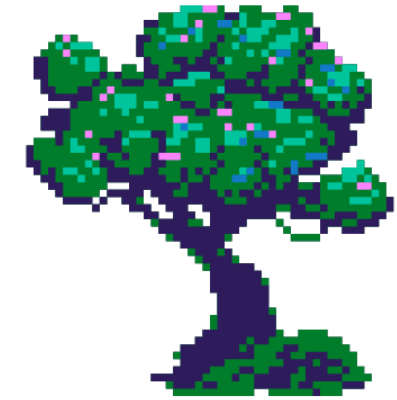
Procesadores de Lenguajes



Lenguaje para la creación de entornos 3D en Unity

Integrantes:

- Francisco Javier Luna Ortiz.
- Alejandro del Hoyo Abad.
- Antonio Gómez Jimeno.
- Felipe Sánchez Mur.

Contenido

1. Presentación del problema.

1.1. Concepto.

El objetivo de GeoCraft consiste en simplificar y automatizar el proceso de creación de terrenos 3D en Unity. Este lenguaje está diseñado para complementar el trabajo de los diseñadores, facilitando la generación inicial de entornos y modelos en las primeras etapas del proyecto.

Para simplificar y facilitar la creación de entornos 3D, se utilizará lo que denominaremos a partir de ahora como descriptores.

1.2. Fundamentos.

Este lenguaje está basado en el modelo de trabajo de Unity, la programación orientada a objetos y en la generación procedural de terrenos (como en el videojuego Minecraft).

1.2.1. Unity.

Unity es un motor de desarrollo de videojuegos que permite crear aplicaciones interactivas en 2D y 3D. Desarrollado por *Unity Technologies*, destaca por su capacidad multiplataforma, lo que facilita la exportación de proyectos a dispositivos como PC, consolas, móviles y entornos de realidad virtual y aumentada.

Utiliza el lenguaje de programación C# y ofrece herramientas avanzadas para gráficos, física, animación y más.

Esta herramienta se enfoca en el desarrollo de videojuegos, pero también se usa en campos como la simulación, arquitectura y educación, siendo apoyado por una amplia comunidad y abundante documentación.

1.2.2. Programación orientada a objetos.

La programación orientada a objetos (POO) es un enfoque de desarrollo de software que utiliza "objetos" como elementos fundamentales. Estos objetos almacenan datos en sus atributos y tienen funciones llamadas métodos, que permiten manipular esos datos.

Este paradigma ha servido de inspiración para la creación del lenguaje. Como ya se ha mencionado, para simplificar y facilitar la creación de entornos 3D, se

utilizará lo que denominaremos a partir de ahora como descriptores. Los descriptores están basados en los objetos que podemos crear en cualquier lenguaje de programación orientado a objetos.

1.2.3. Generación procedural de terrenos.

Técnica utilizada en gráficos por computador y videojuegos para crear escenas de manera automática, sin necesidad de diseñarlos manualmente, en base a unas directrices.

Gracias a esta técnica somos capaces de generar contenido infinito de manera coherente utilizando una semilla. Nuestro proyecto se basará en los mapas de ruido para la generación procedural de los terrenos.

Ejemplos de videojuegos que emplean la generación procedural:

- *Minecraft*.
- *No Man's Sky*.
- *Diablo III*.
- *Borderlands*.
- *Civilization*.

Ilustración 1: Ejemplo de mundo en el videojuego Minecraft.

1.2.4. Mapas de ruido y generación de semillas.

Los mapas de ruido *Perlin Noise* son funciones matemáticas que permiten generar variaciones en el terreno, como montañas, colinas o valles, de manera que el terreno parezca natural y orgánico.

En combinación con una semilla (número inicial que define el patrón de variabilidad), es posible generar grandes extensiones de terreno que mantienen coherencia entre sí, pero con variaciones suficientes para que no sean repetitivas.

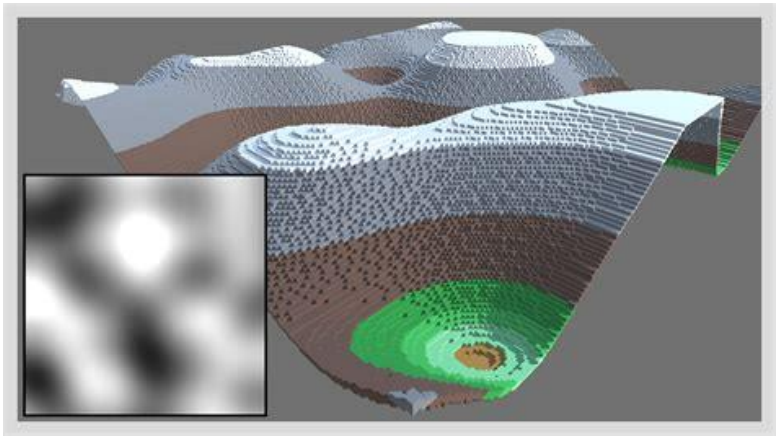


Ilustración 2: Perlin noise y modelo 3D que genera.

1.2.5. Chunks.

Se refiere a una porción o bloque de datos que se maneja como una sola unidad. En el contexto de la generación de terrenos, un chunk es una sección del mundo virtual que se carga y se procesa de manera independiente.

