Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Desarrollo de lenguaje de dominio específico para generación procedural

Development of domain-specific language for procedural generation

Fabio Ovidio Bianchini Cano

La Laguna, 2 de julio de 2022

D. Jesús Miguel Torres Jorge con N.I.F. 43.826.207-Y profesor Contratado Doctor adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor

C E R T I F I C A (N)

Que la presente memoria titulada:

"Desarrollo de lenguaje de dominio específico para generación procedural"

ha sido realizada bajo su dirección por D. **Fabio Ovidio Bianchini Cano**,

con N.I.F. 54113412-R.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 2 de julio de 2022

Agradecimientos

En primer lugar quiero agradecer de todo corazón a Jesús Torres, por ayudarme con la elección del trabajo y por sus buenos y precisos consejos que hicieron del desarrollo un reto pero nunca un objetivo inalcanzable.

A mi familia, porque incluso en los momentos más duros que he vivido en la carrera, confiaron en mi fuerza de voluntad para sobreponerme y al fin conseguir terminarla.

A mi amigo Mario, por su fe en mí, por su preocupación y en definitiva por ser otra mano más a la que poder agarrarme cuando el camino se hacía cuesta arriba.

Y finalmente, a mi pareja, Lucía, porque compartiste mi carga, porque cuidaste de mí y porque sin ti no sé qué habría sido de mí.

Licencia

Resumen

El objetivo de este trabajo ha sido dar algo más de profundidad al concepto de generación por procedimientos, tema del que hice ya un trabajo dentro de la asignatura Interfaces Inteligentes, y en el que he decidido profundizar en éste. La idea con este trabajo es crear una herramienta de generación procedural que sea modificable a través de un lenguaje de dominio específico. Con esto trato de hacer la herramienta moldeable y fácilmente traducible a usuarios que no tengan tanta experiencia con lenguajes de programación, y por ello la introducción del DSL.

Palabras clave: DSL, lenguaje, dominio, específico, generación, procedimientos, procedural.

Abstract

The objective of this project is to give a little more depth to the concept of procedural generation, a topic on which I already did a piece of work within the Intelligent Interfaces subject, and in which I have decided to delve into it. The idea with this project is to create a procedural generation tool that is modifiable through a domain-specific language. With this I try to make the tool moldable and easily translatable to users who do not have much experience with programming languages, and therefore the introduction of DSL.

Keywords: Language, domain, specific, generation, procedures, procedural

Índice general

Índice de ilustraciones

Índice de tablas

Glosario

PCG: Generación por procedimientos *(Procedural Content Generation)*.

DSL: Lenguaje de dominio específico (*Domain Specific Language)*.

Procedural, procedimientos, generación, lenguaje, malla, ruido, normalización,

Capítulo 1
Introducción

1.1 Objetivos

Con esta aplicación mi objetivo es rellenar un hueco dentro del mercado que, si bien es muy específico, todavía nadie ha elaborado.

Como ya he comentado, este proyecto nace como inspiración al proyecto de GIGL. Sin embargo, el mío está completamente orientado a crear un lenguaje de dominio específico que permita la creación de un terreno 3D completamente personalizable.

Podemos subdividir el proyecto en dos importantes subproyectos.

- Crear una herramienta de generación procedural en el que se represente un terreno 3D de la manera más natural posible.
- Desarrollar un lenguaje de dominio específico que permita modificar todas las variables que hacen referencia al terreno. Debe ser lo más legible y fácil de utilizar que se pueda.

Este trabajo está orientado a toda clase de usuarios indistintamente. Tanto aquellos con conocimientos sobre informática y *PCG* como usuarios que no tienen tanta idea de programación informática.

1.2 Fases

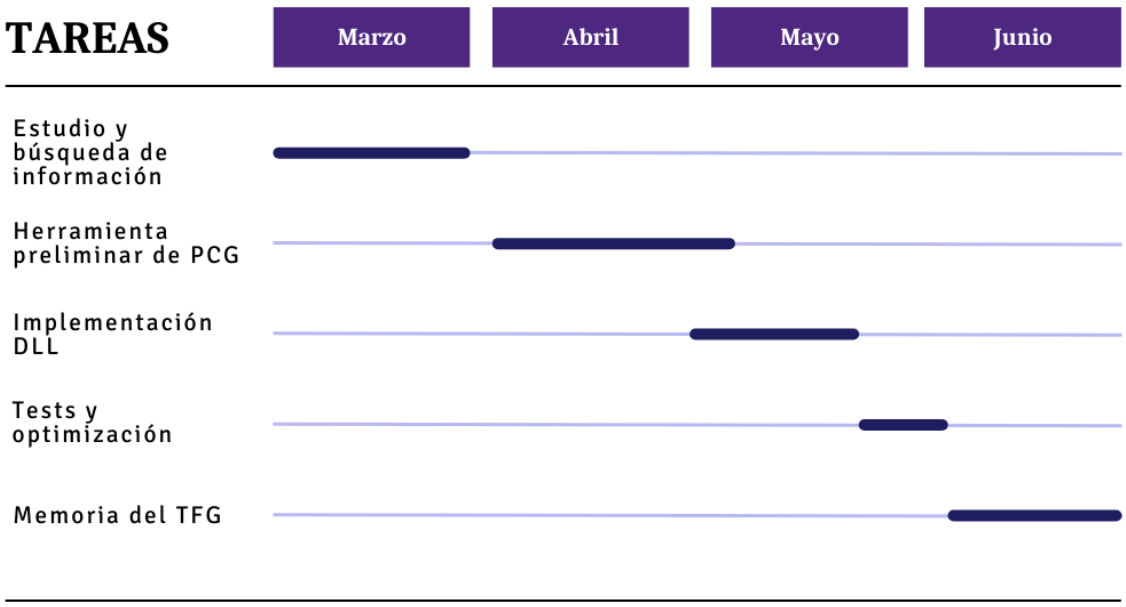


Tabla 1: Cronograma del proyecto

1.3 Generación procedural

La generación procedural o por procedimientos *(PCG)* es una técnica de creación de datos con algoritmos de manera automática. En gráficos por ordenador se utiliza comúnmente para creación de texturas o modelados. En videojuegos también está muy presente para crear todo tipo de contenido, desde terrenos naturales, artículos, misiones, personajes, etc.

Su uso está muy extendido a día de hoy, sin embargo, el concepto nace en parte como solución a un problema. En las últimas décadas los desarrolladores gráficos y de videojuegos han tenido que hacer frente a una demanda mucho más alta de contenido con los mismos plazos de entrega y, posiblemente, mismo número de trabajadores que antes. Esto ha supuesto un crecimiento notable en el volumen de trabajo con el que no podían lidiar de la misma manera. Se precisaba una solución que permitiera a las desarrolladoras incrementar el volumen de contenido dispensado sin encarecer notablemente los costes.

Este problema afecta a todos por igual. A las grandes empresas con proyectos muy grandes, por ejemplo, juegos de mundo abierto, con superficies kilométricas y donde la cantidad de contenido a modelar es inmensa. A los estudios independientes con menos personal también. Todo lo que les ahorre tiempo y dinero pueden invertirlo en crear juegos con más alcance o invertirlo en visibilidad.

Como solución orgánica a este problema nace la generación procedural. Un método de creación de contenido que, en base a unas reglas y parámetros, un ordenador puede utilizar para generar datos a un ritmo mucho más acelerado que cualquier desarrollador.

Por suerte, con el paso del tiempo hemos sido testigos de cómo esta técnica no solo se ha hecho un hueco en la industria como sustituta de nuestras hábiles, pero no tan rápidas manos. Surgieron métodos de creación propios, técnicas complementarias, inclusive géneros nuevos que solo son posibles gracias a la generación procedural.

Ilustración 1: Synthwave

Klaus C. (2022) Synthwave: Una historia de retrofuturismo. Recuperado de https://urbe01.net/synthwave/

Ilustración 2: Terreno de Minecraft

Minecraft Fandom. Recuperado de https://minecraft.fandom.com/es/wiki/Superficie

1.4 Lenguaje de Dominio Específico

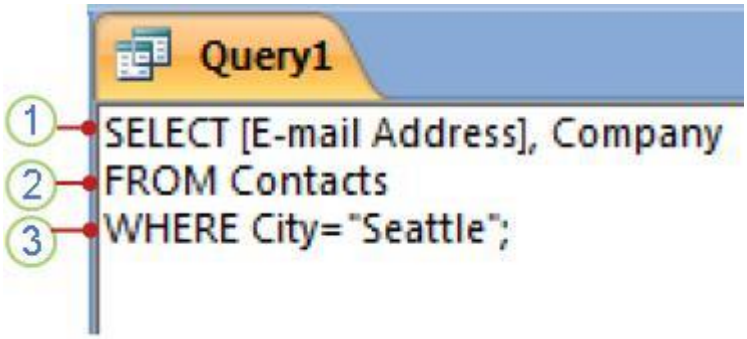
Los lenguajes pueden clasificarse en lenguajes de uso general *(GPL)* o de dominio específico *(DSL)*. Los primeros son multidisciplinares, capaces de solucionar múltiples problemas de muchas maneras diferentes. Se pueden utilizar para escribir aplicaciones autónomas, programas e interfaces.

Sin embargo, hay situaciones en las que un GPL no es la mejor solución.

En informática, como en la construcción, el uso de la herramienta correcta puede marcar la diferencia. Un lenguaje de dominio específico es un lenguaje especializado usado para un propósito específico.

