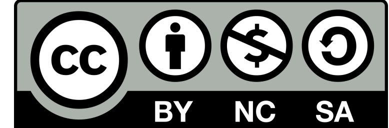
Programación con Python sobre Blender[1]

13 de noviembre de 2021

[1] Para programar sobre Blender con Python.

Este obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-CompartirIgual 4.0 Internacional.



Índice

Primeros pasos 1
- *Variables y operaciones básicas* 3
- *Listas, bucles y sentencias condicionales* 3
- *Funciones* 6
- *Módulos* 7

Operaciones básicas con Blender 7
- *Modificación de las propiedades de un objeto* 8
- *Inserción de fotogramas clave* 9
- *Un poco de limpieza* 9

Manipulación de una malla 10

Creación de una pequeña ciudad 11
- *Control de la densidad de edificios* 13
- *Automatización completa del proceso* 15

Creación de una interfaz de usuario 16
- *Especificación de requisitos de la interfaz* 16
- *Programación de interfaces en Blender* 17
- *Declaración de propiedades* 18
- *Creación de un panel* 19
- *Creación de un operador* 21

Creación de un complemento instalable 24
- *Paquetes Python* 25
- *Creación de un add-on instalable para Blender* 26
- *Importación de módulos durante el desarrollo* 26

Primeros pasos

La finalidad de este ejercicio es conocer los conceptos fundamentales de Python. El ejercicio se realizará utilizando Blender como entorno de ejecución de Python y muchos de los ejercicios permitirán conocer además el funcionamiento de Blender desde el punto de vista del desarrollo de complementos. Para empezar a trabajar debemos abrir Blender y poner la interfaz con la disposición para *Scripting*, según muestra la Figura 1.

En la configuración para desrrollo de *scripts*, el panel central es el editor de texto. Debemos crear un nuevo fichero de texto pulsando sobre el botón «New». En este nuevo fichero podemos empezar a escribir órdenes de Python. Empezaremos con el clásico *Hola mundo*, escribiendo

```
print("Hola mundo!")
```

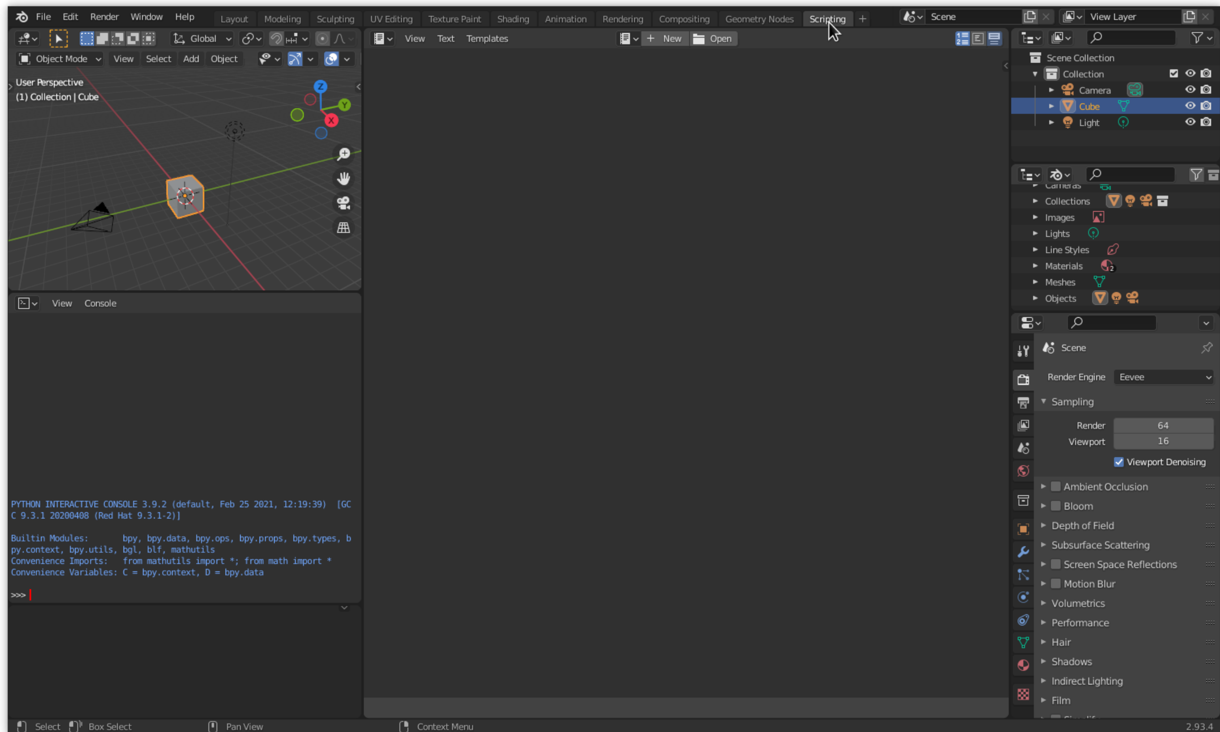


Figura 1: Abrimos Blender y ponemos la interfaz del programa en modo *Scripting* seleccionando la pestaña correspondiente en la parte superior del programa.

en la primera linea del nuevo fichero. La función `print` permite mostrar mensajes en la salida estándar del sistema. Cuando se ejecuta, inserta automáticamente un carácter de fin de linea al final de la cadena que se le pasa como argumento. Para ejecutar el código, pulsaremos el botón *Run* (con el símbolo triangular de *Reproducir*) que hay a la derecha de la barra superior (Figura 2).

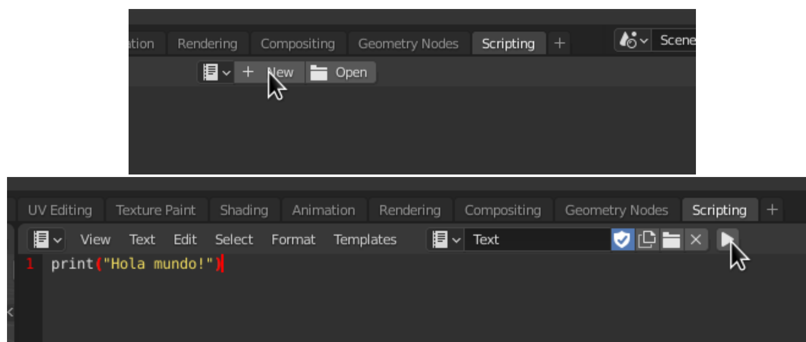


Figura 2: En el editor de texto creamos un nuevo fichero de texto, escribimos nuestro *script* y pulsamos el botón *Run*.

La salida del *script*, así como los posibles errores, aparecerán en la consola. En ordenadores Unix (Linux o macOS), debemos ejecutar Blender desde una terminal del sistema. Los mensajes aparecerán en esa terminal. En el sistema operativo Windows, podemos activar la consola desde el menú *Window*, seleccionando *Toggle system console*. La Figura 3 muestra la salida del *script* anterior.

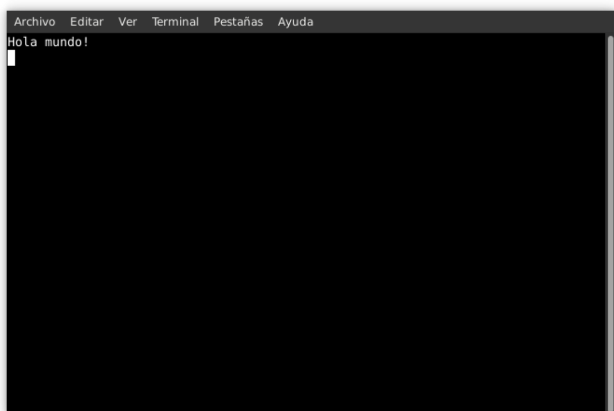


Figura 3: La salida de la ejecución de los scripts se muestra en la consola desde la que se ha ejecutado Blender. En el sistema operativo Windows se puede mostrar y ocultar esta consola desde el menú *Window*.

Variables y operaciones básicas

Python es un lenguaje interpretado. Por tanto no es necesario compilar el código. Cuando pulsamos el botón *Run* en el editor, éste entrega el código al intérprete que ejecuta una línea tras otra.

Python permite el uso de variables. Las variables, en Python no tienen un tipo predefinido como ocurren en C/C++. El tipo de una variable se modifica automáticamente cuando se le asigna un valor. Por ejemplo, si escribimos `a=3`, la variable `a` será de tipo entero, mientras que si escribimos `a='Hola mundo!'` la variable `a` será de tipo cadena[2].

[2] En Python, las constantes de tipo cadena pueden declararse utilizando tanto comillas simples como dobles.

Ejercicio: Para conocer los fundamentos del uso de variables en Python, sigue los pasos del Capítulo *Tipos Básicos* del manual *Python para todos*, de Raúl González Duque [3]. Crea un documento de texto que irás completando con las órdenes que se indican en el manual.

