Trabajo Fin de Máster

Ingeniería de Caminos, Canales y Puertos

Uso de Blender a través de la programación en la Ingeniería de Caminos, Canales y Puertos

Autor: Miguel Puech de Oriol

Tutora: Cristina Torrecillas Lozano

Departamento de Ingeniería Gráfica

Sevilla, 2021

Departamento de Ingeniería Gráfica ETSI

Proyecto Fin de Máster
Ingeniería de Caminos, Canales y Puertos

Uso de Blender a través de la programación en la Ingeniería de Caminos, Canales y Puertos

Autor:

Miguel Puech de Oriol

Tutora:

Cristina Torrecillas Lozano

Profesora Titular de Universidad

Departamento de Ingeniería Gráfica

Escuela Técnica Superior de Ingeniería

Universidad de Sevilla

Sevilla, 2021

Proyecto Fin de Máster: Uso de Blender a través de la programación en la Ingeniería de Caminos, Canales y Puertos

Autor: Miguel Puech de Oriol

Tutora: Cristina Torrecillas Lozano

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2021

El Secretario del Tribunal

Resumen

Este documento introduce la posibilidad de utilizar *Blender* en la Ingeniería de Caminos, Canales y Puertos como herramienta de diseño 3D a través de la programación de utilidades en *Python* y mediante la programación visual con árboles de nodos. Para ello, se han creado cinco herramientas que demuestran las capacidades que un programa como *Blender*, creado para un ámbito ajeno al mundo de la ingeniería, tiene para un ingeniero relacionado con la construcción civil.

Las cinco herramientas creadas incluyen aspectos de Caminos, Canales y Puertos, para comprobar la viabilidad del uso de este software en el sector. Estas utilidades permiten al usuario:

- Caminos: realizar modelos tridimensionales del terreno y extraer perfiles longitudinales de carreteras,
- Canales: crear tuberías normalizadas para distintos materiales,
- Puertos: crear rocas naturales y bloques de hormigón artificial para diques.

Los códigos desarrollados se pueden ejecutar de forma directa en Blender, y en el caso de la utilidad desarrollada para la construcción de tuberías es posible su instalación como un complemento, lo que demuestra la capacidad prácticamente ilimitada de personalización del entorno de trabajo.

Para una mejor comprensión de su programación, se hace una introducción a la creación de objetos, mallas, materiales, modificadores, texturas y otros elementos de modelado existentes en *Blender*.

Se ha podido comprobar a través de las herramientas creadas:

- la capacidad de utilizar distintas fuentes de datos,
- la capacidad de realizar modelos 3D y 2D indistintamente,
- la versatilidad en el desarrollo de los modelos,
- la capacidad de automatización de procesos,
- la posibilidad de obtener resultados fotorrealistas aplicando materiales a los objetos,
- la capacidad de generar elementos de geometría conocida mediante la parametrización de estos
- y modelos de geometría imperfecta y aleatoria mediante generación procedimental.

Finalmente, hay que indicar que la existencia de complementos realizados por terceros permite ampliar las capacidades del programa sin la necesidad de tener conocimientos de programación y, todo esto, a través de un programa completamente gratuito.

Abstract

This document introduces the possibility of using Blender in civil engineering as a 3D modelling tool through programming with Python and visual programming with node trees. For this, five tools have been created to demonstrate the capabilities that a program like Blender, created for a foreign field to the world of engineering, has for an engineer.

The five created tools include aspects of Roads, Channels and Ports, to prove the viability of using the software in this sector. These utilities allow the user:

- Roads: build three-dimensional models of the terrain and extract longitudinal profiles of roads,
- Channels: build standardized pipes for different materials,
- Ports: build natural rocks and artificial concrete breakwaters blocks.

The developed codes can be executed directly in Blender, and in the case of the utility developed to build pipes it is possible to install it as an add-on, which demonstrates the practically unlimited capacity of customization of the working environment.

For a better comprehension, an introduction is made of the creation of objects, meshes, materials, modifiers, textures and other modelling Blender elements.

It has been possible to prove through the developed tools:

- the capability to use different data sources,
- the capability to create 3D and 2D models indistinctly,
- the versatility in the models' development,
- the capability of process automatization,
- the possibility to obtain photorealistic results applying materials to the objects,
- the capability to generate elements of known geometry through parametrization,
- and models of imperfect and random geometry through procedural generation.

Finally, it should be noted that the existence of third-party add-ons allows to increase the program capabilities without the need to have any programming knowledge, and all this within a completely free program.

Índice

Resumen VII
Abstract IX
Índice XI
Índice de Figuras XIII
1. Introducción 16
1.1. Introducción al problema 16
1.2. Sobre Blender 18
1.3. Objetivos 18
2. Blender 20
2.1. Introducción 20
2.2. Entorno de trabajo 21
2.3. Python 24
2.4. Herramientas desarrolladas 25
3. Caminos 26
3.1. Modelo digital del terreno 26
3.1.1. Triangulación de Delaunay 26
3.1.2. Modifiers 30
3.1.3. Añadir materiales al modelo del terreno 33
3.1.4. Blender GIS 38
3.2. Extracción de perfiles longitudinales de una carretera 40
4. Canales 44
4.1. Tuberías 44
5. Puertos 51
5.1. Escollera natural (Generación procedimental) 51
5.2. Escollera artificial (Generación parametrizada) 59
6. Otras herramientas 63
7. Conclusiones 64
Bibliografía 66
ANEJO I: Caminos 68
ANEJO II: Canales 86
ANEJO III: Puertos 102

Índice de Figuras

FIGURA 1. LAYOUT UTILIZADO EN BLENDER, SE APRECIAN EN EL 6 DE LAS 21 VENTANAS EXISTENTES, ASÍ COMO ALGUNAS HERRAMIENTAS. 21
FIGURA 2. VENTANA DE EDICIÓN 3D VIEWPORT EN MODO EDIT MODE. 22
FIGURA 3. VENTANA DE EDICIÓN PROPERTIES. 23
FIGURA 4. EDITOR DE TEXTO INCLUIDO EN BLENDER CON UNA PLANTILLA PARA REALIZAR COMPLEMENTOS CON LOS QUE CREAR OBJETOS. 24
FIGURA 5. ESTRUCTURA Y REPRESENTACIÓN GRÁFICA DE UN ARCHIVO .ASC. 27
FIGURA 6. ESTRUCTURA Y REPRESENTACIÓN GRÁFICA DE UN ARCHIVO .XYZ. LAS COLUMNAS REPRESENTAN DE IZQUIERDA A DERECHA LAS COORDENADAS X, Y Y Z DE LOS PUNTOS. 28
FIGURA 7. MODELO TRIDIMENSIONAL DEL TERRENO GENERADO A PARTIR DE UNA TRIANGULACIÓN DE DELAUNAY CON BLENDER. 30
FIGURA 8. DETALLE DEL MODELO TRIDIMENSIONAL DEL TERRENO GENERADO A PARTIR DE UNA TRIANGULACIÓN DE DELAUNAY CON BLENDER, DONDE SE PUEDE VER LA MALLA DE TRIÁNGULOS. 30
FIGURA 9. MALLA BASE PARA EL MODELO TRIDIMENSIONAL DEL TERRENO GENERADO CON BLENDER. 31
FIGURA 10. MODELO TRIDIMENSIONAL DEL TERRENO GENERADO A PARTIR DE MODIFICADORES CON BLENDER. A) 3 DIVISIONES, B) 6 DIVISIONES, C) 9 DIVISIONES Y D) 11 DIVISIONES. 33
FIGURA 11. MATERIAL TIPO RAMPA DE COLORES PARA UN MODELO TRIDIMENSIONAL DEL TERRENO. 34
FIGURA 12. CONJUNTO DE NODOS PARA LA NORMALIZACIÓN. 35
FIGURA 13. MATERIAL TIPO ORTOFOTO. 36
FIGURA 14. MODELO TRIDIMENSIONAL DEL TERRENO COLOREADO MEDIANTE UNA RAMPA DE COLORES DEFINIDA EN FUNCIÓN DE LA ALTURA. 37
FIGURA 15. MODELO TRIDIMENSIONAL DEL TERRENO COLOREADO MEDIANTE UNA ORTOFOTO. 38
FIGURA 16. COMPARATIVA DE MALLA SUAVIZADA A) Y NO SUAVIZADA B). 38
FIGURA 17. MODELO GENERADO CON BLENDER GIS CON ORTOFOTO, ALTIMETRÍA Y EDIFICIOS (NUEVA YORK). 39
FIGURA 18. PERFIL GENERAL DE UN TRAMO DE LA CARRETERA A-373. SE HA ESTABLECIDO EL PK DE INICIO EN 50, LA SEPARACIÓN ENTRE MARCAS DEL EJE X DE 250M Y DEL EJE Y DE 10M. 43
FIGURA 19. TRAMO DE LA CARRETERA A-373. 43
FIGURA 20. ANILLOS EN FORMA DE POLÍGONO REGULAR CON LOS QUE CREAR LA MALLA QUE REPRESENTA UNA TUBERÍA (CILINDRO HUECO). A) R = 3 Y B) R = 8. 44
FIGURA 21. MALLA CON FORMA TUBULAR. 45
FIGURA 22. DISEÑOS QUE ADOPTA EL PANEL DEL COMPLEMENTO CREADO EN FUNCIÓN DE LOS PARÁMETROS ESCOGIDOS: A) TUBERÍA NO NORMALIZADA, B) TUBERÍA NORMALIZADA DE HA, C) TUBERÍA NORMALIZADA DE POLIETILENO Y D) TUBERÍA NORMALIZADA DE PVC. 49
FIGURA 23. TUBERÍAS OBTENIDAS A TRAVÉS DEL COMPLEMENTO CREADO: A) DE HORMIGÓN ARMADO DN 400, B) DE FUNCIÓN DÚCTIL DN 500, C) DE POLIETILENO DE ALTA DENSIDAD PN-16 DN 315 Y D) DE PVC PN-16 DN 250. 50
FIGURA 24. TUBERÍAS OBTENIDAS A TRAVÉS DEL COMPLEMENTO CREADO CON LOS MATERIALES ELEGIDOS PARA LAS MISMAS: A) DE HORMIGÓN ARMADO DN 400, B) DE FUNCIÓN DÚCTIL DN 500, C) DE POLIETILENO DE ALTA DENSIDAD PN-16 DN 315 Y D) DE PVC PN-16 DN 250. 50
FIGURA 25. A) CUBO PRIMITIVO Y B) CUBO CON VÉRTICES MODIFICADOS DE FORMA ALEATORIA. 52
FIGURA 26. CUBO MODIFICADO: A) REDONDEADO, B) PLANO (ESCALA_Z=0,2) Y C) ACICULAR (ESCALA_Y=0.2, ESCALA_Z=0,2). 53
FIGURA 27. CUBO MODIFICADO CON MALLA DE MAYOR DENSIDAD: A) REDONDEADO, B) PLANO (ESCALA_Z=0,2), C)

ACICULAR (ESCALA_Y=0.2, ESCALA_Z=0,2). 53
FIGURA 28. MUESTRAS DE RUIDO GENERADO DE FORMA PROCEDIMENTAL, MÉTODOS: A) CLOUDS, B) VORONOI Y C) STUCCI. 54
FIGURA 29. ESTRUCTURA DEL MATERIAL CREADO PARA LAS ROCAS. 55
FIGURA 30. IMÁGENES UTILIZADAS PARA DEFINIR EL MATERIAL DE LAS ROCAS. A) COLOR, B) DESPLAZAMIENTO C) NORMAL Y D) RUGOSIDAD. 56
FIGURA 31. ROCAS GENERADAS MEDIANTE RUIDO PROCEDIMENTAL. DE IZQUIERDA A DERECHAS MÉTODOS: CLOUDS, VORONIO Y STUCCI. DE ARRIBA ABAJO: Z_SCALE=1, Z_SCALE=0.4, Y Z_SCALE=0.4 CON MATERIAL. ..57
FIGURA 32. VISTA LATERAL DE UNA CAPA DE DIQUE CON ROCAS GENERADAS MEDIANTE RUIDO PROCEDIMENTAL. 58
FIGURA 33. VISTA FRONTAL DE UNA CAPA DE DIQUE CON ROCAS GENERADAS MEDIANTE RUIDO PROCEDIMENTAL 58
FIGURA 34. MALLA EN FORMA DE CUBO. 59
FIGURA 35. OBJETOS OBTENIDOS MEDIANTE OPERACIONES LÓGICAS: A) INTERSECCIÓN, B) UNIÓN Y C) DIFERENCIA. 59
FIGURA 36. PIEZAS BASE PARA LA CREACIÓN DEL BLOQUE TIPO TETRÁPODO. 60
FIGURA 37. PIEZAS BASE QUE COMPONEN EL BLOQUE TIPO TETRÁPODO POSICIONADAS. 60
FIGURA 38. PROCESO DE UNIÓN REALIZADO POR PASOS. 61
FIGURA 39. BLOQUES ROMPEOLAS GENERADOS MEDIANTE EL CÓDIGO DESARROLLADO: A) COB, B) CUBO, C) AKMON TIPO CRUZ PLANA, D) TRIPOD, E) DOLOS, F) HEXALEG, G) AKMON TIPO ESTÁNDAR, H) TETRÁPODO CON RESOLUCIÓN 67 Y I) MITAD DE AKMON. 62

1. Introducción

1.1. Introducción al problema

La ingeniería hace tiempo que dejo el papel para el diseño y modelado por el Diseño Asistido por Computadora que permiten un desarrollo, creación, modificación, análisis y optimización de los diseños de forma más precisa y eficiente. En este proceso y, como es lógico, en primer lugar, se desarrollaron herramientas de modelado 2D (*AutoCAD*, *LibreCAD*., etc.). Más tarde aparecieron las herramientas de modelado 3D (*Solid Edge*, *CATIA*, etc.), aunque en la actualidad las herramientas suelen permitir el desarrollo en 2D y 3D indistintamente.

La Ingeniería de Caminos, Canales y Puertos no es ajena a este fenómeno, aunque si suele ir por detrás de otras ingenierías en la adopción de estas tecnologías. Así, a día de hoy, el modelado 3D no suele ser utilizado aún en los proyectos en los que la tecnología CAD es utilizada principalmente para la creación de planos, documentos indispensables en los proyectos. Esto no ocurre así en la arquitectura que ha sido capaz de adoptar las nuevas tecnologías de manera mucho más rápida y donde el uso de herramientas 3D es mucho más habitual, como se constata con la adopción del llamado Modelado de Información de Construcción (BIM) que incorpora al diseño 3D, el tratamiento de información.

Aunque de adopción tardía el diseño 3D, muestra ser de gran interés en la ingeniería de caminos permitiendo acercar un poco más el diseño realizado a la realidad. Esto es de gran importancia cuando el ingeniero debe trasladar los conceptos a un cliente que puede ser totalmente ajeno a la ingeniería y al que, por tanto, pueda serle difícil la comprensión de un plano bidimensional. Hoy en día incluso se pueden utilizar los diseños tridimensionales para su visualización a través de entornos de realidad aumentada, que permiten la inmersión tanto del ingeniero que está realizando el diseño como del cliente, lo que permite lograr mejores soluciones a los problemas que se está resolviendo, y una mayor participación del cliente en el desarrollo del proyecto llegando a una solución más personalizada del problema.

Quizás una de las causas por las que la ingeniería de caminos realiza una adopción tardía de las tecnologías frente a otras ingenierías o la arquitectura, es la complejidad de los proyectos que en ella se realizan, puesto que incorporan componentes de muy distinta índole, y las dimensiones de los proyectos son en general mucho mayores. No es lo mismo el proyecto de construcción de un puerto donde será necesario modelar el agua y su comportamiento en el tiempo, el terreno y su geología, el clima..., y ya dentro del aspecto constructivo, la combinación de piezas naturales como la roca, y artificiales como el hormigón armado, que un proyecto de diseño de una pieza industrial totalmente artificial, que será por lo general fácil de parametrizar y, por tanto, de sencillo diseño en una herramienta CAD. Y esto sin contar con las dimensiones de unos y otros proyectos lo que es ciertamente un problema en los proyectos de ingeniería de caminos ya que no sólo introduce una complejidad, sino un coste computacional muy grande que a veces ya de por sí puede limitar o impedir el diseño 3D.

Es muy importante que los programas, además de incorporar el diseño, tengan la capacidad de realizar otras labores propias de la ingeniería de caminos, como el cálculo estructural o hidráulico de la estructura diseñada, de forma directa con el mismo programa de diseño o indirecta por compatibilidad del programa.

Además, la capacidad de captar la atención de un cliente o simplemente ayudar al mismo a la comprensión del diseño como pueda realizar un arquitecto a través de renderizados, vídeos, etc. y que los diseños sean

visualmente agradables a la vista, es también un aspecto de interés, y puede ser un elemento diferenciador a la hora de participar en una licitación, por ejemplo, de urbanización en un entorno urbano.

Por último, hay que mencionar que los programas generalmente utilizados en el mundo de la ingeniería de caminos, que permiten desarrollo de planos o modelos 3D son ampliamente de pago, como las herramientas desarrolladas por *Autodesk* como *AutoCad* o específicamente para la ingeniería de caminos en *Civil 3D.*

Podrían resumirse, por tanto, los aspectos o requisitos positivos que un ingeniero de caminos, canales y puertos debe considerar en la elección de un programa de diseño en los siguientes:

- Modelado 2D y 3D integrados en un mismo programa.
- La capacidad de incorporar elementos propios de la ingeniería de caminos como puede ser un modelo del terreno a través de información geográfica.
- La posibilidad de realizar elementos irregulares e imperfectos para representar elementos constructivos naturales como una roca.
- Permitir realizar otras labores diferentes al propio diseño 3D de forma integrada o por compatibilidad directa con otros programas: cálculo de estructuras, cálculo hidráulico, mediciones, etc.
- Compatibilidad con otros programas a través de la posibilidad de importar y exportar distintas fuentes de datos.
- Permitir producir elementos visualmente atractivos, que sirvan para la captación de clientes y ayuden a una mejor comunicación y comprensión del proyecto para todas las partes como renderizados, vídeos o, por ejemplo, imágenes fotorrealistas.
- Que el programa sea gratuito.
- Capacidad de realizar modelos de grandes dimensiones sin un coste computacional elevado.
- Poseer un gran número de usuarios y documentación de calidad, actualizada, gratuita y accesible, permitiéndose así el aprendizaje autodidacta del programa.

Existen programas o conjuntos de programas que resuelven uno o más de estos aspectos, por ejemplo, para el diseño de una carretera, se puede por un lado utilizar *Civil 3D* y para su visualización 3D *InfraWorks*, pero el uso de varias herramientas sólo añade coste económico y complejidad, además del coste de aprendizaje de varias herramientas. Siguiendo con este mismo ejemplo, hay herramientas de trazado que permiten el desarrollo completo del mismo y su visualización 3D, como puede ser *CLIP*, sin embargo, la comunidad de este tipo de *software* es extremadamente reducida, su documentación es limitada y anticuada, por lo que suelen requerir de un coste económico añadido de asistencia técnica. Otro ejemplo podría ser el programa MIDAS que genera un modelo 3D de la estructura realizada y permite el cálculo estructural del mismo, sin embargo, este tipo de programas es muy especializado, y el diseño es muy limitado al estar orientado principalmente al cálculo de estructura.

Una cuestión aun no mencionada, y de gran importancia en este documento y en general para el futuro de cualquier ingeniero, es la programación, y en concreto la capacidad del programa utilizado de permitir realizar cambios sobre el mismo, o de desarrollar complementos que permitan ampliar las capacidades del programa o automatizar procesos repetitivos. Esto puede suponer que un programa que no incluya uno de los aspectos anteriormente mencionados, mediante el desarrollo de complementos adquiera la capacidad cumpliendo así con una necesidad más del ingeniero. Existe la posibilidad de programar mediante programas comerciales de amplio uso como *AutoCAD*, mediante *Autolisp*, aunque la realidad es que es un pseudo-lenguaje muy específico, con poca comunidad y documentación para su aprendizaje; otro ejemplo podría ser el uso de *Grasshopper 3D* un «lenguaje de programación» visual con un editor basado en nodos desarrollado para su uso en la herramienta de diseño *Rhinoceros 3D*. Otra herramienta interesante en este aspecto es *FreeCAD*, que como su propio nombre indica es gratuita, y permite el desarrollo de funcionalidades a través de *Python*. Poseyendo este último ejemplo una gran ventaja en el ámbito de la programación con respecto a los otros ejemplos de programación mencionados, puesto que Python es uno de los lenguajes de programación más utilizados en la actualidad, y no ha sido desarrollado exprofeso para

un programa de diseño, lo que permite acceder a un gran número de librerías o su propio desarrollo sin limitaciones de ningún tipo.

1.2. Sobre Blender

De la misma forma que la ingeniería de caminos ha adoptado de forma tardía ciertas tecnologías del mundo de la informática frente a otras ingenierías. Las ingenierías, en general, adoptan estas tecnologías con retraso frente a otros sectores, por ejemplo, el denominado Modelado de Información de Construcción (la primera herramienta BIM fue desarrollada en 1984), no es más que la incorporación de almacenamiento de datos en un elemento, el análisis de los mismos y su distribución, aspectos que cualquier persona con conocimientos de programación puede entender como básicos. De hecho, ya la ingeniería de caminos lleva utilizando una tecnología con algunas de estas cualidades desde hace mucho tiempo como son las herramientas SIG (la primera herramienta SIG fue desarrollada en 1962). Esto no deja de ser cierto en el diseño 3D, y en este caso probablemente por las inversiones que otros sectores realizan en programas dedicados al diseño 3D como el mundo de los videojuegos, la animación y la cinematografía, que añaden una serie de requisitos que a priori no son de importancia para un ingeniero, pero que permiten el desarrollo de diseños: fotorrealistas, extremadamente complejos, con la posibilidad de interactuar con los modelos realizados, etc.

Un programa de este estilo es *Blender*, que como en su propia página describe es un programa gratuito y de código abierto para el diseño 3D que «soporta el proceso completo del desarrollo 3D, modelado, *rigging*, animación, simulación, renderizado, composición y captura de movimientos e incluso la edición de vídeos», todo ello, en un entorno totalmente modular y programable mediante *Python*, con la capacidad de utilizar y desarrollar complementos (*add-ons*) tanto gratuitos como de pago. Permite también el esculpido, el dibujo 2D, la composición de texturas, generación de *uv maps*, manejo de iluminación, realizar simulaciones físicas (con gravedad) de fluidos, humo, partículas, etc…y de manera gratuita.

Blender soporta una amplia variedad de formatos, que incluyen los formatos generalmente más utilizados. Siendo por tanto compatible con otros programas, y permitiendo así que los resultados puedan ser distribuidos para su uso en otros programas. Así, se incluyen entre otros formatos, para las imágenes los formatos: .bmp, .png, .jpg, .tiff, etc.; para los vídeos formatos: .mp4, .avi, etc.; como formatos para el diseño tridimensionales: .dxf, .3ds, etc.; y para el diseño bidimensional: .svg, .gimp, etc. Siendo además posible desarrollar códigos que añadan la compatibilidad con nuevos formatos no soportados de forma directa o incluso utilizar complementos públicos desarrollados por otros usuarios para este fin.

Blender tiene además una comunidad de usuarios y documentación extraordinaria que permite el aprendizaje y resolución de dudas sin coste económico alguno a través de internet.

Por todo ello, *Blender*, aun sin ser un programa desarrollado para la ingeniería, parece ser una herramienta bastante interesante que aparentemente puede cumplir con los aspectos de interés anteriormente definidos para un ingeniero.

1.3. Objetivos

El objetivo de este trabajo es estudiar el uso de *Blender* como herramienta para la ingeniería de caminos canales y puertos, tratando de demostrar que cumple con los requisitos anteriormente definidos y que podrían ser de utilidad para el ingeniero en el desarrollo de sus proyectos. Como ya se ha mencionado el aspecto de la gratuidad ya está cumplido al ser *Blender* una herramienta distribuida bajo licencia GNU GPL. Según se indica en la propia página de *Blender* «*Get the world's best 3D CG technology in the hands of artists as free/open source software*» [1].

Al ser *Blender* una herramienta que no ha sido desarrollada exprofeso para la ingeniería de caminos, canales y puertos, es comprensible que no posea de manera directa herramientas específicas para esta profesión, por lo que a lo largo de este documento se comprobará la utilidad del programa a través del desarrollo de

herramientas que podrían ser de aplicación para esta rama de la ingeniería. Hay que aclarar que este documento no pretende ser un curso del uso de *Blender* a través de su interfaz gráfica, de diseño o de programación, sino la demostración de la viabilidad de uso de esta herramienta para la ingeniería de caminos, canales y puertos a través de aplicaciones de interés para un ingeniero de caminos.

En concreto, en el documento se busca demostrar la viabilidad de uso del programa para la ingeniería de caminos a través del desarrollo de las siguientes aplicaciones:

- Caminos: realizar modelos tridimensionales del terreno y extraer perfiles longitudinales de carreteras.
- Canales: crear tuberías normalizadas para distintos materiales.
- Puertos: crear rocas naturales y bloques de hormigón artificial para diques.

2. Blender

2.1. Introducción

Blender es una herramienta gratuita desarrollada para el diseño 3D, dirigida principalmente al sector de las animaciones y los videojuegos.

El modelado se realiza a través de los siguientes tipos de objeto:

- Mallas: superficies definidas mediante vértices.
- Curvas: polilíneas, Bézier y NURBS, que pueden ser 2D o 3D.
- Superficies: definidos mediante puntos de control como las curvas tipo NURBS.
- *Metaball*: superficies definidas mediante fórmulas matemáticas, que son capaces de interactuar entre sí cuando están cerca.
- Texto.
- Volúmenes: elementos utilizados para representar objetos volumétricos dispersos *OpenVDB*, como una nube, agua, una explosión, etc.
- Elementos vacíos.
- Enrejados.
- Imágenes.
- *Amatures*: permiten generar o restringir movimientos en los objetos.
- Luces.
- Cámaras.
- Campos de fuerza.

Los tipos de objetos de utilidad directa para la ingeniería son las mallas y curvas, con las que se realiza el modelado de forma diferente a las herramientas CAD tradicionales, a lo largo del documento se analizará el funcionamiento de estas dos tipologías.

Algunas características de *Blender* son:

- Permite la programación a través de Python y la creación y uso de complementos (*add-ons*).
- Incluye herramientas de audio, vídeo y tratamiento y creación de imágenes y texturas.
- Para el renderizado posee varios motores gráficos: *Evee*, *WorkBench* y *Cycles*.
- Permite realizar simulaciones físicas, con sólidos rígidos, sólidos deformables, fluidos y humo, permitiendo introducir conceptos físicos como la gravedad.
- Es totalmente modular, pudiéndose cambiar la práctica totalidad del entorno de trabajo, personalizándolo a las necesidades y gustos del usuario.

Por lo general el uso de la interfaz gráfica se lleva a cabo mediante atajos de teclado, esto fomenta el uso rápido del programa con el tiempo, sin embargo, esto mismo complica el aprendizaje. En este sentido es importante mencionar que el programa no es sencillo de utilizar, debido a la gran cantidad y variedad de herramientas que posee y a la tipología de objetos utilizada en el diseño distinta de las líneas, sombreados y sólidos utilizados en otros programas de diseño para la ingeniería que son más intuitivos de usar. Por todo esto, la curva de aprendizaje del programa es larga, si bien con el tiempo y gracias al fomento del uso de los atajos, se puede adquirir una gran velocidad en el modelado. Además de los atajos, los menús y botones

que contiene el programa, es posible acceder a las herramientas mediante una línea de comandos similar a la existente en programas como *Autocad*.

2.2. Entorno de trabajo

El entorno de trabajo de *Blender* se compone de 3 partes principales, el menú, los diseños (*layouts*) y las ventanas de edición. Además, cada ventana, de entre todas las disponibles, cuenta con sus propias herramientas, menús deslizables, paneles emergentes, botones, etc, ver Figura 1. Las ventanas incluyen también distintos modos de trabajo que cambian casi por completo el funcionamiento y las utilidades de la misma, esto se puede ver por ejemplo con la ventana *3D Viewport*, que es una de las ventanas de edición de los modelos y que contiene los modos: *Object Mode*, *Edit Mode*, *Sculpt mode*, *Vertex Paint*, *Weight Paint* y *Texture Paint*. También se incluyen en la ventana *3D Viewport*, que puede ser considerada como la ventana principal de *Blender*, la posibilidad de decidir cómo se desea ver el entorno: vista en perspectiva u ortogonal; visualización de los objetos en modo: alambre, sólido, vista previa y renderizado; y múltiples opciones más.

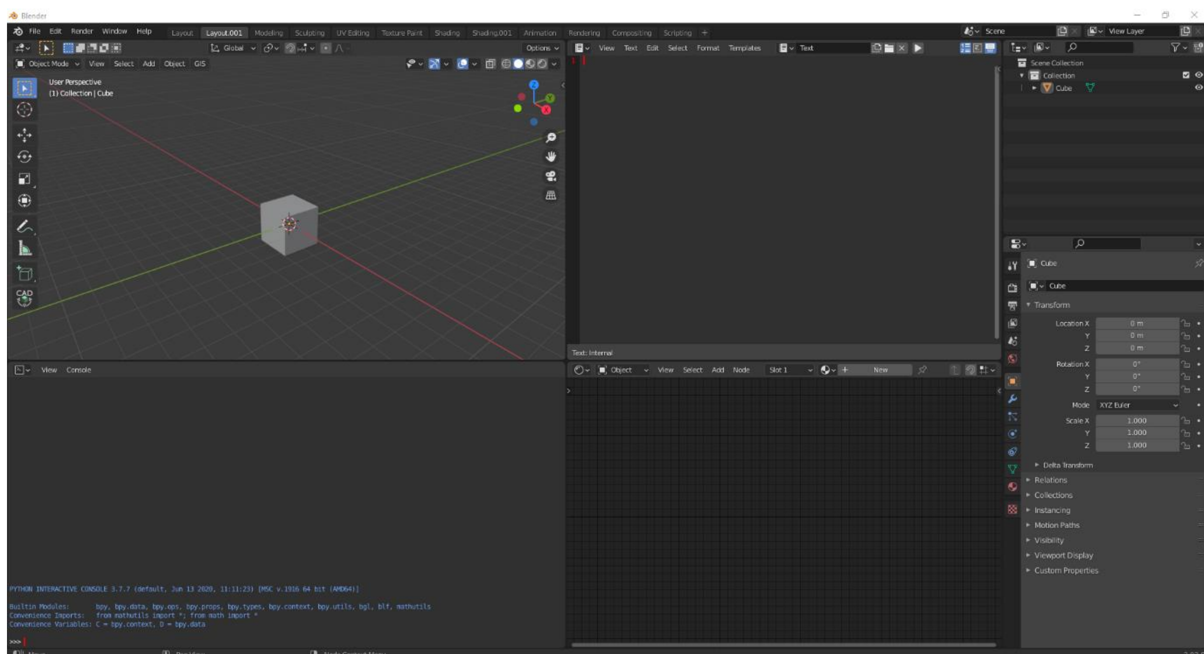


Figura 1. Layout utilizado en Blender, se aprecian en el 6 de las 21 ventanas existentes, así como algunas herramientas.

A continuación, se incluye un listado las ventanas de edición existentes en la versión 2.92 de *Blender* utilizada en el desarrollo de este trabajo:

- De tipo general:
    - *3D Viewport*: es la ventana utilizada para la edición del modelo siendo usada para múltiples propósitos como el modelado, el pintado o la animación entre otros.

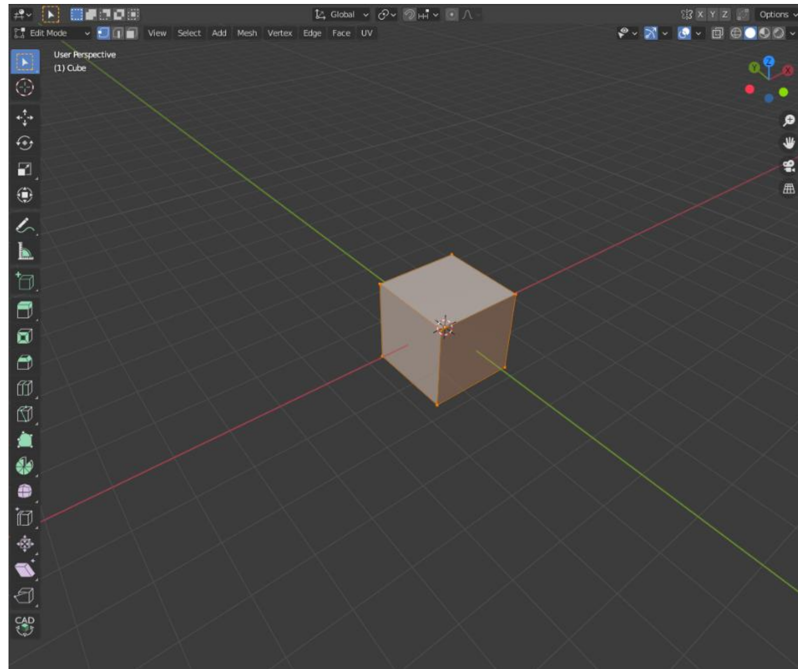


Figura 2. Ventana de edición 3D Viewport en modo Edit mode.

- Image Editor: utilizada para la edición de elementos tipo imagen.
- *UV Editor*: se utiliza para la edición de *uv maps*.
- *Shader Editor*: utilizada para la edición de materiales que dan color a los elementos. Realizando la edición a través de un «lenguaje de programación» visual basado en nodos y uniones.
- *Compositor*: para la composición de imágenes o películas mediante un sistema de nodos y uniones.
- *Geometry Node Editor*: para la edición de geometrías mediante nodos y uniones en el modificador (*modifier*) *Geometry Node.*
- *Texture Nodes*: para la generación de texturas mediante un sistema de nodos y uniones.
- *Video Sequencer*: utilizada para la edición de vídeos.
- *Movie Clip Editor*.

– Para la animación:
  - *Dope Sheet*.
  - *Timeline*.
  - *Graph Editor*.
  - *Drivers Editor*.
  - *Nonlinear Animation*.

– Para la programación:
  - *Text Editor*: editor de texto para la programación y ejecución de códigos.
  - *Python Console*: consola de Python que permite el acceso directo a la *API* de *Blender [2]*.
  - *Info Editor*: es un historial de las acciones que se han realizado en el programa, así como de los mensajes de advertencia y error que puedan surgir.

– Para el manejo de datos:
  - *Outliner*: ventana que permite visualizar los elementos del modelo agrupados en conjuntos llamados escenas y colecciones, en un sistema de carpetas en cascada arborescente. Permite activar o desactivar la visualización de los elementos del modelo, el cambio de nombre su selección o eliminación entre otros.

- *Properties*: utilizada para la edición de variables de la escena o el objeto activo. Posee múltiples categorías, que son cada uno de los iconos que se ven en la parte izquierda de la siguiente figura, y que incluyen infinidad de parámetros y herramientas.

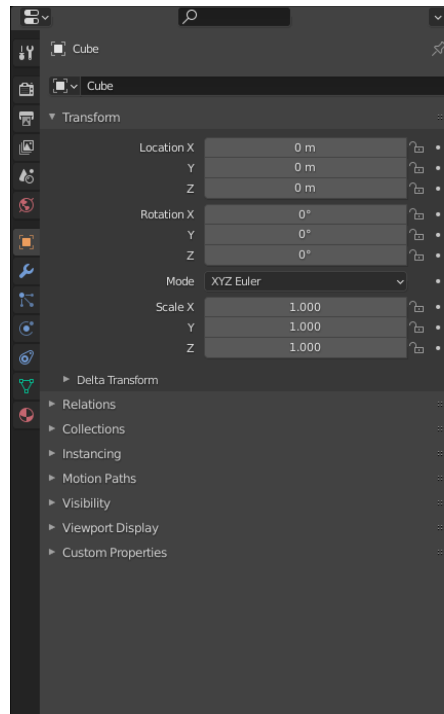


Figura 3. Ventana de edición Properties.

- *File Browser*: buscador de archivo similar al explorador de *Windows*.
- *Preferences*: permite cambiar las preferencias de *Blender*, esta misma ventana de edición es accesible a través del menú principal del programa.

Debe mencionarse que *Blender* permite una gran personalización del entorno de trabajo, desde los colores utilizados en él mismo, hasta la elección de los atajos a utilizar para cada acción que en él se realiza. Además de esto, los distintos *layouts*, pueden ser modificados para cubrir las necesidades del usuario, e incluso se pueden generar nuevos *layouts* y guardarlos.

Por último, hay que mencionar que *Blender* incluye algunas funcionalidades del tipo *snap on point*, que permiten el encaje por vértices, aristas o caras, así como otros elementos del espacio de modelo. Aunque la utilidad es bastante limitada entre otras cuestiones porque no permite realizar traslaciones, escalados o rotaciones a la vez se realizan movimientos del espacio modelo, como si permiten otras herramientas de diseño. Sin embargo, y para que sirva de introducción a las capacidades de personalización infinitas que tiene el programa, existe un complemento gratuito denominado *CAD Transform* que permite realizar transformaciones en los objetos a la vez que se realizan movimientos en el espacio modelo, incluyendo éste un número de utilidades de tipo *snap on point* que para hacerse una idea de las posibilidades que permite posee 31 atajos, que se pueden además concatenar para obtener resultados distintos. Permite además dar marcha atrás en la combinación de atajos realizada sin perderla por completo, y utilizar variedad de unidades y sistemas de medida, así como operaciones matemáticas. Todo esto hace que algo que a priori era un problema para el mundo de la ingeniería por la falta de precisión que la ausencia de capacidades de tipo *snap on point* produce a la hora de realizar el diseño, con la incorporación de un complemento, se convierta en un entorno de trabajo con más capacidades en este sentido que cualquier otro programa de diseño. Además, al ser el complemento gratuito y de código libre, permite al usuario realizar cambios en el código y añadir nuevas funcionalidades si es necesario.

2.3. Python

Como se ha mencionado *Blender*, permite la interacción mediante consola y ejecución de códigos mediante el editor de textos incorporado a través de lenguaje de programación *Python*. Además, es posible la creación de complementos para integrar herramientas en los distintos entornos de trabajo que posee el programa.

Para facilitar, el desarrollo contiene herramientas dedicadas al mismo como son las ya mencionadas ventanas de edición *Text Editor*, *Python Console* y *Info Editor*. Permite visualizar las líneas de código que se ejecutan al usar una herramienta colocando el ratón sobre la misma si se tienen activadas las opciones dedicadas al desarrollo que se encuentran en las preferencias del menú edición. Además, permite copiar dichas líneas para su uso pulsando con el botón derecho del ratón. Se puede acceder al código fuente de la herramienta si es de código libre pulsando con el botón derecho del ratón. También se puede acceder a plantillas de código para los distintos tipos de complementos a través del editor de texto. Por último, incluye la capacidad de visualizar los índices de los vértices, aristas y caras de las mallas, su orientación, longitud área, etc. Todo esto facilita enormemente el desarrollo de herramientas en el programa.

Es muy importante en cuanto al aprendizaje de un programa que este posea una comunidad grande y activa, así como una documentación accesible, actualizada y completa. Esto facilita el aprendizaje al permitir realizar consultas rápidas en la documentación, o mediante foros o vídeos. Esta cuestión es de gran importancia en el mundo de la programación, teniendo *Blender* una comunidad lo suficientemente grande como para resolver la mayoría de las dudas que puedan surgir en el aprendizaje a través de internet, aspecto muy positivo a la hora de escoger un programa para trabajar.