Un ejemplo de DSL podría ser SQL *(Structured Query Language)*, el cual se utiliza como lenguaje de relación con bases de datos. Fuera de este ámbito no se usa, pero es muy potente en la tarea para la que ha sido desarrollado.

Una de las ventajas más destacables de los lenguajes de dominio específicos sobre los de propósito general, es que normalmente pueden ser usados tanto por programadores como por usuarios que no tengan tanta relación con la informática. El lenguaje en sí mismo se suele acercar bastante al lenguaje natural (inglés, habitualmente).



Un usuario con un poco de conocimiento básico sobre el funcionamiento de las tablas y con un poco de imaginación es capaz de comprender búsquedas sencillas, debido a la similitud entre este lenguaje y el nuestro.

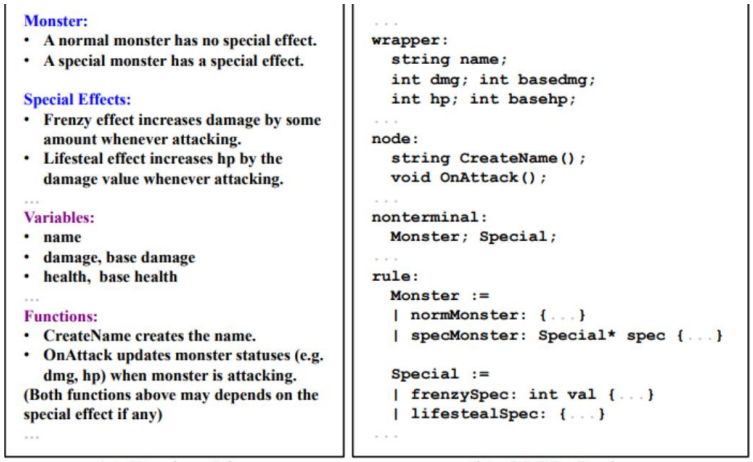
Ilustración 3: Ejemplo básico de SQL

Access SQL: Basic concepts, vocabulary and syntax https://support.microsoft.com/en-us/office/access-sql-basic-concepts-vocabulary-and-syntax-444d0303-cde1-424e-9a74-e8dc3e460671

Capítulo 2
Antecedentes y estado del arte

2.1 Fusión de ambos conceptos

La idea de mezclar ambos conceptos y realizar la aplicación provino de otra que sirvió como inspiración a ésta. GIGL *(Grammatical Item Generation Language)* es también un lenguaje de dominio específico para generar contenido de manera procedimental. Tal y como explican los autores de GIGL: “Muchos de los problemas del *PCG* son inherentemente jerárquicos, y como tal, pueden ser expresados como gramáticas, lo cual provee una manera natural de expresar la relación entre sus componentes”Incorporando gramáticas a la creación de estos objetos, podemos conseguir una representación compacta de dichos objetos complejos, y a su vez reutilizar las estructuras para mejorar la eficiencia computacional



(a) Design Idea (b) GIGL Code

Ilustración 4: Ejemplo de GIGL

Tiannan C. Stephen J. (2018) GIGL: A Domain Specific Language for Procedural Content Generation with Grammatical Representations. University of Minnesota

En estas reglas se basa también mi aplicación, sin embargo, exclusivamente hablando de terreno.

```
public static DLL Script() {

  DLL testDLL =
    DLL.describedAs
      .zoom(43.2f)
      .octaves(5)
      .persistance(0.4f)
      .lacunarity(2f)
      .seed(23)
      .addRegion(0f, "feae34", 0.002f)
      .addRegion(0.3f, "f77622", 0.002f)
      .addRegion(0.5f, "ce4121", 0.002f)
      .addRegion(0.7f, "88352b", 0.002f);

  return testDLL;
}
```

Ilustración 5: Ejemplo de DSL para la aplicación. Elaboración propia.

2.2 Otras aplicaciones

Convivimos diariamente con aplicaciones PCG que nos hacen la vida más fácil. No solo modelando, sino creando estructuras.

Este avance fue tal que incluso dio lugar a la creación de nuevos géneros dentro de este campo. *Rogue-like* nace como un género que hace referencia en su propio nombre al título del juego que popularizó las claves del mismo: *Rogue*. Estos juegos generan estructuras mientras jugamos en función de una serie de reglas preestablecidas.

Ilustración 6: Mapa del videojuego The Binding of Isaac

Hemos nombrado ejemplos relacionados con los videojuegos puesto que es uno de los campos más explotables. Sin embargo, en el cine también está muy presente. Los innumerables ejércitos de las películas de El Señor de los Anillos también están modelados gracias a esta tecnología.

Ilustración 7: Ejército de la película El Señor de los Anillos

Capítulo 3
Generación procedural en profundidad

En este capítulo explicaré a fondo cómo funciona la PCG en mi aplicación, cuáles son sus bases y algoritmos centrales, las variables importantes sobre las que se basa y demás detalles.

3.1 Naturalidad y cómo conseguirla

La generación por procedimientos, como su propio nombre indica, genera contenido en base a unas reglas. Dicha generación tiene un cierto componente aleatorio, puesto que, si no, no existiría variación. Éste es el primer punto importante a tener en cuenta. La generación aleatoria de contenido se basa, por supuesto, en generación aleatoria de números. O, mejor dicho, pseudoaleatoria.

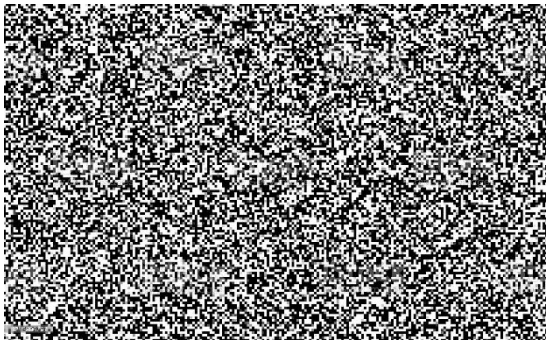


Ilustración 8: Fondo blanco y negro aleatorio

Recuperado de https://www.freejpg.com.ar/imagenes/premium/1256396797/patrr-n-aleatorio-en-blanco-y-negro

Si, para la representación del terreno en 3D, que es lo que queremos conseguir, basáramos la reproducción de números en un algoritmo completamente aleatorio, nos quedaría una imagen parecida a ésta.

La antinaturalidad es algo que debemos evitar, por ello la PCG debe estar estrechamente ligada a una generación estrictamente controlada. Al fin y al cabo, queremos hacer una representación lo más acorde a la entropía natural.

Esto lo hacemos ayudándonos del principal algoritmo que constituye esta aplicación, el algoritmo de Perlin Noise.

3.2 Perlin Noise

Ken Perlin fue el desarrollador de la función matemática que hoy conocemos como Perlin Noise. Ken trabajaba durante 1983 en la película *Tron*, que fue animada artificialmente. Éste algoritmo fue

el resultado de su frustración al solo poder conseguir texturas que pareciesen generadas por ordenador, artificiales y poco naturales. Desarrolló durante los años posteriores el algoritmo hasta en 1997 conseguir el *Academy Award for Technical Achievement.*

Tal y como en su momento creó este algoritmo para solucionar el propio problema de la antinaturalidad de las texturas, nosotros lo utilizaremos con el mismo propósito, combatir la antinaturalidad de la generación de números aleatorios.

El funcionamiento es sencillo. Dados un mínimo y un máximo intraspasables, cada generación de un número está estrictamente relacionada con la anterior. Componiendo una horquilla, que, dependiendo de su valor, la generación será más suave o más brusca.

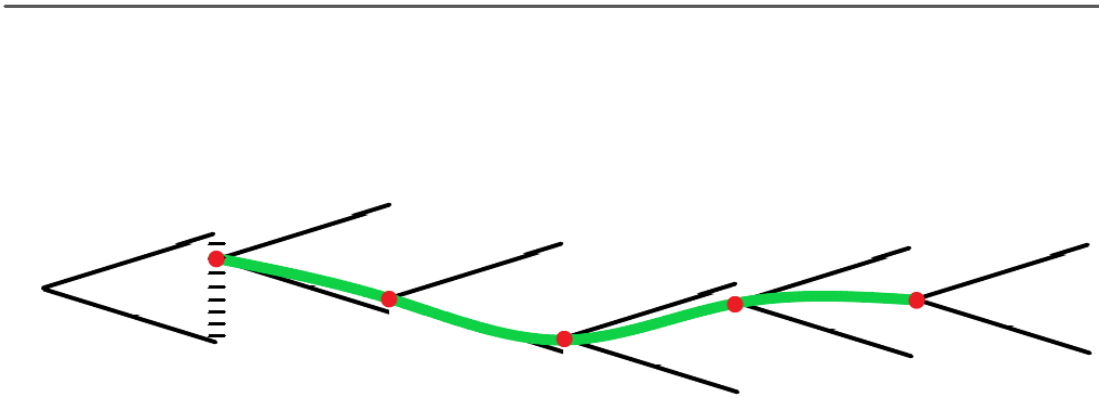


Ilustración 9: Ejemplo de generación de terreno con Perlin Noise. Elaboración propia.

