Calle et al. [44] en donde utilizan un sistema de generación que ignora la necesidad de crear estructuras siguiendo patrones o simetría, en este paper se busca utilizar un espacio de búsqueda relativamente pequeño para ahorrar tiempo de procesamiento, además de esto utilizan una aplicación de código libre llamada Box2D (<https://box2d.org>) sobre la cual está basado el código del juego de Angry Birds y permite realizar simulaciones de manera más rápida debido a que solo se calcula la estabilidad de las estructuras para el valor de fitness de los individuos, sin embargo, debido que el sistema de generación no está encaminado por patrones las estructuras finales generadas carecen de llamatividad visual, cabe mencionar que este trabajo se utilizó como base para el desarrollo del proyecto actual, algunas de las ideas tuvieron origen en este trabajo para ser modificadas y adaptadas en nuestra propuesta.

3.3 Agentes para jugar

De igual manera varios autores toman la ruta de programación de agentes que sean capaces de solucionar los niveles presentados en el juego, de estos se hacen mención algunos.

Dentro de la competencia de diseño de agentes el objetivo es que el agente sea capaz de solucionar los niveles utilizando las mecánicas de gravedad y colisión del juego, por tal manera F. Calimeri et al. [45] propones el uso de una forma de programación declarativa

CAPÍTULO 3. ESTADO DEL ARTE

llamada Answer Set Programming (ASP por sus siglas en inglés) dicha mecánica está orientada hacia algoritmos de búsqueda difíciles, en este caso utilizan ASP para analizar el campo de juego que involucra los bloques utilizados en las estructuras y los puercos estacionados en ellas, de esta manera ASP tratará de identificar el daño que puede ser causado, así como el movimiento resultante de los bloques en base a la gravedad del juego al momento de disparar un ave en un cierto angulo con determinada fuerza, en base a esto se determina el siguiente tiro a realizar para poder completar el nivel.

Diferentes autores han utilizado diferentes técnicas para la generación de contenido, algunas utilizando computación evolutiva y algunas otras simplemente a base de algoritmos de búsqueda, de estos mismos los más relevantes se les hace mención: G. Smith et al.[46] proponen un software de generación de niveles estilo juego de plataformas mediante el uso de un software aplicando inteligencia artificial, en este software se le permite a un usuario seleccionar la altura del punto de inicio y punto final de un nivel en particular y el sistema trata de llenar el espacio restante con plataformas que permitan que el nivel se pueda completar, el sistema se basa en una medición de pulsos en los cuales una persona podría realizar una acción determinada para avanzar a la siguiente sección.

Capítulo 4

Propuesta del proyecto

Este capítulo describe los componentes que integran el método propuesto, así como la manera en cómo estos componentes proveen de apoyo al sistema final, este capítulo está dividido en dos secciones la primera siendo la sección 4.1 en donde se explican los conceptos propuestos y utilizados en el proyecto, esta sección se divide en varias subsecciones en donde se explican partes de la propuesta y el cómo permitirán que el sistema generado cumpla con los objetivos requeridos, la segunda sección 4.2 explica ideas y conceptos que se propusieron a lo largo del desarrollo y el cómo estos ayudarían a mejorar el sistema generado.

4.1 Propuesta utilizada

La propuesta final utilizada en el proyecto está basada en el uso de varias técnicas para la generación de los compuestos y subsecuentemente los niveles para el juego de Angry Birds, la base de todo el proyecto está en el uso de un algoritmo genético para evolucionar los individuos, esto se explica más a detalle en la sección 4.1.3



Fig 4.1: Piezas básicas del juego angry birds

4.1.1 Generar compuestos de piezas base

El videojuego de Angry Birds cuenta con un total de 11 diferentes tipos de bloques, además de 2 grupos de objetos siendo estos cajas de explosivos y los puercos que pueden ser colocados en el nivel, además de esto cuenta con plataformas que pueden ser agregadas y rotadas para establecer plataformas en donde colocar más bloques, de estos solo utilizaremos las 11 piezas básicas mostradas en la Figura 4.1 y los puercos para generar niveles.

Las piezas básicas no pueden ser modificadas, tampoco es posible agregar nuevos bloques al código debido a que la competencia solo utiliza las piezas regulares, para poder librarse de esta restricción se generarán compuestos con las piezas base, estos compuestos pueden estar formados de 1, 2 o más bloques básicos, estos grupos se utilizarán para generar las estructuras que se colocaran en los niveles.

La manera propuesta para generar estos conjuntos es mediante el uso de los valores de alto y ancho de las piezas para poder definir los bordes de las mismas, de esta manera al momento de querer unir dos o más piezas para formar un conjunto la medida del conjunto se calculará buscando el punto más alto, más bajo, así como las posiciones más a la derecha e izquierda teniendo ambas piezas unidas, un ejemplo de la unión de cuatro piezas se muestra en la Figura 4.2 en donde cuatro piezas se colocan una sobre otra para formar un objeto cuadrado. Para poder agregar las piezas como un conjunto se buscan los puntos más alejados, una vez que se ha calculado el alto y ancho del conjunto se procede a encontrar el punto central del mismo, desde este punto se calcula la distancia en x y y hacia los centros de cada una de las piezas del conjunto, estos valores de centros se

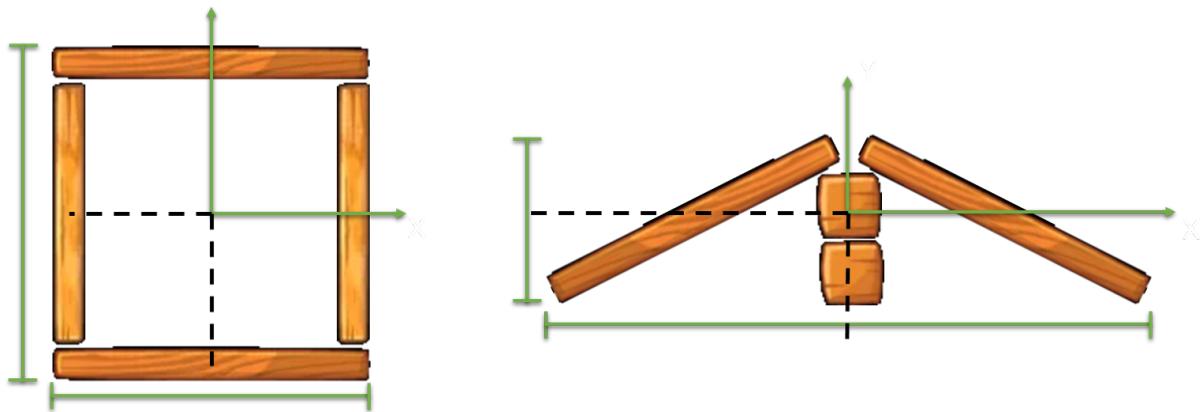


Fig 4.2: Cálculo de bordes de un conjunto

agregan utilizando una estructura de diccionario para poder generar los objetos requeridos al momento de crear a los individuos antes de iniciar las generaciones.

Debido a que los conjuntos son creados antes de la ejecución del algoritmo se permite que el algoritmo se base únicamente manteniendo los apuntadores a las composiciones de piezas y solo trabaja con estas mediante las operaciones genéticas del propio algoritmo, esto permitirá que el algoritmo utilice menos tiempo identificando posibles composiciones.

Utilizando esta manera de generar los compuestos se utiliza un algoritmo orientado a objetos, para poder acomodar los elementos y crear los objetos requeridos según sean necesarios, esto se explica más detalladamente en la sección 4.1.2.

4.1.2 Generación de compuestos mediante objetos de clase

Para mantener un control de los compuestos, principalmente de las piezas individuales, se propuso la una jerarquía de clases con herencia, en donde una clase base contendrá los métodos que las clases derivadas para obtener valores específicos cuando las piezas requieran, principalmente estos métodos se encargan de calcular las esquinas de las piezas particulares las cuales permitirán una vez se tenga un conjunto encontrar la altura y tamaño total del conjunto, así como el centro del mismo.

Utilizando la estrategia mostrada en la Figura 4.2 y explicada en el capítulo anterior

CAPÍTULO 4. PROPUESTA DEL PROYECTO

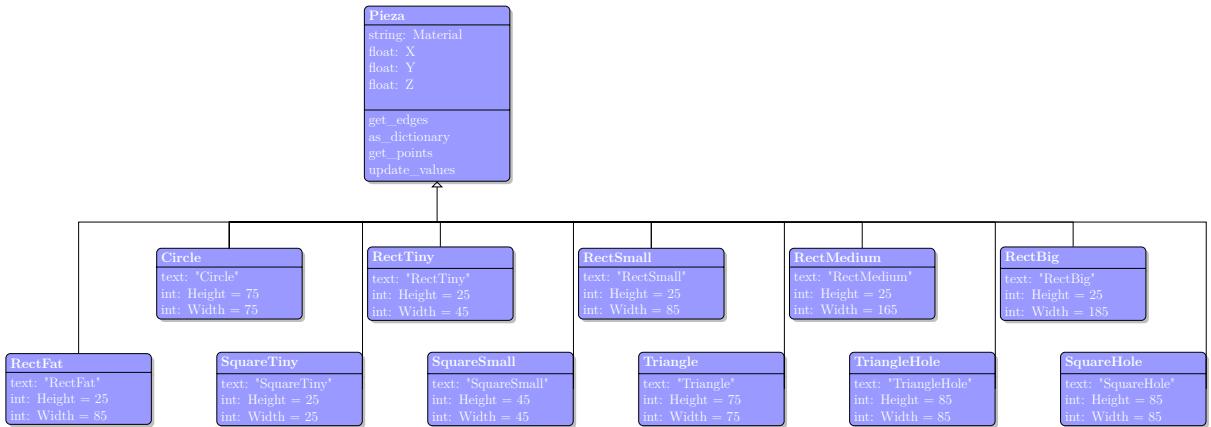


Fig 4.3: Diagrama de clase de piezas

se define una clase para los conjuntos de piezas, aquí se utilizan los cálculos de las clases particulares anteriormente mencionadas para obtener los valores de bordes y tamaño de los conjuntos, estas clases se crean de acuerdo a los apuntadores indicados en los individuos de la población, cuando un apuntador hace referencia a un grupo se obtienen los datos de las piezas que integran al conjunto y se crean objetos nuevos de esas mismas clases que pertenecerán a un individuo particular, esto debido a que al hacer referencia a un objeto previamente creado un cambio realizado en un individuo particular se propagaría a todos los individuos que utilicen ese mismo apuntador, por tal motivo cuando se crean los conjuntos se mantiene una lista de valores que indican la clase que se deberá de crear y la posición en x y y relativa al centro del mismo conjunto.

El sistema de clases se estableció con el fin de reducir la redundancia de código y para permitir que cada clase particular mantenga los datos necesarios para crear los objetos que contiene, así como mantener las listas de las piezas que conforman los niveles antes y después de haber realizado las simulaciones para poder realizar los cálculos de manera más rápida además de las clases que controlan los conjuntos y subsecuentemente las piezas individuales.

Estas clases se desarrollaron utilizando herencia y polimorfismo, cómo se muestra en la Figura 4.3. Las once clases de las piezas básicas heredan los métodos de la clase principal, esto es cómo se menciona anteriormente por que las acciones que las clases particulares

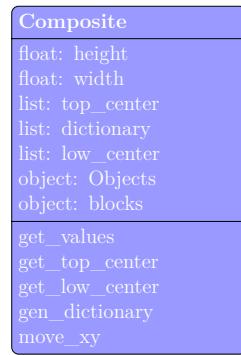


Fig 4.4: Diagrama de clase de Composite

harán serán las mismas sin modificar nada en los métodos, sin embargo, cada una activa un constructor con los datos de tamaño y nombre específicos, estos datos son utilizados para las mediciones y para integrar cada pieza particular como una lista de texto que se utilizará para generar los archivos de los niveles.

De igual manera se utiliza una clase específica para controlar la generación de los conjuntos, la manera en cómo funciona esta clase es que se entrega una lista al generador de compuestos, esta lista contiene una o más piezas que conformaran el conjunto particular, la manera en cómo se entregan estas listas es cada línea de la lista contiene la información necesaria para crear los objetos de clase de las piezas requeridas, estos elementos se utilizan con las clases previamente descritas para generar un conjunto específico, además de eso esta clase se encarga de generar las listas y calcular los valores finales de los conjuntos, siendo estos valores, el punto más alto al centro del conjunto, la altura, el ancho y las esquinas del mismo, así como un método que permite crear, apoyándose de las clases particulares obtiene los valores que definen cada elemento del conjunto, siendo el nombre, material del que esta hecho y los offsets o distancias desde el centro del conjunto al centro de cada pieza particular.

Finalmente, debido a que se requiere controlar una gran cantidad de datos necesarios para el algoritmo genético por cada individuo se utiliza también una clase extra para ellos, esta clase permite que cada individuo mantenga los resultados de fitness después de las simulaciones, así como también controlar el número y listado de las piezas que conformaran



Fig 4.5: Diagrama de clase de Individuo

cada nivel que representa el individuo, además de esto la clase permite que los individuos generen los archivos necesarios de los niveles para ser evaluados, el modelo básico de esta clase se muestra en la Figura 4.4 en donde se puede ver los elementos que utiliza la clase como variables principales así como los métodos que es capaz de ejecutar una vez iniciado el algoritmo.

Mediante el uso de estas clases se controla la generación de individuos de tal manera que un cambio realizado en la clase principal de un individuo deberá de propagarse hasta la base del mismo siendo estos el controlador de conjuntos y las piezas individuales, de esta manera las modificaciones realizadas por las operaciones genéticas de igual manera se deberán de propagar por todo el individuo realizando las actualizaciones y modificaciones necesarias en las posiciones y tipos de piezas utilizadas. Cómo se ha mencionado a lo largo de esta sección el uso de clases para controlar los aspectos básicos de los individuos resultó de gran utilidad, una de las maneras anteriores de controlar este aspecto fue mediante el uso individual de métodos para realizar las modificaciones de los individuos de manera más detallada sin embargo el sistema de clases permitía que muchos de los métodos se fusionaran de tal manera que todos los individuos, compuestos y piezas particulares cuentan con los métodos que utilizarán desde el inicio en lugar de estar llamando los mismos mandando la información de cada individuo de manera particular, sino que al estar integrados en los individuos es posible simplemente llamar los parámetros de clase según sean requeridos,

la manera que se tenía anteriormente de utilizar métodos particulares para la generación, modificación y evolución de los individuos se explica más adelante en la sección 4.2 junto con otras ideas que no fueron utilizadas o fueron modificadas.

4.1.3 Generar niveles utilizando algoritmos genéticos

Utilizando las ideas anteriormente presentadas, se llegó a la definición del algoritmo genético, se decidió que deberá de ser lo que se utilizaría para representar a los individuos de la población dentro del algoritmo, así como el que sería lo que evolucionaría y evaluaría, mientras que el sistema de clases permite tener un control de que es lo que englobará a un individuo el algoritmo genético es el que a final de cuentas se encargará de realizar la evolución necesaria mediante los parámetros propuestos, para esto primero se decidió que sería lo que se alimentaría en el algoritmo para realizar la evaluación de los individuos, para este valor o variable se decidió que lo que se alimentaría sería el listado de posiciones de piezas en el individuo, esto es primero se ejecutaría una instancia del juego con los individuos existentes al momento y se evaluaría la posición final de las piezas contra la misma posición de las piezas antes de entrar a la simulación de esta manera lo el valor de fitness de los individuos sería el total de piezas restantes en el nivel, así como el que tan bien logran conservar sus posiciones originales dentro de la simulación, esto es debido a que se busca crear niveles que logren conservar su forma sin ser manipuladas por el jugador, de igual manera se decidió que sería lo que se utilizaría para evolucionar a los individuos, en este aspecto se decidió utilizar una lista de valores enteros que representan a los individuos, esta lista se explica más a detalle en la sección 5 esta lista se utilizará dentro del algoritmo genético para realizar las operaciones genéticas necesarias para crear a los nuevos individuos de la población.

El diagrama del sistema se muestra en la Figura 4.6, en este diagrama se muestra el proceso por el cual pasa el algoritmo genético, desde el inicio que utiliza un archivo con los parámetros de configuración necesarios para inicializar el sistema, así como el proceso por el cual pasan los individuos para su evaluación, de igual manera los individuos pasan por un

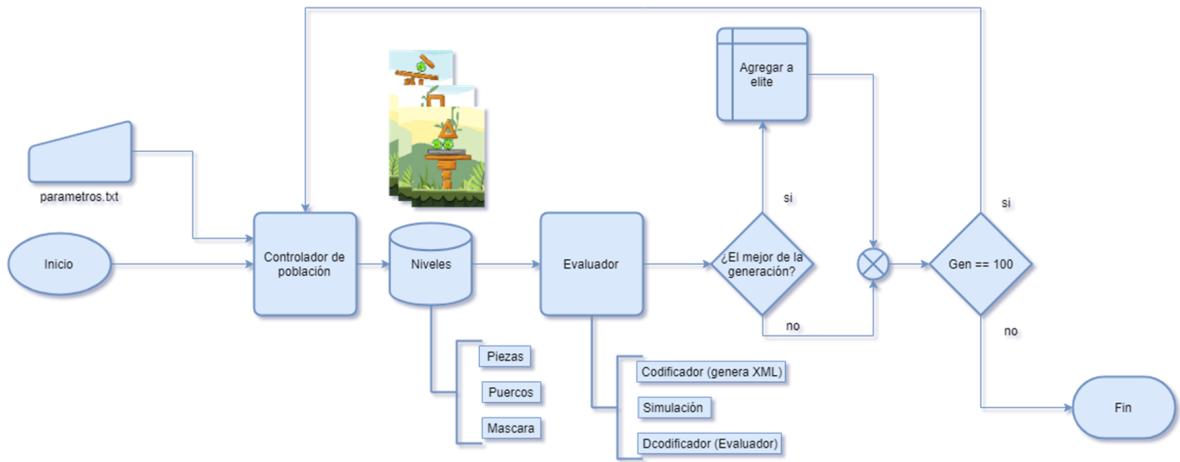


Fig 4.6: Diagrama de flujo del algoritmo

proceso de selección en el cual los mejores o en este caso el mejoro individuo es seleccionado para entrar a un grupo especial con el fin de ser integrado de vuelta a la población en generaciones subsecuentes para encaminar al algoritmo a obtener el mejor valor de fitness posible, de igual manera este proceso se explica más a detalle en el capítulo siguiente en donde se muestra cómo los conceptos vistos en este capítulo fueron implementados para generación del sistema.

4.2 Propuestas anteriores

Dentro del aspecto de generar compuestos nuevos para ser utilizados, previamente se propuso utilizar los resultados de un nivel como compuestos para las siguientes generaciones, un ejemplo de esto se muestra en la Figura 4.7 en donde una estructura generada mediante el algoritmo genético se simula en el juego y después de varios segundos el resultante del mismo podía ser utilizado como un compuesto para la siguiente generación, este método proveería una manera de tener compuestos verificados como viables, sin embargo, se optó por utilizar el generador de compuestos antes de entrar al ciclo del algoritmo genético.

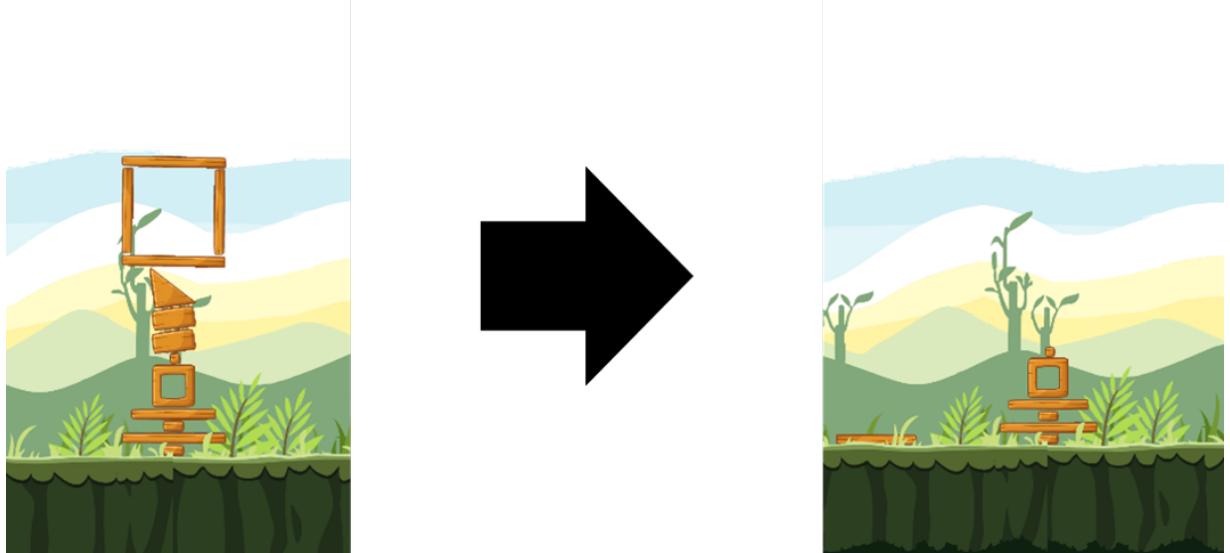


Fig 4.7: Ejemplo de una estructura antes(izquierda) y después(derecha) de una simulación

4.2.1 Propuesta orientada a métodos

Una de las maneras en las que se propuso originalmente el desarrollo del proyecto fue mediante el uso de diccionarios en Python, esto permitiría tener un control de los compuestos generados debido a que mediante el uso de diccionarios se podía generar apuntadores que permitirían que al momento de generar los cromosomas de los individuos de la población estos utilizarán simplemente listas numéricas que hicieran referencia al compuesto al cual pertenecían, esto al final permitiría generar un solo diccionario que contuviera todos los apuntadores a los compuestos generados.

Esto sin embargo presentó un problema para la generación de nuevos compuestos debido a que en caso de querer realizar pruebas restringiendo el uso de ciertas piezas o ciertas combinaciones de material-pieza se tenía que cambiar cada elemento en el diccionario generado, además de esto se requería que las piezas pudieran no solo restringirse, sino que se pudieran realizar cambios en las maneras de cómo se generaban de manera sencilla, por tal se descartó esta propuesta con el fin de utilizar la propuesta de generación de elementos mediante el uso de clases cómo se explicó en el capítulo 4.1.2.