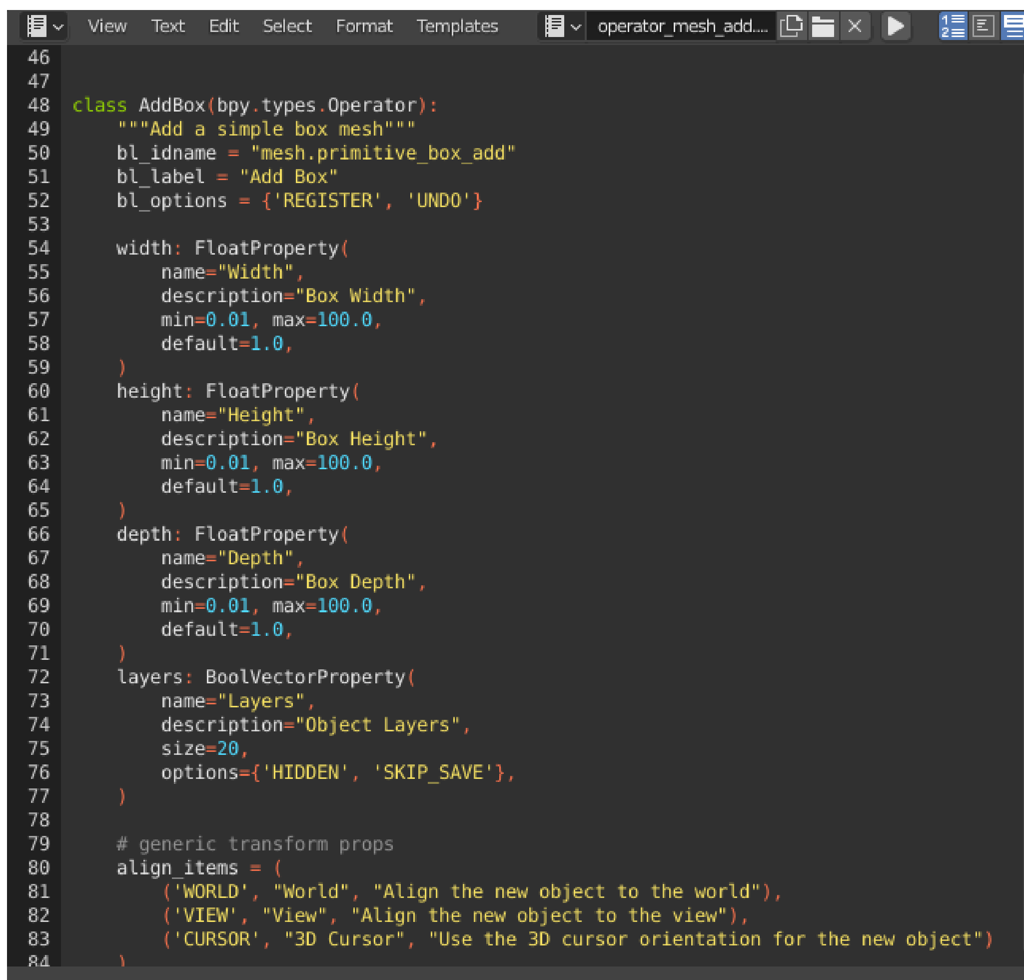


Figura 4. Editor de texto incluido en Blender con una plantilla para realizar complementos con los que crear objetos.

2.4.Herramientas desarrolladas

A lo largo del documento se desarrollan cinco herramientas relacionadas con la ingeniería de caminos, canales y puertos, que, en concreto están ordenadas por su relación con el ámbito de los caminos, los canales o los puertos, como guiño hacia el nombre de la profesión. Cada problema en cierta medida pretende demostrar la capacidad que un ingeniero a través de *Blender* tiene para cumplir con alguno de los requisitos definidos en el primer apartado y que pueden ser de utilidad para el desarrollo de la profesión, y demostrar así el cumplimiento de los objetivos de este trabajo.

Para cada herramienta se realizará una exposición del desarrollo que hay que realizar para la resolución del problema pertinente, y una breve reseña de los resultados obtenidos mediante la misma.

Para no complicar de forma excesiva el documento y facilitar su lectura a cualquier tipo de ingeniero sean cuales sean sus conocimientos de programación, se tratará en la medida de lo posible de no complicar en exceso la explicación en este aspecto. Motivo por el cual los códigos adjuntos en el correspondiente anexo no deben ser tomados como modelo a seguir en la programación. Es por ello por lo que se ha optado por no desarrollar complementos capaces de ser distribuidos para su uso mediante instalación, que complicarían y alargarían en exceso este documento. Si bien, en una de las herramientas desarrolladas sí que se explicará el desarrollo e instalación de un tipo de complemento de los muchos posibles en *Blender*, para así completar el documento, con el desarrollo completo de una idea que pretende resolver un problema ingenieril. Es posible tomar los códigos desarrollados y ejecutarlos para replicar los resultados sin ningún problema o realizar los cambios pertinentes en los mismos para el desarrollo de nuevos complementos con estos. Su uso o instalación en el caso del complemento desarrollado se explica al final de cada apartado.

Por último, hay que mencionar que los códigos han sido desarrollados y comprobados en la versión 2.92 de *Blender* y que no se ha implementado ninguna medida de comprobación de compatibilidad de los módulos utilizados o de *Blender*, ni de validez de los archivos utilizados, con el fin mencionado anteriormente de no complicar en exceso los códigos con, en este caso, cuestiones que no forman parte del objetivo de este documento.

3. Caminos

3.1. Modelo digital del terreno

Para cualquier proyecto de ingeniería de caminos, canales y puertos, la base es el terreno, tanto si el proyecto es en tierra firme (topografía) como si se desarrolla en el agua (batimetría). Esto no es diferente en los proyectos de carreteras, donde de hecho el correcto análisis del terreno es indispensable para la obtención de la solución óptima del problema, siendo por tanto este la base de todo proyecto de construcción de carreteras. Por ello, como primera herramienta se ha elegido el modelado tridimensional del terreno, que serviría de punto de partida para la generación de una herramienta para el trazado de carreteras como las mencionadas *Civil 3D* o *CLIP*, que permita la visualización tridimensional del modelo con toda la realidad que el ingeniero desee, pudiendo competir sin problemas por ejemplo con el modelo tridimensional que se puede generar a través de *InfraWorks*. Permitiendo también utilizar la herramienta como complemento de herramientas *SIG* como *QGIS*.

Al ser la primera herramienta explicada en el documento, esta será utilizada como toma de contacto para el lector hacia *Blender* y en concreto para la programación a través de la *API* de *Blender*. Por ello este apartado ha sido dividido en cuatro subapartados, en el primero se resolverá el problema de forma directa con un uso muy limitado de *Blender* que cualquier usuario podría desarrollar sin prácticamente conocimientos del programa. En el segundo subapartado, se profundizará hacia una solución que utiliza las herramientas de *Blender* (m*odifiers*) y que como se comprobará aumentan la versatilidad y eficiencia del programa. En el tercer subapartado, se añadirán al código aspectos relacionados con la visualización del terreno generado, permitiendo incorporar rampas de colores u ortofotos sobre el mismo. Y, por último, se realizará una breve descripción de un complemento ya existente que permite generar modelos tridimensionales del terreno de múltiples formas.

A la hora de utilizar el código hay que tener en cuenta que los códigos de las tres partes desarrolladas en este apartado han sido integrados para formar un único código, más versátil, en el que el usuario decide la forma de desarrollar el modelo a través del método que considere más adecuado en su problema.

3.1.1. Triangulación de Delaunay

Desarrollo

Para la definición de un modelo digital se requiere de la altimetría del terreno como dato de partida. Habitualmente esta se puede adquirir en servidores de datos públicos como el centro de descargas del IGN en España, aunque también es posible la obtención propia de estos datos mediante levantamiento topográfico con estación total o drones. Normalmente los datos suelen venir en archivos con formato .asc o .tiff (también .tif), es decir, archivos tipo ráster, en los que queda representado el terreno como una imagen o matriz en la que cada píxel o celda representa un área del terreno. En la siguiente figura se puede ver la estructura junto con una representación gráfica del contenido de un archivo .asc, siendo tomado este formato por sencillez y al ser este un archivo de tipo ASCII cuyo contenido es visible a través de un simple editor de texto.

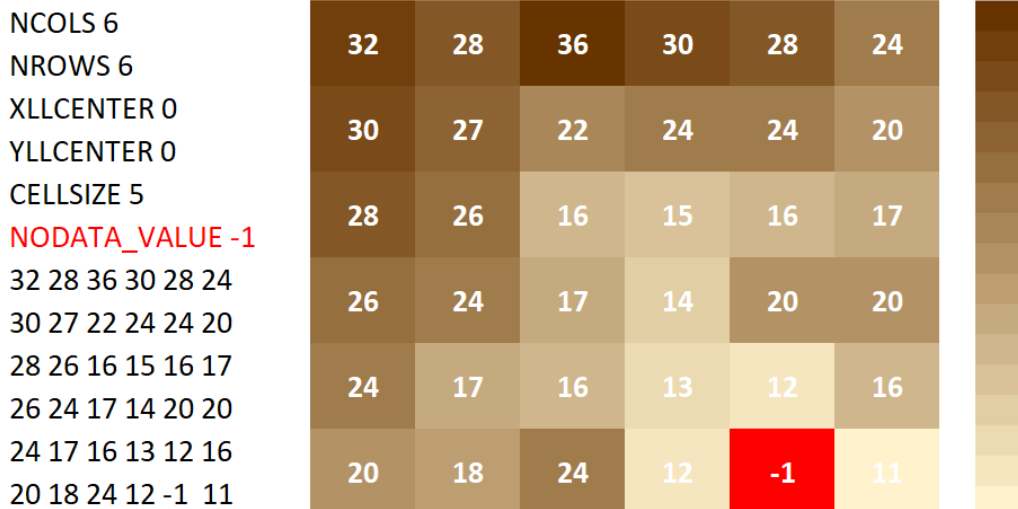
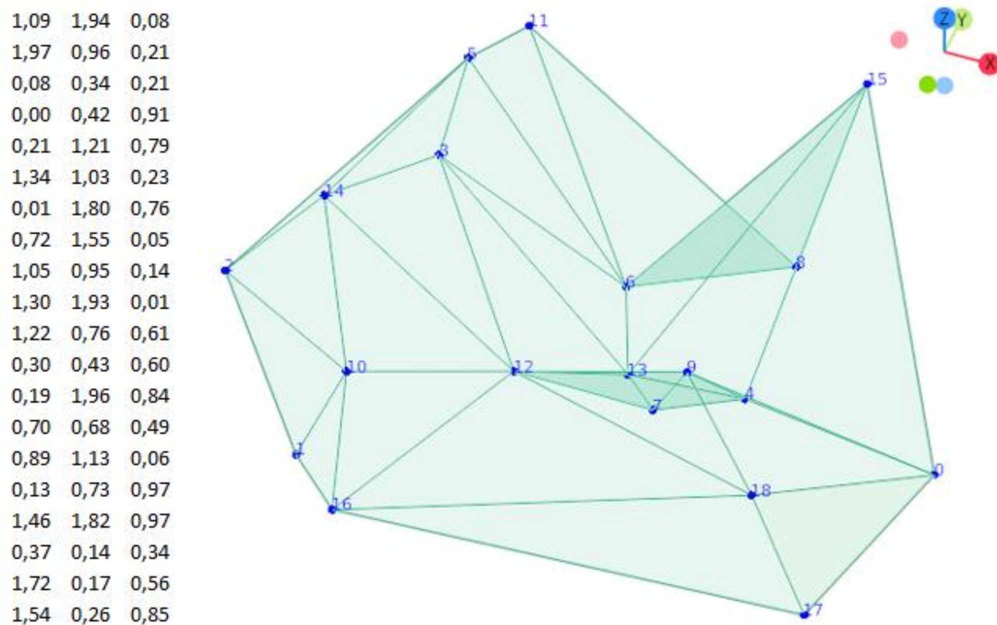


Figura 5. Estructura y representación gráfica de un archivo .asc.

Como se puede ver este tipo de archivo no contiene sistema de coordenadas asociado, siendo el único indicativo de su localización el origen establecido en la cabecera, y la posición del mismo en la celda que podrá ser en la esquina inferior derecha de la celda inferior derecha (XLLCORNER/YLLCORNER) o en el centro de la misma (XLLCENTER/YLLCENTER). Aunque pueden utilizarse archivos auxiliares para transportar esta información como los archivos: .prj, .qpj o .tfw. Por su parte los archivos .tiff que contienen altimetría y que por tanto han sido generados específicamente para aplicaciones de información geográfica e ingeniería, suelen tener asociados el sistema de coordenadas en la cabecera de los mismos, si bien este tipo de archivo no puede ser abierto por un editor de texto para ver su estructura al ser estos archivos de tipo binario.

En la actualidad se impone cada vez más la toma de datos a través de *LIDAR*, que permite la obtención de puntos con mayor precisión tanto por la densidad de datos obtenida por área, como por la propia precisión en la toma del dato que permite la tecnología. Sin embargo, los datos obtenidos por este sistema suelen contener lo que para la aplicación que aquí se desarrolla se considera como ruido, generalmente datos tomados erróneamente por la presencia de vegetación. Aunque es posible realizar procesos de limpiado de este ruido a través de algoritmos accesibles mediante herramientas *SIG*. El formato de distribución general es el .las o el comprimido de este .laz, por ser los archivos de gran tamaño ya que por cada punto pueden almacenarse hasta 26 valores en la última versión del archivo. Siendo la herramienta utilizada para el tratamiento de estos archivos y que permite entre otras cosas su transformación a otros formatos como el .tiff o la eliminación del mencionado ruido *LASTools [3]*, que está integrada en herramientas SIG como QGIS.

Los archivos provenientes del LIDAR, tienen una estructura diferente puesto que en este caso el formato es de tipo nube de puntos, por lo que la estructura es algo más compleja que con los archivos tipo ráster, a la hora de definir el modelo. Otro archivo muy utilizado con formato nube de puntos es el archivo .xyz, que de nuevo por su sencillez, se representa en la Figura 6.



1,09	1,94	0,08
1,97	0,96	0,21
0,08	0,34	0,21
0,00	0,42	0,91
0,21	1,21	0,79
1,34	1,03	0,23
0,01	1,80	0,76
0,72	1,55	0,05
1,05	0,95	0,14
1,30	1,93	0,01
1,22	0,76	0,61
0,30	0,43	0,60
0,19	1,96	0,84
0,70	0,68	0,49
0,89	1,13	0,06
0,13	0,73	0,97
1,46	1,82	0,97
0,37	0,14	0,34
1,72	0,17	0,56
1,54	0,26	0,85

Figura 6. Estructura y representación gráfica de un archivo .xyz. Las columnas representan de izquierda a derecha las coordenadas X, Y y Z de los puntos.

La metodología elegida para generar el modelo de elevaciones en este apartado es a través de una triangulación de los valores de partida. Esto permite indistintamente tomar como datos un archivo tipo ráster o nube de puntos. Como se verá a continuación, el código desarrollado tiene la capacidad de utilizar archivos tipo .asc, .tiff y .xyz, para así demostrar las capacidad que tiene el ingeniero de tomar cualquier tipo de archivo como fuente de datos en el uso de *Blender*. En concreto el método de triangulación elegido es la triangulación de Delaunay y ha sido elegido simplemente para demostrar que no es necesario realizar todo el desarrollo del programa, sino que se pueden tomar librerías externas a la *API* de *Blender* y utilizarlas en los programas desarrollados, reduciendo la necesidad de desarrollar algoritmos complejos. A modo de resumen una triangulación consiste en la generación de un conjunto de triángulos a partir de un conjunto de puntos de manera que todo punto utilizado sea un vértice de un triángulo (ver **Figura 6**). El resultado de la triangulación es el conjunto de triángulos dados por el índice que representa a cada punto en el conjunto. De esta forma se tiene por un lado una lista de puntos con valores (x, y, z) y una lista de triángulos con valores (índice del vértice 1, índice del vértice 2, índice del vértice 3). Que es concretamente lo que *Blender* necesita para generar una malla, es decir, que sólo hay que generar una lista de vértices de la malla (los puntos) y una lista de caras de la malla (los triángulos).

En resumen, para la generación de una modelo tridimensional del terreno es tan solo necesario un conjunto de puntos que serán los vértices de la malla que representa al terreno y conjunto de triángulos que serán las caras de la misma. Por tanto, la complejidad de esta aplicación es la lectura de los archivos al utilizarse en este caso una librería externa para generar la triangulación y en concreto el módulo *spatial de* la librería *scipy* [4]. Las estructuras de los archivos .asc y .xyz pueden verse en la **Figura 5** y la **Figura 6** y su lectura es sencilla (ver 527 y 583 del anexo I), en el caso de los archivos .tiff al ser estos de gran complejidad se ha optado por usar también una librería externa denominada *Pillow PIL[5]* (ver líneas 435 del anexo I).

Una vez se tienen los vértices y las caras sólo hay que generar una malla (*mesh*) con *Blender*, para lo cual se ejecuta una línea de código como la que sigue:

```
malla = bpy.data.meshes.new(
  nombre_de_la_malla)  # Se crea una nueva malla.
mesh.from_pydata(
  vértices, [], caras) # Se define la geometría de la malla.
```

esta malla es añadida a un objeto (*object*) de Blender, que pertenecerá a su vez a una colección (*collection*) que no es más que una agrupación de objetos como podría ser un directorio de archivos.

```
objeto = bpy.data.objects.new(
  nombre_de_la_malla, malla) # Se crea un nuevo objeto y se le asigna la malla
                             # creada.
colleccion = bpy.data.collections.new(
  nombre_de_la_coleccion)    # Se crea una nueva colección.
bpy.context.scene.collection.children.link(
  coleccion)                 # Se añade la colección a la escena y contexto de
                             # Blender.
colleción.objects.link(
  objeto)                    # Se añade el objeto a la colección creada.
```

estas líneas serán habituales a lo largo de este trabajo, y una construcción similar será utilizada en la definición de elementos distintos a las mallas o los objetos.

En el código se han desarrollado, entre otras, la capacidad de escoger el punto en el que el modelo de datos aparece, bien el centro del sistema de coordenadas de *Blender* o bien la posición georreferenciada del mismo. Una cuestión de gran importancia cuando se trabaja con elementos de grandes o pequeñas dimensiones en *Blender*, es que con el objetivo de que el manejo de la aplicación sea lo más fluido posible y que por tanto el programa no quede bloqueado por ser bajas las capacidades del ordenador respecto al modelo que se está desarrollando, *Blender* limita el espacio visible a través de los parámetros *clip_start* y *clip_end* de la pestaña *view* (pulsar letra N para verla) dentro de la ventana *3D Viewport*. Esta es una de las muchas formas en las que *Blender* reduce el coste computacional en el desarrollo de los modelos, permitiendo así trabajar con modelos que en otras aplicaciones no serían posible.

Resultados y uso de la herramienta creada

Para utilizar el código terrain creator desarrollado basta con incluirlo en cualquier ventana de tipo *Text Editor* con una de las siguientes líneas al final de este en función del tipo de dato de entrada.

```
# Si el archivo de entrada es un archivo .tiff.
tiff_dem = Tiff(filepath=ruta_del_archivo, name=nombre_del_dem_resultante)
tiff_dem.blendify()

# Si el archivo de entrada es un archivo .asc.
asc = Asc(filepath=ruta_del_archivo, name=nombre_del_dem_resultante)
asc.blendify()

# Si el archivo de entrada es un archivo .xyz.
xyz_dem = Xyz(filepath=ruta_del_archivo, name=nombre_del_dem_resultante)
xyz_dem.blendify()
```

Simplemente se genera un objeto de las clases creadas *Tiff*, *Asc* o *Xyz* que toman la información de los archivos y realizan la transformación pertinente a vértices para la malla, posteriormente se llama al método *blendify,* que en este caso realizará la malla por triangulación de Delaunay. El resultado del proceso puede verse en la siguiente figura:

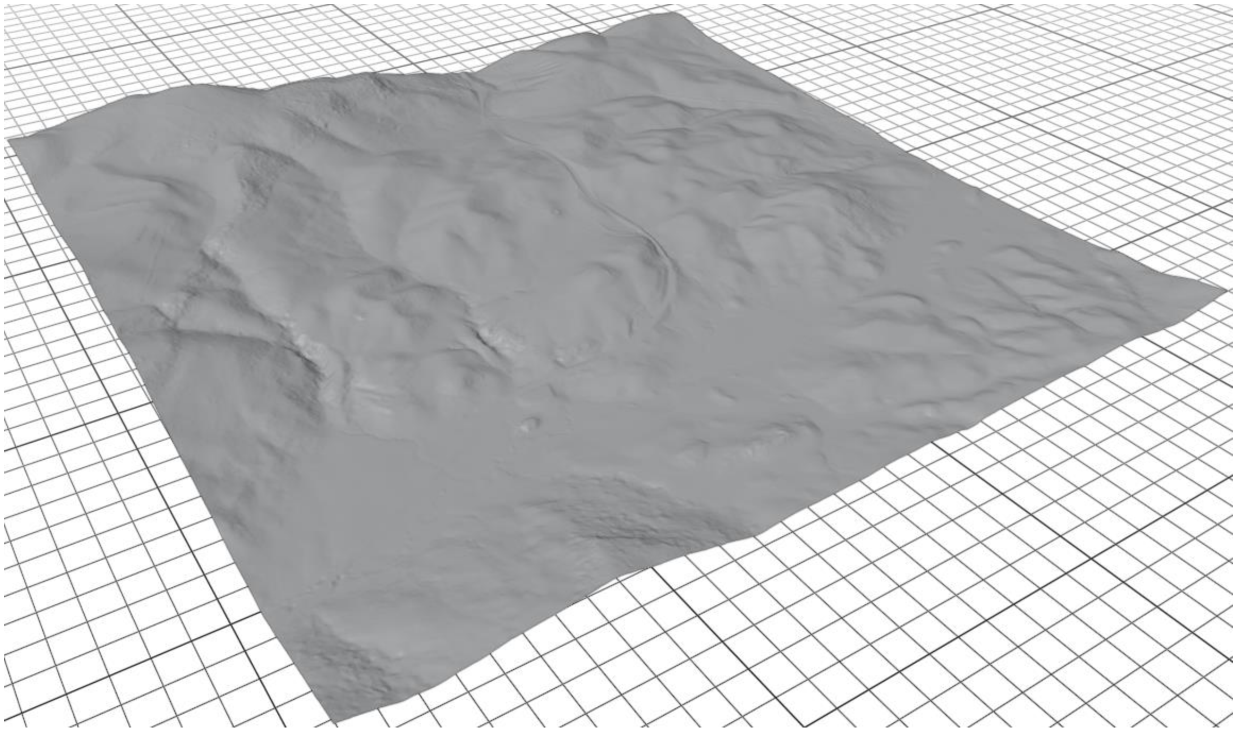


Figura 7. Modelo tridimensional del terreno generado a partir de una triangulación de Delaunay con Blender.

Figura 8. Detalle del modelo tridimensional del terreno generado a partir de una triangulación de Delaunay con Blender, donde se puede ver la malla de triángulos.

Como se puede ver la precisión del modelo que se puede generar únicamente depende de la precisión del dato de partida y de las capacidades del ordenador utilizado, si bien *Blender* ayuda en lo posible para que el ordenador no sea el problema.

3.1.2. Modifiers

Desarrollo

En la herramienta anterior se hizo un uso muy limitado de la API de *Blender*, en este apartado sin embargo se introducen nuevos conceptos que como se verá ayudan también a esa capacidad que tiene Blender de realizar el desarrollo de los modelos con un consumo reducido de recursos computacionales. Para ello se utilizan los denominados modificadores o *modifiers* en inglés que permiten realizar cambios sobre una malla ya existente. En este caso por tanto se requiere además de un dato inicial de altimetría en formato imagen

para esta herramienta (se usarán archivos .tiff), de una malla ya creada. Y la malla más simple capaz de representar la forma de una imagen es un rectángulo cuyos vértices son los límites máximos y mínimos de la imagen o en programación *minimun bounding box*. El código por tanto debe leer la imagen y extraer estos límites a partir de los cuales se crea una malla con vértices y una cara como se expuso en el apartado anterior.

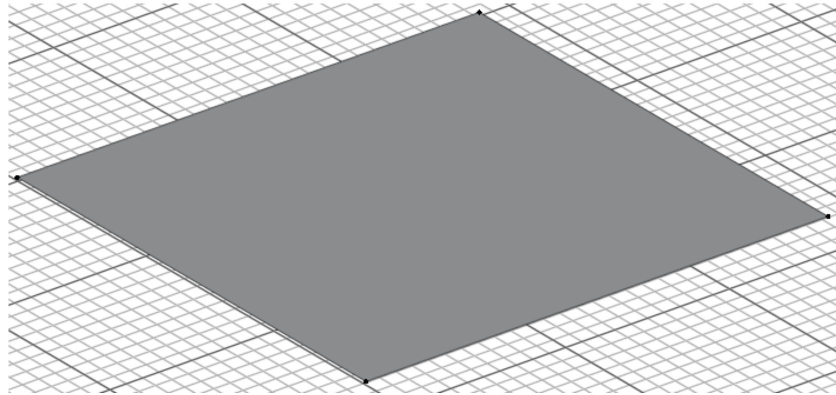


Figura 9. Malla base para el modelo tridimensional del terreno generado con Blender.

La imagen ahora es utilizada mediante un modificador denominado *Displace* que desplaza los vértices de la malla basándose en los valores de una textura, la imagen. Sin embargo, la malla actual sólo tiene cuatro vértices por lo que el resultado será de poca utilidad. Para incrementar la definición del modelo se usa otro modificador denominado *Subdivision Surface* que divide las caras (en principio una) de la malla en partes más pequeñas, incrementando así la precisión del modificador anterior. Los modificadores por tanto se pueden aplicar uno sobre otro, de hecho, en el código desarrollado se realizan dos modificadores tipo *Subdivision Surface* y uno de tipo *Displace* para poder así alcanzar el tamaño de área que representan los píxeles de las imágenes utilizadas (es necesario utilizar dos *Subdivision Surface* porque las subdivisiones se limitan a 6 por cada modificador, aunque esto sólo será de utilidad si la precisión de la imagen es alta).

Por tanto, se necesita añadir una imagen a *Blender,* y para ello se recurre a las siguientes líneas de código:

```
imagen = bpy.data.images.load(
  ruta_del_archivo)                                # Se carga la imagen en el
                                                   # programa.
imagen.colorspace_settings.name = 'Non-Color'      # Se define que se utilizarán
                                                   # los datos de cada píxel con
                                                   # fines distintos al de dar
                                                   # color.
```

Sin embargo, esta imagen aún no representa una textura en *Blender* sino un dato, por lo que hay que generar la textura de forma similar a como se creó un objeto en el apartado anterior.

```
textura = bpy.data.textures.new(
nombre_de_la_textura, type='IMAGE') # Se crea la textura,
textura.image = imagen              # Se define la textura a partir de la
                                    # imagen.
```

Por último, se crean los modificadores definidos sobre el objeto:

```
subivision_surface_modifier = objeto.modifiers.new(
```

```
    nombre_del_modificador, type='SUBSURF')           # Se crea el modificador tipo
                                                      # Subdivision Surface para
                                                      # el objeto.
  subivision_surface_mod.subdivision_type = 'SIMPLE'  # Tipo de subdivisión
  subivision_surface_mod.levels = 6                   # Número de divisiones a
                                                      # realizar.
                                                      # 3D Viewport.
  subivision_surface_mod.render_levels = 6            # Número de divisiones a
                                                      # realizar para el
                                                      # renderizado.

  displace_modifier = obj.modifiers.new(
    nombre_del_modificador, type='DISPLACE')          # Se crea el modificador tipo
                                                      # Displace para el objeto.
  displace_modifier.texture = textura                 # Se añade la textura cuyos
                                                      # datos serán tomados para
                                                      # producir los
                                                      # desplazamientos de los
                                                      # vértices.
```

mediante los atributos *levels* y *render_levels*, se controla la precisión del modelo en el desarrollo y su renderizado y por tanto el consumo que se hace de los recursos del ordenador.

Resultados y uso de la herramienta creada

Para el uso de esta herramienta, se toma el código terrain_creator como en el apartado anterior y se introduce en una ventana tipo *Text Editor* añadiendo al final del mismo en este caso las líneas siguientes:

```
  tiff_dem = Tiff(
    filepath=ruta_al_archivo, mode='MODIFIER', name=nombre_del_dem_resultante)
  tiff_dem.blendify()
```

como se puede ver el único cambio realizado respecto al modelo anterior es el argumento *mode*, que anteriormente estaba establecido por defecto con valor *'DELAUNAY'*.

El resultado obtenido puede verse en las siguientes figuras para varios niveles de división sobre la superficie. Se puede comprobar que en este caso las caras de la malla no son triángulos, dado que la división realizada por el modificador consiste en una división por la mitad de cada arista.

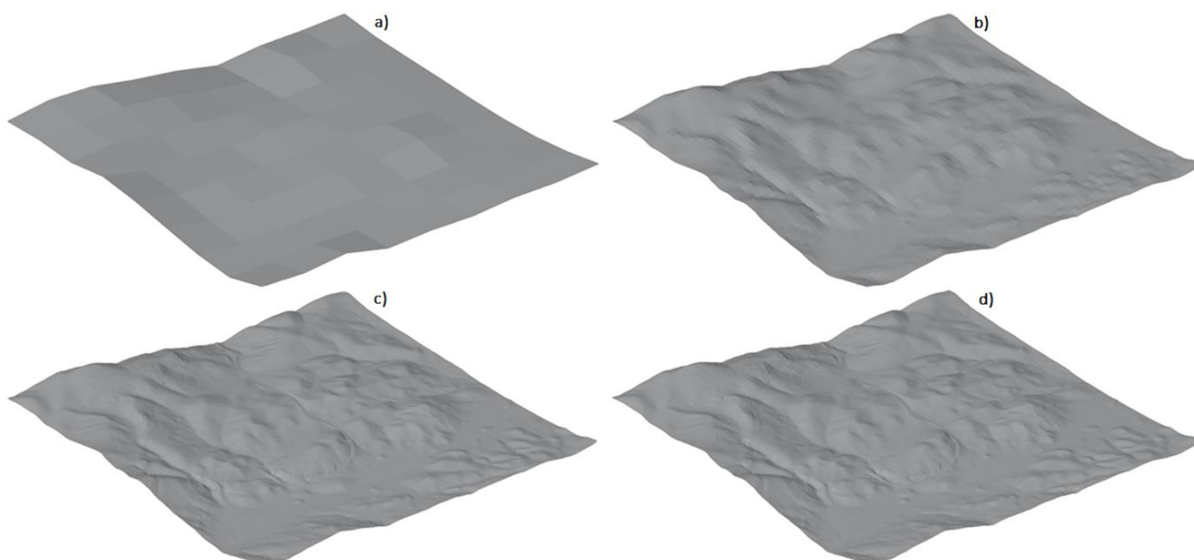


Figura 10. Modelo tridimensional del terreno generado a partir de modificadores con Blender. a) 3 divisiones, b) 6 divisiones, c) 9 divisiones y d) 11 divisiones.

3.1.3. Añadir materiales al modelo del terreno

Desarrollo

A diferencia de la herramienta anterior el objetivo de la textura generada en este caso (material) si es el de dar color a la malla. En este apartado, se verá cómo crear un material que aporte color a través de una rampa de colores en función de altimetría, y otro material que aporte color mediante el uso de una ortofoto, que puede ser obtenida, como la altimetría, a través del centro de descargas del IGN en España. Para tener un pequeño descanso de la programación y utilizar un poco la interfaz gráfica de *Blender* el primera material se definirá de forma manual en la ventana denominada *Shader Editor* y se explicará cómo se programa con el segunda material ya teniendo una noción del funcionamiento de la definición de materiales complejos con *Blender*. Si bien ambos materiales han sido programados y se encuentra en el correspondiente código (ver 257 para el material tipo rampa de colores y 373 para el material tipo ortofoto en el anexo I).

Rampa de colores

La edición manual de materiales se realiza a través de la ventana *Shader Editor* en la que se definen los materiales con nodos (*nodes*) y uniones de estos (*links*), como un «lenguaje de programación» visual. Los nodos por su parte poseen unos atributos que permiten realizar cambios sobre el material, y que dependerán del tipo de nodo utilizado, así como unas entradas (*inputs*) y salidas (*outputs*), que son los datos requeridos por el nodo, y los resultantes del mismo, de manera que existen también nodos de entrada de datos y un nodo final denominado (*Material Output Node*) que es el material resultante.

Es el conjunto de nodos (*node tree*) el que genera el resultado final al ir realizando cambios sobre los datos de entrada en los nodos intermedios. Es posible también crear nodos a partir de un conjunto de nodos (*node group*).

Para definir un material lo primero que debe hacerse es crear el material (pulsar el botón *New*), donde aparecerá el material base que incluye los dos nodos más utilizados, que son el nodo final *Material Output* y el nodo *Principled BSDF* que eliminaremos en este caso.

Para poder realizar una rampa de colores en el modelo como las que se pueden generar en las herramientas *SIG* sobre las capas ráster, es necesario en primer lugar definir la propia rampa que de forma general se define como la división de un intervalo unidad mediante distintos colores, e interpolando la zonas entre colores. Los colores en este caso deben ser establecidos en función de la altimetría, por lo que es necesario

extraer los límites máximo y mínimo del modelo, y posteriormente normalizarlos al rango unidad con el que se define la rampa de colores. Por último, sólo es necesario aportar la geometría del conjunto para saber a qué lugar corresponde cada color.

Para añadir un nodo se busca en el menú *Add* de la ventana *Shader Editor* y se selecciona para que aparezca en el editor. Cambiar las propiedades de cada nodo es tan sencillo como escribir un número, mover el cursor habiendo pinchado sobre la misma, etc. es decir, un manejo igual al de cualquier otra ventana gráfica. Para unir nodos se pincha con el cursor sobre uno de los círculos del lateral derecho de un nodo (*output*) y se arrastra este hacia uno de los círculos del lateral izquierdo (*input*) de otro nodo, momento en el que se deja de pulsar el ratón, quedando la conexión fijada. Para saber qué elementos de entrada y salida conectar la interfaz ayuda a través de un código de colores y el nombre de cada elemento. De esta forma se añaden los siguientes nodos:

- *Geometry*: Otorga la información geométrica de la malla al material.
- *Separate XYZ*: Separa los valores de la geometría a valores individuales X, Y y Z; del que el material en este caso sólo necesita los valores Z.
- *Value*: Para representar la Z mínima para realizar la normalización al rango unitario de la rampa de colores.
- *Value*: Para representar la Z máxima para realizar la normalización al rango unitario de la rampa de colores.
- *ColorRamp*: Representa la rampa de colores. Con los botones + y - se añaden colores a la rampa, y pinchando en cada color o marcando su índice se permite el cambio de color. En el cuadro de texto *Pos* (*Position*) se indica la posición del color en el rango unitario.

Y se conectan entre sí como aparece en la siguiente figura:

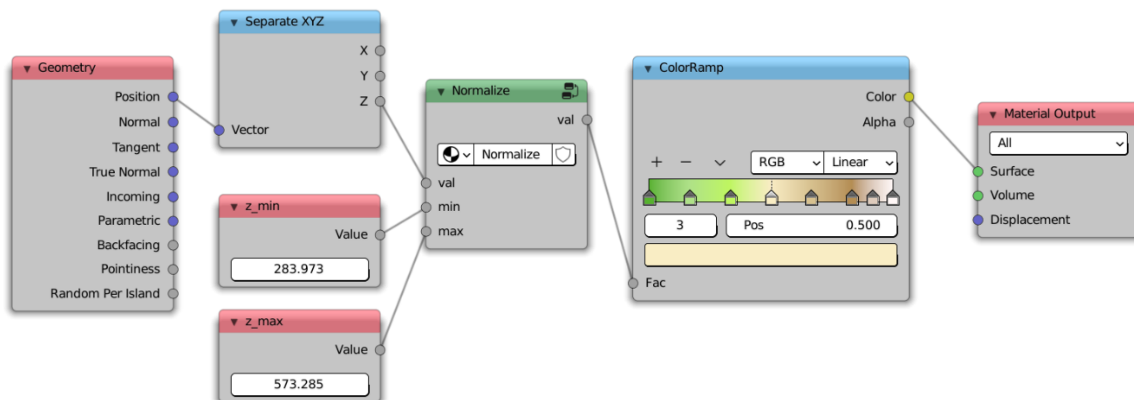


Figura 11. Material tipo rampa de colores para un modelo tridimensional del terreno.

En la figura aparece otro nodo denominado *Normalize* que como su propio nombre indica realiza la normalización de la altimetría, pero este nodo no es un nodo como tal sino un *node group* que hay que crear. Para ello, se añaden los siguientes nodos:

- *Math*: seleccionándolo como función *Subtract* y tomando los valores de la geometría y el valor mínimo para hacer la diferencia entre ellos de manera que queden valores que empiecen en cero.
- *Math*: seleccionándolo como función *Subtract* y tomando los valores máximo y mínimo cuya diferencia se usa para conocer el tamaño del rango de valores original.
- *Math*: seleccionándolo como función *Divide* para realizar la normalización con los valores anteriores.

Se seleccionan y se agrupan con *Add* → *Group* → *Make Group*, apareciendo los nodos *node input* y *node output* se sale de la ventana de edición del grupo de nodos mediante el botón con un icono de flecha hacia arriba que se encuentra en la esquina superior derecha de la ventana *Shader Editor* y aparece en la ventana el nuevo nodo creado.

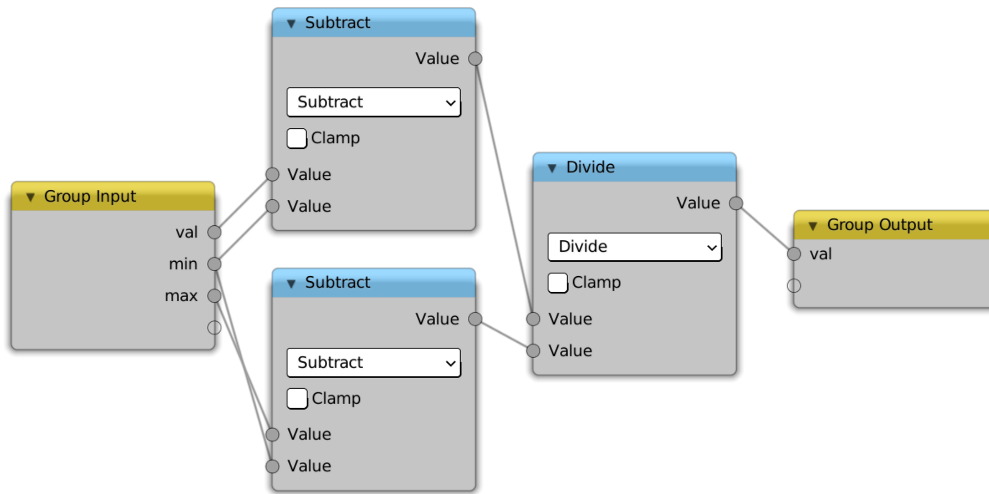


Figura 12. Conjunto de nodos para la normalización.

Por último, si este material no había sido creado con la malla seleccionada se añade el material a la malla cambiando al modo de edición al cambiar de *Object Mode* a *Edit mode* en la ventana *3D Viewport* y pulsando ahora en la ventana *Properties* en el apartado de *Material Properties* en *assign*. Finalmente, para poder ver los colores hay que estar en el modo *Solid* o *Rendered* del *3D Viewport*, lo cual se cambia en los botones con iconos en forma de circunferencia que hay en la esquina superior derecha del *3D Viewport*.