Con este algoritmo, la ilustración 6, la podemos transformar en algo como esto:

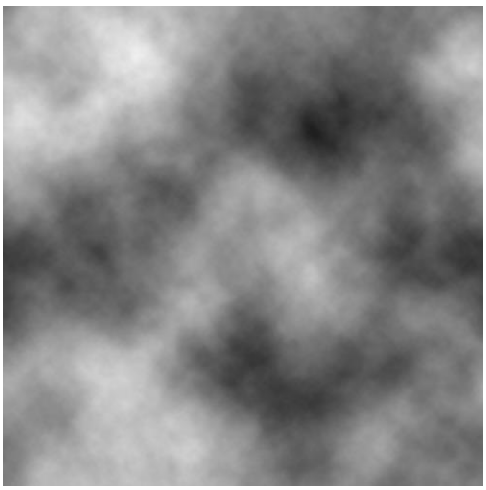


Ilustración 10: Nube de Perlin Noise

Jens K. (2020) Understanding basic noise terms. Recuperado de *http://www.campi3d.com/External/MariExtensionPack/userGuide5R4v1/Understandingsomebasicnoiseterms.html*

Si bien a simple vista no nos recuerda a ningún elemento de la naturaleza, podemos apreciar que hay concentraciones de números más bajos que otros, ésos siendo representados por las zonas más oscuras de la imagen. No hay un salto de luminosidad muy brusco entre pixel y pixel. Así solucionamos el principal problema.

Sin embargo, el ruido Perlin tiene muchas variables que tocar y modificar, las cuales conoceremos en profundidad en el siguiente capítulo.

3.3 Variables

Máximo y mínimo

Las más obvias y que ya hemos nombrado son el máximo y el mínimo. Si no establecemos unos límites los números pueden desbordarse tanto por un lado como por el otro. Normalmente las librerías de donde utilizamos la función de Perlin Noise ya vienen con dichos límites establecidos en 0 y 1.

Frecuencia

La frecuencia corresponde al número de acontecimientos de determinado evento por unidad. La nube de puntos en algún momento empezará a repetirse, la frecuencia determina el tamaño o la escala de las características del ruido. En otras palabras, la frecuencia con la que se repite el patrón.

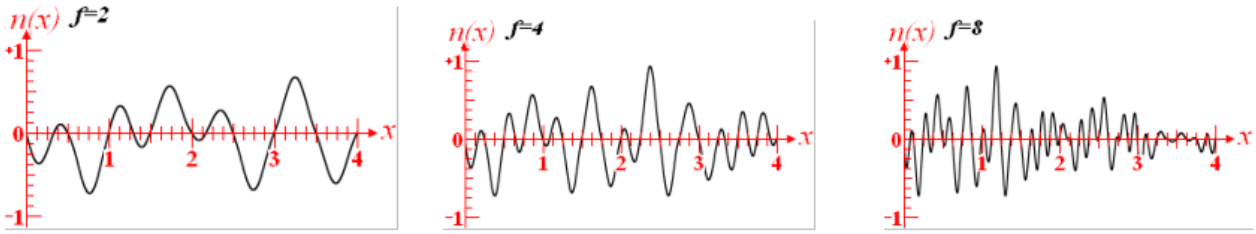


Ilustración 11: Funciones con frecuencia cada vez más alta

Jason B. (2005) Glossary for Perlin Noise terms. Recuperado de http://libnoise.sourceforge.net/glossary/index.html

También se compone como la inversa de la longitud de la onda. **F=1/L**.

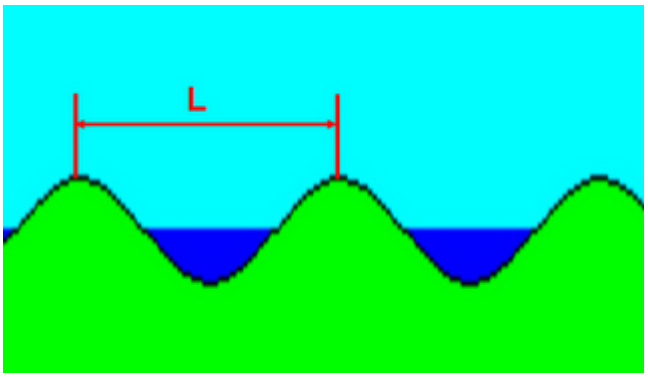


Ilustración 12: Longitud de onda

Hugo E. (2010) Perlin Noise: Funciones pseudoaleatorias. Recuperado de http://piziadas.com/2010/04/perlin-noise-funciones-pseudoaleatorias.html

Amplitud

Se define oficialmente como la perturbación máxima desde la posición no perturbada o neutral de la onda.

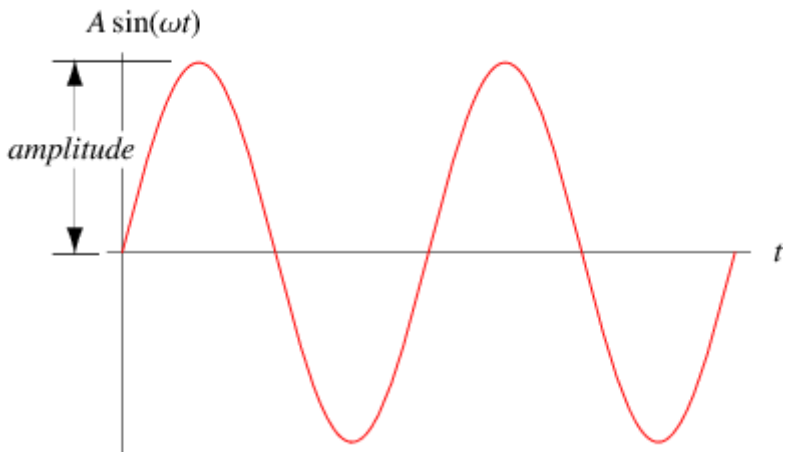


Ilustración 13: Amplitud de onda

Jason B. (2005) Glossary for Perlin Noise terms. Recuperado de http://libnoise.sourceforge.net/glossary/index.html

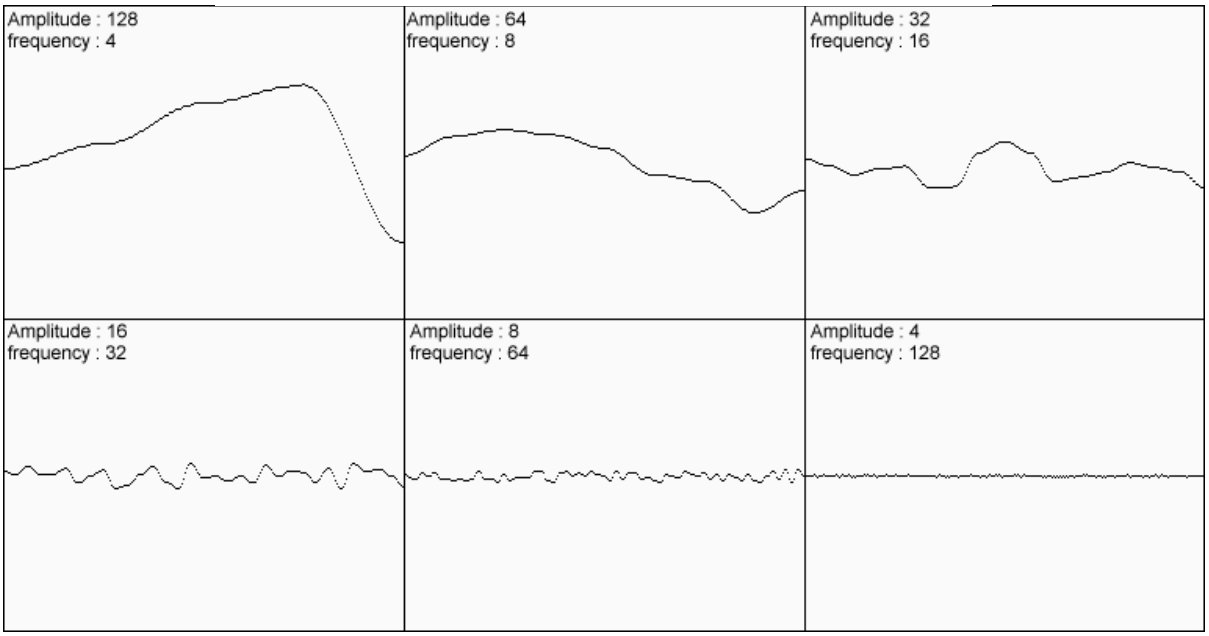


Ilustración 14: Ejemplo de generaciones variando frecuencia y amplitud.

Adrian B. (2014) Understanding Perlin Noise. Recuperado de https://adrianb.io/2014/08/09/perlinnoise.html

Octavas

Las octavas son uno de los conceptos más importantes del algoritmo, puesto que sobre éstas se apoyan todas las demás.

Describe el número de bucles que el código se ejecuta, generando una nube por cada una de esas veces. Obteniendo así un detalle cada vez más y más fino.