Además de esto otro de los problemas que se presentaban mediante el uso de métodos

para la generación de los cromosomas fue que los métodos se asignaban de manera directa a los individuos lo cual provocaba que simplemente se asignaran diccionarios que especifican el conjunto de métodos que se deberían de llamar, sin embargo, al no tener los beneficios de un sistema de clases que puede ser re-instanciado en caso de ser requerido se tiene el problema de que cada que se realiza una modificación en los métodos, los cambios se propagan a los demás individuos provocando errores en los niveles generados.

4.2.2 Regla de tercios

Uno de los temas que más interesaba en la generación de niveles fue la manera en cómo se acomodarían las estructuras en al área del juego, debido a que lo que se buscó en el proyecto originalmente fue crear estructuras que tuvieran una gran altura y lograran mantenerse estables se buscaron maneras de lograr que el posicionamiento de las piezas o compuestos fuese lo más '*'controlado'*' posible debido a que simplemente colocar piezas al azar generaría demasiada incongruencia en los niveles generados, por tal motivo una de las primeras propuestas fue el uso de la *regla de tercios*.

La regla de tercios es una técnica utilizada en el área de fotografía cuyo propósito es crear imágenes estéticamente agradables a la vista en donde el punto focal o punto de interés se encuentra colocado en la alguna de las áreas de la imagen, un ejemplo de esto se puede apreciar en la Figura 4.8 en donde el objeto principal que se quiere resaltar se trata de colocar en el área central de la imagen completa.

La regla de tercios hace uso de líneas imaginarias que dividen la imagen en nueve áreas, y mediante el uso de la cuadricula generada se colocan los objetos principales de una imagen entre líneas divisorias de tal manera que en caso de imágenes grandes donde existen dos o más elementos importantes un solo objeto no tome el centro absoluto de la imagen, sino que se mantenga alineado a un punto donde las líneas se cruzan para que los demás elementos de la imagen tengan un mismo nivel de interés y en el caso de que solo sea un elemento no utilice toda la parte central de la imagen sino que exista un nivel de balance en la imagen en donde el cielo o el fondo tenga una tercera parte del espacio de

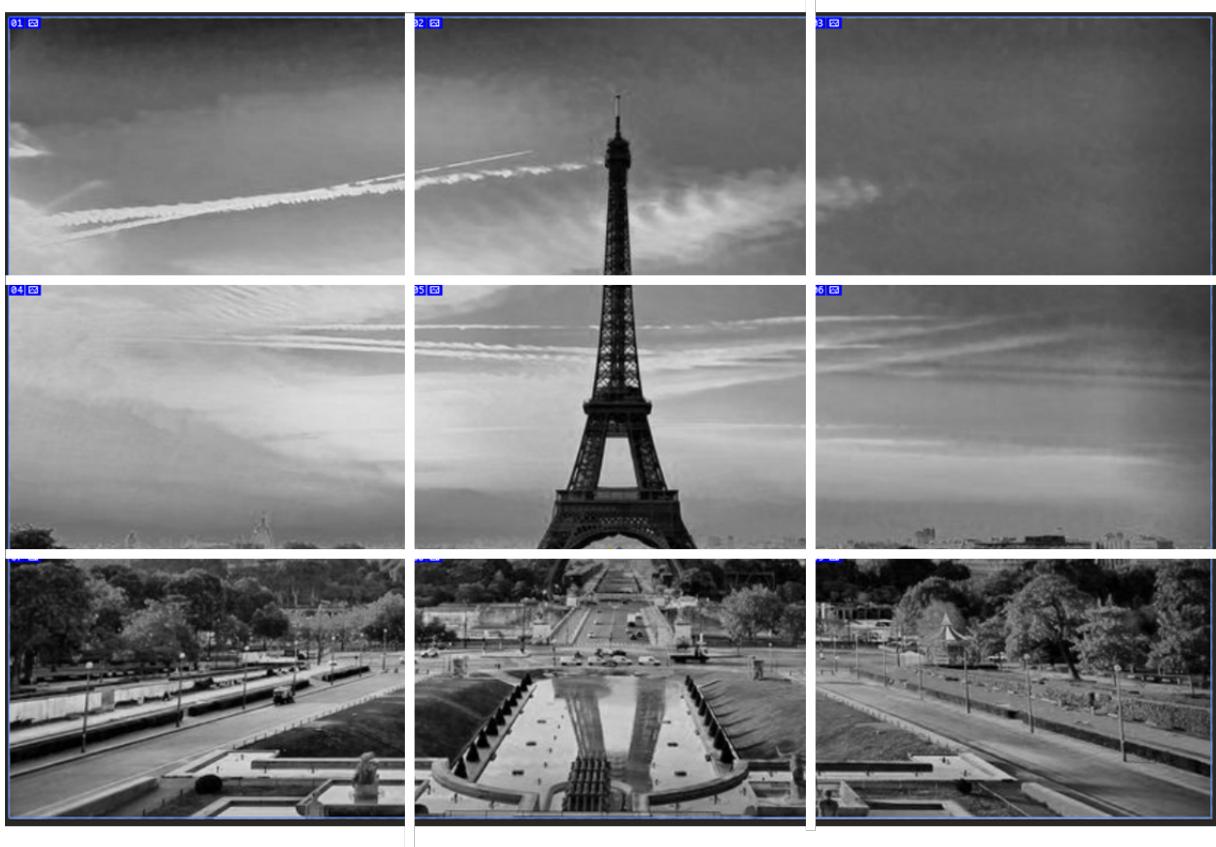


Fig 4.8: Ejemplo del uso de la regla de tercios en una imagen, el punto de interés se encuentra en la parte central

la fotografía, mientras que el área activa o el área inmediatamente adyacente al sujeto de la imagen cubra el resto de la misma.

La manera en cómo se planeó utilizar esta técnica es mediante el uso de una cuadricula de tres por tres de la misma manera que se me muestra en la Figura 4.8, pero en el caso de los niveles generados se espera utilizar la cuadricula para llenar áreas del nivel iniciando desde la parte inferior para generar niveles más robustos en sentido de que la distribución de las piezas es más enfocada al centro creando una pirámide o llenando todas las áreas dependiendo de la cantidad de piezas utilizadas.

Debido a que los niveles generados tienen un área de visión o área jugable variante hasta cierto punto, se definirá un área de juego estática para los niveles de tal forma que todos siempre se generen con las mismas dimensiones, de esta manera se evita el tener que

CAPÍTULO 4. PROPUESTA DEL PROYECTO

estar calculando áreas para los nueve cuadrantes utilizando la regla de tercios y se utilizan siempre los mismos valores para distribuir las piezas en los niveles.

El uso de la regla de tercios dentro del proyecto fue mediante la utilización de máscaras de generación, estas máscaras se generaron mediante la modificación de la idea detrás de la regla de tercios, esto se utilizó de igual manera una cuadricula que cubriera el área de juego, la cuadricula se modificó para cubriera el área con un tamaño de 3 cuadros de altura y 7 cuadros de largo, de esta manera las piezas tendrían un poco más de libertad sobre el dónde sería posible colocarlas, el ejemplo básico de la idea detrás del uso de la regla de tercios se muestra en la Figura 4.9 en donde se utiliza una máscara simple para acomodar los elementos en un individuo de tal manera que el ordenamiento dentro de la cuadricula permitiría la generación de estructuras más complejas, sin embargo esta idea a pesar de proveer una buena mecánica en el acomodo de las piezas conllevaba un error debido que no solo se debería de buscar los puntos en los cuales los elementos se podrían apoyar entre ellos, si no que era requerido tomar en cuenta las posiciones, alturas, tamaños y ángulos de todos los elementos presentes en el cromosoma una vez colocados en el nivel, los resultados obtenidos utilizando la regla de tercios se muestran en la Figura 4.10, en esta imagen se muestra cómo una máscara pre-generada con una forma de castillo mostrada del lado izquierdo de la imagen se combina con el cromosoma entrante de un individuo para generar el nivel mostrado del lado derecho de la imagen, este nivel se trata de asemejar a lo establecido en la máscara, mientras que el uso de máscaras pre-generadas permite encaminar el sistema de generación a un conjunto de resultados de igual manera inhibe que se logre tener una diversidad de niveles debido a que siempre se mantendrán dentro de las formas especificadas por las máscaras, debido a esto se optó por modificar la idea para permitir que los individuos tengan una máscara no pre-generada, sino que de manera pseudoaleatoria se les asigna una máscara que contendrá de igual manera varias columnas en donde el total de piezas se reparte de manera aleatoria, esto permitirá tener más diversidad en los niveles generados.

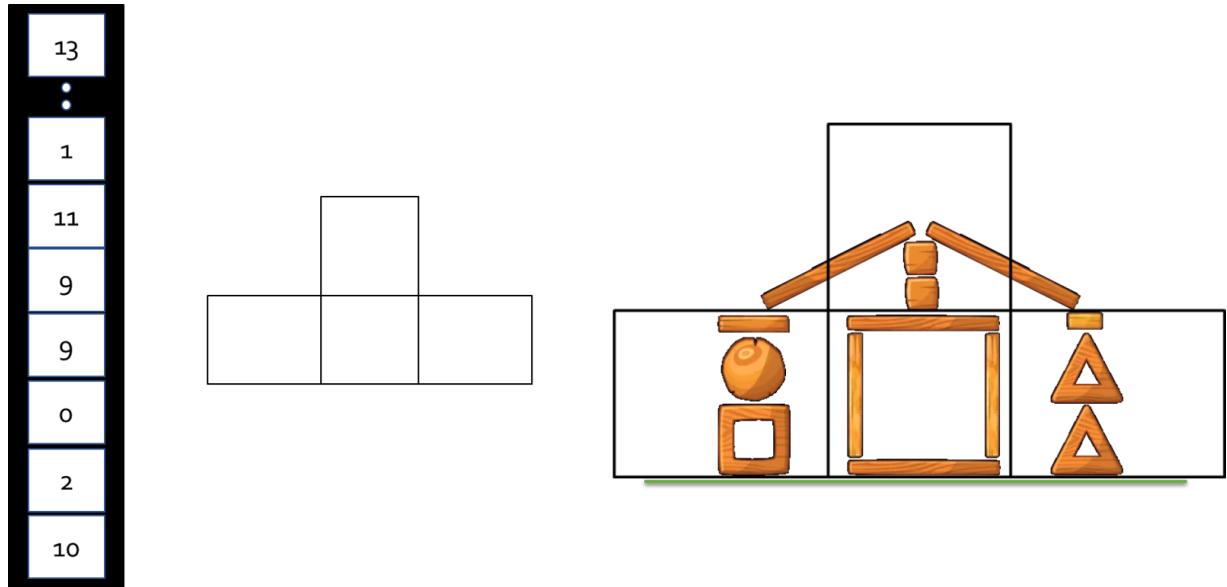


Fig 4.9: Ejemplo del uso de la regla de tercios en un conjunto de piezas, el cromosoma de un individuo(izquierda) se combina con la máscara(centro) para generar una estructura más compleja(derecha)

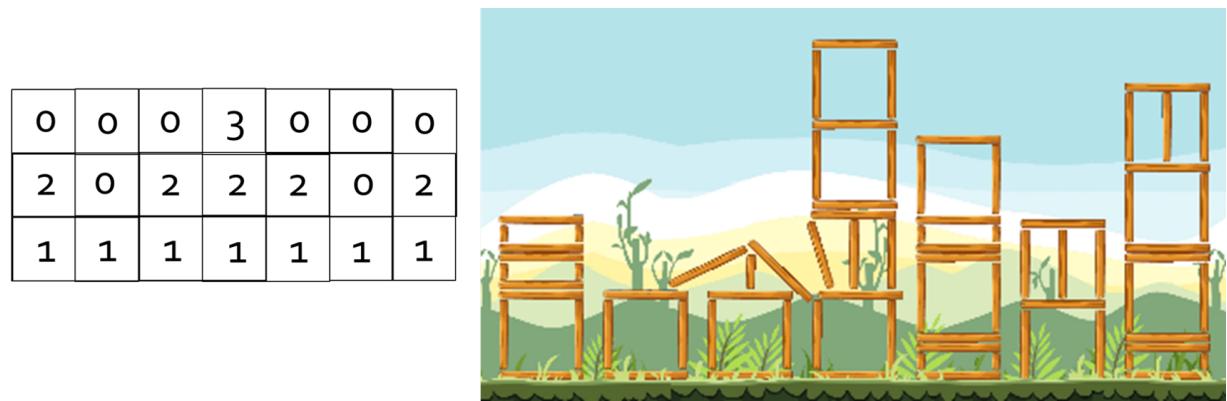


Fig 4.10: Resultado obtenido utilizando la regla de tercios, la máscara(izquierda) utilizada y el resultado generado(derecha)

Capítulo 5

Implementación

El siguiente conjunto de secciones describen la implementación del método propuesto según las explicaciones presentadas en el Capítulo 4. La implementación propuesta utiliza el software de simulación mostrado en [47] (<https://aibirds.org/other-events/level-generation-competition/basic-instructions.html>), el cual está programado en Unity, con código fuente en lenguaje C#.

5.1 Codificando los diccionarios de piezas

Para poder integrar el listado de piezas al sistema de generación se decidió la manera en la que se establecerían las diferentes piezas, como las mostradas en la Figura 4.1 del Capítulo 4, para esto se tomaron en cuenta varias posibilidades, la primera, cómo se explica en la sección 4.2.1 se buscaba ordenar las diferentes piezas como diccionarios que contuvieran una o más referencias a las piezas que integraban cada entrada del diccionario, para esto las primeras 11 posiciones eran elementos individuales, uno para cada pieza como el mostrado en el código 1, en el código se aprecia la manera en la que se designaba una pieza del juego, para poder trabajar de manera dinámica y agregar más compuestos. Según fuese necesario se utilizaban dos diccionarios extra, estos contenían los datos de los tipos de materiales y la información de los tamaños de las piezas, de esta manera en caso de tener más de una pieza en algún punto del diccionario era posible modificar el valor de offset de

Código 1 Ejemplo de diccionario con un solo elemento

```

1   BLOCKS = {
2       0: [
3           {'type': BLOCK_TYPE['Circle'],
4            'material': BLOCK_MATERIAL['wood'],
5            'offset': [0, 0, 0] # x, y, z - Calculated from the center of the figure
6        }]
7   }
8
9   BLOCK_TYPE = {
10      'Circle': {'height': 72, 'length': 72},
11      'RectTiny': {'height': 22, 'length': 42},
12      'RectSmall': {'height': 22, 'length': 42},
13      'RectBig': {'height': 22, 'length': 42},
14      'RectMedium': {'height': 22, 'length': 42},
15      'RectFat': {'height': 42, 'length': 82},
16      'SquareTiny': {'height': 22, 'length': 22},
17      'SquareSmall': {'height': 42, 'length': 42},
18      'Triangle': {'height': 72, 'length': 72},
19      'TriangleHole': {'height': 82, 'length': 82},
20      'SquareHole': {'height': 82, 'length': 82}
21    }
22
23   BLOCK_MATERIAL = {
24       'wood': 0,
25       'stone': 1,
26       'ice': 2
27     }

```

cada una de las piezas calculando la distancia del punto central de cada pieza al centro de todo el conjunto.

De esta manera, era más sencillo crear compuestos, sin embargo, la desventaja de utilizar este método era que no se podían realizar modificaciones a las restricciones de las piezas, por ejempl, para el caso de querer evitar que un conjunto de piezas con determinados materiales no sean generados.

En luz de esta información se optó por utilizar el método orientado a objetos mencionado en la sección 4.1.2, el código 2 muestra la nueva estructura utilizada según el

diagrama de clases presentado en la Figura 4.3, una clase base con las operaciones y propiedades necesarias funcionaría como una clase base para las clases de piezas particulares, de esta manera los individuos se representan como un conjunto de referencias a las clases que deben de ser generadas para la composición de niveles, sin embargo, está manera de representar a los individuos no contempla las restricciones que pueden ser definidas para la combinación de piezas-materiales, debido a esto se optó por generar una lista global que mantuviera un récord de las restricciones aplicadas en el sistema, y al momento de generar cada re-instanciacion de clase para los individuos esta información se pasará a las clases según fuese requerido, en el código se presenta una variable con el nombre *mat*, esta variable tomará una lista de 3 valores que define si alguno o algunos de los materiales no puede ser utilizado al momento de asignar las clases a los individuos, al momento de encontrar una pieza que de la coincidencia no puede ser utilizada con ningún material se eliminarán los datos de la pieza generada y se pedirá al sistema genere otra de manera aleatoria.

Código 2 Ejemplo de estructura de las clases hija que heredan de la principal

```

1   class Circle(Piece):
2       def __init__(self, mat, x=0, y=0, r=0):
3           self.Name = "Circle"
4           self.Height = 75
5           self.Width = 75
6           Piece.__init__(self, x, y, r, mat)
7           self.update_values()

```

5.2 Creación de compuestos

Una vez definidas las clases que controlarán la información de las piezas en el algoritmo se procede a definir la manera en la que se entregarán y controlarán los diferentes compuestos de piezas, para términos más simples los compuestos son grupos de una o más piezas y se mostrarán a manera de diccionario de listas cómo se muestra en el código

3, de esta manera cada entrada en el diccionario tiene por nombre o apuntador el valor posición según se fueron agregando al diccionario, mientras que el contenido es una lista con tuplas donde se estipula la información pertinente de cada pieza en el compuesto. Un ejemplo claro es el elemento en la posición 9, el cual cuenta con 3 piezas que muestran los valores de offset medida desde el centro del compuesto, visto gráficamente en el juego, está lista se utiliza para mantener un control de los compuestos que se pueden ir agregando durante la ejecución del algoritmo, de tal manera que las clases derivadas de *Composite* harán referencia a una o más piezas según la lista que se haya proporcionado al crear el compuesto.

Código 3 Diccionario con los compuestos existentes

```

1   Composites = {
2       0: [("RectTiny", 0, 0, 0, "wood")],
3       1: [("RectSmall", 0, 0, 0, "wood")],
4       2: [("RectMedium", 0, 0, 0, "wood")],
5       3: [("RectBig", 0, 0, 0, "wood")],
6       4: [("RectFat", 0, 0, 0, "wood")],
7       5: [("SquareSmall", 0, 0, 0, "wood")],
8       6: [("SquareHole", 0, 0, 0, "wood")],
9       7: [("Circle", 0, 0, 0, "wood")],
10      8: [("TriangleHole", 0, 0, 0, "wood")],
11      9: [("RectBig", 100, 5, -27, "wood"), ("RectBig", -100, 5, 27, "wood"), ...
12          ...("RectSmall", 0, 0, 90, "wood")]
13 }
```

5.3 Definición de clase auxiliar de individuo

Mediante el uso de la clase *Composite*, se tiene un mejor control de los genes que conformarán cada individuo de la población. El siguiente paso es el de programar la clase que llevará el control de los cromosomas de un individuo, siendo los cromosomas los compuestos asignados a cada individuo, de tal manera que un individuo dentro de la información de cromosomas hará referencia a otra lista de compuestos, la manera en la que un *Individuo*

relaciona los compuestos es mediante el uso de la línea de código:

```
chromosome_objects = [Composite(Composites[composite]) for composite in self.chromosomes]
```

Mediante esta línea de código, se crea una lista en donde cada elemento será una instancia de un compuesto asignado al individuo, de esta manera si se requiere modificar la posición o materiales de un compuesto particular en el individuo es posible hacerlo desde la clase de *Individuo*, de igual manera está clase tiene otras funciones como se muestra en el diagrama de clase de la Figura 4.5, por ejemplo, realizar los cálculos de fitness y generar las listas de piezas que serán utilizadas para generar los archivos de salida para las simulaciones de los niveles generados.

5.4 Generación de individuos

Una vez definida la manera de controlar las clases el paso siguiente es el de comenzar con la generación de los individuos de la población, para poder lograr esto se creó una línea de código en la cual los puntos necesarios para la generación de los individuos se entregan totalmente, mediante la siguiente línea de código:

```
pop = [Individual(chromosome = get_random_chrom(ind_pieces), ...
    ..másk = create_new_másk(ind_pieces)) for i in range(population)]
```

Mediante el uso de esta línea de código se le indica al sistema que se requiere una lista que representará a los individuos de la población, cada elemento de esta lista será una instancia independiente de la clase *Individual*, para generar esta instancia de clase es requerido dos valores siendo estos primero una lista de valores numéricos que representan cuales piezas se asignaran a los individuos, para generar esta lista se utiliza una función como se muestra en el código 4, el segundo dato requerido es una segunda lista que denota una máscara mediante la cual las piezas asignadas serán acomodadas al momento de generar los archivos de simulación, finalmente, la última sección del código - *for i in range(population)* - denota que el proceso se deberá de repetir una cierta cantidad de

CAPÍTULO 5. IMPLEMENTACIÓN

veces, es decir que para cada individuo se generara una instancia con valores diferentes a los anteriores, la cantidad de veces que se deberá de repetir depende de la cantidad de individuos con los que se quiere estar trabajando en el algoritmo en el caso de las pruebas realizadas se utilizó un total de 10 individuos lo que significa que la lista *pop* generada consta de 10 instancias diferentes de la clase *Individual*.

Para la generación de la lista de piezas o cromosomas asignados a un individuo mostrada en la parte (1) del código en 4, aquí el valor que se recibe denota la cantidad de piezas o compuestos que se deberán de asignar en la lista del individuo, para asignar estos valores primero se obtiene un número aleatorio que va desde 0 hasta la cantidad de piezas o compuestos presentes en la lista, menos la unidad, para evitar errores de posicionamiento de lista. Posteriormente se revisa si la pieza seleccionada puede ser utilizada al menos en una combinación dentro de la generación, en caso de que no pueda ser utilizada se obtiene otra y de igual manera se revisa, en caso de poder ser utilizada, se agrega a la lista y avanza un contador para saber cuando se llegue al límite de piezas posibles. Finalmente se entrega la lista de piezas para la generación de la instancia de clase. Para la generación de la máscara de acomodo de piezas, se utiliza la sección de código mostrada en la parte (2). En esta parte se revisa la cantidad de compuestos que se pueden utilizar, como en la parte anterior, este valor se utiliza posteriormente para generar una lista aleatoria, la lista que se genera tiene un total de 7 posiciones que denotan las 7 divisiones que se realizan en el área de un nivel para el acomodo de las piezas. Mediante el uso del valor de piezas en cada individuo se realiza una aleatorización de números desde 0 hasta la cantidad máxima de piezas, una vez que se obtienen estos valores aleatorios se revisa si la suma de estos da la cantidad de piezas en el individuo, en caso de que no sea así, se repite el proceso, en caso de que si se cumpla entonces la lista se regresa para la creación del individuo.

