ALGOMAD 2010

Seminario sobre Métodos Generativos en Arquitectura y Diseño

Barcelona 1, 2 y 3 de julio 2010

Ediciones ETSAB/ETSAV, junio de 2011

Dirección: Edifici ETSAB, Diagonal 649, 08028 Barcelona
Tel 934 016 384
ISBN: 978-84-95249-53-1

Indice

Presentación

La idea de organizar este seminario surgió de una ilusión compartida. La ilusión de creer que, pese a todo, la universidad puede llegar a ser, en algunos momentos especiales, lo que está llamada a ser: un lugar en donde se desarrolle y se transmita el conocimiento, incluyendo la experimentación con técnicas que acaban de aparecer y que dificilmente podrían incorporarse a cursos establecidos.

Quienes trabajamos desde hace ya muchos años en la universidad, a tiempo completo real, sabemos que, con demasiada frecuencia, la palabra "ilusión" pierde su sentido positivo, de impulso, de creencia optimista en algo, y se hunde en el negativo, de engaño, de falta de correspondencia entre lo que imaginamos y la cruda realidad. La deriva de la universidad española, incluyendo la reciente y tortuosa adaptación al "Espacio Europeo de Educación", aún no resuelta, es una triste muestra de hasta que punto esto es así. Pero, afortunadamente, las cosas no son ni blancas ni negras. Y aunque los últimos años estén resultando algo obscuros creemos que estamos en un momento en que pueden surgir iniciativas ágiles, que dejen espacio a profesores y profesionales jóvenes y estudiantes de cursos avanzados para experimentar y poner a prueba todo tipo de ideas.

Así nació este seminario que se ha celebrado en la ETSAV (Escuela Técnica Superior de Arquitectura del Vallès) los días 1, 2 y 3 de julio de 2010.

El título "algomad" es un juego de palabras que tiene por lo menos tres intenciones. Por un lado quiere subrayar la idea de que el ordenador es "Algo más que un instrumento", pues los saltos cuantitativos a menudo posibilitan saltos cualitativos, y la rapidez y la potencia de los ordenadores nos permiten explorar terrenos que no podrían ser explorados de otro modo. Por otro lado responde a las siglas del tema fundamental del seminario: Métodos Generativos en Ar-

quitectura y Diseño. Y, en tercer lugar, siguiere (pues “mad” en inglés, como es bien sabido, significa “loco”) que los interesados en dedicar tiempo y esfuerzos a estas aventuras quizás no estén muy bien de la cabeza, habida cuenta del poco apoyo real que hay para estas iniciativas y, también, su poca relación directa con la práctica cotidiana.

En un tono un poco más académico, queremos recordar que los Métodos Generativos han ido apareciendo en los últimos años a caballo de la creciente ampliación de los recursos informáticos, tanto desde el punto de vista del hardware como del software. Y han sido objeto de un creciente interés.

Hay varias razones que pueden ayudar a explicar este interés. Una de ellas es, sin duda, el enorme desarrollo que se ha dado en la propia ciencia de la genética y que ha contribuido a que nos resulte cada vez más familiar la idea de que formas complejas y muy diversas puedan surgir a partir de componentes simples. Otra razón es la relación incipiente que se ha establecido entre las bases genéticas que incorporan información codificada y las bases de datos digitales que incorporan información que puede ser objeto de modificación automática por medio de series de algoritmos. La propia investigación genética está estrechamente ligada a la potencia de análisis combinatorio de los ordenadores actuales.

Otra razón no menos importante, dentro ya del campo específico del diseño, es la generalización de programas informáticos orientados a objetos que incorporan parámetros contextuales y que permiten modificar con facilidad, interactivamente, las dimensiones y las relaciones de los componentes básicos de elementos diversos. Para un usuario cada vez más familiarizado con este tipo de procedimientos, presentes desde hace ya algunos años en programas como AutoCad, Microstation, Rhinoceros o 3D Studio Max (por no mencionar sino los más ubicuos en nuestras escuelas) resulta natural plantearse la posibilidad de “hacer evolucionar” un elemento mediante pruebas semiautomáticas concatenadas.

Cuanto estos procedimientos resultan habituales, el paso siguiente, que también es fácil de imaginar sin necesidad de haberse adentrado en los complejos vericuetos de la historia reciente y en el trabajo de algunos investigadores pioneros, es activar estos procedimientos por medio de algoritmos personales más o menos simples.

Esta posibilidad, que ahora está mucho más al alcance de la mano que hace años, requiere sin embargo cierta dedicación, un mínimo de habilidades informáticas que pasan por utilizar algún lenguaje de programación. Pero son habilidades que se adquieren con facilidad si uno está interesado en dar este paso e investigar las posibilidades de “hacer crecer las formas” en direcciones más o menos controladas.

Con esto se abre un camino fascinante y lleno de interrogantes a los que sólo

podemos apuntar en esta breve introducción.

Pues, por un lado, de un modo relacionado más directamente con la práctica profesional, los métodos algorítmicos permiten ensayar soluciones que serían inabordables por métodos tradicionales. Pensemos, por ejemplo, en el diseño de una serie de paneles que queremos que se adaptan a superficies curvas o de geometría complicada, paneles que pueden incluir elementos de diferentes grados de complejidad. Si queremos estudiar diferentes alternativas, es sencillamente imposible en circunstancias normales, plantearse el modelado de estos elementos uno por uno y su reelaboración, uno por uno, cada vez que el proyecto obligue a cambiar la superficie de base, sea porque la concepción del proyecto general lo exige, sea porque el cliente lo exige, sea porque las soluciones constructivas lo exigen, sea por las razones que sean. Pero si una estructura semejante está generada por un algoritmo complejo, bastará cambiar los parámetros, reiniciar el proceso y, si es necesario, dejar el ordenador trabajando toda la noche.

Por otro lado más especulativo, relacionado de un modo menos directo con la práctica profesional corriente, estos mismos métodos algorítmicos pueden entroncar con toda una serie de movimientos artísticos, científicos y culturales, que han ido surgiendo hace ya muchos años, en los que la exploración de los procesos, la investigación de los mecanismos autorreguladores de estructuras complejas, el uso de sistemas con reglas propias que obligan a poner entre paréntesis la idea de “Autor”, el énfasis en la idea de cooperación y en la reflexión sobre el modo real en que realmente se da la emergencia de obras de arquitectura, como una actividad colectiva, todo esto hace que el sentido de estas experiencias, si se reflexiona sobre sus posibles implicaciones, pueda ir mucho más alla que el descubrimiento con nuevas técnicas.

Este sería el marco en que se inserta este seminario. Pero, volviendo al comienzo, lo que nos ha importado, lo que esperamos que tenga cierta continuidad, es poder encontrarnos un grupo de arquitectos, ingenieros, diseñadores, que comparten un mismo interés por curiosear, por experimentar, por jugar y por intercambiar experiencias.

Isabel Crespo, Javier Monedero

Presentación de los talleres

Uno de los objetivos fundamentales de Algomad 2010 fue acercar a un público de carácter general el uso y aplicación de las distintas herramientas de diseño generativo para arquitectura existentes en la actualidad. La apuesta de Algomad 2010 ha sido decididamente instrumental, pues se buscó el aprendizaje de la herramienta a través de ejemplos prácticos más que la aplicación de aquella a una temática concreta. Esto supone una diferencia respecto a otros seminarios y talleres de la materia, en los que la herramienta es el medio para un proceso de especulación creativo, formal o tecnológico.

Algomad 2010 ha sido mucho más austero en sus pretensiones, el seminario sólo buscó acercar al mayor número de gente con unos conocimientos mínimos de CAD, al uso de herramientas que después podrían aplicar en sus exploraciones y proyectos personales, tanta a nivel académico como profesional.

No ha sido objetivo de Algomad 2010 constituir una experiencia cerrada, entendida como algo con inicio, desarrollo y resultados, sino que se ha buscado más bien inocular en los asistentes el interés por las herramientas de diseño paramétrico-generativo y ayudarles en los primeros pasos, esos que más cuestan, para que después las preguntas y necesidades surjan y se multipliquen, para que cada uno eligiera su camino y aplicase el conjunto de herramientas ajustándolas a sus necesidades.

El formato elegido para cumplir estos objetivos se ha estructurado en la forma de 4 talleres referidos a otras tantas herramientas, 4 seminarios de 8 horas entendidos como píldoras de conocimiento muy concentrado que permitieran a los asistentes entender el marco general y la mecánica interna de cada herramienta

y así poder indagar con garantías de una manera más personal.

Los talleres se han desarrollado en dos únicos días, lo que obligó a los asistentes a elegir una combinación de dos de ellos, siendo esta una manera que permitía a cada uno definir un itinerario personal a la vez que posibilitaba mantener la duración de todo el evento dentro del breve lapso de 3 días que tomó, eliminando de paso la posibilidad de que algún interesado se viera obligado a asistir a un curso sobre algo que ya conocía.

En la elección de las herramientas objeto de los talleres se ha buscado cubrir el espectro más amplio posible junto con una voluntad clara de independencia en la plataforma, apostando básicamente por aquel software que, a juicio de los organizadores, más aceptación tiene a nivel internacional. Así pues, se optó por Grasshopper (Rhino) y Generative Components (Microstation) como herramientas de diseño paramétrico y programación simbólica, por Rhinoscript (Rhino) como entorno de programación de geometría 3D y por Processing (Java) como librería de programación para visualización, interactividad y gráficos.

La estructura y metodología de cada taller fue consensuada previamente entre los tutores, el caso práctico fue el elemento central de cada taller y se acordó destinar una parte sustancial de tiempo al desarrollo de ejercicios con un carácter más personal o ajustados a intereses más particulares, lo que permitió que cada asistente desarrollara sobre sus intereses o como mínimo pusiera algo de sí mismo en los ejercicios.

Roberto Molinos

Taller 1

Generative Components

Carlos de la Barrera

GenerativeComponents (GC), es un sistema de diseño paramétrico y asociativo que permite explorar diferentes alternativas de diseño de manera rápida y económica. GC captura las intenciones del diseño por medio de una representación gráfica, un diagrama de relaciones y por medio de código.

GC puede ser visto como una caja de herramientas, en la cual existen una serie de objetos como puntos, polígonos, curvas, superficies, hojas de cálculo, cables para conectarse con otros programas, textos, vectores, elipses, solidos, planos, círculos, sistemas de coordenadas, etc. Cada uno de estos objetos, tienen características, propiedades y están diseñados para agruparse y formar objetos más complejos. La principal ventaja está en que estos objetos pueden ser creados mediante interfaz gráfica o programación.

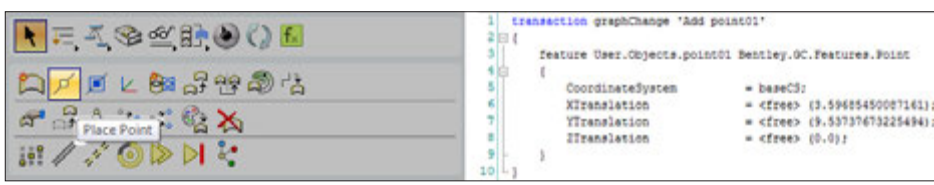


Figura 1. Izquierda. Botón para crear un punto: Interfaz gráfica. Derecha: Código para crear un punto en el espacio cartesiano: programación.

Objetos y Clases

Los objetos son la clave para entender la estructura de GC y para entender la Programación Orientada a Objetos (POO). La POO se basa en la estructura del mundo real donde todas las cosas que vemos comparten dos estados: propiedades y comportamiento.

En el mundo real, así como también en programación, los objetos varían en complejidad. Por ejemplo, una lámpara de mesa tiene en su comportamiento dos estados posibles, encendido y apagado. En cambio una radio/cd tiene más estados en su comportamiento; encendido y apagado, volumen, modo (cd o radio), posición de la antena, etc. Si se mira con atención se notará que estos objetos están compuestos por otros objetos que también tienen sus propiedades y comportamientos. Está abstracción del mundo real es la programación orientada a objetos.