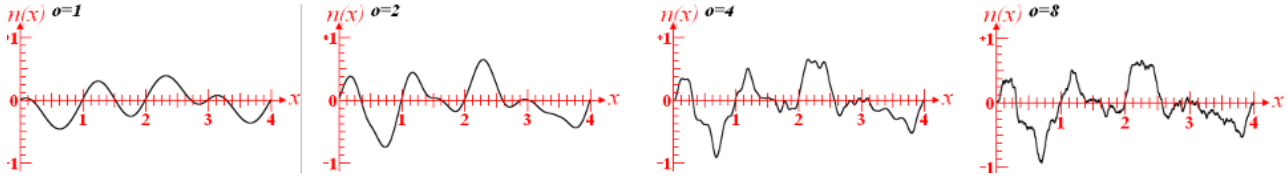


Ilustración 15: Función con cada vez más octavas

Jason B. (2005) Glossary for Perlin Noise terms. Recuperado de http://libnoise.sourceforge.net/glossary/index.html

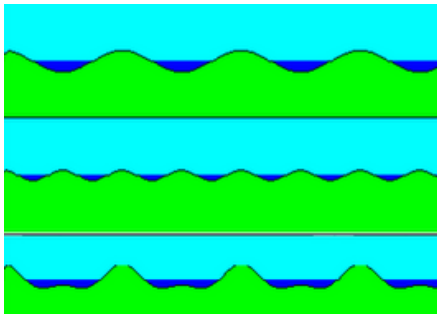


Ilustración 16: Dos funciones matemáticas sumadas

Hugo E. (2010) Perlin Noise: Funciones pseudoaleatorias. Recuperado de http://piziadas.com/2010/04/perlin-noise-funciones-pseudoaleatorias.html

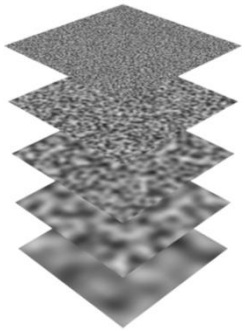


Ilustración 17: Octavas independientes

Value Noise for procedural patterns. Recuperado de https://www.scratchapixel.com/lessons/procedural-generation-virtual-worlds/procedural-patterns-noise-part-1/simple-pattern-examples

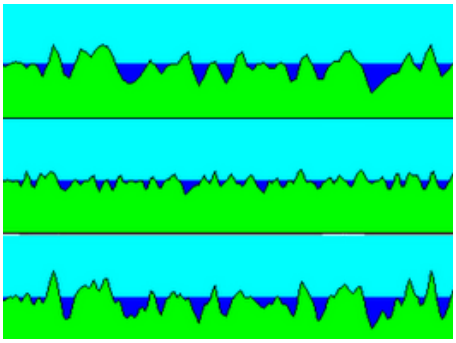


Ilustración 18: Dos funciones pseudoaleatorias sumadas

Hugo E. (2010) Perlin Noise: Funciones pseudoaleatorias. Recuperado de http://piziadas.com/2010/04/perlin-noise-funciones-pseudoaleatorias.html

Persistencia

Se define la persistencia como un multiplicador que determina cómo de rápido disminuye la amplitud por cada octava consecutiva. Es un valor entre 0 y 1, con lo cual cuanto más pequeña sea la persistencia, más rápido disminuirá la amplitud. Y en consecuencia menos afecta cada sucesiva octava al resultado final.

La amplitud de cada octava es el equivalente del producto de la amplitud de la anterior octava por el valor de la persistencia.

Previous Octave Amplitude * Persistence = Current Octave Amplitude.

Ilustración 17: Función con cada vez menos persistencia

Jason B. (2005) Glossary for Perlin Noise terms. Recuperado de http://libnoise.sourceforge.net/glossary/index.html

Lagunaridad

Es otro multiplicador que determina cómo de rápido aumenta la frecuencia por cada octava consecutiva. Es un valor mayor que 1, cuanto más cerca de 1 esté, más lentamente aumentará la frecuencia.

La frecuencia de cada octava es el equivalente del producto de la frecuencia de la anterior octava por el valor de la lagunaridad.

Previous Octave Frequency * Lacunarity = Current Octave Frequency.

Como se puede comprobar, tanto la persistencia como la brecha son valores con un comportamiento muy similar. Uno disminuye un valor y el otro aumenta otro, pero siguiendo la misma lógica.

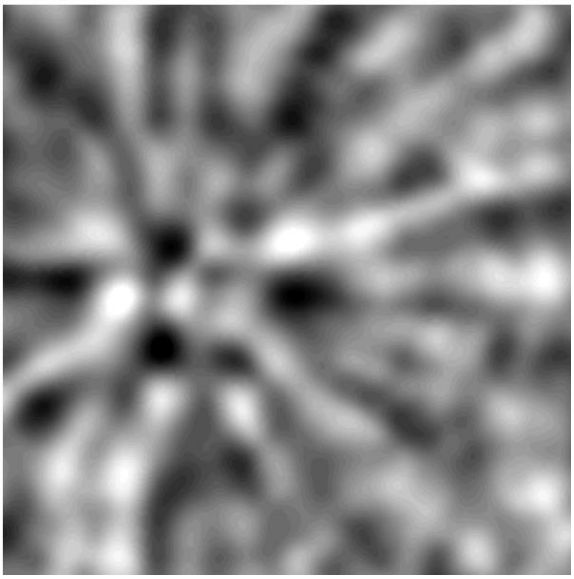


Ilustración 20: Nube con lagunaridad baja

Jens K. (2020) Understanding basic noise terms. Recuperado de http://www.campi3d.com/External/MariExtensionPack/userGuide5R4v1/Understandingsomebasicnoiseterms.html

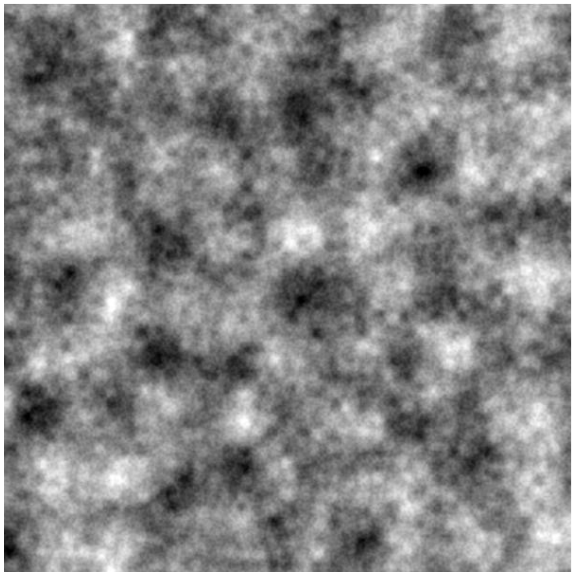


Ilustración 21: Nube con lagunaridad alta

Jens K. (2020) Understanding basic noise terms. Recuperado de http://www.campi3d.com/External/MariExtensionPack/userGuide5R4v1/Understandingsomebasicnoiseterms.html

Semilla

Se utiliza de la misma manera que en generaciones de números aleatorios estándar. Cada semilla tiene su propia generación, y siempre son diferentes entre sí.

Noise Scale (Zoom)

Se multiplica por la frecuencia, para hacer de ella un valor más grande, y que parezca que estamos viendo la imagen más de cerca. Por eso nos referimos a ella directamente como *zoom*.

Capítulo 4
Aplicación

En el capítulo de introducción hemos aprendido cuáles son los antecedentes del trabajo, cuál es el propósito del mismo, los detalles de cada una de las áreas que lo componen, entre otros.

Ahora en éste, entraremos en materia viendo cómo implementar cada uno de los conceptos teóricos ya vistos.

4.1 Herramientas

4.1.1 PCG

La aplicación fue desarrollada enteramente en el lenguaje de programación C#. Esta decisión tuvo una estrecha relación con el motor gráfico donde decidí trabajar: Unity.

Unity ofrece un apartado gráfico y tecnológico suficientemente potente y comprensible que hace una combinación perfecta para el resultado final tuviese un decente acabado gráfico y el trabajo no fuese completamente tedioso.

La aplicación funciona enteramente en modo edición. Al fin y al cabo, el resultado final es generar el propio terreno, no interactuar con él.

4.1.2 DSL

El DSL también ha sido desarrollado en C#. Ofrece una implementación sencilla y ortodoxa, la cual se relaciona directamente con la aplicación en sí, sin pasar por lectura de ficheros externos ni actividades que empeoren el rendimiento del programa.

4.2 Implementación del PCG

Lo que necesitamos para ver la primera implementación de una nube de ruido Perlin es un objeto 2D donde visualizarla (un plano), y un objeto controlador de la visualización.

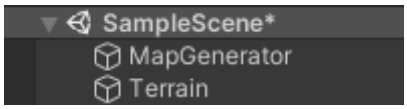


Ilustración 22: Objetos MapGenerator y Terrain. Elaboración propia.

4.2.1 Clase Noise

Para generar una nube de ruido Perlin, creé una clase que se encargara exclusivamente de ello. Y dentro creé la función GenerateNoiseMap, la cual, recibiendo todas las variables explicadas anteriormente, crearía un mapa de puntos de unas dimensiones dadas, que serían las variables *mapWidth* y *mapHeight.*

```
1 reference
public static float[,] GenerateNoiseMap(
    int mapWidth,
    int mapHeight,
    int seed,
    float scale,
    int octaves,
    float persistence,
    float lacunarity,
    Vector2 offset
) {
```

Ilustración 23: Cabecera GenerateNoiseMap. Elaboración propia.

```
float[,] noiseMap = new float[mapWidth, mapHeight];
System.Random rng = new System.Random(seed);
float amplitude = 1;
float frequency = 1;
```

Ilustración 24: Inicialización de las variables. Elaboración propia.