Una vez que se ha logrado crear la lista de individuos se procede a inicializar el algoritmo evolutivo, este algoritmo se ejecutará una determinada cantidad de veces, el pseudocódigo se enlista a continuación en el código 5, este pseudocódigo engloba los aspectos principales del sistema de generación de niveles, el código se explica más a detalle en las secciones siguientes.

Codigo 4 Código de asignación de cromosomas(piezas)[1] y código para generar máscaras [2]

```

1   (1)  def get_random_chrom(sl):
2       asl = 0
3       chrom = []
4       while asl < sl:
5           prop = random.randint(0, len(Composites)-1)
6           if clases[Composites[prop][0][0]].Valid == True:
7               chrom.append(prop)
8               asl += 1
9           #random.randint(0,len(Composites)-1) for p in range(ind_pieces)
10      return chrom
11
12 (2)  def create_new_másk(pieces):
13     div_list = []
14     while True:
15         div_list = [random.randint(0, pieces-1) for col in range(7)]
16         if sum(div_list) == pieces:
17             break
18
19     return div_list

```

5.4.1 Integración de miembros elite

Está parte se encarga de iniciar los bucles de generación, la función principal de este bloque de código es la revisar la lista auxiliar de miembros de elite que existe en memoria, en caso de tener elementos en la lista estos se integran a la lista de la población integrándose al inicio de la misma, una vez agregados todos los miembros la lista de la población se corta hasta el número máximo de miembros permitidos en las generaciones, este valor se asigna como un valor de entrada al inicio del archivo que contiene el código.

Una vez terminada esta acción se procede a denotar la cantidad de padres que serán necesarios para generar los hijos de las generaciones futuras, esto se hace mediante el cálculo:

```
many = len(pop) * per_cross
```

En donde *pop* representa la lista de miembros de la población, *per_cross* representa un

Codigo 5 Pseudo-código del algoritmo genético

<i>Integrar miembros "elite" (1-4)</i>	<pre>if elite not null for member in elite population <- member trim population</pre>
<i>Simular individuos (6)</i>	execute process (game)
<i>Calcular fitness (8-9)</i>	<pre>for individual in population individual.getfitness</pre>
<i>Obtener padres de la generacion (10)</i>	parents <- selector_operator(population, required)
<i>Realizar crossover (12)</i>	crossover_operation(population, parents)
<i>Seleccionar elite (14-15)</i>	<pre>population.order('fitness') elite.add(population[1])</pre>

valor entre $0\dot{0}$ y $1\dot{0}$, este valor representa que tanto porcentaje de la población realizará cruces, de esta manera la variable *many* representa la cantidad de individuos que se deberán utilizar para cumplir el porcentaje de cruce, debido a que el valor de *per_cross* puede generar un valor flotante se tiene una parte de código en donde si el total de individuos que realizarán cruces es un número impar entonces el valor que se deberá de utilizar será el numero par redondeado hacia abajo, es decir, si de 10 individuos en la población se quiere que un total de individuos cercano al 50% realicen un cruce entonces el valor resultante para la variable *many* sería de 5, en este caso lo que se hace es restar 1 al resultado y después se redondea hacia abajo, lo cual haría que el número de padres requeridos para los cruces sea de 4.

Una vez que se tiene el valor de los padres requeridos para los cruces de la generación se procede a crear los archivos auxiliares de los individuos de la población, los archivos generados son en formato *XML*, la información que contienen estos archivos es la posición, tipo y material de las piezas que serán utilizadas en el nivel, además de esto también contiene la información de la cantidad y tipo de aves así como de los enemigos que serán colocados, estos archivos son almacenados en un folder en donde el software de simulación se encarga de obtenerlos y entregar los resultados después de la simulación.

Una vez que los archivos de simulación han sido generados el algoritmo manda un

CAPÍTULO 5. IMPLEMENTACIÓN

comando que manda la ejecución del software de simulación, el software en particular cuenta con una ventana gráfica en la que se puede apreciar los niveles que se generaron o evolucionaron en caso de estar simulando archivos de una generación después de la primera, de igual manera el software genera un archivo en formato *XML*, similar al que se genera antes de la simulación, la diferencia entre ambos archivos es que el entregado después de la simulación entrega los datos de posición de el último estado registrado del nivel, además de que se puede solicitar que entregue el valor de la aceleración de la misma. El valor de aceleración es tomado en cuenta desde el punto en el que la pieza comienza su descenso, en caso de que una pieza sea destruida durante el proceso de simulación ninguno de los datos de esa pieza son grabados en los archivos, la manera de llevar control de las piezas que entran y las que salen es mediante el uso de un número de lista asignado al momento de generar los archivos de simulación de la sección anterior.

Posterior a la simulación los archivos entregados son revisados y en base a la información de las piezas se calcula el *Fitness* de las mismas, posteriormente a esto se procede a ordenar la lista, y realizar las operaciones de selección, cruce y mutación los cuales se explican en la sección siguiente.

5.4.2 Operador de selección

El proceso de selección se definió de dos maneras, la primera es mediante el uso de una ruleta, en esta ruleta los valores que se toman en cuenta es el fitness de cada individuo, mientras más alto sea el fitness mayor será la probabilidad de ser seleccionado para el cruce en la siguiente sección.

El segundo método utilizado es la selección mediante torneo, en este tipo de selección lo que se realiza es que los individuos se seleccionan en pares de manera aleatoria, entre cada par se realiza un cálculo de cual tiene el mejor fitness y ese individuo es seleccionado como padre para la generación.

5.4.3 Operador de cruce

Para los operadores de cruce se tienen codificados dos tipos, cruce de un punto y cruce de dos puntos, este operador de cruce se realiza no solo en los individuos de la población sino también en las máscaras que tienen los mismos individuos, esto es para aumentar un grado más el nivel de diversidad que se puede tener en los individuos generados, la idea en este caso es lograr que existan casos en donde la máscara sea muy chica y varias piezas de los individuos no sean mostradas y el caso contrario permitirá que un individuo pueda o no quedar con el mismo caso y puede que se dé el caso en donde alguno de ambos elementos logren obtener un mejor nivel de fitness en la generación.

Un ejemplo de estos dos tipos de cruce se puede apreciar en la Figura 5.1 en donde dos individuos realizan un cruce con sus máscaras y las máscaras generadas para los hijos resultan con diferente cantidad de piezas.

La manera en cómo se aplican estos tipos de operador es:

1. Definir un punto de corte igual para los dos individuos en cualquier posición de la lista de genotipos.
2. Separar las dos partes cortadas de ambos individuos de tal manera que se tenga la parte de inicio (del inicio hasta el corte) y la parte final (desde el corte hasta el final) de cada individuo.
3. Para crear al primer hijo tomar la parte inicial del primer individuo y unirla con la parte final del segundo individuo
4. Para el segundo hijo tomar la parte inicial del segundo individuo y unirla con la parte final del primer individuo.

De esta manera los hijos se pueden comprender de la siguiente manera:

```
padre1 = padre[1, punto_corte]
padre2 = padre[punto_corte, fin]
```

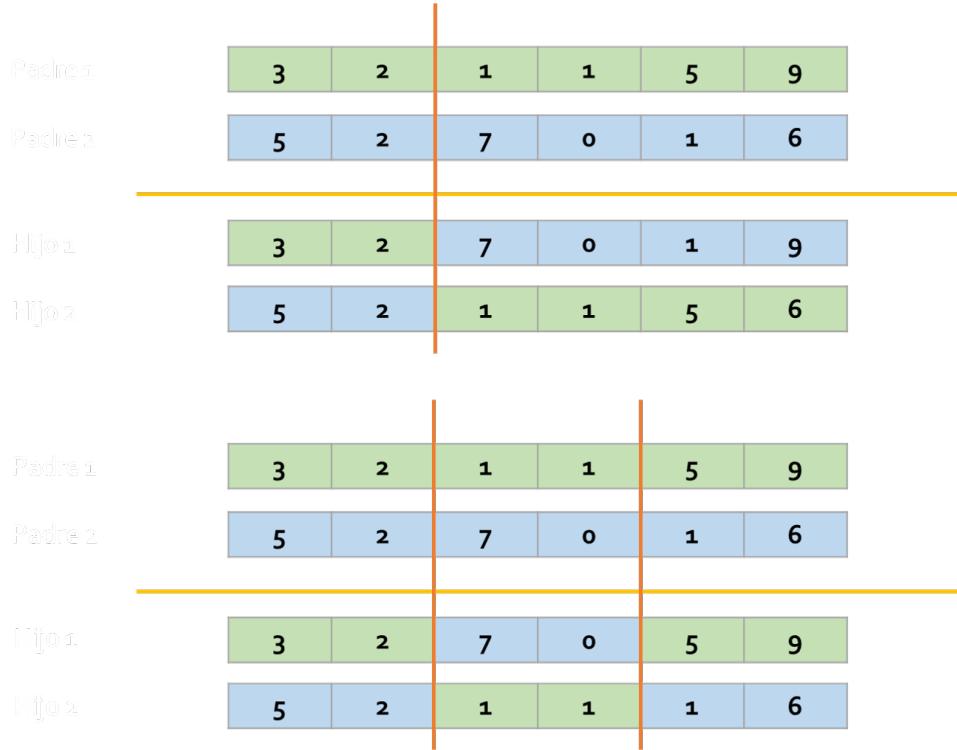


Fig 5.1: Cruce de un punto (arriba) y cruce a dos puntos (abajo) aplicada a la máscara de individuos

```

madre1 = madre[1, punto_corte]
madre2 = madre[punto_corte, fin]

hijo1 = padre1 + madre2
hijo2 = madre1 + padre2

```

Para el caso del cruce de dos puntos, en vez de dividir el genotipo de un individuo en dos partes, se divide en tres y al momento de realizar la combinación se toma la primer parte del primer individuo, después la parte de enmedio del segundo individuo y finalmente la parte final del primero, de tal forma que solo la parte central de ambos individuos se cambia.

5.4.4 Operador de mutación

La mutación de los individuos de la población ocurre solo al momento de hacer el cruce de los mismos, es decir solo a los elementos nuevos se les aplica el operador de mutación, este operador puede modificar los siguientes elementos de un individuo:

1. La cantidad de piezas que conforman al individuo.
2. El material del cual están formados los elementos.
3. El tipo de elemento que conforma a un individuo, es decir, modificar el valor del compuesto que forma parte del individuo.
4. La posición individual (x o y) de los compuestos.

El operador de mutación se describe cómo sigue, primero al momento de generar al nuevo individuo se realiza un cálculo de porcentaje en donde se decide si tendrá o no mutación, dependiendo del porcentaje de mutación que se haya asignado al inicio del algoritmo, posteriormente en caso de que el valor de porcentaje quede dentro del rango, se toma para las operaciones siguientes:

Para mutar la cantidad de compuestos en un individuo primero se realiza una selección aleatoria entre las opciones de agregar o quitar compuestos, posteriormente en caso de requerir agregar más se realiza una selección aleatoria de entre la cantidad de compuestos creados, estos mismos se integran al final de la lista de compuestos del individuo. Para eliminar elementos de la lista se realiza una corrida por todos los elementos y en cada uno se realiza una probabilidad de 50% de sea borrado de la lista, un ejemplo de este proceso se muestra en la Figura 5.2 en donde se presenta una tabla que denota las posiciones de los compuestos en base a la máscara, en este ejemplo se realizó una corrida para remover elementos (color rojo) y una segunda para agregar nuevos compuestos (color verde).

La segunda mutación se encarga de modificar el material del que están conformados los compuestos del individuo, primero se selecciona aleatoriamente cuales compuestos serán modificados sin repetir valores, posteriormente se modifica de manera aleatoria el valor

			0				
4		1	2	6		2	
1		5	8	9		3	
0	0	4	3	2	4	5	

			6				
			0				9
		1	2	6		2	
1		5	8	9	2	3	
0	0	4	3	2	4	5	

Fig 5.2: Ejemplo del operador de mutación enfocado a remover y agregar compuestos

que define el tipo de material del que se conforma, para esto se toma en cuenta que *0* representa material de madera, *1* representa material de hielo y finalmente *2* representa el material de piedra, debido a que el juego solo acepta esta lista de compuestos no es posible asignar valores diferentes a estos.

En la Figura 5.3 se muestra un ejemplo de la manera en cómo opera la mutación del tipo de compuesto, en este ejemplo se muestra el genotipo y fenotipo de un individuo, el genotipo muestra los colores naranja, azul y gris para denotar los materiales de madera, hielo y piedra respectivamente, en este caso la mutación cambia el material de 5 elementos del genotipo de madera a hielo, esto se refleja del lado derecho en donde se muestra la vista actualizada del individuo después de la mutación.

Debido a que los compuestos generados pueden contener diferentes cantidades de elementos en estos casos se cambia el material equitativamente a todos elementos del compuesto, de esta manera se evita tener que realizar un recorrido por todo el compuesto y mutar el material de cada uno.

La tercera mutación se encarga de cambiar el compuesto que representa una posición del genotipo del individuo, este trabaja de una manera similar al anterior en el sentido de que primero se selecciona a los individuos a mutar, posteriormente a la selección se realiza una segunda selección aleatoria entre las posiciones del genotipo del individuo, aquellos compuestos seleccionados son cambiados por otro de la lista existente, para esto se

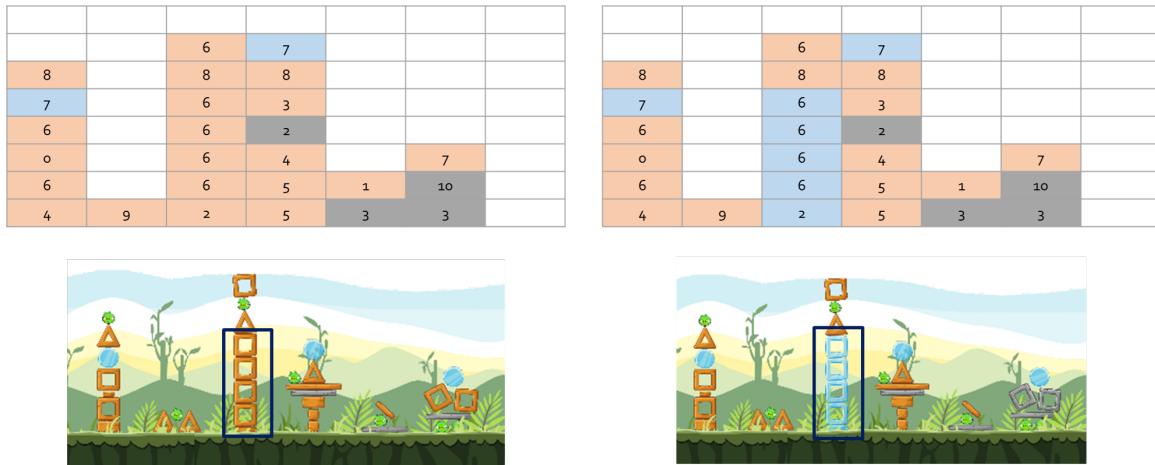


Fig 5.3: Ejemplo del operador de mutación enfocado a cambiar el material de los compuestos, pre-mutación (izquierda) y post-mutación (derecha)

realiza una selección de un valor aleatorio que va desde 0 hasta la cantidad de compuestos existentes, por ejemplo, suponiendo que se tiene solo la cantidad base de piezas en el algoritmo entonces la selección será un valor aleatorio desde 0 hasta 10 .

La Figura 5.4 nos muestra un ejemplo de cómo es que actúa la mutación de tipo de compuesto, en el ejemplo mostrado se tiene el genotipo de un individuo combinado con su respectiva máscara así como su fenotipo respectivo (lado izquierda) al ser representado en la pantalla del juego, en el lado derecho de la misma imagen se aprecia la modificación de compuestos realizada al genotipo base (marcado en amarillo), la modificación de compuestos en los individuos puede traer la consecuencia de que los niveles generados sean inestables como en este ejemplo, esto a su vez puede permitir identificar máscaras de los individuos que logran mantener la estabilidad de las estructuras permitiendo así mejorar el algoritmo.

El último tipo de mutación realizada en los individuos es la modificación de las posiciones x y y de algunos compuestos dentro del individuo, la modificación de las posiciones se realiza mediante una selección de valores aleatorios de una distribución gaussiana dentro de un cierto rango de distancia para no permitir que los elementos se alejen demasiado de punto central original.

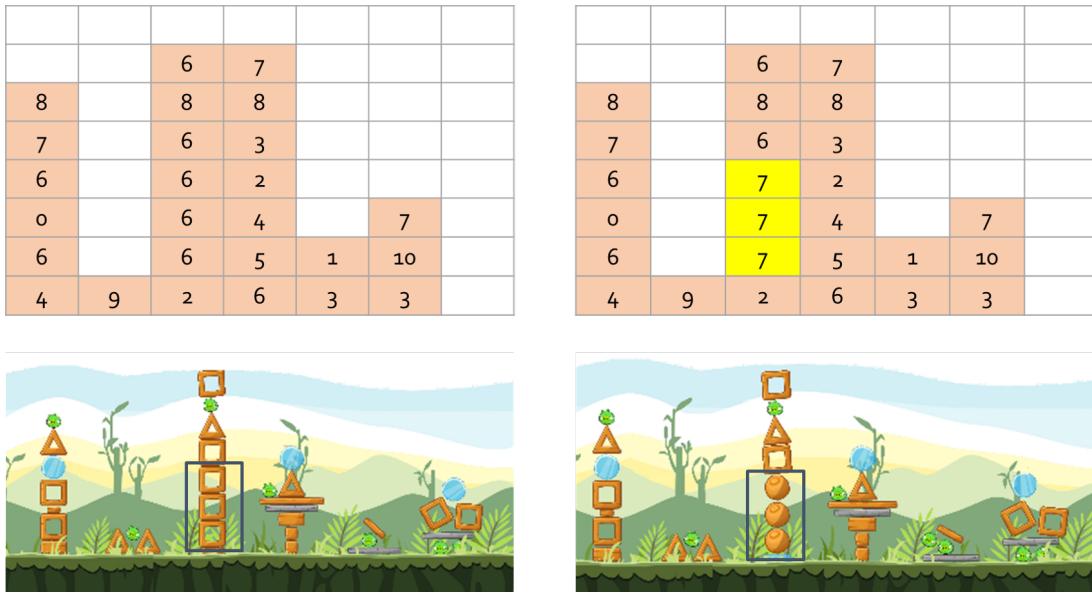


Fig 5.4: Ejemplo del operador de mutación enfocado a cambiar los compuestos asignados, pre-mutación (izquierda) y post-mutación (derecha)

5.4.5 Representacion de individuos

Una vez que se han agregado nuevos individuos a la población, y que estos individuos pasaron por su fase de mutación, si es que requerían, se procede a realizar la combinación de los elementos que conforman al individuo con el fin de generar los archivos requeridos para la simulación de estos mismos. Cabe resaltar que esta combinación se realiza también en los individuos iniciales antes de la simulación inicial, debido a que el juego requiere una representación mediante el uso de archivos en formatos *XML*, se requiere primero obtener la información de todos los elementos que conforman a un individuo y posteriormente armar los archivos mediante el uso de estos datos.

En la Figura 5.5 muestra la estructura de los componentes de un individuo, un individuo es representado por 2 capas diferentes en forma de lista, la primera es la capa de compuestos, esta capa se muestra la lista de compuestos que representan a un individuo, esta lista puede variar en tamaño, dependiendo de si durante las operaciones de mutación se le agregaron o quitaron elementos al individuo, de igual manera cómo se explicó en el la sección 5.2, cada valor de la lista representa no una pieza del juego sino uno de los

CAPÍTULO 5. IMPLEMENTACIÓN

Composites layer

8 | 7 | 6 | o | 6 | 4 | 9 | 6 | 8 | 6 | 6 | 6 | 6 | 2 | 7 | 8 | 3 | 2 | 4 | 5 | 5 | 1 | 3 | 4 | 10 | 3

Mask layer

6	1	7	7	2	3	0
---	---	---	---	---	---	---

Resulting level

		6	7			
8		8	8			
7		6	3			
6		6	2			
0		6	4		7	
6		6	5	1	10	
4	9	2	5	3	3	

Fig 5.5: Ejemplo de la combinación de capas de un individuo

CAPÍTULO 5. IMPLEMENTACIÓN

compuestos que se crean antes de iniciar el algoritmo o aquellos que se agregan durante el mismo.

La segunda parte que compone la representación del individuo es la máscara, cómo se explica en la sección 4.2.2. La idea detrás del uso de esta máscara, es la de generar estructuras utilizando la lista de compuestos del individuo, en esta misma sección se presentó una propuesta de crear máscaras en donde se denotará la forma que se quería que tuviera un individuo particular, aplicando estas estructuras en los individuos, generaba los resultados esperados, siendo el caso niveles en donde la distribución de los compuestos crearan formas diferentes como castillos, torres, casas o diferentes figuras. Sin embargo utilizar este tipo de estructuras no permitía que el algoritmo lograra evolucionar, sino que simplemente encontraría la mejor combinación de piezas-máscara en determinado momento, por esto se optó por utilizar la generación de máscaras que se muestra en la Figura 5.5 y que fue explicado en la sección 5.4 la cual únicamente muestra una determinada cantidad de piezas a asignar en posiciones diferentes del nivel, esto permite tener mejor diversidad en los individuos creados.