[3] Raúl González Duque. *Python para todos*, 2008. URL http://mundogeek.net/tutorial-python/

Listas, bucles y sentencias condicionales

En Python, una lista se construye por medio de corchetes, indicando los elementos de la lista separados por comas. Es posible crear una lista vacía por medio de la expresión `[]`. Algunas operaciones básicas son la inserción de un nuevo elemento al final de una lista, por medio del método `append` o borrar un elemento por medio del método `remove`. Las listas son heterogéneas. Es decir, que en una misma lista es posible mezclar elementos de diferentes tipos.

Las listas son objetos[4] sobre los que es posible iterar. Comúnmente, se les denomina iterables. Cualquier objeto iterable se puede recorrer por medio de un bucle `for`. La estructura de un bucle `for` en Python es:

[4] En Python, todos los tipos son, en realidad, objetos de alguna clase, incluidos los tipos básicos como los enteros o los números de coma flotante.

```
for variable in iterable:
    sentencias
```

Si la variable del bucle no estaba definida con anterioridad, se define en la primera iteración del bucle. Si el bucle se ejecuta al menos una vez, la variable queda definida para el resto del código y su valor tras el bucle es el que tenga al terminar la última iteración. El siguiente código crea una lista e imprime sus elementos:

```
##########
# Imprime:
# 1
# 2
# hola
# adios
a = [1,2,'hola']
a.append("adios")
for v in a:
    print(v)
```

En este último ejemplo vemos que las primeras 6 lineas empiezan con el carácter almohadilla, `#`. Este símbolo se utiliza para insertar comentarios. Cualquier texto que vaya a continuación de una almohadilla será ignorado por el intérprete. También es muy importante la indentación del código dentro del bucle `for`. En otros lenguajes, como C++ o Java, el conjunto de sentencias que deben ejecutarse dentro del bucle van delimitadas por llaves. En Python, todas las lineas que aparezcan a continuación de la linea del `for` y que tengan la misma cantidad de espacio en blanco delante están dentro del bucle. De hecho, en Python todo el código que esté en un mismo nivel de anidamiento debe tener la misma cantidad de espacio en blanco delante. Por ejemplo, el código

```
a = [1,2,'hola']
 a.append("adios")
 for v in a:
    print(v)
```

dará error antes de ejecutarse[5], porque la segunda y la tercera líneas están al mismo nivel de anidamiento que la primera pero tienen una cantidad diferente de espacio en blanco. Es importante fijarse bien en la indentación porque esto puede dar lugar a programas completamente diferentes con sólo cambiar el espacio en blanco de una linea.

[5] Cuando esto ocurre, Python aroja un error de indentación con el mensaje `IndentationError: unexpected indent`

Por ejemplo, el siguiente código recorre la lista `a`, mostrando sus elementos uno a uno, e insertándolos en una nueva lista `b` que está inicialmente vacía. Al finalizar, mostrará la lista `b`.

```
a = [1,2,'hola','adios']
b = []
for v in a:
    b.append(v)
    print(v)
print(b)
```

La salida que se mostrará en la consola al ejecutar el código anterior es:

```
1
2
hola
adios
[1, 2, 'hola', 'adios']
```

Por el contrario, el siguiente código muestra la lista `b` cada vez que inserta un elemento en ella, porque la linea `print(b)` está dentro del bucle `for`.

```
a = [1,2,'hola']
a.append("adios")
b = []
for v in a:
    b.append(v)
    print(v)
    print(b)
```

que da lugar a la siguiente salida en la consola:

```
1
[1]
2
[1, 2]
hola
[1, 2, 'hola']
adios
[1, 2, 'hola', 'adios']
```

Las sentencias condicionales en Python tienen la forma[6]

```
if expresion_booleana:
    sentencias
```

donde `expresion_booleana` debe ser una expresión que pueda evaluarse como una variable Booleana[7]. En las sentencias condicionales se aplican las mismas reglas de indentación indicadas anteriormente: todas las sentencias que aparezcan tras un `if` y que tengan un nivel de indentación similar forman parte de ese bloque condicional.

Las estructuras de control pueden anidarse, para lo que habrá que añadir más espacio en blanco delante de las lineas de cada nivel. A continuación se muestra un ejemplo en el que se calcula la intersección de dos listas y se imprime el resultado[8]. Para ello, con un bucle `for` se recorre una lista `a` y para cada elemento se comprueba si éste pertenece a otra lista `b` y lo inserta en una tercera lista `intersec_ab`.

[6] Las sentencias de control de flujo en Python terminan siempre con el carácter dos puntos.

[7] Como ocurre en otros lenguajes, muchos tipos permiten una conversión automática a Booleano. Por ejemplo, un entero se convierte a `True` si es no nulo y a `False` si es 0 y una lista se convierte a `False` si está vacía y a `True` en caso contrario.

[8] En el ejemplo aparece otro uso de la palabra reservada `in`. Además de emplearse en los bucles `for`, la palabra `in`, usada como operador infijo, devuelve `True` si el primer operando es un elemento del segundo.

```
1  a = [1,3,5,2,7,9]
2  b = [1,4,1,9,8]
3  intersec_ab
4  for x in a:
5      if x in b: # True si x es un elemento de b
6          intersec_ab.append(x)
7  print(intersec_ab)
```

La linea del `if` tiene 4 espacios en blanco para indicar que está dentro del bloque del `for` y la linea `intersec_ab...` tiene 8 espacios en blanco para indicar que se encuentra, a su vez, dentro del `if`. La cantidad de espacio en blanco es irrelevante, aunque la recomendación de las guías de estilo de Python es emplear 4 espacios adicionales para cada nuevo nivel. También se desaconseja el uso de tabuladores y, en Python 3, no se permite mezclar espacios y tabuladores en un mismo fichero de código[9].

[9] Las recomendaciones de estilo se recogen en la guía PEP8 (`https://www.python.org/dev/peps/pep-0008/`). Es importante conocerlas, porque muchos estudios de animación y VFX exigen que su código se acoja a esta recomendación.

Ejercicio: Sigue los pasos de los Capítulos *Colecciones* y *Control de flujo* del manual de González Duque anteriormente citado. Añade al fichero del ejercicio anterior las órdenes que se indican en el manual.

Funciones

Como en otros lenguajes de programación, en Python es posible encapsular bloques de código por medio de funciones para poder invocarlas desde cualquier parte de un programa. La sintaxis para definir una función es

```
def nombre(lista_args):
    sentencias
```

donde `lista_args` es una lista de argumentos[10] separados por comas (y sin corchetes). Si se desea que la función devuelva un valor, puede emplearse la orden `return` seguida del valor a devolver.
El último ejemplo puede reescribirse, usando una función, de la siguiente forma:

[10] En Python los argumentos de las funciones se pasan por referencia.

```
1  def intersect(l1,l2):
2      intersec = []
3      for x in l1:
4          if x in l2: # True si x es un elemento de b
5              intersec.append(x)
6      return intersec
7
8  a = [1,3,5,2,7,9]
9  b = [1,4,1,9,8]
10 intersec_ab = intersect(a,b)
11 print(intersec_ab)
```

Ejercicio: Sigue el Capítulos *Funciones* del manual de González Duque anteriormente citado. Añade al fichero de los ejercicios anteriores las órdenes que se indican en el manual.

Módulos

Para la realización de actividades muy frecuentes es común escribir un conjunto de rutinas, funciones, clases,... y guardarlos para su uso posterior desde diferentes programas. En Python, a cada uno de estos conjuntos de código se les denomina *módulos*. Para utilizar un módulo basta con utilizar la orden *import*, seguida del nombre del módulo, antes de emplear sus funciones o clases. Por ejemplo, el módulo `math` contiene constantes y funciones matemáticas frecuentes. En el siguiente ejemplo se muestra la raíz cuadrada de los 10 primeros números[11].

```
1 import math
2
3 for n in range(10):
4     print(math.sqrt(n))
```

[11] La función `range(x)` devuelve un iterable equivalente a la lista `[0,1,2,...,x-1]`.

Para desarrollar herramientas para Blender es necesario importar el módulo `bpy` (de Blender Python) que proporciona acceso a todos los tipos de datos definidos en el programa y a los datos existentes en un fichero `.blend` en particular. Si se escribe el siguiente código en un fichero de texto del editor de Blender y se ejecuta, imprimirá en la consola el nombre de todos los objetos contenidos en la escena actual.

```
1 import bpy
2
3 for ob in bpy.context.scene.objects:
4     print(ob.name)
```

Operaciones básicas con Blender

A continuación vamos a llevar a cabo algunas operaciones sencillas habituales en Blender utilizando un *script* en Python. Para programar sobre Blender, es muy útil estar familiarizados con la documentación de la API de Blender [12]. Sin embargo, existe una forma muy sencilla de desarrollar *scripts* sin necesidad de consultar continuamente esta documentación.