Los software están diseñados conceptualmente según los objetos del mundo real, basados en propiedades y comportamientos. Un objeto en programación almacena estos dos estados en variables (propiedades) y en funciones o métodos (comportamientos). Las funciones operan internamente en los objetos y sirven como primer mecanismo para comunicarse con otros objetos.

En la POO se llama encapsulamiento a esconder los datos miembros de un objeto de manera que sólo se puedan modificar las operaciones definidas para

ese objeto. Cada objeto está aislado de su exterior, de modo natural, y la aplicación entera es un rompecabezas de objetos compilados (DLL). El aislamiento del objeto protege los datos asociados a un objeto de su modificación por quien no tenga derecho a acceder a ellos. De esta manera el usuario de este objeto puede olvidarse de la implementación de nuevas funciones y métodos para concentrarse sólo en como usarlos y evitando cambiar sus estados de manera imprevista.

Para trabajar con objetos lo primero que se necesita es crearlos. Para esto, es preciso contar con el plano o modelo del objeto, que recibe el nombre de clase. Una clase es una representación abstracta de un objeto y de como se debe construir. Por ejemplo, si una casa es un objeto con sus características y comportamientos, la clase del objeto sería el plano para construir la casa. En la clase aparecen las propiedades de la ventana, como el largo, el ancho y el alfeizar. Además deberían aparecer los comportamientos de la ventana, por ejemplo si está abierta o cerrada o si es abatible o corredera.

En la programación orientada a objetos, la clase es un constructor que es usado como un plan para crear objetos. Este plan incluye atributos (propiedades) y métodos (comportamientos) que han creado otros objetos. Una clase encapsula el estado y comportamiento de lo que representa conceptualmente. Los atributos son encapsulados en las llamadas variables miembro y su comportamiento a través de código reusable implantado en métodos. Sin los planos no podemos construir la casa, de la misma manera que un objeto no puede ser construido sin sus clases o clase.

La POO además de ser una tecnología de programación ha impulsado corrientes de pensamiento sobre cómo el hombre se relaciona con los objetos a su alrededor. De cómo todo lo que nos rodea podría llegar a ser virtual en algún momento. De cómo los objetos son construidos a partir de otros objetos más simples y estos a su vez de otros aún más simples. de cómo se relacionan entre sí para generar objetos de comportamiento complejo que son mucho más que la simple suma de sus componentes.

El taller de GC fue impartido en dos mañanas, durante el seminario de Algomad 2010. A continuación se describen los conceptos y ejercicios desarrollados durante el curso.

Taller de Generative Components, primer día: Conceptos

GC tiene una curva de aprendizaje muy parecida a los lenguajes de programación, lo que implica trabajar y aprender durante un tiempo antes de comenzar a ver progreso en el aprendizaje. GC es un sistema de clases geométricas, aritméticas y de utilidades, que usa como interfaz el programa Microstation de Bentley para representar la geometría y las formas. Incluye un diagrama simbó-

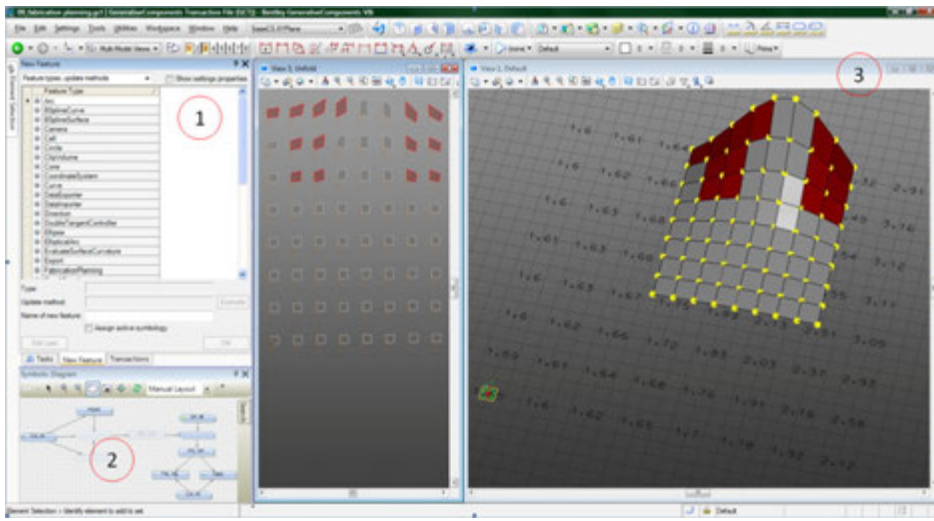


Figura 2. En la zona 1 se encuentra la librería de clases para construir todos los objetos que provee GC. Existen clases (Features) para construir puntos, círculos, exportar datos a Excel, etc. En la zona 2 se encuentra el diagrama simbólico donde se muestran todas las relaciones y jerarquías que existen en el modelo geométrico. En la zona 3 es donde se construye la geometría. En este caso existen dos ventanas: Una para contener la superficie y la segunda (izquierda) para mostrar el despiece de cada componente. Los componentes rojos, en la superficie y en el despiece indican que no son planos.

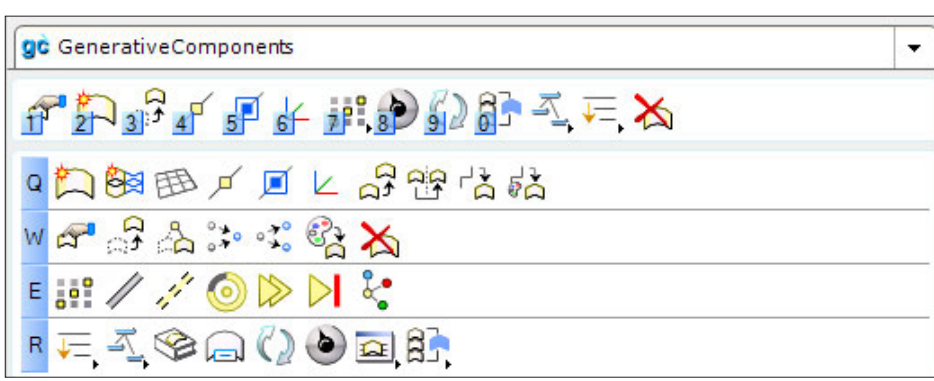


Figura 3. Barra de herramientas de GC

lico para representar las relaciones y jerarquías entre las clases y una barra de herramientas que permite acceder a la lista de todas las clases.

Una particularidad que posee GC es que es autodocumentado lo que significa que a medida que el modelo se va construyendo suceden dos cosas: la primera es que el diagrama simbólico va representando el árbol de clases y la

segunda es que se va construyendo el código que hace posible la forma de la geometría. De esta manera se puede revisar el árbol desde el código, desde el diagrama simbólico o desde la ventana de transacciones. El concepto de GC es dar a los usuarios la habilidad de crear sus propios componentes y relaciones. En este sentido GC es una buena herramienta para hacer herramientas. Para diseñar, primero se deben construir las herramientas que se usarán durante el proceso. Además se deben crear todos los pasos necesarios para construir la geometría, la cual debe ser suficientemente robusta para soportar variaciones geométricas, lo que es esencial en el proceso de exploración.

En GC todo es programación aunque no lo veamos. Cualquier cosa que hagamos, por simple que sea, creará un trozo de código que luego podremos editar. Cuando creas un *feature* por botones o cuando ingresas un valor dentro de un campo, lo que haces es crear código, ya que toda la geometría es guardada en el archivo GCT que no es otra cosa que un archivo de texto sin formato.

Otra manera de generar código dentro de GC es cuando llenas un campo del tipo *Expression* al crear un *Feature*. Dentro de un campo *Expression*, puedes ingresar un valor, el nombre de una variable, una expresión compleja, inicializar o definir un nuevo *Feature*, etc. La única condición es que la expresión este contenida en una sola línea.

EJEMPLOS DE EXPRESIONES

```
5                                          //valor
numdePuntos                                //nombre de una variable
5/(line01.Length + 1)                      //expresiones complejas o más complejas que  esta
Line01.Length * ((i > 12) ? 1 : -1)        // estructuras condicionales
{3,5,12,7,8,9}                             //listas
new Point().ByCartesianCoordinates(baseCS, 5, 5, 0)  //inicializar features
```

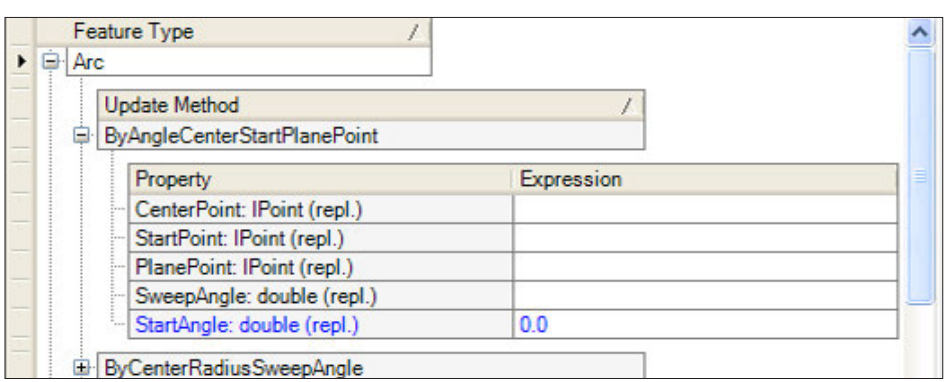


Figura 4. Ejemplo del campo Expression para crear Features. Al rellenar la información que falta dentro del campo se crea el objeto Arc. El Update Method es la clase que construye el objeto Arc. Existen varias maneras de crear un arco, si la información se ingresa correctamente se creará el arco.

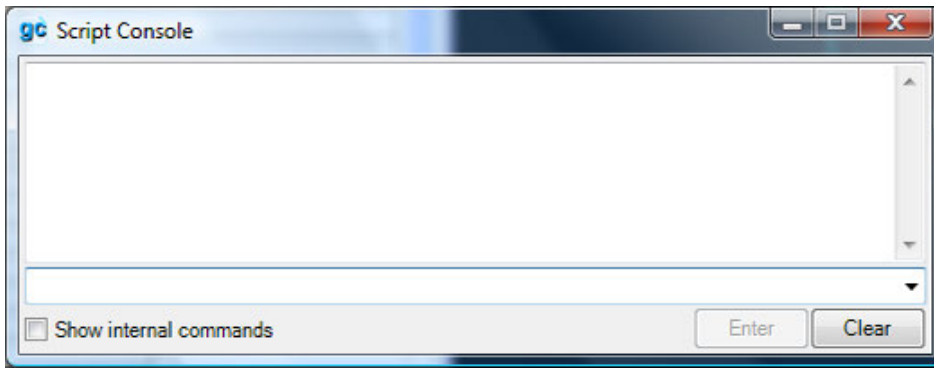


Figura 5. Está herramienta funciona con un campo de expresiones, donde se escribe la instrucción. Posee una pestaña, la cual guarda las expresiones ejecutadas. Un área mayor donde son mostrados los cálculos, y errores de las expresiones. Dos botones. Un Enter para ejecutar las expresiones y un Clear para limpiar el área de resultados.

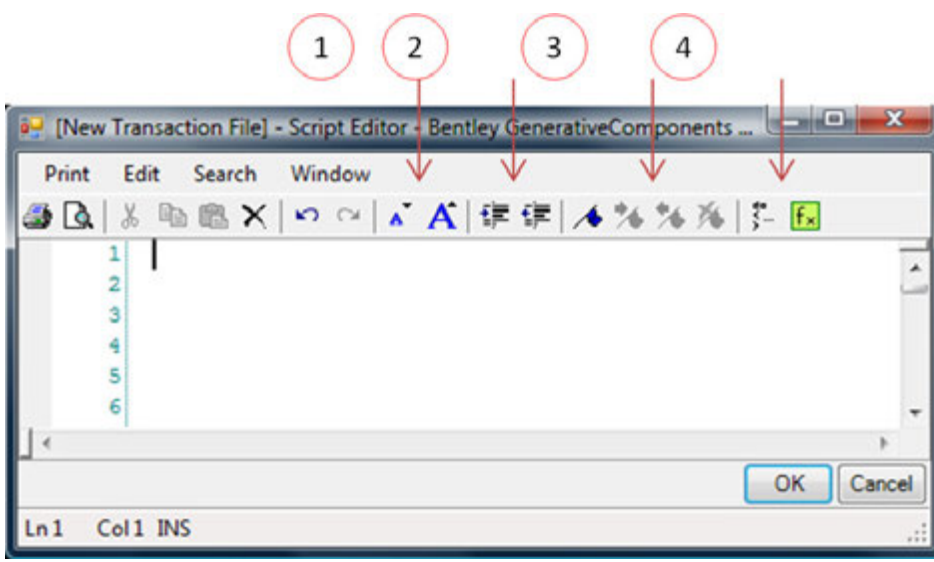


Figura 6. Editor de texto de GC, en él se escribe el código para manipular y crear nuevos Features. Mediante el editor de texto se puede revisar y editar el archivo completo de transacción.

