Generación de Contenido Procedural: Un acercamiento a los Sistemas Lindenmayer

Procedural Content Generation: An approach to Lindenmayer Systems

Jesús Bertani Ramírez
Estudiante de la Licenciatura en Computación Matemática
j.bertaniramirez@ugto.mx

Resumen

El Contenido Procedural se ha convertido en un Standard para el desarrollo de videojuegos AAA pues permite reducir costos en la creación de mapas muy grandes o de objetos que son muy similares entre sí. Sin embargo, este concepto es malentendido entre el público general pues usualmente se asocia con la aleatoriedad.

Buscando aclarar esta confusión, en este proyecto se pretende dar una introducción a uno de los métodos usados para la Generación de Contenido Procedural: Los Sistemas Lindenmayer, y su aplicación para la creación de estructuras que simulan sistemas biológicos como plantas y árboles.

Palabras clave: Sistema-L, Sistema, Lindenmayer, Tortuga, Encorchetado, Paramétrico, Unity, Processing.

Introducción

La popularidad de juegos como Minecraft y No Man's Sky ha levantado la curiosidad del público general hacia la generación de contenido automático en los juegos. En su búsqueda de información, llegarán a encontrarse con el concepto de "G*eneración Procedural de Contenido*" (o PCG por sus siglas en inglés). Sin embargo, esto lleva a una confusión pues dicha palabra se termina asociando a generar mundos o niveles aleatorios. En realidad, PCG *es la creación algorítmica de contenido de un juego con o sin interacción del usuario* [1], donde entendemos por contenido de un juego por todo lo que uno contiene, como reglas, texturas, historias, ítems, música, etc., y no solo mundos o niveles como se tiende a malinterpretar. Y lo que es más, que el proceso sea algorítmico significa que no es inherentemente aleatorio, sino que hay pasos bien definidos para obtener resultados deseables.

Conforme los videojuegos se volvieron más realistas, la vegetación pasó a ser un tema importante pues se encuentra en casi todas, si no es que todas, las escenas. Sin embargo, generar la cantidad de vegetación necesaria para crear un ambiente creíble llevaría cantidades gigantes de tiempo y esfuerzo; es por esto que se prefiere generar automáticamente. En la naturaleza, las plantas están llenas de estructuras auto similares y sería deseable que existieran herramientas que nos permitan emular dichos patrones al momento de generar las plantas.

Figura 1. Hoja de helecho. Imagen por Siegella

Es aquí donde entran los Sistemas Lindenmayer, que fueron concebidos para emular dichos patrones de la naturaleza. Programas como Unity y Houdini contienen herramientas para generar árboles usando como base los Sistemas Lindenmayer obteniendo resultados alucinantes que no tienen nada que envidiar a los árboles reales.

Figura 2. Árbol hecho en Houdini

Figura 3. Árbol hecho en Unity usando TREEGEN

Sin embargo, en esta ocasión no nos interesa llegar a ese nivel de detalle, más bien, queremos acercar al público general a los Sistemas Lindenmayer que hay detrás por lo que se decidió realizar dos programas que sirvan para generar las estructuras biológicas de los primeros capítulos del libro “The Algorithmic Beauty of Plants” [2] mediante la utilización de Gráficos Tortuga. Para simular las estructuras bidimensionales se usó el lenguaje Processing y se utilizó la herramienta Unity con el lenguaje C# para simular las estructuras tridimensionales y así poder darles una apariencia un poco más realista.

Metodología

Para poder llegar a generar estructuras que asemejan a aquellas encontradas en la naturaleza con Sistemas Lindenmayer (también llamados Sistemas-L), primero debemos comprender qué son estos Sistemas y el por qué es deseable usarlos. Es por eso que comenzaremos esta sección planteando los conceptos necesarios para entender y definir a los Sistemas Lindenmayer para posteriormente continuar con su interpretación gráfica que es la que nos permitirá generar estructuras y finalmente describir implementaciones que nos permitan visualizar e interactuar con estos Sistemas.

Sistemas Lindenmayer

Una *Gramática* es un conjunto de caracteres, llamado *alfabeto*, y *reglas de producción* que nos indican cómo reescribir una cadena de caracteres mediante la sustitución de letras [1]. Las reglas son de la forma (caracter(es))→(caracter(es)), por ejemplo:

1. A → AB
2. B → A,

donde lo que está a la izquierda de la flecha (llamado *predecesor*) se sustituye por lo de la derecha (llamado *sucesor*) y esto se puede hacer de dos maneras: Secuencialmente y Paralelamente. Sustituir **Secuencialmente** nos permite elegir qué regla aplicar en cada paso, lo que nos da cierta libertad. Por ejemplo, si comenzamos con la letra A y usamos las reglas anteriores, podemos obtener la siguiente secuencia al realizar 5 sustituciones:

1. A
2. AB
3. ABB
4. AAB
5. AABB
6. ABABB

Pero también podemos obtener la siguiente

1. A
2. AB
3. AA
4. AAB
5. AAA
6. AAAB

Como podemos observar, el resultado no es único y no hay un patrón aparente. Ahora, sustituir **Paralelamente** significa usar todas las reglas en el mismo paso. Así, si comenzamos nuevamente con A, después de 4 sustituciones obtendremos en esta ocasión la siguiente secuencia:

1. A
2. AB
3. ABA
4. ABAAB
5. ABAABABA

que es única (en este caso) y, además, muestra regularidad. Es por esta misma razón que este tipo de sustitución nos ayudará a simular los patrones de las plantas y además, da paso a los Sistemas Lindenmayer.

Los S*istemas Lindenmayer* (Sistemas-L) son una *gramática* donde las reglas son tales que el predecesor es siempre de un solo carácter y todos los caracteres del alfabeto tienen una regla de sustitución correspondiente (Si no se da explícita se asume que le corresponde la sustitución identidad, es decir, no se cambia ni se borra). Además, las sustituciones se hacen de manera paralela y se da una secuencia de caracteres inicial llamada *axioma*. Una cadena resultante de un sistema Lindenmayer es cierta cadena que

se obtiene de aplicar n veces las reglas al axioma. Si ninguna pareja de reglas repite el mismo predecesor ocurre que nuestro Sistema-L es *determinístico* pues siempre se obtendrá el mismo resultado al realizar las sustituciones. [1, 4] Es este tipo de sistemas con el que trabajaremos por el resto del escrito.