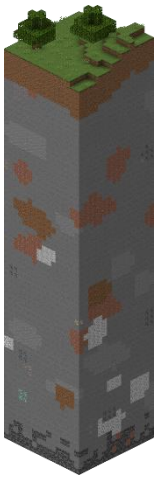


Ilustración 3: Chunk en Minecraft.

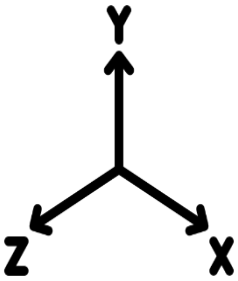


Ilustración 4: Ejes cardinales

1.3. ¿Cómo trabajaremos?

El objetivo del proyecto es poder generar terrenos 3D en Unity de manera sencilla usando nuestro lenguaje. Nos basaremos en el concepto de chunks, que son secciones o "trozos" del terreno que se pueden manejar de manera individual.

El proyecto estará dividido en escenas. Cada una representará un modelo 3D diferente y estará compuesta por múltiples chunks, lo que permite una organización modular.

La idea es que cada chunk puede ser trabajado de forma independiente, lo que significa que se pueden modificar sus características, como el tamaño, la forma, la textura y otros atributos sin afectar a los demás. Esta modularidad permite a los desarrolladores personalizar y ajustar cada parte del terreno según las necesidades específicas del proyecto.

El lenguaje también permitirá añadir modelos 3D, tales como árboles y rocas, dentro de cada chunk.

1.4. Justificación del proyecto.

Este proyecto concuerda con los contenidos de la asignatura de Procesadores de Lenguajes debido a que se centra en el diseño y la implementación de un lenguaje específico de dominio (DSL) para la creación de terrenos 3D en Unity y de su respectivo procesador de lenguaje. El objetivo de todo esto consiste en la resolución de un problema, en nuestro caso la generación de terrenos 3D en Unity.

El desarrollo de este proyecto seguirá un enfoque incremental. Cada etapa del proceso estará alineada con los temas estudiados en la asignatura de Procesadores de Lenguajes.

Etapas del desarrollo:

- Analizador léxico.
- Analizador sintáctico.
- Analizador semántico.
- Generador de código.

2. EBNF del lenguaje y diagramas de Conway.

2.1. Tabla de tokens

A continuación, se presentan los tokens del lenguaje. Un token es la unidad mínima de significado reconocidas por el analizador léxico y cada uno de ellos, representa una secuencia de caracteres que corresponde a un patrón especifico.

TOKEN	EJEMPLO	CONSTRUCCIÓN
DEFINE	‘DEFINE’	‘DEFINE’
SETUP	‘SETUP’	‘SETUP’
WORLD	‘WORLD’	‘WORLD’
ADD	‘ADD’	‘ADD’
APPEND	‘APPEND’	‘APPEND’
SCENE	‘SCENE’	‘SCENE’
FOR	‘FOR’	‘FOR’
FROM	‘FROM’	‘FROM’
TO	‘TO’	‘TO’
IN	‘IN’	‘IN’
DESCRIPTOR	‘GAMEOBJECT’, ‘CHUNK’	‘GAMEOBJECT’ | ‘CHUNK’
LIST	‘LIST’	‘LIST’
INT	‘INT’	‘INT’
FLOAT	‘FLOAT’	‘FLOAT’
STRING	‘STRING’	‘STRING’
LPAREN	‘(‘	‘(‘
RPAREN	‘)’	‘)’
LBRACE	‘{‘	‘{‘
RBRACE	‘}’	‘}’
LSQUARE	‘[‘	‘[‘
RSQUARE	‘]’	‘]’
COMMA	‘,’	‘,’
LT	‘<’	‘<’
GT	‘>’	‘>’
ASSIGN	‘=’	‘=’
OP_ARIT	‘+’,’-‘,’*’,’/’	‘+’ | ’-‘ | ’*’ | ’/’
ID	‘variable_1’	[a-zA-Z] [a-zA-Z0-9_]*
INT_LITERAL	‘123’	[0-9]+
FLOAT_LITERAL	‘123.45’	[0-9]+ '.' [0-9]+
STRING_LITERAL	‘”cadena”’	'"' ~["]* '"'

2.2. Producciones

2.2.1. Program

Esta producción define la estructura básica del programa, que consiste en un bloque centrado en la definición de objetos, variables y chunks globales (setup) y un bloque que contiene la estructura organizativa principal que contiene todas las escenas del entorno. Además, la producción finaliza con el token *EOF*, que indica el fin del archivo.

La sintaxis de esta producción es la siguiente:

```
program ::= define_setup define_world EOF ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 3: Diagrama de Conway de la produccion "program"

2.2.2. Define_setup

La producción *define_setup* especifica la definición de objetos, variables y chunks globales que podrán ser utilizados en cualquier escena del proyecto.

La sintaxis de esta producción es la siguiente:

```
define_setup ::= DEFINE SETUP LPAREN RPAREN LBRACE setup_statement* RBRACE ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

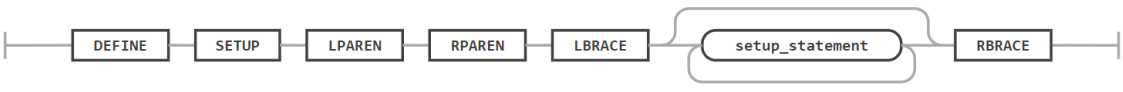


Ilustración 4: Diagrama de Conway de la produccion "define_setup"

2.2.3. Define_world

La producción *define_world especifica* la estructura organizativa principal que contiene todas las escena del entorno.