[12] Blender Online Community. *Blender 2.83.0 Python API Documentation*. Blender Foundation, Blender Institute, Amsterdam, 2020. URL https://docs.blender.org/api/2.93/

Para saber cómo realizar cualquier acción en Python, es tan sencillo como llevarla a cabo a través de la interfaz de Blender y comprobar el panel de información. En este panel, cada vez que realizamos una acción desde la interfaz se muestra la orden en Python que se ha ejecutado (Figura 4).

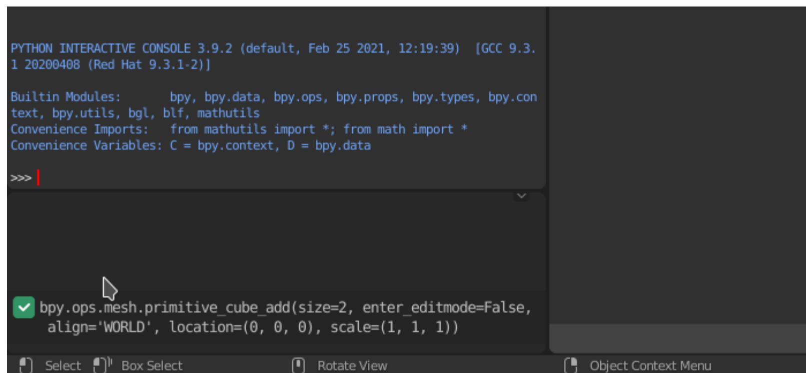


Figura 4: Para conocer la orden en Python que lleva a cabo una acción podemos ejecutarla en la interfaz de usuario y la orden se mostrará en el panel de información de Blender.

En primer lugar vamos a añadir un objeto. Para ello, situamos el ratón sobre la vista 3D de la parte superior izquierda, pulsamos `Mays+A`[13] y seleccionamos *Mesh* → *Cube* de menú. Una vez realizada la acción, vemos que, en el panel de información, aparece la orden[14]

```
1 bpy.ops.mesh.primitive_cube_add(size=2,
2             enter_editmode=False, align='WORLD',
3             location=(0, 0, 0), scale=(1, 1, 1))
```

Si creamos un *script* que contenga esa orden y lo ejecutamos, se añadirá un cubo. El cubo tendrá por arista el valor del parámetro `size`, estará centrado en el punto `location`, y se le aplicará un escalado descrito en el parámetro `scale`. Para una lista completa de argumentos y su significado se puede consultar la página de documentación de esta orden[15].

[13] `Mays+A` es el atajo de teclado para *Añadir*

[14] Es común dividir las listas de argumentos en varias lineas si la linea es muy larga.

[15] https://docs.blender.org/api/2.93/bpy.ops.mesh.html

Modificación de las propiedades de un objeto

En la mayoría de los casos, cuando llevamos a cabo una acción en Blender, ésta se realiza sobre el *objeto activo*. El objeto activo es el último que se seleccionó o el último que se creó. Para acceder al objeto activo podemos acceder a la variable `bpy.context.active_object`. También podemos acceder a un objeto arbitrario utilizando su nombre como clave en el diccionario `bpy.context.scene.objects`. El siguiente código crea un cubo, guarda una referencia al objeto activo en una variable y modifica su posición, fijándola en las coordenadas $(1, 1, 1)$. Además, sitúa el cubo original de la escena en las coordenadas $(-1, -1, -1)$ accediendo al objeto por su nombre.

```
1 import bpy
2 
3 bpy.ops.mesh.primitive_cube_add(size=2,
4             enter_editmode=False, align='WORLD',
5             location=(0, 0, 0), scale=(1, 1, 1))
6 
7 nuevo_cubo = bpy.context.active_object
```

```
8  nuevo_cubo.location = (1,1,1)
9
10 cubo_original = bpy.context.scene.objects['Cube']
11 cubo_original.location = (-1,-1,-1)
```

Ejercicio: Utilizando un bucle, genera una hilera de 10 cubos separados entre sí 2 metros. Utilizando las funciones del módulo `random` de Python, modifica ligeramente las dimensiones de los cubos, evitando que se toquen entre sí.

Inserción de fotogramas clave

El problema de fijar la posición de un objeto simplemente modificando su propiedad[16] `location` es que el objeto se quedará estático en esa posición cuando se reproduzca la animación.

[16] En Blender se denominan *propiedades* a lo que en orientación a objetos se le suele llamar *atributos*.

Para llevar a cabo animaciones, la herramienta fundamental son los fotogramas clave, que permiten asignar posiciones en instantes concretos de tiempo. Para insertar un fotograma clave de posición, podemos invocar el método `keyframe_insert` de la clase *Object*, de la que heredan la mayoría de objetos de la escena.

```
obj = bpy.context.active_object
obj.keyframe_insert(data_path='location')
```

que insertará un fotograma clave que fijará la posición actual para el fotograma actual. Si queremos que la posición actual quede registrada en otro fotograma utilizaremos el parámetro `frame`. Por ejemplo, para forzar a que el objeto se encuentre en la posición $(1, 1, 1)$ en el fotograma 20

```
obj.keyframe_insert(data_path='location',frame=20)
```

Ejercicio: Extiende el *script* que genera la hilera de cubos, insertando fotogramas clave para que su posición inicial tenga una coordenada z muy alta, quedando fuera de cuadro, y bajen hasta su posición original tras un par de segundos.

Un poco de limpieza

La programación de *scripts* no sólo sirve para generar escenas y programar animaciones. Una utilidad muy importante es la automatización de tareas que, de otra forma pueden ser tediosas. El *script* resultante del último ejercicio añade múltiples objetos cada vez que se ejecuta. Así, tras ejecutarlo varias veces podemos acabar con un buen número de cubos superpuestos en la escena.

Puede ser de utilidad disponer de un script que elimine todos los objetos que hemos creado. Para ello, recorreremos todos los objetos de la escena y comprobaremos su tipo, por medio de la propiedad `type` presente en todos los objetos y la orden `bpy.ops.object.delete()`. Esta orden borra todos los objetos que estén seleccionados. Por tanto, debemos identificar si un determinado objeto es una malla (su propiedad `type` tendrá el valor `'MESH'`) y seleccionarlo con la orden

```
obj.select_set(True)
```

Para evitar borrar objetos no deseados, debemos deselccionar todos los objetos antes de empezar a seleccionar objetos, con la orden

```
bpy.ops.object.select_all(action='DESELECT')
```

Ejercicio: Crea un nuevo *script* que recorra todos los objetos de la escena, seleccionando los que sean de tipo `'MESH'`, y los elimine.

El problema que tiene la solución anterior es que también borrará cualquier otra malla que introduzcamos en la escena. Para solucionar esta limitación podemos modificar el nombre de cada uno de los objetos que introduzcamos, introduciendo alguna cadena que nos permita identificar qué objetos debemos borrar. En los siguientes ejemplos usaremos la cadena `Proc_Object` para indicar los objetos que hemos añadido de forma procedimental. Bastará con introducir la linea

```
nuevo_cubo.name = 'Proc_Object'
```

cuando introduzcamos los nuevos cubos. Blender añadirá un sufijo consistente en un número tras la cadena introducida por nosotros, para evitar duplicados.

A la hora de eliminar los cubos se puede cambiar la comprobación de que el objeto sea una malla por la comprobación de si el nombre del objeto contiene la subcadena `Proc_Object`[17].

[17] Para saber si una cadena es subcadena de otra podemos usar de nuevo el operador `in`; la expresión Booleana `cadena1 in cadena2` se evaluará como `True` si la cadena `cadena1` es una subcadena de la cadena `cadena2`.

Ejercicio: Modifica el *script* anterior para que utilice la comprobación basada en una cadena para identificar los objetos a borrar.

Manipulación de una malla

Blender cuenta con un módulo para manipular las mallas de los objetos que la tienen[18]. Se trata del módulo bmesh. Para acceder a la malla de un objeto de la escena hay que crear un objeto BMesh e inicializarlo con la información geométrica del objeto. El siguiente código imprimie las coordenadas de los vértices del objeto activo[19].

[18] No todos los objetos de una escena tienen malla. Objetos como las cámaras, las luces o los huesos no tienen una malla.

[19] El objeto `bpy.context.object` es otra referencia al objeto activo.

```
import bpy
import bmesh

obj = bpy.context.object

# Creamos un objeto de tipo BMesh y cargamos los datos
bm = bmesh.new()
bm.from_mesh(obj.data)

# Imprimimos
# Las coordenadas estan en el sistema de referencia
# local del objeto
for v in bm.verts:
    print(v.co)
```

También es posible modificar las coordenadas. Pero, en este caso, debemos volver a grabar la información sobre la malla del objeto. Es decir, cuando manipulamos una malla por medio de un objeto *BMesh* éstas se hacen sobre una copia y no afectan al objeto. El siguiente código escala las coordenadas de los vértices del objeto activo y graba los cambios.

```
# Escalamos las coordenadas de los vertices
for v in bm.verts:
    v.co *= 0.5

# Guardamos el resultado y borramos el objeto bm
bm.to_mesh(obj.data)
bm.free()
```

Ejercicio: Escribe un *script* que genere un objeto de tipo `grid` y modifique la altura de los vértices (coordenada z) de forma aleatoria en un rango de $\pm 0{,}1$ unidades.