Al momento de combinar ambos componentes del individuo se obtiene un resultado similar al mostrado en la parte inferior de la imagen en donde los compuestos se organizan en manera de cola, es decir el primer elemento en la lista será el primero en ser colocado, una vez que se definen cuales elementos serán colocados en cada columna, el sistema calcula la altura total de cada compuesto y comenzando desde la parte inferior coloca un compuesto en el nivel, después calcula la distancia desde la parte superior del mismo hasta la parte central del siguiente para colocarlo de tal forma que al comenzar la simulación no aparezcan en caída libre, sino que aparezcan una pieza sobre otra, de esta manera se previenen problemas de balance al permitir que las piezas caigan y reboten en direcciones que provocaran que las estructuras terminen por caer totalmente, en la misma imagen se aprecian cuadros de diferentes colores, estos mismos representan el material del cual está construido cada compuesto como se explicó en la sección de mutación anterior.

Finalmente, una vez que se tiene la información anterior definida se procede a realizar una búsqueda de los lugares en donde aparecerán los enemigos del juego, se define dónde

pueden aparecer estos mediante una función que revisa los compuestos que existen en el nivel. Cuando un compuesto cumple con ciertas características se permite que uno de los enemigos pueda ser colocado en la parte central o superior del mismo, al iniciar este método se busca aquellos que cumplen esta característica y se ordena en una lista auxiliar, después de manera pseudo-aleatoria se decide la cantidad de enemigos que tendrá el nivel, de acuerdo a este valor se seleccionan x cantidad de posiciones para colocar a los enemigos donde x es el valor de enemigos requeridos, una vez se definió en donde estarán colocados estos enemigos se regresa una lista con las coordenadas x y y de las mismas.

Mediante la combinación de los enemigos con el genotipo construido de un individuo se obtiene el nivel a mostrar en el juego, una representación de esto se muestra en la Figura 5.6, en donde al genotipo armado de acuerdo a los puntos anteriormente mencionados se le agrega el listado de posiciones de los enemigos. En este caso los enemigos son representados por un color verde en las posiciones en donde estarán. Esto permite tener la estructura de los niveles completa, esta información se utiliza al momento de generar los archivos de *XML* los cuales en el software de simulación generarán una vista de los mismos, cómo se muestra en la esquina inferior derecha de la misma imagen.

5.4.6 Cálculo de fitness

Cuando los individuos pasan por el proceso de simulación se genera un archivo en formato *XML* que contiene los resultados de los compuestos que fueron puestos en los archivos antes de la simulación, con la diferencia de que estos archivos contienen la información del último estado de los mismos compuestos. Al terminar dicha simulación, se obtiene un listado de piezas y enemigos con la información de sus posiciones x y y así como del ángulo con en el cual quedaron al concluir la simulación, además de esto se entrega la información de la velocidad promedio del movimiento que tuvieron.

Esta información permite conocer las imperfecciones que tuvieron los niveles al concluir las simulaciones, para mejorar la evaluación obtenida que determina la aptitud de los individuos se toma en cuenta una evaluación de dos fases, la primera fase tiene que ver

CAPÍTULO 5. IMPLEMENTACIÓN

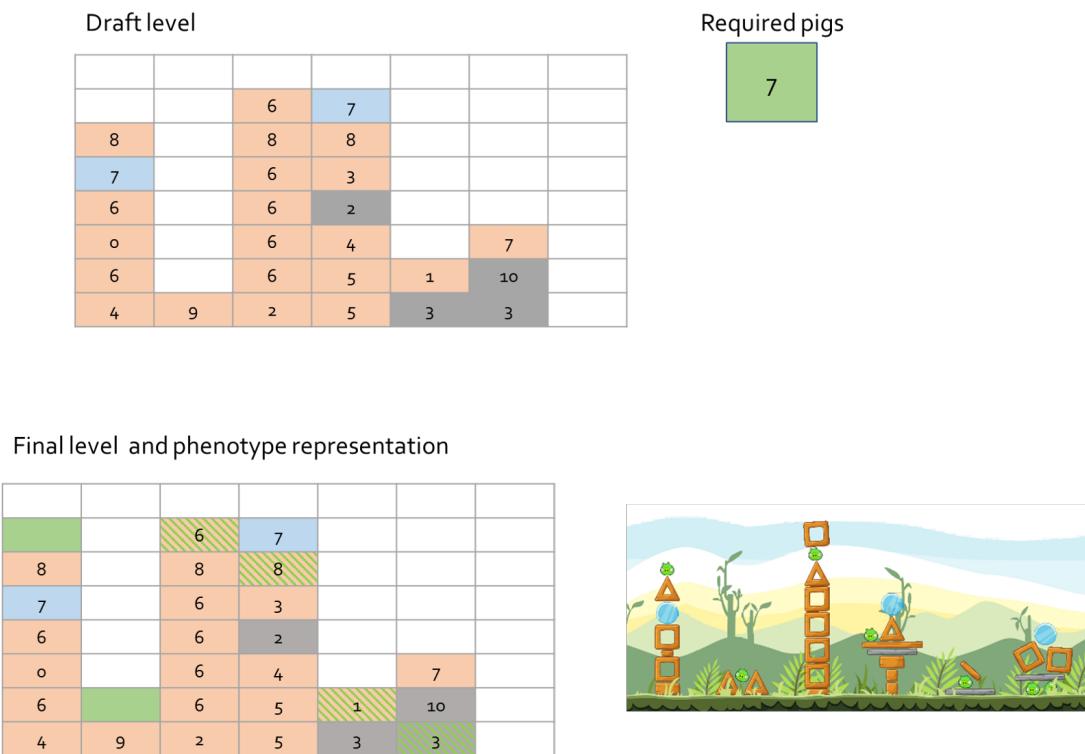


Fig 5.6: Ejemplo de la combinación de un genotipo armado de la Figura 5.5 con la colocación de enemigos

CAPÍTULO 5. IMPLEMENTACIÓN

con la estabilidad de los individuos y se evalúa en base a dos métricas:

1. La cantidad de piezas que no son destruidas durante la simulación.
2. Las posiciones finales de dichas piezas al terminar la simulación.

Para poder realizar el cálculo de las piezas presentes antes y después de la simulación se utiliza la información de las piezas anteriormente mencionada, sus posiciones tanto x como y así como del angulo de las mismas, a esta información se le agrega un indicador de posición para poder compararlo con los datos del mismo indicador para las funciones siguientes, esta lista de información se realiza también antes de la simulación con el fin de tener datos con los cuales comparar, una vez que se tienen los datos se procede a realizar el proceso de cálculo de fitness, para calcular la diferencia de piezas se compara la cantidad de elementos en ambas, debido que al momento de caer de alturas grandes algunos elementos tienden a romperse entonces la cantidad de elementos después de la simulación podrá ser menor, para obtener el primer valor se realizará un cálculo para obtener un valor entre 0 y 100 que define la calificación en forma de porcentaje del total de piezas que se salvaron en la simulación, es decir, si antes de la simulación se tenía un total de 10 piezas y al terminar la simulación en el archivo correspondiente solo existen 5 entonces el individuo tendrá una calificación de 50% debido a que solo la mitad de las piezas no fueron destruidas.

El segundo valor toma se utiliza para definir si los conjuntos que no se destruyeron se lograron mantener los más estables posibles, esto se calcula mediante el uso de dos fórmulas mostradas en las formulas 5.1 y 5.6 mediante el uso de estas fórmulas se comprueba si las piezas lograron o no mantenerse en sus posiciones originales. En los casos donde las piezas no se hayan destruido se utilizan estas fórmulas para revisar si las posiciones y ángulos de inicio y final de la simulación son los mismos, en caso de que así sea, no se modifica el fitness obtenido previamente, en caso contrario se obtiene una penalización al fitness, equivalente a una tercera parte del valor de porcentaje por una pieza si se movió

de posición y otra tercera parte, si su ángulo cambió.

$$error_{xy} = \begin{cases} 0 & \text{si } 0.08 > d \\ \frac{100}{longitud_piezas} * -0.33 & \text{si } 0.08 < d \end{cases} \quad (5.1)$$

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

$$error_r = \begin{cases} 0 & \text{si } -5 < r < 5 \\ \frac{100}{longitud_piezas} * -0.33 & \text{si } 0.08 < d \end{cases} \quad (5.2)$$

$$r = |rotacion_o| - |rotacion_f|$$

Por otro lado, la segunda fase de la evaluación busca tener más diversidad en los individuos, esto es, que tan *novedosos* o *diferentes* logran ser, la manera en cómo se propone realizar esta evaluación de diversidad es mediante el uso de las siguientes métricas:

1. La diversidad de las piezas utilizadas en los individuos
2. El resultado de la función de entropía con las piezas utilizadas.
3. La distancia *Hamming* del mismo conjunto de piezas.

Para evaluar la diversidad de los elementos primero se realiza un cálculo de los diferentes compuestos que integran a un individuo, para la evaluación de esta parte se toma en cuenta la diversidad que logra tener el individuo, en donde el utilizar diferentes compuestos de la lista total permite tener una mejor calificación, esto es debido que mientras más compuestos sean utilizados los niveles que se pueden generar serán diversos entre sí.

Posterior a esto se realiza un cálculo de la entropía de un individuo, el cálculo de entropía es utilizado en la teoría de información para calcular el nivel de desorden o incertidumbre en los individuos, la fórmula que se utiliza para esto se muestra en 5.3

$$s = - \sum_i P_i \log P_i \quad (5.3)$$

La manera en cómo la entropía es utilizada en este proyecto es para determinar si un individuo se vuelve "*aburrido*" o "*interesante*" la manera en cómo se define este concepto para el individuo es mediante la medición de las piezas semejantes en el mismo, de acuerdo a esto si la cantidad y tipos de compuestos se repiten demasiado en el individuo entonces el individuo tiene entropía baja o en otras palabras se vuelve "*aburrido*" y *monotono*, es decir si se tiene entropía baja entonces significa que los compuestos utilizados son mayormente iguales dentro del individuos los cual generará un nivel demasiado simple a diferencia de si se tiene entropía alta significa que la cantidad de compuestos diferentes es alta los cual generara niveles con formas un poco más impredecibles.

$$d = \min \left\{ d(x, y) : x, y \in C, \text{ si } x \neq y \text{ entonces } 1 \text{ sino } 0 \right\} \quad (5.4)$$

Una vez que se obtuvo el cálculo de la entropía del individuo se procede a obtener la distancia *Hamming* del mismo, de manera sencilla la *distancia de Hamming* (Hamming distance en Inglés) representa el cálculo realizado en dos cadenas de valores en donde el punto es encontrar el valor mínimo de sustituciones necesarias para cambiar una cadena a otra, visto de otra manera, la *distancia de Hamming* se encarga de encontrar la distancia más corta entre dos cadenas, la manera en cómo funciona está función se muestra en la formula 5.4, en esta misma ecuación los objetos x y y representan listas, en este caso se manejan listas de valores boléanos es decir listas de valores 0 y 1, una explicación más

simple se presenta en la función 5.5.

$$\begin{aligned}
 C &= \{a, b, c\} \\
 a &= (00000) \\
 b &= (10110) \\
 c &= (01011) \\
 d(a, b) &= 3 \quad d(a, c) = 3 \quad d(b, c) = 4
 \end{aligned} \tag{5.5}$$

$$error_r = \begin{cases} 0 & \text{si } -5 < r < 5 \\ \frac{100}{longitud_piezas} * 0.5 & \text{si } 0.08 < d \end{cases} \tag{5.6}$$

$$r = |rotacion_o| - |rotacion_f|$$

En esta función se tiene un conjunto de listas representado por C , en este conjunto existen las listas a , b y c , para poder encontrar la *distancia de hamming* de cada par de listas se utiliza la formula 5.4 esto es, se revisa cada par de elementos de la listas y se suma 1 si los elementos son diferentes, de esta manera se logra determinar que la cantidad de cambios necesarios para hacer que la lista b sea igual a la lista a es 3 debido a que solo se requiere cambiar los tres valores 1 en la lista b , la razón por la que no es requerido cambiar valores en la lista a es debido a que en la función la segunda lista utilizada es la que se quiere asemejar a la primera no solamente cambiar una lista a otra.

Al utilizar la distancia de *hamming* se obtiene la cantidad menor de cambios para asemejar una lista a otra, sin embargo, como el fin del sistema es crear niveles que sean diferentes entre si, entonces es posible incluso modificar la evaluación para obtener este valor para obtener la mayor distancia posible de tal modo que se sabrá cuales individuos tendrán la tendencia de ser diferentes para generar más diversidad.

5.4.7 Selección de miembros élite

Finalmente, después de obtener los resultados de aptitud de los individuos, se procede a realizar la selección de aquellos que lograrán integrar el grupo de élite que se utilizará para sustituir miembros de la población al inicio de la siguiente generación, para realizar esto, primero se requiere ordenar todos los individuos por medio de su respectivo valor de fitness, desde el valor más alto hasta el valor más bajo. Posteriormente se toma una determinada cantidad de individuos, iniciando desde el mejor. La cantidad de individuos que se tomarán para el grupo de elite, es una cantidad que se define antes de iniciar el algoritmo genético.

Una vez que se ha tomado la cantidad de individuos requerida se procede a integrar estos mismos individuos a el grupo de élite, para realizar la integración al grupo de elite primero se requiere analizar el estado del grupo, en caso de que no existan elementos en la lista lo cual solo sucede en la primera generación, el grupo seleccionado se integra automáticamente a la lista de elite, después se ordena la lista de acuerdo al valor de aptitud y se corta de tal manera que la longitud de la lista de elite sea menor o igual a un valor que se determina antes de iniciar el algoritmo, en caso de que ya existan elementos en el grupo entonces estos miembros se integran al final de la lista del grupo elite, se realiza un reordenamiento en la lista en base al valor de aptitud de los individuos y se corta la lista en caso se ser requerido.

Una vez que se han completado todos los procesos explicados anteriormente, el algoritmo agrega datos de los mejores, los perores y el promedio de los valores de aptitud de los individuos en listas que sirven para realizar promedios al final de toda la ejecución. Además de esto, también se guarda información de los resultados individuales de las funciones de *entropía* y la *distancia de Hamming* con el fin de ver en manera de gráficas el comportamiento del algoritmo bajo los datos que le fueron proveídos al inicio de la ejecución.

Al terminar este proceso de integración de datos, se realiza el último movimiento en la información de la generación, siendo este obtener los archivos *XML* de los individuos

CAPÍTULO 5. IMPLEMENTACIÓN

y guardándolos en una nueva locación, con el fin de mantener un registro del progreso de los individuos durante la ejecución del algoritmo.

Capítulo 6

Experimentos y Resultados

Este capítulo es el encargado de describir los diferentes experimentos realizados con el sistema, dichos experimentos se realizaron para comprobar la correcta funcionalidad de lo creado, así como para revisar que efectivamente el sistema hace lo que fue propuesto en capítulos anteriores, eso es la generación de niveles estables y diferentes entre sí.

La manera en cómo se representarán estos resultados será mediante el uso de dos gráficas, la primera establece los valores de fitness mínimos y máximos que se logran alcanzar durante las generaciones, además de esto en la misma grafica se presenta una línea extra que representa el promedio de los fitness de los mismos individuos, siguiente de esto se presenta una segunda grafica que representa la *distancia hamming* mínima alcanzada en cada conjunto de individuos durante las generaciones, como se explicó anteriormente esta grafica representa el valor mínimo de cambios entre los conjuntos de listas de genotipos de los individuos, esta grafica tendrá la tendencia de ir reduciéndose conforme avanzan las generaciones debido a que mientras más se avanza en las generaciones los elementos elite tenderán a ir apareciendo más.

La manera en cómo se define el valor de fitness de los individuos es como se explica en la sección 5.4.6 es mediante la separación de los dos aspectos de importancia en los niveles, primero la *estabilidad* que define el comportamiento del genotipo durante la simulación y la segunda siendo *diversidad* que define lo *nuevo* que logran ser los niveles generados.

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS

Los parámetros del algoritmo genético explicados en el capítulo 5 se muestran en la tabla 6.1.

Tabla 6.1: Parámetros utilizados en el algoritmo genético

Parametro	Valor
Fitness Function	Estabilidad Diversidad
Tamaño de la población	10 o 20
Numero de Generaciones	100 o 10 (respectivamente)
Criterio de parada	Numero de Generaciones
Operador de selección	Selección por torneo
Operador de cruce	Cruce de un punto
Porcentaje de cruce	30%
Operador de mutación	Mutación de individuo Mutación de compuestos Mutación de material
Porcentaje de mutación	30%

El contenido de este capítulo se encargará de mostrar las capacidades de generación del sistema propuesto e implementado según lo mostrado en el capítulo 5 de acuerdo a las áreas explicadas anteriormente. Cada experimento mostrado se describirá en los ámbitos de la capacidad de generar niveles *estables* así como también la capacidad de generar niveles *diversos*, puesto que se tienen una gran cantidad de individuos en las simulaciones entonces para demostrar ambos ámbitos se tomarán el *mejor* y el *peor* del final de cada experimento, así como un individuo aleatorio de la primera generación, utilizando estos niveles generados se explicará la manera en cómo se evolucionó la diversidad de los niveles y cuáles fueron los compuestos que se pueden rescatar de las simulaciones para ser utilizados en experimentos subsecuentes, para tener un campo nivelado para todos los experimentos aquellos compuestos que aparecen durante los experimentos no son reutilizados en otros experimentos subsecuentes.

Todos los experimentos mostrados en esta sección fueron optimizados durante un total, de 10 generaciones, esto es debido principalmente a que para realizar las simulaciones se requiere una gran cantidad de tiempo, esto sumado con la cantidad diferente de individ-

uos, en experimentos de 100 generaciones se utilizaban un total de 10 individuos y para experimentos de 10 generaciones se utilizo un total de 20, además de esto en muchos de los experimentos de 100 generaciones el algoritmo llegaba a un punto de estancamiento generalmente antes de las primeras 50 generaciones, por tal motivo se decidió reducir la cantidad de generaciones a 10 pero incrementar el total de individuos para obtener una mayor diversidad.

Como se explico en el párrafo anterior los experimentos que se presentaran cubrirán las variables de 20 individuos durante un total de 10 generaciones y utilizando las variables presentadas en la tabla 6.1. Utilizando estas configuraciones se realizaron un total de [Write number here?] sin embargo, debido a que los resultados mostrados por cada experimento son demasiados solo algunos resultados de experimentos serán mostrados.

6.1 Generación de niveles estables

La primera sección de resultados se enfoca a mostrar lo que el sistema es capaz de generar en cuanto a niveles estables se refiere, para este aspecto se tomaron en cuenta los puntos propuestos en la sección 4 en donde se busca obtener los mejores niveles tomando en cuenta las posiciones de las piezas colocadas antes y después de las simulaciones siempre teniendo en cuenta que aquellos niveles que logren mantenerse de pie o inclusive mantener la mayor cantidad de compuestos aun presentes en el nivel son los que se consideran como los mejores niveles para cada generación, de la misma manera en cómo se explico anteriormente aquellos niveles que logren cumplir estas características son registrados en una lista que mantiene la información de los mejores niveles generados a lo largo de las generaciones y elementos de este mismo grupo son reintegrados a la población al iniciar la siguiente generación.

La manera en cómo se representan los resultados de las simulaciones es mediante el uso de 2 graficas que muestran los valores de la función de aptitud obtenidos, en esta grafica se muestra el promedio de los individuos en una generación dada, el valor mínimo que se alcanza en dicha generación, así como el valor de aptitud máximo que se logra obtener

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS

en las generaciones, en este aspecto cabe resaltar que obtener no todos los individuos con un valor de 100% de aptitud serán los mejores elementos de la población, sino que simplemente la manera en cómo se generaron los niveles no permite que los compuestos sean destruidos para esto es necesario revisar de manera visual los resultados y darse cuenta de que elementos que logran obtener valores diferentes al 100% de aptitud pueden ser elementos validos si las estructuras generadas logran cumplir con los objetivos buscados, después de la gráfica de aptitud de presentan dos ejemplos de los niveles generados, aquellos denotados por b) representan el "mejor" individuo de la generación, mientras que aquellos representados por c) representan el "peor" elemento de la primera generación, estos ejemplos son representados a su vez de dos maneras, la imagen superior representa el estado inicial del individuo, es decir al momento de iniciar la simulación, mientras que la imagen inferior representa el estado final del mismo individuo, es decir el punto en el que la simulación termino, mediante el uso de estas imágenes es posible apreciar la manera en cómo los niveles generados se comportan en términos de estabilidad bajo la gravedad proporcionada en el juego.

De igual manera se presenta información de la última generación de la simulación, en este caso se toman únicamente los dos individuos (diferentes entre sí) que representan los mejores individuos de la población, en este caso se toman dos individuos diferentes debido a que debido a la naturaleza de los algoritmos genéticos la tendencia de los individuos es asemejarse unos a otros y mediante esto se busca mostrar que es posible inclusive en la generación de niveles obtener elementos que no necesariamente son los mejores generados pero logran acercarse lo suficiente a los objetivos del sistema.

6.1.1 Experimento 1

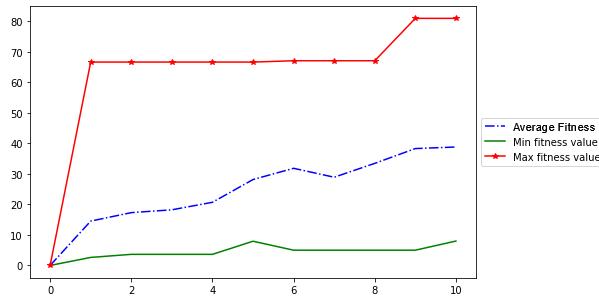
La Figura 6.1 más específicamente en la parte 6.1(a) la gráfica de los resultados de aptitud obtenidos durante las simulaciones, en esta grafica se puede notar que durante las primeras generaciones se un elemento de la población siempre mantiene el mismo nivel de estabilidad, una vista de este elemento de la población se puede aprecia en la sub

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS

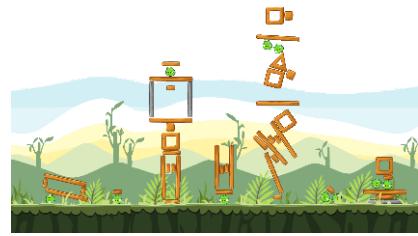
Figura 6.1(b) de la misma imagen, así mismo se obtuvo uno de los "peores" elementos de la generación el cual se muestra en la sub Figura 6.1(c), como se puede apreciar en la imagen este elemento de la población inicia la simulación con varias piezas muy elevadas e inclusive mal acomodadas lo cual genera que al terminar la simulación solo una parte de las estructuras originales se mantenga de pie.