Todo esto es muy bueno, ya tenemos una forma de obtener regularidad, pero ¿qué tiene que ver con las plantas? Resulta que podemos darles una interpretación gráfica a nuestras cadenas resultantes como si fueran instrucciones para una tortuga en los *Gráficos Tortuga*.

En los Gráficos Tortuga pensaremos que una tortuga que sujeta un lápiz se mueve en el espacio y con él va dibujando su camino. En esta sección comenzaremos con describir sus instrucciones para un **Espacio Bidimensional**. Aquí, un estado de la tortuga es descrito por la tripleta (x, y, α) donde (x, y) indica su *posición* en el plano *y* α indica su *frente*, mediante un ángulo que indica en qué dirección está viendo la tortuga con respecto al eje X y en sentido antihorario. Dado un *tamaño de paso* d y un *ángulo de incremento* δ, la tortuga responderá a las siguientes reglas:

- F — Se moverá hacia adelante d unidades. Es decir, su estado cambiará a (x', y', α) donde $x' = x + d\cos(\alpha)$ y $y' = y + d\sin(\alpha)$ y además, se trazará una línea entre (x, y) y (x', y').
- f — Se moverá hacia adelante d unidades sin dibujar una línea.
- + — Girar a la izquierda un ángulo delta. Así, el siguiente estado de la tortuga será $(x, y, \alpha + \delta)$
- - — Girar a la derecha un ángulo delta. Así el siguiente estado será $(x, y, \alpha - \delta)$

Usando estas reglas somos capaces de recrear estructuras regulares y autosimilares, como los fractales de este tipo.

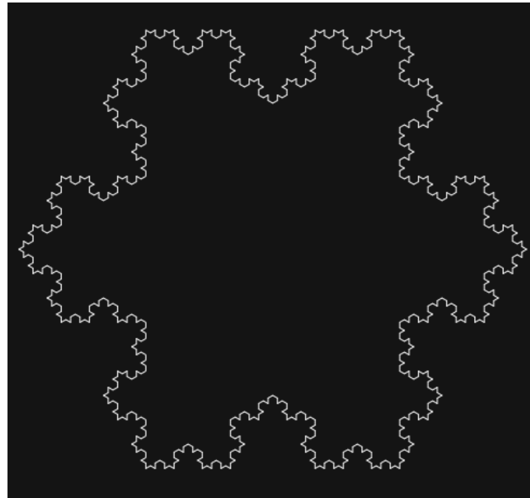


Figura 4. Resultado de usar el axioma "F++F++F", la regla "F ⟶F-F++F-F", $\delta = 60$ y 4 iteraciones

Pero aún no podemos generar plantas convincentes pues solo podemos dibujar cosas que existen sobre una misma curva y, como podemos observar, las plantas no cumplen esto, pues en cierto punto la curva deberá tomar 2 o más direcciones para poder dibujar las ramas. Para lograr este efecto daremos una extensión a los sistemas que ya tenemos: *Los sistemas Lindenmayer Encorchetados.*

Estos sistemas contienen en su alfabeto los símbolos "[" y "]" que se usarán para definir ramas. Al realizar su interpretación en gráficos tortuga, además del estado de la tortuga, tendremos una pila de estados y estos dos símbolos los interpretaremos de la siguiente manera:

- [— Agregamos el estado actual de la tortuga a la pila.

-] Sacamos un estado de la pila y lo volvemos el estado actual de la tortuga. No se traza ninguna línea.

Estas dos nuevas reglas nos permiten obtener estructuras como la siguiente

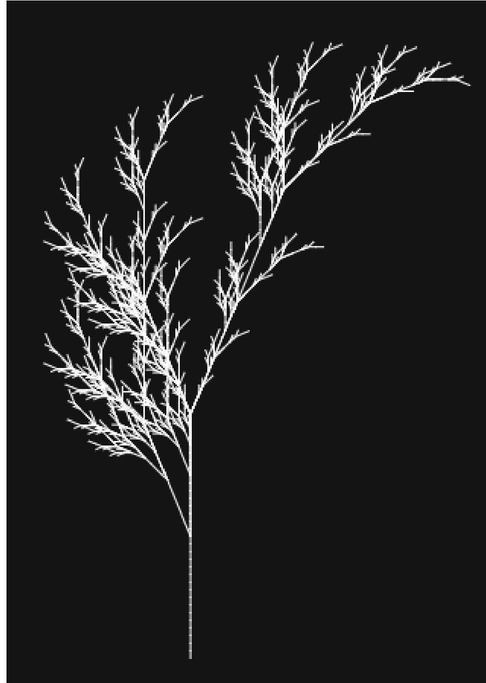


Figura 5. Resultado de usar el axioma "X", las reglas "X ⟶F-[[X]+X]+F[+FX]-X" y "F⟶FF", $\delta = 22.5$ y 5 iteraciones

Las cuales ya tienen un parecido con las plantas de la vida real. Es así que el primer objetivo de este proyecto fue el generar un programa que permita a los usuarios experimentar con los sistemas Lindenmayer Encorchetados para generar estructuras de esta clase usando el lenguaje de programación Processing.

Sistemas Lindenmayer Encorchetados en Processing

Para poder obtener los diagramas expuestos anteriormente lo primero que debemos hacer es solicitarle al usuario las partes que definen al Sistema Lindenmayer y lo necesario para que la tortuga pueda dibujar. A continuación explicamos las suposiciones y herramientas utilizadas para ingresar satisfactoriamente los datos requeridos.

Para definir a la tortuga no lo complicamos y asumimos que siempre comienza en el punto (0,0) por lo que solo debemos solicitar su orientación inicial, ya sea mediante un ángulo o un vector que posteriormente convertimos al ángulo buscado. En nuestro caso solicitamos un vector. También necesitamos el *tamaño de paso* d y el *ángulo de incremento* δ, sin embargo, asumiremos un tamaño de paso predeterminado y solo solicitamos el *ángulo de incremento*.

Ahora, ingresar el sistema-L es un poco más complicado. En teoría, este necesita el alfabeto, las reglas y el axioma, sin embargo, solicitar el axioma y las reglas es suficiente pues el alfabeto se puede obtener de estos. Para nuestra implementación se decidió que solo las letras mayúsculas y minúsculas del alfabeto inglés, "+", "-", "[" y "]" sean caracteres válidos del alfabeto del sistema y es por esta razón que debemos verificar si el usuario ingresó un axioma o regla correcta. Para lograr esto usaremos *expresiones regulares* de las que puede obtener más información en [3].