Creación de una pequeña ciudad

Combinando los *scripts* diferentes ejercicios anteriores, vamos a crear un pequeño paisaje urbano. Para ello, seguiremos los siguientes pasos:

1. Crearemos un objeto de tipo `grid` y modificaremos la coordenada z de sus vértices de forma que represente un terreno con cierto relieve. Para el cálculo de la altura se recomienda usar una suma de Gaussianas o de funciones polinómicas de base radial.

2. Para cada vértice crearemos un cubo, que escalaremos para que represente un edificio. Sus coordenadas (x, y) serán las mismas que las del vértice de la malla y la coordenada z deberá estar por encima del suelo a la altura adecuada.

3. Los cubos deben estar cerca, pero no tocarse, para que se formen calles.

4. Las alturas de los diferentes cubos deberán ser distintas. La coordenada z de cada cubo debe calcularse para que se adapte al terreno, de forma que su base quede ligeramente enterrada. No se considera correcto que medio edificio esté enterrado.

5. Los edificios deberán aparecer con una animación en la que parecerá que caen desde el cielo.

La Figura 5 muestra imágenes de diferentes fases de la construcción de la ciudad y la Figura 6 muestra una imagen renderizada.

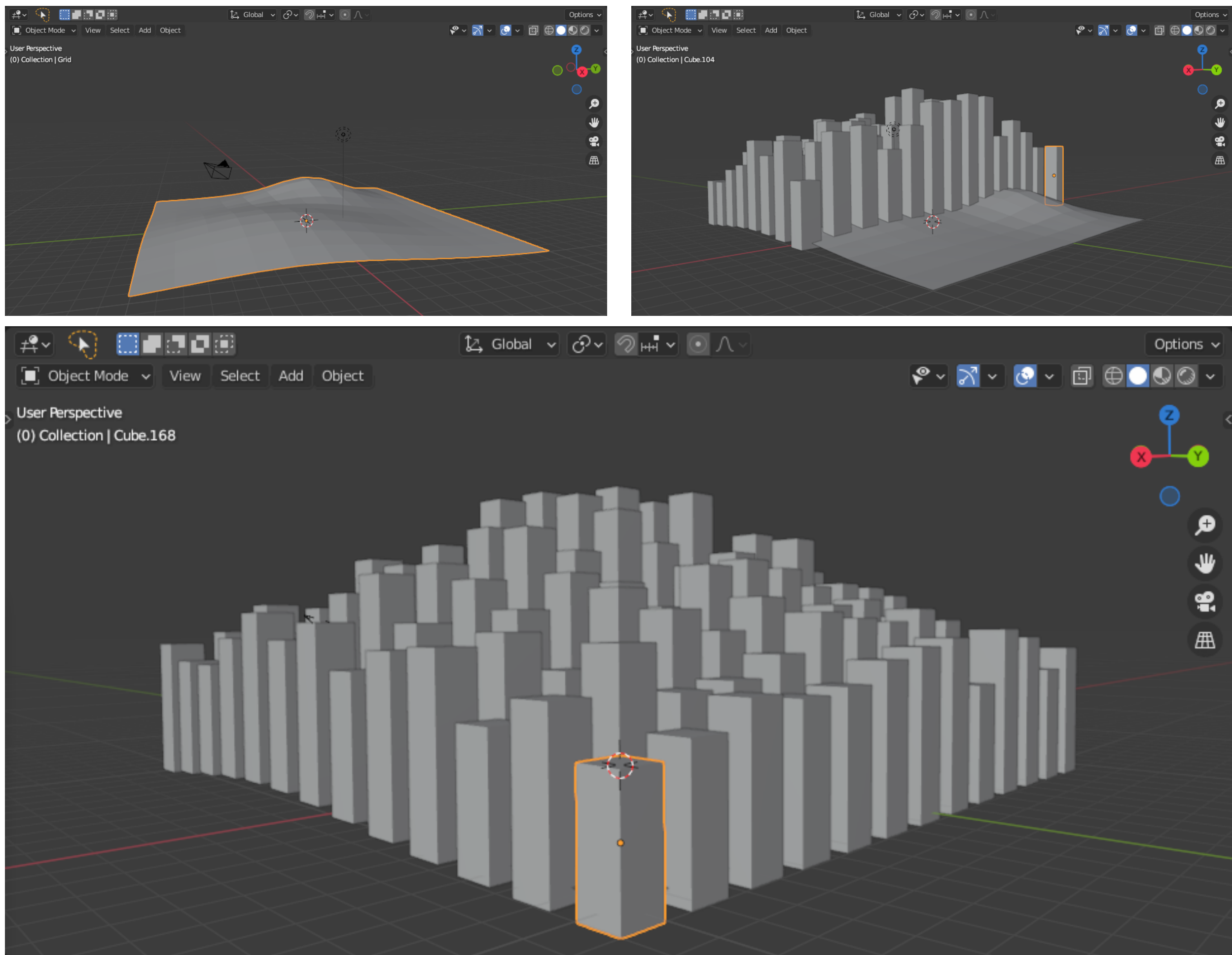


Figura 5: Diferentes fases de la construcción de una ciudad por medio de un script.

Ejercicio: Escribe un *script* que genere la escena descrita. El *script* debe cumplir los siguientes requisitos:

- Debe estar guardado en un fichero fuera de Blender. Para ejecutarlo deberemos emplear la plantilla *External script stub* disponible en el editor de texto de Blender. El *script* en Python debe estar en la misma carpeta que el fichero Blend.

- Cada edificio debe tener una altura diferente. Se valorará que tenga también una anchura diferente, pero dejando siempre espacio para que se formen calles.
- Para el cálculo de la altura del suelo debe implementarse una función `altura(x,y)` que, dadas dos coordenadas (x, y) devuelva el valor de la altura z.
- El *script* debe empezar con una llamada a la función que borra las mallas de la escena. De esta forma, cada vez que se ejecute el *script* aparecerá un nuevo escenario sin dejar las geometrías del escenario anterior.
- En los fotogramas iniciales, los edificios deberán estar muy por encima del suelo. Los edificios caerán hasta su posición final. Para ello, bastará con insertar dos fotogramas clave cada vez que se cree el cubo de un edificio. La animación no debería durar más de 3-5 segundos.

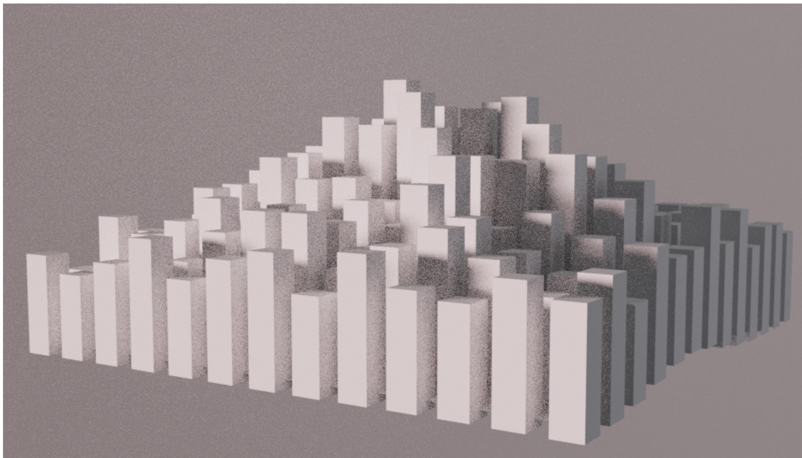


Figura 6: Imagen renderizada de la ciudad procedural.

Control de la densidad de edificios

Nuestra cuidad tendrá edificios en todos los vértices de la malla. Esto provoca que la forma de la ciudad sea completamente cuadrada y regular. Generar una cuidad que tuviera más densidad de edificios en el centro de la ciudad y que ésta decreciera a medida que nos alejamos hacia el exterior.

Para conseguir este efecto podemos usar una función que tenga un valor próximo a 1 en la región central y cuyo valor decrezca según nos alejamos. Podemos llamar a esta función p, porque será la probabilidad de que haya un edificio a una determinada distancia del centro.

Si llamamos p_{max} a la probabilidad máxima que queremos considerar y p_{min} a la probabilidad mínima, una función que podemos usar para modelar el comportamiento deseado es la tangente hi-

perbólica. Esta función converge a -1 cuando $x \to -\infty$, converge a 1 cuando $x \to \infty$ y presenta una transición suave alrededor del 0 (Figura 7).