La siguiente manera de generar código, dentro de GC, es bastante particular, ya que no genera código dentro del archivo GCT, sino que más bien puede ser usado para probar expresiones como las anteriores y obtener y comprobar información del modelo, Esta particular herramienta se llama *Script Console*.

EL *Script Console* manipula sólo expresiones y declaraciones que se ajusten a una sola línea de entrada. por lo que no leerá una estructura de función ni la ejecución de un bucle de varias líneas. Para obtener tu resultado presiona *Enter*. Sin embargo al editor de texto por excelencia de *GenerativeComponents* se accede desde este botón: 

El editor de texto de GC, tiene una serie de funciones y herramientas para facilitar el trabajo de programación. Dentro de las más relevantes se pueden destacar:

1 Los botones que sirven para agrandar y empequeñecer la letra. Muy útil cuando se muestra código a través de un proyector o cuando la vista está cansada.
2 El orden automático del GC a veces no basta, por lo que es ideal poder ordenar el código a nuestro antojo, para facilitar el entendimiento, sobre todo cuando se trabaja con bucles anidados.
3 Marcar puntos dentro del código es útil cuando es muy largo. Se puede ir saltando de función en función, para no tener que estar buscando cada una de ellas.
4 El primero de estos botones se llama *Builder Statement* y contiene una lista desplegable con una serie de estructuras de programación pre-construidas, donde lo único necesario es elegir el tipo y rellenar los valores o datos que faltan. Muy útil para evitar tener que escribir muchas veces las mismas estructuras y ahorrarse errores tontos. El botón que lo sigue es el de los recursos que dispone GC. Dentro del apartado de *Functions* es posible encontrar una larga lista de funciones pre-construidas con diversas aplicaciones. Como por ejemplo, cálculos trigonométricos, distancias, métodos para matrices, etc. etc.

GCScript se usa principalmente para personalizar, generar o modificar un *Feature*. Un buen ejemplo es cuando necesitamos ahorrar pasos para generar una geometría más compleja. Otro ejemplo es el de disponer *Features* de una manera personalizada dentro del espacio. Dentro de GC, no creo que exista un límite bien definido entre script y “botón”. Lógicamente no todo será “botón” ni tampoco todo será script. Hay un balance a encontrar, donde juegan experiencia, tipo de proyecto, lógica y gusto.

Conceptos básicos de GCScript (y otros lenguajes de programación).

C#

C# es un lenguaje de programación orientado a objetos desarrollado por Microsoft como parte de la iniciativa .NET y luego aprobada como estándar por ECMA e ISO. El lenguaje C# se basa en procedimientos, es orientado a objetos y de sintaxis basada en C++ e incluye una serie de aspectos de lenguajes de programación como Delphi y Java, pero con mayor énfasis en la simplificación.

TOP LEVEL FEATURE

Los *Top level Feature*, son *Features* mostrados en su propio nodo en la vista simbólica y tienen un nombre tal como “point01”. Estos requieren la mayor cantidad de memoria y permiten el conjunto completo de interacciones provista en GC. Los *Top Level Features* son por defecto el tipo de *Feature* creado en el modelado de diseño básico.

TEMPORARY FEATURE

Los *Temporary Features* pueden ser creados con *scripting*. Se crean sólo localmente, en la función, y siempre que la función se esté ejecutando; luego son eliminados. Los *Temporary Features*, se usan corrientemente en scripts para crear puntos intermediarios, por ejemplo para construir alguna otra forma. Estos no aparecen en la vista simbólica, pero puede ser completamente manipulados como un *Feature de GenerativeComponent*.

CHILD FEATURE

La replicación de un punto es un ejemplo de un *Child Feature*.

REFLECTION

El proceso por el cual un programa puede leer sus propios METADATA (*Reflection* es la habilidad que tiene un código para revisar su propia estructura, es decir, poder revisar la METADATA que está en el *assembly*, en el caso de .NET, y manipularla). Con *Reflection* podemos encontrar cualquier detalle sobre un objeto. Se dice que el programa se refleja en sí mismo, extrayendo sus metada desde su código y usándolos tanto para informar como para modificar su propio comportamiento. Figurativamente hablando, es cuando se dice que el programa tiene la habilidad de “observar” y la posibilidad de modificar su propia estructura y comportamiento.

FUNCTION

Una función representa un bloque independiente de declaraciones de *GCScript*. En la ciencia de computación, una función, subrutina, método, procedimiento o subprograma es una porción de código dentro de una larga estructura. Una función desarrolla una tarea específica y puede ser relativamente independiente del código restante. La sintaxis de muchos lenguajes de programación incluye soporte para la creación de subrutinas auto contenidas, y para llamarlas y retornar datos desde ellas. Una función tiene entradas y casi siempre salidas.

ENUM

La palabra clave *enum* se utiliza para declarar una enumeración, un tipo distinto que consiste en un conjunto de constantes con nombre, denominado lista de enumeradores. Cada tipo de enumeración tiene un tipo subyacente, que puede ser cualquier tipo integral excepto *char*. El tipo predeterminado subyacente de los elementos de la enumeración es *int*. De forma predeterminada, el primer enumerador tiene el valor 0 y el valor de cada enumerador sucesivo se incrementa en 1 (ver http://msdn.microsoft.com/es-es/library/sbbt4032%28VS.80%29.aspx [2010/02/10]).

Ejemplo:

```
enum Days {Sat, Sun, Mon, Tue, Wed, Thu, Fri};
```

NAMESPACE

Es un contenedor abstracto o de entorno creado para contener una agrupación lógica de identificadores o símbolos únicos (ejemplo nombres). Un identificador definido en un *namespace* queda asociado a ese *namespace*. El mismo identificador puede ser independientemente definido en múltiples *namespaces*. Su significado asociado con un identificador definido en un *namespace* puede o no puede tener el mismo significado en un *namespace* diferente.

Por ejemplo, “curva L” pertenece a la sección norte del proyecto, “curva L” pertenece a la sección sur del proyecto. La razón de porque la misma “curva L” pertenece a diferentes secciones es porque pertenecen a diferentes compañías las cuales son representadas por diferentes *namespaces*. Los *namespaces* han sido creados para organizar los objetos dentro del programa.

Componentes básicos de GCScript

GCScript es un robusto lenguaje de programación basado en C# que proporciona estructuras condicionales, declaraciones, bloques, funciones y sub-funciones, objetos y métodos, argumentos pasados por valor o por referencia, un rico

grupo de expresiones de operadores y muchos otros aspectos. *GCScript* provee una herramienta única que incluye la replicación automática. Esta consiste en la habilidad de usar listas o grupos de listas anidadas, donde quiera que un solo valor es esperado. *GCScript* es el pegamento que conecta todos los *Features* en un modelo de *GenerativeComponents* y aparece en una serie de diferentes niveles.

Cuando escribimos código en GC todo lo que hacemos es tomar datos y aplicarlos dentro de una función. En realidad lo que estamos haciendo es hacer cálculos y aplicarlos sobre los *UpdateMethods* con el fin de crear nuevos *Features* o manipular los existentes. Casi todos los *Features* en GC tienen un *UpdateMethod* llamado *ByFunction*. Este es el método que se usa cuando se programa en GC. Es posible personalizar funciones que podrán ser usadas como nuevos *UpdateMethods*, definir qué es lo que va hacer la función, que datos necesitamos para que la función "funcione" y que datos esperamos de la función, crear *Features*, hacer cálculos, definir condiciones geométricas y estructuras condicionales, exportar e importar datos a otros programas. Todas estas tareas son escritas dentro de *Function: function field*.

Las funciones proveen una muy buena manera de organizar el código en módulos para realizar tareas específicas. Escribir funciones hace el código más reusable y estás pueden ser llamadas repetidamente una y otra vez. Cuando especificamos una función dentro de un *Feature* utilizamos el *UpdateMethod ByFunction*. Sin embargo no se puede llamar esta función repetidamente. El diseño de estas funciones está pensado para desarrollar una tarea única. Por el contrario, si sabemos que la función que escribimos será llamada repetidamente, podemos escribirla dentro de su propio *Feature*, llamándola cuantas veces queramos. *GraphFunction -> Define*. Dentro de la definición podemos escribir nuestra funcion, darle un nombre y una tarea a realizar. La nueva función estará inmóvil hasta que no sea llamada dentro de un *UpdateMethod ByFunction*.

Cada vez que escribimos una función lo hacemos para crear un tipo de *Feature* existente por medio de un *UpdateMethod* personalizado. GC espera como salida sólo el tipo de *Feature*. Pueden ser línea o líneas, circulo o círculos, tantos como queramos, pero todos del mismo tipo. Así que cuando creas una línea por el *UpdateMethod ByFunction*, aunque puedes crear diferentes tipos auxiliares de *Features* dentro de la función, GC espera que el resultado sea una línea o una lista de líneas.

Dentro de *GenerativeComponents* es posible distinguir tres tipos de funciones:

1 Funciones locales, que se encuentran en un ámbito local, estás sólo pueden ser llamadas dentro del ámbito de la función. Son los *UpdateMethod ByFunction* de los *Features*.

2 Funciones globales, son aquellas que pueden ser llamadas desde cualquier lugar dentro del código o archivo de transacción. GC tiene un número grande de funciones globales construidas, como *Sin*(), *Series*(), *Random*(), etc. Las funciones globales son las funciones que escribimos dentro del *Feature GraphFunction*.
3 Funciones anónimas, son funciones dentro de una expresión o declaración que no requieren asociarles un nombre, por esto se llaman anónimas.

EJERCICIO SIMPLE PARA CREAR UNA FUNCION LOCAL

1 Crea dos puntos de manera tradicional sobre el espacio de trabajo.
2 Create Feature -> Feature type -> Point -> UpdateMethod -> ByFunction.
3 Dentro del editor de texto escribe esto:

```
function (CoordinateSystem cs, Point pt1, Point pt2)
{
  Point myPoint = new Point(this);
  double xpos = (pt1.X + pt2.X) / 2;
  double ypos = (pt1.Y + pt2.Y) / 2;
  myPoint.ByCartesianCoordinates(cs, xpos, ypos, 0);
  return myPoint;
}
```

En FunctionArguments incluye esto:

```
{baseCS, point01, point02}
```

EJERCICIO SIMPLE PARA CREAR UNA FUNCION GLOBAL

1 Create Feature -> Feature type -> GraphFunction -> UpdateMethod -> Define.
2 Escribe el siguiente código en el editor de texto.

```
function (int i)
{
  Point myCrazyPoint = new Point("myCrazyPt" + i);
  myCrazyPoint.ByCartesianCoordinates(baseCS, Random(1, 10), Random(1, 10), Random(1, 10));
}
```

3 Llama a la función con este nombre:

```
myfuncCrazyPoint
```

4 Guarda la transacción y luego abre el editor de texto de GC para acceder al código del archivo. Escribe esto:

```
transaction script 'calling the function'
{
  for (int i = 0; i < 10; ++i)
  {
    myfuncCrazyPoint(i);
    UpdateGraph();
  }
}
```

Guarda la transacción y ejecuta el código. Verás cómo se generan cada uno de los 10 puntos aleatorios.