Diremos que un **axioma es válido** si coincide con la siguiente expresión regular:

^(([a-zA-Z\-\+])|(\[[a-zA-Z\-\+]+\])|(\[(([a-zA-Z\-\+])|(\[[a-zA-Z\-\+]\]))+\]))+$

Notemos que permitimos cualquier secuencia de letras y símbolos “+” y “–” pero que por cada “[“ debe existir un “]” correspondiente y que además, permitimos expresiones con corchetes anidados. Esto se hizo así para evitar cualquier problema pues por cada estado que se ingresa a la pila, eventualmente deberá eliminarse.

Ahora, diremos que una **regla es válida** si coincide con la siguiente expresión regular:

^[a-zA-Z]=(([a-zA-Z\-\+])|(\[[a-zA-Z\-\+]+\])|(\[(([a-zA-Z\-\+])|(\[[a-zA-Z\-\+]\]))+\]))+$

Como habrá notado, en lugar de usar una flecha para separar los dos elementos de la regla, usaremos el signo "=". Como predecesor solamente permitimos letras pues permitir sustituir corchetes podría llevar nuevamente a problemas y se consideró innecesario el sustituir los símbolos “+” y “-”. Finalmente, como sucesor permitimos cualquier expresión que sea un axioma válido.

Ya sabemos cómo vamos a definir una regla válida, pero no al conjunto de ellas. Diremos que un **conjunto de reglas es válido** si es vacío o cada una de las reglas es válida y estas dan paso a un Sistema-L determinístico. Consideraremos al conjunto vacío como reglas válidas pues esto significa que solo se debe aplicar la sustitución identidad a todos los elementos.

Una vez habiendo definido estos conceptos, solo basta solicitar al usuario que ingrese el axioma y las reglas en forma de texto y los aceptaremos si son válidos. Sin embargo, aún falta definir cuantas veces aplicaremos las reglas y esto es otro parámetro que también le solicitaremos al usuario.

Tomando todas estas consideraciones, ya solo queda construir un lugar en donde el usuario pueda ingresar todos estos parámetros y es por eso que se utilizó en Processing la librería G4P para generar la interfaz de usuario que cumple dicha función, obteniendo el siguiente resultado:

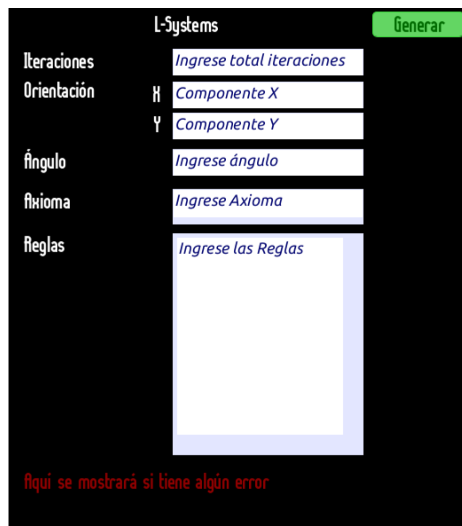


Figura 6. Interfaz de usuario generada en Processing usando G4P

Notemos que las reglas se piden en una sola caja de texto por lo que aquí también asumimos que lo que separa una regla de otra es el caracter ‘\n’.

Una vez solicitados y guardados los datos necesarios debemos aplicar las reglas. Esto es un proceso directo y una forma de lograrlo es la siguiente:

Algorithm 1 Aplicar reglas de un Sistema Lindenmayer Encorchetado

Input: AXIOM (Un axioma válido), RULES (Un conjunto de reglas válido), ITERATIONS (Las veces que se van a aplicar las reglas)

Output: La palabra resultante de aplicar las reglas una cantidad ITERATIONS de veces

1: Inicializar $Reglas$ como un mapa que tenga como llaves caracteres, guarde strings y esté vacío
2: **for** $rule$ en RULES **do**
3: Separa $rule$ en el caracter antes de "=", c, y el string que hay después, s
4: Asigna $Reglas[c] = s$
5: **end for**
6: Inicializa $Paralabra_resultante \leftarrow$ AXIOM
7: Inicializa $Paralabra_temporal \leftarrow$ ""
8: **for** $iteration = 1, 2, \ldots$ ITERATIONS **do**
9: $Paralabra_temporal =$ ""
10: **for** $letra$ en $Paralabra_resultante$ **do**
11: **if** $Reglas[letra]$ existe **then**
12: $Paralabra_temporal \leftarrow Paralabra_temporal + Reglas[letra]$
13: **else**
14: $Paralabra_temporal \leftarrow Paralabra_temporal + letra$
15: **end if**
16: **end for**
17: $Paralabra_resultante \leftarrow Paralabra_temporal$
18: **end for**
19: **return** $Paralabra_resultante$

Finalmente, solo queda dibujar la cadena resultante. Para este propósito, en lugar de hacerlo como lo indican los gráficos tortuga, vamos a guardar los puntos y las rectas que se trazan para poder tener el dibujo de manera abstracta y así poder trazarlo en cualquier parte de la pantalla. El pseudocódigo para lograr esto se encuentra a continuación en donde se asume que se tienen implementadas funciones que realicen las actualizaciones mencionadas anteriormente.

Algorithm 2 Guardar los trazos definidos por una palabra

Input: PALABRA (Un string con la palabra a interpretar), α (la orientación inicial de la tortuga)

Output: Una lista con todos los puntos alcanzados por la tortuga y otra lista que contiene las parejas de puntos que formarán las líneas a trazar