El grueso de la función está en los dos bucles que recorren cada pixel de la nube:

```
if (scale <= 0) scale = 000.1f;

float maxNoise = float.MinValue;
float minNoise = float.MaxValue;

for (int y = 0; y < mapHeight; y++) {
  for (int x = 0; x < mapWidth; x++) {
    float noiseHeight = 0;
    amplitude = 1;
    frequency = 1;

    for (int i = 0; i < octaves; i++) {
      float sampleX = (x - (mapWidth / 2) + octaveOffsets[i].x) / scale * frequency;
      float sampleY = (y - (mapHeight / 2) + octaveOffsets[i].y) / scale * frequency;
      // to not always have a positive value for the height, we multiply the perlin noise value calculated by 2 and substract 1
      float perlinValue = Mathf.PerlinNoise(sampleX, sampleY) * 2 - 1;
      noiseHeight += perlinValue * amplitude;
      amplitude *= persistence; // persistence is a value between 0 and 1 (amplitude decreases over time)
      frequency *= lacunarity; // lacunarity is a value over 1 (frequency increases over time)
    }
    maxNoise = (noiseHeight > maxNoise) ? noiseHeight : maxNoise;
    minNoise = (noiseHeight < minNoise) ? noiseHeight : minNoise;

    noiseMap[x, y] = noiseHeight;
  }
}
// Normalize value to be in between 0 and 1
for (int y = 0; y < mapHeight; y++)
  for (int x = 0; x < mapWidth; x++)
    noiseMap[x, y] = Mathf.InverseLerp(minNoise, maxNoise, noiseMap[x, y]);

return noiseMap;
```

Ilustración 25: Generación de la nube de puntos. Elaboración propia.

La clave de toda esta función está en crear los valores *sampleX* y *simpleY* para generar un valor del pixel acorde a sus vecinos, con la función ya integrada *Mathf.PerlinNoise(x,y)*.

Importante notar como la amplitud y la frecuencia varían en función de la persistencia y la lagunaridad, tal y como explicamos en el capítulo anterior.

Al final de la función se normaliza el valor de cada pixel para dejarlo entre 0 y 1.

Esta función devuelve un vector bidimensional de flotantes. Éste será el que represente la nube de puntos. Pero para hacer algo semejante, primero tenemos que hacer representable esta matriz, por ello hay que crear una textura.

4.2.2 Clase TextureGenerator

Creando una nueva clase *TextureGenerator*, y dentro de ella las funciones *TextureFromHeightMap,* que se encarga de aplicarle un color en base al valor del pixel, y *TextureFromColorMap,* que se encarga de crear la propia textura a partir de la matriz de color; conseguimos crear la textura aplicable a un objeto 3D.

```
1 reference
public static class TextureGenerator {

    1 reference
    public static Texture2D TextureFromColorMap(Color[] colorMap, int width, int height) {
        Texture2D texture = new Texture2D(width, height);
        texture.filterMode = FilterMode.Point;
        texture.wrapMode = TextureWrapMode.Clamp;
        texture.SetPixels(colorMap);
        texture.Apply();
        return texture;
    }

    1 reference
    public static Texture2D TextureFromHeightMap(float[,] heightMap) {
        int width = heightMap.GetLength(0);
        int height = heightMap.GetLength(1);

        Color[] colorMap = new Color[width * height];
        for (int y = 0; y < height; y++)
            for (int x = 0; x < width; x++)
                colorMap[y * width + x] = Color.Lerp(Color.black, Color.white, heightMap[x, y]);


        return TextureFromColorMap(colorMap, width, height);
    }
```

Ilustración 26: Clase TextureGenerator. Elaboración propia.

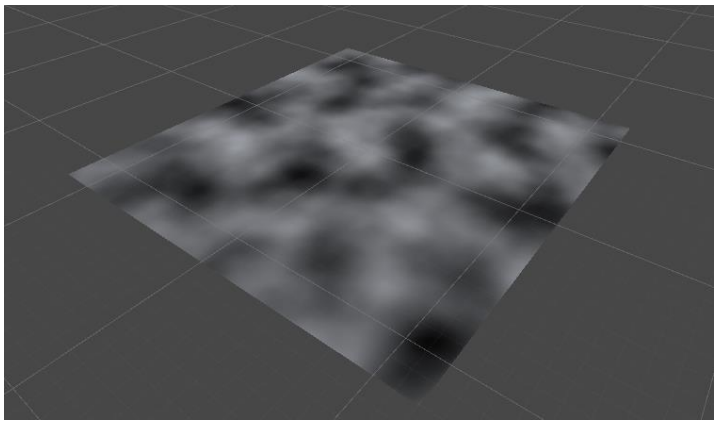


Ilustración 27: Plano con nube de ruido Perlin como textura. Elaboración propia.

Alcanzamos, al fin, el primer paso importante para la representación de nuestro terreno. Sin embargo, el terreno ha de ser en 3D. Hay que dar el siguiente paso y transformar el plano 2D en una malla 3D.

4.2.3 Mesh

Para esta parte tenemos que hacer varios cambios importantes. Primero, si no queremos tirar por la borda el trabajo hecho hasta ahora de ver representada la nube de puntos en un plano 2D, tenemos que dar la posibilidad de elegir qué tipo de representación queremos ver.

Para ello primero añadimos un objeto más a la escena (aparte de la luz y la cámara, que son complementarios).

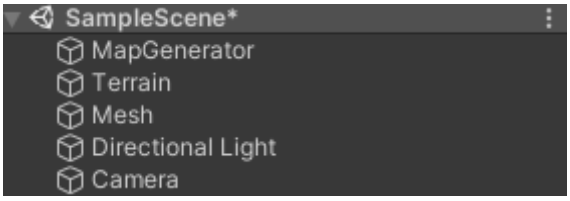


Ilustración 28: Objeto Mesh añadido a la escena. Elaboración propia.

Para dar la posibilidad a elegir, tenemos que modificar nuestro objeto *MapGenerator,* y crear uno nuevo, *MapDisplay.*

MapGenerator contendrá ahora una *flag* que determinará con qué tipo de visualización queremos proceder:

```
MapDisplay display = FindObjectOfType<MapDisplay>();
switch (drawMode) {
  default:
  case DrawMode.NoiseMap: display.DrawTexture(TextureGenerator.TextureFromHeightMap(noiseMap)); break;
  case DrawMode.Mesh: display.DrawMesh(MeshGenerator.GenerateTerrainMesh(noiseMap, terrainData.meshHeightMult, terrainData.meshHeightCurve, leve
}
```

Ilustración 29: Switch para tipo de visualización. Elaboración propia.

```
2 references
public class MapDisplay : MonoBehaviour {

  2 references
  public Renderer textureRender;
  1 reference
  public MeshFilter meshFilter;
  0 references
  public MeshRenderer meshRenderer;
  1 reference
  public void DrawTexture(Texture2D texture) {
    textureRender.sharedMaterial.mainTexture = texture;
    textureRender.transform.localScale = new Vector3(texture.width, 1, texture.height);
  }

  1 reference
  public void DrawMesh(MeshData meshData) {
    meshFilter.sharedMesh = meshData.CreateMesh();

  }

}
```

Ilustración 30: Clase MapDisplay. Elaboración propia.

MapDisplay llama al renderizador que toque en función del tipo de visualización elegida.

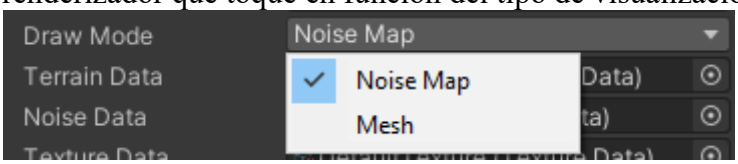


Ilustración 31: Elección de la visualización. Elaboración propia.

El objeto *Mesh* es un objeto vacío que contiene un componente *Mesh Filter*, *Mesh Renderer* y *Mesh Material*. Ahora, en vez de proporcionarle al objeto una textura 2D, tenemos que proporcionarle un material.

Clase MeshGenerator

La malla, aparte de recibir todas las variables pertenecientes a la nube de ruido Perlin, las cuales procesa la clase *Noise,* también recibe nuevas variables que solo puede utilizar la propia malla:

- *Height Multiplicator (float)*

Cuanto más grande sea este valor, más grandes serán los valores representados, y en consecuencia, más altura tendrán.

- *Height Curve (AnimationCurve)*

Es un objeto propio de Unity, que establece una función matemática a cada valor, creando así una imagen (valor en el eje de las y) para cada valor en el eje de las x.

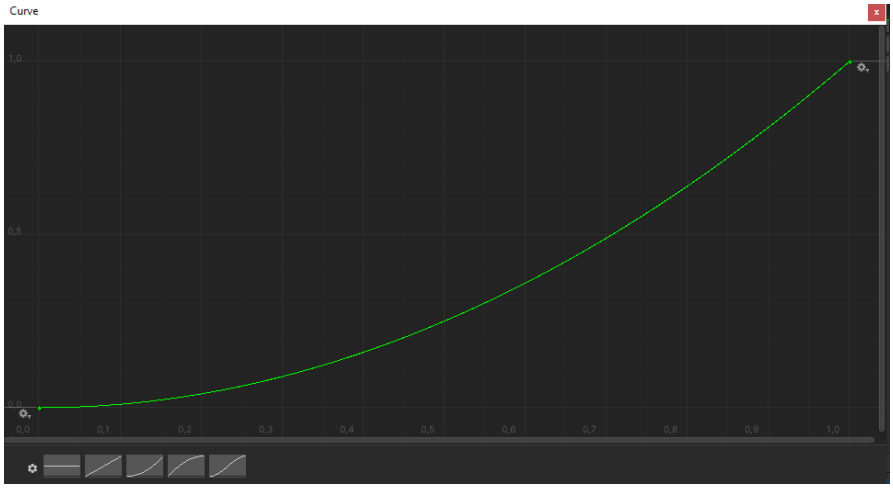


Ilustración 32: Animation Curve. Elaboración propia.