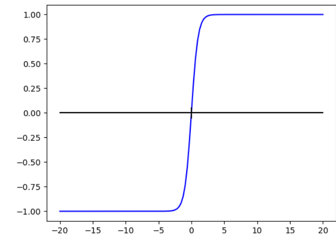


Figura 7: La función tangente hiperbólica, en el intervalo $[-20, 20]$.

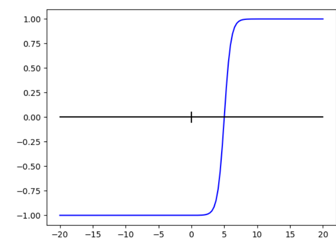
Si queremos que la transición tenga lugar en un determinado punto (por ejemplo, en $x = 5$), entonces podemos restar 5 al valor de la x en el argumento de la función. Así la función

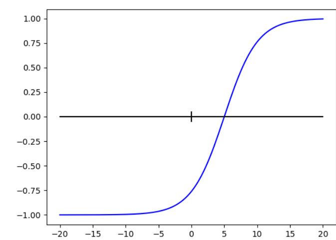
$$f(x) = \tanh(x-5)$$

tendrá la transición desde -1 a 1 alrededor de $x = 5$. También podemos controlar la brusquedad de la transición multiplicando el argumento de la función por un número positivo a, de forma que la transición será tanto más brusca cuanto mayor sea el valor de a. Además, si el valor de a es negativo, la función hará la transición de 1 a -1. La Figura 8 presenta las gráficas de las funciones

$$f(x) = \tanh(x-5); \qquad f(x) = \tanh(0{,}2(x-5));$$

$$f(x) = \tanh(-0{,}2(x-5)).$$





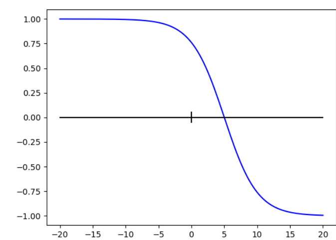


Figura 8: Diferentes transformaciones de la función tangente hiperbólica para modificar el punto, brusquedad y sentido de la transición entre $y = -1$ e $y = -1$.

Combinando todas estas ideas podemos diseñar una función que modele una probabilidad alta de aparición de edificios en el centro y una probabilidad baja en las zonas periféricas de la escena. Vamos a construir una función que tenga un valor próximo a 1 en el origen, cuyo valor se mantenga alto en un rango de distancias y luego decrezca hasta un valor próximo a 0. La Figura 9 muestra la gráfica de la función

$$p(x) = 0{,}5 + 0{,}45 \tanh\left[-0{,}4(x-12)\right].$$

que pasa de un valor muy próximo a $p = 0,95$ cuando $x = 0$ a un valor próximo a $p = 0{,}05$ cuando $x \to \infty$.

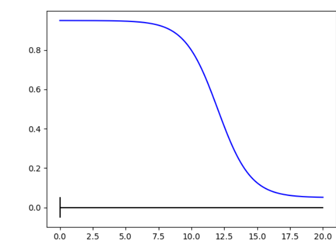


Figura 9: Función de probabilidad para la aparición de un edificio en función de la distancia al origen.

Para emplear esta función, debemos modificar el código de nuestro generador de ciudades de la siguiente forma. En primer lugar, implementaremos una función en Python `probabilidad_edificio` que implemente la función matemática anterior

```
1  def probabilidad_edificio(x,y):
2      # Calculamos la distancia al origen del punto (x,y)
3      dist = math.sqrt(x**2 + y**2)
4  
5      # Calculamos y devolvemos la probabilidad de
6      # edificio para ese punto
7      prob = math.tanh(-0.4*(dist-12))*0.45 + 0.5
8      return prob
```

Cada vez que vayamos a generar y configurar un edificio, haremos la siguiente comprobación antes de su creación:

```
if random() > probabilidad_edificio(v.co.x,v.co.y):
    # codigo para generar el edificio.
```

De una forma equivalente podemos generar un perfil para la altura promedio de los edificios en función de la distancia al centro o, incluso, para diferentes regiones. La Figura 10 muestra una imagen de una ciudad generada con ambas modificaciones.

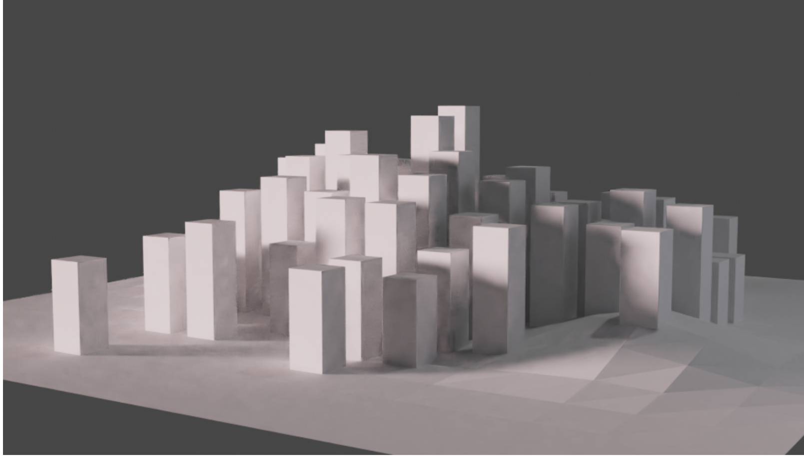


Figura 10: Imagen renderizada de una ciudad procedural en la que los edificios aparecen con mayor probabilidad y tienen mayor altura en la zona centro.

Ejercicio: Modificad el código para incorporar la probabilidad de que aparezca un edificio en función de la distancia al centro y, opcionalmente, para modificar su altura, haciendo que los edificios sean más altos en el centro.

Automatización completa del proceso

Con el *script* que hemos creado hasta ahora necesitamos tener una escena configurada sobre la que nuestro programa genera la ciudad. Podemos automatizar completamente proceso para que la escena se genere completamente, incluyendo la colocación de las luces, la cámara y otros elementos de escenario.

En este caso, nos convendría iniciar el *script* seleccionando todos los objetos de la escena y eliminándolos (¡¡cuidado con esta acción!!). A partir de ahí, crearíamos la cámara, el suelo, los bloques y cualquier otro objeto que queramos incorporar a la escena[20].

[20] Una forma cómoda de generar el código que permite generar, a su vez, la escena, es construir la escena con la interfaz de Blender y copiar las instrucciones que aparecen en el panel de información para pegarlas en el *script*.

Una vez tengamos un *script* con todo automatizado, podemos generar el render y guardarlo a disco con las siguientes órdenes al final del *script*:

```
# Mostramos el ultimo fotograma, para ver los edificios
bpy.context.scene.frame_set(bpy.context.scene.frame_end)
# Se genera en el mismo sitio en que est'a el fichero blend
# La doble barra inicial indica ruta relativa
bpy.context.scene.render.filepath="//ciudad.png"
bpy.ops.render.render(write_still=True)
```

Por último, podemos realizar todo el proceso sin necesidad de abrir la interfaz de Blender. Simplemente ejecutando Blender desde la terminal y diciéndole que utilice el *script*, que previamente habremos guardado en disco. Si el script lo hemos guardado en el fichero `script.py`, entonces ejecutaremos la orden:

```
$ blender --background --python ciudad.py
```

Ejercicio: Modificad el *script* para que genere la escena automáticamente y genere un render.

Creación de una interfaz de usuario

La herramienta que hemos desarrollado hasta el momento nos permite generar una ciudad con relativa facilidad y flexibilidad. Sin embargo, para poder utilizarla es necesario ejecutar el *script* Python de forma explícita. Además, cualquier modificación de los parámetros del proceso (número de edificios, altura, variabilidad, etc), requiere tener unas nociones mínimas de programación. Esto hace que no sea adecuado como herramienta para su uso en un contexto de producción. A continuación vamos a crear una interfaz gráfica, que se integrará con la interfaz de usuario de Blender y que nos permitirá utilizar el generador de ciudades de forma cómoda y sin conocimientos de programación. Este apartado está parcialmente basado en el Capítulo 4 del libro-wiki «Blender 3D: Noob to pro»[21]. Puede encontrarse también información adicional en el libro de Chris Conlan (2017)[22]. Aunque ambos textos utilizan versiones de Blender anteriores a la 2.8, la mayoría de los ejemplos que se presentan allí funcionan en las versiones más recientes de Blender.

[21] Wikibooks. Blender 3D: Noob to Pro — Wikibooks, the free textbook project, 2019. URL https://en.wikibooks.org/w/index.php?title=Blender_3D:_Noob_to_Pro&oldid=3848363. [Online; accessed 29-September-2021]

[22] Chris Conlan. *The Blender Python API*. Apress, 2017. DOI: 10.1007/978-1-4842-2802-9

Especificación de requisitos de la interfaz

Para poder llevar a cabo la creación de la interfaz gráfica para nuestra herramienta de creación de ciudades, en primer lugar vamos a detallar qué opciones de configuración vamos a exponer al usuario y cuál va a ser el comportamiento que esperamos de la herramienta resultante.