Como es posible apreciar en la gráfica este conjunto de niveles con dicha estabilidad se logra mantener durante varias generaciones no es sino hasta las últimas dos generaciones en donde por medio de las operaciones de cruce y mutación se obtiene un nuevo individuo capaz de llegar al 80% de estabilidad, este nuevo elemento se puede apreciar en 6.2(a), aquí mismo se puede apreciar que a pesar de tener el mejor resultado de aptitud de la generación en ocasiones no es posible mantener el balance en todos los elementos, sin embargo, muchos se logran mantener sin ser destruidos, de igual manera se presenta en 6.2(b) otro individuo representante de la ultima generación de la simulación, en este caso este segundo individuo representa el segundo mejor valor de aptitud que no es completamente igual a aquel que está en la primera posición, esto es debido a que al integrar miembros de elite de vuelta a la población muchas veces se integran más de uno que son individuos completamente iguales debido a que si durante una generación dada el valor de aptitud no fue mejor que el de un miembro de elite ya existente entonces el mejor de la generación muchas veces siendo el mismo que se reintegro a la población es el que se agrega de vuelta a la lista de miembros elite.

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS



(a) Resultados de aptitud para el experimento



(b) Mejor individuo, generación 1

(c) Peor individuo, generación 1

Fig 6.1: Resultados del experimento, gráfica de la función de aptitud figura a), ejemplos de individuos: figuras b) y c)



(a) Mejor individuo, ultima generación



(b) Segundo mejor individuo, ultima generación

Fig 6.2: Resultados del experimento, primer y segundo mejor nivel generado (sin repetir)

6.1.2 Experimento 2

Como se puede apreciar en la Figura 6.3 o más directamente en 6.3(a) esta experimento inicio mal debido a la manera en cómo se generaron los compuestos al iniciar las generaciones, de igual manera este inicio no tan bueno se puede aprecia en la sub Figura 6.3(b) en donde varios de los elementos generados no logran mantenerse en buen balance y terminan cayendo, algunos destruyéndose en el proceso, de igual manera en la sub Figura 6.3(c) del peor individuo de la generación se puede ver que muchos de los elementos inician su posición en posiciones muy altas o en el caso mostrado en esta Figura con acomodos muy extraños que terminan por destruir toda la estructura que se esperaba crear en ese punto.

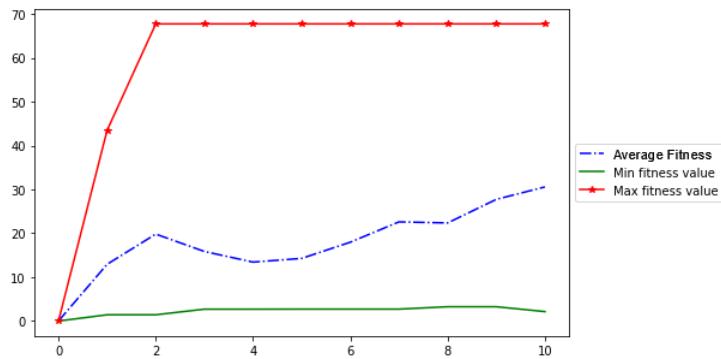
A pesar de tener mal acomodo y distribución de los elementos en el caso del peor individuo, se puede apreciar que a pesar de generar estructuras mal ordenadas es posible generar estructuras que por la manera en cómo se acomodan debido a la gravedad son capaces de mantenerse de pie, este tipo de estructuras aun siendo visualmente un acomodo "errado" de las piezas puede ser considerado como una estructura que puede ser rescatada para su uso en futuras generaciones.

A diferencia del primer experimento presentado en esta sección este es capaz de comenzar a subir el nivel de aptitud a partir de la segunda generación de individuos, a pesar de ser capaz de subir la aptitud de la primera a la segunda generación después de eso el sistema no es capaz de continuar mejorando estos resultados en este caso los resultados se quedan estancados en niveles como los que se muestran en la sub Figura 6.4(a) la cual a pesar de lograr mantener la mayor cantidad de elementos durante la simulación tiene el problema de que las piezas inician en posiciones muy elevadas y al momento de ser movidas por la gravedad pierden su posición original lo cual provoca que el valor de aptitud se vea afectado, el mejor individuo comparte el mismo problema con el segundo mejor mostrado en la sub Figura 6.4(b) donde se puede apreciar que las estructuras a pesar de estar aun de pie dado el tiempo necesario estas estructuras terminaran cayendo hacia un lado u otro, a pesar de esto este tipo de estructuras generadas permiten ver que existen compuestos en donde a pesar de mantener los elementos de pie la base de alguna de las estructuras tiene

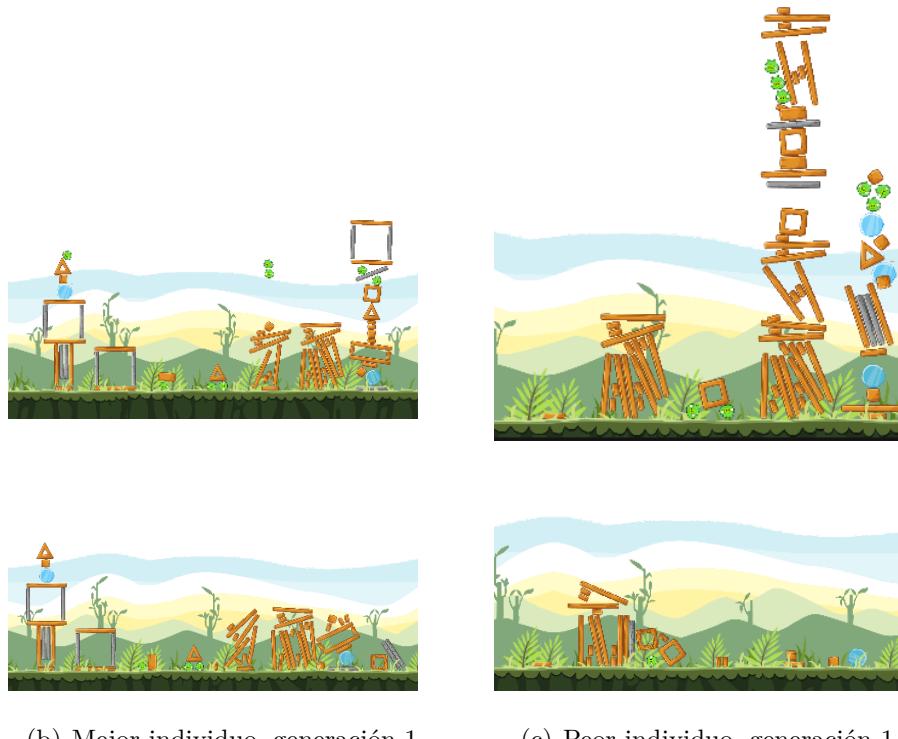
CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS

el comportamiento de una balanza debido al uso de estructuras triangulares en las cuales un pequeño cambio de peso en alguno de los lados del mismo provocara que la estructura completa termine cayendo, sin embargo, este tipo de estructuras proveen un punto de interés debido a lo visualmente complejo que pueden ser este tipo de estructuras.

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS



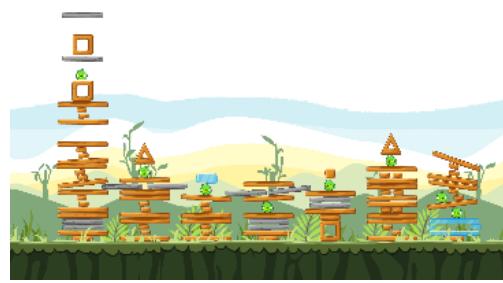
(a) Resultados de aptitud para el experimento



(b) Mejor individuo, generación 1

(c) Peor individuo, generación 1

Fig 6.3: Resultados del experimento, gráfica de la función de aptitud figura a), ejemplos de individuos: figuras b) y c)



(a) Mejor individuo, ultima generación



(b) Segundo mejor individuo, ultima generación

Fig 6.4: Resultados del experimento, primer y segundo mejor nivel generado (sin repetir)

6.1.3 Experimento 3

Un tercer grupo de resultados de experimentos se presenta en la Figura 6.5 más directamente mediante la gráfica de 6.5(a) este experimento en particular se puede apreciar tuvo un inicio no muy bueno iniciando en aproximadamente 40% de aptitud de estabilidad, esto se puede apreciar más detalladamente en la imagen 6.5(b) donde se puede ver que el mejor individuo de la primera generación inicia muy mal la simulación al tener elementos que inicial con muy mal acomodo lo cual provoca que la estructura que se había construido se desplome y muchos de los elementos que la conformaban se destruyan en el proceso, aun teniendo un mal inicio se puede apreciar dos estructuras generadas que logra salvarse, estos dos tipos de estructuras son interesantes debido a la manera en cómo están conformadas lo cual permite que se mantengan estables a pesar de que la parte superior de ambas se desplome y pudiera afectar la estabilidad de las mismas.

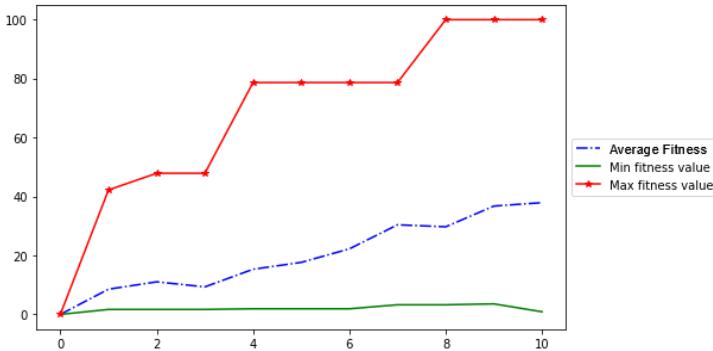
El segundo caso mostrado en 6.5(c) demuestra el problema del que se ha estado hablando de los niveles generados, pero a una más grande escala en donde la mayor parte de los componentes del nivel aparecen demasiado elevados y separados entre sí además de tener el problema del mal acomodo de los mismo lo cual provoca que terminen cayendo todos al piso y se destruyan en el proceso, aun así, como se ha estado tratando de hacer énfasis en estos resultados es que conjuntos como los que se lograron generar en este nivel no son completamente malos como los que se pueden ver en las esquinas del nivel arriba de los elementos de concreto o inclusive el conjunto de la base de la estructura central, estos elementos pueden ser buenos compuestos para futuras pruebas pero debido a la manera en cómo se comportan los niveles en la simulación son destruidos y por tal se pierden en futuras generaciones.

Este experimento se puede asemejar mucho al que se discutió en 6.1.2 en donde los resultados de aptitud de los individuos se mantenían en incrementos graduales eventualmente logrando llegar a un individuo que logra obtener el mejor promedio de las generaciones, tal es el caso del nivel presentado en 6.6(a) la cual se puede apreciar logra mantener en balance los componentes del nivel mediante un acomodo que asemeja la apilación de piezas,

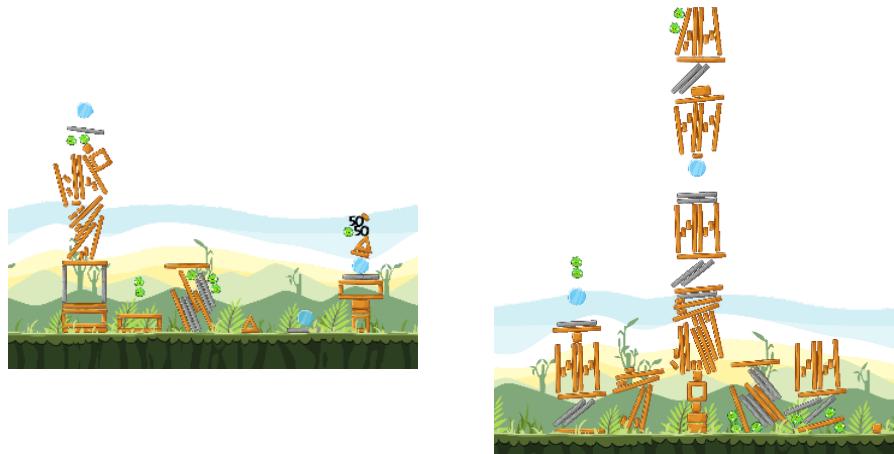
CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS

a pesar de tener una distribución de este tipo se puede observar cómo algunos elementos se mantienen en un tipo de balance en sus posiciones debido que algunas de las bases utilizan elementos demasiado chicos de igual manera que con estructuras que utilizan piezas triangulares cuando se balancean las demás estructuras estas también se pueden ver afectadas por el cambio de peso hacia alguno de los lados de los elementos que sostienen y eventualmente caer hacia los lados, sin embargo, debido a que las piezas cuadradas que utilizan en la base son más estables que las piezas triangulares entonces es un poco más complicado lograr que estos conjuntos se desplomen.

De igual manera se presenta el segundo mejor elemento de la última generación mostrado en 6.6(b), en este caso el segundo "mejor" elemento de la última generación está conformada por estructuras que en su estado inicial están mal distribuidas como las que se mostraron anteriormente en 6.5(b) de igual manera que con los casos anteriores que utilizan este tipo de elementos mal acomodados las estructuras terminan cayendo y algunos componentes terminan destruyéndose en el proceso, de igual manera que los individuos anteriormente presentados se puede apreciar un cierto conjunto de elementos que logran mantenerse en pie a pesar de los errores de distribución de los demás componentes lo cual demuestra al igual que en los ejemplos anteriores es posible generar conjuntos que pueden ser utilizados en experimentos subsecuentes para la generación de niveles.



(a) Resultados de aptitud para el experimento



(b) Mejor individuo, generación 1



(c) Peor individuo, generación 1

Fig 6.5: Resultados del experimento, gráfica de la función de aptitud figura a), ejemplos de individuos: figuras b) y c)

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS



(a) Mejor individuo, ultima generación



(b) Segundo mejor individuo, ultima generación

Fig 6.6: Resultados del experimento, primer y segundo mejor nivel generado (sin repetir)

6.2 Diversidad del contenido creado

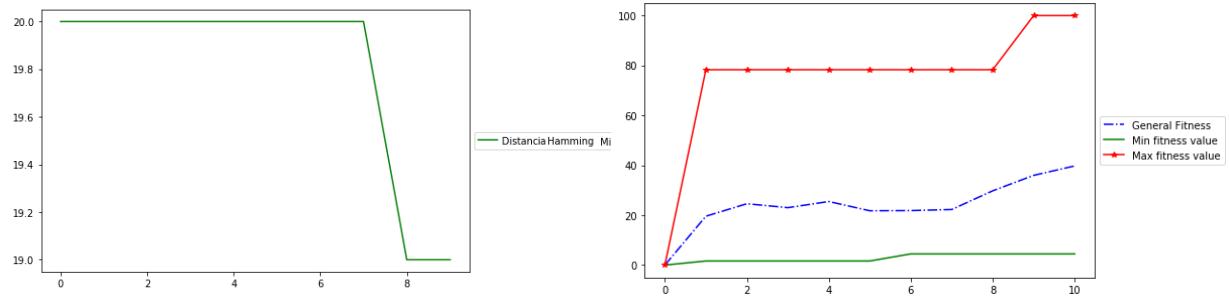
La segunda sección de resultados se enfoca en mostrar los resultados obtenidos en cuanto a las creación de diversidad de niveles dentro de los experimentos, la manera en como se trata este caso de generación es mediante el uso de un valor de aptitud en los individuos combinando los puntos establecidos en el capítulo 5 específicamente en la sección 5.4.6, esto es, los niveles generados toman en cuenta ambos aspectos de las funciones siendo que se buscan tener los individuos que logren mantener estabilidad durante las simulaciones y al mismo tiempo que logren ser diferentes a los niveles que ya se tienen, los puntos que evalúa son:

- Estabilidad
- Diferentes compuestos utilizados
- Diferencia en la estructura de los niveles

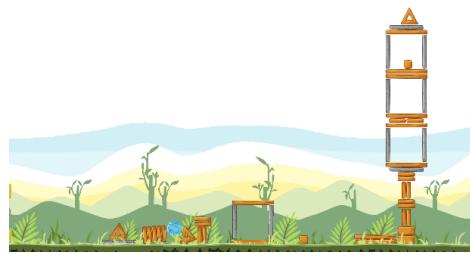
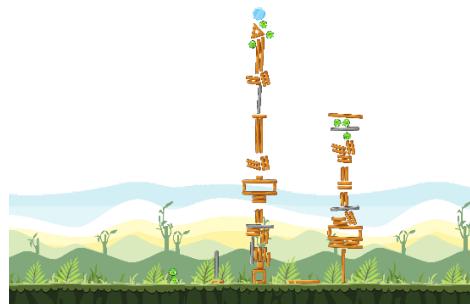
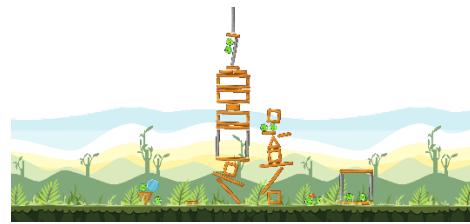
El ejemplo mostrado en la Figura 6.7 muestra un experimento en donde la primera generación crea niveles relativamente diferentes, conforme se avanza en las generaciones los niveles tienden a asemejarse unos de otros debido al uso de los miembros elite en las generaciones, sin embargo al final del experimento se logra obtener un nivel que mantiene un cierto nivel de diferencia para con los demás de la población, uno de los puntos que se debe de tratar en este aspecto es el hecho de que al no tener una facilidad de agregar compuestos a la lista de posibles compuestos utilizables se tiene el problema de que los niveles no cambian mucho de manera interna, el método de agregar compuestos nuevos a la población es mediante el uso de la mutación de los individuos, sin embargo este solo se realiza con los nuevos individuos y no con aquellos que siguen existiendo dentro de la población.

El uso de ambas funciones de fitness en la evaluación de los individuos conlleva un cierto nivel de complejidad al momento de evolucionar a los individuos, esto se debe a que los individuos tratan de ser lo mas diferentes posibles por medios de selección

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS



(a) Resultados de aptitud para el experimento



(b) Individuos de la generación 1

(c) Individuos generación 10 (no repetidos)

Fig 6.7: Evolución de los individuos de un experimento

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS

y combinación, esto provoca que los individuos obtenidos en las ultimas generaciones tengan una tendencia de ser diferentes en cuanto a la distribución de los compuestos de cada uno, pero muchas veces al tratar de ser diferentes generan puntos de inestabilidad en las estructuras generadas. De igual manera como se explico anteriormente los individuos obtenidos en estas evoluciones tienen mayor tendencia a ser similares a aquel con mejor valor de estabilidad, en este caso la tendencia de este experimento es que los elementos se mantengan en su mayoría parecidos al mostrado en la parte inferior de la Figura 6.7(c).

Así mismo esta sección de resultados se enfoca en los diferentes resultados obtenidos de diversidad de los compuestos generados, es decir, se mostrarán los diferentes compuestos que se lograron generar durante las simulaciones del sistema, el conjunto de ejemplos que se utilizaran para esto se presenta en la Figura 6.8 en donde se muestran tres grupos de agrupamientos que se lograron obtener en diferentes experimentos, en estos se puede mostrar que el sistema es capaz de generar estructuras que logran ser interesantes, innovadoras y llamativas para el usuario, de manera independiente algunas de las estructuras generadas tienen la posibilidad de caer si se les da el tiempo suficiente para que el desplazamiento de peso logre tomar efecto, sin embargo muchos de los conjuntos obtenidos pueden ser evolucionados de manera independiente de tal manera que se pueda obtener elementos como los mostrados en las Figuras 6.9 en los cuales las bases de los compuestos sean modificadas para que en vez de tener una sola base que puede ser inestable se tengan dos en ambos lados que permitan al compuesto tener la estabilidad necesaria para que no se vean tan afectadas por la gravedad del juego o por acciones del jugador al estar interactuando en el nivel.

Para estos conjuntos de resultados cabe remarcar que debido a la manera en como esta codificado el sistema los conjuntos que se presentan como los resultantes así como los niveles que terminan conformando son muy difíciles de poder replicar, es posible replicar partes que conforman las estructuras completas debido a que muchos de estos son combinaciones a veces sencillas, sin embargo, no se tiene la seguridad de que se logren replicar todas las estructuras en su totalidad, estos conjuntos obtenidos pueden a su vez ser integrados en listas extras que permitan que al iniciar experimentos subsecuentes puedan ser

CAPÍTULO 6. EXPERIMENTOS Y RESULTADOS

integrados a la lista de compuestos disponibles del experimento, como se explico anteriormente se proponía que todos los experimentos realizados iniciaran en el mismo nivel de complejidad es solamente utilizando los elementos básicos del juego y cada experimento generara compuestos para utilizar esto permitió que los niveles que se lograban obtener al final del experimento siempre fuesen diferentes entre otros experimentos realizados.



Fig 6.8: Conjunto 1 de estructuras resultantes en las simulaciones

A pesar de que los resultados que se muestran en esta sección logran demostrar que el sistema es capaz tanto de crear niveles diferentes e interesantes así como de generar compuestos de estructuras nuevas con la base de las piezas básicas del juego el sistema aun cuenta con detalles que se pueden pulir para mejorar el rendimiento del mismo, estos detalles se verán más detalladamente en la sección 7.



Fig 6.9: Conjunto 2 de estructuras resultantes en las simulaciones

Capítulo 7

Conclusiones y trabajo futuro

En las siguientes secciones se presentan las conclusiones a las que se llegaron durante el desarrollo del proyecto de tesis presentado en este documento, sobre la manera en cómo se implementaron los aspectos propuestos, así como de los resultados que se lograron obtener mediante el uso de estas propuestas. Así mismo dentro de la última sección de este capítulo se discuten y proponen diferentes métodos que pueden ser implementados a futuro para mejorar los resultados obtenidos, rutas de investigación que se teorizaron podrían funcionar así como aspectos que requieren de mejorar para un funcionamiento más fluido del sistema.