Ortofoto

Conociendo el funcionamiento de la edición de nodos a partir del desarrollo manual anterior, el desarrollo a través de la API de *Blender* se entiende con mayor facilidad. En primer lugar, se crea un material, y se define el árbol de nodos (*node tree*) como un conjunto de nodos unidos entre sí. Para ello se crean los nodos en el material y se unen los mismos en función del parámetro *input* y *output* requerido. En este caso son necesarias los siguientes nodos:

- *Image Texture*: utiliza la imagen como dato de entrada para establecer el color. Por lo que es necesario añadir una imagen en *Blender* como se hizo en el apartado de los modificadores, pero esta vez con atributo *colorspace_settings.name* en modo sRGB.
- *Principled BSDF*: contiene las propiedades más utilizadas en la creación de un material en un mismo nodo.
- *Material Output*: nodo final de cada material.

las líneas tipo quedarían de la siguiente forma:

```
material = bpy.data.materials.new(
  name=nombre_del_material)                         # Se crea el material
                                                    # con el que de forma
                                                    # automática se
                                                    # generan los nodos
                                                    # Principled BSDF y
                                                    # Material Output.

material_nodes.new('ShaderNodeTexImage')            # Se crea el nodo
                                                    # Image Texture que
                                                    # servirá como input
                                                    # para la imagen.

material_nodes['Image Texture'].image = imagen      # Se especifica la
                                                    # imagen a utilizar en
```

```
                                                                   # el nodo Image
                                                                   # Texture. De la
                                                                   # misma forma se
                                                                   # definen los
                                                                   # distintos
                                                                   # propiedades de cada
                                                                   # nodo.

material.node_tree.links.new(
  material_nodes['Image Texture'].outputs['Color'],
  material_nodes['Principled BSDF'].inputs['Base Color']) # Se unen los nodos
                                                                   # Image Texture y
                                                                   # Principled BSDF
                                                                   # mediante los
                                                                   # correspondientes
                                                                   # elementos.
material.node_tree.links.new(
  material_nodes['Principled BSDF'].outputs['BSDF'],
  material_nodes['Material Output'].inputs['Surface'])    # Se unen los nodos
                                                                   # Principled BSDF y
                                                                   # Material Output
                                                                   # mediante los
                                                                   # correspondientes
                                                                   # elementos.

objeto.data.materials.append(material)                  # Se aplica el
                                                                   # material sobre el
                                                                   # objeto activo.
```

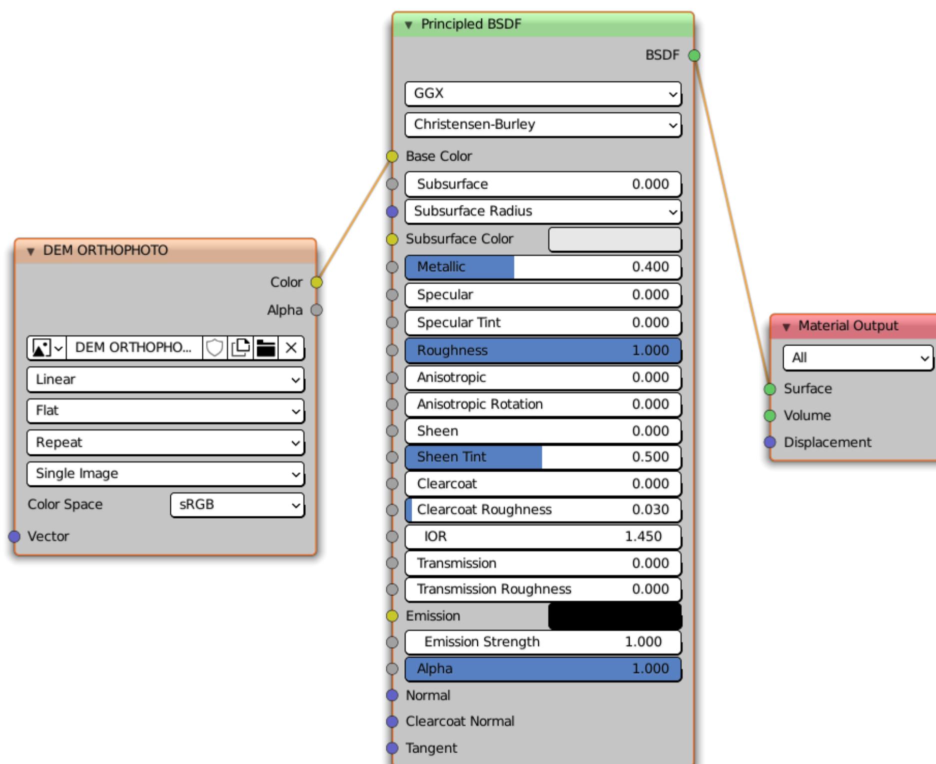


Figura 13. Material tipo ortofoto.

Resultados y uso de la herramienta creada

A continuación, se muestran los resultados de la asignación de los materiales definidos sobre los modelos del terreno. Hay que recordar que sólo serán visibles si la ventana *3D Viewport* se encuentra en modo *Solid* o en modo *Rendered*.

Para utilizar, el código desarrollado, terrain_creator, se utilizan en primer lugar las mismas líneas definidas en el apartado anterior para generar el modelo del terreno, y a estas se les aplica uno de los siguientes métodos:

```
tiff_dem.set_rc_material(
 color_ramp=nombre_de_la_rampa_de_colores) # Para utilizar una rampa de colores.
                                           # Se han creado dos rampas con
                                           # nombres RGB y ELEVATION.
tiff_dem.set_o_material(
  filepath=ruta_a_la_ortofoto)             # Para utilizar una ortofoto.
```

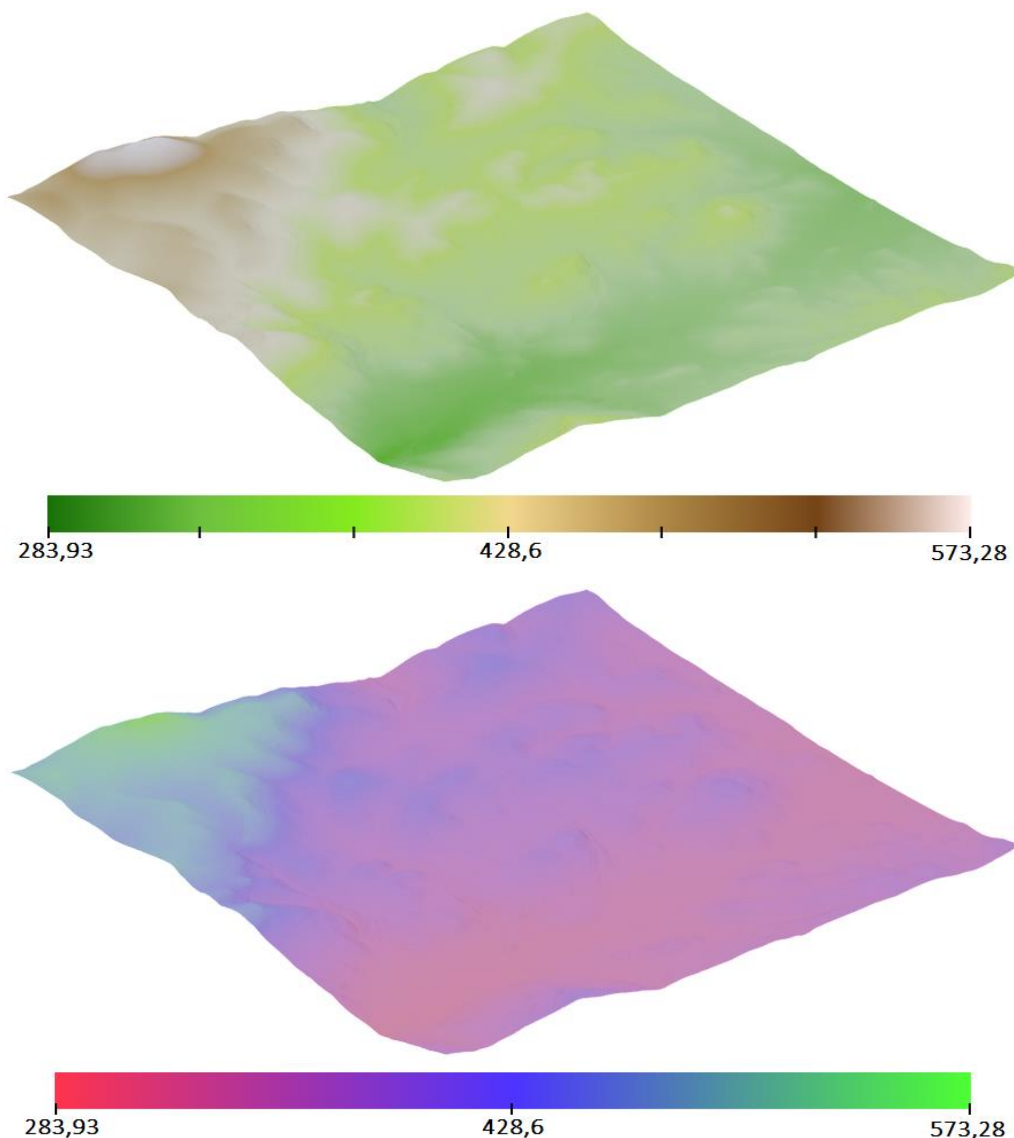


Figura 14. Modelo tridimensional del terreno coloreado mediante una rampa de colores definida en función de la altura.

Para demostrar las capacidades de Blender, la siguiente figura ha sido generada con una malla de 4,2 millones de caras y la ortofoto [6] utilizada es de 0,75 gigabytes.

Figura 15. Modelo tridimensional del terreno coloreado mediante una ortofoto.

Se puede ver en estas últimas figuras, que la malla no es apreciable, esto no se debe a la precisión de la mismas sino a que se ha añadido un proceso de suavizado (*Smoothing*) sobre las mallas, otra propiedad interesante que incluye *Blender* y que ayuda a un resultado más realista y vistoso de los modelos. En la siguiente figura se puede apreciar mejor el cambio entre una malla no suavizada y una suavizada:

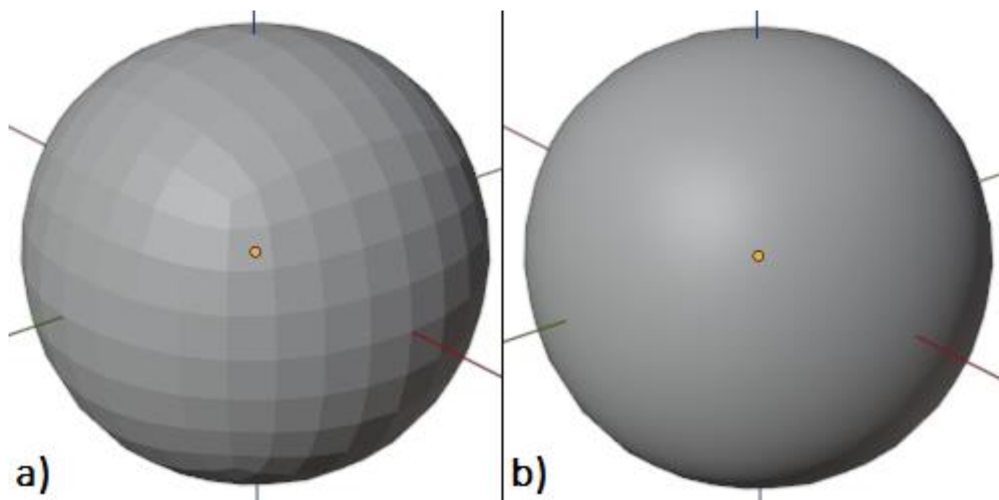


Figura 16. Comparativa de malla suavizada a) y no suavizada b).

3.1.4. Blender GIS

Si bien en los apartados anteriores se ha desarrollado la metodología y el código necesarios para producir un modelo tridimensional del terreno, esta idea no es nueva para *Blender*, y es que las industrias que dan uso a este programa tienen también la necesidad de generar modelos tipo terreno, para conformar los ambientes en los que se desarrollan los videojuegos, animaciones o películas. Es por esto por lo que métodos

similares a los que se han descrito, pero por lo general sin la necesidad de partir de un dato real (la altimetría usada), sino que, mediante la generación de elevaciones por ruido, son utilizados de forma habitual en *Blender*.

En concreto el complemento gratuito *Blender GIS* permite realizar modelos tridimensionales del terreno a partir de imágenes como se ha realizado anteriormente, pero también mediante curvas de nivel (.shp). Permite la descarga directa de datos espaciales como la altimetría, ortofotos, edificios, carreteras, etc. de servicios web como *Open Street Map*. Igualando de esta forma el servicio ofrecido por *Infraworks*, pero permitiendo en todo momento el control total del modelo por parte del usuario, así puede utilizarse la ortofoto procedente de un servicio web, los edificios, carreteras, etc. y a la vez utilizar una fuente propia para el modelo de elevaciones que está limitado por los servicios web a 30x30m de precisión. En la siguiente figura se muestra un modelo obtenido a través de *Blender GIS*, que contiene una ortofoto, las elevaciones del terreno y edificios, todo ello extraído de servicios web incluidos en la herramienta.

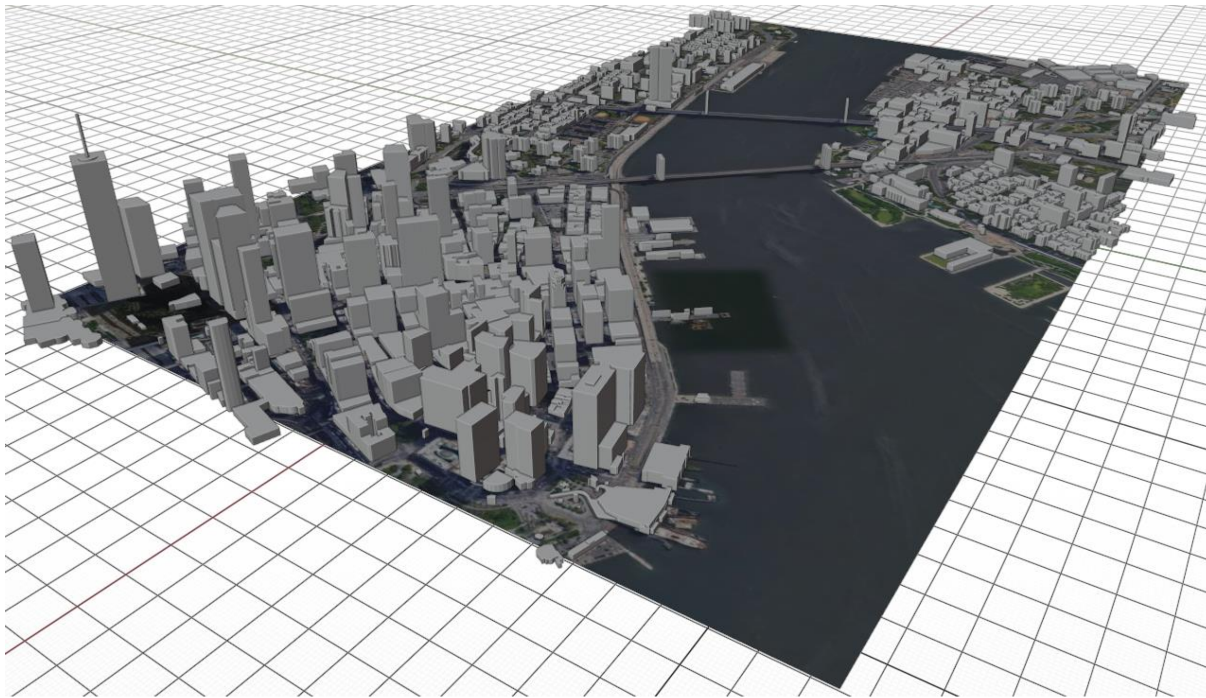


Figura 17. Modelo generado con Blender GIS con ortofoto, altimetría y edificios (Nueva York).

Conclusiones

Como se ha podido ver la programación en *Blender* es relativamente sencilla, e incluso permite realizar programas con resultados increíbles sin apenas conocimientos de la *API*, además, incorpora una serie de herramientas, los *modifiers*, que simplifican una gran variedad de procesos de gran utilidad en el desarrollo de aplicaciones puesto que de lo contrario tendrían que ser desarrollados por el programador.

Además, con el uso de archivos de distintos formatos para la obtención del modelo de elevaciones tanto si son directamente soportados por *Blender* o alguna librería de *Python*, como si no lo son, se ha demostrado la compatibilidad del programa con distintas fuentes de información. También se ha comprobado en este caso que no existe limite en la precisión del modelo digital a generar más que el impuesto por los datos utilizados y la capacidad de ordenador que se tenga, ayudando *Blender* a realizar modelos complejos y de gran tamaño con poca capacidad computacional a través, como se ha visto, de distintos aspectos en el modelado, evitando así que el programa se quede bloqueado de forma constante o vaya lento como puede ocurrir con otros programas de diseño.

3.2. Extracción de perfiles longitudinales de una carretera

Continuando con la temática del terreno en el ámbito de las carreteras en este apartado se desarrolla una herramienta que permite extraer perfiles longitudinales para carreteras a través un perfil en planta conocido. Con esto se demuestra la capacidad que tiene el programa de realizar elementos bidimensionales. Esto es indispensable al ser los planos parte fundamental en la documentación actual de los proyectos de ingeniería de caminos, canales y puertos.

Para la definición de elementos bidimensionales es posible utilizar mallas planas (por defecto *Blender* incluye el plano como elemento primitivo). Sin embargo, son las curvas las que toman mayor protagonismo en este ámbito. En concreto el programa tiene capacidad de realizar tres tipos principales de curvas. En primer lugar, las curvas tipo NURBS, definidas a través de un orden, y un conjunto de puntos de control a los que se les asigna un peso, de manera que cada punto de la curva se calcula como una suma ponderada de los (orden) puntos más cercanos. En segundo lugar, las curvas tipo Bézier, en las que se unen dos puntos de anclaje a través de una curva definida por puntos de control mediante funciones polinómicas. Y, por último, las polilíneas.

Desarrollo

Para la creación del perfil longitudinal de una carretera existente es necesario como información de partida:

- La altimetría de la zona, cuyas fuentes pueden ser las mismas que se han utilizado en el apartado anterior. En este caso se ha optado por un archivo .tiff.
- El trazado en planta de la carretera. Este se puede definir directamente en *Blende*r con el uso de una ortofoto o mediante la información geográfica existente en centros de descarga públicos como el centro de descargas del IGN en España, a través de una capa tipo *Shape* de carreteras. En este caso se ha optado por una capa *Shape* de carreteras procedente del IGN.

La información de la altimetría es accesible mediante las técnicas ya desarrolladas en el apartado anterior. Mientras que la información procedente del archivo *Shape* puede ser obtenida mediante la creación de un lector propio como se hizo con los archivos .asc o mediante librerías externas como se hizo con los archivos .tiff. Sin embargo, por simplificar, y aunque los archivos Shape no son de gran complejidad, en este apartado se optado por extraer la información de los vértices que componen la polilínea de una única carretera en un archivo .csv mediante QGIS que tiene la misma estructura que tenía el archivo .xyz descrito en el apartado anterior.

Con la información de la altimetría y la posición de los vértices que definen el trazado en planta de la carretera, únicamente es necesario extraer el valor de la cota perteneciente a cada vértice de la altimetría. Es posible, si se considera necesario, incrementar el número de vértices definiendo la posición de nuevos vértices mediante geometría básica, al extraer nuevos vértices de las rectas que unen dos vértices conocidos. Para obtener la cota de cada vértice únicamente hay que definir una transformación del sistema de coordenadas de los mismos al sistema de coordenadas propio de la información altimétrica, que para este caso es el propio de una imagen georreferenciada (una matriz FxC georreferenciada).

Conociendo toda la información de los vértices del trazado en planta, se debe calcular la distancia proyectada sobre un plano horizontal existente entre un vértice y el origen del perfil (vértice inicial). Esto se obtiene con el módulo del vector que une los dos vértices. Así el perfil longitudinal quedaría definido por las cotas de los vértices y las distancias. Para modelar todo esto se ha creado las siguientes clases:

- Una que representa cada vértice del perfil denominada *Point2D*.
- Otra que representa un vector bidimensional definido para el cálculo de las distancias entre puntos denominada *Vector2D*.
- Y otra que representa al perfil longitudinal y que lee los archivos de altimetría y trazado en planta, obteniendo la información de altura y distancia requerida denominada *Profile.*

Por otro lado, para poder crear una gráfica con sus ejes, etiquetas, marcas, etc. que contenga la información

necesaria del perfil longitudinal, se crean también las siguientes clases:

- Para definir las propiedades de cada eje (escala, separación de marcas, etc.) se crea la clase *TickProperty*.
- Para utilizar toda la información creada y dibujar el perfil en *Blender*, se crea la clase *ProfilePloter*, que recibe como atributos el perfil creado y las propiedades de cada eje, así como el valor que se desea para el primer punto kilométrico o las escalas a utilizar para cada eje.

La clase *ProfilePloter* crea mediante el método *blendify* un conjunto de objetos que componen la representación gráfica del perfil longitudinal. Estos objetos son los siguientes:

- Polilíneas para representar el marco de la gráfica y las marcas sobre la esta. Para crear este tipo de curva en *Blender*, en primer lugar, es necesario crear el elemento "curva"; este tipo de elemento está conformado a su vez de un conjunto de *splines*, que se definen mediante las coordenadas de sus vértices. La creación tipo de una curva de dos dimensiones tipo polilínea quedaría de la siguiente forma:

```
curva = bpy.data.curves.new(
  name=nombre_de_la_curva, type='CURVE')      # Se crea la curva
curva.dimensions = '2D'                       # Se indica que la curva será
                                              # bidimensional.
cur_spline = curva.splines.new('POLY')        # Se crea un spline de tipo
                                              # polilínea.
cur_spline.points.add(1)                      # Se añaden n vértices al spline.
cur_spline.points[n].co = (
  coordenada_X, coordenada_Y, 0.0)            # Se indican las coordenadas
                                              # (distancia al origen y cota)
                                              # del vértice.
obj = bpy.data.objects.new('X Ticks', curva) # Se crea el objeto.
```

- Curvas, en este caso se ha escogido las de tipo Bézier para la representación gráfica del perfil longitudinal. Para crear este tipo de curvas el procedimiento es similar al definido con las polilíneas, si bien es necesario en este caso definir también los puntos de control de cada vértice. En este caso se han utilizado puntos de control de tipo *AUTO* (automáticos) para simplificar el proceso, obteniéndose buenos resultados. La creación tipo de una curva de dos dimensiones tipo Bézier quedaría de la siguiente forma:

```
curva = bpy.data.curves.new(
  name=nombre_de_la_curva, type='CURVE')               # Se crea la curva.
curva.dimensions = '2D'                                # Se indica que la curva
                                                       # será bidimensional.
cur_spline = curva.splines.new('BEZIER')               # Se crea un spline de
                                                       # tipo Bézier.
cur_spline.bezier_points.add(n)                        # Se añaden n vértices
                                                       # al spline.
cur_spline.bezier_points[0].co(
  coordenada_X, coordenada_Y, 0.0)                     # Se indican las
                                                       # coordenadas (distancia
                                                       # al origen y cota) del
                                                       # los vértices.
cur_spline.bezier_points[n].handle_right_type = 'AUTO'# Se indica el tipo de
                                                       # punto de control
                                                       # izquierdo.
cur_spline.bezier_points[n].handle_left_type = 'AUTO' # Se indica el tipo de
```

```
                                                   # punto de control
                                                   # derecho.
obj = bpy.data.objects.new('Profile', cur)         # Se crea el objeto.
```

– Objetos de tipo texto que representan las etiquetas de la gráfica (alturas y puntos kilométricos). Estos son también representados en *Blender* mediante curvas, pero en este caso un tipo especial no definido anteriormente denominado *FONT*, que permite controlar las propiedades del texto como: el tipo de letra, el tamaño, la alineación horizontal y vertical, etc. Su creación se lleva a cabo de la siguiente forma:

```
curva = bpy.data.curves.new(
  type="FONT", name=nombre_de_la_curva)      # Se crea la curva.
curva.body = contenido_del_texto             # Se indica el texto que
                                             # representa la curva.
objeto = bpy.data.objects.new(
  name=nombre_del_objeto, object_data=curva) # Se crea el objeto.
```

Una combinación de estas curvas asignadas a objetos del espacio modelo producen la representación final del perfil longitudinal de la carretera.

Resultados y uso de la herramienta creada

Para el uso del código desarrollado, longitudinal, es necesario partir de la información altimétrica mediante un archivo .tiff, y del trazado en planta de la carretera a partir de las coordenadas de los vértices del trazado ordenadas en un archivo .csv. Se crea con esta información, el objeto que representa el perfil:

```
perfil = Profile.by_paths(
  ruta_al_archivo_csv, ruta_al_archivo_iff) # Obtiene las distancias entre los
                                            # vértices del trazado y las cotas
                                            # de los mismos.
```

Con este se definen las propiedades de los ejes X e Y, que con los atributos por defecto se definen:

```
propiedades_eje_X = TickProperty(axis='X') # Define las propiedades del eje X.
propiedades_eje_Y = TickProperty(axis='Y') # Define las propiedades del eje Y.
```

Por último, se crea el objeto que construye la gráfica con los objetos anteriores y se llama a su método *blendify*:

```
constructor = ProfilePloter(
  perfil, propiedades_eje_X, propiedades_eje_Y) # Se crea el objeto que permite
                                                # construir el perfil.
constructor.blendify()                          # Se construye el perfile en
                                                # Blender.
```

En la figuras siguientes se muestra el perfil longitudinal creado a través de la herramienta desarrollada y la carretera que se ha utilizado.

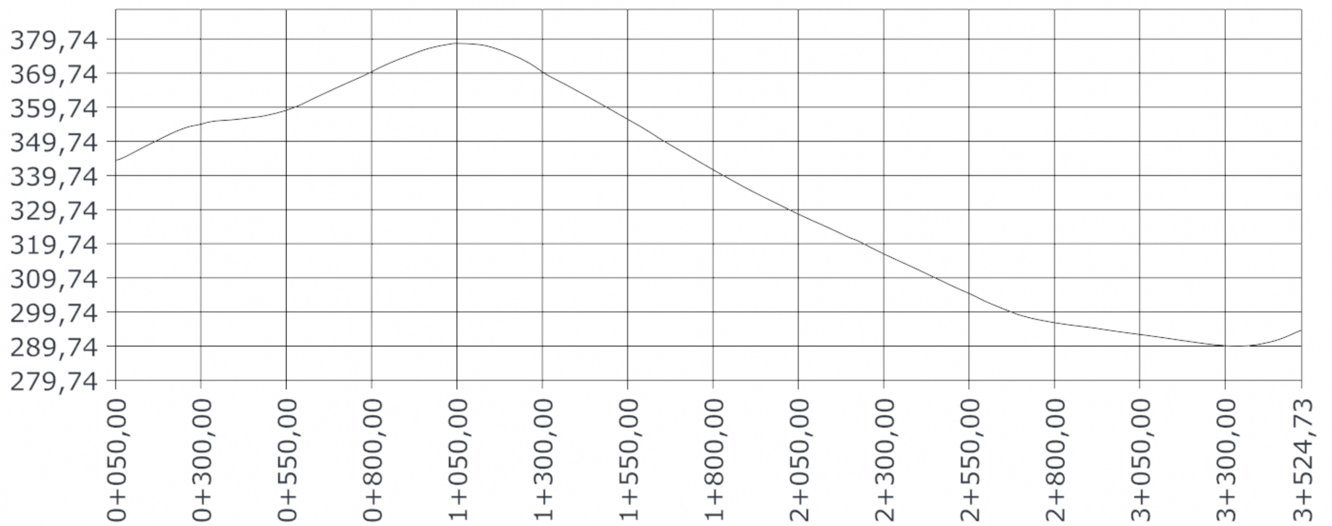


Figura 18. Perfil general de un tramo de la carretera A-373. Se ha establecido el PK de inicio en 50, la separación entre marcas del eje X de 250m y del eje Y de 10m.

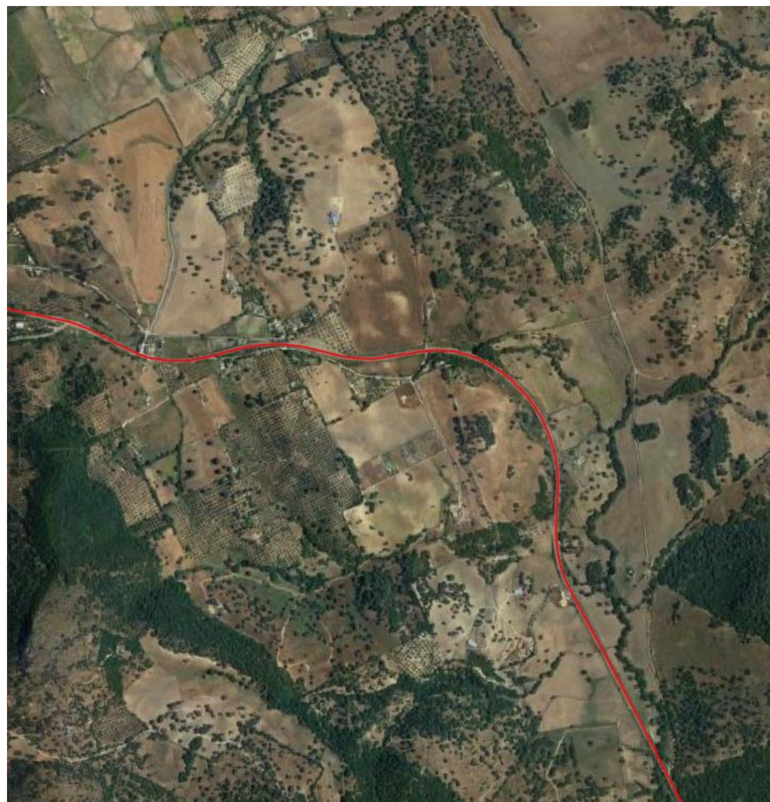


Figura 19. Tramo de la carretera A-373.

Conclusiones

En este apartado se puede ver de nuevo lo sencillo que es compatibilizar distintas fuentes de información a través del programa. Además, se introduce la capacidad de generar modelos 2D y por tanto la posibilidad de realizar planos con *Blender*, aspecto indispensable en la ingeniería de caminos, canales y puertos. Es posible además exportar los resultados obtenidos al formato .dxf, permitiendo la compatibilidad con otros programas de diseño.

4. Canales

4.1. Tuberías

Esta aplicación será usada para enseñar la creación de un complemento básico que añade un objeto (tubería) al espacio de trabajo. En primer lugar, se describirá como crear la tubería y en segundo lugar como crear con este código el complemento correspondiente. Se realizará además de forma que permite tomar tuberías normalizadas de distintos materiales para demostrar la facilidad con la que se puede añadir una norma o restricciones sobre los programas, y almacenar así información útil para el proceso de modelado en la ingeniería.

Desarrollo

La tubería puede ser definida a través de la parametrización de una malla que produce un cilindro hueco. En este sentido sólo hay que definir mediante la fórmula de la circunferencia la posición de cuatro anillos, dos internos y dos externos, separados dos a dos una distancia igual a la longitud de la tubería. Por tanto, son necesarios como datos de partida el diámetro interno y externo de la tubería o en su caso el espesor de la misma, así como la longitud de esta. Además, será necesario también definir el grado de precisión que los cilindros realizados deben tener, puesto que como se generará la geometría a través de una malla poligonal, en realidad los cilindros serán prismas con base en forma de polígono regular de r lados, siendo r la resolución definida. En la siguiente figura se muestran los cuatro anillos para dos resoluciones:

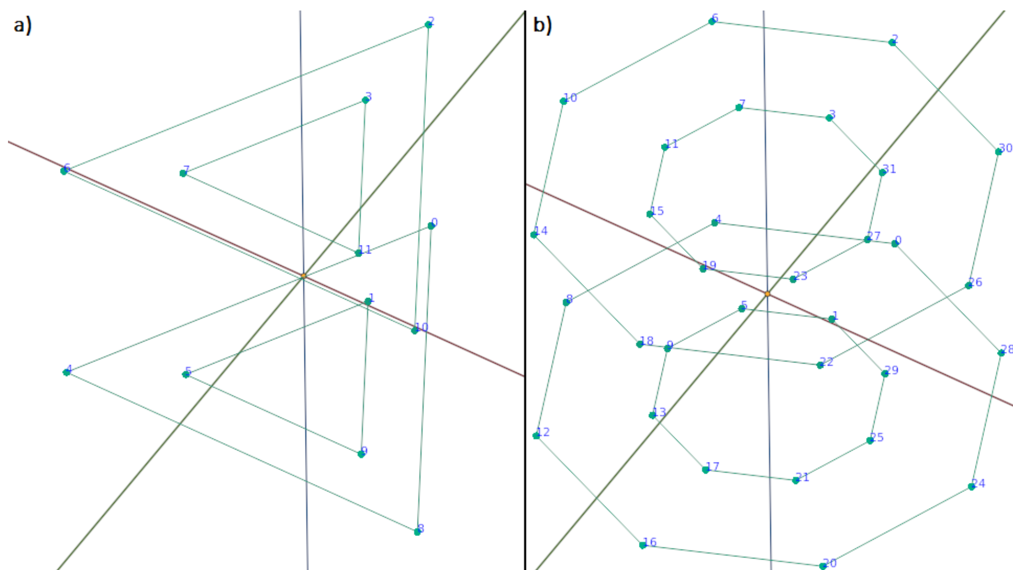


Figura 20. Anillos en forma de polígono regular con los que crear la malla que representa una tubería (cilindro hueco). a) r = 3 y b) r = 8.

Como es lógico, la resolución mínima será 3.

Una vez se tienen las coordenadas de los puntos que definen la geometría de la tubería, sólo hace falta definir las caras de la malla. Se ha escogido generar las caras mediante rectángulos que como se explicará

más adelante en este documento deben ser definidos en sentido antihorario. Con los vértices y las caras sólo hay que crear la malla y el objeto de la misma forma que en el primer apartado del documento.

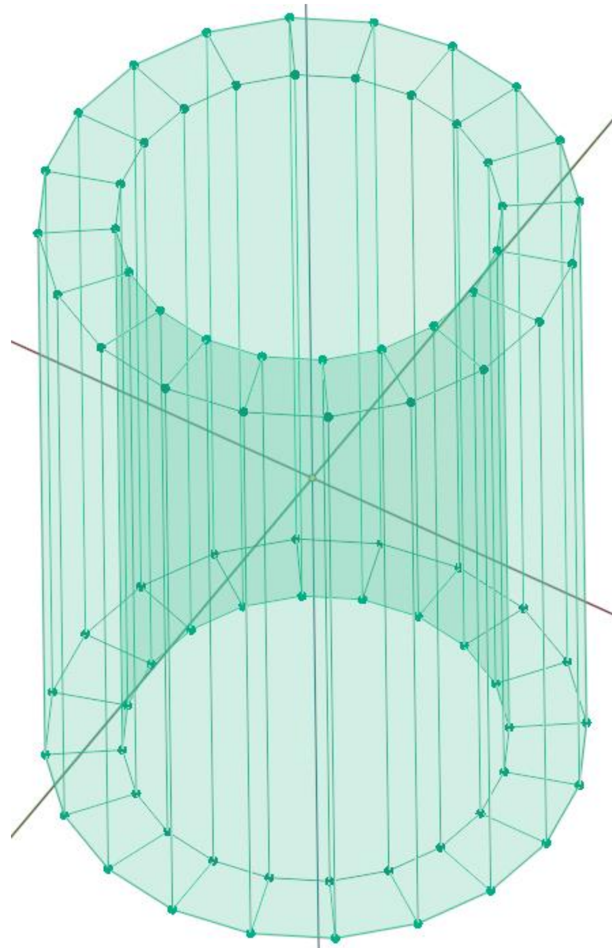


Figura 21. Malla con forma tubular.

Una vez parametrizada una pieza, si se quiere utilizar una geometría normalizada basta con utilizar los datos (diámetro interno y externo) correspondientes. En este caso se incluyen como ejemplo las tuberías de hormigón armado, fundición dúctil, polietileno y PVC. En cuanto al código únicamente es necesario almacenar los datos de las mismas de forma que sean accesibles a partir de unos parámetros de entrada, en este caso el tipo de material y diámetro nominal y para el caso de las tuberías plásticas densidad y presión de servicio. Para ello se han utilizado diccionarios de Python, como el que se muestra a continuación:

```
HORMIGON_ARMADO = {
    '400': (0.400, 0.065), # 'DN': (diámetro nominal en metros, espesor)
    '500': (0.500, 0.070),
    '600': (0.600, 0.080)
}
```

Complemento (Add-on)

Sabiendo cómo generar el objeto, y los parámetros que se quieren utilizar para la creación se puede crear el complemento. Debe aclararse que este no es el único tipo de complemento posible, ni es la única forma de crearlo.

Como se mencionó en el apartado 2.3, es posible acceder a una plantilla de cada tipo de complemento a través de la ventana *Text Editor* como base en el desarrollo de estos. En este caso será del tipo *Operator Mesh Add*.

En primer lugar, es obligatorio definir el siguiente diccionario en el archivo __init__init, que contiene los metadatos del complemento:

```
bl_info = {
  "name": "Pipe",                  # Nombre.
  "description": "Adds a pipe", # Descripción.
  "author": "",                    # Autor.
  "version": (1, 0, 0),            # Versión del complemento.
  "blender": (2, 92,),             # Versión mínima requerida de Blender para
                                   # ejecutar el complemento.
  "location": "",                  # Define en qué lugar del entorno de trabajo
                                   # puede ser encontrado el complemento. En este
                                   # caso al ser una herramienta que añade un
                                   # objeto
                                   # tipo malla a la ventana 3D Viewport, quedaría
                                   # como "View3D > Add > Mesh".
  "category": ""}                  # Categoría a la que el código pertenece. En
                                   # este caso "Add Mesh" por añadir una nueva
                                   # malla.
```

y al que se le pueden añadir otros argumentos. Por otro lado, en el archivo *__init__* se llama al archivo que contiene toda la lógica de la herramienta (add_pipe) a través de las siguientes funciones:

```
def register():   # Esta función sólo se ejecuta para activar el complemento.
  add_pipe.register()

def unregister(): # Esta función sólo se ejecuta al desactivar el complemento,
                  # para desactivar todos los cambios realizados por la función
                  # register.
  add_pipe.unregister()
```

Posteriormente en el archivo que contiene la lógica del complemento (add_pipe), se crea una clase con herencia de las clase *bpy.types.Operator* y *bpy.bpy_extras.object_utils*, que permiten acceder a distintas funcionalidades dentro de la *API* de forma directa. Es necesario crear dentro de la clase los siguientes atributos:

```
class nombre_de_la_clase(bpy.types.Operator):
  bl_idname = "object.nombre_del_objeto"     # Nombre interno del objeto para
                                              # Blender.
  bl_label = ""                               # Texto que aparece para
                                              # identificar al botón por el que
                                              # se accede a la herramienta.
  bl_description = ""                         # Descripción de la herramienta
                                              # que aparece al poner el ratón
                                              # sobre el botón.
  bl_options = {"REGISTER", "UNDO", "PRESET"} # Opciones para este operador.
```

Después se incluyen el resto de las propiedades que requiere el objeto a través de una definición especial con la siguiente estructura:

```
nombre_del_atributo: = bpy.props.tipo_de_atributo(argumentos_del_atributo)
```

donde el tipo de atributo (*tipo_de_atributo*) indica el tipo del dato necesario, por ejemplo, en este caso: los diámetros o el espesor serán números decimales (*bpy.props.FloatProperty*), el tipo de material será un *enumerator* (*bpy.props.EnumProperty*) y la resolución será un número entero (*IntProperty*). En función del tipo de dato escogido, el panel gráfico generada para la aplicación permitirá incluir los parámetros de la tubería de una forma u otra, así en el caso de los números decimales y enteros, aparecerá una casilla en la que se podrá escribir, y en el caso del *enumerator* aparecerá un menú desplegable, puesto que por la propia naturaleza del mismo sólo puede ser escogido un material ya definido.

Por otro lado, los argumentos del atributo (*argumentos_del_atributo*), son para los tres tipos descritos:

```
nombre_del_atributo: bpy.props.EnumProperty(
  items= [(
    identificador,
    nombre,
    descripción),
    ...]          # Listado de posibilidades para el atributo, cada posibilidad
                  # debe contener al menos el identificador (nombre que se usará
                  # internamente en el código para acceder al valor), el nombre
                  # (nombre que aparecerá en el panel del complemento) y
                  # la descripción (descripción que aparece al poner el cursor
                  # sobre el parámetro en el panel).
  name="",        # Nombre con el que aparece el atributo en el panel.
  description="", # Descripción que aparece al poner el cursor sobre el
                  # parámetro en el panel.
  default=0)      # Valor por defecto del atributo.

nombre_del_atributo: bpy.props.FloatProperty(
  name="",
  default=1,
  min=0.01,       # Valor mínimo que se permite utilizar en el atributo.
  max=10,         # Valor máximo que se permite utilizar en el atributo.
  unit="",        # Tipo de unidad de medida: longitud (LENGTH), área
                  # (AREA), etc.
  precision=3)    # Número de decimales que se pueden utilizar.
```

Por último, se crean los siguientes métodos para la clase:

```
def draw(self, context):    # Contiene la información con la que se crea el
                            # panel gráfico de la aplicación creada, en el que
                            # el usuario podrá añadir el valor de los atributos
                            # definidos: el tipo de material, los diámetros,
                            # etc.

def execute(self, context): # Código que se quiere ejecutar con cada cambio
                            # sobre el panel de la aplicación. Por ejemplo,
                            # en este caso si se cambia el diámetro, en la
                            # ventana de modelado la tubería deberá ajustarse
```

```
                            # a la nueva geometría, cambiando la malla creada,
                            # mediante el código que genera la geometría del
                            # prisma tubular.

def register():             # Esta función sólo se ejecuta para activar el
                            # código creado, siendo la función a la que se llama
                            # desde el archivo __init__ al activar el
                            # complemento.
  bpy.utils.register_class(OBJECT_OT_pipe)
  bpy.types.VIEW3D_MT_mesh_add.append(menu_func)

def unregister():           # Esta función sólo se ejecuta al desactivar el
                            # código, para desactivar todos los cambios
                            # realizados por la función register. Siendo por
                            # tanto la función llamada al desactivar el
                            # complemento desde el archivo __init__.
  bpy.utils.unregister_class(OBJECT_OT_pipe)
  bpy.types.VIEW3D_MT_mesh_add.remove
```

Por tanto, en la función *execute* lo que se hace es ejecutar el código que genera la geometría incluyéndose las distintas formas en la que esto se lleva a cabo mediante condicionales. Y en la función *draw* se diseña el panel del complemento creado. Para ello se utilizan entre otras las siguientes líneas:

```
estructura = self.layout                # Objeto con el que se define la
                                        # estructura del panel del complemento.
columna = estructura.column()           # Crea una columna en el panel.
columna.prop(self, nombre_del_atributo) # Añade un atributo a la columna.
```

Debe mencionarse que cada cambio dentro del panel puede producir cambios además de en el propio objeto, en el panel, por lo que esta función también es ejecutada con cada cambio realizado en los atributos.