Constantes: Tamaño de paso d y ángulo de incremento δ

1: Inicializar $Tortugas$ como un arreglo de estados de tortugas vacío, $Aristas$ como un arreglo de parejas de enteros vacío, $Puntos$ como un arreglo de puntos vacío y $Secuancia_De_Puntos$ como un arreglo de enteros vacío
2: Agregar a $Tortugas$ el estado $((0, 0), \alpha)$
3: Agregar a $Puntos$ el punto $(0, 0)$
4: Inicializar $Punto_actual \leftarrow 0$
5: **for** $letra$ en PALABRA **do**
6:     **switch** letra **do**
7:         **case** F
8:             Avanzar hacia el frente la tortuga una distancia d
9:             Agregar a $Puntos$ la nueva ubicación de la tortuga
10:             Agregar a $Aristas$ $(Punto_actual, Puntos.Length - 1)$
11:             $Punto_actual \leftarrow Punto_actual + 1$
12:         **case** f
13:             Avanzar hacia el frente la tortuga una distancia d
14:             Agregar a $Puntos$ la nueva ubicación de la tortuga
15:             $Punto_actual \leftarrow Punto_actual + 1$
16:         **case** $+$
17:             Rotar la tortuga δ grados
18:         **case** $-$
19:             Rotar la tortuga $-\delta$ grados
20:         **case** [
21:             Agregar a $Tortugas$ el estado actual de la tortuga
22:             Agregar a $Secuancia_De_Puntos$ el valor de $Punto_actual$
23:         **case**]
24:             $Punto_actual \leftarrow Secuancia_De_Puntos.End$
25:             Eliminar el último elemento de $Tortugas$
26:             Eliminar el último elemento de $Secuancia_De_Puntos$
27: **end for**
28: **return** $Puntos$, $Aristas$

Una vez implementados ambos algoritmos obtenemos un programa que usando la interfaz anteriormente mostrada puede trazar el dibujo obtenido en una sección específica de la pantalla. A continuación se muestra un ejemplo de lo que dicho programa puede lograr.

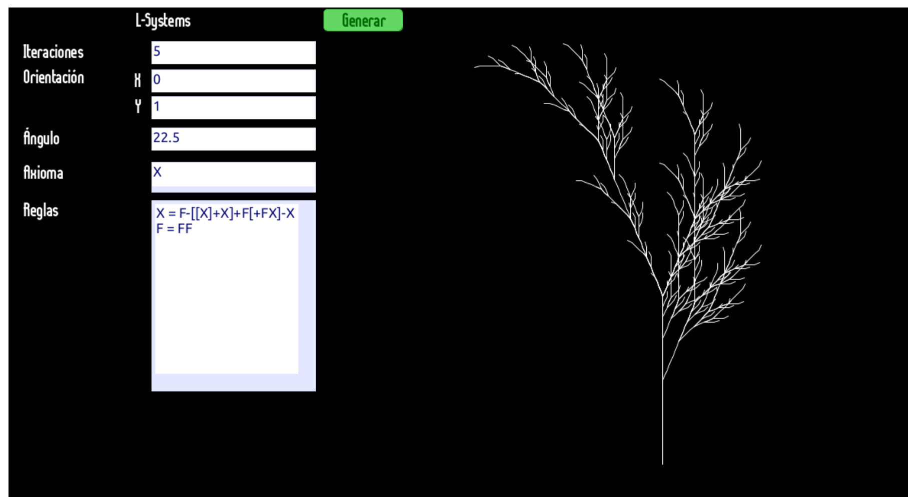


Figura 7. Dibujo generado (a la derecha) con el Sistema Lindenmayer ingresado a la izquierda

Notemos que se logró satisfactoriamente el motivo de la creación del programa pues pudimos emular correctamente una estructura de apariencia natural contenida en el capítulo 1.6.3 de "The Algorithmic Beauty of Plants".

Sistemas Lindenmayer Paramétricos

Para la siguiente parte del proyecto debemos definir una nueva extensión de los Sistemas Lindenmayer. En los *Sistemas Lindenmayer Paramétricos* ya no vamos a operar con cadenas de caracteres; ahora lo vamos a hacer con *cadenas de módulos*. Un *módulo* es simplemente un *caracter* con un *parámetro* asociado donde nuevamente los caracteres pertenecen a un *alfabeto* y los parámetros son números reales. Formalmente un módulo puede tener una cantidad ilimitada de parámetros pero nos concentraremos en aquellos que solo tienen uno o ninguno. Así, si A es un caracter y x un parámetro arbitrario, un módulo de un solo parámetro se denota por $A(x)$ y uno sin parámetros, solo por A. Por extensión, ahora el *axioma* será una secuencia de módulos, por ejemplo: F(1.1)AF(2.2)BCD(2.2).

Ahora, como vamos a operar con módulos, las reglas también deben cambiar. Estas ahora son de la forma *predecesor : condición → sucesor*. El **predecesor** es un módulo donde si este lleva parámetro se debe indicar que es arbitrario. En la **condición** debe ir alguna condición matemática que debe cumplir el parámetro del predecesor para poder aplicar la regla o bien, puede omitirse para que siempre sea válida. Finalmente, el **sucesor** es un cadena de módulos en donde a cada módulo se le da un nuevo valor como parámetro (si lleva) o se le da una modificación del parámetro que provee el predecesor. Estas modificaciones se llevan a cabo mediante sumas, restas, multiplicaciones, divisiones y exponenciaciones. Así, las reglas de un Sistema-L paramétrico pueden verse así:

1. F(x) : x > 2 → F(x*3)F(x+1.2)A(x^0.5)
2. A(x) → A(x/3.14159)B

Una regla se aplica sobre un módulo si el caracter del módulo coincide con el del predecesor y el número de parámetros del módulo es el mismo que el del predecesor. Por ejemplo, si comenzamos con el axioma F(3) después de una iteración obtendremos la cadena F(9)F(4.2)A(1.732) pero si hubiéramos comenzado con F(1) no se hubiera aplicado ninguna regla.

Por motivos de sencillez, en este proyecto solo trabajaremos con Sistemas-L Paramétricos con reglas que no lleven condición y, como ya se dijo, con módulos de a lo mucho un solo parámetro.

Ahora, de la misma manera que los Sistemas-L usuales, estos también pueden ser interpretados como instrucciones para gráficos tortuga pero en esta ocasión, como preámbulo a la siguiente sección, daremos su interpretación para una tortuga en el **Espacio Tridimensional**.

En tres dimensiones ya no será suficiente una tripleta y en su lugar usaremos 4 elementos para describir un estado de la tortuga: (p, H, L, U), donde p es un punto que indica su posición en el espacio y H, L y U son tres vectores unitarios que representan (respecto a la tortuga) dónde está su *frente*, su *izquierda* y su *arriba*, respectivamente, y además satisfacen que $H \times L = U$.