EJEMPLO DE UNA FUNCION ANONIMA

```
function(a, b) { return a + b; }
```

La función anónima no tiene nombre y retorna la suma de a+b.

```
transaction script 'anonymous function' {
  int[] data = {3, 5, 7, 8, 9, 2, 12, 35, 27, 9, 567, 149};
  int[] largeValues = data.Select(function(x){return x > 50;});

  Print(largeValues);
}
```

Se imprimirán los valores mayores que 50 dentro de la lista.

RETORNAR VALORES DE FUNCIONES

La declaración *return* es usada para indicar lo que es retornado desde la función. El tipo devuelto por la función debe ser el mismo especificado en la cabecera de la función. Sólo un valor puede ser retornado desde la función, aunque este puede ser un array de múltiples dimensiones.

La otra manera de pasar más de un valor fuera de la función es usando *ref* o *out* como argumentos de la función. Las funciones que no retornan ningún valor, son conocidas como funciones *void*.

```
Void manage()
```

REF y OUT

Cualquier función en cualquier contexto, puede tener argumentos *ref* y/o *out*. Estás palabras reservadas son herramientas adicionales que ayudan al arquitecto a expresar sus ideas en el código.

Cualquier argumento en una función puede ser *ref*, *out* o ninguno:

- Un argumento que no es *ref* o *out* es sólo una entrada a la función.
- Un argumento que es *out* solamente es una salida de la función.

- Un argumento que es *ref* es ambos, entrada y salida de la función.

Escribe el siguiente código:

```
transaction script 'usando ref y out'
{
  void testing1(int a, int b, int c) {
    a += 3;
    b += 3;
    c += 3;
  }

  void testing2(int a, ref int b, int c) {
    a += 3;
    b += 3;
    c += 3;
  }

  void testing3(ref int a, out int b, out int c) {
    a += 3;
    b += 3;
    c += 3;
  }

  void callingFunctions() {
    int a = 3, b = 4, c = 5;

    testing1(a, b, c);
    PrintFormat("salida de function 1:"+" a={0}, b={1}, c={2}", a, b, c);

    testing2(a, ref b, c);
    PrintFormat("salida de function 2:"+" a={0}, b={1}, c={2}", a, b, c);

    testing3(ref a, out b, out c);
    PrintFormat("salida de function 3:"+" a={0}, b={1}, c={2}", a, b, c);
  }

  callingFunctions();
}
```

Salida

```
salida de function 1: a=3, b=4, c=5
salida de function 2: a=3, b=7, c=5
salida de function 3: a=6, b=3, c=3
```

La primera función pasa los argumentos de manera tradicional, por lo que las modificaciones de los valores, permanecen dentro de la función y sus valores

de fuera permanecen igual. Cuando se llama a la segunda función, *b* es pasado como referencia y modifica el valor original de la variable fuera de la función.

Desde ahora *b* tiene un valor de 7. Finalmente cuando se llama a la última función, *a* es modificado porque se pasa como referencia. Las otras dos variables son pasadas como *out* y entran a la función con un valor de 0, que sumado dan 3 como resultado esperado.

Aparte de *ref* y *out* algunos valores de las funciones globales predeterminadas pueden ser opcionales, como es el caso de las funciones *Random*() o *Round*(), entre otras. Usualmente los valores opcionales se denotan con corchetes [valor opcional] dentro de la función.

GenerativeComponents es un universo de geometría e interoperabilidad que es imposible de explicar y entender en 8 horas de clases. La idea de la primera mañana de clases fue impartir nociones generales de cómo opera GC y la POO. El trabajo en GC es muy cercano a la programación en C#.NET por lo que los conceptos explicados durante la primera sesión serán válidos para cualquier lenguaje o programa basado en esta tecnología.

Taller de GenerativeComponents, segundo día: Proyectos

EJERCICIO 1, SIMPLES RELACIONES

El primer ejercicio consistió en crear simples relaciones de distancia entre un punto y una lista de puntos de la cual salían una serie de líneas verticales.

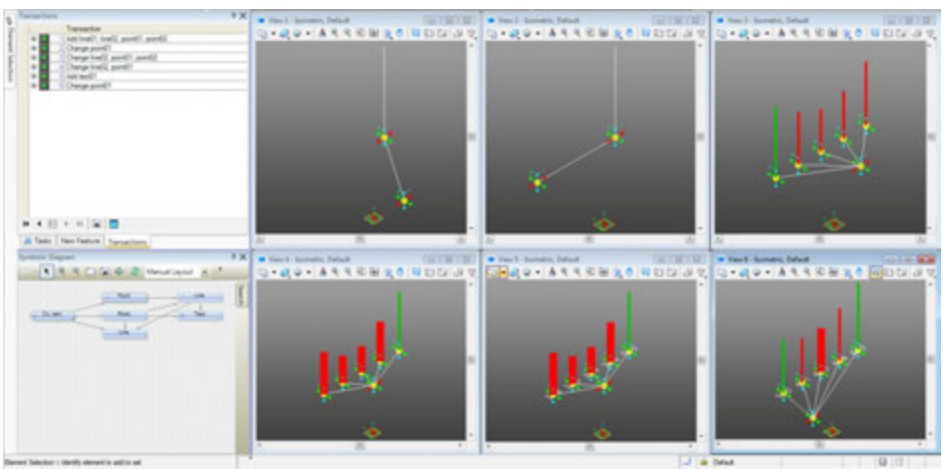


Figura 7. Ejercicio 1. Simples relaciones

EJERCICIO 2, COORDENADAS CILÍNDRICAS

Las generaciones de puntos por medio de Seno y Coseno, están ampliamente vistas en libros de geometría computacional. Si quieres encontrar buenos ejemplos de código investiga el *Microstation GenerativeComponents V8i Essentials* 08.11.05.36 (o la versión de GC actual). Esta es una pequeña introducción de generación de puntos por medio de coordenadas cilíndricas. El código generará una espiral en cada punto.

ECUACIÓN PARA GENERAR LAS ESPIRALES:

```
c = 0;
inc = 0.5;
rad = 0.5;

for (double theta = 0; theta < 1840; theta += 10) {
Xpos = rad += 0.005;
Ypos = theta;
newPt2[c] = new Point();
newPt2[c].ByCylindricalCoordinates(cs2, Xpos, Ypos, inc += 0.02);
c++;
}
```

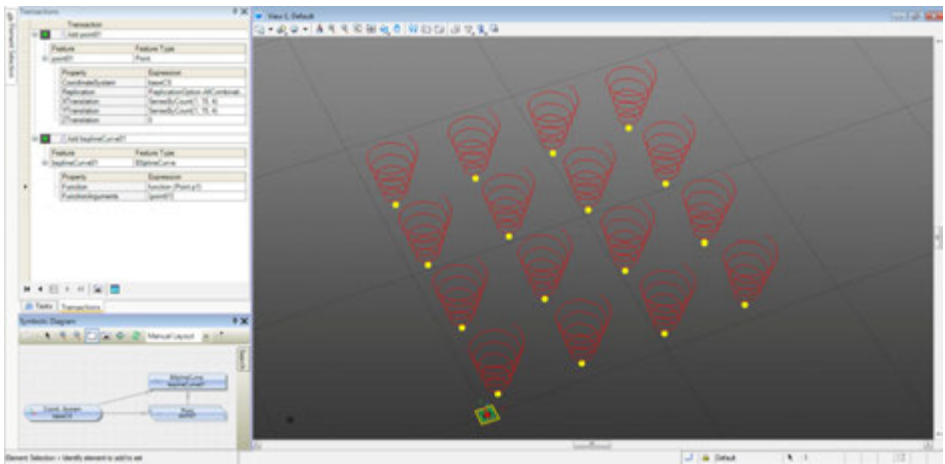


Figura 8. Ejercicio 2. Coordenadas cilíndricas

EJERCICIO 3, ATRACTORES

El uso de atractores en arquitectura es un tema cada vez de mayor importancia. Tiene múltiples aplicaciones en arquitectura y urbanismo. Por ejemplo para controlar la altura de edificios en un centro urbano o el asoleamiento en fachadas. Usualmente este tipo de problemas se suele describir con una curva no polinómica del tipo x = 1/y. Para el ejercicio desarrollado en Algomad se utilizó una curva signoidal usada en estadística económica.

ECUACIÓN PARA LOS RADIOS DE LOS CIRCULOS:

```
feature User.Objects.circle01 Bentley.GC.Features.Circle
{
 CenterPoint = point02;

 Radius = A1+(A2-A1)/(1 + Exp(Log(1/S-1)*(B1+B2-2*Distance(point01,point02)) /(B2- B1)));

    Support = baseCS.XYPlane;
}
```

EJERCICIO 4, LAW CURVES

Una *law curve* describe una relación entre el eje X e Y en un gráfico. Así se tiene el control del comportamiento de un *Feature*, en este caso un superficie.

EJERCICIO 5, SUPERFICIE REACTIVA

Obtener información, digital o real, del medio ambiente para informar el modelo

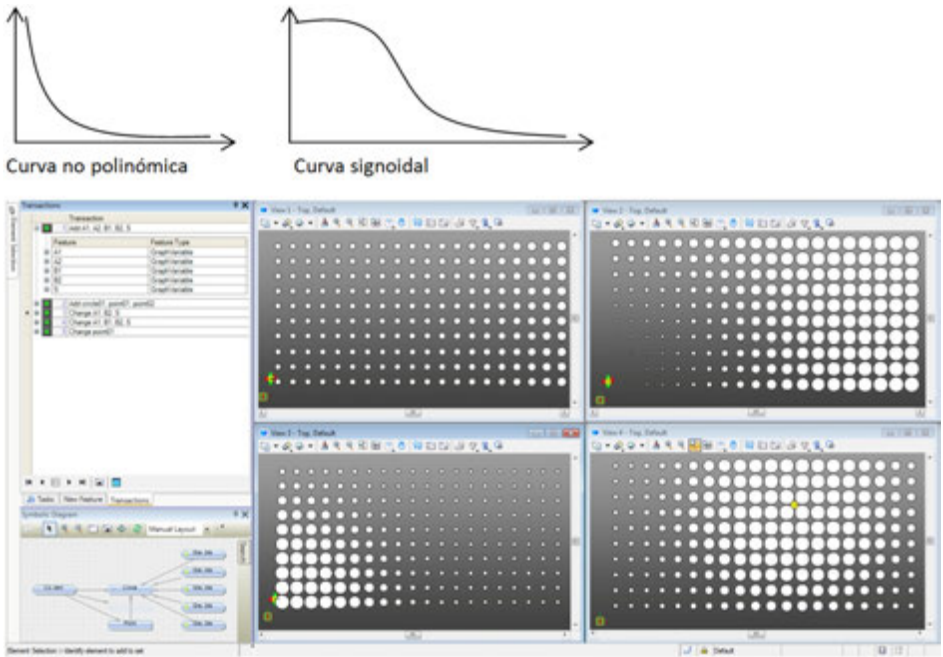


Figura 9. Ejercicio 3. Atractores

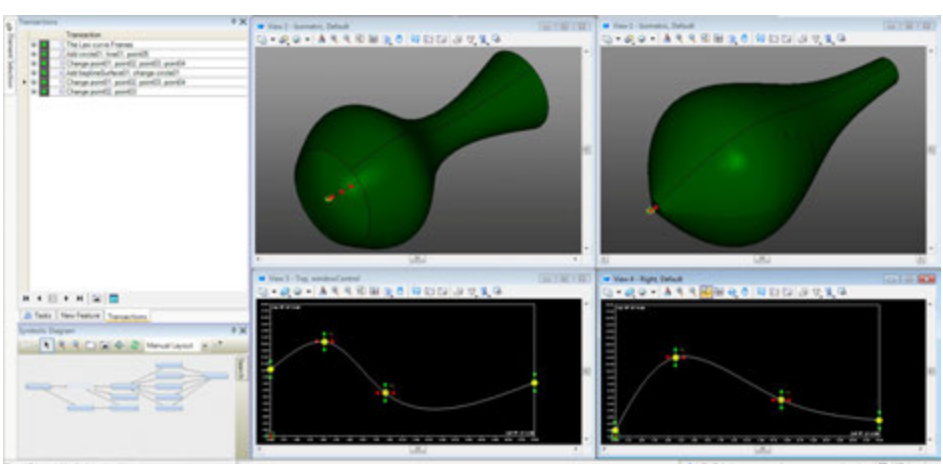


Figura 10. Ejercicio 4. Law curves

con el cual trabajamos, permite tomar decisiones basadas en datos y no en intuición. En este caso un malla de polígonos es creada por medio de una función de seno coseno. Cada polígono es coloreado dependiendo de la distancia a la que se encuentra de un punto en el espacio. Los colores usados corresponden a la lista de Bentley Microstation.