- *Level Of Detail (int)*

Indica cuántas veces se recorren los bucles generadores del material. Cuantas más veces se recorran, más detalle tendrá el objeto final, pero más costoso será generarlo. Este valor es muy importante para la optimización del juego. Lo ideal sería crear una mecánica de renderización variable en función de la distancia, cuanto más cerca esté el jugador, más detalles tendrá la textura, y viceversa.

La función *GenerateTerrainMesh,* que es la que veremos a continuación, creará en base a las variables explicadas hasta ahora, una textura UV, que aplicará al material de la malla.

En base a crear triángulos que cada uno tendrá su propia altura. Lo mejor es ver una comparación de dos imágenes, una con un nivel de detalle alto (más triángulos por unidad de espacio) y otra con un nivel de detalle bajo (menos triángulos). Funciona igual que la resolución de una imagen.

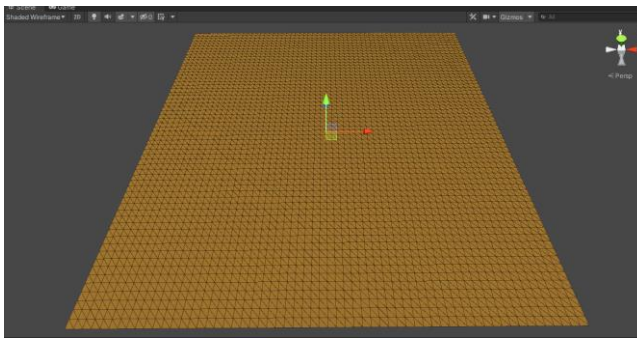


Ilustración 33: Malla con nivel de detalle alto. Elaboración propia.

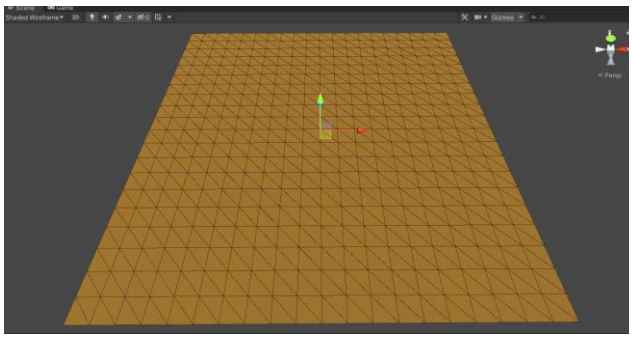


Ilustración 34: Malla con nivel de detalle bajo. Elaboración propia.

La función responsable de aplicar este material es la siguiente:

```
1 reference
public static class MeshGenerator {
    1 reference
    public static MeshData GenerateTerrainMesh(float[,] heightMap, float heightMult, AnimationCurve heightCurve, int levelOfDetail) {
        int width = heightMap.GetLength(0);
        int height = heightMap.GetLength(1);

        // These variables serve to center the image
        float topLeftX = (width - 1) / -2f;
        float topLeftZ = (height - 1) / 2f;

        int simpleIncrement = (levelOfDetail <= 0) ? 1 : levelOfDetail * 2;
        int verticesPerLine = (width - 1) / simpleIncrement + 1;

        MeshData meshData = new MeshData(verticesPerLine, verticesPerLine);
        int vertexIndex = 0;

        for (int y = 0; y < height; y += simpleIncrement) {
            for (int x = 0; x < width; x += simpleIncrement) {
                meshData.vertices[vertexIndex] = new Vector3(topLeftX + x, heightCurve.Evaluate(heightMap[x, y]) * heightMult, topLeftZ - y);
                meshData.uvs[vertexIndex] = new Vector2(x / (float)width, y / (float)height);
                if (x < width - 1 && y < height - 1) {
                    meshData.AddTriangle(vertexIndex, vertexIndex + verticesPerLine + 1, vertexIndex + verticesPerLine);
                    meshData.AddTriangle(vertexIndex + verticesPerLine + 1, vertexIndex, vertexIndex + 1);
                }
                vertexIndex++;
            }
        }
        return meshData;
    }
}
```

Ilustración 35: Clase MeshGenerator. Elaboración propia.

Resultado final:

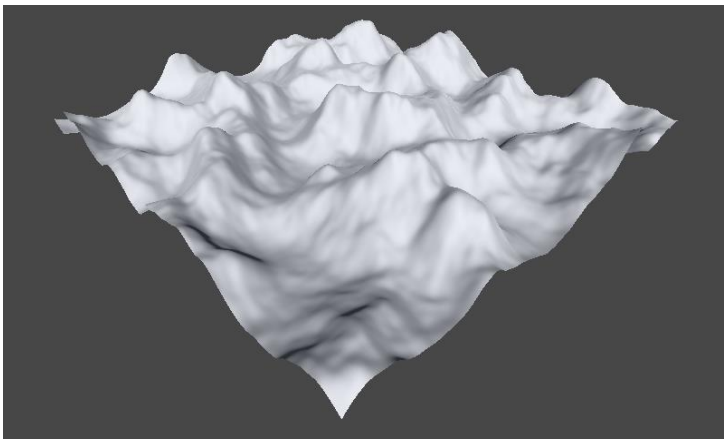


Ilustración 36: Malla en función de la nube de ruido Perlin. Elaboración propia.

Por fin tenemos un resultado acorde a lo que buscábamos. Ahora pasemos a darle un toque de color.

4.2.4 Shader y Material

El color se ajusta en función del valor de la altura del pixel. Para cada pixel entre 0 y 0.2, un color azul, entre 0.2 y 0.5 rojo, y así.
Para ello creamos un objeto *shader* y le aplicamos la técnica descrita.

```
const static int maxColorCount = 8;
const static float epsilon = 1E-4;


int baseColorCount;
float3 baseColors[maxColorCount];
float baseStartHeights[maxColorCount];
float baseBlends[maxColorCount];

float minHeight;
float maxHeight;

struct Input {
  float3 worldPos;
};

float inverseLerp(float min, float max, float value) {
  return saturate((value - min)/(max - min));
}

void surf (Input IN, inout SurfaceOutputStandard o) {
  float heightPercent = inverseLerp(minHeight,maxHeight, IN.worldPos.y);
  for (int i = 0; i < baseColorCount; i ++) {
    float drawStrength = inverseLerp(-baseBlends[i] / 2 - epsilon, baseBlends[i] / 2, heightPercent - baseStartHeights[i]);
    o.Albedo = o.Albedo * (1-drawStrength) + baseColors[i] * drawStrength;
  }
}
```

Ilustración 37: Shader para el terreno 3D. Elaboración propia.

Las variables que entran a colación aquí son:

- ❖ *Colors (float3)*

Vector de colores en un vector de 3 flotantes, uno para cada valor RGB. Ya procesados anteriormente desde una cadena hexadecimal.

- ❖ *Start Heights (float)*

Vector de flotantes que representan el valor de la propia altura del pixel.

- ❖ *Blends (float)*

Vector de flotantes que representan el gradiente entre el color anterior y el siguiente. Cuanto más bajo sea, más notorio será el paso de un color a otro. Cuanto más alto sea, más mezclados estarán ambos colores.

Surf es un método que se ejecuta cada vez que se actualiza cualquier valor que potencialmente cambie el material. En esta función se recorre el vector de colores y aplica a cada pixel por encima de la altura estipulada una fuerza a la hora de pintar el propio color, representada por la variable *drawStrength*. Dicha variable es un valor normalizado que va de 0 a 1. Si la altura calculada *heightPercent* da un valor negativo entonces significa que está por debajo de la altura que le corresponde. Si es positivo, se pinta el pixel del color que le corresponde con la fuerza calculada.

Rellenando dichos vectores con unos valores de ejemplo obtenemos este resultado:

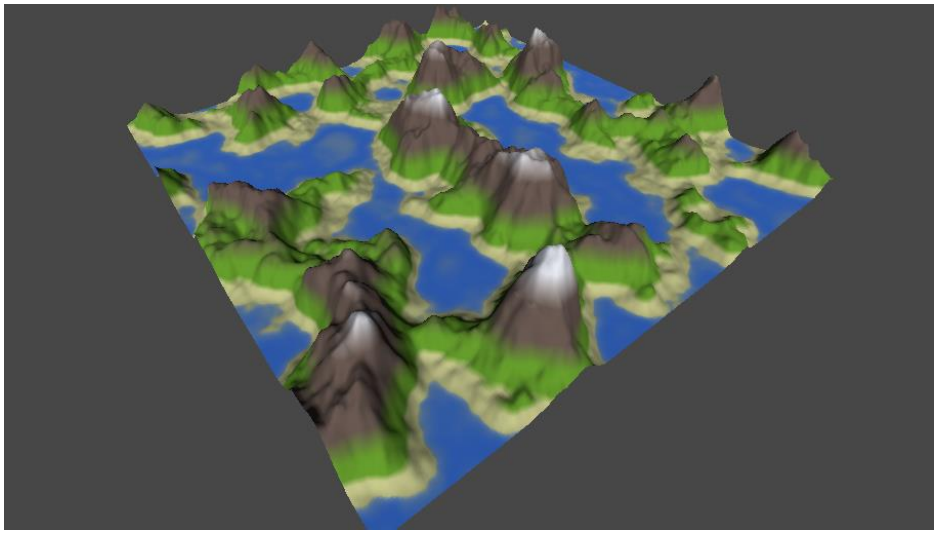


Ilustración 38: Malla con colores realistas. Elaboración propia.