La sintaxis de esta producción es la siguiente:

```
define_world ::= DEFINE WORLD LPAREN expression RPAREN LBRACE define_scene+ RBRACE
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 5: Diagrama de Conway de la produccion "define_setup"

2.2.4. Define_scene

La producción *define_scene* permite la definición de una escena específica dentro del mundo. Cada escena es independiente y puede estar compuesta por diferentes objetos y chunks.

La sintaxis de esta producción es la siguiente:

```
define_scene ::= DEFINE SCENE LPAREN parameters RPAREN LBRACE statement* RBRACE ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 6: Diagrama de Conway de la produccion "define_setup"

2.2.5. Parameters

En la producción *parameters* se definen los parámetros que pueden ser utilizados en diferentes definiciones. Cada uno de estos parámetros están separados por comas.

La sintaxis de esta producción es la siguiente:

```
parameters ::= expression (COMMA expression)* ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

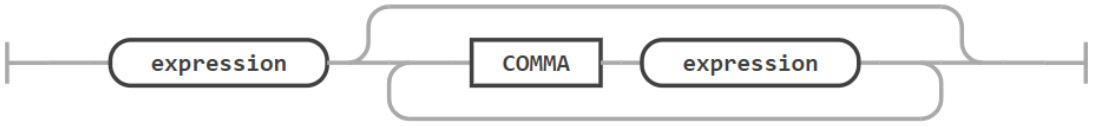


Ilustración 7: Diagrama de Conway de la produccion "define_setup"

2.2.6. Expression

La producción *expresión* define la estructura general de una expresión. Para evitar la recursividad, se utiliza el componente auxiliar *expresión_aux*.

La sintaxis de esta producción es la siguiente:

```
expression ::= expression_aux (OP_ARIT expression_aux)* ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

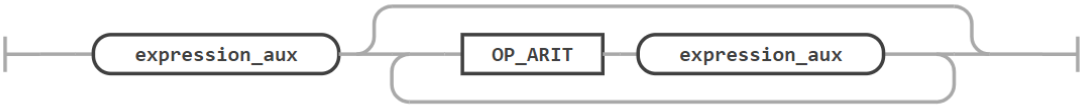


Ilustración 8: Diagrama de Conway de la produccion "define_setup"

2.2.7. Expression_aux

La producción *expresión_aux* enumera las posibles formas que puede adoptar una expresión.

La sintaxis de esta producción es la siguiente:

```
expression_aux ::= STRING_LITERAL
        | INT_LITERAL
        | FLOAT_LITERAL
        | ID
        | LPAREN expression RPAREN
        | LSQUARE expression (COMMA expression)* RSQUARE
        | descriptor_constructor ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

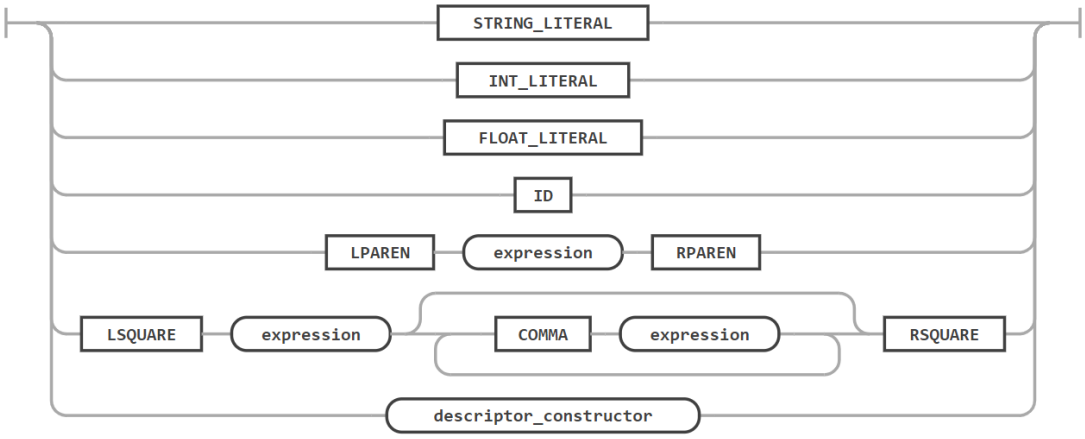


Ilustración 9: Diagrama de Conway de la produccion "define_setup"

2.2.8. Descriptor_constructor

La producción *descriptor_constructor* define cómo se puede crear una instancia de descriptor a través de un constructor.

La sintaxis de esta producción es la siguiente:

```
descriptor_constructor ::= DESCRIPTOR LPAREN parameters RPAREN ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 10: Diagrama de Conway de la produccion "define_setup"

2.2.9. Setup_statement

La producción *setup_statement* agrupa las diferentes instrucciones que pueden realizarse dentro del bloque *setup*. Incluye asignaciones, declaraciones, instrucciones del tipo *append* y bucles for propios de este bloque. La principal diferencia entre los *statements* del bloque *setup* y el bloque *world* es que no se pueden utilizar instrucciones del tipo *add*, ya que, estas indican los chunks que se quieren crear en una escena.