Las rotaciones ya no se pueden representar simplemente con una suma de ángulos y en su lugar usaremos la siguiente fórmula matricial para calcular la nueva orientación de la tortuga:

$$[H'\, L'\, U'] = [H\, L\, U]R$$

Donde R es cierta matriz de rotación de 3 x 3. Si queremos definir una rotación de ángulo α alrededor de los vectores U, L y H usaremos las matrices

$$R_U = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) & 0 \\ -\sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$R_L = \begin{pmatrix} \cos(\alpha) & 0 & -\sin(\alpha) \\ 0 & 1 & 0 \\ \sin(\alpha) & 0 & \cos(\alpha) \end{pmatrix}$$

$$R_H = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) \\ 0 & \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

respectivamente.

Como ahora tenemos parámetros, ya no es necesario usar dos símbolos para indicar la dirección de la rotación, pues podemos pasar el ángulo o su inverso aditivo para lograr este efecto. Es así que la tortuga obedecerá a las siguientes pocas reglas:

- F(a) Se moverá hacia adelante a unidades. Es decir, su estado cambiará a (p', H, L, U) donde $p' = p + aH$ y además, se trazará una línea entre p y p'.
- f(a) Se moverá hacia delante a unidades sin trazar una línea.
- +(a) Girará sobre U un ángulo de a grados usando la matriz $R_U(a)$.
- &(a) Girará sobre L un ángulo de a grados usando la matriz $R_L(a)$.
- /(a) Girará sobre H un ángulo de a grados usando la matriz $R_H(a)$.

Pero esto no es todo, podemos extender estas reglas aún más y agregar las siguientes:

- !(a) Aumenta el grosor del trazo a a.
- $ Rota la tortuga para que L quede en posición horizontal. Esto se logra haciendo $L = \frac{V \times H}{|V \times H|}$ y $U = H \times L$, donde V es el vector $(0, 1, 0)$.

Y así, como su equivalente bidimensional, también podemos extender los Sistema-L paramétricos a aceptar corchetes, obteniendo los *Sistemas Lindenmayer Paramétricos Encorchetados* donde los corchetes fungen exactamente la misma función. Todo esto nos permite construir estructuras complejas y muy familiares:

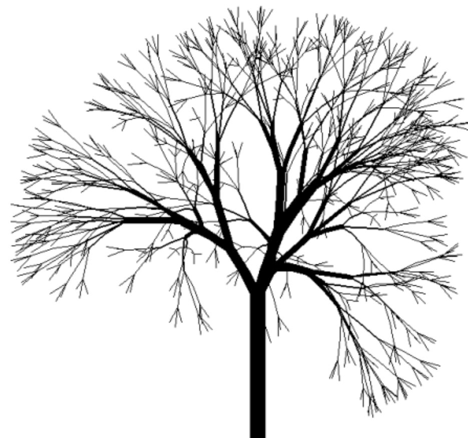


Figura 8. Figura parecida a un árbol generada por Sistemas-L paramétricos. Extraída de "The algorithmic Beauty of Plants" ←—---

Es por esto que el segundo objetivo de este proyecto fue el generar un programa que permita a los usuarios crear árboles medianamente realistas pero que también permita experimentar con los Sistemas Lindenmayer Paramétricos Encorchetados usuales mediante la creación de estructuras tridimensionales, el cual se realizó usando el motor Unity.

Sistemas Lindenmayer Paramétricos Encorchetados en Unity

Una vez más, lo primero que debemos hacer es solicitarle al usuario las partes que definen al Sistema Lindenmayer y lo necesario para que la tortuga pueda dibujar. A continuación explicamos las suposiciones y herramientas utilizadas para ingresar satisfactoriamente los datos requeridos.

Asumimos nuevamente que la tortuga siempre comienza en el punto $(0, 0, 0)$ y los vectores H, L, U tienen por valor temporal inicial

- $H = (0, 1, 0)$
- $L = (-1, 0, 0)$
- $U = (0, 0, 1)$

Debemos entonces solicitar la orientación inicial de la tortuga para sustituir estos valores temporales. En esta ocasión la pedimos mediante un vector V, que asignamos eventualmente a H después de normalizarlo. Pero notemos que ya no se cumple que $H \times L = U$, para corregir esto vamos a obtener el eje de rotación y el ángulo por el cual $(0, 1, 0)$ pasó a ser V y así aplicarle dicha rotación al vector L para finalmente calcular U como el productor cruz de H y L. Una vez hecho esto, tendremos el estado inicial de la tortuga.

Ahora, ingresar el sistema-L es mucho más complicado que en el programa anterior. Vamos a limitar los símbolos en su alfabeto a las letras del alfabeto inglés y los símbolos que se usan como instrucciones de la tortuga. Teniendo esto en cuenta, para lograr verificar si un axioma o una regla es válida usaremos nuevamente expresiones regulares.

Diremos que un **axioma es válido** en este caso si

- Coincide con la siguiente expresión regular:

  ```
  ^((\[([a-wy-zA-Z\+\&\/\$\!](\(([-]?(?:\d*\.*\d+))\))?)+\])|(([a-wy-zA-Z\+\&\/\$\!](\(([-]?(?:\d*\.*\d+))\))?)+))$
  ```

  Notemos que permitimos los símbolos ya dichos exceptuando la x y pueden o no tener parámetro. Esto puede causar problemas pero más adelante se le dará solución. El hecho de exceptuar la x como caracter válido de un módulo es porque la vamos a utilizar en las reglas como indicador de

un parámetro arbitrario. En este caso no permitimos corchetes anidados porque esto requeriría una expresión demasiado larga que puede afectar con el rendimiento del programa.

- Y las letras no tienen redefiniciones. Es decir, si a una letra no se le asigna parámetro, todas las demás veces que aparece en el axioma también debe aparecer sin parámetro y viceversa.

Ahora, diremos que una **regla es válida** si

- Coincide con la siguiente expresión regular:

```
^[a-wy-zA-Z\+\&\$\/\!](\(x\))?=((\[([a-wy-zA-Z\+\&\/\$\!](\(((x([\+\-\^\*\/]([-]?(?:\d*\.*\d+)))?)|(([-]?(?:\d*\.*\d+))))\))?)+\])|(([a-wy-zA-Z\+\&\/\$\!](\(((x([\+\-\^\*\/](([-]?(?:\d*\.*\d+))))?)|(([-]?(?:\d*\.*\d+))))\))?)+))+$
```

  Nuevamente usaremos el símbolo “=” en lugar de la flecha para separar los elementos de la regla. En esta ocasión, como predecesor permitimos letras (excepto la x) así como los símbolos especiales. Esto pues puedes querer aumentar el grado de rotación por cierta constante en cada iteración; notaremos que el módulo del predecesor puede llevar x como parámetro o no llevar nada. Como sucesor permitimos todo lo que se considera un axioma válido así como también los módulos que indican su parámetro como actualización del parámetro del predecesor; Notemos que si se aplicará una operación al parámetro del predecesor para crear un módulo en el sucesor, el operador siempre debe ir después de la x.