7.1 Conclusiones

Esta tesis presenta un método novedoso para la generación de niveles del juego de Angry Birds apoyándose en las mecánicas de la evolución genética y de tomando aspectos de open-ended evolution. El sistema que se logró generar cuenta con las capacidades para generar estructuras compuestas utilizando elementos básicos del juego, además que de que debido a la manera en cómo fue programado es posible modificar los módulos de clases sin requerir modificaciones enormes en el código fuente, por ejemplo, es posible realizar modificaciones a las funciones de selección y mutación que se reflejen de manera rápida en el sistema al realizar las simulaciones de igual manera el sistema de evaluación se puede

CAPÍTULO 7. CONCLUSIONES Y TRABAJO FUTURO

mejorar ya sea cambiando las métricas de evaluación propuestas o agregando nuevas según sea requerido.

Para fines de este proyecto el sistema fue probado con el simulador de Science Birds utilizando todos los puntos propuestos en la sección 4 y se obtuvieron resultados como los mostrados en la sección 6, sin embargo, una de las ideas en el desarrollo de este proyecto es el de permitir la adaptabilidad en el sentido de que se espera que siempre y cuando un caso de estudio particular es decir un juego diferente tenga niveles que puedan ser descompuestos hasta las expresiones mínimas, es decir los bloques de construcción que permitirán generar un nivel será posible en teoría utilizar este mismo sistema para generar niveles en otros juegos simplemente ajustando algunas variables que permitan comparar los compuestos a fin de obtener los resultados esperados.

La implementación mostrada en la sección 4 muestra cómo se explicó anteriormente las ideas que se consideran fueron las mejores para desarrollar el sistema, sin embargo, aquellas mostradas como propuestas anteriores demuestran los cambios por los cuales el sistema tuvo que pasar para poder obtener lo que consideramos como el "mejor" en términos de lo que se proponía lograr, así como estos mismos métodos podían aproximar resultados a los obtenidos en el sistema final pueden existir diferentes técnicas que no se exploraron a detalle que logren obtener resultados iguales o inclusive mejores, ejemplos como los presentados en la sección 3 demuestran que diferentes personas lograron tener diferentes métodos de atacar esta problemática, desde el uso de sistemas inteligentes o sistemas auto-adaptables, mediante el uso de estas diferentes técnicas se observa una gran diversidad de resultados, en el caso de este sistema los métodos que se opina pueden sustituir al algoritmo genético son técnicas basadas en búsquedas meta-heurísticas debido a la naturaleza del sistema.

Uno de los puntos tratados durante los capítulos 5 y 6 es el problema relacionado con el tiempo de simulación, debido a que los niveles generados requieren ser evaluados dentro de los parámetros del juego como tal se requiere que el juego se ejecute y evalúe cada nivel uno a la vez, esto provoca que los tiempos de simulación sean tardados debido a que cada nivel requiere ser simulado durante 10 segundos con el fin de que las estructuras

generadas tengan tiempo de reaccionar bajo la gravedad, si están mal acomodadas caerán y si están correctamente acomodadas se mantendrán de pie o balanceadas además de este tiempo de simulación se toma también en cuenta el tiempo que tarda en escribir los resultados de cada nivel en sus respectivos archivos eh inclusive el tiempo que el software de simulación tarda en ser ejecutado, este punto fue uno de los obstáculos más grandes en el sistema debido a que solo analizar si un cambio funcionaba bien podía tardar mucho, sin embargo, este tiempo se reduce considerablemente cuando se trabaja en generaciones diferentes a la inicial debido a que en generaciones subsecuentes solo se programó que simulara aquellos individuos que fueron generados, gracias a que este fue un obstáculo que se tuvo que mantener en el sistema se lograron adaptar partes del sistema para realizar las simulaciones de manera más fluida y no requerir de volver a ejecutar una simulación para analizar el flujo de los datos lo cual permitió encontrar errores más rápidamente.

7.2 Trabajo futuro

Las conclusiones descritas previamente proveen dan un entendimiento de los obstáculos que por los cuales se tuvo que pasar para poder completar el proyecto, el problema y posible solución tratados en las conclusiones sobre las simulaciones generadas es el tiempo desperdiciado debido al uso del software de simulación, debido a que las simulaciones requieren de este software para poder obtener los resultados es necesario encontrar una manera en la cual se pueda estimar mediante el uso de fórmulas las posiciones en las cuales terminaran los elementos del nivel, considerando que el juego como tal tiene un sistema de gravedad que provoca que las piezas caigan al suelo debería de ser posible calcular u obtener el valor de gravedad utilizada para desarrollar un sistema capaz de calcular el movimiento de las piezas considerando también la interacción de las piezas unas con otras, así como las resistencias con las que cuentan.

El ámbito de la evaluación de los individuos es uno de los principales que puede ser mejorado o modificado, esto es debido a que las funciones que se utilizaron para la evaluación en esta tesis son las que se consideraron óptimas para la problemática tratada, de

CAPÍTULO 7. CONCLUSIONES Y TRABAJO FUTURO

acuerdo a lo que se requiera obtener es posible modificar la manera en cómo se obtienen los resultados de aptitud de los individuos, en el caso de lo que se utilizó para el desarrollo del proyecto es posible modificar los cálculos de los resultados de estabilidad, en este caso lo que se esperaba obtener es el valor más cercano a 100 para cada individuo en donde 100 representaba que ninguno de los elementos que conformaban el nivel se destruyeron o cayeron, sin embargo, es posible utilizar algunas otras funciones o cálculos para poder obtener estos resultados inclusive cambiar la manera de interpretar la estabilidad como el uso de cálculos de aceleración promedio o movimiento angular de las mismas piezas.

Uno de los ámbitos que también requiere ser tratado es el de la estabilidad de los niveles generados, es decir el sistema logra generar niveles mediante el acomodo de los compuestos que se tienen registrados, sin embargo, una de las maneras en cómo el proceso de evaluación, así como el de simulación puede ser optimizado un poco mas es mediante el uso de reglas de generación o revisión de patrones en el ordenamiento de los compuestos en los niveles para poder detectar áreas en donde se pueda estimar que durante la simulación ciertas estructuras mal acomodadas debido a los componentes básicos que utilizan puedan causar que los niveles tengan mal rendimiento, es decir, analizar casos en donde una pieza en particular seguida de otras desfazadas en sus coordenadas x o y provoquen que los compuestos se caigan, esto podría ser analizado y prevenido mediante el uso de diferentes máscaras o en caso de que no sea posible evitarlo retirar los elementos de la simulación a fin de requerir menos tiempo.

Otro de los factores que pueden ser mejorados es el uso de la lógica detrás de open-ended evolution, debido a que la consideración de este algoritmo es de tener una evolución casi infinita y que los niveles del juego se generan una sola vez al inicio de la evolución realmente no es muy requerido generar nuevos compuestos de manera continua, pero es posible desarrollar un sistema de evolución que utilice compuestos resultantes como los mostrados en la sección 6 y los integre en una nueva evolución para combinarlos con componentes básicos e inclusive consigo mismos para generar nuevos compuestos más complejos que puedan ser integrados a futuras simulaciones.

Finalmente, un último aspecto que se debe de tomar en cuenta es el número de experi-

CAPÍTULO 7. CONCLUSIONES Y TRABAJO FUTURO

mentos utilizados, como se explicó anteriormente el sistema requería de mucho tiempo para realizar las simulaciones requeridas por los individuos, esto inhibió la capacidad de generar una gran cantidad de simulaciones para realizar comparaciones, sin embargo, con la cantidad actual de resultados obtenidos es posible validar las capacidades del generador, sin embargo, se requiere realizar aun más simulaciones utilizando diferentes configuraciones del algoritmo con el fin de determinar que el sistema es capaz de trabajar correctamente no solo con el conjunto de configuraciones determinado en la sección de resultados.

Bibliografia

- [1] Rovio Entertainment Corporation, “Angry Birds,” 2009.
- [2] L. Mossmouth, “Spelunky World.” [Online]: <https://www.spelunkyworld.com/>, 2013.
- [3] M. Mateas and A. Stern, “Façade: An Experiment in Building a Fully-Realized Interactive Drama,” tech. rep., 2005.
- [4] N. I. Holtar, “AUDIOVERDRIVE: EXPLORING BIDIRECTIONAL RELATIONSHIPS BETWEEN MUSIC AND GAME,” tech. rep., 2013.
- [5] S. Risi, J. Lehman, D. B. D’ambrosio, R. Hall, and K. O. Stanley, “Combining Search-based Procedural Content Generation and Social Gaming in the Petalz Video Game,” tech. rep., 2012.
- [6] J. H. Holland, “Adaptation in Natural and Artificial Systems,” *Sgart Newsletter*, 1975.
- [7] D. C. S. U. Whitley, “A Genetic Algorithm Tutorial,” *Statistics and Computing*, no. 4, pp. 65–85, 1994.
- [8] G. R. Harik, F. G. Lobo, and D. E. Goldberg, “The Compact Genetic Algorithm,” Tech. Rep. 4, 1999.
- [9] D. Andrew, “Spelunky.” [Online]: <http://pcg.wikidot.com/pcg-games:spelunky>, 2009.
- [10] A. Burch, “Infinite Caves, Infinite Stories.” [Online Article]: https://v1.escapistmagazine.com/articles/view/video-games/issues/issue{_}209/6235-Infinite-Caves-Infinite-Stories, 2009.
- [11] M. Persson, “The Word of Notch — Terrain generation, Part 1.” [Online]: <https://notch.tumblr.com/post/3746989361/terrain-generation-part-1>, 2011.
- [12] D. Andrew, “Minecraft.” [Online]: <http://pcg.wikidot.com/pcg-games:minecraft>, 2008.

BIBLIOGRAFIA

- [13] K. Perlin, “An image synthesizer,” in *Tutorial: computer graphics; image synthesis*, pp. 333–342, Computer Science Press, Inc., 1988.
- [14] D. Andrew, “Procedural content generation wiki.” [Online]: <http://pcg.wikidot.com/pcg-algorithm:whittaker-diagram>, 2008.
- [15] D. Andrew, “Disgaea.” <http://pcg.wikidot.com/pcg-games:disgaea>, 2008.
- [16] D. Andrew, “Runtime Random Level Generation.” [Online]: <http://pcg.wikidot.com/pcg-algorithm:runtime-random-level-generation>, 2008.
- [17] E. Hastings, R. Guha, and K. Stanley, “Automatic Content Generation in the Galactic Arms Race Video Game,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 1, no. 4, pp. 245–263, 2009.
- [18] T. Garner and M. Grimshaw, “Sonic virtuality: Understanding audio in a virtual world,” *The Oxford Handbook of Virtuality*, p. 364, 2014.
- [19] P. Lopes, A. Liapis, and G. N. Yannakakis, “Sonancia: Sonification of procedurally generated game levels,” in *In Proceedings of the 1st Computational Creativity and Games Workshop*, 2015.
- [20] D. Fitterer, “Audiosurf.” [Online]: <http://www.audio-surf.com/>, 2010.
- [21] D. L. Brown, “Mezzo: An adaptive, real-time composition program for game soundtracks,” in *Eighth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2012.
- [22] M. Mateas and A. Stern, “Façade: An experiment in building a fully-realized interactive drama,” in *Game developers conference*, vol. 2, pp. 4–8, 2003.
- [23] R. Evans and E. Short, “Versu—a simulationist storytelling system,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 6, no. 2, pp. 113–130, 2013.
- [24] R. DeMaria and J. L. Wilson, *High score! : the illustrated history of electronic games*. McGraw-Hill/Osborne, 2004.
- [25] *Super Mario Bros. Instruction Booklet*. USA: Nintendo of America, 1985.
- [26] “Super Mario World - Videogame by Nintendo,” *The International Arcade Museum*, 1995.
- [27] *Super Mario World Instruction Booklet*. Nintendo of America, 1991.
- [28] C. Browne and F. Maire, “Evolutionary Game Design,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 2, no. 1, pp. 1–16, 2010.

BIBLIOGRAFIA

- [29] J. Togelius and J. Schmidhuber, “An experiment in automatic game design,” in *2008 IEEE Symposium On Computational Intelligence and Games*, pp. 111–118, IEEE, 2008.
- [30] T. S. Nielsen, G. A. B. Barros, J. Togelius, and M. J. Nelson, “Towards generating arcade game rules with VGDL,” in *2015 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 185–192, IEEE, 2015.
- [31] M. Treanor, B. Blackford, M. Mateas, and I. Bogost, “Game-o-matic: Generating videogames that represent ideas,” in *Proceedings of the The third workshop on Procedural Content Generation in Games*, p. 11, ACM, 2012.
- [32] M. J. Nelson and M. Mateas, “An interactive game-design assistant,” in *Proceedings of the 13th international conference on Intelligent user interfaces - IUI '08*, (New York, New York, USA), p. 90, ACM Press, 2008.
- [33] M. Cook and S. Colton, “Multi-faceted evolution of simple arcade games,” in *2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*, pp. 289–296, IEEE.
- [34] J. G. Sánchez, N. P. Zea, and F. L. Gutiérrez, “Playability: How to identify the player experience in a video game,” in *IFIP Conference on Human-Computer Interaction*, pp. 356–359, Springer, 2009.
- [35] T. Taylor, M. Bedau, A. Channon, D. Ackley, W. Banzhaf, G. Beslon, E. Dolson, T. Froese, S. Hickinbotham, T. Ikegami, B. McMullin, N. Packard, S. Rasmussen, N. Virgo, E. Agmon, E. Clark, S. McGregor, C. Ofria, G. Ropella, L. Spector, K. O. Stanley, A. Stanton, C. Timperley, A. Vostinar, and M. Wiser, “Open-Ended Evolution: Perspectives from the OEE Workshop in York,” *Artificial Life*, vol. 22, no. 3, pp. 408–423, 2016.
- [36] J. Renz, X. Ge, R. Verma, and P. Zhang, “Angry Birds as a Challenge for Artificial Intelligence,” *Thirtieth AAAI Conference on Artificial Intelligence*, 2016.
- [37] M. Stephenson, J. Renz, X. Ge, and P. Zhang, “The 2017 aibirds competition,” *CoRR*, vol. abs/1803.05156, 2018.
- [38] M. J. B. Stephenson, J. Renz, X. Ge, L. N. Ferreira, J. Togelius, and P. Zhang, “The 2017 AIBIRDS Level Generation Competition,” *IEEE Transactions on Games*, pp. 275–284, 2018.
- [39] L. Ferreira and C. Toledo, *A search-based approach for generating Angry Birds levels*. IEEE Conference on Computational Intelligence and Games, 2014.
- [40] Y. Jiang, T. Harada, and R. Thawonmas, *Procedural generation of angry birds fun levels using pattern-struct and preset-model*. IEEE Conference on Computational Intelligence and Games, 2017.

BIBLIOGRAFIA

- [41] M. Kaidan, C. Y. Chu, T. Harada, and R. Thawonmas, “Procedural generation of angry birds levels that adapt to the player’s skills using genetic algorithm,” in *2015 IEEE 4th Global Conference on Consumer Electronics (GCCE)*, pp. 535–536, IEEE, 2015.
- [42] M. Kaidan, T. Harada, C. Y. Chu, and R. Thawonmas, “Procedural generation of angry birds levels with adjustable difficulty,” in *2016 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1311–1316, IEEE, 2016.
- [43] J. F. ALLEN, “Maintaining Knowledge about Temporal Intervals,” *Readings in Qualitative Reasoning About Physical Systems*, pp. 361–372, 1990.
- [44] L. Calle, J. J. Merelo, A. Mora-García, and J.-M. García-Valdez, “Free Form Evolution for Angry Birds Level Generation,” pp. 125–140, Springer, Cham, 2019.
- [45] F. Calimeri, M. Fink, S. Germano, A. Humenberger, G. Ianni, C. Redl, D. Stepanova, A. Tucci, and A. Wimmer, “Angry-HEX: An Artificial Player for Angry Birds Based on Declarative Knowledge Bases,” *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 8, no. 2, pp. 128–139, 2016.
- [46] G. Smith, M. Treanor, J. Whitehead, and M. Mateas, “Rhythm-Based Level Generation for 2D Platformers,” tech. rep., 2009.
- [47] J. Renz and M. Stephenson, “AI Birds.org -Level Generation Competition,” 2013.

Appendix A

Anexos

A.1 Código del sistema

Código principal

```
1  from __future__ import division, absolute_import
2  # to avoid integer devision problem
3  import datetime
4  import scipy
5  import pylab
6  #
7  import random
8  import math
9  #
10 import datetime
11 import time
12 from yaspin.yaspin import yaspin
13 import os
14 import json
15 import sys
16 import subprocess
17 import itertools
18 import operator
19 import shutil
20 import numpy as np
21 from copy import deepcopy
22
```

APPENDIX A. ANEXOS

```
23     # Local files
24     import XmlHelpers as xml
25     import Evaluation as Eval
26     from Selection import Selection
27     from Mutation import Mutation
28
29     author = "Salinas Hernandez Jaime"
30     copyright = "Copyright 2018, Tijuana Institute of Technology"
31     credits = ["Dr. Mario García Valdez", ""]
32     license = "ITT"
33     version = "1.4.1"
34     date = "May 08, 2019 18:30"
35     maintainer = "Salinas Hernandez Jaime"
36     email = "jaime.salinas@tectijuana.edu.mx"
37     status = "Development"
38     t_begin = datetime.datetime.now()
39     t_prev = datetime.datetime.now()
40
41
42     ## Values used for the genetic algorithm
43     population = 10          # For now it can only be below 10
44     max_gen = 100            # Max number of generations
45     fits = [0]                # Variable to save the fitness of each generation
46     gen = 0                  # Generation 1
47     per_cross = 0.5          # Percentage of cross-over (cross-over operator)
48     per_comb = 0.3           # Percentage of combination (combination operator)
49     per_mut = 0.8             # Percentage of mutation
50     sel_type = 1              # Selection (for crossover) type [ 0: Random - 1: Tournament - TBD]
51     cross_type = 0            # Type of cross-over [ 0: One point CO - 1: Random point CO - TBD]
52     ind_pieces = 20           # Number of pieces that define an individual
53     all_fit = [0]              # Average fitness for all generations
54     fit_ham = []               # Average piece length fitness for all generations
55     fit_Amov = []              # Average movement fitness for all generations
56     max_elite = 1                # Maximum number of elite members in the generation
57     elite = []
58     best_gen = []
59     mu = 0                    # Mean for normal distribution
60     sigma = 20                 # SD for normal distribution
61
62     min_fit = [0]              # Average fitness for all generations
63     max_fit = [0]                # Average fitness for all generations
64
65     ## Data required to create the xml files
```

APPENDIX A. ANEXOS

```

66
67 #####< Data required to create the xml files >#####
68 #####
69 #####
70
71 project_root = os.getcwd()
72 json_data = {}
73 json_data['all_fit'] = []
74 json_data['hamming_avg'] = []
75 json_data['min_fit'] = []
76 json_data['max_fit'] = []
77 ruleset = open("Ruleset/parameters.txt", "r")
78 config_param = json.loads(open("ga_parameters.json", "r").read())
79
80 game_path = config_param['game_path']
81 write_path = config_param['write_path']
82 elite_path = config_param['elite_path']
83 read_path = config_param['read_path']
84 log_path = config_param['log_dir']
85 log_base_name = config_param['log_base_name']
86 child_level = config_param['child_level']
87 child_output = config_param['child_output']
88
89 # For tournament
90 game_path_tourney = config_param['game_path_tourney']
91 write_path_tourney = config_param['write_path_tourney']
92 read_path_tourney = config_param['read_path_tourney']
93
94 # To have a copy of the generations
95 level_files_path = config_param['level_files']
96 current_gen_folder = config_param['generation']
97 sourcefolder = os.path.join(project_root, level_files_path + "/Levels")
98 experimentsource = os.path.join(project_root, level_files_path)
99 foldername = ''.join(e for e in str(datetime.datetime.now()) if e.isalnum())
100 experimentdest = config_param['experiment_results'] + "/" + foldername
101 experimentdestination = os.path.join(project_root, experimentdest)
102
103 # Data for the competition
104 req_levels = int(deepcopy(ruleset.readline()))
105 req_np_combinations = ruleset.readline().split(',')
106 for i, e in enumerate(req_np_combinations):
107     req_np_combinations[i] = req_np_combinations[i].split()
108 req_pigs = ruleset.readline().split(',')

```

APPENDIX A. ANEXOS

```
109
110     max_elite = req_levels
111
112     # Clean previous experiment
113     #shutil.rmtree(os.path.join(project_root, write_path))
114     #shutil.rmtree(os.path.join(project_root, elite_path))
115     shutil.rmtree(os.path.join(project_root, level_files_path))
116
117     os.makedirs(os.path.join(project_root, level_files_path), exist_ok=True)
118
119     os.makedirs(os.path.join(project_root, write_path), exist_ok=True)
120     os.makedirs(os.path.join(project_root, elite_path), exist_ok=True)
121     os.makedirs(os.path.join(project_root, read_path), exist_ok=True)
122
123     os.makedirs(os.path.join(project_root, log_path), exist_ok=True)
124
125
126
127     ## "Dictionary" to save the base pieces and structures
128     SW_HIDE = 0
129     info = subprocess.STARTUPINFO()
130     info.dwFlags = subprocess.STARTF_USESHOWWINDOW
131     info.wShowWindow = SW_HIDE
132
133     #just for fun making further development easier and with joy
134     pi      = scipy.pi
135     dot     = scipy.dot
136     sin     = scipy.sin
137     cos     = scipy.cos
138     ar      = scipy.array
139     rand    = scipy.rand
140     arange  = scipy.arange
141     plot    = pylab.plot
142     show    = pylab.show
143     axis    = pylab.axis
144     grid    = pylab.grid
145     title   = pylab.title
146     rad     = lambda ang: ang*pi/180                      #lovely lambda: degree to radian
147
148
149
150     # Function to rotate points
151     def Rotate2D(pts,cnt,ang=pi/4):
```