La sintaxis de esta producción es la siguiente:

```
setup_statement ::= assignment
        | declaration
        | append_statement
        | for_loop_setup ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

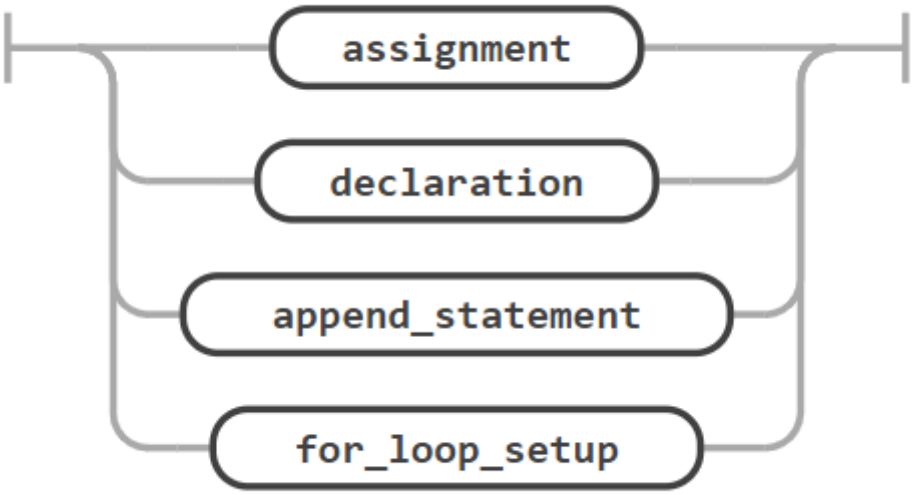


Ilustración 11: Diagrama de Conway de la produccion "define_setup"

2.2.10. Statement

La producción *statement*, como se ha explicado anteriormente, incluye los bucles específicos del bloque *world* y las instrucciones del tipo *add*.

La sintaxis de esta producción es la siguiente:

```
statement ::= setup_statement
      | add_statement
      | for_loop_world ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

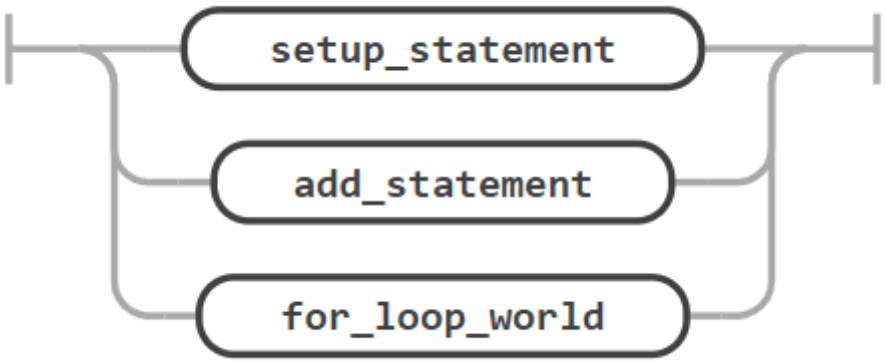


Ilustración 12: Diagrama de Conway de la produccion "define_setup"

2.2.11. Assignment

La producción *assignment* describe cómo se puede asignar un valor a una variable.

La sintaxis de esta producción es la siguiente:

```
assignment ::= data_type ID ASSIGN expression ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 13: Diagrama de Conway de la produccion "define_setup"

2.2.12. Declaration

La producción *declaration* permite declarar variables sin inicializarlas con un valor concreto.

La sintaxis de esta producción es la siguiente:

```
declaration ::= data_type ID ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 14: Diagrama de Conway de la produccion "define_setup"

2.2.13. Data_type

La producción *data_type* especifica los posibles tipos de datos que pueden ser utilizados en el lenguaje.

La sintaxis de esta producción es la siguiente:

```
data_type ::= INT
        | FLOAT
        | STRING
        | DESCRIPTOR
        | list_type ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

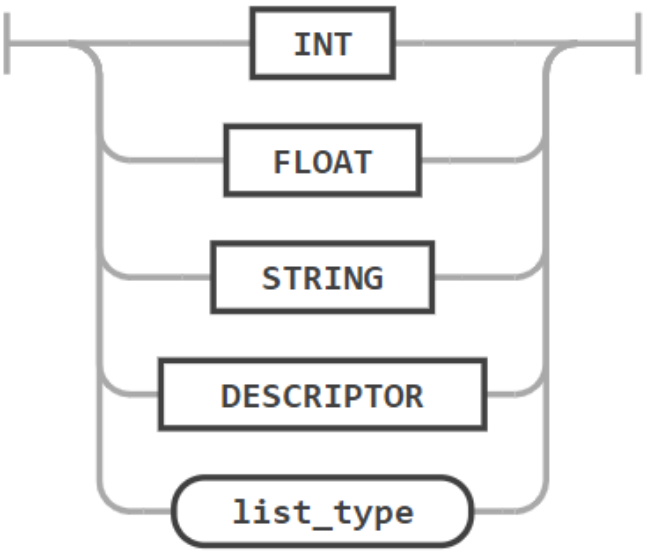


Ilustración 15: Diagrama de Conway de la produccion "define_setup"

2.2.14. List_type

La producción *list_type* define la estructura del tipo *lista*, que puede contener instancias de un descriptor en concreto.

La sintaxis de esta producción es la siguiente:

```
list_type ::= LIST LT DESCRIPTOR GT ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 16: Diagrama de Conway de la produccion "define_setup"

2.2.15. Append_statement

La producción *append_statement* detalla cómo se pueden agregar elementos a una lista.