Resultados, instalación y uso del complemento creado

El complemento desarrollado (965 y 990 en anexo II), es algo más complejo de lo que se ha descrito, he incluye la capacidad de escoger si se desea realizar la tubería mediante datos de tuberías normalizadas o de forma libre. En función de la elección, que se realiza a través de una casilla, la interfaz permite una forma u otra de elección de los diámetros para lo cual cambia su diseño. Además, se incluye la capacidad de realizar transformaciones (traslaciones y rotaciones), en la pieza desde el panel. Se incluye también la generación del material (texturas) correspondiente al material de tubería elegido. Por último, también en función de los valores de ciertos atributos que se hayan escogido se muestran o no en el panel las medidas, en concreto si se escoge la creación de tuberías normalizadas, se muestran los valores correspondientes a las medidas normalizadas. En la siguiente figura se pueden ver las distintas formas que toma el panel gráfico en función de los atributos elegidos.

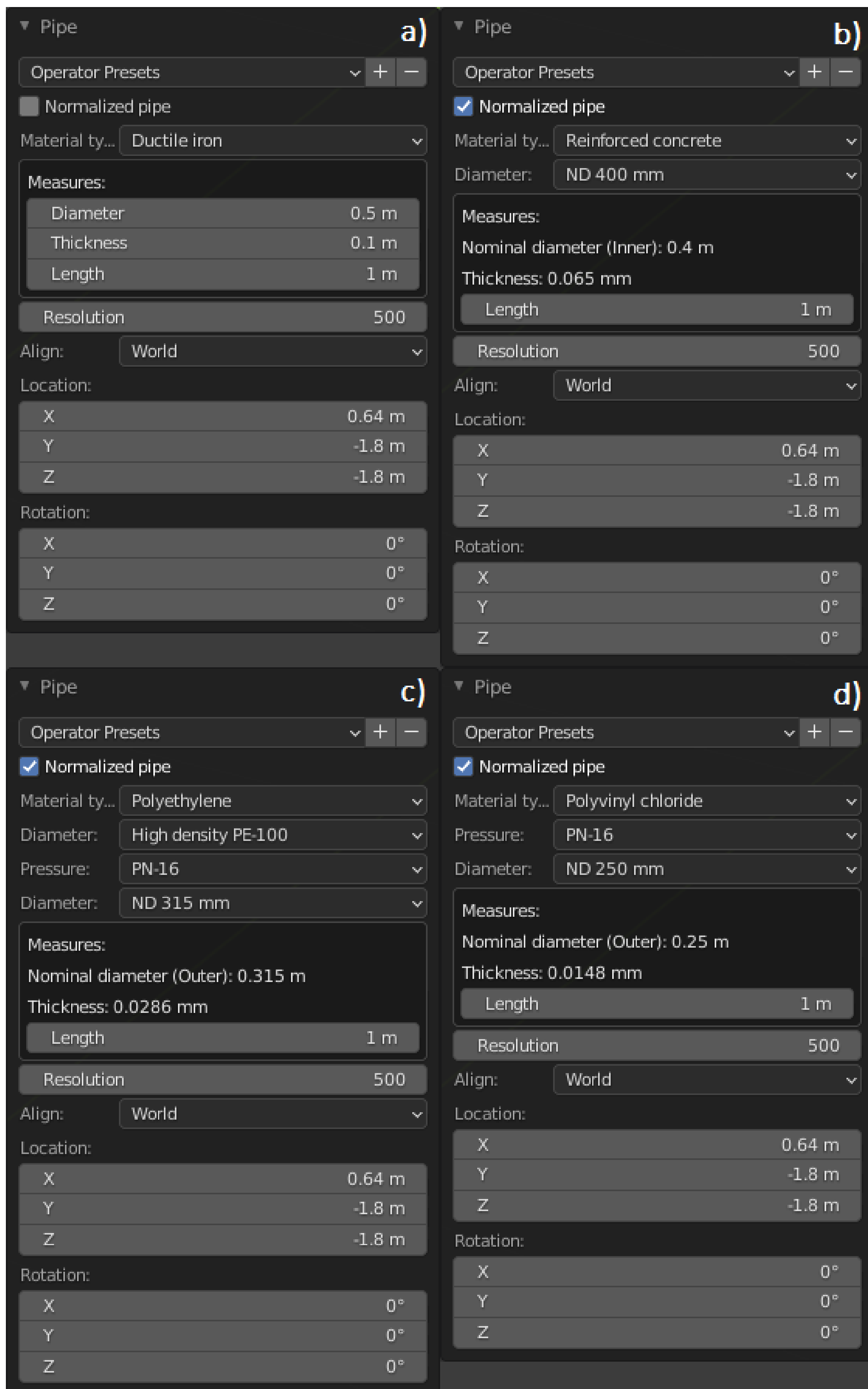


Figura 22. Diseños que adopta el panel del complemento creado en función de los parámetros escogidos: a) tubería no normalizada, b) tubería normalizada de HA, c) tubería normalizada de polietileno y d) tubería normalizada de PVC.

Para poder utilizar el complemento sin instalarlo, se puede ejecutar el código add_pipe desde el editor de texto llamando a la función *register* del archivo que contiene el código del complemento, y añadiendo el inicio de este la variable *bl_info* que se incluyó dentro del archivo __*init*__.

Si se quiere instalar, primero se crea un .zip con el archivo __*init*__ y el archivo add_pipe. Posteriormente se instala buscando el .zip en el apartado de *add-ons* de las preferencias del menú de edición pinchando en el botón *Install...*

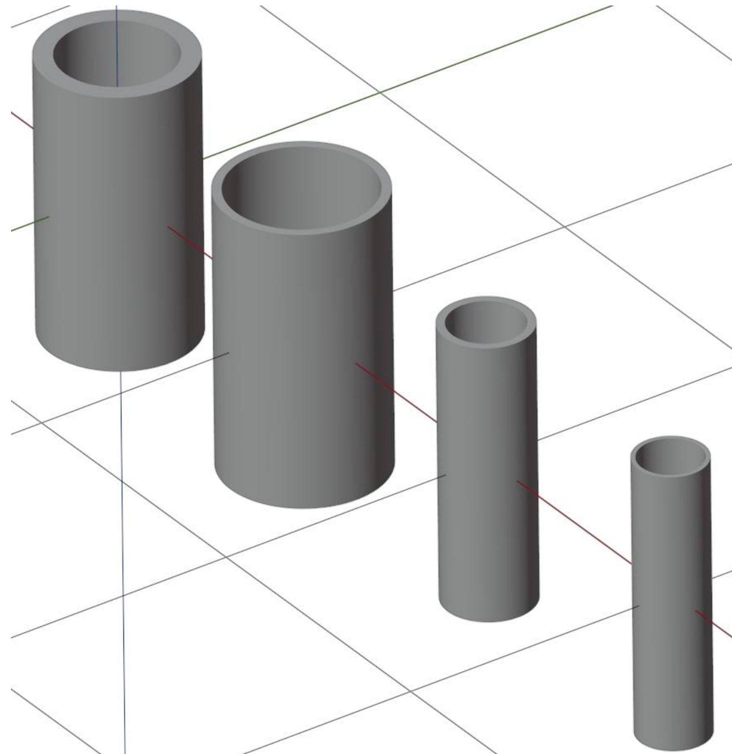


Figura 23. Tuberías obtenidas a través del complemento creado: a) de hormigón armado DN 400, b) de función dúctil DN 500, c) de polietileno de alta densidad PN-16 DN 315 y d) de PVC PN-16 DN 250.

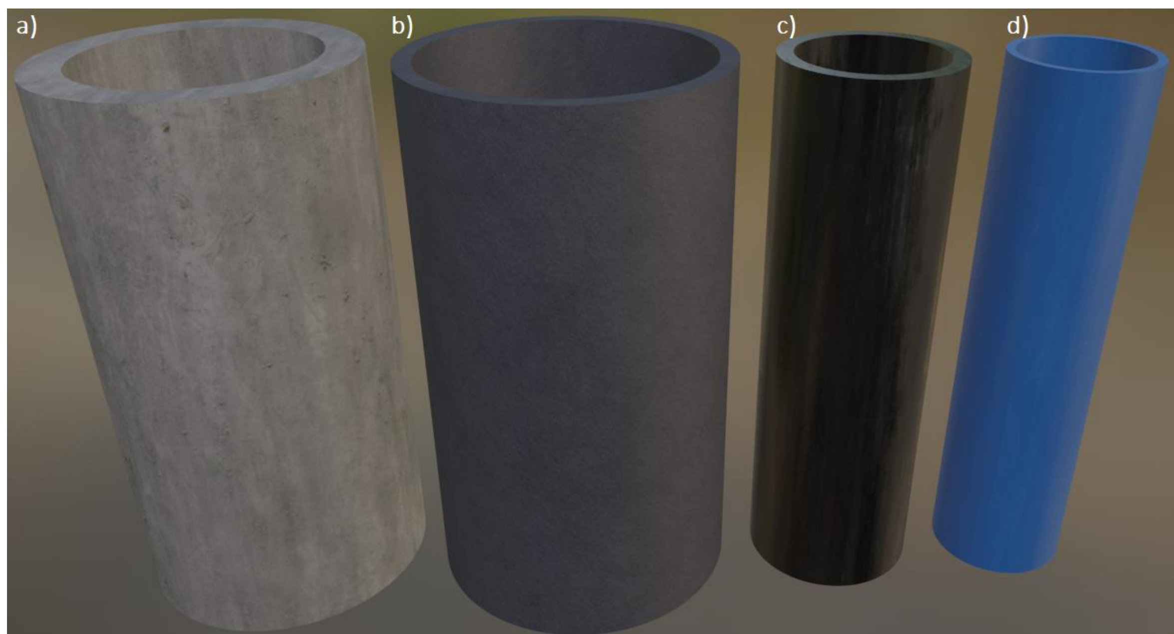


Figura 24. Tuberías obtenidas a través del complemento creado con los materiales elegidos para las mismas: a) de hormigón armado DN 400, b) de función dúctil DN 500, c) de polietileno de alta densidad PN-16 DN 315 y d) de PVC PN-16 DN 250.

Como se puede ver el resultado es fotorrealista. Y esto se ha conseguido sin entrar en el tema de la iluminación, cámaras y renderizado de *Blender*, y sin realizar grandes esfuerzos en la creación del material, ya que ello complicaría en exceso el trabajo, pudiendo ser por tanto los resultados mucho mejores y realistas que los obtenidos.

Conclusiones

Se ha logrado en este apartado ver la capacidad de integrar herramientas propias dentro de *Blender* a través del archivo __init__, que permite llamar al código que genera la geometría y crear un panel para retocar los atributos utilizados en su creación. También se ha visto lo fácil que es realizar una instalación y la distribución del mismo, sólo es necesario crear un archivo .zip. Esto demuestra las capacidades de personalización que *Blender posé*.

El uso de tuberías normalizas, es también muy interesante, porque al incluir esta información se facilita la labor del ingeniero en el modelado, pudiéndose evitar errores al realizar el diseño con tuberías de diámetros no fabricados.

También se ha podido ver la capacidad que tiene *Blender* de crear materiales fotorrealistas. Los nodos utilizados para crear estos materiales pueden verse en el apartado siguiente.

5. Puertos

En este apartado se analizan dos formas diferentes de generar elementos para un modelo, por un lado, se ahonda en el concepto de la parametrización de piezas ya introducido en apartado anterior; y por otro lado mediante la generación procedimental de objetos, que permite la creación de elementos de naturaleza aleatoria.

5.1. Escollera natural (Generación procedimental)

Como ya se mencionó en la introducción del documento, la ingeniería de caminos utiliza, a diferencia de otros campos de la ingeniería, además de piezas artificiales de geometrías conocidas, elementos naturales como una roca, que son de geometría aleatoria e imperrfecta. Por ello, es necesaria una metodología que permita la creación de dichos elementos. Y aunque a través de la programación se puede otorgar aleatoriedad a parámetros de una pieza parametrizada (lo cual se utiliza también en este apartado), es de mayor interés la capacidad de generar objetos con una componente de aleatoriedad aún mayor a través de la generación procedimental (*Procedural generation*). Esta metodología es ampliamente utilizada por programas como *Blender* con múltiples objetivos, crear terrenos sin poseer datos altimétricos, texturas, nubes, etc. permitiendo producir elementos que en la realidad son imperfectos e irrepetibles.

Desarrollo

La escollera natural utilizada en los diques de puertos suele colocarse por capas con rangos de tamaños, de manera que se van colocando capas de menor a mayor tamaño desde el interior del dique hacia el exterior del mismo, con el objetivo de que cada capa haga de filtro de la anterior, protegiendo el dique de la perdida de material. Por tanto, será necesario en el programa definir, para la creación de cada roca, un determinado

orden de magnitud que indique sus dimensiones. Para hacer esto sencillo, la roca se generará partiendo de un cubo de volumen conocido (cubo primitivo), y los vértices de este serán movidos en el espacio desde su posición original de forma aleatoria para cada eje entre un rango de valores definido.

Para crea el cubo se ha optado por el uso de un método de la *API*, en vez de la creación de la malla y el objeto como se ha realizado en anteriores apartados.

```
bpy.ops.mesh.primitive_cube_add() # Se crea el cubo primitivo.
objeto = bpy.context.object       # Se selecciona el cubo creado.
```

Para acceder y modificar las coordenadas de los vértices de un objeto existente:

```
objeto.data.vertices[indice_del_vertice].co[indice_de_la_coordenada] *= \
  numero_aleatorio
```

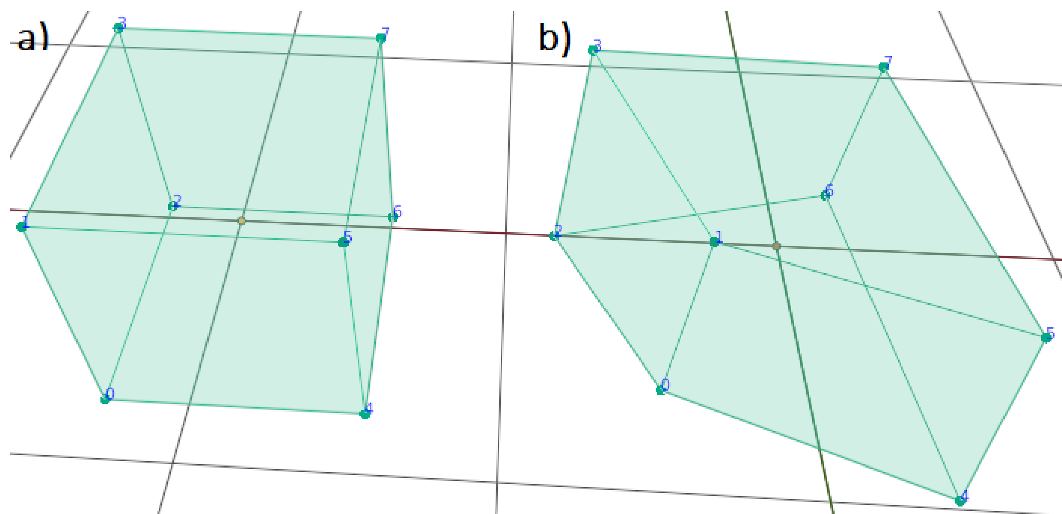


Figura 25. a) cubo primitivo y b) cubo con vértices modificados de forma aleatoria.

Es interesante también poder definir si la roca utilizada (el cubo modificado) es redondeada, acicular o plana, para ello se introduce un nuevo cambio en el cubo modificado, escalando este, o no, en función de la forma deseada de la roca.

```
objeto.scale = (1.0, escala_Y, escala_Z) # Escalado de los ejes Y y Z. Valores
                                         # iguales a 1 darían una roca
                                         # redondeada; si la escala Y se mantiene
                                         # en 1 y la Z se reduce, la roca tendría
                                         # forma de laja; si tanto la escala Y
                                         # como la Z se reducen, la roca sería
                                         # acicular.
```

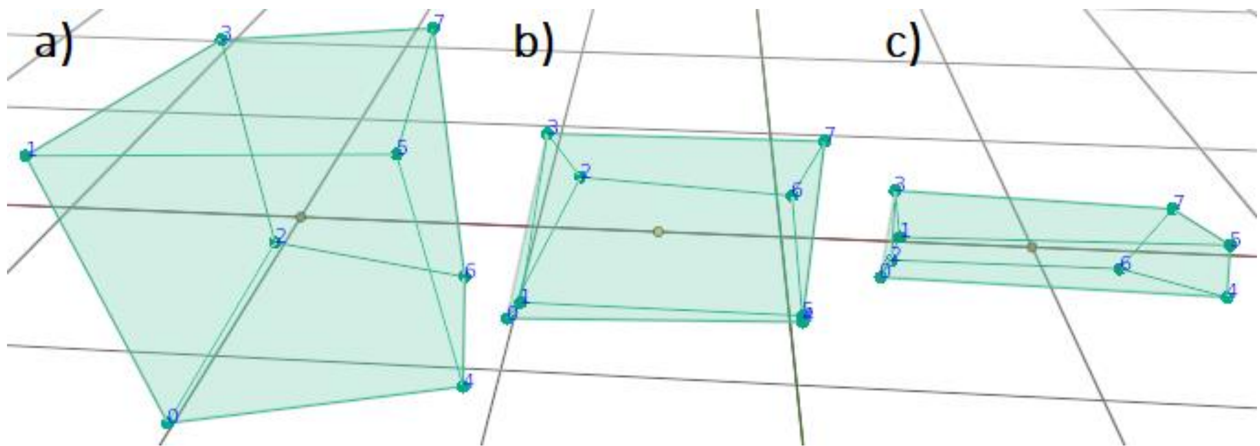


Figura 26. Cubo modificado: a) redondeado, b) plano (escala_Z=0,2) y c) acicular (escala_Y=0.2, escala_Z=0,2).

Hasta el momento, simplemente se ha introducido cierta aleatoriedad y cambios de dimensión sobre un cubo, pero este no parece una roca. Para cambiar esto, es necesario primero incrementar el número de caras del cubo, que permitirán introducir cambios en un mayor número de vértices, es decir, permitirán una mayor irregularidad de la superficie de la malla. Esto se lleva a cabo con el modificador *Subdivision surface* ya utilizado en el apartado 3.1.2.

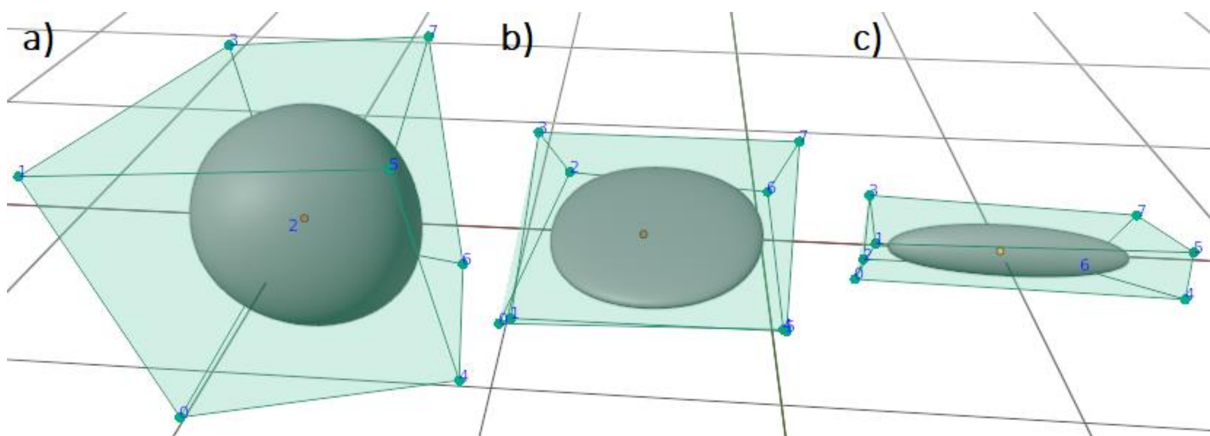


Figura 27. Cubo modificado con malla de mayor densidad: a) redondeado, b) plano (escala_Z=0,2), c) acicular (escala_Y=0.2, escala_Z=0,2).

Una vez se tiene la malla con un mayor número de vértices para dar irregularidad a esta, hay que aplicar nuevos desplazamientos aleatorios a los vértices. Para poder realizar esto en un rango de valores definido y que además el resultado de un aspecto real de imperfección o aleatoriedad, se aplica la técnica de generación procedimental. En este caso consiste en utilizar una imagen generada procedimentalmente que contiene ruido en un rango de valores definido, y utilizar esta como la textura usada por un modificador de tipo *Displace* también utilizado en el apartado 3.1.2. La aleatoriedad de los desplazamientos viene definida por medio de la textura creada. Sin embargo, no es necesario saber crear dichas texturas procedimentales, puesto que *Blender*, al ser esta técnica ampliamente utilizada para el diseño, incluye múltiples formas de generarla de forma automática. Estas texturas se crean matemáticamente, y algunas de las incluidas en *Blender* son:

- *Clouds.*
- *Musgrave.*
- *Stucci.*
- *Voronoi.*
- *Noise* (ruido generado de forma totalmente aleatoria).

Todas estas texturas a excepción de *Voronoi* y *Noise* tiene una propiedad que permite elegir el algoritmo

utilizado para generar el ruido base. Además, de esta propiedad cada método tiene sus propias propiedades que otorgan una gran flexibilidad a la hora de generar las texturas. De esta forma es necesario encontrar la combinación de valores que produce una textura que produzca desplazamientos en la malla de modo que parezca una roca. En el código creado (breakwater_rock), se han incluido tres de los métodos existentes, en primer lugar, el método *Clouds* que genera ruido de tipo *Perlin*, en segundo lugar, el método *Voronoi* y por último, el método *Stucci* que como indica la documentación de *Blender* «es a menudo usado para rocas, asfalto…».

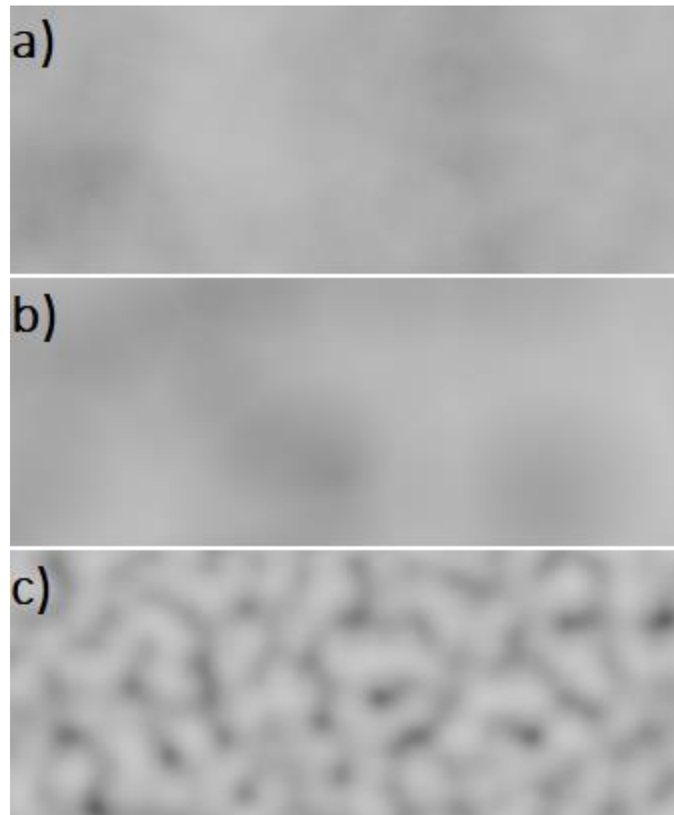


Figura 28. Muestras de ruido generado de forma procedimental, métodos: a) Clouds, b) Voronoi y c) Stucci.

Para la creación de las texturas, por tanto, se utiliza la *API* de *Blender* considerando para cada método utilizado los siguientes parámetros:

```
# Para el método Clouds.
textura = bpy.data.textures.new(
  nombre_de_la_textura, type='CLOUDS')  # Crea la textura de tipo Clouds.
textura.noise_basis = ruido_base        # Define el algoritmo a utilizar para
                                        # generar el ruido.
textura.noise_scale = escala            # Definida de forma aleatoria.
textura.noise_depth = profundidadS

# Para el método Voronoi.
textura = bpy.data.textures.new(
  nombre_de_la_textura, type='VORONOI') # Crea la textura de tipo Voronoi.
textura.noise_intensity = intensidad    # Definida de forma aleatoria.
textura.noise_scale = escala            # Definida de forma aleatoria.
textura.weight_1 = peso_1
textura.weight_2 = peso_2               # Definido de forma aleatoria.

# Para el método Stucci.
textura = bpy.data.textures.new(
  nombre_de_la_textura, type='STUCCI')  # Crea la textura de tipo Stucci.
textura.noise_basis = ruido_base        # Define el algoritmo a utilizar para
```

```
# generar el ruido.
```

El resto de las propiedades quedarían con los valores por defecto. Siendo la textura creada la introducida en el modificador *Displace*.

Además, se ha añadido la capacidad de crear un material para la roca, para ello. Se utiliza el siguiente árbol de nodos:

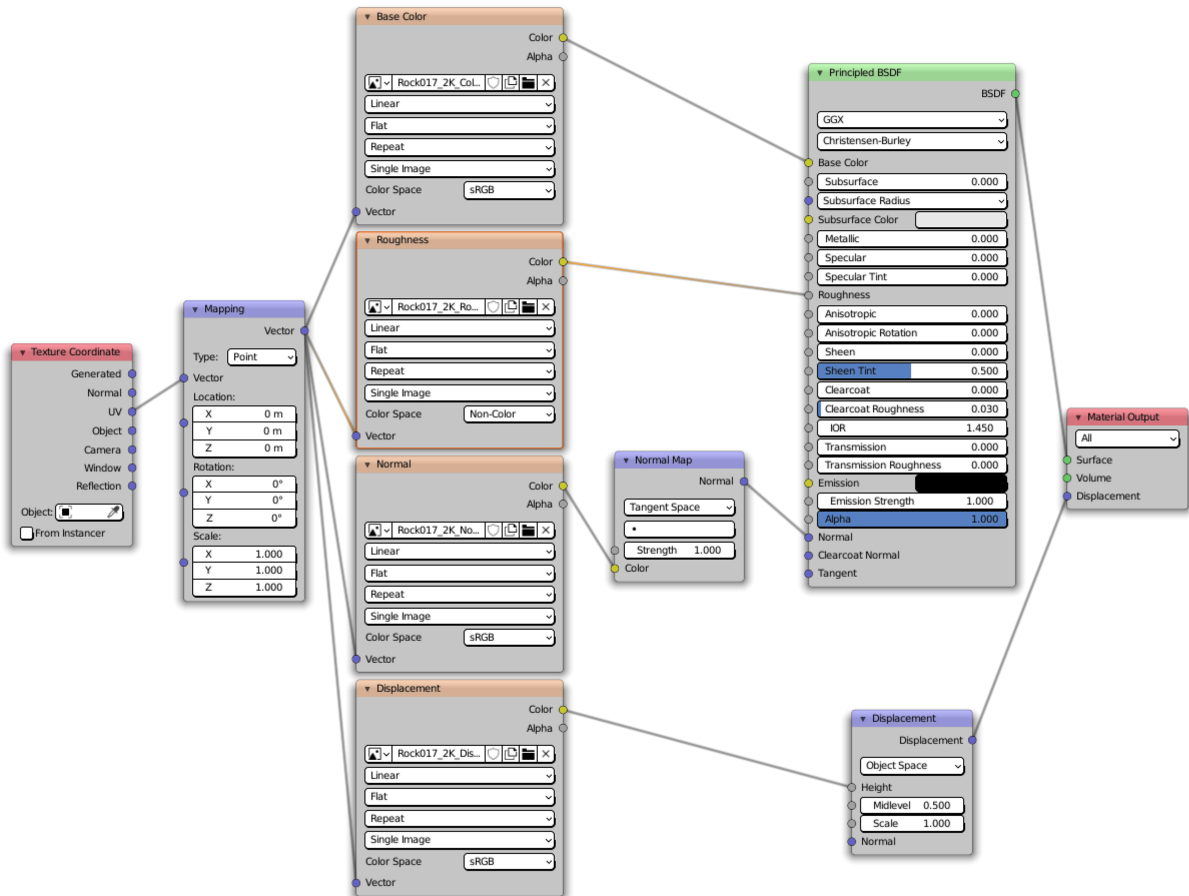


Figura 29. Estructura del material creado para las rocas.

En este caso se introducen los valores del nodo *Principled BSDF* a través de imágenes, mediante nodos del tipo *Image Texture* como las que se pueden ver a continuación:

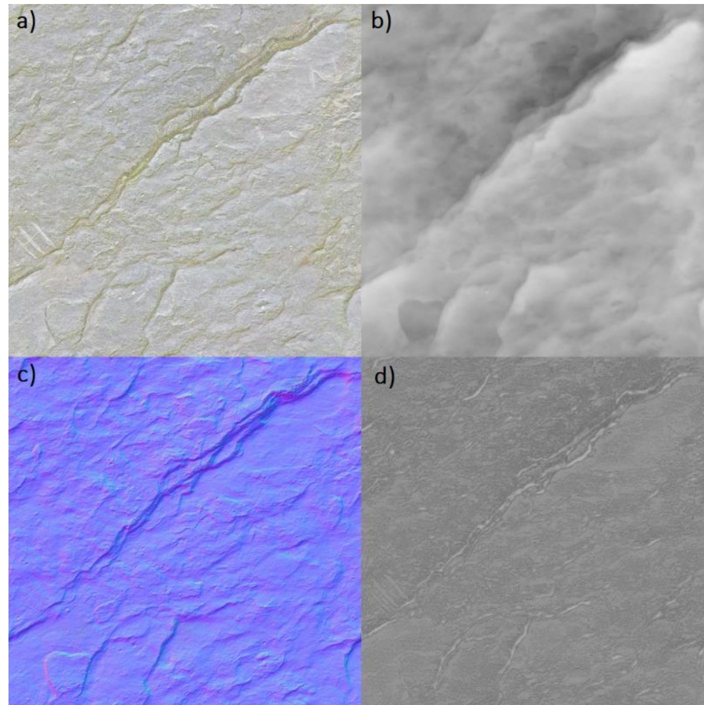


Figura 30. Imágenes utilizadas para definir el material de las rocas. a) color, b) desplazamiento c) normal y d) rugosidad.

Resultados y uso de la herramienta creada

Para utilizar el código creado ([breakwater_rock]), se crea el objeto roca y se llama al método *blendify* del objeto:

```
material = RockMaterial(
  base_color_filepath,
  roughness_filepath,
  normal_filepath,
  displacement_filepath) # Objeto que contiene las rutas a las imágenes para crear un
                         # material.
roca = NaturalRockFill(
  material=None,
  min_weight=1.0,
  max_weight=10.0,
  density=2.65,
  y_scale=1.0,
  z_scale=1.0,
  levels=4,
  noise_type='CLOUDS',
  noise_basis='ORIGINAL_PERLIN',
  strength=0.2,
  seed=0) # Al modificar los atributos se obtienen distintos tipos de roca. El
          # atributo material, es un objeto de la clase RockMaterial que
          # contiene únicamente las direcciones de los archivos que contiene las
          # imágenes utilizadas para la creación del material. Los atributos
          # min_weight y max_weight definen el rango de masas del cubo primitivo
          # con el que se creará la roca, teniendo en cuenta la densidad
          # (density). Modificando los atributos y_scale y z_scale, se obtienen
          # rocas: redondeadas, planas o aciculares. El atributo levels indica el
          # número de subdivisiones de la malla que se desean realizar. El
          # atributo noise_type indica el tipo de método con el que se va a
          # generar el ruido ('CLOUDS', 'VORONOI' y 'STUCCI'). El atributo
          # noise_basis indica el algoritmo a utilizar cuando se toma como
          # noise_type 'STUCCI' ('BLENDER_ORIGINAL', 'ORIGINAL_PERLIN',
```

```
            # 'VORONOI_F1', 'VORONOI_F2', 'VORONOI_F3' y 'VORONOI_F4'). Strength
            # indica la proporción con la que se utiliza el ruido. El atributo seed,
            # permite generar una semilla para la generación de los números
            # aleatorios usados en el código para conseguir así poder repetir
            # los resultados.
roca.blendify() # Se crea la roca en Blender.
```

A continuación, se muestran una serie de rocas generadas por los distintos métodos utilizados.

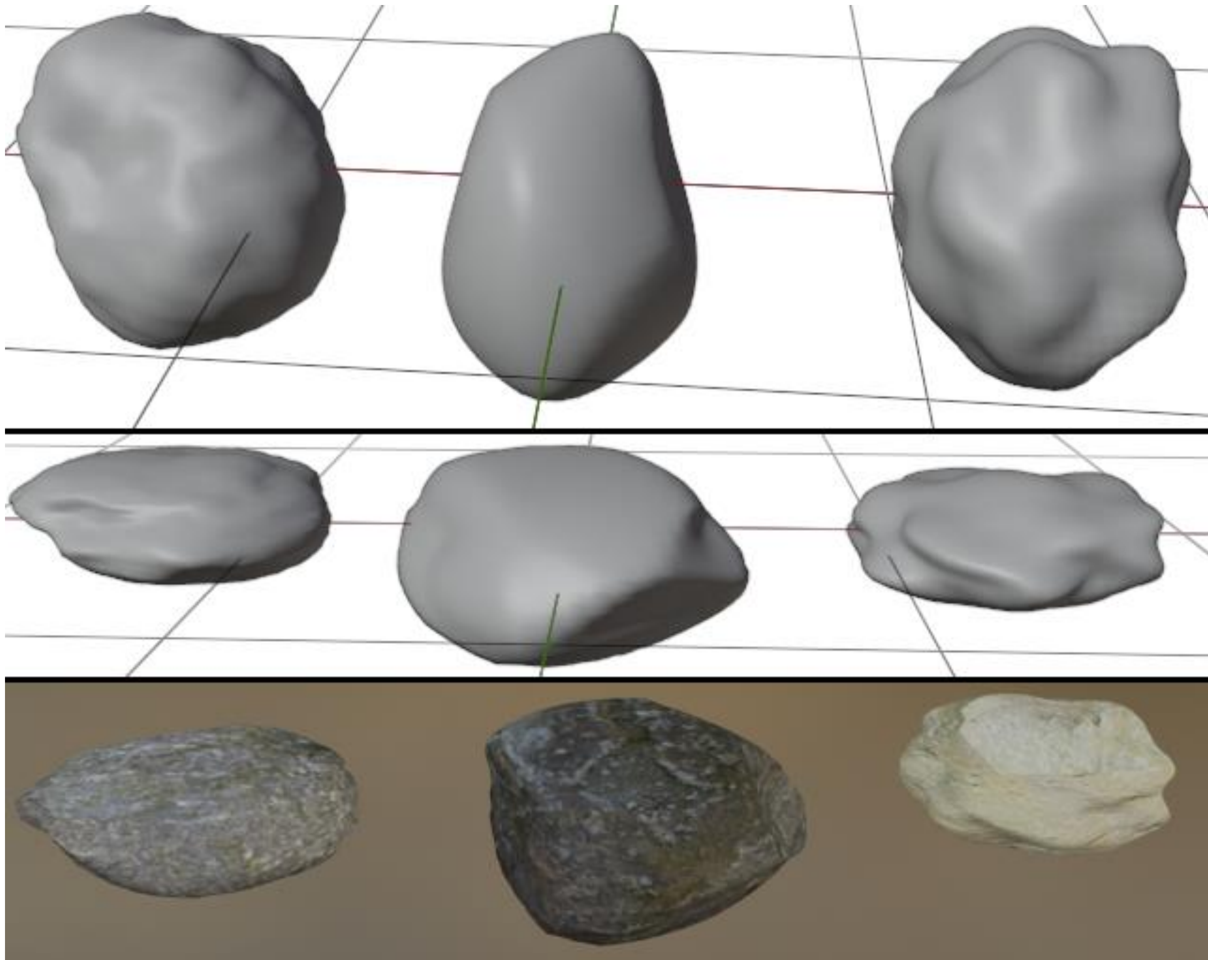


Figura 31. Rocas generadas mediante ruido procedimental. De izquierda a derechas métodos: Clouds, Voronio y Stucci. De arriba abajo: z_scale=1, z_scale=0.4, y z_scale=0.4 con material.

Es posible generar el talud del dique de forma automática, así como sus distintas capas, mediante un simple bucle. En las siguientes figuras se muestra como quedaría una capa de un dique.

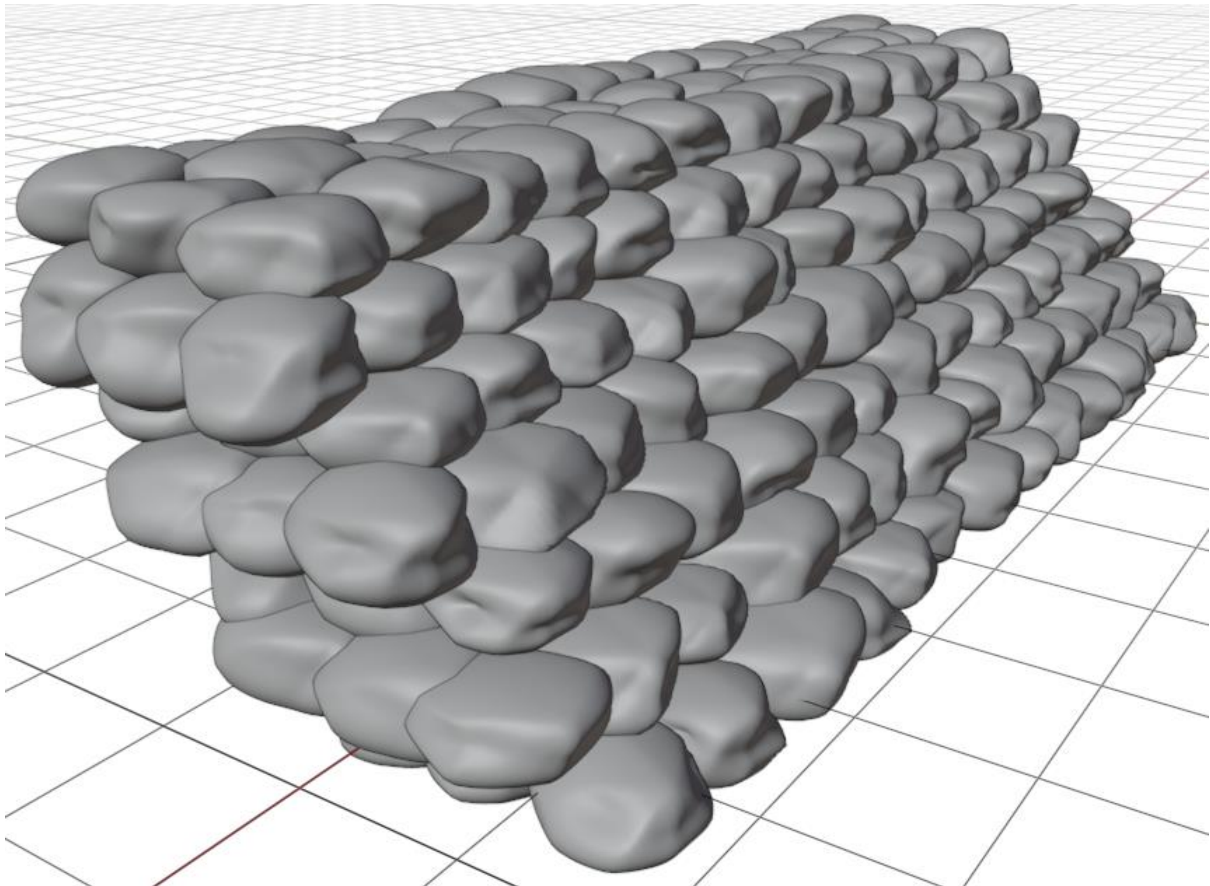


Figura 32. Vista lateral de una capa de dique con rocas generadas mediante ruido procedimental.