APPENDIX A. ANEXOS

```

152     '''pts = {} Rotates points(nx2) about center cnt(2) by angle ang(1) in radian'''
153     return dot(pts-cnt,ar([[cos(ang),sin(ang)],[-sin(ang),cos(ang)]]))+cnt
154
155
156     # Generation masks
157     type_small_1 = [[0,0,0,0,0,0,0],  

158                     [0,0,2,0,0,0,0],  

159                     [0,0,1,1,0,0,0]]  

160     type_large_1 = [[0,0,3,0,0,0,0],  

161                     [0,0,2,0,0,0,0],  

162                     [0,0,1,1,0,0,0]]  

163     type_small_cube = [[0,0,0,0,0,0,0],  

164                     [0,0,2,2,0,0,0],  

165                     [0,0,1,1,0,0,0]]  

166     type_large_cube = [[0,3,3,3,0,0,0],  

167                     [0,2,2,2,0,0,0],  

168                     [0,1,1,1,0,0,0]]  

169     type_small_floor = [[0,0,0,0,0,0,0],  

170                     [0,0,0,0,0,0,0],  

171                     [0,1,1,1,1,1,0]]  

172     type_large_floor = [[0,0,0,0,0,0,0],  

173                     [0,0,0,0,0,0,0],  

174                     [1,1,1,1,1,1,1]]  

175     type_castle = [[0,0,0,3,0,0,0],  

176                     [2,0,2,2,2,0,2],  

177                     [1,1,1,1,1,1,1]]  

178     type_house = [[0,0,3,3,0,0,0],  

179                     [0,2,2,2,2,0,0],  

180                     [0,1,1,1,1,0,0]]  

181     type_towers = [[3,0,3,0,3,0,0],  

182                     [2,0,2,0,2,0,0],  

183                     [1,0,1,0,1,0,0]]  

184
185     Mask_List = [  

186         type_small_1,  

187         type_large_1,  

188         type_small_cube,  

189         type_large_cube,  

190         type_small_floor,  

191         type_large_floor,  

192         type_castle,  

193         type_house,  

194         type_towers

```

APPENDIX A. ANEXOS

```
195     ]
196
197     # Global Piece class and a class for each piece in the game
198     class Piece:
199         def __init__(self, x, y, r, mat):
200             #self.Height = 72
201             #self.Width = 72
202             self.Material = mat
203             self.Dict = []
204             self.Valid = True
205             self.X = x
206             self.Y = y
207             self.R = r
208             if self.Materials[0] == 0:
209                 self.Material = "stone"
210             if self.Materials[1] == 0:
211                 self.Material = "ice"
212
213
214         def get_edges(self):
215             ed_lis = []
216             ed_lis.append([self.Width/2 +      self.X,      self.Height/2 + self.Y])
217             ed_lis.append([self.Width/2 +      self.X, 0 - self.Height/2 + self.Y])
218             ed_lis.append([0 - self.Width/2 + self.X, 0 - self.Height/2 + self.Y])
219             ed_lis.append([0 - self.Width/2 + self.X,      self.Height/2 + self.Y])
220             self.Edges = ed_lis
221
222         def get_points(self, r):
223             ang = self.R * pi / 180
224             ots = Rotate2D(self.Edges, ar([0 + self.X, 0 + self.Y]), ang)
225             self.Points = ots.tolist()
226             #print(self.Points)
227
228         def change_material(self, m):
229             self.Material = m
230
231         def as_dictionary(self):
232             self_list = []
233             self_list.append(self.Name)
234             self_list.append(self.Material)
235             self_list.append(self.X)
236             self_list.append(self.Y)
237             self_list.append(self.R)
```

APPENDIX A. ANEXOS

```
238         self.Dict = self_list
239
240     def update_values(self):
241         #if 'errormessage' in kwargs:
242         #    print("llegue")
243         #else:
244         #    self.Material = kwargs.get('material')
245         #self.Material = "ice"
246         self.get_edges()
247         self.get_points(self.R)
248         self.as_dictionary()
249
250     class Circle(Piece):
251         def __init__(self, mat, x=0, y=0, r=0):
252             self.Name = "Circle"
253             self.Height = 75
254             self.Width = 75
255             Piece.__init__(self, x, y, r, mat)
256             self.update_values()
257
258     class RectTiny(Piece):
259         def __init__(self, mat, x=0, y=0, r=0):
260             self.Name = "RectTiny"
261             self.Height = 25
262             self.Width = 45
263             Piece.__init__(self, x, y, r, mat)
264             self.update_values()
265
266
267     class RectSmall(Piece):
268         def __init__(self, mat, x=0, y=0, r=0):
269             self.Name = "RectSmall"
270             self.Height = 25
271             self.Width = 85
272             Piece.__init__(self, x, y, r, mat)
273             self.update_values()
274
275
276     class RectMedium(Piece):
277         def __init__(self, mat, x=0, y=0, r=0):
278             self.Name = "RectMedium"
279             self.Height = 25
280             self.Width = 165
```

APPENDIX A. ANEXOS

```
281     Piece.__init__(self, x, y, r, mat)
282     self.update_values()
283
284
285     class RectBig(Piece):
286         def __init__(self, mat, x=0, y=0, r=0):
287             self.Name = "RectBig"
288             self.Height = 25
289             self.Width = 185
290             Piece.__init__(self, x, y, r, mat)
291             self.update_values()
292
293
294     class RectFat(Piece):
295         def __init__(self, mat, x=0, y=0, r=0):
296             self.Name = "RectFat"
297             self.Height = 45
298             self.Width = 85
299             Piece.__init__(self, x, y, r, mat)
300             self.update_values()
301
302
303     class SquareTiny(Piece):
304         def __init__(self, mat, x=0, y=0, r=0):
305             self.Name = "SquareTiny"
306             self.Height = 22
307             self.Width = 22
308             Piece.__init__(self, x, y, r, mat)
309             self.update_values()
310
311
312     class SquareSmall(Piece):
313         def __init__(self, mat, x=0, y=0, r=0):
314             self.Name = "SquareSmall"
315             self.Height = 45
316             self.Width = 45
317             Piece.__init__(self, x, y, r, mat)
318             self.update_values()
319
320
321     class Triangle(Piece):
322         def __init__(self, mat, x=0, y=0, r=0):
323             self.Name = "Triangle"
```

APPENDIX A. ANEXOS

```
324         self.Height = 75
325         self.Width = 75
326         Piece.__init__(self, x, y, r, mat)
327         self.update_values()
328
329
330     class TriangleHole(Piece):
331         def __init__(self, mat, x=0, y=0, r=0):
332             self.Name = "TriangleHole"
333             self.Height = 85
334             self.Width = 85
335             Piece.__init__(self, x, y, r, mat)
336             self.update_values()
337
338
339     class SquareHole(Piece):
340         def __init__(self, mat, x=0, y=0, r=0):
341             self.Name = "SquareHole"
342             self.Height = 85
343             self.Width = 85
344             Piece.__init__(self, x, y, r, mat)
345             self.update_values()
346
347
348     # Composite class to control the generation of the individuals
349     class Composite:
350
351         bl_list_x = []
352         bl_list_y = []
353
354         def __init__(self, blocks):
355             # Blocks must be a list
356             self.Objetos2 = []
357             self.bl_list_x = []
358             self.bl_list_y = []
359             self.blocks = blocks.copy()
360             self.Objetos = [clases[clase](m,x,y,r) for (clase,x,y,r,m) in
361                             self.blocks.copy()]
362             self.get_values()
363             self.height = self.height()
364             self.width = self.width()
365             self.top_center = self.gen_top_center()
366             self.as_dictionary = self.gen_dictionary()
```

APPENDIX A. ANEXOS

```
367         self.low_center = self.get_low_center()
368         self.overtop_center = self.get_overtop()
369
370     def get_values(self):
371         if len(self.blocks) > 2:
372             self.Pig = True
373         else:
374             self.Pig = False
375         self.Borders = [[(x,y) for (x,y) in Obj.Points] for Obj in self.Objetos]
376         self.Borders = sum(self.Borders, [])
377         self.Width = abs(min(self.Borders, key=lambda t:t[0])[0] - max(self.Borders,
378                         key=lambda t:t[0])[0])
379         self.Height = abs(min(self.Borders, key=lambda t:t[1])[1] - max(self.Borders,
380                         key=lambda t:t[1])[1])
381         return 0
382
383     def height(self):
384         return 0.0
385
386     def width(self):
387         return 0.0
388
389     def set_borders(self):
390
391         return 0.0
392
393     def gen_top_center(self):
394         height_center = self.Height/2
395         lenght_center = 0
396         return [lenght_center, height_center]
397
398     def get_low_center(self):
399         return [0, self.Height/2]
400
401     def get_overtop(self):
402         return [0, self.Height]
403
404     def gen_dictionary(self):
405         block_list = []
406         block_list.append([self.Width, self.Height])
407         # Width represents the x coordinate and Height the y one
408         for piece in self.Objetos:
409             block_comp = []
```

APPENDIX A. ANEXOS

```
410         block_comp.append(piece.Dict[0])
411         block_comp.append(piece.Dict[1])
412         block_comp.append(piece.Dict[2])
413         block_comp.append(piece.Dict[3])
414         block_comp.append(piece.Dict[4])
415         block_list.append(block_comp)
416 #block_list.append(block_comp)
417 #print(block_list)
418 return block_list
419
420 def move_xy(self, n_x, n_y, mat_f, el_r):
421     #self.X = n_x
422     #self.Y = n_y
423     self.Objetos = [clases[clase](mat_f, x + n_x, y + n_y, el_r) for
424                     (clase,x,y,r, mat) in self.blocks.copy()]
425     return 0
426
427 def as_json(self):
428     return {}
429
430
431 clases = {
432     "Circle":Circle,
433     "RectTiny":RectTiny,
434     "RectSmall":RectSmall,
435     "RectMedium":RectMedium,
436     "RectBig":RectBig,
437     "RectFat":RectFat,
438     "SquareTiny":SquareTiny,
439     "SquareSmall":SquareSmall,
440     "SquareHole":SquareHole,
441     "Triangle":Triangle,
442     "TriangleHole":TriangleHole }
443
444 clases1 = {
445     "Circle":Circle
446 }
447
448 Composites = {
449     0: [("RectTiny", 0, 0, 0, "wood")],
450     1: [("RectSmall", 0, 0, 0, "wood")],
451     2: [("RectMedium", 0, 0, 0, "wood")],
452     3: [("RectBig", 0, 0, 0, "wood")],
```

APPENDIX A. ANEXOS

```

453     4: [("RectFat", 0, 0, 0, "wood")],
454     5: [("SquareSmall", 0, 0, 0, "wood")],
455     6: [("SquareHole", 0, 0, 0, "wood")],
456     7: [("Circle", 0, 0, 0, "wood")],
457     8: [("TriangleHole", 0, 0, 0, "wood")]
458 }
459
460 Composites_res = {
461     0: [("RectTiny", 0, 0, 0, "wood", True)],
462     1: [("RectSmall", 0, 0, 0, "wood", True)],
463     2: [("RectMedium", 0, 0, 0, "wood", True)],
464     3: [("RectBig", 0, 0, 0, "wood", True)],
465     4: [("RectFat", 0, 0, 0, "wood", True)],
466     5: [("SquareSmall", 0, 0, 0, "wood", True)],
467     6: [("SquareHole", 0, 0, 0, "wood", True)],
468     7: [("Circle", 0, 0, 0, "wood", True)],
469     8: [("TriangleHole", 0, 0, 0, "wood", True)]
470 }
471
472 materials = {
473     "wood": 0,
474     "stone": 1,
475     "ice": 2}
476
477 restrictions = {
478     "Circle": [1,1,1],
479     "CircleSmall": [1,1,1],
480     "RectTiny": [1,1,1],
481     "RectSmall": [1,1,1],
482     "RectMedium": [1,1,1],
483     "RectBig": [1,1,1],
484     "RectFat": [1,1,1],
485     "SquareTiny": [1,1,1],
486     "SquareSmall": [1,1,1],
487     "SquareHole": [1,1,1],
488     "Triangle": [1,1,1],
489     "TriangleHole": [1,1,1] }
490
491 for element in req_np_combinations:
492     restrictions[element[1]][materials[element[0]]] = 0
493
494 for piece in Composites_res:
495     clases[Composites_res[piece][0][0]].Materials = restrictions[Composites_res[piece][0][0]]

```

APPENDIX A. ANEXOS

```
496
497     for piece in Composites_res:
498         if sum(clases[Composites_res[piece][0][0]].Materials) == 0:
499             clases[Composites_res[piece][0][0]].Valid = False
500         else:
501             clases[Composites_res[piece][0][0]].Valid = True
502             #if clases[Composites_res[piece][0][0]].Materials[0] == 0:
503
504             #if sum(piece.Materials) == 0:
505                 #piece.Valid = False
506
507     def get_random_chrom(sl):
508         asl = 0
509         chrom = []
510         while asl < sl:
511             prop = random.randint(0, len(Composites)-1)
512             if clases[Composites[prop][0][0]].Valid == True:
513                 chrom.append(prop)
514                 asl += 1
515             #random.randint(0, len(Composites)-1) for p in range(ind_pieces)
516
517     return chrom
518
519
520     def combine_pieces():
521         piece1 = random.randint(0, len(Composites)-1)
522         piece2 = random.randint(0, len(Composites)-1)
523         new_value = len(Composites)
524         prop1 = Composite(Composites[piece1])
525         prop2 = Composite(Composites[piece2])
526         final_group = []
527         if piece1 == piece2:
528             initial = random.randint(0,1)
529
530             if initial == 0: # Estaran pegados
531                 # Checar si los bordes coinciden
532                 #print("Igual pegados")
533                 prop1.move_xy(prop1.Width/2,0,prop1.Objetos[0].Material, 0)
534                 prop2.move_xy((0-(prop2.Width/2)),0,prop2.Objetos[0].Material, 0)
535                 prop1.get_values()
536                 prop2.get_values()
537                 list1 = prop1.Borders
538                 list2 = prop2.Borders
539                 # Calculate center point of each piece and add to list
540                 test=Composites[piece1].copy()
541                 for comp in test:
```

APPENDIX A. ANEXOS

```

539         temp = []
540         temp.append(comp[0])
541         temp.append(int(comp[1]) + (prop1.Width/2))
542         temp.append(int(comp[2]))
543         temp.append(int(comp[3]))
544         temp.append(comp[4])
545         fin = tuple(temp)
546         final_group.append(fin)
547     test = Composites[piece2].copy()
548     for comp in test:
549         temp = []
550         temp.append(comp[0])
551         temp.append(int(comp[1]) - (prop2.Width/2))
552         temp.append(int(comp[2]))
553         temp.append(int(comp[3]))
554         temp.append(comp[4])
555         fin = tuple(temp)
556         final_group.append(fin)
557     else:
558         mov_rand = random.randint(math.ceil(prop1.Width/2),90)
559         #print("Igual separados")
560         prop1.move_xy((prop1.Width/2) + mov_rand,0,prop1.Objetos[0].Material, 0)
561         prop2.move_xy((0-(prop2.Width/2) + mov_rand)),0,prop2.Objetos[0].Material, 0)
562
563         prop1.get_values()
564         prop2.get_values()
565         prop3 = Composite(Composites[3])
566
567         prop3.move_xy(0,(prop1.Height/2)+(prop3.Height/2), prop3.Objetos[0].Material, 0)
568         prop3.get_values()
569
570         points = [[(prop1.Width/2) + mov_rand,0], [(0-((prop2.Width/2) + mov_rand)),0],
571                   [0,(prop1.Height/2)+(prop3.Height/2)]]
572         com = np.mean(points, axis=0)
573         delta = np.array((0,0)) - com
574         shifted_points = points + delta
575
576         list1 = prop1.Borders
577         list2 = prop2.Borders
578         list3 = prop3.Borders
579         #final_group.extend(Composites[piece1])
580         #final_group.extend(Composites[piece2])
581         #final_group.extend(Composites[3])

```

APPENDIX A. ANEXOS

```

582     #print(list3)
583     test=Composites[3].copy()
584     for comp in test:
585         temp = []
586         temp.append(comp[0])
587         #temp.append(int(comp[1]))
588         #temp.append(int(comp[2]) + (prop1.Height/2)+(prop3.Height/2) + 1)
589         temp.append(shifted_points[2][0])
590         temp.append(shifted_points[2][1]+1)
591         temp.append(int(comp[3]))
592         temp.append(comp[4])
593         fin = tuple(temp)
594         final_group.append(fin)
595     test = Composites[piece1].copy()
596     for comp in test:
597         temp = []
598         temp.append(comp[0])
599         #temp.append(int(comp[1]) + (prop1.Width/2) + mov_rand + 1)
600         #temp.append(int(comp[2]))
601         temp.append(shifted_points[0][0])
602         temp.append(shifted_points[0][1])
603         temp.append(int(comp[3]))
604         temp.append(comp[4])
605         fin = tuple(temp)
606         final_group.append(fin)
607     test = Composites[piece2].copy()
608     for comp in test:
609         temp = []
610         temp.append(comp[0])
611         #temp.append(int(comp[1]) + (0-((prop2.Width/2) + mov_rand)) - 1)
612         #temp.append(int(comp[2]))
613         temp.append(shifted_points[1][0])
614         temp.append(shifted_points[1][1])
615         temp.append(int(comp[3]))
616         temp.append(comp[4])
617         fin = tuple(temp)
618         final_group.append(fin)
619     Composites.update({new_value:final_group})
620 else:
621     #print("Diferente pegados")
622     ap1 = prop1.Width*prop1.Height
623     ap2 = prop2.Width*prop2.Height
624     if ap1 > ap2:

```

APPENDIX A. ANEXOS

```

625     if prop1.Width >= 90:
626         #print("Vertical")
627         prop1.Objetos[0].R = 90
628         prop1.Objetos[0].update_values()
629         prop1.move_xy(0,(0-(prop1.Width/2)),prop1.Objetos[0].Material, 90)
630
631         prop2.move_xy(0,(prop2.Height/2),prop2.Objetos[0].Material, 0)
632
633         points = [[0,(0-(prop1.Width/2))], [0,(prop2.Height/2)]]
634         com = np.mean(points, axis=0)
635         delta = np.array((0,0)) - com
636         shifted_points = points + delta
637         test = Composites[piece1].copy()
638         for comp in test:
639             temp = []
640             temp.append(comp[0])
641             #temp.append(int(comp[2]) + (prop1.Width/2))
642             temp.append(shifted_points[0][0])
643             temp.append(shifted_points[0][1]-1)
644             temp.append(90)
645             temp.append(comp[4])
646             fin = tuple(temp)
647             final_group.append(fin)
648         test = Composites[piece1].copy()
649         for comp in test:
650             temp = []
651             temp.append(comp[0])
652             #temp.append(int(comp[1]) + (0-((prop2.Width/2))) - 1)
653             temp.append(shifted_points[1][0])
654             temp.append(shifted_points[1][1]+1)
655             #temp.append(int(comp[2]))
656             temp.append(int(comp[3]))
657             temp.append(comp[4])
658             fin = tuple(temp)
659             final_group.append(fin)
660     else:
661         #print("Horizontal")
662         prop1.move_xy((0-(prop1.Width/2)),0,prop1.Objetos[0].Material, 0)
663         prop2.move_xy((prop2.Width/2),0,prop2.Objetos[0].Material, 0)
664         points = [[0,(0-(prop1.Width/2))], [0,(prop2.Width/2)]]
665         com = np.mean(points, axis=0)
666         delta = np.array((0,0)) - com
667         shifted_points = points + delta

```

APPENDIX A. ANEXOS

```

668         test = Composites[piece1].copy()
669         for comp in test:
670             temp = []
671             temp.append(comp[0])
672             #temp.append(int(comp[1]))
673             #temp.append(int(comp[2]) + (prop1.Width/2))
674             temp.append(shifted_points[0][0])
675             temp.append(shifted_points[0][1])
676             temp.append(int(comp[3]))
677             temp.append(comp[4])
678             fin = tuple(temp)
679             final_group.append(fin)
680         test = Composites[piece2].copy()
681         for comp in test:
682             temp = []
683             temp.append(comp[0])
684             #temp.append(int(comp[1]) + (0-(prop2.Width/2)) - 1)
685             temp.append(shifted_points[1][0])
686             temp.append(shifted_points[1][1])
687             #temp.append(int(comp[2]))
688             temp.append(int(comp[3]))
689             temp.append(comp[4])
690             fin = tuple(temp)
691             final_group.append(fin)
692     else:
693         if prop2.Width >= 90:
694             #print("Vertical")
695             prop2.Objetos[0].R = 90
696             prop2.Objetos[0].update_values()
697             prop2.move_xy(0,(0-(prop2.Width/2)),prop2.Objetos[0].Material, 90)
698             prop1.move_xy(0,(prop1.Height/2),prop1.Objetos[0].Material, 0)
699             points = [[0,(0-(prop2.Width/2))], [0,(prop1.Height/2)]]
700             com = np.mean(points, axis=0)
701             delta = np.array((0,0)) - com
702             shifted_points = points + delta
703             test = Composites[piece1].copy()
704             for comp in test:
705                 temp = []
706                 temp.append(comp[0])
707                 #temp.append(int(comp[1]))
708                 #temp.append(int(comp[2]) + (prop1.Width/2))
709                 temp.append(shifted_points[0][0])
710                 temp.append(shifted_points[0][1]+1)

```

APPENDIX A. ANEXOS

```
711         temp.append(int(comp[3]))
712         temp.append(comp[4])
713         fin = tuple(temp)
714         final_group.append(fin)
715         test = Composites[piece2].copy()
716         for comp in test:
717             temp = []
718             temp.append(comp[0])
719             #temp.append(int(comp[1]) + (0-((prop2.Width/2))) - 1)
720             temp.append(shifted_points[1][0])
721             temp.append(shifted_points[1][1]-1)
722             #temp.append(int(comp[2]))
723             temp.append(90)
724             temp.append(comp[4])
725             fin = tuple(temp)
726             final_group.append(fin)
727     else:
728         #print("Horizontal")
729         prop2.move_xy((0-(prop2.Width/2)),0,prop2.Objetos[0].Material, 0)
730         prop1.move_xy((prop1.Width/2),0,prop1.Objetos[0].Material, 0)
731         points = [[0,(0-(prop2.Width/2))], [0,(prop1.Width/2)]]
732         com = np.mean(points, axis=0)
733         delta = np.array((0,0)) - com
734         shifted_points = points + delta
735         test = Composites[piece1].copy()
736         for comp in test:
737             temp = []
738             temp.append(comp[0])
739             #temp.append(int(comp[1]))
740             #temp.append(int(comp[2]) + (prop1.Width/2))
741             temp.append(shifted_points[0][0])
742             temp.append(shifted_points[0][1])
743             temp.append(int(comp[3]))
744             temp.append(comp[4])
745             fin = tuple(temp)
746             final_group.append(fin)
747         test = Composites[piece2].copy()
748         for comp in test:
749             temp = []
750             temp.append(comp[0])
751             #temp.append(int(comp[1]) + (0-((prop2.Width/2))) - 1)
752             temp.append(shifted_points[1][0])
753             temp.append(shifted_points[1][1])
```