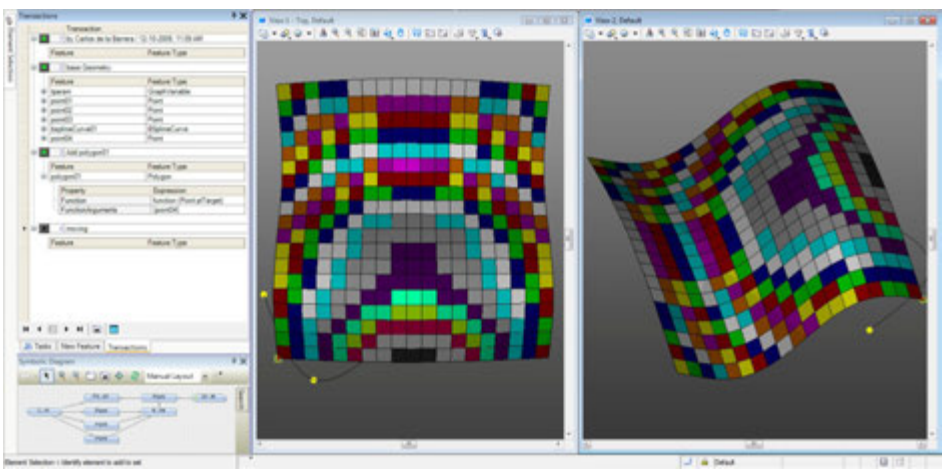


Figura 11. Ejercicio 5. Superficie reactiva

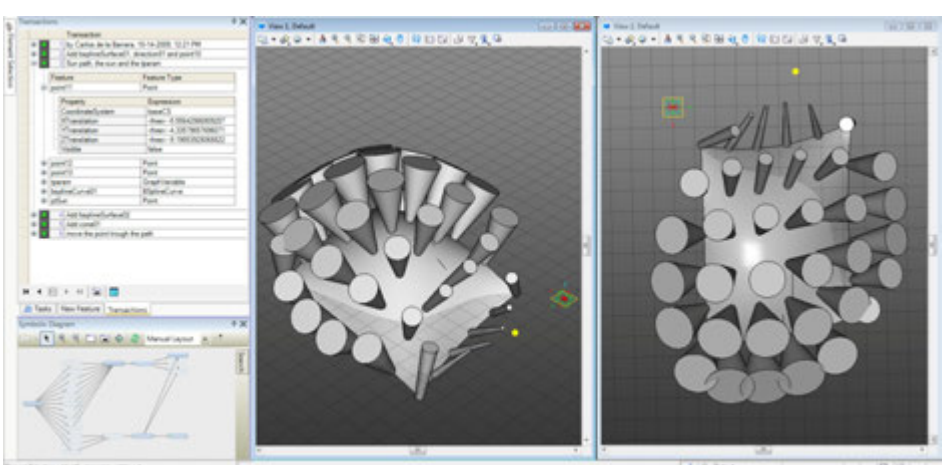


Figura 12. Ejercicio 6. Superficie sol, reactiva

ECUACIÓN PARA LA MALLA DE POLIGONOS:

```
for (int j = 0; j < 20; ++j)
{
  pt[i][j] = new Point();
  pt[i][j].ByCartesianCoordinates(baseCS, i*2, j*2,(Sin(j*20)+Sin(i*10))* 6);
  myColor[i][j] = Distance(ptTarget, pt[i][j]);
}
```

EJERCICIO 6, SUPERFICIE SOL - REACTIVA

Similar al ejercicio anterior. Esta vez una serie de cilindros se encuentran ubicados en la normal de los puntos en la superficie. El radio de los cilindros corresponde al producto vector entre un “sol” (punto amarillo) y su ángulo de incidencia y la normal de la superficie en cada punto.

PRODUCTO VECTOR:

```
for (int j = 0; j < dir1[i].Count; ++j)
{
  cone[i][j] = new Cone();
  cone[i][j].ByPoints(dir1[i][j], ptTop[i][j], 0.2, Angle(dir1[i][j],                ptTop[i][j], sun) * 0.01);
}
```

EJERCICIO 7, CUBIERTA Y COMPONENTE GENERATIVO.

Este ejercicio es parte de los ejemplos de GC del libro *GC_V8i_Essentials*. Es muy interesante en el sentido de que explica la construcción de una cubierta comenzando con puntos y haciendo más compleja la geometría a medida que se va incorporando más información.

EJERCICIO 8, SUPERFICIE Y PLANO DE FABRICACIÓN.

Durante este ejercicio los alumnos debieron construir una superficie que reaccionara a un punto y luego subdividirla por medio de polígonos para crear un plano de fabricación con todos los polígonos. Los polígonos que no fuesen planos, serían coloreados con algún color.

ESTRUCTURA CONDICIONAL PARA COLOREAR LOS POLIGONOS.

```
feature User.Objects.polygon01 Bentley.GC.Features.Polygon
{
  Color = polygon01.OutOfPlane > 0.1 ? 3 : 80;
  Points = point03;
}
```

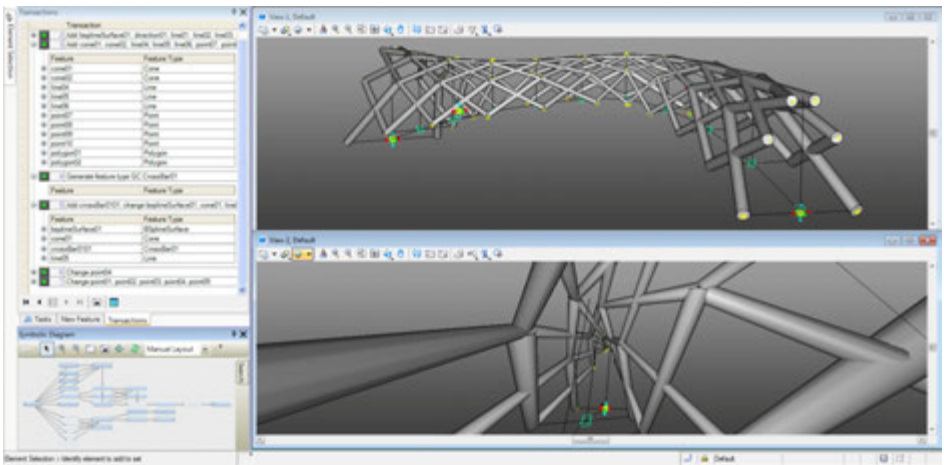


Figura 13. Ejercicio 7. Cubierta y componente generativo

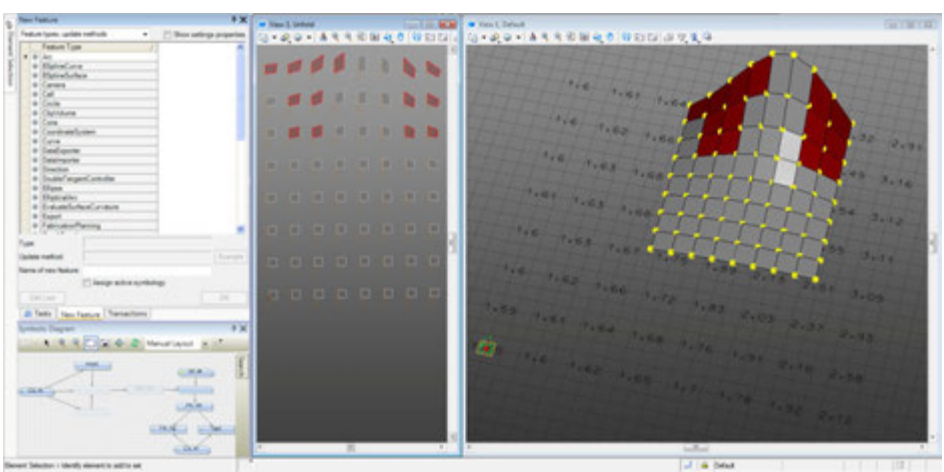


Figura 14. Ejercicio 8. Superficie y plano de fabricación

EJERCICIO 9, SUPERFICIES PARAMETRICAS.

Las superficies paramétricas son muy difíciles de modelar a mano, pero fáciles de definir mediante ecuaciones.

ECUACIONES DE LAS SUPERFICIES.

Boy Surface:

x=(Cos[u]*((1/3)*Sqrt[2]*Cos[u]*Cos[2*v]+(2/3)*Sin[u]*Cos[v]))/(1- Sqrt[2]*Sin[u]*Cos[u]*Sin[3*v]);
y=(Cos[u]*((1/3)*Sqrt[2]*Cos[u]*Sin[2*v]-(2/3)*Sin[u]*Sin[v]))/(1- Sqrt[2]*Sin[u]*Cos[u]*Sin[3*v]);
z=(Cos[u]*Cos[u])/(1-Sqrt[2]*Sin[u]*Cos[u]*Sin[3*v])-1;

Paraboloide gusano:

x=Cos(u) * (4 + Cos(v));
y=Sin(u) * (4 + Cos(v));
z=4 * Sin(2*u) + Sin(v) * (1.2 - Sin(v));

Superficie compleja:

x=(20*Cos(v)) * Cos(u + 20);
y=(20*Cos(v)) * Sin(u + 20);
z=Cos(u) * 10;

Jarro
x = ((u*0.5) - Sin(u)) * Cos(v);
y = (200 - Cos(u)) * Sin(v);
z = u;
Cilindro
x = Cos(u) * 50;
y = Sin(u) * 50;
z = v * 0.1;

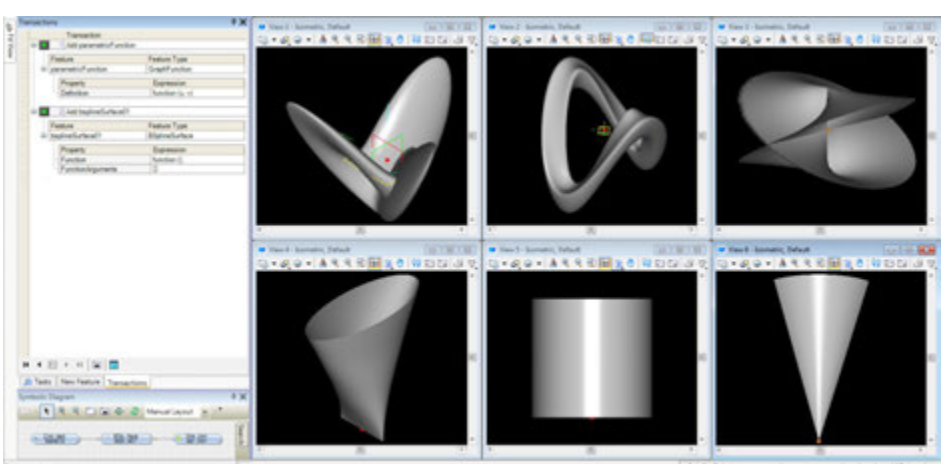


Figura 15. Ejercicio 9. Superficies paramétricas

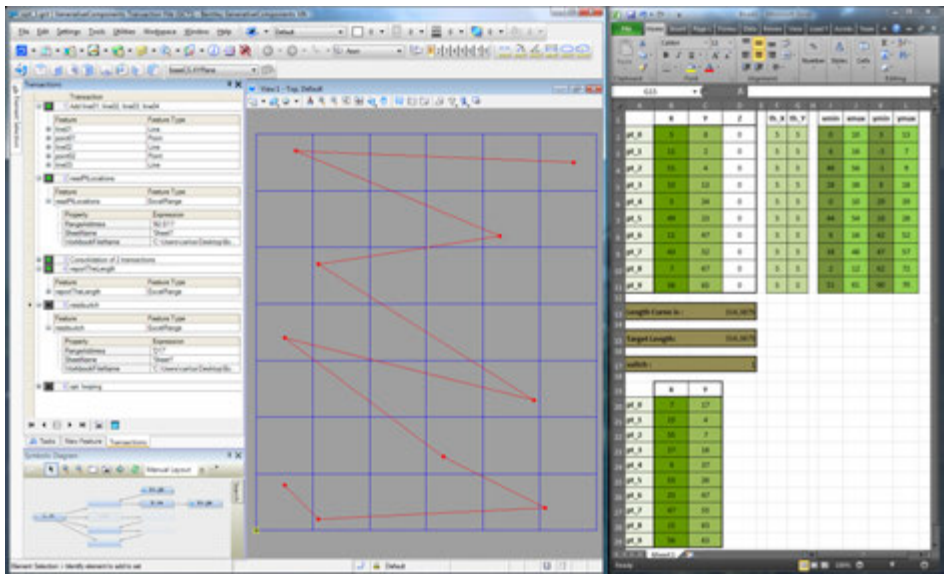


Figura 16a. Ejercicio 10. Optimización con Excel. Imagen de la curva antes de la optimización. Su longitud es de 314, 3875 Uds. En la tabla de Excel aparecen las coordenadas iniciales y las restricciones.