Desde la interfaz se deberán poder seleccionar las dimensiones del suelo en los dos ejes y el número de calles en cada una de las dos direcciones. Además, se deberá poder seleccionar si se desea modificar el suelo de forma aleatoria e introducir parámetros que permitan controlar la aleatoriedad[23]. Una vez configurados los parámetros del suelo, el usuario deberá poder generar el suelo sin generar los edificios.

[23] Si hemos utilizado funciones de base radial, la aleatoriedad se puede controlar indicando cuántas funciones se deben generar.

Para la generación de los edificios, desde la interfaz se deberá poder indicar la altura promedio y la variabilidad de esta altura.

También se deberá poder indicar la anchura mínima de las calles.Por último, deberá poderse indicar un porcentaje de celdas que, en promedio, deberán contener edificios, de forma que algunas queden sin ocupar. Configurados los parámetros de generación de los edificios, se deberá poder crear el conjunto de bloques sobre los vértices del objeto activo utilizando los parámetros anteriores.

Con esta definición de la interfaz, la persona que haga uso de la herramienta deberá configurar el suelo, generar la malla, configurar los edificios y generar la ciudad teniendo seleccionada la malla del suelo. Pero, además, permite generar los edificios sobre otra malla que puede haberse generado utilizando las herramientas de modelado de Blender o algún algoritmo de generación de terreno.

Resumiendo todo lo anterior el conjunto de parámetros que utilizaremos queda recogido en la Tabla 1. Vemos que, con el conjunto de requisitos que hemos definido, algunas de las funcionalidades que habíamos implementado en el apartado anterior no quedan contempladas, como la reducción de la probabilidad de edificios al alejarnos del centro. Este tipo de funcionalidades pueden añadirse incorporando nuevos parámetros de configuración que se emplearían en las fórmulas correspondientes y quedarán como un ejercicio.

Tabla 1: Parámetros que utilizaremos para la definición de la interfaz de usuario de la herramienta.

Nombre	Tipo	Descripción
`tam_x_suelo`	Float	Tamaño del suelo en el eje x, en unidades de longitud
`tam_y_suelo`	Float	Tamaño del suelo en el eje y, en unidades de longitud
`n_calles_x`	Int	Número de calles en la dirección del eje x
`n_calles_y`	Int	Número de calles en la dirección del eje y
`deformar_suelo`	Booleano	Indica si el suelo debe deformarse
`n_rbf`	Int	Número de funciones de base radial para deformar el suelo
`alt_edificios`	Float	Altura promedio de los edificios (en el eje z)
`var_edificios`	Float	Variabilidad de la altura de los edificios
`ancho_calles`	Float	Anchura de las calles
`prob_edificio`	Float	Probabilidad de edificio en cada vértice. Valor entre 0 y 1.

Programación de interfaces en Blender

Una vez tenemos una definición de cómo queremos interactuar con nuestra herramienta generadora de ciudades, debemos conseguir que esa funcionalidad quede integrada en la interfaz gráfica de

Blender. Blender cuenta con una Interfaz de Programación de Aplicaciones (API, por sus siglas en inglés) que nos permite conseguir este objetivo[24].

[24] La documentación de la API de Blender puede encontrarse en la URL https://docs.blender.org/api/2.93/index.html

Aunque disponemos de más formas de interacción, como los menús, en este ejercicio nos limitaremos al uso de *paneles*, que son conjuntos de elementos de la interfaz que se agrupan en una parte de la interfaz general de Blender. Los diferentes elementos que aparecen a la derecha en la vista por defecto de Blender y que permiten configurar las propiedades de los objetos son paneles. Un panel nos permite exponer al usuario el valor de una propiedad (de un objeto, de la escena,. . .) y también permite que este valor se pueda modificar. Además, permite incluir botones vinculados a acciones que, en la mayoría de los casos, son lo que se denominan *operadores*. A continuación crearemos las propiedades listadas en la Tabla 1, un panel para poder modificarlas y dos operadores para crear el suelo y los edificios.

Declaración de propiedades

En el contexto de la API de Blender a los atributos de las clases se les denomina *propiedades* (*properties* en la terminología en inglés). Los atributos de las clases son dinámicos y pueden añadirse durante la ejecución del programa. Si bien no es posible declarar variables globales, se pueden añadir propiedades a todos los tipos de datos existentes en Blender. Los tipos de datos a los que pueden pertenecer estas propiedades no son, sin embargo, libres, sino que deben pertenecer a alguno de los tipos definidos en la API bajo el módulo `bpy.props`[25]. Así, por ejemplo, si queremos declarar una propiedad de tipo entero instanciaremos un objeto de la clase `bpy.props.IntPoperty`[26].

[25] https://docs.blender.org/api/2.93/bpy.props.html

[26] El constructor permite definir un valor por defecto, un rango de valores permitido o funciones de *callback*, entre otros parámetros.

Las propiedades deben asignarse a una clase o tipo concreto de la jerarquía de tipos de Blender. Es decir, no podemos simplemente crear una propiedad, sino que debemos indicar que determinado objeto va a tener esa propiedad. En el siguiente ejemplo, hacemos que todos los objetos de tipo `bpy.types.object` tengan una propiedad, de tipo `float`, que se llamará `ancho_calles`.

```
bpy.types.Object.ancho_calles = bpy.props.FloatProperty()
```

Una vez ejecutemos esa linea en la consola de Python de Blender o en un *script* todos los objetos de la escena que sean de la clase `Object` contendrán un nuevo atributo llamado `ancho_calles` de tipo `float`. Si ejecutamos la orden

```
bpy.data.objects['Camera'].ancho_calles = 3
print(bpy.data.objects['Camera'].ancho_calles)
```

se asignará el valor 3 a ese atributo de la cámara y a continuación mostrará el valor 3.

Siguiendo este procedimiento, vamos a declarar todas las propiedades necesarias para nuestra herramienta. Las propiedades relativas al suelo las añadiremos sobre el tipo `bpy.types.Scene`. De esta forma se podrán modificar aunque aún no hayamos creado un suelo. Las propiedades relativas a los edificios las añadiremos sobre el tipo `bpy.types.Object`, de forma que los edificios se generarán sobre el objeto que esté seleccionado utilizando sus propios parámetros.

Para que más tarde podamos hacer instalable la herramienta, todo el código relativo a la declaración de variables debe ir en una función llamada `register()`. Esta función quedará de la siguiente forma:

```
1 def register():
2     bpy.types.Scene.tam_x_suelo = bpy.props.FloatProperty(min = 1,default = 10)
3     # continua...
4 
5     bpy.types.Object.alt_edificios = bpy.props.FloatProperty(min = 1,default = 2)
6     # continua...
```

Para poder deshacer los cambios introducidos por el código anterior debemos implementar la función `unregister()` que invocaremos si queremos eliminar las propiedades declaradas. Esta función se invocará de forma automática para desinstalar la herramienta si la hemos instalado en Blender. En ella, eliminaremos las diferentes propiedades con la orden `del`

```
1 def unregister():
2     del bpy.types.Scene.tam_x_suelo
3     del bpy.types.Scene.tam_y_suelo
4     # continua...
```

Ejercicio: En un nuevo *script*, implementad las funciones `register()` y `unregister()` para declarar y eliminar todas las propiedades que se han indicado. Estas funciones deben ser las últimas del *script*. Modificad el código que genera la ciudad para que utilice estas propiedades en lugar de variables locales del *script* o constantes. Para hacerlo funcionar habrá que ejecutar el *script* que declara las propiedades antes de generar la ciudad por primera vez.

Creación de un panel

La creación de un nuevo panel en Blender requiere *registrar* una clase de tipo Panel. Para ello declararemos una nueva clase, que herede de la clase `bpy.types.Panel` y a continuación la registraremos con la función `bpy.utils.register_class()`. El aspecto del panel se configura en el método `draw()`, donde los elementos se organizan en filas (*rows*) en las que se pueden añadir propiedades. A

continuación presentamos un ejemplo en el que se declara y registra un panel básico[27]. En primer lugar encontramos la declaración de la clase:

[27] Este ejemplo se ha creado modificando la plantilla *Simple panel* que puede cargarse desde el menú *Templates* del editor de texto de Blender.

```
1  import bpy
2
3  class ProceduralCityPanel(bpy.types.Panel):
4      """ Creates a Panel to generate a Procedural
5          City in the 3D viewport
6      """
7      bl_label = "Procedural City"
8      bl_idname = "OBJECT_PT_ProceduralCity"
9      bl_space_type = 'VIEW_3D'
10     bl_region_type = 'UI'
11     bl_category = "Procedural City"
12
13     def draw(self, context):
14         layout = self.layout
15
16         obj = context.object
17         scene = context.scene
18
19         row = layout.row()
20         row.label(text="Creacion del suelo", icon='WORLD_DATA')
21         row = layout.row()
22         row.prop(scene, "tam_x_suelo")
23         row.prop(scene, "tam_y_suelo")
24         # Continua ...
25
26         row = layout.row()
27         row.label(text="Creacion de los edificios")
28         row = layout.row()
29         row.prop(obj, "alt_edificios")
30         row.prop(obj, "var_edificios")
31         # Continua ...
```