La sintaxis de esta producción es la siguiente:

```
append_statement ::= APPEND ID ID
                   | APPEND ID descriptor_constructor ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

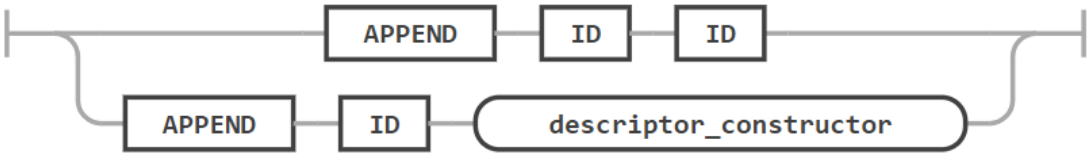


Ilustración 17: Diagrama de Conway de la produccion "define_setup"

2.2.16. For_loop_setup

La producción *for_loop_setup*, como se ha explicado antes, establece las estructuras que puede tener un bucle en la sección *setup*.

La sintaxis de esta producción es la siguiente:

```
for_loop_setup ::= FOR ID FROM expression TO expression LBRACE setup_statement* RBRACE
                 | FOR ID IN ID LBRACE setup_statement* RBRACE ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

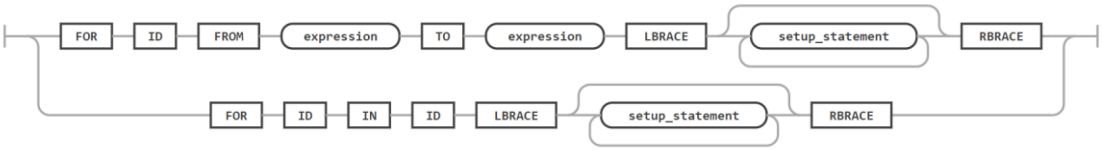


Ilustración 18: Diagrama de Conway de la produccion "define_setup"

2.2.17. For_loop_world

La producción *for_loop_world*, como se ha explicado antes, establece las estructuras que puede tener un bucle en la sección *world*.

La sintaxis de esta producción es la siguiente:

```
for_loop_world ::= FOR ID FROM expression TO expression LBRACE statement* RBRACE
                 | FOR ID IN ID LBRACE statement* RBRACE ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

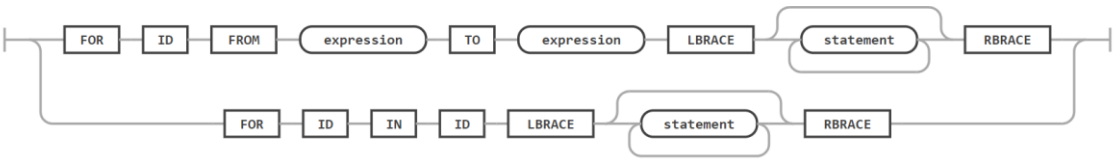


Ilustración 19: Diagrama de Conway de la produccion "define_setup"

2.2.18. Add_statement

La producción *add_statement* especifica una instrucción para agregar un elemento a una escena en concreta.

La sintaxis de esta producción es la siguiente:

```
add_statement ::= ADD ID ;
```

Para ilustrar mejor el funcionamiento de esta producción, se muestra el diagrama de *Conway*:

Ilustración 20: Diagrama de Conway de la produccion "define_setup"

3.Semántica.

3.1. Variables y tipos de datos.

Una variable se utiliza para almacenar y representar datos en el programa. Asocia un identificador (nombre de la variable) con un valor de un tipo de dato específico.

En el lenguaje existen distintos tipos de datos:

- STRING → para cadenas de texto.
- INT → para números enteros.
- FLOAT → para números decimales.

¿Cómo definimos una variable?

data_type ID = valor

Ejemplo:

```
INT entero = 2

FLOAT decimal = 3.5

STRING cadena = “Hola mundo”
```

Reglas para los identificadores.

Un identificador válido debe comenzar con una letra, ya sea mayúscula o minúscula, seguida opcionalmente de una secuencia de letras, dígitos o guiones bajos.

Cambiar de valor de una variable a lo largo del programa.

```
INT var_1 = 2

INT  var_2 = 4

var_1 = var_2

var_2 = 18
```

Al final de la ejecución de esta sección de código los valores de las variables serían:

var_1 = 4 var_2 = 18

Además, hay que tener en cuenta los descriptores y las listas de descriptores, los cuales se consideran tipo de dato, pero tienen una manera distinta de ser definidos. Más adelante los desarrollaremos.

3.2. Comentarios.

GeoCraft permite escribir comentarios. Toda línea que comience por "//" no se tiene en cuenta y solo servirá para que el propio usuario pueda escribir lo que crea oportuno.

```
// Esto es un comentario, no se tiene en cuenta.