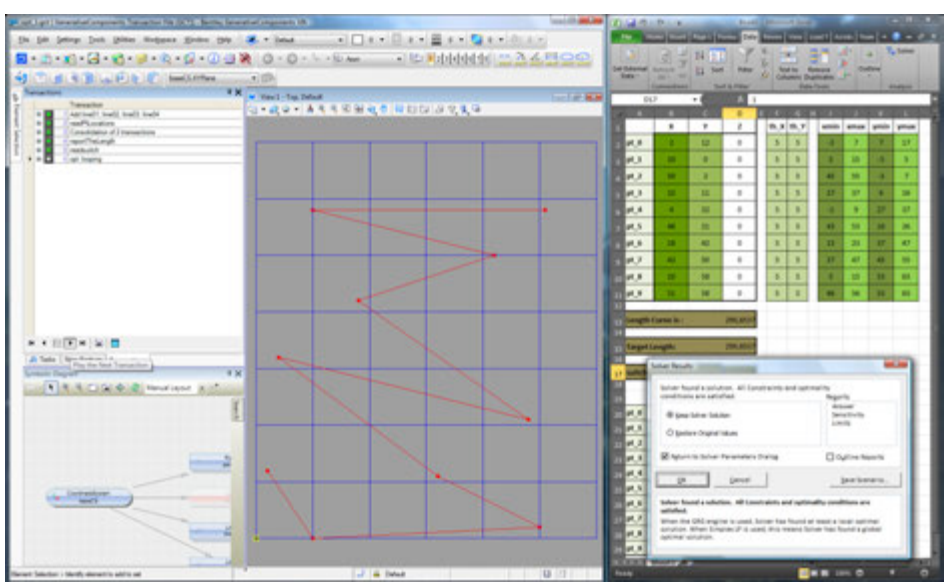


Figura 16b. Ejercicio 10. Optimización con Excel. Después de la optimización la longitud de la curva es de 290,8537 Uds.

```
Hoja de cono
x = Cos(u) * (v * 0.2);
y = Sin(u) * (v * 0.2);
z = v*0.5;
```

FUNCION PARA GENERAR LAS SUPERFICIES

```
function ()
{
  BSplineSurface srf = new BSplineSurface();
  Point mypt = {};
  double stepSize = 1;
  double coord;

  for (int i = 0; i < 100; ++i)
  {
    mypt[i] = {};
    for (int j = 0; j < 100; ++j)
    {
      coord = parametricFunction(i * stepSize, j* stepSize);
      mypt[i][j] = new Point();
      mypt[i][j].ByCartesianCoordinates(baseCS, coord[0], coord[1], coord[2]);
    }
  }
  return srf.ByPoints(mypt);
}
```

EJERCICIO 10, OPTIMIZACIÓN CON EXCEL

La interoperabilidad juega un papel importante en el diseño de formas y geometría. GC logra comunicarse con Excel de manera automática, permitiendo el envío y recibo de datos. Para el ejercicio final se construyó una simple curva por 10 puntos definidos en Excel. Dentro de GC se construye una curva que pasa por cada uno de los puntos y su longitud es retornada a Excel. Usando el plugin SOLVE de Excel es posible optimizar la longitud de la curva moviendo los puntos dentro de un umbral establecido.

Conclusiones

Las conclusiones se podrían dividir en 3 partes: la primera es el efecto de estas herramientas en el diseño. Si bien los modelos paramétricos llevan bastante tiempo en el campo de la ingeniería, en la arquitectura hace poco que están dando sus primeros pasos y en este sentido los participantes tomaron conciencia de que la arquitectura debía aprender de otras disciplinas, como por ejemplo de la fabricación en la industria aeroespacial. También se dieron cuenta de que

lo que llamamos geometría es en realidad muchas geometrías y no una relación estática como es la que se aprende hoy en día. La extrapolación de este concepto en ese sentido es bastante actual porque refleja el mundo cambiante en el que nos encontramos. En el diseño paramétrico, los aspectos del modelo dependen de relaciones entre sus diferentes partes. Creando y modificando estas relaciones el modelo se modifica. Usualmente un modelo paramétrico es definido por reglas y restricciones, las cuales definen diferentes aspectos del edificio y sus relaciones. En cambio, en un modelo dibujado, la geometría es explícita, se expresa con claridad, porque lo vemos directamente. Sus reglas son implícitas, se entiende que están incluidas aunque no se diga cómo ni tampoco aparezcan (siempre hay reglas y restricciones en un modelo arquitectónico). La diferencia es que la herramienta de modelado no mantienen los seguimientos de estas reglas, por lo que es el usuario quién debe realizarlo. En contraste, en los modelos paramétricos, las reglas son explícitas, se expresan de manera clara y exacta. Sin embargo su geometría es implícita, en el sentido que se entiende como parte del modelo aunque no aparezca gráficamente. Por ejemplo los vectores o planos en un modelo paramétrico.

Los participantes pudieron entender las posibilidades del diseño paramétrico y tener una leve noción de un amplio campo como es el uso de esta tecnología en el diseño. Durante el curso de 8 horas los participantes se quedaron con más preguntas que respuestas, pero con la idea de continuar desarrollando sus habilidades.

Quisiera agradecer la ayuda de Ramón González durante la realización del curso.

Taller 2

Grasshopper

Roberto Molinos

Grasshopper es una de las herramientas cuyo uso está creciendo más en los últimos años. Es probablemente la más orientada al diseño en arquitectura de todas y las más fácil de aprender para los no iniciados en el diseño paramétrico sin descuidar unas prestaciones realmente potentes que abarcan desde la programación simbólica más sencilla hasta la definición de componentes personales en un entorno de desarrollo completo.

Grasshopper es desarrollado por un equipo liderado por arquitectos con inquietudes computacionales, y este hecho lo hace diferente a las demás herramientas objeto de los talleres, que fueron ideadas por ingenieros. La orientación al diseño de Grasshopper es total y se ve claramente en el cuidado de su interfaz gráfica y en la multitud de opciones que ofrece al diseñador para transmitir sus ideas.

La curva de aprendizaje de Grasshopper es muy suave ya que los conceptos que maneja son cercanos a cualquier persona con un mínimo interés en geometría y matemáticas. Sin embargo, no deja de ser una herramienta de diseño generativo y paramétrico e implica que los proyectos han de ser estructurados en procedimientos, variables y condiciones, por lo que estructurar un taller de 8 horas que permita al que lo siga introducirse con garantías en su manejo no deja de ser un reto.

El taller de Grasshopper, siguiendo las directrices comunes a los demás talleres de Algomad 2010, se orientó al aprendizaje por ejemplos y se dividió en un primer día dedicado a completar un ejercicio guiado y un segundo día de trabajo más autónomo en el que los asistentes, de manera individual o por equipos, desarrollaron problemáticas concretas.

El taller de Grasshopper de Algomad 2010 fue coordinado e impartido por Roberto Molinos, profesor de IE School of Architecture y contó con la colaboración como tutores de Enrique Soriano y Pep Tornabell.

A continuación reseñamos la documentación que sirvió como base al ejercicio conjunto del primer día así como los resultados de las experiencias particulares de algunos de los asistentes durante el segundo día.

DIA 1. EJERCICIO GUIADO

1. INTRODUCCIÓN

Grasshopper (GH de ahora en adelante) es un entorno de programación gráfica que corre sobre Rhinoceros. La programación gráfica consiste en la definición de algoritmos, rutinas y relaciones de modo gráfico y esquemático. El gesto gráfico por excelencia en GH es la conexión de componentes y parámetros por medio de cables.

GH se descarga gratuitamente de www.grasshopper3d.com y se instala sobre Rhinoceros. Para ejecutarlo basta introducir ***grasshopper*** en la **barra de comandos** de Rhino.

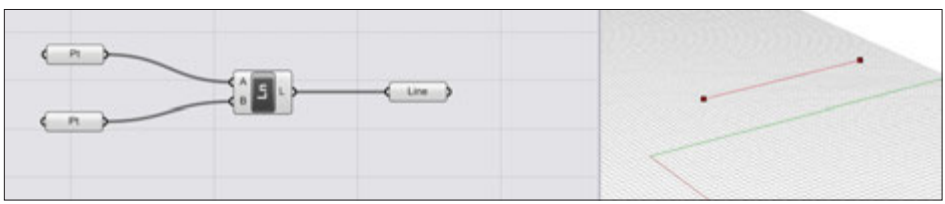


Figura 1.1

El conjunto de **cables**, **componentes** y **parámetros** configura lo que se conoce como una **definición**. La definición toma unos datos de partida y los procesa en tiempo real y ofrece unos resultados numéricos y gráficos. El procesado, el recorrido por el circuito de la definición, se hace siempre de izquierda a derecha, y no es posible formar bucles entre definiciones.

Los datos de partida en GH, y cualquier conjunto de datos, se conocen en GH como **parámetros**. Los parámetros son muy importantes porque son la materia que GH transforma, interpreta y utiliza para construir geometrías más complejas. Las operaciones que transforman unos datos en otros, que los procesan, ocurren dentro de los **componentes**. Los componentes y parámetros tienen un comportamiento un tanto especial. Veámoslo:

Parámetros:

Los parámetros pueden ser de muchos tipos; pueden ser geométricos - punto, plano, vector, línea, curva, superficie, brep, mesh…- pero también pueden ser de tipos un poco más matemáticos – número real, número entero, booleano (si o no)…- Los parámetros en realidad responden a espacios o casillas de la memoria del ordenador donde un determinado dato es guardado. Los parámetros pueden introducirse de varias formas en GH:

Figura 1.2

Set one XXX permite definir un parámetro proveniente de, o como mínimo en,

la interfaz de Rhino. Así podemos heredar objetos de Rhino dentro de GH. Por supuesto esta forma no aplica a la definición de parámetros como números o booleanos.

Set multiple XXX permite hacer lo mismo que lo anterior pero definiendo varios datos para un mismo parámetro. Esta es una de las grandes características de GH, no distingue entre UN dato (un punto, una curva, un plano) y UN MILLÓN de datos (un millón de puntos, un millón de curvas..)

Manage XXX Collection permite definir, numéricamente y dentro de la interfaz de GH, los datos que deseemos. Por ejemplo, un punto podrá ser definido por sus coordenadas {x,y,z}, un círculo por su plano, centro y radio, y así con diferentes objetos. Esto no se aplica a superficies o mallas.

Los parámetros pueden convertirse en otros si el sentido común lo permite. Por ejemplo, un número real puede ser heredado como entero y perderá su parte decimal, un número entero puede heredarse como real pero no tendrá parte decimal. Una línea recta podrá heredarse como curva, y lo mismo le pasa a un círculo. Sin embargo, una curva sólo podrá heredarse como círculo si efectivamente lo es, y una curva sólo podrá heredarse como recta si también lo es.