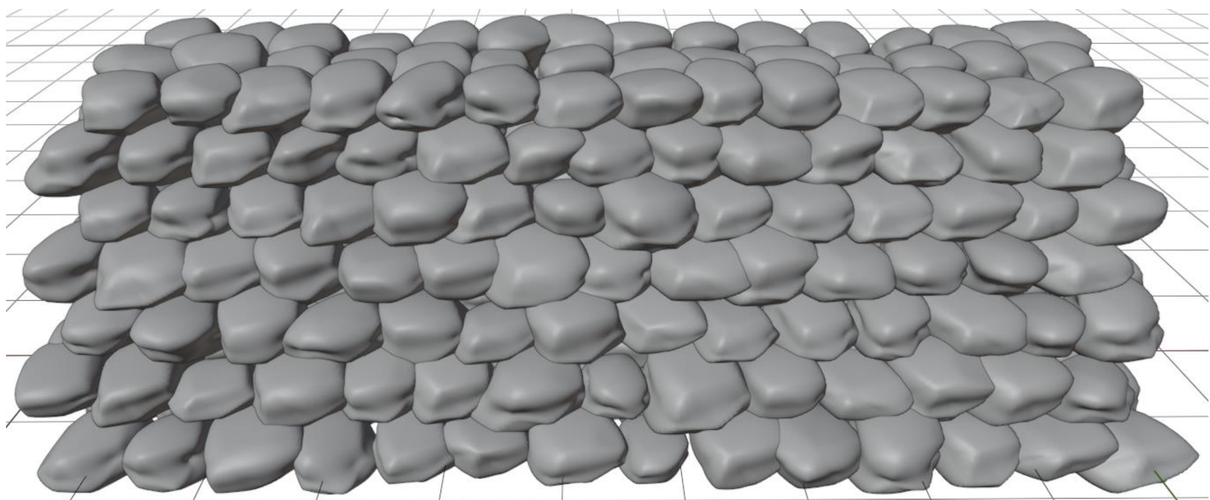


Figura 33. Vista frontal de una capa de dique con rocas generadas mediante ruido procedimental

Conclusiones

El resultado obtenido en las figuras **Figura 32**y **Figura 33** podría ser mejorado para que la posición de las rocas sea también aleatoria, si bien se puede comprobar que no existen dos rocas iguales en toda la capa, habiéndose logrado el objetivo de producir elementos irregulares e imperfectos de utilidad en la ingeniería de caminos, canales y puertos.

5.2. Escollera artificial (Generación parametrizada)

Como es lógico y ya se ha visto anteriormente también es posible la creación piezas de geometría conocida y definida. Mediante la parametrización de estas, se consigue sólo tener que crearlas una vez, y reutilizar el trabajo realizado la próxima vez que la pieza sea necesaria. En este apartado se crean varias piezas que representan bloques de hormigón utilizadas en diques rompeolas.

Desarrollo

Para parametrizar una pieza únicamente es necesario encontrar los parámetros necesarios para la creación de la malla. El caso más sencillo es el de un cubo de lado 2·h. Los vértices del mismos tomarían una de las siguientes combinaciones para sus coordenadas: (±h, ±h, ±h). Y una vez definidos los vértices se definen las seis caras del cubo con los índices de los vértices en sentido antihorario. Con los vértices y las caras se crea un objeto y su correspondiente malla como se definió en el apartado 3.1.1.

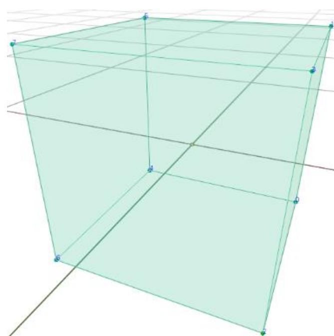


Figura 34. Malla en forma de cubo.

Es posible realizar la parametrización de piezas más complejas utilizando en el proceso transformaciones, modificadores y otras herramientas existentes en *Blender*. En este sentido, una herramienta de gran utilidad son los modificadores de tipo lógico (*Boolean*), que permiten crear un nuevo objeto a partir de la intersección (*Intersect*), la unión (*Union*) o la diferencia (*Difference*) de dos objetos.

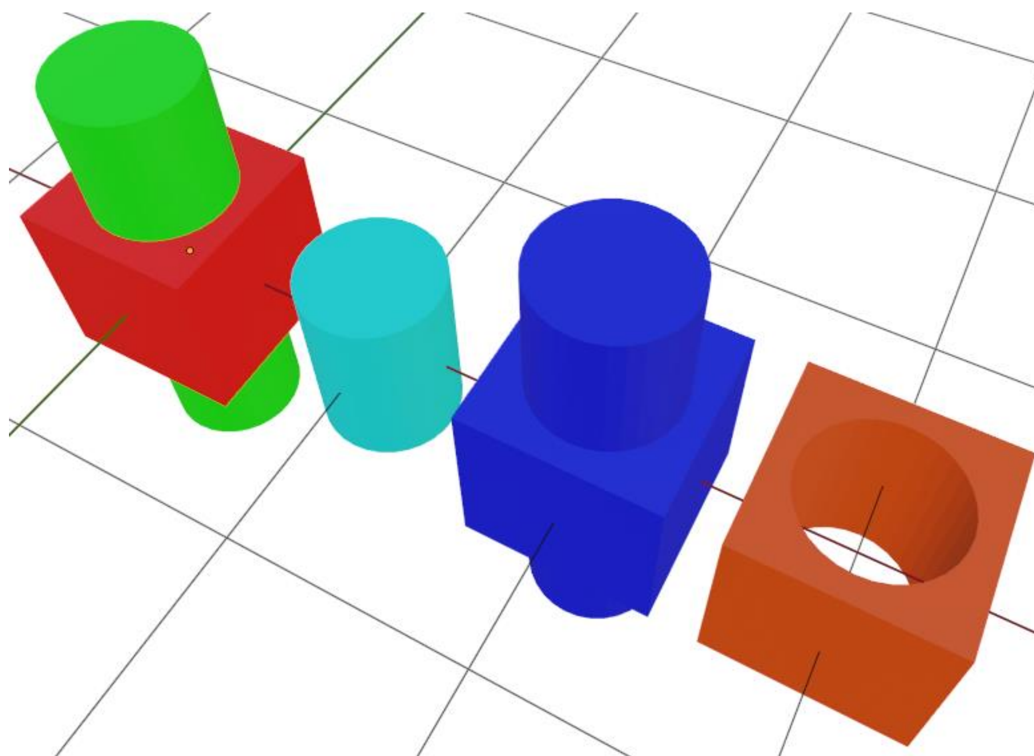


Figura 35. Objetos obtenidos mediante operaciones lógicas: a) intersección, b) unión y c) diferencia.

Para ello es de gran importancia crear las caras de las mallas correctamente, es decir, en sentido antihorario, al definir el sentido con el que se crean las caras la orientación de la superficie. Y si la orientación de las caras no es correcta la operación produce resultados inesperados.

A continuación, se muestra el uso de la unión para la creación de un bloque de hormigón rompeolas de tipo tetrápodo. En primer lugar, se define como pieza base un cilindro troncocónico cuya geometría se define de manera similar a como se creó el cilindro hueco que representaba la tubería en el apartado 4.1.

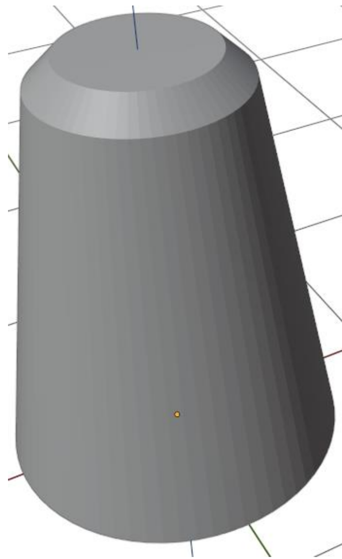


Figura 36. Piezas base para la creación del bloque tipo tetrápodo.

Es necesario crear cuatro piezas base, y una vez creadas, posicionarlas en el espacio de manera que el ángulo existente entre los ejes de dos piezas sea siempre 120 grados. Además, el centro de la cara inferior de las piezas base se debe encontrar en el mismo punto. Para esto se realiza el producto escalar de los vértices de cada una de las cuatro piezas base por una matriz de rotación.

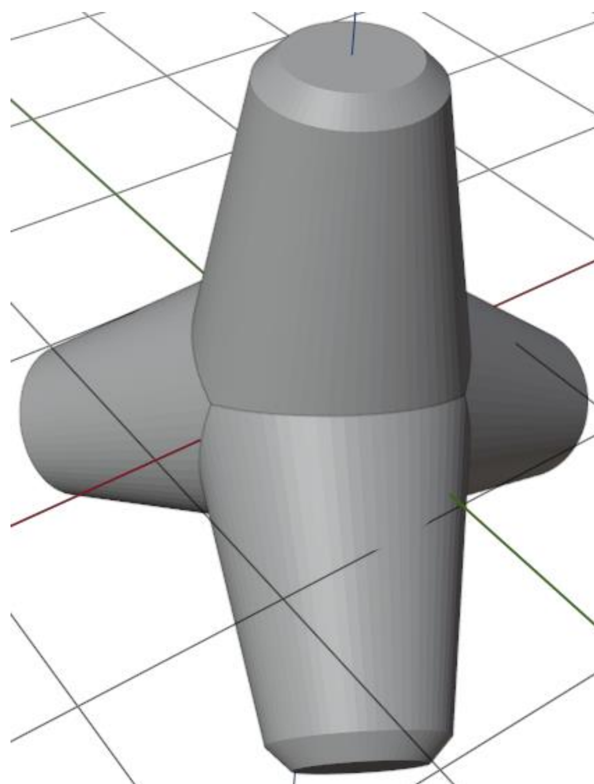


Figura 37. Piezas base que componen el bloque tipo tetrápodo posicionadas.

Ahora es necesario elegir una pieza principal, y a esta, unirle mediante modificadores lógicos de tipo unión las piezas restantes. Para ello se utiliza un código similar al expuesto a continuación:

```
pieza_principal                            # Objeto que representa la pieza
                                           # principal con la que se va a
                                           # realizar la unión.
pieza_secundaria                           # Objeto que representa a una de las
                                           # otras piezas con la que se va a
                                           # realizar la unión.
modificador_union = pieza_principal.modifiers.new(
  type='BOOLEAN', name='UNION')            # Se crea el modificador sobre la
                                           # pieza elegida como principal.
modificador_union.object = pieza_secundaria # Se define que el segundo objeto
                                           # necesario para la unión es una de
                                           # las otras piezas base creadas.
modificador_union.operation = 'UNION'      # Se especifica que la operación
                                           # lógica corresponde a la unión de
                                           # objetos.
bpy.ops.object.modifier_apply(
  modifier=modificador_union.name)         # Se aplica el modificador.
bpy.data.objects.remove(pieza_secundaria)  # Se elimina la otra pieza usada.
bpy.data.meshes.remove(malla_otra_pieza)   # Se elimina la malla perteneciente
                                           # a la otra pieza usada.
```

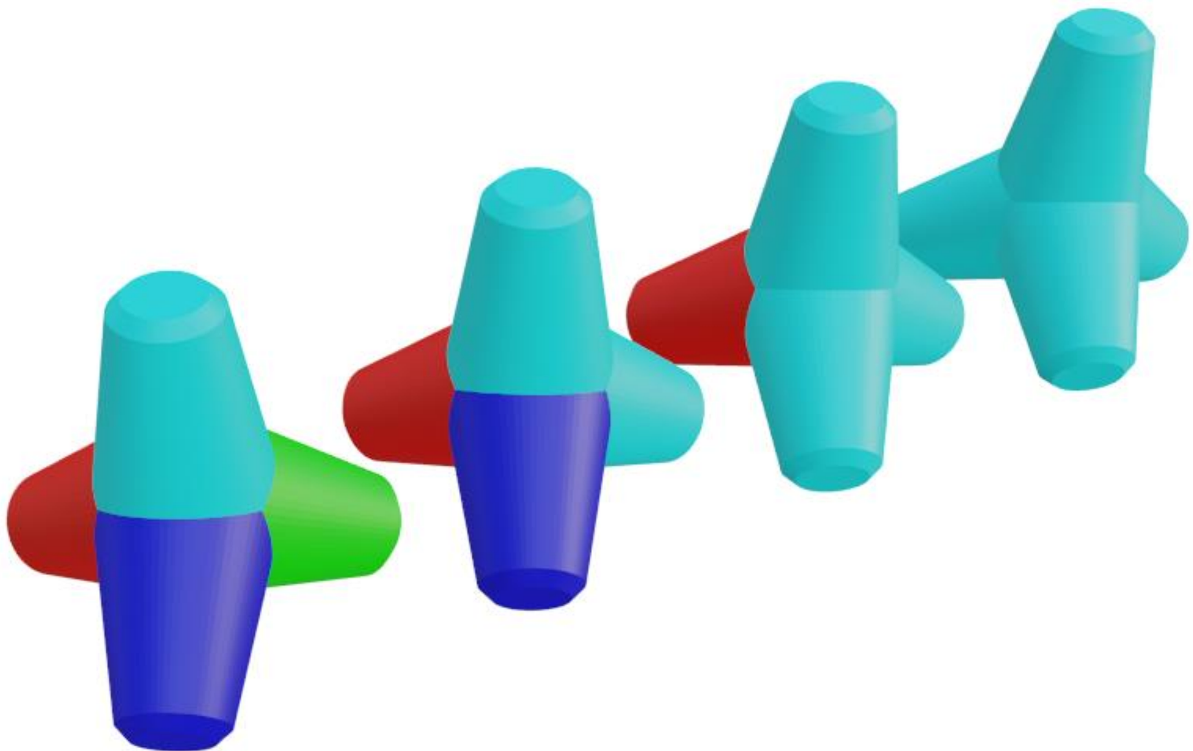


Figura 38. Proceso de unión realizado por pasos.

Permitiendo este tipo de herramienta la creación de piezas de geometría conocida pero que son difíciles de parametrizar de forma directa.

Al código creado se le han añadido valores normalizados de los parámetros que define cada una de las piezas creadas, de manera que las piezas se generan tomando la masa de la pieza normalizada. En el bloque tipo tetrápodo se incluye además la posibilidad de elegir la resolución de las piezas troncocónicas que lo conforman.

Resultados y uso de la herramienta creada

En la siguientes figuras se expone una muestra de los resultados que pueden ser obtenidos con el código creado (breakwater_blocks). Hay que tener en cuenta que, al realizarse el proceso mediante modificadores, el resultado no está siempre asegurado, los algoritmos pueden fallar, en este caso se puede comprobar que no se obtiene la solución deseada para cualquier resolución. Algunas resoluciones comprobadas para piezas de una tonelada son: 35, 49 o 67.

Para crear una de las piezas definidas, se ejecuta el código en el editor de texto con alguna de las siguientes líneas de código:

```
create_tetrapod(tn='16', resolution=67)
create_standard_akmon(tn='16')
create_half_akmon(tn='16')
create_flat_cross_akmon(tn='16')
create_hexaleg(tn='16')
create_tripod(tn='16')
create_cube(tn='16')
create_cob(tn='16')
create_dolos(tn='40')
```

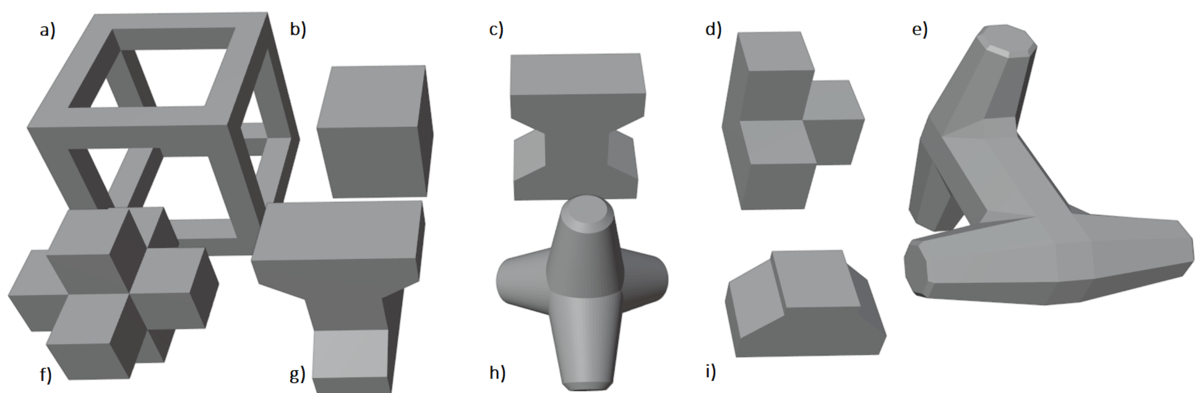


Figura 39. Bloques rompeolas generados mediante el código desarrollado: a) Cob, b) Cubo, c) Akmon tipo cruz plana, d) Tripod, e) Dolos, f) Hexaleg, g) Akmon tipo estándar, h) Tetrápodo con resolución 67 y i) mitad de Akmon.

Conclusiones

La parametrización permite automatizar la creación de piezas una vez creadas, de forma que se ahorra tiempo en la generación del modelo, otro aspecto positivo de *Blender*, al ser, una vez se domina la creación de mallas, un proceso relativamente sencillo. Pudiendo realizarse mediante las herramientas internas que tiene *Blender* la parametrización de piezas de complicada parametrización como es el caso del tetrápodo.

6. Otras herramientas

Además de las herramientas propias de *Blender* y de las que un usuario con conocimientos de programación puede desarrollar para sí mismo, existen un gran número de complementos creados por otros desarrolladores que pueden ser gratuitos o de pago. Algunos de estos complementos que tienen utilidad directa en la ingeniería de caminos, además de los ya mencionados son *Blender GIS* y *CAD Transform*, se mencionan a continuación:

- *Blender BIM*: es un complemento gratuito, que permite utilizar el programa como *software* BIM, permitiendo el uso de archivos .ifc.
- Archimesh y *Archipad*: estos complementos permiten generar de forma sencilla elementos arquitectónicos: paredes, puertas, ventanas, tejados, etc.
- *Flip fluids*: este complemento es de pago, si bien su coste no es muy alto (76 dólares en el momento de escribir este documento por una versión perpetua con actualizaciones gratuitas de por vida). Permite realizar simulaciones de fluidos de alta fidelidad que son realmente impresionantes a través de un motor que utiliza la técnica de simulación FLIP.
- *Freestyle SVG*: permite crear archivos svg de los modelos generados, lo cual puede ser utilizado para crear planos.

Es posible también crear estructuras en *Blender* y exportarlas para mediante *FreeCad* (herramienta de diseño gratuita) realizar cálculos estructurales.

7. Conclusiones

Con este trabajo se ha podido comprobar la viabilidad de utilizar *Blender*, un programa gratuito de diseño tridimensional, como herramienta de modelado en la ingeniería de caminos, canales y puertos. Este *software* permite obtener modelos tanto 3D como 2D. Como conclusiones se pueden mencionar a nivel de programación:

- Se comprueba la posibilidad de usar fuentes de datos muy variadas, mediante el soporte directo de las mismas o con la creación de programas para la importación y exportación de estas, permitiéndose así la compatibilidad con otros programas.
- Con la creación de un complemento y la capacidad de utilizar complementos de terceros se demuestra la capacidad prácticamente ilimitada de personalización del entorno de trabajo.
- En general los códigos demuestran la capacidad de automatización de procesos de forma sencilla que tiene un programa como *Blender* mediante un lenguaje de programación ampliamente utilizado como *Python* o mediante la programación visual. El uso de *Python*, permite también el acceso a multitud de librerías reduciendo los costes de la programación puesto que en ocasiones la herramienta o parte de la herramienta que se quiere realizar ha sido ya creada.
- Para el aprendizaje del programa y la *API* se ha utilizado como fuente de información la documentación de *Blender*, y conocidos fórums de programación como S*tack Overflow,* lo que demuestra que existen en internet fuentes de información de calidad, actualizadas y gratuitas para *Blender*, así como un gran número de usuarios. Siendo este último aspecto indispensable para decidir utilizar un programa.

En cuanto a sus capacidades específicas para a ingeniería:

- Se ha podido comprobar también la capacidad de introducir elementos propios de la ingeniería de caminos como modelos del terreno de grandes dimensiones, y las capacidades que tiene *Blender* para reducir los costes computacionales del modelo.
- Además de los modelos del terreno se ha demostrado la posibilidad de crear elementos constructivos naturales e imperfectos como las rocas.
- Se comprueba también la posibilidad de obtener resultados fotorrealistas.
- Debe tenerse en cuenta que la curva de aprendizaje en un programa de diseño como *Blender* no es sencilla al ser generados los modelos de forma diferente a la herramienta *CAD* tradicional, y por incluir este una gran variedad de herramientas que en un principio pueden parecer no ser de utilidad para el ingeniero. Sin embargo, con respecto a esto último, la posibilidad de generar películas, modelos en movimientos, simulaciones, texturas o simplemente renderizados amplían las capacidades del ingeniero a la hora de captar clientes, y ayudan a una mejor comunicación y comprensión de los proyectos, siendo por tanto un aspecto positivo para aquel que quiera aportar mejoras a sus proyectos.

Bibliografía

[1] Ton Roosendaal, et al. *Blender, 2000-2021. https://www.blender.org/*

[2] Blender Documentation Team. *The Blender 2.92 Manual. https://docs.blender.org/manual/en/2.92/*

[3] Martin Isenburg and Jonathan Shewchuk. *LAStools. https://rapidlasso.com/lastools/*

[4] Jones E, Oliphant E, Peterson P, et al. *SciPy: Open Source Scientific Tools for Python*, 2001-2021, *http://www.scipy.org/*

[5] Clark A, and contributors. *Pilow (PIL Fork)*, 2010-2021, *https://pillow.readthedocs.io/*

[6] Instituto Geográfico Nacional. *IGN. "http://centrodedescargas.cnig.es/CentroDescargas/index.jsp"*

[7] Jones E, Oliphant E, Peterson P, et al. *NumPy: Open Source Scientific Tools for Python*, 1995-2021, *https://numpy.org/*

ANEJO I: Caminos

Dependencias externas:

- Scipy
- Pillow Pil
- numpy

terrain_creator.py

```python
from scipy.spatial import Delaunay
from mathutils import Vector
from enum import Enum
from PIL import (
  Image,
  TiffTags
)
import numpy as np
import bpy
import os


Image.MAX_IMAGE_PIXELS = None # Cancel: Decompression bomb error.


UNIT = {
  0: 0.0, 9001: 1, 9002: 0.3048, 9003: 0.304800609601219, 9005: 0.3047972654,
  9014: 1.8288, 9030: 1852, 9031: 1.0000135965, 9033: 20.1168402336805,
  9034: 0.201168402336805, 9035: 1609.34721869444, 9036: 1000.0,
  9037: 0.9143917962, 9038: 20.1166195164, 9039: 0.201166195164,
  9040: 0.914398414616029, 9041: 0.304799471538676, 9042: 20.1167651215526,
  9043: 0.201167651215526, 9050: 0.9143992, 9051: 0.304799733333333,
  9052: 20.1167824, 9053: 0.201167824, 9060: 0.914399204289812,
  9061: 0.304799734763271, 9062: 20.1167824943759, 9063: 0.201167824943759,
  9070: 0.304800833333333, 9080: 0.304799510248147, 9081: 0.30479841,
  9082: 0.3047996, 9083: 0.3047995, 9084: 0.914398530744441, 9085: 0.91439523,
  9086: 0.9143988, 9087: 0.9143985, 9093: 1609.344, 9094: 0.304799710181509,
  9095: 0.3048007491, 9096: 0.9144, 9097: 20.1168, 9098: 0.201168,
  9099: 0.914398, 9101: 1.0, 9102: 0.0174532925199433, 9103: 0.000290888208665721,
  9104: 4.84813681109535e-006, 9105: 0.015707963267949, 9107: 0, 9108: 0,
  9109: 1e-006, 9110: 0, 9111: 0, 9112: 0.000157079632679489,
  9113: 1.5707963267949e-006, 9114: 0.000981747704246809, 9115: 0,
  9116: 0, 9117: 0, 9118: 0, 9119: 0, 9120: 0, 9121: 0, 9122: 0.0174532925199433,
  9300: 0.304799333333333, 9301: 20.116756, 9302: 0.20116756
}

LENGHT_MULTIPLES = {
  'METRIC': {
    'ADAPTATIVE': (None, ''),
    'KILOMETERS': (1 / 1000, 'km'),
    'METERS': (1, 'm'),
    'CENTIMETERS': (100, 'cm'),
    'MILIMETERS': (1000, 'mm'),
    'MICROMETERS': (1000000, '\u00b5m')
  },
  'IMPERIAL': {
```

```
47      'ADAPTATIVE': (None, ''),
48      'MILES': (1 / 5280, 'mi'),
49      'FEET': (1, '\''),
50      'INCHES': (12, '"'),
51      'THOU': (12000, 'thou')
52    }
53  }
54
55  COLOR_RAMPS = {
56    'RGB': [
57      {'position': 0.000, 'color': (1.000, 0.200, 0.300, 1.000)},
58      {'position': 0.500, 'color': (0.300, 0.200, 1.000, 1.000)},
59      {'position': 1.000, 'color': (0.300, 1.000, 0.100, 1.000)}
60    ],
61    'ELEVATION': [
62      {'position': 0.000, 'color': (0.094, 0.443, 0.027, 1.000)},
63      {'position': 0.166, 'color': (0.427, 0.756, 0.247, 1.000)},
64      {'position': 0.333, 'color': (0.521, 0.921, 0.117, 1.000)},
65      {'position': 0.500, 'color': (0.949, 0.843, 0.552, 1.000)},
66      {'position': 0.666, 'color': (0.682, 0.533, 0.274, 1.000)},
67      {'position': 0.833, 'color': (0.458, 0.266, 0.086, 1.000)},
68      {'position': 1.000, 'color': (1.000, 0.933, 0.925, 1.000)}
69    ]
70  }
71
72
73  class OriginPosition(Enum):
74    UPPER_LEFT_CELL_UPPER_LEFT_CORNER = 1
75    UPPER_LEFT_CENTER = 2
76    LOWER_LEFT_CELL_LOWER_LEFT_CORNER = 3
77    LOWER_LEFT_CENTER = 4
78
79
80  def lenght_conversion(lenght_epsg_code):
81    scene_unit_system = bpy.context.scene.unit_settings.system
82    scene_length_units = bpy.context.scene.unit_settings.length_unit
83
84    if scene_unit_system == 'METRIC':
85      return UNIT[lenght_epsg_code] * \
86        LENGHT_MULTIPLES[scene_unit_system][scene_length_units][0]
87
88    elif scene_unit_system == 'IMPERIAL':
89      return UNIT[lenght_epsg_code] * 1 / 0.3048 * \
90        LENGHT_MULTIPLES[scene_unit_system][scene_length_units][0]
91
92
93  class BBox():
94    def __init__(self, obj):
95      b_box = [obj.matrix_world @ Vector(p) for p in obj.bound_box]
96      self.x_min = min([p[0] for p in b_box])
97      self.x_max = max([p[0] for p in b_box])
98      self.y_min = min([p[1] for p in b_box])
99      self.y_max = max([p[1] for p in b_box])
100     self.z_min = min([p[2] for p in b_box])
101     self.z_max = max([p[2] for p in b_box])
102
103
104   def sizes(self):
105     return (
```

```
106         self.x_max - self.x_min,
107         self.y_max - self.y_min,
108         self.z_max - self.z_min
109       )
110
111     def __repr__(self):
112       return """x_min = {0}
113   x_max = {1}
114   y_min = {2}
115   y_max = {3}
116   z_min = {4}
117   z_max = {5}\n""".format(self.x_min, self.x_max, self.y_min, self.y_max,
118   self.z_min, self.z_max)
119
120
121   def reposition_3D_view(b_box, orthographic=True):
122     max_size = int(max(b_box.sizes()))
123     for area in bpy.context.screen.areas:
124       if area.type == 'VIEW_3D':
125         for space in area.spaces:
126           if space.type == 'VIEW_3D':
127             if orthographic:
128               space.region_3d.view_perspective = 'ORTHO'
129             if max_size > 100:
130               space.clip_start = 1
131             if max_size > 1000:
132               space.clip_start = 10
133             space.clip_end = max_size * 5
134
135             ctx = bpy.context.copy()
136             ctx['area'] = area
137             ctx['region'] = area.regions[-1]
138             a = bpy.ops.view3d.view_selected(ctx)
139
140
141   class Dem():
142     def __init__(self, filepath, mode='DELAUNAY', name='DEM', col_name='DEMs',
143         origin_zero=False, smooth=False):
144       self.filepath = filepath
145       self.mode = mode
146       self.name = name
147       self.col_name = col_name
148       self.origin_zero = origin_zero
149       self.smooth = smooth
150
151     def _voronoi_blendify(self, points, reposition=True):
152       bpy.ops.object.select_all(action='DESELECT')
153       mesh = bpy.data.meshes.new(self.name)
154       obj = bpy.data.objects.new(mesh.name, mesh)
155       if self.col_name in [c.name for c in bpy.data.collections]:
156         col = bpy.data.collections.get(self.col_name)
157       else:
158         col = bpy.data.collections.new(self.col_name)
159         bpy.context.scene.collection.children.link(col)
160       col.objects.link(obj)
161       bpy.context.view_layer.objects.active = obj
162       obj.select_set(True)
163       mesh.from_pydata(points, [], Delaunay(points[:, 0:-1]).simplices.tolist())
164
```

```
165        if reposition:
166          reposition_3D_view(BBox(obj))
167
168        if self.smooth:
169          bpy.ops.object.shade_smooth()
170
171      def _modifier_blendify(self, num_cols, num_rows, cell_size_x, cell_size_y,
172          origin_coords, origin_position, reposition=True):
173        bpy.ops.object.select_all(action='DESELECT')
174
175        # Mesh creation.
176        mesh = bpy.data.meshes.new(self.name)
177        if self.origin_zero:
178          x_2 = num_cols / 2 * cell_size_x
179          y_2 = num_rows / 2 * cell_size_y
180          verts = np.array((
181            (-x_2, -y_2, 0),
182            (x_2, -y_2, 0),
183            (x_2, y_2, 0),
184            (-x_2, y_2, 0)))
185        else:
186          x = num_cols * cell_size_x
187          y = num_rows * cell_size_y
188          verts = np.array((
189            (self.origin_coords[0], self.origin_coords[1] - y, 0),
190            (self.origin_coords[0] + x, self.origin_coords[1] - y, 0),
191            (self.origin_coords[0] + x, self.origin_coords[1], 0),
192            (self.origin_coords[0], self.origin_coords[1], 0)))
193        faces = [[0, 1, 2, 3]]
194        mesh.from_pydata(verts, [], faces)
195
196        # Object creation.
197        bpy.ops.object.select_all(action='DESELECT')
198        obj = bpy.data.objects.new(self.name, mesh)
199        if self.col_name in [c.name for c in bpy.data.collections]:
200          col = bpy.data.collections.get(self.col_name)
201        else:
202          col = bpy.data.collections.new(self.col_name)
203          bpy.context.scene.collection.children.link(col)
204        col.objects.link(obj)
205        bpy.context.view_layer.objects.active = obj
206        obj.select_set(True)
207
208        # Create UV layer.
209        uv_map = mesh.uv_layers.new(name='UvMap')
210        uv_map.data[0].uv = [
211          cell_size_x / (num_cols * cell_size_x * 2),
212          cell_size_y / (num_rows * cell_size_y * 2)]
213        uv_map.data[1].uv = [
214          1 - cell_size_x / (num_cols * cell_size_x * 2),
215          cell_size_y / (num_rows * cell_size_y * 2)]
216        uv_map.data[2].uv = [
217          1 - cell_size_x / (num_cols * cell_size_x * 2),
218          1-cell_size_y / (num_rows * cell_size_y * 2)]
219        uv_map.data[3].uv = [
220          cell_size_x / (num_cols * cell_size_x * 2),
221          1-cell_size_y / (num_rows * cell_size_y * 2)]
222
223        # Create subdivision surface modifier.
```

```
224        subivision_surface_mod = obj.modifiers.new('SUBDIVISION 1', type='SUBSURF')
225        subivision_surface_mod.subdivision_type = 'SIMPLE'
226        subivision_surface_mod.levels = 1
227        subivision_surface_mod.render_levels = 6
228
229        subivision_surface_mod = obj.modifiers.new('SUBDIBISION 2', type='SUBSURF')
230        subivision_surface_mod.subdivision_type = 'SIMPLE'
231        subivision_surface_mod.levels = 6
232        subivision_surface_mod.render_levels = 6
233
234        # Create displace modifier.
235        dem_image = bpy.data.images.load(self.filepath)
236        dem_image.name = 'DEM IMAGE'
237        dem_image.colorspace_settings.name = 'Non-Color'
238
239        dem_texture = bpy.data.textures.new('DEM HEIGHT TEXTURE', type='IMAGE')
240        dem_texture.image = dem_image
241        dem_texture.use_interpolation = True
242        dem_texture.extension = 'CLIP'
243        dem_texture.use_clamp = False
244
245        displace_mod = obj.modifiers.new('DISPLACE', type='DISPLACE')
246        displace_mod.texture = dem_texture
247        displace_mod.texture_coords = 'UV'
248        displace_mod.uv_layer = uv_map.name
249        displace_mod.mid_level = 0
250
251        if reposition:
252          reposition_3D_view(BBox(obj))
253
254        if self.smooth:
255          bpy.ops.object.shade_smooth()
256
257      def set_rc_material(self, color_ramp='RGB'):
258        color_ramp = COLOR_RAMPS[color_ramp]
259        obj = bpy.context.view_layer.objects.active
260        b_box = BBox(obj)
261
262        # Create Normalize nodes group.
263        normalize_node_group = bpy.data.node_groups.new('Normalize', 'ShaderNodeTree')
264        normalize_node_group.nodes.new('NodeGroupInput')
265        normalize_node_group.inputs.new('NodeSocketFloat','val')
266        normalize_node_group.inputs.new('NodeSocketFloat','min')
267        normalize_node_group.inputs.new('NodeSocketFloat','max')
268        normalize_node_group.nodes.new('NodeGroupOutput')
269        normalize_node_group.outputs.new('NodeSocketFloat', 'val')
270
271        normalize_nodes = normalize_node_group.nodes
272        normalize_nodes.new('ShaderNodeMath') # Math -> First subtract.
273        normalize_nodes.new('ShaderNodeMath') # Math.001 -> Second subtract.
274        normalize_nodes.new('ShaderNodeMath') # Math.002 -> Divide.
275
276        normalize_nodes['Math'].operation = 'SUBTRACT'
277        normalize_nodes['Math.001'].operation = 'SUBTRACT'
278        normalize_nodes['Math.002'].operation = 'DIVIDE'
279
280        normalize_nodes['Group Input'].location = (-340.0, 40.0)
281        normalize_nodes['Math'].location = (-160.0, 160.0)
282        normalize_nodes['Math.001'].location = (-160.0, -20.0)
```

```
283         normalize_nodes['Math.002'].location = (20.0, 60.0)
284         normalize_nodes['Group Output'].location = (200.0, 20.0)
285 
286         normalize_node_group.links.new(
287           normalize_nodes['Group Input'].outputs['val'],
288           normalize_nodes['Math'].inputs[0])
289         normalize_node_group.links.new(
290           normalize_nodes['Group Input'].outputs['min'],
291           normalize_nodes['Math'].inputs[1])
292         normalize_node_group.links.new(
293           normalize_nodes['Group Input'].outputs['max'],
294           normalize_nodes['Math.001'].inputs[0])
295         normalize_node_group.links.new(
296           normalize_nodes['Group Input'].outputs['min'],
297           normalize_nodes['Math.001'].inputs[1])
298         normalize_node_group.links.new(
299           normalize_nodes['Math'].outputs['Value'],
300           normalize_nodes['Math.002'].inputs[0])
301         normalize_node_group.links.new(
302           normalize_nodes['Math.001'].outputs['Value'],
303           normalize_nodes['Math.002'].inputs[1])
304         normalize_node_group.links.new(
305           normalize_nodes['Math.002'].outputs['Value'],
306           normalize_nodes['Group Output'].inputs['val'])
307 
308         # Create material.
309         mat = bpy.data.materials.new(name='MAT')
310         obj.data.materials.append(mat)
311 
312         mat.use_nodes = True
313         mat_nodes = mat.node_tree.nodes
314 
315         # Creating nodes
316         mat_nodes.remove(mat_nodes['Principled BSDF'])
317         mat_nodes.new('ShaderNodeNewGeometry')
318         mat_nodes.new('ShaderNodeValue')
319         mat_nodes['Value'].label = 'z_min'
320         mat_nodes['Value'].outputs[0].default_value = b_box.z_min
321         mat_nodes.new('ShaderNodeValue')
322         mat_nodes['Value.001'].label = 'z_max'
323         mat_nodes['Value.001'].outputs[0].default_value = b_box.z_max
324         mat_nodes.new('ShaderNodeGroup')
325         mat_nodes['Group'].node_tree = bpy.data.node_groups[normalize_node_group.name]
326         mat_nodes.new('ShaderNodeSeparateXYZ')
327         mat_nodes.new('ShaderNodeValToRGB')
328         mat_nodes.new('ShaderNodeBsdfDiffuse')
329 
330         # Locating nodes
331         mat_nodes['Geometry'].location = (-420.0, 100.0)
332         mat_nodes['Separate XYZ'].location = (-240.0, 140.0)
333         mat_nodes['Value'].location = (-240.0, -20.0)
334         mat_nodes['Value.001'].location = (-240.0, -120.0)
335         mat_nodes['ColorRamp'].location = (120.0, 100.0)
336         mat_nodes['Group'].location = (-60.0, 80.0)
337         mat_nodes['Diffuse BSDF'].location = (400.0, 60.0)
338         mat_nodes['Material Output'].location = (600.0, 60.0)
339 
340         # Create Links
341         mat.node_tree.links.new(
```

```
342         mat_nodes['Geometry'].outputs['Position'],
343         mat_nodes['Separate XYZ'].inputs['Vector'])
344       mat.node_tree.links.new(
345         mat_nodes['Separate XYZ'].outputs['Z'],
346         mat_nodes['Group'].inputs['val'])
347       mat.node_tree.links.new(
348         mat_nodes['Value'].outputs['Value'],
349         mat_nodes['Group'].inputs['min'])
350       mat.node_tree.links.new(
351         mat_nodes['Value.001'].outputs['Value'],
352         mat_nodes['Group'].inputs['max'])
353       mat.node_tree.links.new(
354         mat_nodes['Group'].outputs['val'],
355         mat_nodes['ColorRamp'].inputs['Fac'])
356       mat.node_tree.links.new(
357         mat_nodes['ColorRamp'].outputs['Color'],
358         mat_nodes['Diffuse BSDF'].inputs['Color'])
359       mat.node_tree.links.new(
360         mat_nodes['Diffuse BSDF'].outputs['BSDF'],
361         mat_nodes['Material Output'].inputs['Surface'])
362 
363       # Create colour ramp.
364       for _ in range(len(mat_nodes['ColorRamp'].color_ramp.elements) - 1):
365         mat_nodes['ColorRamp'].color_ramp.elements.remove(
366           mat_nodes['ColorRamp'].color_ramp.elements[-1])
367 
368       for i in range(len(color_ramp)):
369         if color_ramp[i]['position'] != 0:
370           mat_nodes['ColorRamp'].color_ramp.elements.new(color_ramp[i]['position'])
371         mat_nodes['ColorRamp'].color_ramp.elements[i].color = color_ramp[i]['color']
372 
373     def set_o_material(self, filepath):
374       # Create material.
375       mat = bpy.data.materials.new(name='DEM ORTHOPHOTO')
376 
377       mat.use_nodes = True
378       mat_nodes = mat.node_tree.nodes
379 
380       # Create material nodes.
381       mat_nodes.new('ShaderNodeTexImage')
382 
383       # Locate material nodes.
384       mat_nodes['Image Texture'].location = (-420.0, 120.0)
385       mat_nodes['Principled BSDF'].location = (-120.0, 300.0)
386       mat_nodes['Material Output'].location = (180.0, 60.0)
387 
388       # Load material images.
389       img = bpy.data.images.load(filepath)
390       img.name = 'DEM ORTHOPHOTO'
391 
392       # Specify material node parameters.
393       mat_nodes['Image Texture'].image = img
394       mat_nodes['Principled BSDF'].inputs[4].default_value = 0.4 # Metallic.
395       mat_nodes['Principled BSDF'].inputs[5].default_value = 0.0 # Specular
396       mat_nodes['Principled BSDF'].inputs[7].default_value = 1.0 # Roughness.
397 
398       # Create material links.
399       mat.node_tree.links.new(
400         mat_nodes['Image Texture'].outputs['Color'],
```

```
401          mat_nodes['Principled BSDF'].inputs['Base Color'])
402        mat.node_tree.links.new(
403          mat_nodes['Principled BSDF'].outputs['BSDF'],
404          mat_nodes['Material Output'].inputs['Surface'])
405
406        # Assign material to object.
407        obj = bpy.context.view_layer.objects.active
408        obj.data.materials.append(mat)
409
410    class Tiff(Dem):
411      def __init__(self, filepath, mode='DELAUNAY', name='DEM', col_name='DEMs',
412          origin_zero=False, smooth=False):
413        super().__init__(filepath, mode, name, col_name, origin_zero)
414        self.image = Image.open(self.filepath)
415        self.metadata_dict = {
416          key : self.image.tag[key] for key in self.image.tag.keys()}
417        self.tags_dict = {
418          key : TiffTags.TAGS[key] for key in self.image.tag.keys()}
419        self.num_cols = self.metadata_dict[256][0]
420        self.num_rows = self.metadata_dict[257][0]
421        self.cell_size_x = self.metadata_dict[33550][0]
422        self.cell_size_y = self.metadata_dict[33550][1]
423        self.origin_coords = self.metadata_dict[33922][3:5]
424        self.coordinate_system_type = self.metadata_dict[34735][
425          self.metadata_dict[34735].index(1024) + 3]
426        self.origin_position = OriginPosition(self.metadata_dict[34735][
427          self.metadata_dict[34735].index(1025) + 3])
428        self.angle_epsg_code = self.metadata_dict[34735][
429          self.metadata_dict[34735].index(2054) + 3]
430        self.coordinate_system_code = self.metadata_dict[34735][
431          self.metadata_dict[34735].index(3072) + 3]
432        self.lenght_epsg_code = self.metadata_dict[34735][
433          self.metadata_dict[34735].index(3076) + 3]
434
435      def _get_points(self):
436        if self.origin_zero:
437          x_2 = self.num_cols / 2 * self.cell_size_x
438          y_2 = self.num_rows / 2 * self.cell_size_y
439          x = np.tile(np.arange(
440              -x_2,
441              self.cell_size_x * self.num_cols -x_2,
442              self.cell_size_x),
443            self.num_rows)
444          y = np.tile(np.arange(
445              y_2,
446              -self.cell_size_y * self.num_rows + y_2,
447              -self.cell_size_y),
448            self.num_cols).reshape(self.num_cols, self.num_rows).T.flatten()
449        else:
450          x = np.tile(np.arange(
451              self.origin_coords[0],
452              self.origin_coords[0] + self.cell_size_x * self.num_cols,
453              self.cell_size_x),
454            self.num_rows)
455          y = np.tile(np.arange(
456              self.origin_coords[1],
457              self.origin_coords[1] - self.cell_size_y * self.num_rows,
458              -self.cell_size_y),
459            self.num_cols).reshape(self.num_cols, self.num_rows).T.flatten()
```

```
460     z = np.asarray(self.image).reshape(self.image.width * self.image.height)
461
462     points = np.column_stack((x, y, z))
463     if self.origin_position == OriginPosition.UPPER_LEFT_CELL_UPPER_LEFT_CORNER:
464       points[:, 0] += self.cell_size_x / 2
465       points[:, 1] -= self.cell_size_y / 2
466     points *= lenght_conversion(self.lenght_epsg_code)
467
468     return points
469
470   def blendify(self, reposition=True):
471     if self.mode == 'DELAUNAY':
472       self._voronoi_blendify(self._get_points(), reposition)
473     else:
474       self._modifier_blendify(self.num_cols, self.num_rows, self.cell_size_x,
475         self.cell_size_y, self.origin_coords, self.origin_position, reposition)
476
477   def __repr__(self):
478     return """Number of columns      -> {0}
479 Number of rows         -> {1}
480 Cell size X            -> {2}
481 Cell size Y            -> {3}
482 Origin coordinates     -> {4}
483 Coordinate system type -> {5}
484 Origion Position       -> {6}
485 Angle EPSG code        -> {7}
486 Coordinate system code -> {8}
487 Length EPSG code       -> {9}\n""".format(
488       self.num_cols,
489       self.num_rows,
490       self.cell_size_x,
491       self.cell_size_y,
492       self.origin_coords,
493       self.coordinate_system_type,
494       self.origin_position,
495       self.angle_epsg_code,
496       self.coordinate_system_code,
497       self.lenght_epsg_code
498     )
499
500
501 class Asc(Dem):
502   def __init__(self, filepath, mode='DELAUNAY', name='DEM', col_name='DEMs',
503     delimiter=' ', lenght_epsg_code=9001, origin_zero=False, smooth=False):
504     super().__init__(filepath, mode, name, col_name, origin_zero)
505     with open(filepath, 'r') as f:
506       self.origin_coords = []
507       try:
508         self.num_cols = int(f.readline().split()[1])
509         self.num_rows = int(f.readline().split()[1])
510         line = f.readline().split()
511         if line[0].lower() in ('xllcenter', 'xll_center'):
512           self.origin_position = OriginPosition(4)
513           self.origin_coords.append(float(line[1]))
514           self.origin_coords.append(float(f.readline().split()[1]))
515         else:
516           self.origin_position = OriginPosition(3)
517           self.origin_coords.append(float(line[1]))
518           self.origin_coords.append(float(f.readline().split()[1]))
```

```
519          self.cell_size = float(f.readline().split()[1])
520          self.nodata_value = float(f.readline().split()[1])
521        except:
522          print('Error')
523          return
524        self.delimiter = delimiter
525        self.lenght_epsg_code = lenght_epsg_code
526
527    def _get_points(self):
528      self.image = np.loadtxt(self.filepath, dtype='float32',
529  delimiter=self.delimiter, skiprows=6)
530
531      if self.origin_zero:
532        x_2 = self.num_cols / 2 * self.cell_size
533        y_2 = self.num_rows / 2 * self.cell_size
534        x = np.tile(np.arange(
535            -x_2, self.cell_size * self.num_cols -x_2, self.cell_size),
536          self.num_rows)
537        y = np.tile(np.arange(
538            y_2, -self.cell_size * self.num_rows + y_2, -self.cell_size),
539          self.num_cols).reshape(self.num_cols, self.num_rows).T.flatten()
540      else:
541        x = np.tile(np.arange(
542            self.origin_coords[0] ,
543            self.origin_coords[0] + self.cell_size * self.num_cols,
544            self.cell_size),
545          self.num_rows)
546        y = np.tile(np.arange(
547            self.origin_coords[1] + self.cell_size * self.num_rows,
548            self.origin_coords[1],
549            -self.cell_size),
550          self.num_cols).reshape(self.num_cols, self.num_rows).T.flatten()
551      z = self.image
552      z.flatten()
553
554      points = np.column_stack((x, y, z.flatten()))
555      if self.origin_position == OriginPosition.LOWER_LEFT_CELL_LOWER_LEFT_CORNER:
556        points[:, 0] += self.cell_size / 2
557        points[:, 1] -= self.cell_size / 2
558      points *= lenght_conversion(self.lenght_epsg_code)
559
560      return points
561
562    def blendify(self, reposition=True):
563      if self.mode == 'DELAUNAY':
564        self._voronoi_blendify(self._get_points(), reposition)
565      else:
566        pass
567
568    def __repr__(self):
569      return """Number of columns  ->{0}
570  Number of rows     ->{1}
571  Cell size          ->{2}
572  Origin coordinates ->{3}
573  Origin position    ->{4}
574  No data value      ->{5}\n""".format(
575        self.num_cols,
576        self.num_rows,
577        self.cell_size,
```

```
578         self.origin_coords,
579         self.origin_position,
580         self.nodata_value)
581 
582 
583 class Xyz(Dem):
584   def __inti__(self, filepath, name='DEM', col_name='DEMs', origin_zero=False,
585       smooth=False):
586     super().__init__(filepath, name, col_name, origin_zero)
587 
588   def _get_points(self, header=False, delimiter=' ', lenght_epsg_code=9001,
589       origin_zero=False):
590     if header:
591       points = np.loadtxt(
592         self.filepath, dtype='float32', delimiter=delimiter, skiprows=1)
593     else:
594       points = np.loadtxt(
595         self.filepath, dtype='float32', delimiter=delimiter, skiprows=0)
596 
597     if self.origin_zero:
598       min_x = np.min(points[:, 0])
599       max_x = np.max(points[:, 0])
600       if min_x > 0:
601         d_x = max_x - min_x
602         points[:, 0] -= min_x + d_x / 2
603       else:
604         if max_x > 0:
605           d_x = max_x + abs(min_x)
606         else:
607           d_x = max(abs(min_x), abs(max_x)) - min(abs(min_x), abs(max_x))
608         points[:, 0] += min_x - d_x / 2
609       min_y = np.min(points[:, 1])
610       max_y = np.max(points[:, 1])
611       if min_y > 0:
612         d_y = max_y - min_y
613         points[:, 1] -= min_y + d_y / 2
614       else:
615         if max_y > 0:
616           d_y = max_y + abs(min_y)
617         else:
618           d_y = max(abs(min_y), abs(max_y)) - min(abs(min_y), abs(max_y))
619         points[:, 1] += min_y - d_y / 2
620     points *= lenght_conversion(lenght_epsg_code)
621 
622     return points
623 
624   def blendify(self, reposition=True):
625     self._voronoi_blendify(self._get_points(reposition))
626 
627 
628 if __name__ == '__main__':
629   pass
```