4.2.5 Editor funcional

Una vez sabemos que funciona, podemos crear editores customizados para cambiar los valores rápidamente.

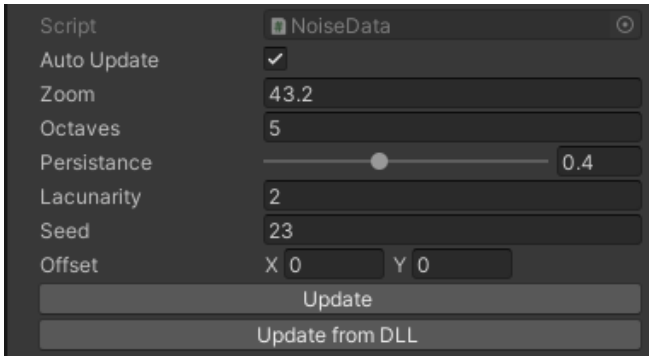


Ilustración 39: Editor para ajustar las variables del ruido Perlin. Elaboración propia.

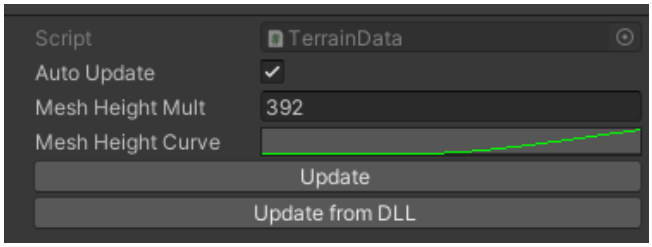


Ilustración 40: Editor para ajustar las variables de la malla. Elaboración propia.

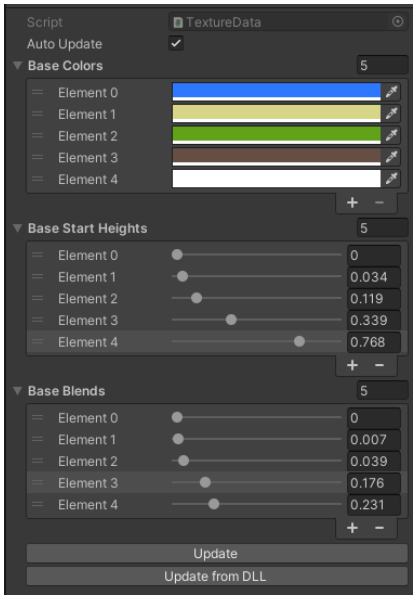


Ilustración 41: Editor para ajustar los colores de la malla. Elaboración propia.

Sin embargo, esto no es el propósito del trabajo. Ahora falta la segunda parte. Crear un lenguaje de dominio específico funcional que sirva para modificar todos estos valores a través de código.

4.3 Implementación del DSL

Tal y como dije al principio de este capítulo, el DSL estará implementado en C#. Ofrece herramientas suficientes como para que la implementación sea sencilla y su uso suficientemente potente.

Hay variables que se benefician notablemente de este tipo de implementación, poder elegir directamente el valor de las octavas, de la persistencia, la semilla… Sin embargo, otras como la curva de alturas son más difíciles de gestionar. Con lo cual he pasado por un proceso de selección, eligiendo qué tipo de variables merece la pena implementar y cuáles no.

Esta implementación está basada en un patrón de API o interfaz fluida, como hacen otros lenguajes tales como *LINQ*, *Scala* o *Raku*

4.3.1 Clase DSL

Cada una de las características que queremos añadir al objeto devuelven el mismo objeto con la característica añadida.

```
12 references
public class DLL {
    0 references
    static public DLL describedAs {
        get {
            return new DLL();
        }
    }

    0 references
    public DLL zoom(float z) {
        Zoom = z;
        return this;
    }

    0 references
    public DLL octaves(int o) {
        Octaves = o;
        return this;
    }
```

Ilustración 42: Clase DSL. Elaboración propia.

```
2 references
public class Script_DLL {

    2 references
    public static DLL Script() {

        DLL testDLL =
            DLL.describedAs
                .zoom(43.2f)
                .octaves(5)
                .persistance(0.4f)
                .lacunarity(2f)
                .seed(23)
                .addRegion(0f, "feae34", 0f)
                .addRegion(0.3f, "f77622", 0.181f)
                .addRegion(0.5f, "ce4121", 0.162f)
                .addRegion(0.7f, "88352b", 0.231f);

        return testDLL;
    }
}
```

Ilustración 43: Ejemplo de implementación. Elaboración propia.

Al hacer que cada función devuelva en sí misma el propio objeto, hace que la implementación sea fácil tanto de leer como de escribir.

La única excepción es la función *addRegion*, que, apoyándose en la clase *Region*, añade a los vectores del *shader* los colores y sus valores necesarios

```
4 references
public DLL addRegion(float height, string color, float blend) {
  Regions.Add(new Region(height, color, blend));
  return this;
}
```

n.cs ×

Scripts > DLL > Region.cs > Region

```
4 references
public class Region {
  2 references
  public float height;
  2 references
  public Color color;
  2 references
  public float blend;

  1 reference
  public Region(float height, string color, float blend) {
    this.height = height;
    this.color = GetColorFromString(color);
    this.blend = blend;

  }

  1 reference
  private Color GetColorFromString(string hex) {
    float red = HexToFloatNormalized(hex.Substring(0, 2));
    float green = HexToFloatNormalized(hex.Substring(2, 2));
    float blue = HexToFloatNormalized(hex.Substring(4, 2));
    return new Color(red, green, blue);
```

Ilustración 44: Clase Region. Elaboración propia.

Ejemplos

Probamos diversas combinaciones:

```
DLL dll_earth =
  DLL.describedAs
    .zoom(51.3f)
    .octaves(4)
    .persistance(0.31f)
    .lacunarity(1.65f)
    .seed(76)
    .addRegion(0f, "2F59A5", 0f)
    .addRegion(0.096f, "DBE08F", 0.018f)
    .addRegion(0.159f, "659E2A", 0.022f)
    .addRegion(0.3f, "685750", 0.056f)
    .addRegion(0.7f, "E6E9F0", 0.316f);
```

Ilustración 45: DLL para representar la Tierra. Elaboración propia.

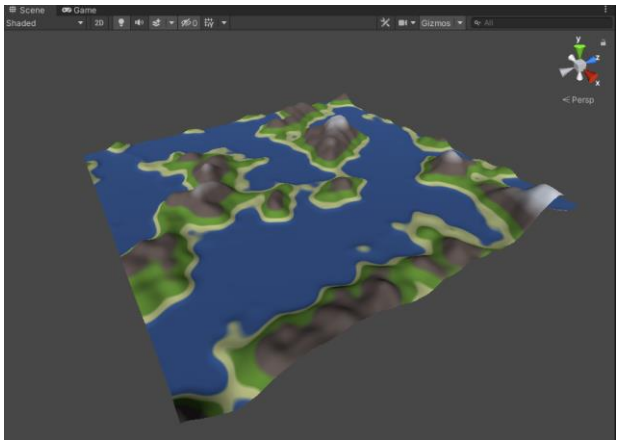


Ilustración 46: Terreno de la Tierra. Elaboración propia.

```
DLL dll_moon =
  DLL.describedAs
    .zoom(60.63f)
    .octaves(3)
    .persistance(0.555f)
    .lacunarity(1.15f)
    .seed(29)
    .addRegion(0f, "3D3737", 0f)
    .addRegion(0.034f, "706A6C", 1f)
    .addRegion(0.3f, "848188", 1f)
    .addRegion(0.5f, "B8B2B2", 1f)
    .addRegion(0.7f, "C3BDBD", 1f);
```

Ilustración 47: DLL para representar la Luna. Elaboración propia.

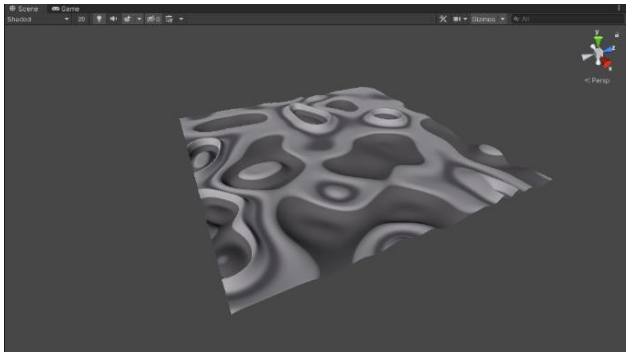


Ilustración 48: Terreno de la Luna. Elaboración propia.

```
DLL dll_mars =
  DLL.describedAs
    .zoom(43.2f)
    .octaves(5)
    .persistance(0.4f)
    .lacunarity(2f)
    .seed(23)
    .addRegion(0f, "feae34", 0f)
    .addRegion(0.3f, "f77622", 0.181f)
    .addRegion(0.5f, "ce4121", 0.162f)
    .addRegion(0.7f, "88352b", 0.231f);
```

Ilustración 49: DLL para representar Marte. Elaboración propia.