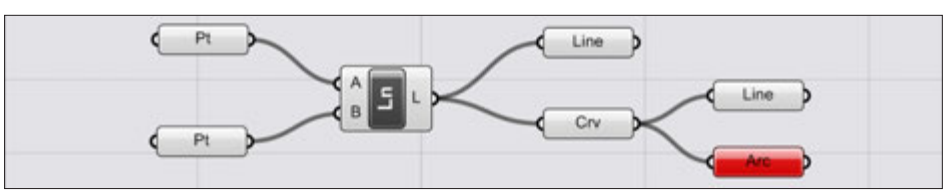


Figura 1.3

En el ejemplo anterior, dos parámetros punto se usan para construir por medio de un componente Line una línea recta. Esa línea puede ser heredada como curva, que a su vez puede volver a ser interpretada como línea, pero no como arco, pues no lo es.

Componentes:

Los componentes son las piezas de GH que se encargan de tomar datos y convertirlos en otros datos, en el caso anterior el componente **Line** tomaba dos parámetros punto y generaba una línea.

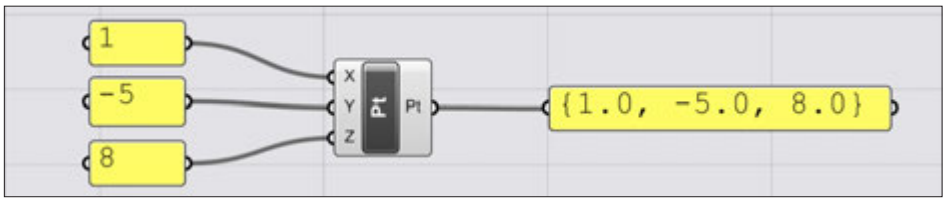


Figura 1.4

La figura anterior muestra un componente **Point XYZ** que toma 3 números y construye un punto usando esos números como coordenadas **{x,y,z}**.

La salida de un componente será normalmente un parámetro. Los componentes y también los parámetros tienen diferentes estados en función de si han conseguido procesar adecuadamente los parámetros que les hemos suministrado.

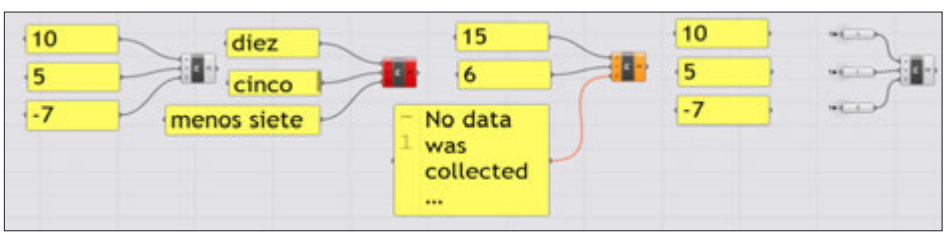


Figura 1.5

Gris si todo está en orden. Rojo si los datos suministrados no son del tipo adecuado (GH espera números no palabras). Naranja si todo está bien pero faltan datos. Los datos también pueden enviarse inalámbricamente.

Cables:

Si activamos la opción Draw Fancy Wires en el menú View de GH podremos ver que los cables se dibujan de 3 formas distintas en función de la cantidad y forma de la información que transmitan:

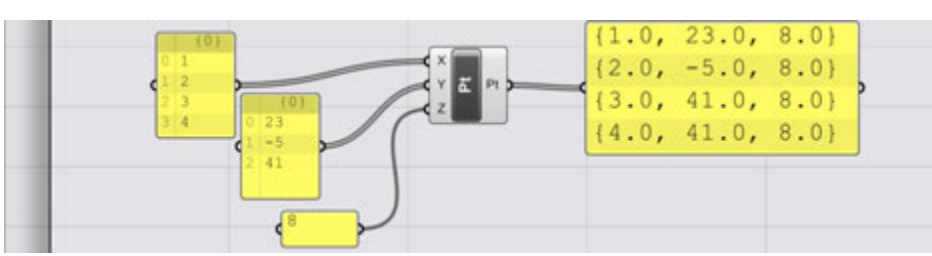


Figura 1.6

El cable sólido y fino sólo está transportando un valor para z (8) mientras que los cables con alma blanca, que son más gruesos, llevan más de un valor, en con-

creto una lista de valores. Si el cable se dibuja a trazos es que está mandando un árbol de listas.

Listas:

Hablar de cables da pie a hablar de listas. Como hemos dicho GH no distingue entre un valor y n valores para sus parámetros. Esto es así y es útil porque GH tiene una manera muy simple y clara de gestionar las listas.

Básicamente GH toma los primeros elementos de cada lista y los combina para formar lo que corresponda, luego toma el segundo elemento y así sucesivamente. A partir de aquí existen tres comportamientos que hay que conocer:

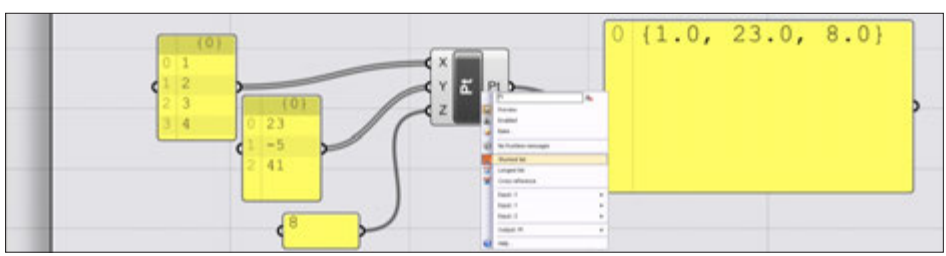


Figura 1.7

Shortest List hará combinaciones hasta que se quede sin valores en una de las listas involucradas, por tanto la longitud de la lista de salida será igual a la menor de las listas de entrada. En este caso tenemos 4 valores para X, 3 para Y pero sólo 1 para Z, por lo que GH sólo genera un punto.

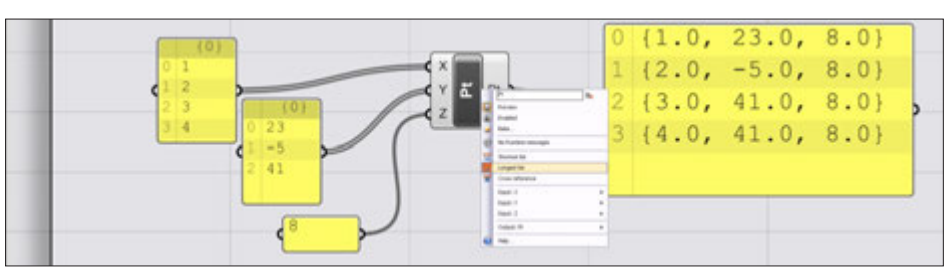


Figura 1.8

Longest List hará combinaciones hasta que agote todos los valores de todas las listas, si en alguna se queda sin valores nuevos, tomará el último valor y lo repetirá las veces que haga falta. La longitud de la lista será igual a la mayor longitud de las listas de entrada. En el ejemplo todos los puntos tienen coordenada Z = 8 porque GH ha tenido que repetirla varias veces al ser el único valor proporcionado.

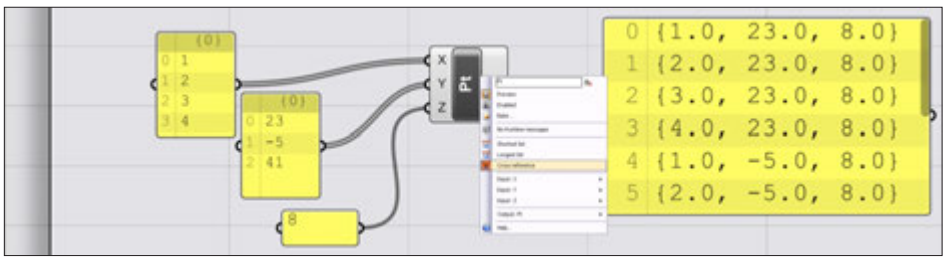


Figura 1.9

Cross Reference hará todas las combinaciones posibles entre datos, y la lista final tendrá la misma longitud que el producto de las longitudes de las listas de entrada. En el ejemplo habrá 4*3*1 = 12 puntos.

2. EJERCICIO PRÁCTICO:

Interfaz:

La interfaz de GH está en constante mejora y es una de sus características más apreciadas por los arquitectos que lo usan, sin embargo, conforme aparecen más componentes, se hace más difícil encontrarlos dentro de la estructura de menús que funciona por pestañas. Afortunadamente, si hacemos doble click sobre el tapiz (el espacio en blanco) aparece un cuadro de búsqueda en el que introduciremos el componente o parámetro buscado.

Controladores:

Además de los parámetros que hemos visto, GH dispone de otra serie de “parámetros” que permiten la definición interactiva de datos por el usuario. Son el **panel**, el **number slider** y el **boolean toggle**. Pueden encontrarse haciendo doble click y escribiendo su nombre. Todos los controladores se configuran al hacer doble click sobre ellos.

SI hacemos doble click sobre un **Number Slider** podremos definir el tipo de slider – números enteros, reales, pares o impares – la precisión, el límite inferior y superior, la longitud y por supuesto el valor.

Si hacemos doble click sobre un **panel** podemos editar su contenido. Hay que tener en cuenta que si la casilla **multiline data** está activa, GH interpretará todo lo que se escriba como un solo valor, mientras que si se desactiva, cada línea será entendida como un valor y entonces obtendremos una lista.

Los **panels** son muy útiles tanto para definir listas y números o palabras sencillas como para ver la salida de componentes y entender un poco mejor qué está pasando en la definición.

Por último, el **boolean toggle** es muy útil para apagar y encender determinados comportamientos en la definición. Su funcionamiento es tan sencillo como el de un bit, así que no hay mucho que explicar, basta con hacer doble click.

Figura 1.10

Visibilidad:

Los modelos y definiciones de GH pueden volverse tremendamente desastrosos y complicados si sus creadores no son ordenados y metódicos. Además, como estamos modelado de forma generativa, cada paso intermedio en nuestra definición genera un resultado que por defecto se muestra en pantalla.

Figura 1.11

GH permite controlar la visibilidad de los objetos de dos formas. La más directa consiste en hacer click con el botón derecho y apagar o encender la opción de Preview. Esto nos permite apagar los elementos uno a uno, no se muestran pero siguen computándose.

La segunda manera permite apagar conjuntos de objetos para quedarnos, por ejemplo, únicamente con el resultado y no ver los pasos intermedios. Funciona seleccionando todos los componentes y parámetros que queramos, bien uno a uno, bien con una ventana de selección, y pulsando los botones de visualización propios del espacio de trabajo, reconocibles porque son una cabeza con y sin una venda en los ojos.

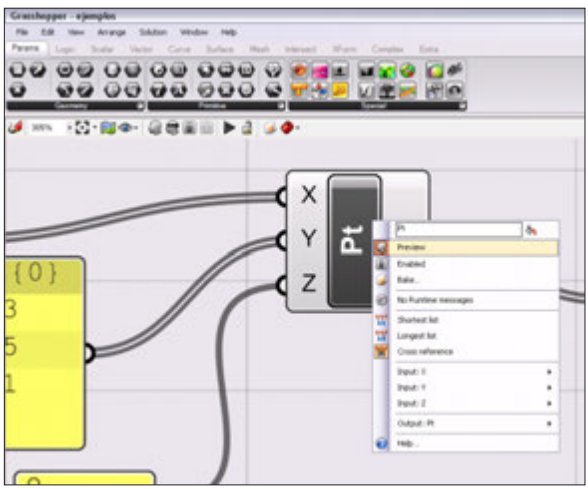


Figura 1.12

Datos persistentes:

Los objetos de GH, los datos y operaciones de nuestras definiciones son, normalmente, volátiles. Esto quiere decir que no existen como tales en el modelo de Rhino sino que son calculados en tiempo real para cada estado diferente de parámetros que definamos (modelado paramétrico quiere decir esto…) y visualizados.