Dependencias externas:

- Pillow Pil
- numpy

longitudinal.py

```
630  from PIL import Image, TiffTags
631  from math import pi as PI
632  from math import sqrt
633  from enum import Enum
634  from copy import copy
635  import numpy as np
636  import bpy
637  import os
638
639
640  class OriginPosition(Enum):
641    UPPER_LEFT_CELL_UPPER_LEFT_CORNER = 1
642    UPPER_LEFT_CENTER = 2
643    LOWER_LEFT_CELL_LOWER_LEFT_CORNER = 3
644    LOWER_LEFT_CENTER = 4
645
646
647  class Point2D():
648    def __init__(self, x, y):
649      self.x = x
650      self.y = y
651
652    @ classmethod
653    def from_array(cls, a):
654      return cls(a[0], a[1])
655
656    @ classmethod
657    def from_vector2D(cls, v, k):
658      return cls(v.origin.x + v.x * k , v.origin.y + v.y * k)
659
660    def __copy__(self):
661      return type(self)(self.x, self.y)
662
663    def __repr__(self):
664      return "{x:<10.4f} {y:<10.4f}".format(x=self.x, y=self.y)
665
666
667  class Vector2D(Point2D):
668    def __init__(self, x, y, origin=Point2D(0, 0)):
669      super().__init__(x, y)
670      self.origin = origin
671
672    @classmethod
673    def from_points(cls, p1, p2, set_origin=True):
674      if set_origin:
675        return cls(p2.x - p1.x, p2.y - p1.y, p1)
676      else:
```

```
677         return cls(p2.x - p1.x, p2.y - p1.y)
678
679     def module(self):
680       return sqrt(self.x**2 + self.y**2)
681
682     def unitary(self):
683       m = self.module()
684       return Vector2D(self.x / m,  self.y / m, self.origin)
685
686
687   class Profile():
688     def __init__(self, points, raster, initial_x=0):
689       self.num_points = len(points)
690       self.initial_x = initial_x
691       self.x = np.zeros(self.num_points) # Distance.
692       self.y = np.zeros(self.num_points) # Height.
693       metadata_dict = {key : raster.tag[key] for key in raster.tag.keys()}
694       self.raster_cell_size_x = metadata_dict[33550][0]
695       self.raster_cell_size_y = metadata_dict[33550][1]
696       self.raster_origin = Point2D.from_array(metadata_dict[33922][3:5])
697       self.y[0] = raster.getpixel(
698         ((points[0].x - self.raster_origin.x) / self.raster_cell_size_x,
699         (self.raster_origin.y - points[0].y) / self.raster_cell_size_y))
700       for i in range(1, self.num_points):
701         self.x[i] = self.x[i - 1] + Vector2D.from_points(
702           points[i - 1], points[i]).module()
703         self.y[i] = raster.getpixel(
704           ((points[i].x - self.raster_origin.x) / self.raster_cell_size_x,
705           (self.raster_origin.y - points[i].y) / self.raster_cell_size_y))
706
707     @classmethod
708     def by_paths(cls, csv_filepath, raster_filepath, create_new_points=False):
709       csv = np.loadtxt(csv_filepath, delimiter=',', skiprows=1, usecols=(1, 2, 3))
710
711       raster = Image.open(raster_filepath)
712       metadata_dict = {key : raster.tag[key] for key in raster.tag.keys()}
713       cell_size_x = metadata_dict[33550][0]
714       cell_size_y = metadata_dict[33550][1]
715       min_cell_size = min(cell_size_x, cell_size_y)
716
717       points = []
718
719       last_point = Point2D.from_array(csv[0, :-1])
720       points.append(last_point)
721       for i in range(1, csv.shape[0]):
722         p = Point2D.from_array(csv[i, :-1])
723         v = Vector2D.from_points(last_point, p)
724         m = v.module()
725         if m > min_cell_size:
726           num_new_points = int(m / min_cell_size) + 1
727           step = m / num_new_points
728         else:
729           num_new_points = 0
730
731         if num_new_points and create_new_points:
732           vu = v.unitary()
733           for j in range(num_new_points):
734             points.append(Point2D.from_vector2D(vu, (j + 1) * step))
735         else:
```

```
736         points.append(p)
737       last_point = copy(p)
738 
739     return cls(points, raster)
740 
741 
742 class TickProperty():
743   def __init__(self, axis, spacing='DEFAULT', axis_spacing='DEFAULT',
744       scale='DEFAULT', overflow_lenght='DEFAULT', color=None):
745     if isinstance(spacing, str):
746       if axis.upper() == 'X':
747         self.spacing = 250
748       else:
749         self.spacing = 10
750     else:
751       self.spacing = spacing
752     if isinstance(axis_spacing, str):
753       if axis_spacing.upper() == 'DEFAULT':
754         self.axis_spacing = 5
755       elif axis_spacing.upper() == 'AUTO':
756         self.axis_spacing = 'AUTO'
757     else:
758       self.axis_spacing = axis_spacing
759     if isinstance(scale, str):
760       if scale.upper() == 'DEFAULT':
761         self.scale = 10
762       elif scale.upper() == 'AUTO':
763         self.scale = 'AUTO'
764     else:
765       self.scale = scale
766     if isinstance(overflow_lenght, str):
767       if overflow_lenght.upper() == 'DEFAULT':
768         self.overflow_lenght = 2.5
769       elif overflow_lenght.upper() == 'AUTO':
770         self.overflow_lenght = 'AUTO'
771     else:
772       self.overflow_lenght = overflow_lenght
773     self.color = color
774 
775   def _set_axis_spacing(self, x_range, y_range):
776     self.axis_spacing =  min(x_range[1] - x_range[0],
777       y_range[1] - y_range[0]) / 20
778     print('[5] axis_spacing ==', self.axis_spacing)
779 
780   def _set_scale(self, x_range, y_range):
781     self.scale =  min(x_range[1] - x_range[0],
782       y_range[1] - y_range[0]) / 10
783     print('[10] scale ==', self.scale)
784 
785   def _set_overflow_lenght(self, x_range, y_range):
786     self.overflow_lenght =  min(x_range[1] - x_range[0],
787       y_range[1] - y_range[0]) / 40
788     print('[2.5] overflow_lenght ==', self.overflow_lenght)
789 
790   def _revise(self, x_range, y_range):
791     if self.axis_spacing == 'AUTO':
792       self._set_axis_spacing(x_range, y_range)
793     if self.scale == 'AUTO':
794       self._set_scale(x_range, y_range)
```

```
795         if self.overflow_lenght == 'AUTO':
796           self._set_overflow_lenght(x_range, y_range)
797
798
799   class ProfilePloter():
800     def __init__(self, profile, x_ticks_props, y_ticks_props, x_scale=0.1,
801         y_scale=1, first_pk=0, origin=(0, 0)):
802         self.profile = profile
803         self.x_ticks_props = x_ticks_props
804         self.y_ticks_props = y_ticks_props
805         self.x_scale = x_scale
806         self.y_scale = y_scale
807         self.first_pk = first_pk
808         self.origin = origin
809
810         self.x_range = (0, max(self.profile.x))
811         self.y_range = (
812           min(self.profile.y) - 10 if min(self.profile.y) - 10 > 0 else 0,
813           max(self.profile.y) + 10)
814         self.num_x_ticks = abs(abs(
815           self.x_range[1]) - abs(self.x_range[0])) // self.x_ticks_props.spacing + 1
816         self.num_y_ticks = abs(abs(
817           self.y_range[1]) - abs(self.y_range[0])) // self.y_ticks_props.spacing + 1
818         self.x_ticks = np.hstack((
819           self.x_range[0] + np.arange(0, self.num_x_ticks) * \
820           self.x_ticks_props.spacing, self.x_range[1]))
821         self.num_x_ticks += 1
822         self.y_ticks = self.y_range[0] + np.arange(0, self.num_y_ticks) * \
823           self.y_ticks_props.spacing
824
825         self.x_ticks_props._revise(self.x_range, self.y_range)
826         self.y_ticks_props._revise(self.x_range, self.y_range)
827
828     def _get_pk(self, x, num_decimals=2, decimal_separator=','):
829         return '{0:.0f}+{1:0{2}.{3}f}'.format(
830           x // 1000, x % 1000, num_decimals + 4 if num_decimals > 0 else 3,
831           num_decimals).replace('.', decimal_separator)
832
833     def _get_elevation(self, y, num_decimals=2, decimal_separator=','):
834         return '{0:.{1}f}'.format(y, num_decimals).replace('.', decimal_separator)
835
836     def blendify(self):
837         col = bpy.data.collections.new('Profile')
838         bpy.context.scene.collection.children.link(col)
839         col_name = col.name
840
841         fon = bpy.data.fonts.load(r'C:\WINDOWS\Fonts\verdana.ttf')
842
843         # box.
844         cur = bpy.data.curves.new(name='Box', type='CURVE')
845         cur.dimensions = '2D'
846         cur_spline = cur.splines.new('POLY')
847         cur_spline.points.add(3)
848         cur_spline.points[0].co = (self.origin[0], self.origin[1], 0.0, 0.0)
849         cur_spline.points[1].co = (
850           self.origin[0] + (self.x_range[1] - self.x_range[0]) * self.x_scale,
851           self.origin[1], 0.0, 0.0)
852         cur_spline.points[2].co = (
853           self.origin[0] + (self.x_range[1] - self.x_range[0]) * self.x_scale,
```

```
854           self.origin[1] + (self.y_range[1] - self.y_range[0]) * self.y_scale,
855           0.0, 0.0)
856         cur_spline.points[3].co = (
857           self.origin[0], self.origin[1] + (self.y_range[1] - self.y_range[0]) * \
858           self.y_scale, 0.0, 0.0)
859         cur_spline.use_cyclic_u = True
860
861         obj = bpy.data.objects.new('Box', cur)
862         bpy.data.collections[col_name].objects.link(obj)
863         bpy.context.view_layer.objects.active = obj
864
865         # x_ticks.
866         cur = bpy.data.curves.new(name='X Ticks', type='CURVE')
867         cur.dimensions = '2D'
868         for x_tick in self.x_ticks:
869           cur_spline = cur.splines.new('POLY')
870           cur_spline.points.add(1)
871           cur_spline.points[0].co = (
872             self.origin[0] + (x_tick - self.x_range[0]) * self.x_scale,
873             self.origin[1] - self.x_ticks_props.overflow_lenght, 0.0, 0.0)
874           cur_spline.points[1].co = (
875             self.origin[0] + (x_tick - self.x_range[0]) * self.x_scale,
876             self.origin[1] + (self.y_range[1] - self.y_range[0]) * self.y_scale,
877             0.0, 0.0)
878
879         obj = bpy.data.objects.new('X Ticks', cur)
880         bpy.data.collections[col_name].objects.link(obj)
881         bpy.context.view_layer.objects.active = obj
882
883         # x_labels.
884         for x_tick in self.x_ticks:
885           cur = bpy.data.curves.new(type="FONT", name="FF")
886           cur.body = self._get_pk(x_tick + self.first_pk)
887           cur.font = fon
888           cur.align_x = 'RIGHT'
889           cur.align_y = 'CENTER'
890           obj = bpy.data.objects.new(name="Font Object", object_data=cur)
891           bpy.data.collections[col_name].objects.link(obj)
892           bpy.context.view_layer.objects.active = obj
893           obj.rotation_euler = (0.0, 0.0, PI/2)
894           obj.location = (
895             self.origin[0] + (x_tick - self.x_range[0]) * self.x_scale,
896             self.origin[1] - self.x_ticks_props.axis_spacing, 0.0)
897           obj.scale = (
898             self.x_ticks_props.scale, self.x_ticks_props.scale,
899             self.x_ticks_props.scale)
900
901         # y_ticks.
902         cur = bpy.data.curves.new(name='Y Ticks', type='CURVE')
903         cur.dimensions = '2D'
904         for y_tick in self.y_ticks:
905           cur_spline = cur.splines.new('POLY')
906           cur_spline.points.add(1)
907           cur_spline.points[0].co = (
908             self.origin[0] - self.y_ticks_props.overflow_lenght,
909             self.origin[1] + (y_tick - self.y_range[0]) * self.y_scale, 0.0, 0.0)
910           cur_spline.points[1].co = (
911             self.origin[0] + (self.x_range[1] - self.x_range[0]) * self.x_scale,
912             self.origin[1] + (y_tick - self.y_range[0]) * self.y_scale, 0.0, 0.0)
```

```
913
914         obj = bpy.data.objects.new('Y Ticks', cur)
915         bpy.data.collections[col_name].objects.link(obj)
916         bpy.context.view_layer.objects.active = obj
917
918         # y_labels.
919         for y_tick in self.y_ticks:
920           cur = bpy.data.curves.new(type="FONT", name="FF")
921           cur.body = self._get_elevation(y_tick)
922           cur.font = fon
923           cur.align_x = 'RIGHT'
924           cur.align_y = 'CENTER'
925           obj = bpy.data.objects.new(name="Font Object", object_data=cur)
926           bpy.data.collections[col_name].objects.link(obj)
927           bpy.context.view_layer.objects.active = obj
928           obj.location = (
929             self.origin[0] - self.y_ticks_props.axis_spacing,
930             self.origin[1] + (y_tick - self.y_range[0]) * self.y_scale, 0.0)
931           obj.scale = (
932             self.y_ticks_props.scale, self.y_ticks_props.scale,
933             self.y_ticks_props.scale)
934
935         # profile.
936         cur = bpy.data.curves.new(name='Profile', type='CURVE')
937         cur.dimensions = '2D'
938         cur_spline = cur.splines.new('BEZIER')
939         cur_spline.bezier_points.add(self.profile.num_points - 1)
940         cur_spline.bezier_points.foreach_set('co',
941           np.hstack((
942             (self.origin[0] + (self.profile.x - self.x_range[0]) * \
943               self.x_scale).reshape((self.profile.num_points, 1)),
944             (self.origin[1] + (self.profile.y - self.y_range[0]) * \
945               self.y_scale).reshape((self.profile.num_points, 1)),
946             np.zeros((self.profile.num_points, 1))
947           )).flatten())
948         for i in range(self.profile.num_points):
949           cur_spline.bezier_points[i].handle_right_type = 'AUTO'
950           cur_spline.bezier_points[i].handle_left_type = 'AUTO'
951
952         obj = bpy.data.objects.new('Profile', cur)
953         bpy.data.collections[col_name].objects.link(obj)
954         bpy.context.view_layer.objects.active = obj
955
956
957     DIR_PATH = r"C:\Users\MIG17\Desktop\MIGUEL\ano7\TFM\codigos\312"
958
959     points_filepath = os.path.join(DIR_PATH, r".\points.csv")
960     raster_filepath = os.path.join(DIR_PATH, r".\elevation.tif")
961
962
963     if __name__ == '__main__':
964       pass
```

ANEJO II: Canales

Estructura del archivo .zip:

- data_materials: Esta carpeta debe contener las imágenes que definen el material del objeto que se desea crear (color, rugosidad, normales y desplazamiento).
- __init__.py
- add_pipe.py