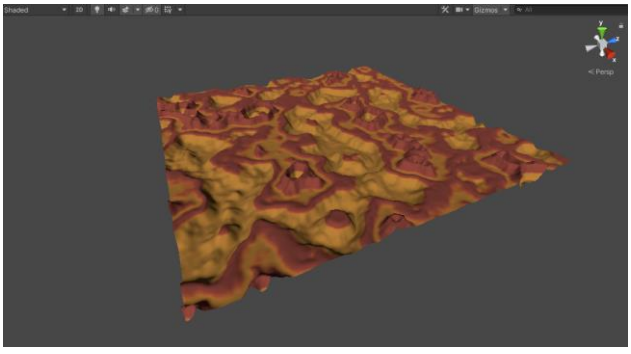


Ilustración 50: Terreno de Marte. Elaboración propia.

4.3.2 Tests

Añadí también tests, para comprobar que el DSL funciona correctamente.

```
public DLL dll =
  DLL.describedAs
    .zoom(43.2f)
    .octaves(5)
    .persistance(0.4f)
    .lacunarity(2f)
    .addRegion(0.1f, "003D79", 0.002f)
    .addRegion(0.3f, "0054AA", 0.002f)
    .addRegion(0.4f, "0061C2", 0.002f);
```

Ilustración 51: Objeto DLL a testear. Elaboración propia.

```
public void regions() {
  List<float> RegionsTest = new List<float>() {
    0.1f, 0.3f, 0.4f
  };

  for (int i = 0; i < dll.Regions.Count; i++) {
    Assert.AreEqual(dll.Regions[i].height, RegionsTest[i]);
  }
}

[Test]
0 references
public void zoom() {
  Assert.AreEqual(dll.Zoom, 43.2f);
}
[Test]
0 references
public void octaves() {
  Assert.AreEqual(dll.Octaves, 5);
}
[Test]
0 references
public void persistance() {
  Assert.AreEqual(dll.Persistance, 0.4f);
}
[Test]
0 references
public void lacunarity() {
  Assert.AreEqual(dll.Lacunarity, 2f);
}
[Test]
0 references
public void seed() {
  Assert.IsTrue(isInBetween(-100000, 100000, dll.Seed));
}
```

Ilustración 52: Tests individuales. Elaboración propia.

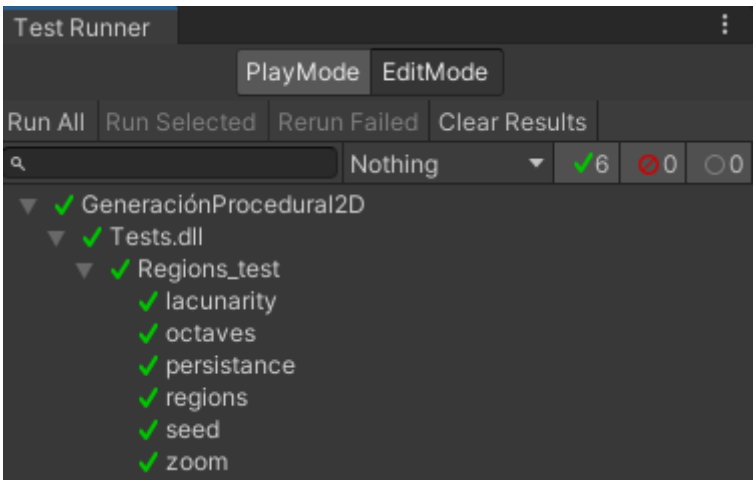


Ilustración 53: Tests pasando correctamente en Unity. Elaboración propia.

Capítulo 5
Problemática

En primer lugar, el tiempo. Tal y como está planteado el plan de estudios de la universidad, el Trabajo de Fin de Grado y las Prácticas Externas coinciden en tiempo y forma, los estudiantes tienen que dedicarse a un trabajo de casi jornada completa, además de ir dedicándole poco a poco el tiempo necesario al proyecto, para luego no verse desbordados en las últimas fases del mismo.

En segundo lugar, familiarizarme con el motor gráfico. Unity, a pesar de ser relativamente amigable con el usuario inexperto, sigue siendo una herramienta compleja. Y la curva de dificultad, sobre todo al principio, es muy empinada.

Por último, el estudio de los lenguajes de dominio específico a través de interfaces fluidas. En la asignatura de Lenguajes y Paradigmas de la Programación se aprendió la teoría de los DSL y se aplicó en diversos ejemplos sencillos en el lenguaje de programación Ruby. Con todo, el desarrollo en C# se distingue mucho del de Ruby, así que conllevó un estudio de la teoría desde cero, esta vez para entender cómo se aplicaba al esquema de programación de C#.

Capítulo 6
Conclusiones

En líneas generales, estoy muy orgulloso con el trabajo que realizado. Ha costado el suficiente esfuerzo para sentir que he tenido que dar todo de mí para poder sacarlo adelante.

Con respecto al proyecto en sí, si bien una herramienta de generación de terrenos 3D es fácilmente utilizable con un editor normal y corriente, creo, sinceramente, que un lenguaje de dominio específico es una buena manera de relacionarse con ella. Otorga claridad al novato y suficiente complejidad al entendido. He cumplido con la principal agenda que me propuse, y es que fuese todo lo polivalente que pudiese.

¿Qué añadiría? Me gustaría incluir nuevas características propias de terrenos realistas, como, por ejemplo, la posibilidad de incluir ríos, y que se puedan especificar sus variables: desde qué punto a qué punto parte, la anchura, cómo de profundo sea, etc. Texturas en vez de colores añadiría un ambiente mucho más creíble. Desarrollar una demo para navegador, sencilla de utilizar, para que se pueda probar sin necesidad de descargar e instalar la aplicación completa. También pulir y refactorizar parte del código para hacerlo más legible y presentable al público.

En la asignatura de Interfaces Inteligentes tuve la suerte de poder realizar un trabajo sobre generación procedural y ahí es donde descubrí lo mucho que me interesaba. Por eso mi elección de este proyecto. En el futuro me gustaría ahondar más en este campo, veo que realmente he rascado solo la punta del iceberg. Hay muchas más cosas que estudiar, por hacer, y donde poder aprender muchísimo al respecto. Veo el futuro con muy buenos ojos.

Capítulo 7
Conclusions

Overall, I am very proud of the work I have done. It took enough effort to feel that I had to give my all to get it done.

Regarding the project itself, while a 3D terrain generation tool is easily usable with an ordinary editor, I honestly believe that a domain-specific language is a good way to interact with it. It gives clarity to the beginner and sufficient complexity to the connoisseur. I have accomplished the main goal I set for myself, and that is to make it as versatile as I can. I would like to continue developing new applications to include features of real terrains to make it more realistic, for example, the possibility of including rivers, or textures instead of colors. I would also like to polish and refactor some of the code to make it more readable and presentable to the public.

In the Intelligent Interfaces subject I was lucky enough to be able to do a paper on procedural generation and that is where I discovered how much I was interested in it. That's why I chose this project. In the future I would like to delve deeper into this field, I see that I have really only scratched the tip of the iceberg. There are many more things to study, to do, and where I can learn a lot about it. I see the future with very good eyes.

Capítulo 8
Presupuesto

Para el desarrollo de la aplicación lo único que se ha necesitado es un editor de texto y el motor gráfico Unity. Unity ha puesto a disposición de los usuarios una serie de planes con más mejoras y ventajas cuanto más caro sea. La implementación de la aplicación no necesitó nada más que el programa con el plan base, ni la compra de ningún *asset* de terceros.

Han sido unos 4 meses de trabajo, invirtiendo en total aproximadamente 300 horas. Si para un programador junior el salario por hora fuese de 11€, en total la mano de obra de un solo programador para la implementación de la aplicación es de 3300€.

Elemento	Tipo de recurso	Unidades	Precio por unidad	Costo
Plan de Unity	Informático	1	0	0
Assets de terceros	Informático	N/A	N/A	0
Mano de obra	Personal	300	11	3300
			Total	3300€

Tabla 2: Presupuesto

Bibliografía

Wikipedia. (2022). Procedural Generation. Recuperado de https://en.wikipedia.org/wiki/Procedural_generation

Yvan Scher. (2017). Playing with Perlin Noise: Generating Realistic Archipelagos. Recuperado de https://medium.com/@yvanscher/playing-with-perlin-noise-generating-realistic-archipelagos-b59f004d8401

Jens Kafitz. (2020). Understanding some noise terms. Recuperado de http://www.campi3d.com/External/MariExtensionPack/userGuide5R4v1/Understandingsomebasicnoiseterms.html

Hugo Elías. (2010). Perlin Noise: Funciones pseudoaleatorias. Recuperado de http://piziadas.com/2010/04/perlin-noise-funciones-pseudoaleatorias.html

Jason Bevins. (2005). Libnoise Glossary. Recuperado de http://libnoise.sourceforge.net/glossary/

Unity. (2022). Unity User Manual Guide. Recuperado de https://docs.unity3d.com/Manual/index.html

Study.com. (Sin fecha). What is a Domain Specific Language? – Tools & Examples. Recuperado de https://study.com/academy/lesson/what-is-a-domain-specific-language-tools-examples.html

Microsoft. (2022). Language integrated query (LINQ) (C#). Recuperado de https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/

Neal Ford. (2019). Building domain specific languages. Recuperado de https://www.codemag.com/article/0902041/Building-Domain-Specific-Languages-in-C

Sergiy Nikolayev. (2022). NRules. Recuperado de https://nrules.net/