// Solo sirve para que el usuario escriba lo que crea oportuno.
```

3.3. Descriptores

Los descriptores funcionan como bloques para la definición de los datos (presentando un enfoque muy parecido a la programación orientada a objetos). Cada descriptor está compuesto por una serie de atributos que permiten describir sus características. Existen 2 tipos de descriptores:

- GAMEOBJECT.
- CHUNK.

3.3.1. Declarar descriptores

Un descriptor en el lenguaje es una estructura que agrupa varios atributos o propiedades bajo un único identificador, permitiendo representar objetos complejos. Para definir un descriptor, se declara el tipo de descriptor seguido de un nombre de variable, al cual se le asigna una instancia del descriptor con sus atributos inicializados.

DESCRIPTOR ID = DESCRIPTOR(atributo_1, atributo_2,..., atributo_n)

3.3.2. GAMEOBJECT

El descriptor GAMEOBJECT está diseñado para definir modelos 3D que se dibujarán de manera procedural en el terreno, como árboles, rocas, hierba u otros elementos decorativos del entorno. Este descriptor permite especificar ciertos atributos que controlan la apariencia y distribución de estos objetos dentro del terreno.

Atributos:

- **Ruta del modelo**: Define la ubicación del modelo 3D que se va a utilizar. En este caso, el ID "src/models/tree" se refiere a un archivo de un modelo 3D en esa ruta específica.

- **Densidad**: Define la densidad del objeto, lo que significa cuántos de estos objetos se generarán en el área correspondiente del terreno. Un valor de 56 indica que habrá 56 instancias del objeto dispersas por el chunk.

- **Escala**: Este atributo controla la escala del modelo. Sin embargo, si quieres que algunos sean más grandes y otros más pequeños, creando una distribución más natural. Para ello deberás definir los atributos MAX_SCALE y MIN_SCALE.

Ejemplo:

GAMEOBJECT objeto = GAMEOBJECT(“Ruta del modelo”, Densidad, Escala)

```
GAMEOBJECT object1 = GAMEOBJECT("src/models/tree", 10, 2.0)
GAMEOBJECT object2 = GAMEOBJECT("src/models/rock", 5, 0.5, 1.5)
```

En el primer ejemplo, define el modelo “tree” que se encuentra en la ruta “src/models” del proyecto, una densidad de 10 y una escala de 2.0.

En el segundo ejemplo, define el modelo “rock” que se encuentra en la ruta “src/models” del proyecto, una densidad de 5 y cada objeto se dibujará con una escala aleatoria entre 0.5 y 1.5.

3.3.3. CHUNK

Descriptor que representa una porción de terreno dentro de una escena. Este CHUNK tiene varios atributos clave que definen su apariencia y estructura, así como la posibilidad de contener objetos adicionales.

Atributos:

- **Posición x:** Posición X dentro de la matriz de chunks de la escena.
- **Posición y:** Posición Y dentro de la matriz de chunks de la escena.
- **Escala:** Escala del Ruido Perlin para controlar el detalle (transición suave o abrupta entre montaña y llano).
- **Multiplicador de altura:** Multiplicador de altura para ajustar la elevación del terreno.
- **Textura:** Cadena de texto que hace referencia a la textura del terreno en el proyecto.
- **Lista de objetos:** Lista de objetos dentro del chunk (ver apartado de listas de descriptores).

Ejemplo:

CHUNK objeto = CHUNK(x, y, escala, multiplicador de altura, textura, lista de objetos)

```
CHUNK global_chunk1 =  CHUNK(0,0,20.0, 7.5, "src/textures/grass", global_objects)
```

3.4. Listas de descriptores.

En nuestro lenguaje, es posible utilizar listas para dos tipos específicos de descriptores: CHUNK y GAMEOBJECT. Una vez creada una lista de uno de estos tipos, se podrán añadir elementos correspondientes a ese tipo. Para agregar un nuevo CHUNK o GAMEOBJECT a la lista, se utilizará la palabra reservada APPEND, donde el primer argumento será la lista a la que se desea añadir el elemento, y el segundo argumento será el elemento por añadir.

Por ejemplo:

```
GAMEOBJECT global_object1 = GAMEOBJECT("src/models/tree", 10, 2.0)
GAMEOBJECT global_object2 = GAMEOBJECT("src/models/rock", 5, 1.0)
LIST<GAMEOBJECT> global_objects = [global_object1, global_object2]
```

// Ejemplo de APPEND

```
APPEND global_objects GAMEOBJECT("src/models/house", 10, 2.0)
```

Además, la declaración de APPEND resulta especialmente útil para añadir descriptores de tipo CHUNK a una lista previamente declarada, particularmente cuando estos se crean dentro de un bucle. Esto permite generar múltiples CHUNKs de manera eficiente y rápida. Un ejemplo de su uso sería el siguiente:

```
// Creacion de 10 chunks con arboles y rocas
    LIST<CHUNK> chunks
    FOR i FROM 0 TO 10 {
      CHUNK c = CHUNK(0,i,15.0, 10.0, "src/textures/mountain", objects)
      APPEND chunks c
    }
```

3.5. Operaciones básicas.

- **Asignación (=):** nos servirá para asignar una expresión a una variable de un tipo de dato específico.

- **Operaciones aritméticas**:
  - **Suma (+):** suma dos elementos aritméticos de manera que dé lugar a un resultado. Cuando se sumen dos elementos de distinto tipo como un elemento de tipo int y otro de tipo float dará como resultado otro elemento de tipo float. También se pueden concatenar cadenas de caracteres alfanuméricos.

- **Resta (-):** resta dos elementos aritméticos de tipo float o entero. Cuando se resten dos elementos de distinto tipo como un elemento de tipo int y otro de tipo float dará como resultado otro elemento de tipo float.

- **Multiplicación (*):** multiplica dos elementos aritméticos de tipo float o entero de forma que dos elementos del mismo tipo darán lugar a un resultado con el mismo tipo de dato que el de los elementos que se van a multiplicar. De modo que cuando tengamos una multiplicación de dos enteros de lugar como resultado a otro entero y de forma similar con dos de tipo float. Cuando se multipliquen un entero con un float o viceversa, dará lugar un float como forma predeterminada.

- **División (/):** divide dos elementos aritméticos de tipo float o entero.

❖ **Operación ADD:**

Esta operación nos servirá para pintar un chunk dentro de una escena asegurando que solo se puedan utilizar chunks que hayan sido previamente definidos y declarados. Asimismo, se puede utilizar esta operación dentro de un bucle for para poder dibujar los múltiples chunks que están dentro de una lista.

Ejemplo:

```
// Añadir chunks
FOR c IN chunks {
  ADD c
}
```

3.6. Bucles for.

En nuestro lenguaje, existen dos tipos de bucles for. El primero recorre una variable de tipo FLOAT o INT previamente definida, la cual se incrementa hasta alcanzar un valor determinado. Un ejemplo de esta estructura es el siguiente, donde la variable i se incrementa de 1 en 1 hasta llegar a 5:

```
INT i = 0
FOR i FROM 0 TO 5 {
  // Código del bucle
}
```

El segundo tipo de bucle for está diseñado para recorrer una lista de elementos, como en el caso de una lista de CHUNKs. Un ejemplo de esta estructura es el siguiente:

```
FOR a IN chunks {
}
```

En este caso, a es un descriptor de tipo CHUNK y chunks es una lista que contiene descriptores de este tipo.

3.7. Estructura principal de un programa GeoCraft.

3.7.1. SETUP

Nuestro programa se organiza en dos bloques principales, comenzando con SETUP, que se centra en la definición de objetos, variables y chunks globales.

Ejemplo:

```
DEFINE SETUP(){
   // Definición de variables globales
   INT WIDTH_CHUNK = 20
   INT LENGH_CHUNK = 20

// Definición de objetos globales
   GAMEOBJECT global_object1 = GAMEOBJECT("src/models/tree", 10, 2.0)
   GAMEOBJECT global_object2 = GAMEOBJECT("src/models/rock", 5, 1.0)
   LIST<GAMEOBJECT> global_objects = [global_object1, global_object2]

   // Ejemplo de APPEND
   APPEND global_objects GAMEOBJECT("src/models/house", 10, 2.0)

   // Ejemplo de APPEND
   GAMEOBJECT global_object3 = GAMEOBJECT("src/models/car", 10, 2.0)
   APPEND global_objects global_object3

   // Ejemplo de CHUNK
   CHUNK global_chunk1 =  CHUNK(0,0,20.0, 7.5, "src/textures/grass", global_objects)
   }
```

3.7.2. WORLD

El bloque WORLD representa la metáfora del proyecto, actuando como la estructura organizativa principal que contiene todas las escenas del entorno. El atributo de WORLD es el nombre de la carpeta del proyecto, lo que ayuda a identificar y agrupar los elementos relacionados. Dentro de este bloque, se definen

varias escenas, donde cada escena corresponde a un archivo .cs individual. En estas escenas se llevan a cabo las definiciones de los chunks, así como la lógica del sistema, que incluye la creación de objetos, la utilización de bucles, y la gestión de listas. Así, WORLD sirve como un contenedor que agrupa y organiza todos los componentes necesarios para la construcción y funcionamiento del entorno virtual.