__init__.py

```
965 bl_info = {
966   'name': 'Pipe',
967   'author': 'Miguel Puech de Oriol',
968   'version': (1, 0, 0),
969   'blender': (2, 92, 0),
970   'location': 'View3D > Add > Mesh',
971   'description': 'Adds a Pipe',
972   'warning': '',
973   'category': 'Add Mesh',
974 }
975
976
977 import bpy
978 from . import add_pipe
979
980
981 def register():
982   add_pipe.register()
983
984 def unregister():
985   add_pipe.unregister()
986
987
988 if __name__ == '__main__':
989   register()
```

```
990  from math import pi as PI
991  from math import (
992    sin,
993    cos
994  )
995  from bpy_extras import (
996    object_utils,
997    image_utils
998  )
999  import bmesh
1000 import bpy
1001 import os
1002
1003
1004 REINFORCED_CONCRETE_MEASUERES = {
1005   '400':  (0.400, 0.065), '500':  (0.500, 0.070), '600':  (0.600, 0.080),
1006   '800':  (0.800, 0.095), '1000': (1.000, 0.120), '1200': (1.200, 0.135),
1007   '1500': (1.500, 0.162), '1800': (1.800, 0.200), '2000': (2.000, 0.212),
1008   '2500': (2.500, 0.250),
1009 }
1010
1011 DUCTIL_IRON_MEASUERES = {
1012   '80':   (0.080, 0.018), '100':  (0.100, 0.018), '125':  (0.125, 0.019),
1013   '150':  (0.150, 0.020), '200':  (0.200, 0.022), '250':  (0.250, 0.024),
1014   '300':  (0.300, 0.026), '350':  (0.350, 0.028), '400':  (0.400, 0.029),
1015   '450':  (0.450, 0.030), '500':  (0.500, 0.032), '600':  (0.600, 0.035),
1016   '700':  (0.700, 0.038), '800':  (0.800, 0.042), '900':  (0.900, 0.045),
1017   '1000': (1.000, 0.048),
1018 }
1019
1020 POLYETHYLENE_MEASUERES = {
1021   'LOW_DENSITY PE-40': {
1022     'PN-4': {
1023       '32': (0.032, 0.0020), '40': (0.040, 0.0024), '50': (0.050, 0.0030),
1024       '63': (0.063, 0.0038), '75': (0.075, 0.0045), '90': (0.090, 0.0054),
1025     },
1026     'PN-6': {
1027       '20': (0.020, 0.0020), '25': (0.025, 0.0023), '32': (0.032, 0.0030),
1028       '40': (0.040, 0.0037), '50': (0.050, 0.0046), '63': (0.063, 0.0058),
1029       '75': (0.075, 0.0068), '90': (0.090, 0.0082),
1030     },
1031     'PN-10': {
1032       '20': (0.020, 0.0030), '25': (0.025, 0.0035), '32': (0.032, 0.0044),
1033       '40': (0.040, 0.0055), '50': (0.050, 0.0069), '63': (0.063, 0.0086),
1034       '75': (0.075, 0.0103), '90': (0.090, 0.0123),
1035     }
1036   },
1037   'HIGH_DENSITY PE-100': {
1038     'PN-10': {
1039       '50':  (0.050, 0.0030), '63':  (0.063, 0.0038), '75':  (0.075, 0.0045),
1040       '90':  (0.090, 0.0054), '110': (0.110, 0.0066), '125': (0.125, 0.0074),
1041       '140': (0.140, 0.0083), '160': (0.160, 0.0095), '180': (0.180, 0.0107),
1042       '200': (0.200, 0.0119), '250': (0.250, 0.0148), '315': (0.315, 0.0187),
1043     },
1044     'PN-16': {
```

```
1045          '50':  (0.050, 0.0046), '63':  (0.063, 0.0058), '75':  (0.075, 0.0068),
1046          '90':  (0.090, 0.0082), '110': (0.110, 0.0100), '125': (0.125, 0.0114),
1047          '140': (0.140, 0.0127), '160': (0.160, 0.0146), '180': (0.180, 0.0164),
1048          '200': (0.200, 0.0182), '250': (0.250, 0.0227), '315': (0.315, 0.0286),
1049        }
1050      }
1051    }
1052
1053    POLYVINYL_CHLORIDE_MEASUERES = {
1054      'PN-6': {
1055        '75':  (0.075, 0.0022), '90':  (0.090, 0.0027), '110': (0.110, 0.0027),
1056        '125': (0.125, 0.0031), '140': (0.140, 0.0035), '160': (0.160, 0.0041),
1057        '180': (0.180, 0.0044), '200': (0.200, 0.0050), '250': (0.250, 0.0062),
1058        '315': (0.315, 0.0077), '400': (0.400, 0.0098), '500': (0.500, 0.0123),
1059        '630': (0.630, 0.0154),
1060      },
1061      'PN-10': {
1062        '75':  (0.075, 0.0036), '90':  (0.090, 0.0043), '110': (0.110, 0.0042),
1063        '125': (0.125, 0.0048), '140': (0.140, 0.0054), '160': (0.160, 0.0062),
1064        '180': (0.180, 0.0070), '200': (0.200, 0.0077), '250': (0.250, 0.0096),
1065        '315': (0.315, 0.0121),
1066      },
1067      'PN-16': {
1068        '75':  (0.075, 0.0056), '90':  (0.090, 0.0067), '110': (0.110, 0.0066),
1069        '125': (0.125, 0.0074), '140': (0.140, 0.0083), '160': (0.160, 0.0095),
1070        '180': (0.180, 0.0107), '200': (0.200, 0.0119), '250': (0.250, 0.0148),
1071        '315': (0.315, 0.0187),
1072      }
1073    }
1074
1075    def get_dbase():
1076      return os.path.join(__file__.rstrip('add_pipe.py'), 'data_materials')
1077
1078    MATERIAL_DBASE = get_dbase()
1079
1080    REINFORCED_CONCRETE_MATERIAL = {
1081      'rc_base_color':    os.path.join(
1082        MATERIAL_DBASE, 'Concrete030_2K_Color.jpg'),
1083      'rc_roughness':     os.path.join(
1084        MATERIAL_DBASE, 'Concrete030_2K_Roughness.jpg'),
1085      'rc_normal':        os.path.join(
1086        MATERIAL_DBASE, 'Concrete030_2K_Normal.jpg'),
1087      'rc_displacement': os.path.join(
1088        MATERIAL_DBASE, 'Concrete030_2K_Displacement.jpg')
1089    }
1090
1091    DUCTIL_IRON_MATERIAL = { # Black metal.
1092      'di_base_color':    os.path.join(
1093        MATERIAL_DBASE, 'Metal027_2K_Color.jpg'),
1094      'di_roughness':     os.path.join(
1095        MATERIAL_DBASE, 'Metal027_2K_Roughness.jpg'),
1096      'di_normal':        os.path.join(
1097        MATERIAL_DBASE, 'Metal027_2K_Normal.jpg'),
1098      'di_displacement': os.path.join(
1099        MATERIAL_DBASE, 'Metal027_2K_Displacement.jpg')
1100    }
1101
1102    POLYETHYLENE_MATERIAL = { # Black plastic.
1103      'pe_base_color':    os.path.join(
```

```
1104          MATERIAL_DBASE, 'Plastic006_2K_Color.jpg'),
1105        'pe_roughness':    os.path.join(
1106          MATERIAL_DBASE, 'Plastic006_2K_Roughness.jpg'),
1107        'pe_normal':       os.path.join(
1108          MATERIAL_DBASE, 'Plastic006_2K_Normal.jpg'),
1109        'pe_displacement': os.path.join(
1110          MATERIAL_DBASE, 'Plastic006_2K_Displacement.jpg')
1111      }
1112
1113      POLYVINYL_CHLORIDE_MATERIAL = { # Blue plastic.
1114        'pvc_base_color':   os.path.join(
1115          MATERIAL_DBASE, 'Plastic008_2K_Color.jpg'),
1116        'pvc_roughness':    os.path.join(
1117          MATERIAL_DBASE, 'Plastic008_2K_Roughness.jpg'),
1118        'pvc_normal':       os.path.join(
1119          MATERIAL_DBASE, 'Plastic008_2K_Normal.jpg'),
1120        'pvc_displacement': os.path.join(
1121          MATERIAL_DBASE, 'Plastic008_2K_Displacement.jpg')
1122      }
1123
1124
1125      def set_geometry(outer_radius, inner_radius, length, resolution):
1126        vertices = []
1127        faces = []
1128        plus_length_2 = length / 2
1129        minus_length_2 = -plus_length_2
1130        d_alfa = -2 * PI / resolution
1131
1132        for i in range(resolution - 1):
1133          alfa = d_alfa * i
1134          a1 = sin(alfa) * outer_radius
1135          b1 = cos(alfa) * outer_radius
1136          a2 = sin(alfa) * inner_radius
1137          b2 = cos(alfa) * inner_radius
1138          vertices.extend((
1139            (a1, b1, minus_length_2),
1140            (a2, b2, minus_length_2),
1141            (a1, b1, plus_length_2),
1142            (a2, b2, plus_length_2)))
1143
1144          _4i = 4 * i
1145          _4im1 = 4 * (i + 1)
1146          _4im2 = 4 * (i + 2)
1147          faces.extend((
1148            (_4i,     _4im1,     _4im1 + 2, _4i + 2),
1149            (_4i + 2, _4im1 + 2, _4im2 - 1, _4im1 - 1),
1150            (_4i + 1, _4im1 - 1, _4im2 - 1, _4im1 + 1),
1151            (_4i,    _4i + 1,    _4im1 + 1, _4im1)))
1152
1153        alfa = d_alfa * (resolution - 1)
1154        a1 = sin(alfa) * outer_radius
1155        b1 = cos(alfa) * outer_radius
1156        a2 = sin(alfa) * inner_radius
1157        b2 = cos(alfa) * inner_radius
1158        vertices.extend((
1159          (a1, b1, minus_length_2),
1160          (a2, b2, minus_length_2),
1161          (a1, b1, plus_length_2),
1162          (a2, b2, plus_length_2)))
```

```
1163
1164      r4 = resolution * 4
1165      faces.extend((
1166        (r4 - 4, 0,       2, r4 - 2),
1167        (r4 - 2, 2,       3, r4 - 1),
1168        (r4 - 3, r4 - 1, 3, 1),
1169        (r4 - 4, r4 - 3, 1, 0)))
1170
1171      return (vertices, faces)
1172
1173    def get_nominal_measures(self):
1174      if self.material_type == 'RC':
1175        diameter = REINFORCED_CONCRETE_MEASUERES[self.nominal_diameter_rc][0]
1176        thickness = REINFORCED_CONCRETE_MEASUERES[self.nominal_diameter_rc][1]
1177      elif self.material_type == 'DI':
1178        diameter = DUCTIL_IRON_MEASUERES[self.nominal_diameter_di][0]
1179        thickness = DUCTIL_IRON_MEASUERES[self.nominal_diameter_di][1]
1180      elif self.material_type == 'PE':
1181        if self.density_pe == 'LOW_DENSITY PE-40':
1182          if self.nominal_pressure_pe_ld == 'PN-4':
1183            diameter = POLYETHYLENE_MEASUERES[self.density_pe][
1184              self.nominal_pressure_pe_ld][self.nominal_diameter_pe_ld_pn4][0]
1185            thickness = POLYETHYLENE_MEASUERES[self.density_pe][
1186              self.nominal_pressure_pe_ld][self.nominal_diameter_pe_ld_pn4][1]
1187          elif self.nominal_pressure_pe_ld == 'PN-6':
1188            diameter = POLYETHYLENE_MEASUERES[self.density_pe][
1189              self.nominal_pressure_pe_ld][self.nominal_diameter_pe_ld_pn6][0]
1190            thickness = POLYETHYLENE_MEASUERES[self.density_pe][
1191              self.nominal_pressure_pe_ld][self.nominal_diameter_pe_ld_pn6][1]
1192          elif self.nominal_pressure_pe_ld == 'PN-10':
1193            diameter = POLYETHYLENE_MEASUERES[self.density_pe][
1194              self.nominal_pressure_pe_ld][self.nominal_diameter_pe_ld_pn10][0]
1195            thickness = POLYETHYLENE_MEASUERES[self.density_pe][
1196              self.nominal_pressure_pe_ld][self.nominal_diameter_pe_ld_pn10][1]
1197        elif self.density_pe == 'HIGH_DENSITY PE-100':
1198          if self.nominal_pressure_pe_hd == 'PN-10':
1199            diameter = POLYETHYLENE_MEASUERES[self.density_pe][
1200              self.nominal_pressure_pe_hd][self.nominal_diameter_pe_hd_pn10][0]
1201            thickness = POLYETHYLENE_MEASUERES[self.density_pe][
1202              self.nominal_pressure_pe_hd][self.nominal_diameter_pe_hd_pn10][1]
1203          elif self.nominal_pressure_pe_hd == 'PN-16':
1204            diameter = POLYETHYLENE_MEASUERES[self.density_pe][
1205              self.nominal_pressure_pe_hd][self.nominal_diameter_pe_hd_pn16][0]
1206            thickness = POLYETHYLENE_MEASUERES[self.density_pe][
1207              self.nominal_pressure_pe_hd][self.nominal_diameter_pe_hd_pn16][1]
1208      elif self.material_type == 'PVC':
1209        if self.nominal_pressure_pvc == 'PN-6':
1210          diameter = POLYVINYL_CHLORIDE_MEASUERES[self.nominal_pressure_pvc][
1211            self.nominal_diameter_pvc_pn6][0]
1212          thickness = POLYVINYL_CHLORIDE_MEASUERES[self.nominal_pressure_pvc][
1213            self.nominal_diameter_pvc_pn6][1]
1214        elif self.nominal_pressure_pvc == 'PN-10':
1215          diameter = POLYVINYL_CHLORIDE_MEASUERES[self.nominal_pressure_pvc][
1216            self.nominal_diameter_pvc_pn10][0]
1217          thickness = POLYVINYL_CHLORIDE_MEASUERES[self.nominal_pressure_pvc][
1218            self.nominal_diameter_pvc_pn10][1]
1219        elif self.nominal_pressure_pvc == 'PN-16':
1220          diameter = POLYVINYL_CHLORIDE_MEASUERES[self.nominal_pressure_pvc][
1221            self.nominal_diameter_pvc_pn16][0]
```

```
1222        thickness = POLYVINYL_CHLORIDE_MEASUERES[self.nominal_pressure_pvc][
1223          self.nominal_diameter_pvc_pn16][1]
1224
1225      return diameter, thickness
1226
1227    def set_material(self, mat):
1228      if self.material_type == 'RC':
1229        mat.node_tree.nodes['Image Texture'].image = bpy.data.images['rc_base_color']
1230        mat.node_tree.nodes['Image Texture.001'].image = bpy.data.images[
1231          'rc_roughness']
1232        mat.node_tree.nodes['Image Texture.002'].image = bpy.data.images[
1233          'rc_normal']
1234        mat.node_tree.nodes['Image Texture.003'].image = bpy.data.images[
1235          'rc_displacement']
1236      elif self.material_type == 'DI':
1237        mat.node_tree.nodes['Image Texture'].image = bpy.data.images['di_base_color']
1238        mat.node_tree.nodes['Image Texture.001'].image = bpy.data.images[
1239          'di_roughness']
1240        mat.node_tree.nodes['Image Texture.002'].image = bpy.data.images[
1241          'di_normal']
1242        mat.node_tree.nodes['Image Texture.003'].image = bpy.data.images[
1243          'di_displacement']
1244      elif self.material_type == 'PE':
1245        mat.node_tree.nodes['Image Texture'].image = bpy.data.images['pe_base_color']
1246        mat.node_tree.nodes['Image Texture.001'].image = bpy.data.images[
1247          'pe_roughness']
1248        mat.node_tree.nodes['Image Texture.002'].image = bpy.data.images[
1249          'pe_normal']
1250        mat.node_tree.nodes['Image Texture.003'].image = bpy.data.images[
1251          'pe_displacement']
1252      elif self.material_type == 'PVC':
1253        mat.node_tree.nodes['Image Texture'].image = bpy.data.images['pvc_base_color']
1254        mat.node_tree.nodes['Image Texture.001'].image = bpy.data.images[
1255          'pvc_roughness']
1256        mat.node_tree.nodes['Image Texture.002'].image = bpy.data.images[
1257          'pvc_normal']
1258        mat.node_tree.nodes['Image Texture.003'].image = bpy.data.images[
1259          'pvc_displacement']
1260
1261    def set_uv_map(obj, outer_radius, inner_radius, resolution):
1262      bpy.ops.object.mode_set(mode='EDIT')
1263      mesh = obj.data
1264      bm = bmesh.from_edit_mesh(mesh)
1265      uv_layer = bm.loops.layers.uv.verify()
1266
1267      d_x = 1 / resolution
1268      d_alfa = -2 * PI / resolution
1269      c = -1
1270      cm1 = 0
1271      r1 = 0.25 # outer_radius
1272      r2 = inner_radius / outer_radius * r1 # inner_radius
1273      for face in bm.faces:
1274        if face.index % 2:
1275          if (face.index + 1) % 4:
1276            face.loops[0][uv_layer].uv = (
1277              0.25 + sin(c * d_alfa) * r1,    0.25 + cos(c * d_alfa) *  r1)
1278            face.loops[1][uv_layer].uv = (
1279              0.25 + sin(cm1 * d_alfa) * r1, 0.25 + cos(cm1 * d_alfa) * r1)
1280            face.loops[2][uv_layer].uv = (
```

```
1281              0.25 + sin(cm1 * d_alfa) * r2, 0.25 + cos(cm1 * d_alfa) * r2)
1282            face.loops[3][uv_layer].uv = (
1283              0.25 + sin(c * d_alfa) * r2,   0.25 + cos(c * d_alfa) *  r2)
1284          else:
1285            face.loops[0][uv_layer].uv = (
1286              0.75 + sin(c * d_alfa) * r1,   0.25 + cos(c * d_alfa) *  r1)
1287            face.loops[1][uv_layer].uv = (
1288              0.75 + sin(c * d_alfa) * r2,   0.25 + cos(c * d_alfa) *  r2)
1289            face.loops[2][uv_layer].uv = (
1290              0.75 + sin(cm1 * d_alfa) * r2, 0.25 + cos(cm1 * d_alfa) * r2)
1291            face.loops[3][uv_layer].uv = (
1292              0.75 + sin(cm1 * d_alfa) * r1, 0.25 + cos(cm1 * d_alfa) * r1)
1293        else:
1294          if not face.index % 4:
1295            c += 1
1296            cm1 += 1
1297            face.loops[0][uv_layer].uv = (c * d_x,   0.5)
1298            face.loops[1][uv_layer].uv = (cm1 * d_x, 0.5)
1299            face.loops[2][uv_layer].uv = (cm1 * d_x, 1.0)
1300            face.loops[3][uv_layer].uv = (c * d_x,   1.0)
1301          else:
1302            face.loops[0][uv_layer].uv = (c * d_x,   0.5)
1303            face.loops[1][uv_layer].uv = (c * d_x,   1.0)
1304            face.loops[2][uv_layer].uv = (cm1 * d_x, 1.0)
1305            face.loops[3][uv_layer].uv = (cm1 * d_x, 0.5)
1306
1307      bmesh.update_edit_mesh(mesh)
1308
1309      bpy.ops.object.mode_set(mode='OBJECT')
1310
1311    def main(context, self):
1312      if self.normalized:
1313        diameter, thickness = get_nominal_measures(self)
1314        radius = diameter / 2
1315      else:
1316        radius = self.diameter / 2
1317        thickness = self.thickness
1318
1319      if self.material_type in ('RC', 'DI'): # Nominal diameter -> Inner diameter.
1320        vertices, faces = set_geometry(
1321          radius + thickness, radius, self.length, self.resolution)
1322      else: # Nominal diameter -> Outer diameter.
1323        vertices, faces = set_geometry(
1324          radius, radius - thickness, self.length, self.resolution)
1325      edges = []
1326
1327      mesh = bpy.data.meshes.new('Pipe')
1328      obj = object_utils.object_data_add(context, mesh, operator=self)
1329      obj.select_set(True)
1330      mesh.from_pydata(vertices, edges, faces)
1331
1332      mat = bpy.data.materials[self.mat_name]
1333      set_material(self, mat)
1334      obj.data.materials.append(mat)
1335
1336      if self.material_type in ('RC', 'DI'): # Nominal diameter -> Inner diameter.
1337        set_uv_map(obj, radius + thickness, radius, self.resolution)
1338      else: # Nominal diameter -> Outer diameter.
1339        set_uv_map(obj, radius, radius - thickness, self.resolution)
```

```
1340
1341
1342  class OBJECT_OT_pipe(bpy.types.Operator, object_utils.AddObjectHelper):
1343    """Creates a pipe object with its uv_layer and material."""
1344    bl_idname = 'object.pipe'
1345    bl_label = 'Pipe'
1346    bl_description = 'Builds a pipe'
1347    bl_options = {'REGISTER', 'UNDO', 'PRESET'}
1348
1349    normalized: bpy.props.BoolProperty(
1350      name='Normalized pipe',
1351      description='Use normalized pipe',
1352      options={'HIDDEN'},
1353      default=True
1354    )
1355    def material_type_items(self, context):
1356      return (
1357        ('RC', 'Reinforced concrete', 'Reinforced concrete pipe'),
1358        ('DI', 'Ductile iron', 'Ductile iron pipe'),
1359        ('PE', 'Polyethylene ', 'Polyethylene pipe'),
1360        ('PVC', 'Polyvinyl chloride ', 'Polyvinyl chloride pipe'),
1361      )
1362    material_type: bpy.props.EnumProperty(
1363      items=material_type_items,
1364      name='Material type',
1365      description='Choose a material type',
1366      default=0
1367    )
1368    nominal_diameter_rc: bpy.props.EnumProperty(
1369      items=(
1370        ('400',  'ND 400 mm',  'ND 400 mm'),  ('500',  'ND 500 mm',  'ND 500 mm'),
1371        ('600',  'ND 600 mm',  'ND 600 mm'),  ('800',  'ND 800 mm',  'ND 800 mm'),
1372        ('1000', 'ND 1000 mm', 'ND 1000 mm'), ('1200', 'ND 1200 mm', 'ND 1200 mm'),
1373        ('1500', 'ND 1500 mm', 'ND 1500 mm'), ('1800', 'ND 1800 mm', 'ND 1800 mm'),
1374        ('2000', 'ND 2000 mm', 'ND 2000 mm'), ('2500', 'ND 2500 mm', 'ND 2500 mm')),
1375      name='Diameter',
1376      description='Pipe diameter',
1377      default=0
1378    )
1379    nominal_diameter_di: bpy.props.EnumProperty(
1380      items=(
1381        ('80',  'ND 80 mm',  'ND 80 mm'),  ('100',  'ND 100 mm',  'ND 100 mm'),
1382        ('125', 'ND 125 mm', 'ND 125 mm'), ('150',  'ND 150 mm',  'ND 150 mm'),
1383        ('200', 'ND 200 mm', 'ND 200 mm'), ('250',  'ND 250 mm',  'ND 250 mm'),
1384        ('300', 'ND 300 mm', 'ND 300 mm'), ('350',  'ND 350 mm',  'ND 350 mm'),
1385        ('400', 'ND 400 mm', 'ND 400 mm'), ('450',  'ND 450 mm',  'ND 450 mm'),
1386        ('500', 'ND 500 mm', 'ND 500 mm'), ('600',  'ND 600 mm',  'ND 600 mm'),
1387        ('700', 'ND 700 mm', 'ND 700 mm'), ('800',  'ND 800 mm',  'ND 800 mm'),
1388        ('900', 'ND 900 mm', 'ND 900 mm'), ('1000', 'ND 1000 mm', 'ND 1000 mm')),
1389      name='Diameter',
1390      description='Pipe diameter',
1391      default=0
1392    )
1393    density_pe: bpy.props.EnumProperty(
1394      items=(
1395        ('LOW_DENSITY PE-40',   'Low density PE-40',   'Low density PE-40'),
1396        ('HIGH_DENSITY PE-100', 'High density PE-100', 'High density PE-100')),
1397      name='Diameter',
1398      description='Pipe diameter',
```

```
1399          default=0
1400        )
1401        nominal_pressure_pe_ld: bpy.props.EnumProperty(
1402          items=(
1403            ('PN-4',  'PN-4',  'PN-4'),
1404            ('PN-6',  'PN-6',  'PN-6'),
1405            ('PN-10', 'PN-10', 'PN-10')),
1406          name='Density',
1407          description='Pipe density',
1408          default=0
1409        )
1410        nominal_pressure_pe_hd: bpy.props.EnumProperty(
1411          items=(('PN-10', 'PN-10', 'PN-10'), ('PN-16', 'PN-16', 'PN-16')),
1412          name='Pressure',
1413          description='Pipe nominal pressure',
1414          default=0
1415        )
1416        nominal_diameter_pe_ld_pn4: bpy.props.EnumProperty(
1417          items=(
1418            ('32', 'ND 32 mm', 'ND 32 mm'), ('40', 'ND 40 mm', 'ND 40 mm'),
1419            ('50', 'ND 50 mm', 'ND 50 mm'), ('63', 'ND 63 mm', 'ND 63 mm'),
1420            ('75', 'ND 75 mm', 'ND 75 mm'), ('90', 'ND 90 mm', 'ND 90 mm')),
1421          name='Diameter',
1422          description='Pipe diameter',
1423          default=0
1424        )
1425        nominal_diameter_pe_ld_pn6: bpy.props.EnumProperty(
1426          items=(
1427            ('20', 'ND 20 mm', 'ND 20 mm'), ('25', 'ND 25 mm', 'ND 25 mm'),
1428            ('32', 'ND 32 mm', 'ND 32 mm'), ('40', 'ND 40 mm', 'ND 40 mm'),
1429            ('50', 'ND 50 mm', 'ND 50 mm'), ('63', 'ND 63 mm', 'ND 63 mm'),
1430            ('75', 'ND 75 mm', 'ND 75 mm'), ('90', 'ND 90 mm', 'ND 90 mm')),
1431          name='Diameter',
1432          description='Pipe diameter',
1433          default=0
1434        )
1435        nominal_diameter_pe_ld_pn10: bpy.props.EnumProperty(
1436          items=(
1437            ('20', 'ND 20 mm', 'ND 20 mm'), ('25', 'ND 25 mm', 'ND 25 mm'),
1438            ('32', 'ND 32 mm', 'ND 32 mm'), ('40', 'ND 40 mm', 'ND 40 mm'),
1439            ('50', 'ND 50 mm', 'ND 50 mm'), ('63', 'ND 63 mm', 'ND 63 mm'),
1440            ('75', 'ND 75 mm', 'ND 75 mm'), ('90', 'ND 90 mm', 'ND 90 mm')),
1441          name='Diameter',
1442          description='Pipe diameter',
1443          default=0
1444        )
1445        nominal_diameter_pe_hd_pn10: bpy.props.EnumProperty(
1446          items=(
1447            ('50',  'ND 50 mm',  'ND 50 mm'),  ('63',  'ND 63 mm',  'ND 63 mm'),
1448            ('75',  'ND 75 mm',  'ND 75 mm'),  ('90',  'ND 90 mm',  'ND 90 mm'),
1449            ('110', 'ND 110 mm', 'ND 110 mm'), ('125', 'ND 125 mm', 'ND 125 mm'),
1450            ('140', 'ND 140 mm', 'ND 140 mm'), ('160', 'ND 160 mm', 'ND 160 mm'),
1451            ('180', 'ND 180 mm', 'ND 180 mm'), ('200', 'ND 200 mm', 'ND 200 mm'),
1452            ('250', 'ND 250 mm', 'ND 250 mm'), ('315', 'ND 315 mm', 'ND 315 mm')),
1453          name='Diameter',
1454          description='Pipe diameter',
1455          default=0
1456        )
1457        nominal_diameter_pe_hd_pn16: bpy.props.EnumProperty(
```

```
1458        items=(
1459          ('50',  'ND 50 mm',  'ND 50 mm'),  ('63',  'ND 63 mm',  'ND 63 mm'),
1460          ('75',  'ND 75 mm',  'ND 75 mm'),  ('90',  'ND 90 mm',  'ND 90 mm'),
1461          ('110', 'ND 110 mm', 'ND 110 mm'), ('125', 'ND 125 mm', 'ND 125 mm'),
1462          ('140', 'ND 140 mm', 'ND 140 mm'), ('160', 'ND 160 mm', 'ND 160 mm'),
1463          ('180', 'ND 180 mm', 'ND 180 mm'), ('200', 'ND 200 mm', 'ND 200 mm'),
1464          ('250', 'ND 250 mm', 'ND 250 mm'), ('315', 'ND 315 mm', 'ND 315 mm')),
1465        name='Diameter',
1466        description='Pipe diameter',
1467        default=0
1468      )
1469      nominal_pressure_pvc: bpy.props.EnumProperty(
1470        items=(
1471          ('PN-6',  'PN-6',  'PN-6'),
1472          ('PN-10', 'PN-10', 'PN-10'),
1473          ('PN-16', 'PN-16', 'PN-16')),
1474        name='Pressure',
1475        description='Pipe nominal pressure',
1476        default=0
1477      )
1478      nominal_diameter_pvc_pn6: bpy.props.EnumProperty(
1479        items=(
1480          ('75',  'ND 75 mm',  'ND 75 mm'),  ('90',  'ND 90 mm',  'ND 90 mm'),
1481          ('110', 'ND 110 mm', 'ND 110 mm'), ('125', 'ND 125 mm', 'ND 125 mm'),
1482          ('140', 'ND 140 mm', 'ND 140 mm'), ('160', 'ND 160 mm', 'ND 160 mm'),
1483          ('180', 'ND 180 mm', 'ND 180 mm'), ('200', 'ND 200 mm', 'ND 200 mm'),
1484          ('250', 'ND 250 mm', 'ND 250 mm'), ('315', 'ND 315 mm', 'ND 315 mm'),
1485          ('400', 'ND 400 mm', 'ND 400 mm'), ('500', 'ND 500 mm', 'ND 500 mm'),
1486          ('630', 'ND 630 mm', 'ND 630 mm')),
1487        name='Diameter',
1488        description='Pipe diameter',
1489        default=0
1490      )
1491      nominal_diameter_pvc_pn10: bpy.props.EnumProperty(
1492        items=(
1493          ('75',  'ND 75 mm',  'ND 75 mm'),  ('90',  'ND 90 mm',  'ND 90 mm'),
1494          ('110', 'ND 110 mm', 'ND 110 mm'), ('125', 'ND 125 mm', 'ND 125 mm'),
1495          ('140', 'ND 140 mm', 'ND 140 mm'), ('160', 'ND 160 mm', 'ND 160 mm'),
1496          ('180', 'ND 180 mm', 'ND 180 mm'), ('200', 'ND 200 mm', 'ND 200 mm'),
1497          ('250', 'ND 250 mm', 'ND 250 mm'), ('315', 'ND 315 mm', 'ND 315 mm')),
1498        name='Diameter',
1499        description='Pipe diameter',
1500        default=0
1501      )
1502      nominal_diameter_pvc_pn16: bpy.props.EnumProperty(
1503        items=(
1504          ('75',  'ND 75 mm',  'ND 75 mm'),  ('90',  'ND 90 mm',  'ND 90 mm'),
1505          ('110', 'ND 110 mm', 'ND 110 mm'), ('125', 'ND 125 mm', 'ND 125 mm'),
1506          ('140', 'ND 140 mm', 'ND 140 mm'), ('160', 'ND 160 mm', 'ND 160 mm'),
1507          ('180', 'ND 180 mm', 'ND 180 mm'), ('200', 'ND 200 mm', 'ND 200 mm'),
1508          ('250', 'ND 250 mm', 'ND 250 mm'), ('315', 'ND 315 mm', 'ND 315 mm')),
1509        name='Diameter',
1510        description='Pipe diameter',
1511        default=0
1512      )
1513      diameter: bpy.props.FloatProperty(
1514        name='Diameter',
1515        default=1,
1516        min=0.01,
```

```
1517          unit='LENGTH',
1518          precision=3
1519        )
1520        thickness: bpy.props.FloatProperty(
1521          name='Thickness',
1522          default=0.1,
1523          min=0.001,
1524          max=100,
1525          unit='LENGTH',
1526          precision=4
1527        )
1528        length: bpy.props.FloatProperty(
1529          name='Length',
1530          default=2,
1531          min=0.002,
1532          unit='LENGTH',
1533          precision=3
1534        )
1535        resolution: bpy.props.IntProperty(
1536          name='Resolution',
1537          default=3,
1538          min=3,
1539          max=500
1540        )
1541
1542        def draw(self, context):
1543          layout = self.layout
1544          col = layout.column()
1545          col.prop(self, 'normalized')
1546          col.prop(self, 'material_type')
1547
1548          box = layout.box()
1549          box_col = box.column(align=True)
1550          box_col.label(text='Measures:')
1551
1552          if self.normalized:
1553            if self.material_type == 'RC':
1554              col.prop(self, 'nominal_diameter_rc')
1555            elif self.material_type == 'DI':
1556              col.prop(self, 'nominal_diameter_di')
1557            elif self.material_type == 'PE':
1558              col.prop(self, 'density_pe')
1559              if self.density_pe == 'LOW_DENSITY PE-40':
1560                col.prop(self, 'nominal_pressure_pe_ld')
1561                if self.nominal_pressure_pe_ld == 'PN-4':
1562                  col.prop(self, 'nominal_diameter_pe_ld_pn4')
1563                elif self.nominal_pressure_pe_ld == 'PN-6':
1564                  col.prop(self, 'nominal_diameter_pe_ld_pn6')
1565                elif self.nominal_pressure_pe_ld == 'PN-10':
1566                  col.prop(self, 'nominal_diameter_pe_ld_pn10')
1567              elif self.density_pe == 'HIGH_DENSITY PE-100':
1568                col.prop(self, 'nominal_pressure_pe_hd')
1569                if self.nominal_pressure_pe_hd == 'PN-10':
1570                  col.prop(self, 'nominal_diameter_pe_hd_pn10')
1571                elif self.nominal_pressure_pe_hd == 'PN-16':
1572                  col.prop(self, 'nominal_diameter_pe_hd_pn16')
1573            elif self.material_type == 'PVC':
1574              col.prop(self, 'nominal_pressure_pvc')
1575              if self.nominal_pressure_pvc == 'PN-6':
```

```
1576              col.prop(self, 'nominal_diameter_pvc_pn6')
1577            elif self.nominal_pressure_pvc == 'PN-10':
1578              col.prop(self, 'nominal_diameter_pvc_pn10')
1579            elif self.nominal_pressure_pvc == 'PN-16':
1580              col.prop(self, 'nominal_diameter_pvc_pn16')
1581          diameter, thickness = get_nominal_measures(self)
1582          if self.material_type in ('RC', 'DI'):
1583            box_col.label(text='Nominal diameter (Inner): ' + str(diameter) + ' m')
1584          else:
1585            box_col.label(text='Nominal diameter (Outer): ' + str(diameter) + ' m')
1586          box_col.label(text='Thickness: ' + str(thickness) + ' mm')
1587        else:
1588          box_col.prop(self, 'diameter')
1589          box_col.prop(self, 'thickness')
1590
1591        box_col.prop(self, 'length')
1592        layout = self.layout
1593        col = layout.column()
1594        col.prop(self, 'resolution')
1595        col.prop(self, 'align')
1596        col.prop(self, 'location')
1597        col.prop(self, 'rotation')
1598
1599      def execute(self, context):
1600        self.mat_name = '_'.join(('PIPE_MATERIAL', self.material_type))
1601        if self.mat_name not in [mat.name for mat in bpy.data.materials]:
1602          # Create material.
1603          mat = bpy.data.materials.new(name=self.mat_name)
1604          self.mat_name = mat.name
1605
1606          mat.use_nodes = True
1607          mat_nodes = mat.node_tree.nodes
1608
1609          # Create material nodes.
1610          mat_nodes.new('ShaderNodeTexCoord')
1611          mat_nodes.new('ShaderNodeMapping')
1612          mat_nodes.new('ShaderNodeTexImage') # Base Colour.
1613          mat_nodes['Image Texture'].label = 'Base Color'
1614          mat_nodes.new('ShaderNodeTexImage') # Roughness.
1615          mat_nodes['Image Texture.001'].label = 'Roughness'
1616          mat_nodes.new('ShaderNodeTexImage') # Normal.
1617          mat_nodes['Image Texture.002'].label = 'Normal'
1618          mat_nodes.new('ShaderNodeTexImage') # Displacement.
1619          mat_nodes['Image Texture.003'].label = 'Displacement'
1620          mat_nodes.new('ShaderNodeNormalMap')
1621          mat_nodes.new('ShaderNodeDisplacement')
1622
1623          # Locate material nodes.
1624          mat_nodes['Texture Coordinate'].location = (-700.0, 125.0)
1625          mat_nodes['Mapping'].location = (-500.0, 175.0)
1626          mat_nodes['Image Texture'].location = (-300.0, 515.0)
1627          mat_nodes['Image Texture.001'].location = (-300.0, 255.0)
1628          mat_nodes['Image Texture.002'].location = (-300.0, -5.0)
1629          mat_nodes['Image Texture.003'].location = (-300.0, -265.0)
1630          mat_nodes['Normal Map'].location = (0.0, 0.0)
1631          mat_nodes['Displacement'].location = (275.0, -300.0)
1632          mat_nodes['Principled BSDF'].location = (225.0, 450.0)
1633          mat_nodes['Material Output'].location = (525.0, 50.0)
1634
```

```
1635        # Load material images.
1636        image_names = [im.name for im  in bpy.data.images]
1637        if 'rc_base_color' not in image_names:
1638          img = image_utils.load_image(
1639            REINFORCED_CONCRETE_MATERIAL['rc_base_color'])
1640          img.name = 'rc_base_color'
1641        if 'rc_roughness' not in image_names:
1642          img = image_utils.load_image(
1643            REINFORCED_CONCRETE_MATERIAL['rc_roughness'])
1644          img.name = 'rc_roughness'
1645          img.colorspace_settings.name = 'Non-Color'
1646        if 'rc_normal' not in image_names:
1647          img = image_utils.load_image(
1648            REINFORCED_CONCRETE_MATERIAL['rc_normal'])
1649          img.name = 'rc_normal'
1650          img.colorspace_settings.name = 'Non-Color'
1651        if 'rc_displacement' not in image_names:
1652          img = image_utils.load_image(
1653            REINFORCED_CONCRETE_MATERIAL['rc_displacement'])
1654          img.name = 'rc_displacement'
1655          img.colorspace_settings.name = 'Non-Color'
1656
1657        if 'di_base_color' not in image_names:
1658          img = image_utils.load_image(
1659            DUCTIL_IRON_MATERIAL['di_base_color'])
1660          img.name = 'di_base_color'
1661        if 'di_roughness' not in image_names:
1662          img = image_utils.load_image(
1663            DUCTIL_IRON_MATERIAL['di_roughness'])
1664          img.name = 'di_roughness'
1665          img.colorspace_settings.name = 'Non-Color'
1666        if 'di_normal' not in image_names:
1667          img = image_utils.load_image(
1668            DUCTIL_IRON_MATERIAL['di_normal'])
1669          img.name = 'di_normal'
1670          img.colorspace_settings.name = 'Non-Color'
1671        if 'di_displacement' not in image_names:
1672          img = image_utils.load_image(
1673            DUCTIL_IRON_MATERIAL['di_displacement'])
1674          img.name = 'di_displacement'
1675          img.colorspace_settings.name = 'Non-Color'
1676
1677        if 'pe_base_color' not in image_names:
1678          img = image_utils.load_image(
1679            POLYETHYLENE_MATERIAL['pe_base_color'])
1680          img.name = 'pe_base_color'
1681        if 'pe_roughness' not in image_names:
1682          img = image_utils.load_image(
1683            POLYETHYLENE_MATERIAL['pe_roughness'])
1684          img.name = 'pe_roughness'
1685          img.colorspace_settings.name = 'Non-Color'
1686        if 'pe_normal' not in image_names:
1687          img = image_utils.load_image(
1688            POLYETHYLENE_MATERIAL['pe_normal'])
1689          img.name = 'pe_normal'
1690          img.colorspace_settings.name = 'Non-Color'
1691        if 'pe_displacement' not in image_names:
1692          img = image_utils.load_image(
1693            POLYETHYLENE_MATERIAL['pe_displacement'])
```

```
1694            img.name = 'pe_displacement'
1695            img.colorspace_settings.name = 'Non-Color'
1696
1697          if 'pvc_base_color' not in image_names:
1698            img = image_utils.load_image(
1699              POLYVINYL_CHLORIDE_MATERIAL['pvc_base_color'])
1700            img.name = 'pvc_base_color'
1701          if 'pvc_roughness' not in image_names:
1702            img = image_utils.load_image(
1703              POLYVINYL_CHLORIDE_MATERIAL['pvc_roughness'])
1704            img.name = 'pvc_roughness'
1705            img.colorspace_settings.name = 'Non-Color'
1706          if 'pvc_normal' not in image_names:
1707            img = image_utils.load_image(
1708              POLYVINYL_CHLORIDE_MATERIAL['pvc_normal'])
1709            img.name = 'pvc_normal'
1710            img.colorspace_settings.name = 'Non-Color'
1711          if 'pvc_displacement' not in image_names:
1712            img = image_utils.load_image(
1713              POLYVINYL_CHLORIDE_MATERIAL['pvc_displacement'])
1714            img.name = 'pvc_displacement'
1715            img.colorspace_settings.name = 'Non-Color'
1716
1717          # Specify material node parameters.
1718          mat_nodes['Image Texture'].image = bpy.data.images['rc_base_color']
1719          mat_nodes['Image Texture.001'].image = bpy.data.images['rc_roughness']
1720          mat_nodes['Image Texture.002'].image = bpy.data.images['rc_normal']
1721          mat_nodes['Image Texture.003'].image = bpy.data.images['rc_displacement']
1722
1723          # Create material links.
1724          mat.node_tree.links.new(
1725            mat_nodes['Texture Coordinate'].outputs['UV'],
1726            mat_nodes['Mapping'].inputs['Vector'])
1727
1728          mat.node_tree.links.new(
1729            mat_nodes['Mapping'].outputs['Vector'],
1730            mat_nodes['Image Texture'].inputs['Vector'])
1731          mat.node_tree.links.new(
1732            mat_nodes['Mapping'].outputs['Vector'],
1733            mat_nodes['Image Texture.001'].inputs['Vector'])
1734          mat.node_tree.links.new(
1735            mat_nodes['Mapping'].outputs['Vector'],
1736            mat_nodes['Image Texture.002'].inputs['Vector'])
1737          mat.node_tree.links.new(
1738            mat_nodes['Mapping'].outputs['Vector'],
1739            mat_nodes['Image Texture.003'].inputs['Vector'])
1740
1741          mat.node_tree.links.new(
1742            mat_nodes['Image Texture.002'].outputs['Color'],
1743            mat_nodes['Normal Map'].inputs['Color'])
1744
1745          mat.node_tree.links.new(
1746            mat_nodes['Image Texture'].outputs['Color'],
1747            mat_nodes['Principled BSDF'].inputs['Base Color'])
1748          mat.node_tree.links.new(
1749            mat_nodes['Image Texture.001'].outputs['Color'],
1750            mat_nodes['Principled BSDF'].inputs['Roughness'])
1751          mat.node_tree.links.new(
1752            mat_nodes['Normal Map'].outputs['Normal'],
```

```
1753            mat_nodes['Principled BSDF'].inputs['Normal'])
1754          mat.node_tree.links.new(
1755            mat_nodes['Image Texture.003'].outputs['Color'],
1756            mat_nodes['Displacement'].inputs['Height'])
1757
1758          mat.node_tree.links.new(
1759            mat_nodes['Principled BSDF'].outputs['BSDF'],
1760            mat_nodes['Material Output'].inputs['Surface'])
1761          mat.node_tree.links.new(
1762            mat_nodes['Displacement'].outputs['Displacement'],
1763            mat_nodes['Material Output'].inputs['Displacement'])
1764
1765        main(context, self)
1766
1767        return {'FINISHED'}
1768
1769
1770    def menu_func(self, context):
1771      self.layout.operator(OBJECT_OT_pipe.bl_idname)
1772
1773    def register():
1774      bpy.utils.register_class(OBJECT_OT_pipe)
1775      bpy.types.VIEW3D_MT_mesh_add.append(menu_func)
1776
1777    def unregister():
1778      bpy.utils.unregister_class(OBJECT_OT_pipe)
1779      bpy.types.VIEW3D_MT_mesh_add.remove(menu_func)
1780
1781
1782    if __name__ == '__main__':
1783      register()
```

ANEJO III: Puertos

breakwater_rock.py

```
1784  from math import pi as PI
1785  import random
1786  import bpy
1787
1788
1789  class RockMaterial():
1790    def __init__(self, base_color_filepath, roughness_filepath, normal_filepath,
1791  displacement_filepath):
1792      self.base_color_filepath = base_color_filepath
1793      self.roughness_filepath = roughness_filepath
1794      self.normal_filepath = normal_filepath
1795      self.displacement_filepath = displacement_filepath
1796
1797
1798  class NaturalRockFill():
1799    def __init__(self, material=None, min_weight=1.0, max_weight=10.0,
1800        density=2.65, y_scale=1.0, z_scale=1.0, levels=4, noise_type='CLOUDS',
1801        noise_basis='ORIGINAL_PERLIN', strength=0.2, seed=0):
1802      self.material = material
1803      self.min_weight = min_weight
1804      self.max_weight = max_weight
1805      self.density = density
1806      self.size_range = ((
1807        self.min_weight / self.density)**(1/3) / 2,
1808        (self.max_weight / self.density)**(1/3) / 2)
1809      self.y_scale = y_scale # Y scale range -> [0.2, 1]
1810      self.z_scale = z_scale # Z scale range -> [0.2, 1]
1811      self.levels = levels
1812      self.noise_type = noise_type
1813      self.noise_basis = noise_basis
1814      self.strength = strength
1815      self.seed = seed
1816
1817    def blendify(self):
1818      # Create collection.
1819      col = bpy.data.collections.new('Breakwater')
1820      bpy.context.scene.collection.children.link(col)
1821
1822      master_col = bpy.context.collection
1823
1824      # Create base cube.
1825      bpy.ops.mesh.primitive_cube_add()
1826      obj = bpy.context.object
1827      obj.name = 'Rock'
1828      col.objects.link(obj)
1829      master_col.objects.unlink(obj)
1830
1831      # Move vertices within ranges.
1832      if self.seed:
1833        random.seed(self.seed)
1834      for i in range(8):
1835        for j in range(3):
1836          obj.data.vertices[i].co[j] *= self.size_range[0] + \
1837            (self.size_range[1] - self.size_range[0]) * random.random()
1838
1839      # Reposition object in the breakwater.
1840      obj.location = (0.0, 0.0, 0.0)
```

```
1841        obj.scale = (1.0, self.y_scale, self.z_scale)
1842        obj.rotation_euler = (0.0, 0.0, 0.0)
1843
1844        # Add subdivision surface modifier.
1845        subivision_surface_mod = obj.modifiers.new(
1846          'Subdivision surface', type='SUBSURF')
1847        subivision_surface_mod.levels = self.levels
1848        subivision_surface_mod.render_levels = self.levels
1849
1850        # Create displace texture.
1851        if self.noise_type == 'CLOUDS':
1852          displace_tex = bpy.data.textures.new('Displace texture', type='CLOUDS')
1853          displace_tex.noise_basis = self.noise_basis
1854          if self.levels == 4:
1855            displace_tex.noise_scale = 0.75 + 1.25 * random.random()
1856          elif self.levels == 5:
1857            displace_tex.noise_scale = 1.125 + 0.875 * random.random()
1858          elif self.levels == 6:
1859            displace_tex.noise_scale = 1.5 + 0.5 * random.random()
1860          displace_tex.noise_depth = 10
1861
1862        elif  self.noise_type == 'VORONOI':
1863          displace_tex = bpy.data.textures.new('Displace texture', type='VORONOI')
1864          displace_tex.noise_intensity = 0.8 + 0.2 * random.random()
1865          displace_tex.noise_scale = 0.9 + 0.3 * random.random()
1866          displace_tex.weight_1 = 1
1867          displace_tex.weight_2 = 0.5 + 1.5 * random.random()
1868        elif  self.noise_type == 'STUCCI':
1869          displace_tex = bpy.data.textures.new('Displace texture', type='STUCCI')
1870          displace_tex.noise_basis = self.noise_basis
1871
1872        # Add displace modifier.
1873        displace_mod = obj.modifiers.new('Displace', type='DISPLACE')
1874        displace_mod.texture = displace_tex
1875        displace_mod.strength = self.strength
1876
1877        # Apply modifiers.
1878        bpy.context.view_layer.objects.active = obj
1879        obj.select_set(True)
1880        bpy.ops.object.modifier_apply(modifier='Subdivision surface')
1881        bpy.ops.object.modifier_apply(modifier='Displace')
1882
1883        # Smooth faces.
1884        bpy.ops.object.shade_smooth()
1885
1886        if self.material:
1887          # Create material.
1888          mat = bpy.data.materials.new(name='Rockfill material')
1889          obj.data.materials.append(mat)
1890
1891          mat.use_nodes = True
1892          mat_nodes = mat.node_tree.nodes
1893
1894          # Create material nodes.
1895          mat_nodes.new('ShaderNodeTexCoord')
1896          mat_nodes.new('ShaderNodeMapping')
1897          mat_nodes.new('ShaderNodeTexImage') # Base Colour.
1898          mat_nodes['Image Texture'].label = 'Base Color'
1899          mat_nodes.new('ShaderNodeTexImage') # Roughness.
1900          mat_nodes['Image Texture.001'].label = 'Roughness'
1901          mat_nodes.new('ShaderNodeTexImage') # Normal.
1902          mat_nodes['Image Texture.002'].label = 'Normal'
1903          mat_nodes.new('ShaderNodeTexImage') # Displacement.
1904          mat_nodes['Image Texture.003'].label = 'Displacement'
1905          mat_nodes.new('ShaderNodeNormalMap')
1906          mat_nodes.new('ShaderNodeDisplacement')
```

```
1907
1908          # Locate material nodes.
1909          mat_nodes['Texture Coordinate'].location = (-700.0, 125.0)
1910          mat_nodes['Mapping'].location = (-500.0, 175.0)
1911          mat_nodes['Image Texture'].location = (-300.0, 515.0)
1912          mat_nodes['Image Texture.001'].location = (-300.0, 255.0)
1913          mat_nodes['Image Texture.002'].location = (-300.0, -5.0)
1914          mat_nodes['Image Texture.003'].location = (-300.0, -265.0)
1915          mat_nodes['Normal Map'].location = (0.0, 0.0)
1916          mat_nodes['Displacement'].location = (275.0, -300.0)
1917          mat_nodes['Principled BSDF'].location = (225.0, 450.0)
1918          mat_nodes['Material Output'].location = (525.0, 50.0)
1919
1920          # Specify material node parameters.
1921          base_color_tex = bpy.data.images.load(self.material.base_color_filepath)
1922          mat_nodes['Image Texture'].image = base_color_tex
1923          roughness_tex = bpy.data.images.load(self.material.roughness_filepath)
1924          mat_nodes['Image Texture.001'].image = roughness_tex
1925          bpy.data.images[-1].colorspace_settings.name = 'Non-Color'
1926          normal_tex = bpy.data.images.load(self.material.normal_filepath)
1927          mat_nodes['Image Texture.002'].image = normal_tex
1928          bpy.data.images[-1].colorspace_settings.name = 'Non-Color'
1929          displacement_tex = bpy.data.images.load(self.material.displacement_filepath)
1930          mat_nodes['Image Texture.003'].image = displacement_tex
1931          bpy.data.images[-1].colorspace_settings.name = 'Non-Color'
1932
1933          mat_nodes['Principled BSDF'].inputs[5].default_value = 0 # Specular.
1934
1935          # Create material links.
1936          mat.node_tree.links.new(
1937            mat_nodes['Texture Coordinate'].outputs['UV'],
1938            mat_nodes['Mapping'].inputs['Vector'])
1939
1940          mat.node_tree.links.new(
1941            mat_nodes['Mapping'].outputs['Vector'],
1942            mat_nodes['Image Texture'].inputs['Vector'])
1943          mat.node_tree.links.new(
1944            mat_nodes['Mapping'].outputs['Vector'],
1945            mat_nodes['Image Texture.001'].inputs['Vector'])
1946          mat.node_tree.links.new(
1947            mat_nodes['Mapping'].outputs['Vector'],
1948            mat_nodes['Image Texture.002'].inputs['Vector'])
1949          mat.node_tree.links.new(
1950            mat_nodes['Mapping'].outputs['Vector'],
1951            mat_nodes['Image Texture.003'].inputs['Vector'])
1952
1953          mat.node_tree.links.new(
1954            mat_nodes['Image Texture.002'].outputs['Color'],
1955            mat_nodes['Normal Map'].inputs['Color'])
1956
1957          mat.node_tree.links.new(
1958            mat_nodes['Image Texture'].outputs['Color'],
1959            mat_nodes['Principled BSDF'].inputs['Base Color'])
1960          mat.node_tree.links.new(
1961            mat_nodes['Image Texture.001'].outputs['Color'],
1962            mat_nodes['Principled BSDF'].inputs['Roughness'])
1963          mat.node_tree.links.new(
1964            mat_nodes['Normal Map'].outputs['Normal'],
1965            mat_nodes['Principled BSDF'].inputs['Normal'])
1966          mat.node_tree.links.new(
1967            mat_nodes['Image Texture.003'].outputs['Color'],
1968            mat_nodes['Displacement'].inputs['Height'])
1969
1970          mat.node_tree.links.new(
1971            mat_nodes['Principled BSDF'].outputs['BSDF'],
1972            mat_nodes['Material Output'].inputs['Surface'])
```

```
1973            mat.node_tree.links.new(
1974              mat_nodes['Displacement'].outputs['Displacement'],
1975              mat_nodes['Material Output'].inputs['Displacement'])
1976
1977
1978      if __name__ == '__main__':
1979        pass
```

Dependencias externas:

- numpy

breakwater_blocks.py

```
1980 from math import pi as PI
1981 from math import (
1982   cos,
1983   sin
1984 )
1985 import numpy as np
1986 import bpy
1987
1988
1989 def create_obj(verts, faces, obj_name, col_name="Breakwater Blocks"):
1990   mesh = bpy.data.meshes.new(obj_name)
1991   obj = bpy.data.objects.new(mesh.name, mesh)
1992
1993   col = bpy.data.collections.get(col_name)
1994   if not col:
1995     col = bpy.data.collections.new(col_name)
1996     bpy.context.scene.collection.children.link(col)
1997   col.objects.link(obj)
1998   bpy.context.view_layer.objects.active = obj
1999
2000   edges = []
2001   mesh.from_pydata(verts, edges, faces)
2002
2003 # TETRAPODS | tn: (H, FLAT, R, r, r_p)
2004 TETRAPODS = {
2005   '0.5':  (0.900, 0.57960, 0.099, 0.135),
2006   '1':    (1.130, 0.72772, 0.124, 0.169),
2007   '2':    (1.420, 0.91448, 0.156, 0.213),
2008   '3.2':  (1.650, 1.06260, 0.181, 0.247),
2009   '4':    (1.790, 1.15276, 0.196, 0.268),
2010   '5':    (1.930, 1.24292, 0.212, 0.289),
2011   '6.3':  (2.075, 1.33630, 0.228, 0.311),
2012   '8':    (2.260, 1.45544, 0.248, 0.339),
2013   '10':   (2.430, 1.56492, 0.267, 0.364),
2014   '12.5': (2.620, 1.68728, 0.288, 0.393),
2015   '16':   (2.830, 1.82252, 0.311, 0.424),
2016   '20':   (3.060, 1.97064, 0.336, 0.459),
2017   '25':   (3.300, 2.12520, 0.363, 0.495),
2018   '32':   (3.550, 2.28620, 0.390, 0.532),
2019   '40':   (3.860, 2.48584, 0.424, 0.579),
2020   '50':   (4.155, 2.67582, 0.457, 0.623),
2021   '64':   (4.505, 2.90122, 0.495, 0.675),
2022   '80':   (5.000, 3.22000, 0.550, 0.750)
2023 }
2024
2025 # STANDARD AKMON tn: (H, A, B, C, D)
2026 STANDARD_AKMON = {
2027   '1':  (1.130, 1.017, 0.226, 0.338, 0.497),
2028   '2':  (1.430, 1.287, 0.286, 0.428, 0.629),
2029   '3':  (1.640, 1.476, 0.328, 0.492, 0.722),
2030   '4':  (1.800, 1.620, 0.360, 0.540, 0.792),
2031   '5':  (1.940, 1.746, 0.388, 0.582, 0.854),
2032   '6':  (2.050, 1.845, 0.410, 0.616, 0.901),
```

```
2033      '8':  (2.260, 2.034, 0.452, 0.678, 0.994),
2034      '12': (2.590, 2.331, 0.518, 0.778, 1.139),
2035      '16': (2.850, 2.565, 0.570, 0.856, 1.255)
2036    }
2037
2038    # HALF AKMON | tn: (H, A, B, C, D)
2039    HALF_AKMON = {
2040      '1':  (0.565, 1.017, 0.226, 0.169, 0.497, 0.260),
2041      '2':  (0.715, 1.287, 0.286, 0.214, 0.629, 0.329),
2042      '3':  (0.820, 1.476, 0.328, 0.246, 0.722, 0.377),
2043      '4':  (0.900, 1.620, 0.360, 0.270, 0.792, 0.414),
2044      '5':  (0.970, 1.746, 0.388, 0.291, 0.854, 0.446),
2045      '6':  (1.025, 1.845, 0.410, 0.308, 0.901, 0.472),
2046      '8':  (1.130, 2.034, 0.452, 0.339, 0.994, 0.520),
2047      '12': (1.295, 2.331, 0.518, 0.389, 1.139, 0.596),
2048      '16': (1.425, 2.565, 0.570, 0.428, 1.255, 0.655)
2049    }
2050
2051    # FLAT CROSS AKMON | tn: (H, A, B, C, D)
2052    FLAT_CROSS_AKMON = {
2053      '1':  (1.130, 1.017, 0.226, 0.338, 0.497),
2054      '2':  (1.430, 1.287, 0.286, 0.428, 0.629),
2055      '3':  (1.640, 1.476, 0.328, 0.492, 0.722),
2056      '4':  (1.800, 1.620, 0.360, 0.540, 0.792),
2057      '5':  (1.940, 1.746, 0.388, 0.582, 0.854),
2058      '6':  (2.050, 1.845, 0.410, 0.616, 0.901),
2059      '8':  (2.260, 2.034, 0.452, 0.678, 0.994),
2060      '10': (2.440, 2.196, 0.488, 0.732, 1.074),
2061      '16': (2.850, 2.565, 0.570, 0.856, 1.255)
2062    }
2063
2064    # HEXALEG | tn: h
2065    HEXALEG = {
2066      '1':  0.195, '2':  0.246, '3':  0.282, '4':  0.310, '5':  0.334,
2067      '6':  0.355, '8':  0.390, '12': 0.447, '16': 0.492
2068    }
2069
2070    # TRIPOD | tn: h
2071    TRIPOD = {
2072      '1':  0.95, '2':  1.20, '3':  1.35, '4':  1.50, '5':  1.60,
2073      '6':  1.70, '8':  1.90, '12': 2.15, '16': 2.35
2074    }
2075
2076    # CUBE | tn: h
2077    CUBE = {
2078      '1':  0.40, '2':  0.50, '3':  0.55, '4':  0.60, '5':  0.65,
2079      '6':  0.70, '8':  0.75, '12': 0.85, '16': 0.95
2080    }
2081
2082    # COB | tn: (h, m)
2083    COB = {
2084      '1':  (1.40, 1.00), '2':  (1.75, 1.20), '3':  (2.00, 1.40), '4':  (2.20, 1.55),
2085      '5':  (2.40, 1.70), '6':  (2.50, 1.80), '8':  (2.80, 1.95), '12': (3.20, 2.20),
2086      '16': (3.50, 2.45)
2087    }
2088
2089    # DOLOS
2090    DOLOS = {
2091      '40': (4.823, 1.543, 0.986, 0.830, 1.441, 0.270, 0.078),
2092      '50': (5.196, 1.663, 1.062, 0.894, 1.552, 0.291, 0.084),
2093      '80': (6.077, 1.945, 1.242, 1.045, 1.816, 0.340, 0.099)
2094    }
2095
2096
2097    def create_tetrapod(tn, resolution=67, obj_name='TETRAPOD',
2098        col_name='Breakwater Blocks'):
```

```
2099    # NOTE: not all combinations of resolution and tn produce the desired geoemtry.
2100    def rotate(verts, rotation_axis, angle):
2101      if rotation_axis == 'Y':
2102        return np.dot(verts, np.array((
2103          (cos(angle), 0, -sin(angle)),
2104          (0, 1, 0),
2105          (sin(angle), 0, cos(angle)))))
2106      elif rotation_axis == 'Z':
2107        return np.dot(verts, np.array((
2108          (cos(angle), -sin(angle), 0),
2109          (sin(angle), cos(angle), 0),
2110          (0, 0, 1))))
2111
2112    col = bpy.data.collections.get(col_name)
2113    if not col:
2114      col = bpy.data.collections.new(col_name)
2115      bpy.context.scene.collection.children.link(col)
2116
2117    H = TETRAPODS[tn][0]
2118    F = TETRAPODS[tn][1]
2119    r = TETRAPODS[tn][2]
2120    r_p = TETRAPODS[tn][3]
2121    F_p = F - r_p + r
2122
2123    R = (F_p + 5.671281819617707 * r_p) / 5.671281819617707
2124
2125    # Create geometry.
2126    verts = []
2127    faces = []
2128    top_face = []
2129    bottom_face = []
2130    d_alfa = 2 * PI / resolution
2131    alfa = 0
2132    for i in range(resolution - 1):
2133      alfa += d_alfa
2134      s_alfa = sin(alfa)
2135      c_alfa = cos(alfa)
2136      verts.extend((
2137        (R * c_alfa,   R * s_alfa,   0),
2138        (r_p * c_alfa, r_p * s_alfa, F_p),
2139        (r * c_alfa,   r * s_alfa,   F)))
2140      _3i = 3 * i
2141      _3im1 = 3 * (i + 1)
2142      faces.extend((
2143        (_3i,     _3im1,     _3im1 + 1, _3i + 1),
2144        (_3i + 1, _3im1 + 1, _3im1 + 2, _3i + 2)))
2145      top_face.append(_3i + 2)
2146      bottom_face.append(3 * (resolution - i - 1))
2147    alfa += d_alfa
2148    s_alfa = sin(alfa)
2149    c_alfa = cos(alfa)
2150    verts.extend((
2151      (R * c_alfa,   R * s_alfa,   0),
2152      (r_p * c_alfa, r_p * s_alfa, F_p),
2153      (r * c_alfa,   r * s_alfa,   F)))
2154    _3r = 3 * resolution
2155    faces.extend((
2156      (_3r - 3, 0, 1, _3r - 2),
2157      (_3r - 2, 1, 2, _3r - 1)))
2158    top_face.append(_3r - 1)
2159    bottom_face.append(0)
2160    faces.extend((
2161      (top_face),
2162      (bottom_face)))
2163
2164    edges = []
```

```
2165
2166      verts_0 = rotate(verts, 'Z', PI / 3)
2167
2168      # Create object.
2169      mesh = bpy.data.meshes.new(obj_name)
2170      mesh.from_pydata(verts, edges, faces)
2171      obj_0 = bpy.data.objects.new(mesh.name, mesh)
2172      col.objects.link(obj_0)
2173      bpy.context.view_layer.objects.active = obj_0
2174
2175      data = zip(
2176        (('Y',), ('Y', 'Z'), ('Y', 'Z')),
2177        ((2 * PI / 3,), (2 * PI / 3, 2 * PI / 3), (2 * PI / 3, 4 * PI / 3)))
2178      for axis, angle in data:
2179        # Create geometry.
2180        verts_1 = rotate(verts, axis[0], angle[0])
2181        if len(axis) > 1:
2182          verts_1 = rotate(verts_1, axis[1], angle[1])
2183
2184        # Create object.
2185        mesh = bpy.data.meshes.new('CYLINDER_1')
2186        mesh.from_pydata(verts_1, edges, faces)
2187        obj_1 = bpy.data.objects.new(mesh.name, mesh)
2188        col.objects.link(obj_1)
2189        bpy.context.view_layer.objects.active = obj_1
2190
2191        # Apply modifier.
2192        union_boolean_modifier = obj_0.modifiers.new(type='BOOLEAN', name='UNION')
2193        bpy.context.view_layer.objects.active = obj_0
2194        union_boolean_modifier.object = obj_1
2195        union_boolean_modifier.operation = 'UNION'
2196        bpy.ops.object.modifier_apply(modifier=union_boolean_modifier.name)
2197        bpy.data.objects.remove(obj_1)
2198        bpy.data.meshes.remove(mesh)
2199
2200    def create_standard_akmon(tn):
2201      H = STANDARD_AKMON[tn][0]
2202      A = STANDARD_AKMON[tn][1]
2203      B = STANDARD_AKMON[tn][2]
2204      C = STANDARD_AKMON[tn][3]
2205      D = STANDARD_AKMON[tn][4]
2206
2207      verts = [( A/2, -D/2, -H/2),   # 0
2208           ( A/2,  D/2, -H/2),   # 1
2209           (-A/2,  D/2, -H/2),   # 2
2210           (-A/2, -D/2, -H/2),   # 3
2211           ( A/2, -D/2, -H/2+B), # 4
2212           ( A/2,  D/2, -H/2+B), # 5
2213           (-A/2,  D/2, -H/2+B), # 6
2214           (-A/2, -D/2, -H/2+B), # 7
2215           ( D/2, -D/2, -C/2),   # 8
2216           ( D/2,  D/2, -C/2),   # 9
2217           (-D/2,  D/2, -C/2),   # 10
2218           (-D/2, -D/2, -C/2),   # 11
2219           ( D/2, -D/2, C/2),    # 12
2220           ( D/2,  D/2, C/2),    # 13
2221           (-D/2,  D/2, C/2),    # 14
2222           (-D/2, -D/2, C/2),    # 15
2223           ( D/2, -A/2, H/2-B),  # 16
2224           ( D/2,  A/2, H/2-B),  # 17
2225           (-D/2,  A/2, H/2-B),  # 18
2226           (-D/2, -A/2, H/2-B),  # 19
2227           ( D/2, -A/2, H/2),    # 20
2228           ( D/2,  A/2, H/2),    # 21
2229           (-D/2,  A/2, H/2),    # 22
2230           (-D/2, -A/2, H/2),]   # 23
```

```
2231    faces = [( 0,  3,  2,  1),                      # 0
2232             ( 0,  1,  5,  4),                      # 1
2233             ( 2,  3,  7,  6),                      # 2
2234             ( 1,  2,  6, 10, 14, 13,  9,  5), # 3
2235             ( 0,  4,  8, 12, 15, 11,  7,  3), # 4
2236             ( 4,  5,  9,  8),                      # 5
2237             ( 6,  7, 11, 10),                      # 6
2238             ( 8,  9, 13, 17, 21, 20, 16, 12), # 7
2239             (10, 11, 15, 19, 23, 22, 18, 14), # 8
2240             (13, 14, 18, 17),                      # 9
2241             (12, 16, 19, 15),                      # 10
2242             (17, 18, 22, 21),                      # 11
2243             (16, 20, 23, 19),                      # 12
2244             (20, 21, 22, 23)]                      # 13
2245
2246    create_obj(verts, faces, 'STANDARD AKMON')
2247
2248  def create_half_akmon(tn):
2249    H = HALF_AKMON[tn][0]
2250    A = HALF_AKMON[tn][1]
2251    B = HALF_AKMON[tn][2]
2252    C = HALF_AKMON[tn][3]
2253    D = HALF_AKMON[tn][4]
2254
2255    verts = [( A/2, -D/2, -H),    # 0
2256             ( A/2,  D/2, -H),    # 1
2257             (-A/2,  D/2, -H),    # 2
2258             (-A/2, -D/2, -H),    # 3
2259             ( A/2, -D/2, -H+B), # 4
2260             ( A/2,  D/2, -H+B), # 5
2261             (-A/2,  D/2, -H+B), # 6
2262             (-A/2, -D/2, -H+B), # 7
2263             ( D/2, -D/2, -C/2), # 8
2264             ( D/2,  D/2, -C/2), # 9
2265             (-D/2,  D/2, -C/2), # 10
2266             (-D/2, -D/2, -C/2), # 11
2267             ( D/2, -D/2, 0),     # 12
2268             ( D/2,  D/2, 0),     # 13
2269             (-D/2,  D/2, 0),     # 14
2270             (-D/2, -D/2, 0),]    # 15
2271    faces = [[ 0,  3,  2,  1],            # 0
2272             [ 0,  1,  5,  4],            # 1
2273             [ 2,  3,  7,  6],            # 2
2274             [ 1,  2,  6, 10,  9,  5], # 3
2275             [ 0,  4,  8, 11,  7,  3], # 4
2276             [ 4,  5,  9,  8],            # 5
2277             [ 6,  7, 11, 10],            # 6
2278             [ 8,  9, 13, 12],            # 7
2279             [10, 11, 15, 14],            # 8
2280             [11,  8, 12, 15],            # 9
2281             [ 9, 10, 14, 13],            # 10
2282             [12, 13, 14, 15]]            # 11
2283
2284    create_obj(verts, faces, 'HALF AKMON')
2285
2286  def create_flat_cross_akmon(tn):
2287    H = FLAT_CROSS_AKMON[tn][0]
2288    A = FLAT_CROSS_AKMON[tn][1]
2289    B = FLAT_CROSS_AKMON[tn][2]
2290    C = FLAT_CROSS_AKMON[tn][3]
2291    D = FLAT_CROSS_AKMON[tn][4]
2292
2293    verts = [( A/2, -D/2, -H/2),    # 0
2294             ( A/2,  D/2, -H/2),    # 1
2295             (-A/2,  D/2, -H/2),    # 2
2296             (-A/2, -D/2, -H/2),    # 3
```

```
2297            ( A/2, -D/2, -H/2+B), # 4
2298            ( A/2,  D/2, -H/2+B), # 5
2299            (-A/2,  D/2, -H/2+B), # 6
2300            (-A/2, -D/2, -H/2+B), # 7
2301            ( D/2, -D/2, -C/2),   # 8
2302            ( D/2,  D/2, -C/2),   # 9
2303            (-D/2,  D/2, -C/2),   # 10
2304            (-D/2, -D/2, -C/2),   # 11
2305            ( D/2, -D/2, C/2),    # 12
2306            ( D/2,  D/2, C/2),    # 13
2307            (-D/2,  D/2, C/2),    # 14
2308            (-D/2, -D/2, C/2),    # 15
2309            ( A/2, -D/2, H/2-B),  # 20
2310            ( A/2,  D/2, H/2-B),  # 21
2311            (-A/2,  D/2, H/2-B),  # 22
2312            (-A/2, -D/2, H/2-B),  # 23
2313            ( A/2, -D/2, H/2),    # 16
2314            ( A/2,  D/2, H/2),    # 17
2315            (-A/2,  D/2, H/2),    # 18
2316            (-A/2, -D/2, H/2),]   # 19
2317     faces = [[ 0,  3,  2,  1], # 0
2318            [ 0,  1,  5,  4], # 1
2319            [ 2,  3,  7,  6], # 2
2320            [ 1,  2,  6, 10, 14, 18, 22, 21, 17, 13,  9,  5], # 3
2321            [ 0,  4,  8, 12, 16, 20, 23, 19, 15, 11,  7,  3], # 4
2322            [ 4,  5,  9,  8], # 5
2323            [ 6,  7, 11, 10], # 6
2324            [ 8,  9, 13, 12], # 7
2325            [12, 13, 17, 16], # 8
2326            [16, 17, 21, 20], # 9
2327            [10, 11, 15, 14], # 10
2328            [14, 15, 19, 18], # 11
2329            [18, 19, 23, 22], # 12
2330            [20, 21, 22, 23]] # 13
2331
2332     create_obj(verts, faces, 'FLAT CROSS AKMON')
2333
2334   def create_hexaleg(tn):
2335     h = HEXALEG[tn]
2336
2337     verts = [( h,      h,      h),      # 0
2338            ( h,     -h,      h),      # 1
2339            ( h,      h,      3 * h), # 2
2340            ( h,     -h,      3 * h), # 3
2341            (-h,      h,      3 * h), # 4
2342            (-h,     -h,      3 * h), # 5
2343            ( h,     -h,     -h),      # 6
2344            (-h,      h,     -h),      # 7
2345            (-h,     -h,     -h),      # 8
2346            (-h,     -h,     -3 * h), # 9
2347            ( h,      h,     -3 * h), # 10
2348            ( h,     -h,     -3 * h), # 11
2349            (-h,      h,     -3 * h), # 12
2350            (-h,      h,      h),      # 13
2351            ( h,      h,     -h),      # 14
2352            (-h,      3 * h,  h),      # 15
2353            ( h,      3 * h,  h),      # 16
2354            (-h,      3 * h, -h),      # 17
2355            ( h,      3 * h, -h),      # 18
2356            ( 3 * h,  h,     -h),      # 19
2357            ( 3 * h, -h,      h),      # 20
2358            ( 3 * h, -h,     -h),      # 21
2359            ( 3 * h,  h,      h),      # 22
2360            (-h,     -h,      h),      # 23
2361            ( h,     -3 * h, -h),      # 24
2362            ( h,     -3 * h,  h),      # 25
```

```
2363            (-h,      -3 * h, -h),      # 26
2364            (-h,      -3 * h,  h),      # 27
2365            (-3 * h, -h,       h),      # 28
2366            (-3 * h,  h,      -h),      # 29
2367            (-3 * h, -h,      -h),      # 30
2368            (-3 * h,  h,       h),]     # 31
2369        faces = [[23,  8, 30, 28], # 0
2370            [ 0,  1, 23, 13], # 1
2371            [23, 13,  4,  5], # 2
2372            [ 2,  3,  5,  4], # 3
2373            [ 1, 23,  5,  3], # 4
2374            [13,  0,  2,  4], # 5
2375            [ 0,  1,  3,  2], # 6
2376            [ 7,  8,  9, 12], # 7
2377            [ 8,  6, 11,  9], # 8
2378            [12,  9, 11, 10], # 9
2379            [ 6, 14, 10, 11], # 10
2380            [14,  7, 12, 10], # 11
2381            [ 7, 13, 31, 29], # 12
2382            [14,  0, 16, 18], # 13
2383            [ 0, 13, 15, 16], # 14
2384            [17, 18, 16, 15], # 15
2385            [ 7, 14, 18, 17], # 16
2386            [13,  7, 17, 15], # 17
2387            [14,  6, 21, 19], # 18
2388            [ 6,  1, 20, 21], # 19
2389            [19, 21, 20, 22], # 20
2390            [ 0, 14, 19, 22], # 21
2391            [ 1,  0, 22, 20], # 22
2392            [ 1,  6, 24, 25], # 23
2393            [ 6,  8, 26, 24], # 24
2394            [24, 26, 27, 25], # 25
2395            [ 8, 23, 27, 26], # 26
2396            [23,  1, 25, 27], # 27
2397            [30, 29, 31, 28], # 28
2398            [13, 23, 28, 31], # 29
2399            [ 8,  7, 29,  1]] # 30
2400
2401        create_obj(verts, faces, 'HEXALEG')
2402
2403    def create_tripod(tn):
2404        h = TRIPOD[tn]
2405
2406        verts = [(h,   0,   0),    # 0
2407            (h,   h/2, 0),    # 1
2408            (h/2, h/2, 0),    # 2
2409            (h/2, h,   0),    # 3
2410            (0,   h,   0),    # 4
2411            (0,   0,   0),    # 5
2412            (h,   0,   h/2), # 6
2413            (h,   h/2, h/2), # 7
2414            (h/2, h/2, h/2), # 8
2415            (h/2, h,   h/2), # 9
2416            (0,   h,   h/2), # 10
2417            (0,   h/2, h/2), # 11
2418            (h/2, 0,   h/2), # 12
2419            (h/2, 0,   h),   # 13
2420            (h/2, h/2, h),   # 14
2421            (0,   h/2, h),   # 15
2422            (0,   0,   h),]  # 16
2423        faces = [[0,  5,  4,  3,  2,  1],  # 0
2424            [0,  6,  12, 13, 16, 5],  # 1
2425            [4,  5,  16, 15, 11, 10], # 2
2426            [0,  1,  7,  6],          # 3
2427            [3,  4,  10, 9],          # 4
2428            [13, 14, 15, 16],         # 5
```

```
2429         [6,  7,  8,  12],         # 6
2430         [12, 8,  14, 13],         # 7
2431         [9,  10, 11, 8],          # 8
2432         [8,  11, 15, 14],         # 9
2433         [1,  2,  8,  7],         # 10
2434         [2,  3,  9,  8]]         # 11
2435
2436     create_obj(verts, faces, 'TRIPOD')
2437
2438   def create_cube(tn):
2439     h = CUBE[tn]
2440
2441     verts = [(-h, -h, -h), # 0
2442          (-h, -h,  h), # 1
2443          (-h,  h, -h), # 2
2444          (-h,  h,  h), # 3
2445          ( h, -h, -h), # 4
2446          ( h, -h,  h), # 5
2447          ( h,  h, -h), # 6
2448          ( h,  h,  h)] # 7
2449     faces = [[ 0,  1,  3,  2], # 0
2450          [ 2,  3,  7,  6], # 1
2451          [ 6,  7,  5,  4], # 2
2452          [ 4,  5,  1,  0], # 3
2453          [ 2,  6,  4,  0], # 4
2454          [ 7,  3,  1,  5]] # 5
2455
2456     create_obj(verts, faces, 'CUBE')
2457
2458   def create_cob(tn):
2459     h = COB[tn][0]
2460     m = COB[tn][1]
2461
2462     verts = [( h/2, -h/2, -h/2),  # 0
2463          ( h/2,  h/2, -h/2),  # 1
2464          (-h/2,  h/2, -h/2),  # 2
2465          (-h/2, -h/2, -h/2),  # 3
2466          ( h/2, -h/2,  h/2),  # 4
2467          ( h/2,  h/2,  h/2),  # 5
2468          (-h/2,  h/2,  h/2),  # 6
2469          (-h/2, -h/2,  h/2),  # 7
2470          ( m/2, -m/2, -m/2),  # 8
2471          ( m/2,  m/2, -m/2),  # 9
2472          (-m/2,  m/2, -m/2),  # 10
2473          (-m/2, -m/2, -m/2),  # 11
2474          ( m/2, -m/2,  m/2),  # 12
2475          ( m/2,  m/2,  m/2),  # 13
2476          (-m/2,  m/2,  m/2),  # 14
2477          (-m/2, -m/2,  m/2),  # 15
2478          ( h/2, -m/2, -m/2),  # 16
2479          ( h/2,  m/2, -m/2),  # 17
2480          (-h/2,  m/2, -m/2),  # 18
2481          (-h/2, -m/2, -m/2),  # 19
2482          ( h/2, -m/2,  m/2),  # 20
2483          ( h/2,  m/2,  m/2),  # 21
2484          (-h/2,  m/2,  m/2),  # 22
2485          (-h/2, -m/2,  m/2),  # 23
2486          ( m/2, -h/2, -m/2),  # 24
2487          ( m/2,  h/2, -m/2),  # 25
2488          (-m/2,  h/2, -m/2),  # 26
2489          (-m/2, -h/2, -m/2),  # 27
2490          ( m/2, -h/2,  m/2),  # 28
2491          ( m/2,  h/2,  m/2),  # 29
2492          (-m/2,  h/2,  m/2),  # 30
2493          (-m/2, -h/2,  m/2),  # 31
2494          ( m/2, -m/2, -h/2),  # 32
```

```
2495          ( m/2,  m/2, -h/2),  # 33
2496          (-m/2,  m/2, -h/2),  # 34
2497          (-m/2, -m/2, -h/2),  # 35
2498          ( m/2, -m/2,  h/2),  # 36
2499          ( m/2,  m/2,  h/2),  # 37
2500          (-m/2,  m/2,  h/2),  # 38
2501          (-m/2, -m/2,  h/2),] # 39
2502     faces = [[0,  1,  17, 16], # 0
2503          [1,  2,  26, 25], # 1
2504          [2,  3,  19, 18], # 2
2505          [3,  0,  24, 27], # 3
2506          [1,  0, 32, 33],  #s 4
2507          [2,  1, 33, 34],  # 5
2508          [3,  2, 34, 35],  # 6
2509          [0,  3, 35, 32],  # 7
2510          [4,  5, 37, 36],  # 8
2511          [5,  6, 38, 37],  # 9
2512          [6,  7, 39, 38],  # 10
2513          [7,  4, 36, 39],  # 11
2514          [5,  4,  20, 21], # 12
2515          [6,  5,  29, 30], # 13
2516          [7,  6,  22, 23], # 14
2517          [4,  7,  31, 28], # 15
2518          [9,  8,  16, 17], # 16
2519          [10, 9,  25, 26], # 17
2520          [11, 10, 18, 19], # 18
2521          [8,  11, 27, 24], # 19
2522          [8,  9,  33, 32], # 20
2523          [9,  10, 34, 33], # 21
2524          [10, 11, 35, 34], # 22
2525          [11, 8,  32, 35], # 23
2526          [13, 12, 36, 37], # 24
2527          [14, 13, 37, 38], # 25
2528          [15, 14, 38, 39], # 26
2529          [12, 15, 39, 36], # 27
2530          [12, 13, 21, 20], # 28
2531          [13, 14, 30, 29], # 29
2532          [14, 15, 23, 22], # 30
2533          [15, 12, 28, 31], # 31
2534          [8,  12, 20, 16], # 32
2535          [12, 8,  24, 28], # 33
2536          [13,  9, 17, 21], # 34
2537          [9, 13,  29, 25], # 35
2538          [10, 14, 22, 18], # 36
2539          [14, 10, 26, 30], # 37
2540          [15, 11, 19, 23], # 38
2541          [11, 15, 31, 27], # 39
2542          [0,  16, 20, 4],  # 40
2543          [1,  5,  21, 17], # 41
2544          [1,  25, 29, 5],  # 42
2545          [2,  6,  30, 26], # 43
2546          [2,  18, 22, 6],  # 44
2547          [3,  7,  23, 19], # 45
2548          [3,  27, 31, 7],  # 46
2549          [0,  4,  28, 24]] # 47
2550     create_obj(verts, faces, 'COB')
2551
2552   def create_dolos(tn):
2553     h = DOLOS[tn][0]
2554     a = DOLOS[tn][1]
2555     b = DOLOS[tn][2]
2556     c = DOLOS[tn][3]
2557     d = DOLOS[tn][4]
2558     f = DOLOS[tn][5]
2559     I = DOLOS[tn][6]
2560
```

```
2561        m = a / (2.4142135623730951)
2562        r = m / 1.4142135623730951
2563        mb = b / (2.4142135623730951)
2564        rb = mb / 1.4142135623730951
2565        mc = c / (2.4142135623730951)
2566        rc = mc / 1.4142135623730951
2567
2568        __t = 0.01
2569
2570        verts = [( h/2-a/2+c/2,          mc/2,     -h/2),      # 0
2571             ( h/2-a/2+c/2,         -mc/2,     -h/2),      # 1
2572             ( h/2-a/2+c/2-rc,      -mc/2-rc,  -h/2),      # 2
2573             ( h/2-a/2+c/2-rc-mc,   -mc/2-rc,  -h/2),      # 3
2574             ( h/2-a/2-c/2,         -mc/2,     -h/2),      # 4
2575             ( h/2-a/2-c/2,          mc/2,     -h/2),      # 5
2576             ( h/2-a/2+c/2-rc-mc,    mc/2+rc,  -h/2),      # 6
2577             ( h/2-a/2+c/2-rc,       mc/2+rc,  -h/2),      # 7
2578             ( h/2-a/2+b/2,          mb/2,     -h/2+I),    # 8
2579             ( h/2-a/2+b/2,         -mb/2,     -h/2+I),    # 9
2580             ( h/2-a/2+b/2-rb,      -mb/2-rb,  -h/2+I),    # 10
2581             ( h/2-a/2+b/2-rb-mb,   -mb/2-rb,  -h/2+I),    # 11
2582             ( h/2-a/2-b/2,         -mb/2,     -h/2+I),    # 12
2583             ( h/2-a/2-b/2,          mb/2,     -h/2+I),    # 13
2584             ( h/2-a/2+b/2-rb-mb,    mb/2+rb,  -h/2+I),    # 14
2585             ( h/2-a/2+b/2-rb,       mb/2+rb,  -h/2+I),    # 15
2586             ( h/2,                  m/2,      -m/2),      # 16
2587             ( h/2,                 -m/2,      -m/2),      # 17
2588             ( h/2-r,               -m/2-r,    -m/2),      # 18
2589             ( h/2-r-m,             -m/2-r,    -m/2),      # 19
2590             ( h/2-d,               -m/2+__t,  -a/2-f),    # 20
2591             ( h/2-d,                m/2-__t,  -a/2-f),    # 21
2592             ( h/2-r-m,              a/2,      -m/2),      # 22
2593             ( h/2-r,                a/2,      -m/2),      # 23
2594             ( h/2,                  m/2,       m/2),      # 24
2595             ( h/2,                 -m/2,       m/2),      # 25
2596             ( h/2-r,               -m/2-r,     m/2),      # 26
2597             ( h/2-r-m,             -m/2-r,     m/2),      # 27
2598             ( h/2-d,               -m/2+__t,   a/2+f),    # 28
2599             ( h/2-d,                m/2-__t,   a/2+f),    # 29
2600             ( h/2-r-m,              a/2,       m/2),      # 30
2601             ( h/2-r,                a/2,       m/2),      # 31
2602             ( h/2-a/2+b/2,          mb/2,      h/2-I),    # 32
2603             ( h/2-a/2+b/2,         -mb/2,      h/2-I),    # 33
2604             ( h/2-a/2+b/2-rb,      -mb/2-rb,   h/2-I),    # 34
2605             ( h/2-a/2+b/2-rb-mb,   -mb/2-rb,   h/2-I),    # 35
2606             ( h/2-a/2-b/2,         -mb/2,      h/2-I),    # 36
2607             ( h/2-a/2-b/2,          mb/2,      h/2-I),    # 37
2608             ( h/2-a/2+b/2-rb-mb,    mb/2+rb,   h/2-I),    # 38
2609             ( h/2-a/2+b/2-rb,       mb/2+rb,   h/2-I),    # 39
2610             ( h/2-a/2+c/2,          mc/2,      h/2),      # 40
2611             ( h/2-a/2+c/2,         -mc/2,      h/2),      # 41
2612             ( h/2-a/2+c/2-rc,      -mc/2-rc,   h/2),      # 42
2613             ( h/2-a/2+c/2-rc-mc,   -mc/2-rc,   h/2),      # 43
2614             ( h/2-a/2-c/2,         -mc/2,      h/2),      # 44
2615             ( h/2-a/2-c/2,          mc/2,      h/2),      # 45
2616             ( h/2-a/2+c/2-rc-mc,    mc/2+rc,   h/2),      # 46
2617             ( h/2-a/2+c/2-rc,       mc/2+rc,   h/2),      # 47
2618             (-h/2+a/2+c/2,          h/2,       mc/2),     # 48
2619             (-h/2+a/2+c/2,          h/2,      -mc/2),     # 49
2620             (-h/2+a/2-c/2+rc+mc,    h/2,      -mc/2-rc),  # 50
2621             (-h/2+a/2-c/2+rc,       h/2,      -mc/2-rc),  # 51
2622             (-h/2+a/2-c/2,          h/2,      -mc/2),     # 52
2623             (-h/2+a/2-c/2,          h/2,       mc/2),     # 53
2624             (-h/2+a/2-c/2+rc,       h/2,       mc/2+rc),  # 54
2625             (-h/2+a/2-c/2+rc+mc,    h/2,       mc/2+rc),  # 55
2626             (-h/2+a/2+b/2,          h/2-I,     mb/2),     # 56
```

```
2627         (-h/2+a/2+b/2,          h/2-I,    -mb/2),      # 57
2628         (-h/2+a/2-b/2+rb+mb,    h/2-I,    -mb/2-rb),   # 58
2629         (-h/2+a/2-b/2+rb,       h/2-I,    -mb/2-rb),   # 59
2630         (-h/2+a/2-b/2,          h/2-I,    -mb/2),      # 60
2631         (-h/2+a/2-b/2,          h/2-I,     mb/2),      # 61
2632         (-h/2+a/2-b/2+rb,       h/2-I,     mb/2+rb),   # 62
2633         (-h/2+a/2-b/2+rb+mb,    h/2-I,     mb/2+rb),   # 63
2634         (-h/2+d,                a/2+f,     m/2-__t),   # 64
2635         (-h/2+d,                a/2+f,    -m/2+__t),   # 65
2636         (-h/2+r,                m/2,      -m/2-r),     # 66
2637         (-h/2+r+m,              m/2,      -m/2-r),     # 67
2638         (-h/2,                  m/2,      -m/2),       # 68
2639         (-h/2,                  m/2,       m/2),       # 69
2640         (-h/2+r+m,              m/2,       a/2),       # 70
2641         (-h/2+r,                m/2,       a/2),       # 71
2642         (-h/2+d,               -a/2-f,     m/2-__t),   # 72
2643         (-h/2+d,               -a/2-f,    -m/2+__t),   # 73
2644         (-h/2+r,               -m/2,      -m/2-r),     # 74
2645         (-h/2+r+m,             -m/2,      -m/2-r),     # 75
2646         (-h/2,                 -m/2,      -m/2),       # 76
2647         (-h/2,                 -m/2,       m/2),       # 77
2648         (-h/2+r+m,             -m/2,       a/2),       # 78
2649         (-h/2+r,               -m/2,       a/2),       # 79
2650         (-h/2+a/2+b/2,         -h/2+I,     mb/2),      # 80
2651         (-h/2+a/2+b/2,         -h/2+I,    -mb/2),      # 81
2652         (-h/2+a/2-b/2+rb+mb,   -h/2+I,    -mb/2-rb),   # 82
2653         (-h/2+a/2-b/2+rb,      -h/2+I,    -mb/2-rb),   # 83
2654         (-h/2+a/2-b/2,         -h/2+I,    -mb/2),      # 84
2655         (-h/2+a/2-b/2,         -h/2+I,     mb/2),      # 85
2656         (-h/2+a/2-b/2+rb,      -h/2+I,     mb/2+rb),   # 86
2657         (-h/2+a/2-b/2+rb+mb,   -h/2+I,     mb/2+rb),   # 87
2658         (-h/2+a/2+c/2,         -h/2,       mc/2),      # 88
2659         (-h/2+a/2+c/2,         -h/2,      -mc/2),      # 89
2660         (-h/2+a/2-c/2+rc+mc,   -h/2,      -mc/2-rc),   # 90
2661         (-h/2+a/2-c/2+rc,      -h/2,      -mc/2-rc),   # 91
2662         (-h/2+a/2-c/2,         -h/2,      -mc/2),      # 92
2663         (-h/2+a/2-c/2,         -h/2,       mc/2),      # 93
2664         (-h/2+a/2-c/2+rc,      -h/2,       mc/2+rc),   # 94
2665         (-h/2+a/2-c/2+rc+mc,   -h/2,       mc/2+rc),   # 95
2666         ( h/2-d-f,             -m/2,      -a/2),       # 96
2667         ( h/2-d-f,              m/2,      -a/2),       # 97
2668         ( h/2-d-f,             -m/2,       a/2),       # 98
2669         ( h/2-d-f,              m/2,       a/2),       # 99
2670         (-h/2+d+f,              a/2,       m/2),       # 100
2671         (-h/2+d+f,              a/2,      -m/2),       # 101
2672         (-h/2+d+f,             -a/2,       m/2),       # 102
2673         (-h/2+d+f,             -a/2,      -m/2),]      # 103
2674     faces = [[0,    1,    2,    3,    4,    5,    6,    7],  # 0
2675         [0,    8,    9,    1],                        # 1
2676         [8,    16,   17,   9],                        # 2
2677         [16,   24,   25,   17],                       # 3
2678         [24,   32,   33,   25],                       # 4
2679         [32,   40,   41,   33],                       # 5
2680         [1,    9,    10,   2],                        # 6
2681         [9,    17,   18,   10],                       # 7
2682         [17,   25,   26,   18],                       # 8
2683         [25,   33,   34,   26],                       # 9
2684         [33,   41,   42,   34],                       # 10
2685         [2,    10,   11,   3],                        # 11
2686         [10,   18,   19,   11],                       # 12
2687         [18,   26,   27,   19],                       # 13
2688         [26,   34,   35,   27],                       # 14
2689         [34,   42,   43,   35],                       # 15
2690         [3,    11,   12,   4],                        # 16
2691         [11,   19,   20,   12],                       # 17
2692         [27,   35,   36,   28],                       # 18
```

```
2693        [35,  43,  44,  36],                        # 19
2694        [4,   12,  13,  5],                         # 20
2695        [12,  20,  21,  13],                        # 21
2696        [28,  36,  37,  29],                        # 22
2697        [36,  44,  45,  37],                        # 23
2698        [5,   13,  14,  6],                         # 24
2699        [13,  21,  22,  14],                        # 25
2700        [29,  37,  38,  30],                        # 26
2701        [37,  45,  46,  38],                        # 27
2702        [6,   14,  15,  7],                         # 28
2703        [14,  22,  23,  15],                        # 29
2704        [22,  30,  31,  23],                        # 30
2705        [30,  38,  39,  31],                        # 31
2706        [38,  46,  47,  39],                        # 32
2707        [7,   15,   8,  0],                         # 33
2708        [15,  23,  16,  8],                         # 34
2709        [23,  31,  24,  16],                        # 35
2710        [31,  39,  32,  24],                        # 36
2711        [39,  47,  40,  32],                        # 37
2712        [47,  46,  45,  44,  43,  42,  41,  40],    # 38
2713        [96,  75,  67,  97],                        # 39
2714        [97,  67,  101, 22],                        # 40
2715        [22,  101, 100, 30],                        # 41
2716        [30,  100, 71,  99],                        # 42
2717        [99,  70,  78,  98],                        # 43
2718        [98,  78,  102, 27],                        # 44
2719        [27,  102, 103, 19],                        # 45
2720        [19,  103, 75,  96],                        # 46
2721        [19,  96,  20],                             # 47
2722        [20,  96,  97,  21],                        # 48
2723        [21,  97,  22],                             # 49
2724        [30,  99,  29],                             # 50
2725        [29,  99,  98,  28],                        # 51
2726        [28,  98,  27],                             # 52
2727        [67,  65,  101],                            # 53
2728        [65,  64,  100, 101],                       # 54
2729        [64,  70,  100],                            # 55
2730        [75,  103, 73],                             # 56
2731        [73,  103, 102, 72],                        # 57
2732        [72,  102, 78],                             # 58
2733        [48, 49, 50, 51, 52, 53, 54, 55],           # 59
2734        [48,  56,  57,  49],                        # 60
2735        [56,  64,  65,  57],                        # 61
2736        [72,  80,  81,  73],                        # 62
2737        [80,  88,  89,  81],                        # 63
2738        [49,  57,  58,  50],                        # 64
2739        [57,  65,  67,  58],                        # 65
2740        [73,  81,  82,  75],                        # 66
2741        [81,  89,  90,  82],                        # 67
2742        [50,  58,  59,  51],                        # 68
2743        [58,  67,  66,  59],                        # 69
2744        [66,  67,  75,  74],                        # 70
2745        [75,  82,  83,  74],                        # 71
2746        [82,  90,  91,  83],                        # 72
2747        [51,  59,  60,  52],                        # 73
2748        [59,  66,  68,  60],                        # 74
2749        [66,  74,  76,  68],                        # 75
2750        [74,  83,  84,  76],                        # 76
2751        [83,  91,  92,  84],                        # 77
2752        [52,  60,  61,  53],                        # 78
2753        [60,  68,  69,  61],                        # 79
2754        [68,  76,  77,  69],                        # 80
2755        [76,  84,  85,  77],                        # 81
2756        [84,  92,  93,  85],                        # 82
2757        [53,  61,  62,  54],                        # 83
2758        [61,  69,  71,  62],                        # 84
```

```
2759          [69,  77,  79,  71],                         # 85
2760          [77,  85,  86,  79],                         # 86
2761          [85,  93,  94,  86],                         # 87
2762          [55,  54,  62,  63],                         # 88
2763          [63,  62,  71,  70],                         # 89
2764          [70,  71,  79,  78],                         # 90
2765          [78,  79,  86,  87],                         # 91
2766          [87,  86,  94,  95],                         # 92
2767          [55,  63,  56,  48],                         # 93
2768          [63,  70,  64,  56],                         # 94
2769          [78,  87,  80,  72],                         # 95
2770          [87,  95,  88,  80],                         # 96
2771          [95,  94,  93,  92,  91,  90,  89,  88]] # 97
2772     create_obj(verts, faces, 'DOLOS')
2773
2774
2775   if __name__ == '__main__':
2776     pass
```