La clase `ProceduralCityPanel` se declara en la linea 3, usando la palabra reservada `class` seguida del nombre. En este caso, el nombre de la clase va seguido de paréntesis que contienen a la clase `bpy.types.Panel`, indicando que la nueva clase es una clase derivada de ésta. La función `draw()` indica, a partir de la linea 13, qué elementos se van a mostrar al usuario. Las declaraciones de atributos en las lineas 7 a 11 nos permiten indicarle a Blender dónde queremos que se muestre el panel. En este caso, en la vista 3D, en el panel derecho (UI) en una pestaña llamada «Procedural City».Para que el panel aparezca, registramos la clase `ProceduralCityPanel` en la función `register()`.

```
32 def register():
33     bpy.types.Scene.tam_x_suelo = bpy.props.FloatProperty(name = "x",
34                                                         description="Tamanyo suelo x",
35                                                         min = 1,
36                                                         default = 10)
37     bpy.types.Scene.tam_y_suelo = bpy.props.FloatProperty(name = "y",
38                                                         description="Tamanyo suelo y",
39                                                         min = 1,
40                                                         default = 10)
41     # continua...
```

```
42
43      bpy.types.Object.alt_edificios = bpy.props.FloatProperty(name = "Altura",
44                                                               description="Altura_media_edificios",
45                                                               min = 1,default = 2)
46      bpy.types.Object.var_edificios = bpy.props.FloatProperty(name = "Variacion",
47                                                               description="Variabilidad_edificios",
48                                                               min = 0,default = 1)
49      # continua...
50
51      bpy.utils.register_class(ProceduralCityPanel)
52
53  def unregister():
54      bpy.utils.unregister_class(ProceduralCityPanel)
55
56      del bpy.types.Scene.tam_x_suelo
57      del bpy.types.Scene.tam_y_suelo
58      # continua...
59
60
61  if __name__ == "__main__":
62      register()
```

Las lineas 62 y 63 del *script* ejecutarán la función `register()` siempre que se ejecute el *script*, pero el `if` de la linea 62 impide que se ejecute si el fichero se carga como un módulo usando `import` desde otro programa[28]. El resultado de ejecutar el código anterior se muestra en la Figura 11.

[28] Si el fichero se carga como un módulo, la variable `__name__` toma el nombre del módulo, mientras que toma el valor `"__main__` cuando se ejecuta como un programa.

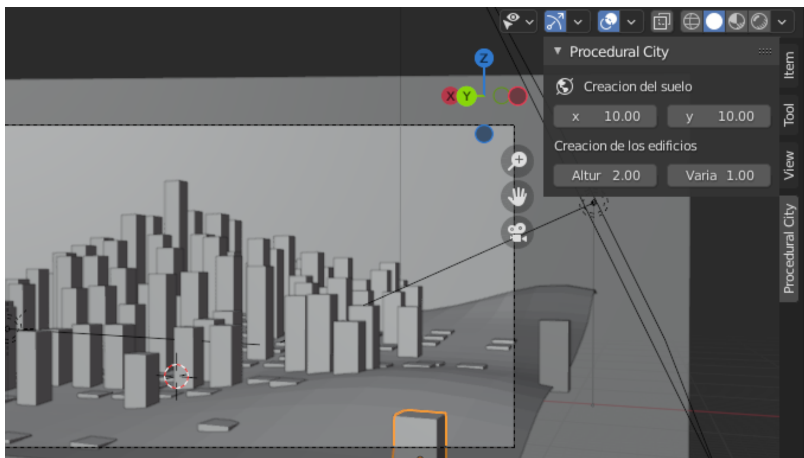


Figura 11: Ejemplo de un panel que expone al usuario varias de las propiedades que hemos creado para nuestra herramienta.

Creación de un operador

Con el panel que hemos creado podemos configurar la creación de la ciudad. Sin embargo, aún necesitamos invocar el *script* de forma explícita. Para poder llevar a cabo una acción en Blender, que esté implementada por medio de código Python, necesitamos crear una clase que herede de la clase `bpy.types.Operator` e invocar nuestro código desde el método `invoke()` de esta clase, que

debemos implementar. Posteriormente podemos hacer referencia al operador desde un panel y de forma automática se mostrará un botón que, al ser pulsado, ejecutará el método invoke().

Para nuestra herramienta vamos a crear dos operadores; uno para introducir el suelo y otro para generar los edificios. A continuación se presenta un primer borrador de operador[29].

[29] El ejemplo parte de la plantilla *Operator Simple* disponible en el editor de texto de Blender.

```
class GenerateGroundOperator(bpy.types.Operator):
    """Genera un grid con deformaciones aleatorias"""
    bl_idname = "object.gen_ground"
    bl_label = "Crea_suelo"

    def invoke(self, context):
        # usamos una variable para la escena
        scene = context.scene
        # mostramos una de las nuevas propiedades, para probar
        print(scene.tam_x_suelo)
        return {'FINISHED'}
```

Para que el operador pueda usarse, debemos registrarlo de la misma forma que registramos el panel. Para ello, añadiremos la siguiente linea a la función register()

```
bpy.utils.register_class(GenerateGroundOperator)
```

y la siguiente linea a la funcion unregister()

```
bpy.utils.unregister_class(GenerateGroundOperator)
```

Para que este operador pueda ejecutarse, al final del método draw() de la clase ProceduralCityPanel añadiremos una referencia a este operador mediante las lineas

```
row = layout.row()
row.operator("object.gen_ground")
```

En la Figura 12 puede verse cómo ha aparecido un botón en la parte inferior del panel que habíamos introducido en el ejercicio anterior. De forma automática, este botón queda tiene vinculada como función de *callback* el método invoke() del operador.

En estos momentos, la ejecución del operador pulsando el botón no dará ningún resultado, salvo mostrar en la consola el valor de la propiedad tam_x_suelo que hemos declarado. Antes de completar la implementación de la herramienta debemos organizar el código de tal forma que sea sencillo invocarlo desde el operador. Para ello, debemos organizar todo el código que teníamos para generar la escena en funciones. La función para generar el suelo podría quedar según se muestra a continuación.

```
def crea_suelo(tam_x,tam_y,n_x,n_y,n_bumps):
    """
    Crea un suelo irregular por medio de un objeto de tipo Grid.

    """
```

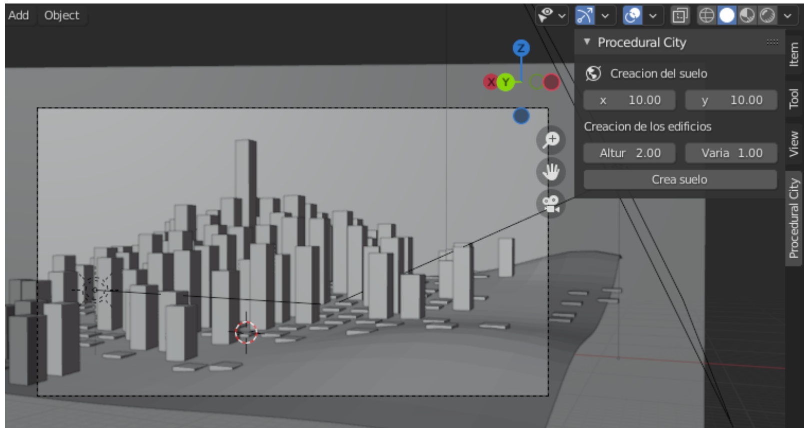


Figura 12: Ejemplo del panel tras añadir un botón que invocará al operador.

```
6
7       # Creacion del suelo de la ciudad
8       bpy.ops.mesh.primitive_grid_add(x_subdivisions=n_x,
9           y_subdivisions=n_y,
10          size=1,
11          enter_editmode=False,
12          align='WORLD',
13          location=(0, 0, 0),
14          scale=(tam_x, tam_y, 1))
15
16      # Las dimensiones las hemos aplicado usando la escala. Ahora hacemos que
17      # esas sean las dimensiones reales del objeto, para que no haya problemas
18      # con las transformaciones
19      bpy.ops.object.transform_apply(location=False, rotation=False, scale=True)
20
21      suelo = bpy.context.object
22
23      # Creamos un objeto de tipo BMesh y cargamos los datos
24      bm = bmesh.new()
25      bm.from_mesh(suelo.data)
26
27      # Modificamos la coordenada z de los vertices
28      for v in bm.verts:
29          # Llamamos a una funcion que genera las alturas
30          v.co.z = altura_suelo(v.co.x,v.co.y,... #completar
31
32      bm.to_mesh(suelo.data)
33      bm.free()
34
35      return suelo
```

Una vez hayamos movido el código de generación del suelo a esta función, el operador puede modificarse para que se genere el suelo al pulsar el botón incluido en el panel de la interfaz gráfica.

```
1   class GenerateGroundOperator(bpy.types.Operator):
2       """Genera un grid con deformaciones aleatorias"""
3       bl_idname = "object.gen_ground"
4       bl_label = "Crea_suelo"
5
6       def invoke(self, context):
```

```
        # usamos una variable para la escena
        scene = context.scene

        # Recogemos los parametros de la generacion del suelo. No
        # es necesario utilizar estas variables intermedias, pero
        # hace mas legible el codigo

        x = scene.tam_x_suelo
        y = scene.tam_y_suelo
        nx = scene.n_calles_x
        ny = scene.n_calles_x
        if scene.deformar_suelo:
            nb = scene.n_rbf
        else:
            nb = 0

        crea_suelo(x,y,nx,ny,nb)

        return {'FINISHED'}
```

Ejercicio: Aplica los cambios necesarios al *script* para que tanto el suelo como los edificios se generen desde el interfaz de Blender. Para ello tendrás que mover todo el código fuente a funciones y crear un operador para generar el suelo, según se ha mostrado, y otro para añadir los edificios.