- Las letras no tienen redefiniciones. Es decir, si a una letra no se le asigna parámetro, todas las demás veces que aparece en el axioma también debe aparecer sin parámetro y viceversa.
- Y no se usa lo que no se da. Es decir, si algún módulo del sucesor usa el parámetro x, entonces el predecesor debe contener el parámetro x.

Podemos entonces definir que un **conjunto de reglas es válido** si

- Es vacío.
- Cada una de las reglas es válida.
- No existen redefiniciones cruzadas. Es decir, si en la regla i aparece el caracter A por primera vez con parámetro, entonces para toda j > i, A debe llevar parámetro y viceversa.
- Las reglas llevan a un Sistema-L determinístico.

Una vez habiendo definido estos conceptos, podemos solicitar al usuario el axioma y las reglas en forma de texto para así guardarlos para su posterior uso, obteniendo finalmente el Sistema-L. Sin embargo, aún falta determinar qué cadena vamos a dibujar y es por eso que también le pedimos al usuario el total de veces que las reglas se aplicarán sobre el axioma.

Todos los parámetros ya mencionados son suficientes para dibujar un Sistema-L paramétrico pero recordemos que queremos generar árboles. Es por esto que debemos definir un nuevo parámetro: El *Tropismo*. En biología, esta es la influencia del ambiente sobre el crecimiento de una planta pero aquí lo consideraremos como un vector [2]. Si T es la dirección del tropismo, cada que avance la tortuga ajustaremos su orientación moviendo el vector H hacia el vector T un ángulo de $\alpha = e|H \times T|$ donde e es la susceptibilidad del árbol a doblarse. Entonces, el tropismo se ingresará como un vector y la susceptibilidad como un número real.

Además, para agregar más detalles al árbol, agregaremos una instrucción a la tortuga:

- A   Agrega una hoja en la dirección en la que mira la tortuga en ese instante.

Tomando todas estas consideraciones, ya solo queda construir un lugar en donde el usuario pueda ingresar todos estos parámetros. Para esto se utilizaron las herramientas de GUI presentes en Unity que permitieron generar una interfaz de usuario que solicite todos los parámetros ya expuestos y que podemos ver a continuación.

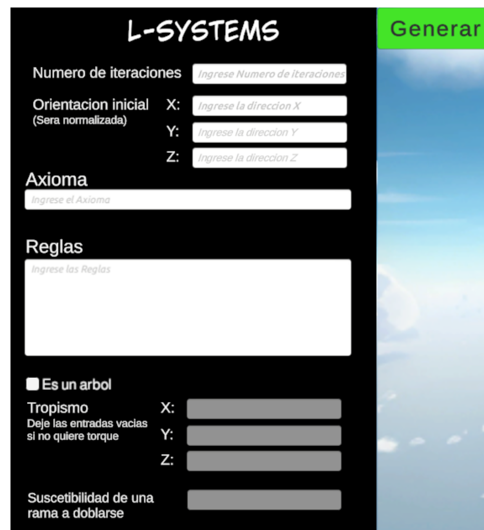


Figura 9. Interfaz creada en Unity usando sus herramientas de Interfaz Gráfica

Notemos que tenemos un botón para indicar si queremos generar un árbol o no. Esto es porque para más realismo se decidió usar textura de madera al momento de trazar el árbol por lo que si no se quiere generar un árbol simplemente se quitará esta textura y también se ignorará el valor del tropismo, así como tampoco se agregarán hojas a la estructura.

Hasta este momento, el axioma y las reglas se tienen en forma de texto, sin embargo, en esta ocasión no es suficiente pues debemos tener una forma de guardar el parámetro asociado al caracter así como poder aplicar reglas que ahora pueden modificar dichos parámetros. A continuación mostramos una forma en la que se puede solucionar estas dificultades.

Primeramente necesitamos una clase **Módulo** que contenga: una variable para guardar el caracter, una variable que indique si tiene parámetro o no y otra para guardar el parámetro.Con esta clase ya somos capaces de guardar un axioma válido pero las reglas, al poder indicar actualización de parámetros, requerirán más trabajo.

Para guardar las reglas válidas usaremos dos clases. La primera será **Átomo**, la cual representará a los elementos más fundamentales que componen el sucesor de una regla. Estos guardarán el caracter que se debe sustituir, una variable para indicar si hay parámetro o no y otras dos que indican que operación se debe hacer sobre el parámetro que apareció en el teórico predecesor y con qué valor se debe operar. Se deben agregar dos operadores extras que indiquen el que no se modifique el parámetro del predecesor y el que se cree un valor nuevo. Los Átomos tendrán una función que puede ser llamada públicamente para que al pasarle un Módulo, se cree otro Módulo usando las instrucciones codificadas en las variables.

Una vez creada la clase Átomo, podemos crear la clase **Regla**. Esta contendrá una variable para indicar la letra en el predecesor y otra para indicar si dicha letra contiene parámetro o no. Además, el sucesor se representará como un vector ordenado de Átomos. Estas variables las creará a partir de una regla válida. Finalmente, la clase debe tener una función que pueda ser llamada públicamente que permita verificar si dado cierto módulo M, la regla es aplicable; y si lo es, proceder a generar una nueva cadena de módulos usando las transformaciones codificadas en los Átomos, pasando como parámetro el módulo M a la función pública mencionada anteriormente.

Con estos elementos ya somos capaces de procesar el axioma y las reglas y obtener la cadena resultante de Módulos se vuelve similar a como ya lo habíamos hecho:

Algorithm 3 Aplicar reglas de un Sistema Lindenmayer Paramétrico Encorchetado

Input: AXIOM (Un axioma válido), RULES (Un conjunto de reglas válido), ITERATIONS (Las veces que se van a aplicar las reglas)

Output: Un vector de módulos que representa la cadena resultante.