APPENDIX A. ANEXOS

```

754         #temp.append(int(comp[2]))
755         temp.append(int(comp[3]))
756         temp.append(comp[4])
757         fin = tuple(temp)
758         final_group.append(fin)
759         #prop1.move_xy(0, (prop1.Height/2), prop1.Objetos[0].Material, 0)
760         #prop2.move_xy(0, (0-(prop2.Height/2)), prop2.Objetos[0].Material, 0)
761         Composites.update({new_value:final_group})
762         prop1.get_values()
763         prop2.get_values()
764         list1 = prop1.Borders
765         list2 = prop2.Borders
766
767         #print(list1)
768         #print(list2)
769
770 #####
771 #####< Code Adaptation >#####
772 #####
773 sel_operator = Selection(project_root, game_path_tourney, info)
774 mut_operator = Mutation(per_mut, mu, sigma, clases)
775 mut_operator.UpdateComposites(Composites)
776 #####
777 #####< Class Definitions >#####
778 #####
779
780
781
782
783 class Individual:
784     def __init__(self, **kwargs):
785         self.chromosome = kwargs.get('chromosome', [])
786         self.mask = kwargs.get('mask') #self.assign_mask(Mask_List[kwargs.get('mask')])
787         self.__dict__.update(kwargs)
788         self.chromosome_objects = [Composite(Composites[composite]) for composite in
789             self.chromosome]
790         self.object_list = self.object_list_gen()
791         self.object_masked = []
792         self.Mut_Movement = [0 for value in self.chromosome]
793         self.Mut_Struct = [-1 for value in self.chromosome]
794         self.Mut_Material = ["wood" for value in self.chromosome]
795
796     def position_chromosome(self):

```

APPENDIX A. ANEXOS

```
797     #To do
798     # Here you can use the chromosome objects etc.
799     pass
800
801     def UpdateMutation(self):
802         for c, struc in enumerate(self.Mut_Struct):
803             if struc != -1:
804                 self.chromosome[c] = struc
805
806         self.chromosome_objects = [Composite(Composites[composite]) for composite in
807             self.chromosome]
808
809         for c, objeto in enumerate(self.chromosome_objects):
810             for pieza in objeto.Objetos:
811                 if self.Mut_Material[c] != 0:
812                     pieza.Material = self.Mut_Material[c]
813                     pieza.change_material(self.Mut_Material[c])
814                     pieza.update_values()
815                     objeto.as_dictionary = objeto.gen_dictionary()
816
817         for c, mov in enumerate(self.Mut_Movement):
818             self.chromosome_objects[c].move_xy(mov, 0, self.Mut_Material[c], 0)
819
820         self.object_list = self.object_list_gen()
821     pass
822
823     def chromosome_coordinates(self):
824
825         return [0, 0]
826
827     def object_list_gen(self):
828         final_list = []
829         for com in self.chromosome_objects:
830             com.as_dictionary = com.gen_dictionary()
831             obj_list = []
832             obj_list.append(com.as_dictionary)
833             final_list.append(obj_list)
834         return final_list
835
836     def generate_xml(self, **kwargs):
837         res_list = []
838         res_list = xml.writeXML(self.object_list, os.path.join(project_root, write_path +
839             "/level-0"+ str(kwargs.get('individual')) +".xml"))
```

APPENDIX A. ANEXOS

```
840         self.ind_height = res_list[0]
841         self.ind_piece = res_list[1]
842         self.Pieces = res_list[2]
843         pass
844
845     def read_xml(self, **kwargs):
846         self.Remaining_Pieces = xml.readXML(os.path.join(project_root, read_path +
847             "/level-"+ str(kwargs.get('individual')) +".xml"))
848         self.ind_piece_count = len(self.Remaining_Pieces)
849         pass
850
851     def read_xml_tourney(self, **kwargs):
852         self.Remaining_Pieces = xml.readXML(os.path.join(project_root, read_path_tourney +
853             "/level-"+ str(kwargs.get('individual')) +".xml"))
854         self.ind_piece_count = len(self.Remaining_Pieces)
855         pass
856
857     def ind_height(self):
858         return 0
859
860     def ind_piece(self):
861         return 0
862
863     def ind_piece_count(self):
864         return 0
865
866     def assign_mask(self, mask_class):
867         masked = mask_class
868         print(masked)
869         #self.mask = mask_class
870         return masked
871
872     def locate_pigs(self):
873         self.Pig_Location = mut_operator.Set_Pigs(self.chromosome_objects, req_pigs,
874             self.mask, self.Composites_Centers)
875         pass
876
877     def get_fitness(self, chrom_list, pos):
878         self.Fitness, self.Fit_Hamming, self.Fit_Pos =
879             Eval.fitness(self.Pieces,
880                 self.Remaining_Pieces, self.chromosome, chrom_list, pos)
881         return 0
882
```

APPENDIX A. ANEXOS

```
883     def combine_mask(self):
884         self.object_masked, self.Composites_Centers =
885             xml.calculate_mask(self.object_list,
886                 self.mask, self.chromosome_objects)
887             #self.object_list = xml.calculate_mask(self.object_list, type_castle)
888         return 0
889     def generate_xml_masked(self, **kwargs):
890         res_list = []
891         res_list = xml.writeXML_masked(self.Pig_Location, self.object_list,
892             os.path.join(project_root, write_path + "/level-0" +
893                 str(kwargs.get('individual')) +
894                 ".xml"))
895         self.ind_height = res_list[0]
896         self.ind_piece = res_list[1]
897         self.Pieces = res_list[2]
898         #print("XML Completo")
899         pass
900
901     def generate_xml_tourney(self, **kwargs):
902         res_list = []
903         res_list = xml.writeXML_masked(self.Pig_Location, self.object_list,
904             os.path.join(project_root, write_path_tourney +
905                 "/level-0"+ str(kwargs.get('individual')) +".xml"))
906         self.ind_height_tourney = res_list[0]
907         self.ind_piece_tourney = res_list[1]
908         self.Pieces = res_list[2]
909         pass
910
911     def generate_xml_elite(self, **kwargs):
912         res_list = []
913         res_list = xml.writeXML_masked(self.Pig_Location, self.object_list,
914             os.path.join(project_root, elite_path + "/level-" + str(kwargs.get('gen')) +
915                 "0" + str(kwargs.get('individual')) + ".xml"))
916         self.ind_height = res_list[0]
917         self.ind_piece = res_list[1]
918         self.Pieces = res_list[2]
919         pass
920
921
922     def create_new_mask(pieces):
923         div_list =[]
924         while True:
925             div_list = [random.randint(0, pieces-1) for col in range(7)]
```

APPENDIX A. ANEXOS

```
926         if sum(div_list) == pieces:
927             break
928
929     return div_list
930
931 # Generate a defined number of new composites to add to the pool of selections
932 for i in range(0,9):
933     combine_pieces()
934
935 pop = [ Individual(chromosome = get_random_chrom(ind_pieces),
936                     mask = create_new_mask(ind_pieces)) for i in range(population)]
937
938 # variable for controlling that the fitness of the population is compared to
939 # at least 2 others in the same timeline (between resets)
940 gen_alive = 0
941
942 with yaspin(text="Executing algorithm", color="cyan") as sp:
943     while gen < max_gen: #and max(fits) < 100:
944
945         # Outside IF statement
946         # Reintegrate the ELITE member to the population and remove the one
947         # in the last position
948         if len(elite):
949             for member in elite:
950                 pop.insert(0, member)
951                 #pop[0] = Individual(chromosome = member[1], mask = member[2])
952                 pop = pop[:population]
953
954         # Check if the current number of population multiplied by the
955         # cross-over percentage is an even or odd number, in the later
956         # case remove 1 from the value
957         many = len(pop) * per_cross
958         if many % 2 == 0:
959             pass
960         else:
961             many = many - 1
962
963         # Combine the mask of the individual before entering the simulation
964         ind_c = 0
965         for ind in pop:
966             if hasattr(ind, 'Fitness'):
967                 pass
968             else:
```

APPENDIX A. ANEXOS

```
969         ind.combine_mask()
970         ind.locate_pigs()
971         ind.generate_xml_masked(individual = ind_c)
972         ind_c = ind_c + 1
973
974     # Runs and instance of the game
975     if hasattr(ind, 'Fitness'):
976         pass
977     else:
978         subprocess.call(r''' + os.path.join(project_root, game_path) + ''',
979                         startupinfo=info) # doesn't capture output
980
981     # After the simulation obtain the fitness value for the population
982     # Read the xml files and get the data
983     ind_c = 0
984     final_ind_list = []
985     for ind in pop:
986         value = ind.read_xml(individual = ind_c)
987         final_ind_list.append(value)
988         ind_c = ind_c + 1
989
990     # Generate a list with the chromosome values of all individuals
991     # (required for Hamming distance)
992     chrom_list = []
993     for ind in pop:
994         chrom_list.append(ind.chromosome)
995
996     # Calculate the fitness for each individual
997     for c, ind in enumerate(pop):
998         ind.get_fitness(chrom_list, c)
999         pass
1000
1001 #####< Selection steps >#####
1002 #####
1003 #####
1004
1005     # Order the population by their fitness
1006     pop.sort(key=lambda x:x.Fitness, reverse=False)
1007
1008     # Obtain the "parents" of the generation
1009     parents = []
1010     pr = 1
1011     parents = sel_operator.Selection_Base(pop, many, sel_type)
```

APPENDIX A. ANEXOS

```
1012
1013 #####< Crossover steps >#####
1014 #####
1015 #####
1016
1017     new_members = []
1018
1019     # Generate the cross-over operation (one-point crossover)
1020     pos_offspring = 0
1021     for cross_parent in range(0, len(parents), 2):
1022         # Generate a copy of each parent for the cross-over operation
1023         father = pop[parents[cross_parent] - 1].chromosome
1024         mother = pop[parents[cross_parent + 1] - 1].chromosome
1025
1026         # "Divide" the parents chromosomes for the operation
1027         father11 = father[0:math.floor(ind_pieces/2)]
1028         father12 = father[math.floor(ind_pieces/2):]
1029
1030         mother11 = mother[0:math.floor(ind_pieces/2)]
1031         mother12 = mother[math.floor(ind_pieces/2):]
1032
1033         # Generate the childs of both parents
1034         son = father11 + mother12
1035         daughter = mother11 + father12
1036
1037         mask_son = create_new_mask(len(son))
1038         mask_daughter = create_new_mask(len(daughter))
1039         # Replace the parents in the generation
1040         pop[pos_offspring].chromosome = son
1041         pop[pos_offspring + 1].chromosome = daughter
1042
1043         pop[pos_offspring] = Individual(chromosome = son, mask = mask_son)
1044         pop[pos_offspring + 1] =
1045             Individual(chromosome = daughter, mask = mask_daughter)
1046
1047         # Mutate the childs (by chance like throwing a 100 side dice)
1048         # If greater than the threshold then mutate
1049         chance = random.randint(1, 100)
1050         threshold = 100 - (100 * per_mut)
1051         #chance = 100
1052         if chance > threshold:
1053             var=0
1054             new_chrom =
```

APPENDIX A. ANEXOS

```

1055         mut_operator.M_Individual(pop[pos_offspring].chromosome)
1056         pop[pos_offspring] = Individual(chromosome = new_chrom,
1057             mask = create_new_mask(len(new_chrom)))
1058         pop[pos_offspring] =
1059             mut_operator.M_Movement(pop[pos_offspring], 0)
1060         pop[pos_offspring] =
1061             mut_operator.M_StructType(pop[pos_offspring], 0)
1062         pop[pos_offspring] =
1063             mut_operator.M_StructMat(pop[pos_offspring], 0)
1064         pop[pos_offspring].UpdateMutation()
1065         #print("Mutate")
1066     else:
1067         var=1
1068         #print("Not Mutate")
1069
1070     # The same for the second child
1071     chance = random.randint(1, 100)
1072     threshold = 100 - (100 * per_mut)
1073     #chance = 100
1074     if chance > threshold:
1075         var=0
1076         new_chrom =
1077             mut_operator.M_Individual(pop[pos_offspring + 1].chromosome)
1078         pop[pos_offspring + 1] = Individual(chromosome = new_chrom,
1079             mask = create_new_mask(len(new_chrom)))
1080         pop[pos_offspring + 1] =
1081             mut_operator.M_Movement(pop[pos_offspring + 1], 0)
1082         pop[pos_offspring + 1] =
1083             mut_operator.M_StructType(pop[pos_offspring + 1], 0)
1084         pop[pos_offspring + 1] =
1085             mut_operator.M_StructMat(pop[pos_offspring + 1], 0)
1086         pop[pos_offspring + 1].UpdateMutation()
1087         #print("Mutate")
1088
1089         pop[pos_offspring].ind_c = pos_offspring
1090         pop[pos_offspring + 1].ind_c = pos_offspring + 1
1091         new_members.append(pop[pos_offspring])
1092         new_members.append(pop[pos_offspring + 1])
1093
1094     # Calculate the new individuals fitness by sending them to a
1095     # different simulation track
1096
1097     # Generate an XML to check the fitness

```

APPENDIX A. ANEXOS

```
1098     for ind_c, ind in enumerate(new_members):
1099         ind.combine_mask()
1100         ind.locate_pigs()
1101         ind.generate_xml_tourney(individual = ind_c)
1102
1103     # Execute the application with the two members
1104     subprocess.call(r'\" + os.path.join(project_root, game_path_tourney) + '\"',
1105                     startupinfo=info)
1106
1107     # Generate a list with the chromosome values of all individuals
1108     # (required for Hamming distance)
1109     chrom_list = []
1110     for ind in pop:
1111         chrom_list.append(ind.chromosome)
1112
1113     # Then obtain the remaining fitness values
1114     for c, ind in enumerate(new_members):
1115         ind.read_xml_tourney(individual = ind.ind_c)
1116         ind.get_fitness(chrom_list, c)
1117
1118 #####< ELITE selection >#####
1119 #####
1120 #####
1121
1122     # Obtain the average fitness of the generation
1123     gen_fit = 0
1124     hamming_fit = 0
1125     mov_fit = 0
1126     best_ind = 0
1127     fit_pop = []
1128     for c, ind in enumerate(pop):
1129         fit_pop.append([c, ind.Fitness])
1130         gen_fit = gen_fit + ind.Fitness
1131         hamming_fit += ind.Fit_Hamming
1132         mov_fit += ind.Fit_Pos
1133
1134     fit_pop.sort(key=lambda x:x[1], reverse=True)
1135     max_fit.append(fit_pop[0][1])
1136     min_fit.append(fit_pop[-1][1])
1137     fit_pop = fit_pop[:5]
1138
1139     #
1140     best_gen.append(fit_pop[0][1])
```

APPENDIX A. ANEXOS

```
1141
1142     # Add the best value to the elite list
1143     for e in fit_pop:
1144         elite.append(pop[e[0]])
1145         elite[0].generate_xml_elite(individual = e[0], gen = gen)
1146
1147         elite.sort(key=lambda x:x.Fitness, reverse=True)
1148         elite = elite[:max_elite]
1149
1150         all_fit.append((gen_fit/len(pop)))
1151         fit_ham.append((hamming_fit/len(pop)))
1152         fit_Amov.append((mov_fit/len(pop)))
1153
1154         # Increase value of the generation for the next cycle
1155         gen = gen + 1
1156         gen_alive += 1
1157
1158         # Print the time of the generation
1159         t_gen = datetime.datetime.now()
1160         sp.write("> Gen " + str(gen) + " complete, gen duration: " +
1161                 str(t_gen-t_prev) + ", current time: " + str(t_gen-t_begin))
1162         t_prev = datetime.datetime.now()
1163
1164         # Copy the resulting files to a folder with the number of generation
1165         destination = os.path.join(project_root, level_files_path + "/generation-" +
1166             str(current_gen_folder))
1167         shutil.copytree(sourcefolder, destination)
1168         current_gen_folder += 1
1169
1170         t_finish = datetime.datetime.now()
1171         #print("Total time is: " + str(t_finish-t_begin))
1172         sp.write("> Total time is: " + str(t_finish-t_begin))
1173
1174         # Stop the spinner
1175         sp.ok("Ok")
1176
1177         # Plot the results for Fitness
1178         pylab.figure(figsize=(8, 5))
1179         plot(all_fit, '-.b', label='General Fitness')
1180         plot(min_fit, '-g', label='Min fitness value')
1181         plot(max_fit, '*-r', label='Max fitness value')
1182         lgd = pylab.legend(loc='center left', bbox_to_anchor=(1, 0.5))
1183         pylab.savefig('Fitness', bbox_extra_artists=(lgd,), bbox_inches='tight')
```

APPENDIX A. ANEXOS

```
1184
1185     # Same for Hamming Distance
1186     pylab.figure(figsize=(8, 5))
1187     plot(fit_ham, '-g', label='Average Hamming Distance')
1188     lgd = pylab.legend(loc='center left', bbox_to_anchor=(1, 0.5))
1189     pylab.savefig('Hamming', bbox_extra_artists=(lgd,), bbox_inches='tight')
1190
1191     # Add the data to the json files
1192     json_data['all_fit'] = all_fit
1193     json_data['hamming_avg'] = fit_ham
1194     json_data['min_fit'] = min_fit
1195     json_data['max_fit'] = max_fit
1196
1197     with open('results.json', 'w') as outfile:
1198         print(outfile)
1199         json.dump(json_data, outfile)
1200
1201     # Clean the workspace
1202     shutil.copytree(experimentsource, experimentdestination)
1203
1204     shutil.copyfile('results.json',
1205                     os.path.join(project_root, experimentdest) + '/results.json')
1206     shutil.copyfile('Fitness.png',
1207                     os.path.join(project_root, experimentdest) + '/Fitness.png')
1208     shutil.copyfile('Hamming.png',
1209                     os.path.join(project_root, experimentdest) + '/Hamming.png')
```

Código para calculo de fitness

```
1     # Evaluation metrics for the fitness of the individuals
2     #
3     # The metric to evaluate the fitness of an individual are:
4     # 1. If the number of pieces is different give a penalty
5     # 2. If the positioning of the pieces is different for more than 1 give a penalty
6     # The penalties are given has follows
7     # Initial fitness 100%
8     # - size_difference
9     #     (remaining_pieces * 100) / original_pieces
10    # - position_error
11    #     for each piece
12    #         if the piece position is different by more than 2
```

APPENDIX A. ANEXOS

```
13      #          ((1 * 100) / original_pieces)
14      #          else
15      #          ((1 * 100) / original_pieces) * (((abs(position) * 100) / 2) / 100)
16      #
17      #          Check the angle
18      #          if different by more/less than ± 5 then penalize
19      ##########
20      #
21      # After the evaluation is over if a stable group remains in the file
22      # this group will be added to the pool of pieces
23      #
24      #
25
26      from AngryBirdsGA import *
27      import math
28      import hashlib
29      from copy import deepcopy
30
31
32      def fitness(ind_orig, ind_fin, chromosome, pop, pos):
33          criteria = 0
34          #criteria = size_dif(ind_orig, ind_fin)
35          criteria += (dif_pieces(chromosome) * 10) # 240
36          criteria += calc_entropy(chromosome) * 100
37          hamming_dist = deepcopy(calc_hamming(chromosome, pop, pos))
38          criteria += hamming_dist
39
40          criteria_new = [deepcopy(criteria), deepcopy(hamming_dist)]
41
42          #Evaluate fitness based on stability
43          tot_fitness = 0
44          tot_fitness += deepcopy(size_dif(ind_orig, ind_fin))
45          tot_fitness += deepcopy(position_error(ind_orig, ind_fin))
46
47          return [tot_fitness, criteria_new, 100]
48
49      def calc_hamming(chromosome, pop, pos):
50          total = []
51          for c, chrom in enumerate(pop):
52              if c != pos:
53                  total.append(deepcopy(hamming_distance(deepcopy(chromosome),
54                                              deepcopy(chrom))))
```

APPENDIX A. ANEXOS

```
56     total.sort(key=lambda x:x, reverse=True)
57     return total[0]
58
59
60 def calc_entropy(ind):
61     # First calculate the frequency of the elements in array
62     value_list = set(ind)
63
64     freqList = []
65     for piece in value_list:
66         ctr = 0
67         for ch in ind:
68             if ch == piece:
69                 ctr += 1
70         freqList.append(float(ctr) / len(ind))
71
72     result = 0
73     for e in freqList:
74         result += (e*math.log(e,2))
75     result = -result
76     return result
77
78 def dif_pieces(b):
79     #type_list = [piece[0] for piece in b]
80     return len(set(b))
81
82 def size_dif(a, b):
83     val_a = len(a)                      # Original amount
84     val_b = len(b)                      # Remaining amount
85     percentage = (val_b * 100)/val_a    # Percentage to remove from total
86     return percentage
87
88 def position_error(a, b):
89     total_error = 0
90     for piece in b:
91         # 0 = Block
92         # 1 = Material
93         # 2 = x
94         # 3 = y
95         # 4 = r
96         # 5 = id
97         try:
98             orig = a[int(piece[5])]
```

APPENDIX A. ANEXOS

```
99         error_xy = 0 if 1.0 > math.hypot((float(piece[2])) - (float(orig[2])),  
100            (float(piece[3])) - (float(orig[3]))) else ((100/len(a)) * -0.5)  
101        error_z = 0 if -5 < (abs(float(orig[4])) - abs(float(piece[4]))) < 5 else  
102            ((100/len(a)) * -0.5)  
103        total_error = total_error + error_xy + error_z  
104    except IndexError:  
105        text = "Missing values - Pieces destroyed or something?..."  
106    return total_error  
107  
108  
109    def hamming_distance(a, b):  
110        # Calculate and return the Hamming distance of the two sets  
111        return sum(c1 != c2 for c1, c2 in zip(a, b))
```