Ejercicio (opcional): Añade los parámetros necesarios para recuperar el control sobre la densidad de edificios en función de la distancia al centro.

Ejercicio (opcional): Si en el *script* original tu programa tenía más funcionalidades de las que se han propuesto en este ejercicio, incorpóralas al interfaz.

Creación de un complemento instalable

Para poder distribuir con facilidad la herramienta que hemos desarrollado convendría que pudiera instalarse como un complemento o *add-on*. Blender permite crear complementos de forma sencilla; es suficiente con crear un *paquete* de Python con nuestro complemento y comprimirlo en un fichero de tipo `zip`. Veremos en primer lugar la creación de paquetes Python y, a continuación, veremos que requisitos impone Blender a la hora de crear un complemento.

Paquetes Python

Python cuenta con una lista de directorios en los que buscar los módulos cuando ejecutamos `import`. Un paquete es un módulo que se instala en uno de estos directorios. Esto consiste en organizar todos los ficheros en un directorio con el nombre del módulo y copiarlo a una de estas rutas en las que Python busca al invocar `import`. Crear e instalar un módulo con nuestro código permite usarlo sin necesidad de tener los ficheros en el mismo directorio en el que estamos trabajando.

Para hacer un paquete debemos tener en cuenta algunos detalles. En primer lugar, debe existir un fichero llamado `__init__.py` que es el que realmente se cargará al importar el módulo. Cualquier otro fichero debe importarse desde este fichero. Además, cuando preparamos código para generar un paquete debemos modificar ligeramente la forma en la que importamos el resto de ficheros desde dentro del módulo[30]. A continuación lo vemos con un ejemplo.

[30] En la documentación de Python, en el apartado sobre módulos, podemos encontrar información más detallada. https://docs.python.org/3/tutorial/modules.html#packages

Supongamos que tenemos un módulo que queremos llamar `city` y que contiene un fichero principal `city.py` y dos ficheros ground.py y `buildings.py` que se importan desde el anterior. En el inicio del fichero `city.py` se importan los otros dos con

```
import ground
import buildings
```

Para crear el paquete copiaremos los tres ficheros a un directorio llamado `city/` y, además, renombraremos el fichero `city.py` como `__init__.py`, quedando la siguiente estructura:

```
city/
city/__init__.py
city/ground.py
citiy/buildings.py
```

Además, las lineas de `import` que hagan referencia a otros ficheros del propio módulo deben cambiarse al formato

```
from path import fichero
```

donde `path` es una ruta relativa al directorio del paquete. Esto nos permite organizar el paquete en varias carpetas que actuarán como submódulos. En nuestro caso no vamos a crear subdirectorios, por lo que la ruta será siempre el directorio actual. De esta forma, en el fichero `__init__.py` cambiaremos las lineas

```
import ground
import buildings
```

por la nueva forma de hacer las importaciones:

```
from . import ground
from . import buildings
```

Una vez organizado y modificado el código, copiamos el directorio (con permisos de administrador) a uno de los directorios en los que Python busca los módulos de forma automática y ya podemos usar nuestro módulo sin tener una copia de los ficheros.

Creación de un add-on instalable para Blender

Los *add-ons* de Blender son, simplemente, paquetes que se instalan en la distribución de Python integrada en el programa. Para preparar un instalador, debemos preparar el paquete Python según se ha indicado anteriormente y comprimirlo en formato `zip`. Pero, además, Blender requiere la definición de un diccionario que proporcione información sobre el *add-on*, como la autoría, el acceso a la documentación o los requisitos de versión. Para ello, debemos definir el diccionario `bl_info` en el fichero `__init__.py`. A continuación se muestra un ejemplo de diccionario para el complemento desarrollado[31]

[31] En el ejemplo se muestran sólo algunas opciones. Para conocer más detalles puede consultarse la URL https://wiki.blender.org/wiki/Process/Addons/Guidelines/metainfo.

```
bl_info = {
    "name": "Procedural city generation",
    "description" : "SPH solver for Blender scenes",
    "author" : "Ignacio Garcia Fernandez",
    "version" : (0, 2),
    "blender" : (2, 90, 0),
    "category" : "Object",
}
```

Además, es necesario que el fichero principal del *add-on* tenga implementadas las funciones `register()` y `unregister()` según se ha indicado en este documento. La función `register()` se ejecutará al instalar el *add-on* y cada vez que arranque Bleneder con el `add-on` instalado. La función `unregister()` se ejecutará al desinstalar el *add-on*.

Importación de módulos durante el desarrollo

La generación del instalador es la fase final del desarrollo del *add-on* y, una vez depurado, yo no suele ser necesario modificar el código con frecuencia. Sin embargo, durante la fase de desarrollo puede resultar incómodo generar e instalar el paquete tras cada modificación. Por eso, el fichero principal lo ejecutaremos directamente desde una instancia de Blender, empleando el *External script stub* descrito en los apartados anteriores. Sin embargo, la configuración de `import` necesaria para usar el código como un paquete no es compatible con esta forma de ejecutar el código.

Para poder gestionar esta situación mientras estamos programando y probando la herramienta debemos incluir en el código las

dos formas de importar módulos y distinguirlas a través de la variable `__name__`. Si el fichero se está ejecutando como un *add-on* esta variable tomará como valor el nombre del módulo (el del directorio en el que hayamos metido el código). Por tanto, comprobando el valor de esta variable podemos elegir la forma de cargar los módulos asociados al *add-on*.

```
if __name__ == "city":
    from . import ground
    from . import buildings
else:
    print("Loading␣procedural␣city␣as␣a␣script")
    import ground
    import buildings
```

Además, debemos tener en cuenta un detalle importante; una vez se han cargado los módulos `ground` y `buildings` Blender ya no los cargará de nuevo aunque ejecutemos el *script*[32]. Para asegurarnos de que los cambios que hagamos en `ground.py` y en `buildings.py` tienen efecto debemos recargarlos cada vez que se ejecuta el *script* usando la función `reload()` del módulo `importlib`, añadiendo las siguientes lineas tras el bloque de importación:

[32] Esto se hace así porque se entiende que un módulo es algo estable, que no cambia, y por tanto no requiere su recarga cada vez que ejecutamos un *script*. Bajo esta idea, los módulos sólo se cargan la primera vez para reducir el tiempo de ejecución del *script*. Sin embargo, esta premisa no es cierta cuando estamos modificando el propio módulo.

```
from importlib import reload
reload(ground)
reload(building)
```

Con todo lo anterior, las primeras lineas de nuestro fichero principal quedarán de la siguiente forma:

```
bl_info = {
    "name": "Procedural␣city␣generation",
    "description" : "SPH␣solver␣for␣Blender␣scenes",
    "author" : "Ignacio␣Garcia␣Fernandez",
    "version" : (0, 2),
    "blender" : (2, 90, 0),
    "category" : "Object",
}


if __name__ == "city":
    from . import ground
    from . import buildings
else:
    print("Loading␣procedural␣city␣as␣a␣script")
    import ground
    import buildings

from importlib import reload
reload(ground)
reload(building)
```

Referencias

Blender Online Community. *Blender 2.83.0 Python API Documentation*. Blender Foundation, Blender Institute, Amsterdam, 2020. URL https://docs.blender.org/api/2.93/.

Chris Conlan. *The Blender Python API*. Apress, 2017. DOI: 10.1007/978-1-4842-2802-9.

Raúl González Duque. *Python para todos*, 2008. URL http://mundogeek.net/tutorial-python/.

Wikibooks. Blender 3D: Noob to Pro — Wikibooks, the free textbook project, 2019. URL https://en.wikibooks.org/w/index.php?title=Blender_3D:_Noob_to_Pro&oldid=3848363. [Online; accessed 29-September-2021].