1: Inicializar $Mod_Resultante$ como un vector de la clase **Módulo** que contiene todos los módulos de AXIOM
2: Inicializar $Reglas$ como un vector de la clase **Regla** que contiene todas las reglas en RULES
3: Inicializar $Mod_Temporal$ como un vector de **Módulos** vacío.
4: **for** $iteration = 1, 2, \ldots$ ITERATIONS **do**
5: Vaciamos $Mod_Temporal$
6: **for** $Modulo$ en $Mod_Resultante$ **do**
7: $transformado \leftarrow false$
8: **for** $Regla$ en $Reglas$ **do**
9: **if** $Regla$ es aplicable sobre $Modulo$ **then**
10: $transformado \leftarrow true$
11: Agregar el resultado de aplicar $Regla$ sobre $Modulo$ a $Mod_Temporal$
12: **end if**
13: **break**
14: **end for**
15: **if** $\neg transformado$ **then**
16: Agregar $Modulo$ a $Mod_Temporal$
17: **end if**
18: **end for**
19: $Mod_Resultante \leftarrow Mod_Temporal$
20: **end for**
21: **return** $Mod_Resultante$

Finalmente, solo queda dibujar siguiendo la cadena de módulos resultante. Para lograr esto, nuevamente guardaremos los puntos y las aristas en arreglos para poder tener el resultado de manera abstracta y así poder modificarlo para que cuando usemos la cámara de Unity para poder visualizarlo, logremos que aparezca en su totalidad en la escena. Pero además, también guardaremos la ubicación de las hojas, así como su orientación para poder colocarlas en el árbol. El algoritmo para obtener estos arreglos es análogo al utilizado para la versión en dos dimensiones y se encuentra a continuación.

Algorithm 4 Guardar los trazos definidos por un vector de Módulos

Input: CADENA (Un vector de Módulos que representa la cadena resultante), L_i, H_i y U_i (Vectores que representan las orientación inicial), T (Un vector que representa el Tropismo), e (Un real que representa la susceptibilidad a doblarse)

Output: Una lista con todos los puntos alcanzados por la tortuga, otra lista que contiene las parejas de puntos que formarán las líneas a trazar y una lista con la posición y dirección de todas las hojas.

1: Inicializar $Tortugas$ como un arreglo de estados de tortugas vacío, $Aristas$ como un arreglo de parejas de enteros vacío, $Puntos$ como un arreglo de puntos vacío, $Hojas$ como un arreglo de parejas que contengan un punto y un vector, $Secuancia_De_Puntos$ como un arreglo de enteros vacío
2: Inicializar $Grosor \leftarrow 1$
3: Agregar a $Tortugas$ el estado $((0,0,0), L_i, H_i, U_i)$
4: Agregar a $Puntos$ el punto $(0,0,0)$
5: Inicializar $Punto_actual \leftarrow 0$
6: **for** $Modulo$ en CADENA **do**
7: **switch** $Modulo$.caracter **do**
8: **case** F
9: Av
10: Agregar a $Puntos$ la nueva ubicación de la tortuga
11: Agregar a $Aristas$ $(Punto_actual, Puntos.Length - 1)$
12: $Punto_actual \leftarrow Punto_actual + 1$
13: **case** f
14: Avanzar hacia el frente la tortuga una distancia $Modulo$.parámetro
15: Agregar a $Puntos$ la nueva ubicación de la tortuga
16: $Punto_actual \leftarrow Punto_actual + 1$
17: **case** $+$
18: Girar la tortuga sobre U $Modulo$.parámetro grados
19: **case** &
20: Girar la tortuga sobre L $Modulo$.parámetro grados
21: **case** /
22: Girar la tortuga sobre H $Modulo$.parámetro grados
23: **case** A
24: Agregar a $Hojas$ la pareja formada por la posición actual de la tortuga y su vector H.
25: **case** !
26: $Grosor \leftarrow Modulo$.parámetro
27: **case** $
28: Rotar la tortuga para llevar al vector L a posición horizontal
29: **case** [
30: Agregar a $Tortugas$ el estado actual de la tortuga
31: Agregar a $Secuancia_De_Puntos$ el valor de $Punto_actual$
32: **case**]
33: $Punto_actual \leftarrow Secuancia_De_Puntos.End$
34: Eliminar el último elemento de $Tortugas$
35: Eliminar el último elemento de $Secuancia_De_Puntos$
36: **end for**
37: **return** $Puntos$, $Aristas$, $Hojas$

Una vez implementados estos algoritmos en Unity pudimos obtener un programa que, usando la interfaz anteriormente dada, logra generar estructuras que se asemejan a los árboles reales. Para trazar las aristas se decidió usar cilindros los cuales se adornaron con una textura de madera. Esto permitió generar estructuras como la siguiente.

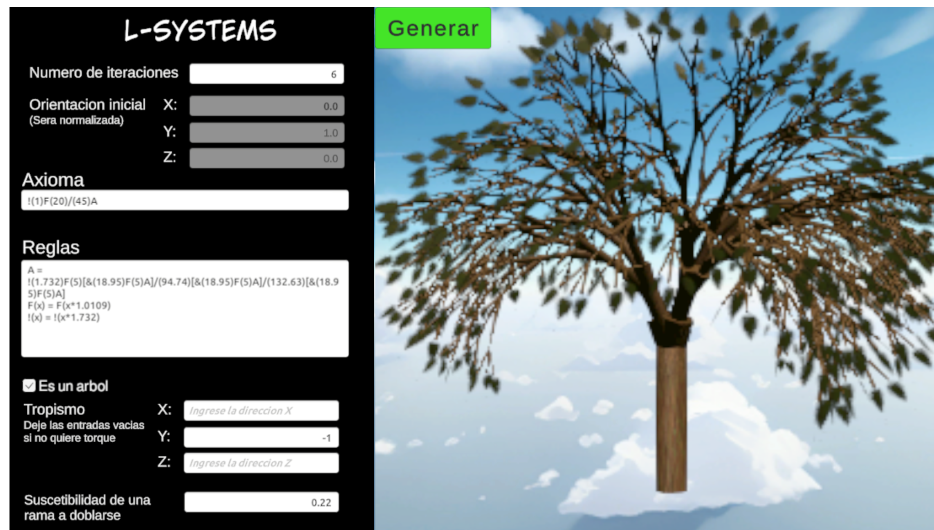


Figura 10. Árbol generado con un Sistema Lindenmayer paramétrico

Pero también otras no necesariamente naturales, como una aproximación de la Curva de Hilbert.