```
DEFINE WORLD("Mi Mundo"){
  // Definición de escena
        DEFINE SCENE("Escena 1", WIDTH_CHUNK, LENGH_CHUNK){
    // Definición de objetos
    GAMEOBJECT object1 = GAMEOBJECT("src/models/tree", 15, 1.8)
    GAMEOBJECT object2 = GAMEOBJECT("src/models/rock", 2, 1.2)
    LIST<GAMEOBJECT> objects = [object1, object2]
    // Creacion de 10 chunks con arboles y rocas
    LIST<CHUNK> chunks
                FOR i FROM 0 TO 10 {
      CHUNK c = CHUNK(0,i,15.0, 10.0, "src/textures/mountain", objects)
      APPEND chunks c
    }
    FOR c IN chunks {
      ADD c
    }
    // Añadir un chunk global
    ADD global_chunk
        }
  DEFINE SCENE("Escena 2", WIDTH_CHUNK + 10, LENGH_CHUNK + 10){
    // Definición de variables
    INT n = 5
    // Añadir un chunk global
    ADD global_chunk
        }
}
```

4. Procesador del lenguaje diseñado.

4.1. Esquema del proyecto.

La idea del proyecto consiste en tomar un código escrito en GeoCraft y pasarlo a través de un procesador de lenguaje diseñado para este fin.

Al ejecutar este proceso, se generará una carpeta que contendrá subcarpetas para cada escena definida en el código. Cada una de estas escenas se traducirá en un archivo de código con la extensión .cs, el cual, al ser ejecutado en Unity, creará automáticamente la escena correspondiente en el entorno 3D.

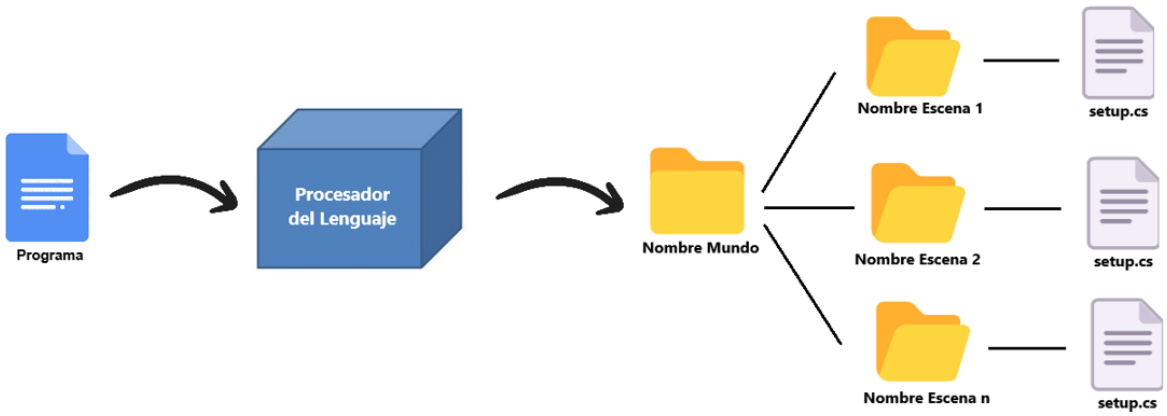


Ilustración 21: Esquema del proyecto.

4.2. Nuestro lenguaje.

GeoCraft es el nombre del nuevo lenguaje, combina las palabras "Geo", que proviene del latín y hace referencia a la Tierra, y "Craft", que se asocia con la idea de "craftear" o crear.

El archivo que contiene código en GeoCraft tendrá la extensión **.gec**, lo que lo identificará fácilmente.

Ilustración 22: Logo de GeoCraft

4.3. Diagrama en forma de “T”.

Lenguaje fuente: El lenguaje de entrada es GeoCraft (.gec).

Lenguaje objeto: Como salida se generarán scripts de C# (.cs).

Lenguaje de implementación: se han estudiado otras posibilidades como Java o C, pero se ha decidido que el lenguaje de implementación sea Python. Algunos de los motivos son su simplicidad y legibilidad, gran cantidad de librerías disponibles, flexibilidad y alto conocimiento del lenguaje por parte de los integrantes del grupo de trabajo.

Diagrama:

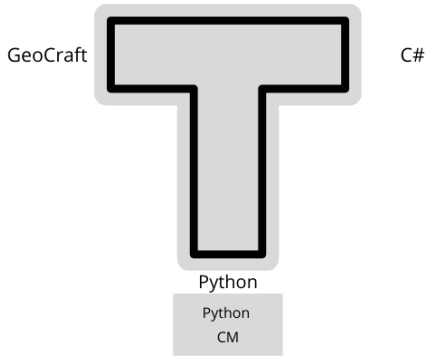


Ilustración 23: Diagrama en forma de T del proyecto.

5. Puntuación.

A continuación, se mostrará el porcentaje de trabajo por parte de los integrantes del equipo (el 100% sería el total de esfuerzo para la realización de la entrega):

Alumno	Esfuerzo
Francisco Javier Luna Ortiz	25%
Alejandro del Hoyo Abad	25%
Antonio Gómez Jimeno	25%
Felipe Sánchez Mur	25%

6. Bibliografía

6.1. Recursos web

https://medium.com/@christian.andres.1204/generaci%C3%B3n-procedural-eficiente-para-juegos-un-enfoque-te%C3%B3rico-5f23bb178a53

https://docs.unity3d.com/Manual/index.html

https://www.gamerfocus.co/juegos/la-generacion-procedimiento/

https://www.youtube.com/watch?v=Xc99wanf9Po&t=244s

https://www.youtube.com/watch?v=A0Bcj0uLHJ4&t=386s

6.2. Recursos gráficos

Ilustración 1: Imagen de allinonemovie en Pixabay

Ilustración 2: https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Ftse1.mm.bing.net%2Fth%3Fid%3DOIP.sUUhvLm_HPXWwLTj5Vk0fQHaEK%26pid%3DApi&f=1&ipt=3e2b741dfc146a3a5605890e6b71eefbfc39165d42f7fcef37aa66f0456a3826&ipo=images

Ilustración 3: https://external-content.duckduckgo.com/iu/?u=https%3A%2F%2Fminecraft.fr%2Fwp-content%2Fuploads%2F2021%2F06%2Fchunk-minecraft.png&f=1&nofb=1&ipt=21f6a2a15828a00c69c2ed7ee79ef09220cf9b51ae8558e3de12a7b77848808a&ipo=images

Ilustración 4: Coordenadas iconos creados por afif fudin - Flaticon