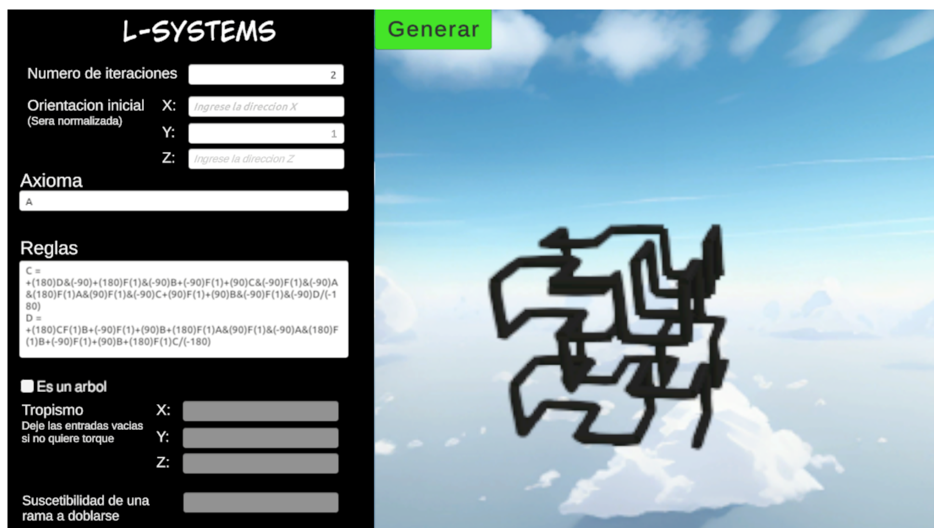


Figura 11. Aproximación simple a la curva de Hilbert usando Sistemas Lindenmayer

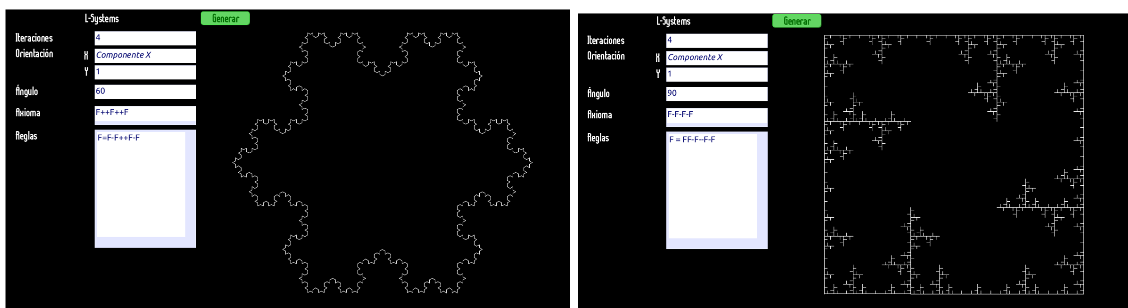
Nuevamente podemos asegurar que se consiguió lograr el objetivo del proyecto pues logramos imitar uno de los árboles contenidos en el Capítulo 2 del libro “The Algorithmic Beauty of Plants”.

Resultados

Como se mencionó en un inicio, lo que se buscaba en este proyecto era lograr replicar algunos de los sistemas biológicos contenidos en el libro "The Algorithmic Beauty of Plants". Esto se consiguió satisfactoriamente pues se consiguieron imitar en su totalidad los sistemas del capítulo 1.6.3 utilizando la implementación en Processing. Los resultados los podemos ver a continuación:

Figuras 12-17. Estructuras similares a plantas generadas con Sistemas-L encorchetados

Pero esto no es todo, este programa también es capaza de generar diversas aproximaciones a fractales como vemos ahora:



Figuras 18-21. Aproximación a Fractales logradas con Sistemas-L

Así mismo, usando la implementación en Unity se lograron replicar satisfactoriamente los 4 árboles de la figura 2.8 del capítulo 2.

Figuras 22-25. Estructuras parecidas a árboles generados por Sistemas-L paramétricos

Conclusiones

Como ya se sostuvo en diversas partes del proyecto, los programas se obtuvieron satisfactoriamente pues permitieron recrear las estructuras generadas por Lindenmayer y Prusinkiewicz en “The Algorithmic Beauty of Plants” y lo que es más, estos tienen un alto grado de interacción con el usuario lo cual permite que los programas sean una excelente forma en que las personas ajenas a los Sistemas Lindenmayer puedan aprender sobre estos, inventar sus propios sistemas y visualizarlos sin la necesidad de saber programar.

Aunque cumplieron con su objetivo, los programas tienen un vasto espacio para mejoras. Por ejemplo, permitir Sistemas-L estocásticos o Sistemas dónde más de una letra le indique a la Tortuga que trace una línea para obtener estructuras más complejas. En cuanto al apartado estético, el programa implementado

en Unity tiene varias fallas, como no renderizar las estructuras en una definición deseable o que la transición entre ramas de diferente ancho es abrupta, cosa que afecta a la credibilidad del modelo. Aunque se trató de mejorar la definición, no se encontró la manera de lograrlo por lo que se recurrió al antialiasing lo que provoca que los modelos se vean un poco distorsionados. En el caso de la anchura de las ramas, usar curvas que puedan variar su ancho a lo largo del trazo parece ser la solución sin embargo, esto requeriría cambiar la forma de construir el árbol y aunque se consideró utilizarlas, por falta de tiempo no se llegó a implementar.

La belleza de la naturaleza nunca dejará de fascinar al ser humano, desde la forma de objetos tan pequeños como los copos de nieve, hasta la forma y estructura de cosas tan grandes como los árboles. Aunque en un principio se podría pensar que alcanzan tal belleza por regirse con reglas muy complejas, el poder imitarlos con los Sistemas Lindenmayer nos enseña que tal vez dicha belleza proviene de la repetición de estructuras y reglas que no escapan de nuestra comprensión. Recordemos que uno de los patrones más populares para indicar la belleza de las plantas, la sucesión de Fibonacci, no es más que sumar los dos últimos números para obtener el siguiente.

Bibliografía/Referencias

[1] Shaker, N., Togelius, J., Nelson M. J. (2016) Procedural Generation in Games. Springer International Publishing.

[2] Prusinkiewicz, P. y Lindenmayer A. (2004) The Algorithmic Beauty of Plants. Springer. http://www.algorithmicbotany.org/papers/abop/abop.pdf

[3] Google. (s.f.). *Acerca de las expresiones regulares (regex).* https://support.google.com/analytics/answer/1034324?hl=es