Cada vez que se hace un cambio en los parámetros, en un slider, o se mueve un punto de Rhino referenciado dentro de GH, el programa recalcula toda la definición y ofrece un resultado nuevo. Este proceso puede ser más o menos largo en función de lo complicado que sea el modelo y de lo desastrosa o acertada que haya sido la estrategia al construirlo.

Es posible *freir* la geometría derivada de determinados componentes y convertirla en objetos nativos y persistentes de Rhino. Este proceso convierte ese objeto volátil en algo estable y permanente que puede ser exportado, renderizado, guardado, impreso, etc.

La utilidad que fríe las geometrías de GH en la parrilla de Rhino tiene forma de huevo frito. Podemos acceder a ella de igual forma que con la **Preview**, directamente componente a componente haciendo RC o en la barra de herramientas del tapiz.

Navegación:

Navegar por el tapiz de GH es muy sencillo: Con la rueda se hace zoom en la definición y manteniendo pulsado el botón derecho se desplaza el tapiz. Por defecto aparece un artilugio, un compás, que nos indicará en que direcciones

dentro de la definición vamos a encontrar objetos.

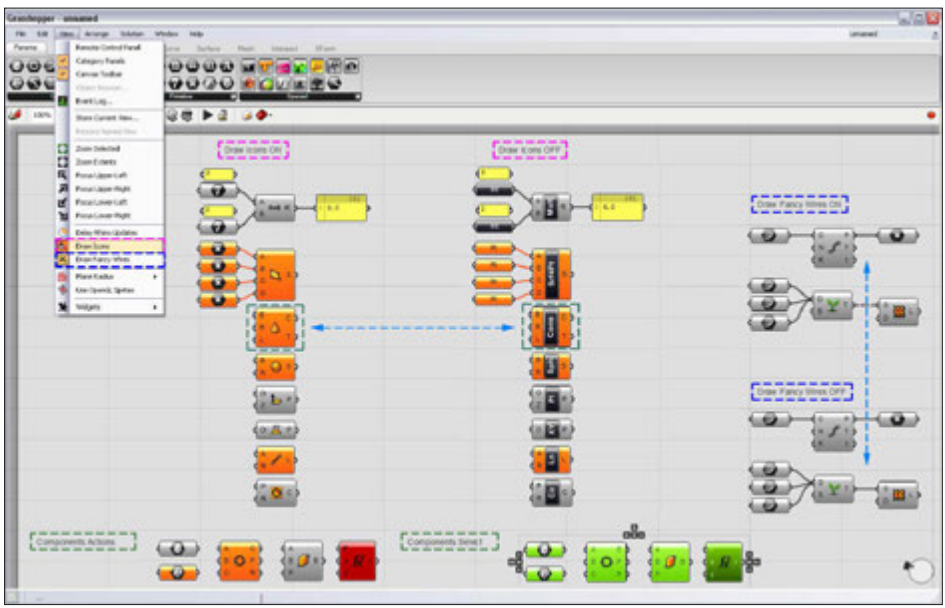


Figura 1.13

Algunas opciones más:
En el menú view de GH (no el de Rhino) se encuentran una serie de opciones que permiten ajustar algunos parámetros de la interfaz:

view>draw icons: en lugar de representar cada componente o parámetro con un texto lo hará con un icono, es ideal si se quiere compartir las definiciones en un foro pero un follón cuando se está aprendiendo porque hay que saberse los iconos. Recomendamos dejarlo por defecto.

view>plane radius: permite definir cómo de grandes se verán los planos de GH en las ventanas de Rhino.

view> widgets > profiler: el profiler permite hacerse una idea de cuánto le cuesta a GH calcular la definición, dándo el tiempo en milisegundos para cada componente.

3. TORRE ALGOMAD

0. Empezamos

Desde la web de algomad -www.algomad.org- se puede descargar el conjunto de archivos utilizados en este documento. Abre el archivo de Rhino (extensión .3dm) ***Algomad_GH_Ejercicio01.3dm*** Deberás ver tres curvas periódicas de forma libre y tres puntos. En la barra de comandos de Rhino escribe grasshopper y pulsa INTRO. Debería aparecer el editor de definiciones de GH y un tapiz en blanco. Ya estás listo para empezar.

1. Referencia de objetos de Rhino

Haz doble click en el tapiz y carga 3 parámetros curva (busca **curve** y escoge el icono con forma de hexágono negro) y 3 parámetros punto (busca **point** y escoge…el icono con forma de hexágono negro)

Renómbralos a **Curva0**, **Curva1**, **Curva2**, **Punto0**, **Punto1**, y **Punto2** haciendo RC y escribiendo el nuevo nombre.

Sobre cada uno de los parámetros curva, haz botón derecho y dale a ***set one curve***, selecciona la curva correspondiente en la ventana de Rhino. **¡Has referenciado tu primera geometría de Rhino en GH!**

Ahora haz lo mismo con los puntos, referenciándolos de manera coherente, Punto0 dentro de Curva0, Punto1 dentro de Curva1 y así. Hazlo haciendo RC sobre el parámetro punto y dale a ***set one point***.

Ya tienes tus tres curvas y tus tres puntos dentro de GH.

Ahora carga tres Planos XY, doble click y busca **XY Plane**. Enlaza el origen de cada plano XY a cada uno de tus tres puntos.

Por último, carga dos componentes **Merge 03** y enlaza al primero de ellos las tres curvas, y al segundo los tres planos XY.

Debería quedar así:

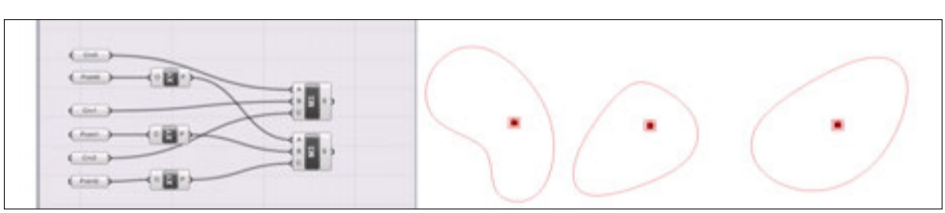


Figura 1.14

Ya tienes la geometría de Rhino dentro de GH, ahora vamos a crear datos exclusivamente dentro de GH.

Lo primero es crear dos sliders, uno para definir la altura de la torre y otra

para definir el número de cortes de control que vamos a tener sobre ella.

doble click en tapiz> **Number Slider** > doble click > Precision = 1, de 0 a 100, nombre **Altura**.
doble click en tapiz> **Number Slider** > doble click > Integers, de 10 a 20, nombre **Cortes**.

Ahora vamos a usar un componente muy útil en GH, el Rango.
El rango toma un dominio (de 0 a 14 o de -8 a 2048.5 por ejemplo) y lo divide en cachitos de igual tamaño, dándonos los valores intermedios.

Crea dos paneles -**panel**- y dales valor 0
Crea dos dominios -**domain**- y enlaza cada panel a A y el slider altura a B
Crea dos rangos -**range**- y en el parámetro **D** dales los dominios anteriores, y en el **N** dale al primero valor **2** (puedes hacerlo con un panel) y al segundo enlázale el slider **cortes**.
Carga dos **Point XYZ** y enlázales en el parámetro **Z** la salida de los rangos anteriores.
Crea planos XY y usa los puntos anteriores como origen. Si te fijas, acabamos de generar dos columnas de planos superpuestos, una columna tiene sólo tres planos, mientras que la otra tiene tantos planos como hayamos fijado en el parámetro ***cortes***. Como puedes ver, el componente **XY Plane** no distingue entre uno o dos mil planos.

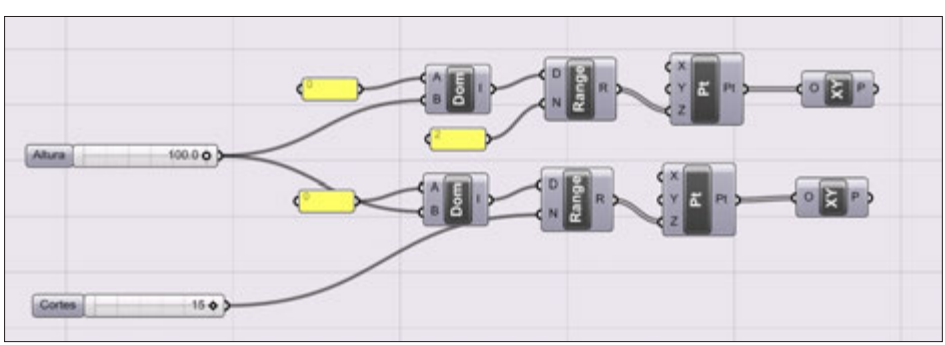


Figura 1.15

Ahora deberías ser capaz de repetir lo anterior y conseguir que tu definición se parezca a lo siguiente. Los sliders van de **1.00** a **2.00**.

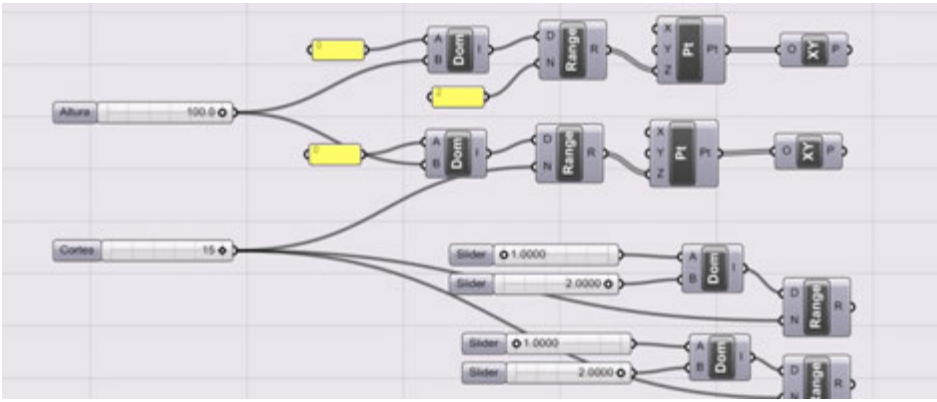


Figura 1.16

Hasta ahora no hemos hecho nada del otro mundo en GH, pero hemos generado un montón de datos que nos van a hacer falta para poder construir nuestra torre.

2. Superficie Básica

Tomaremos la **Curva0** y usando como referencia el plano XY centrado en el Punto0 la llevaremos al primero de los planos de nuestra columna de tres planos, la Curva1 la llevaremos al segundo plano (que estará a una altura mitad que la altura total) y la Curva2 la llevaremos a un plano a una altura igual a la que hayamos definido. Una vez las tres curvas estén en vertical, construiremos un Loft que pase por las mismas, y así tendremos nuestra superficie inicial.

Los componentes interesantes aquí son dos:

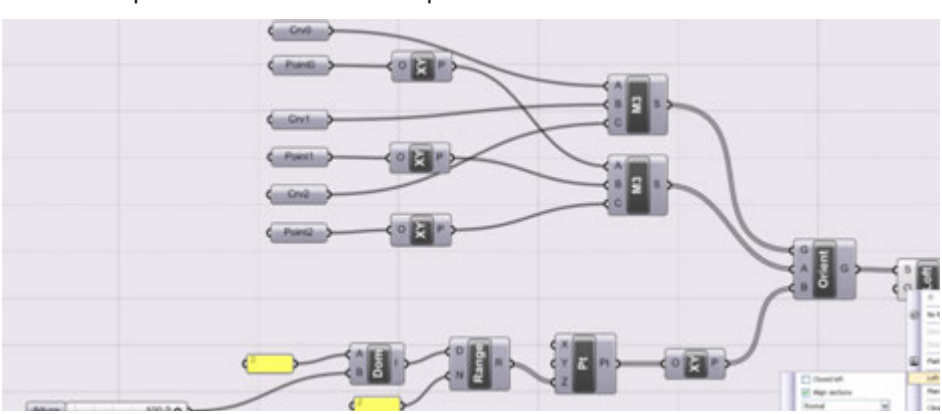


Figura 1.17

Y así es como construimos nuestra superficie de referencia:

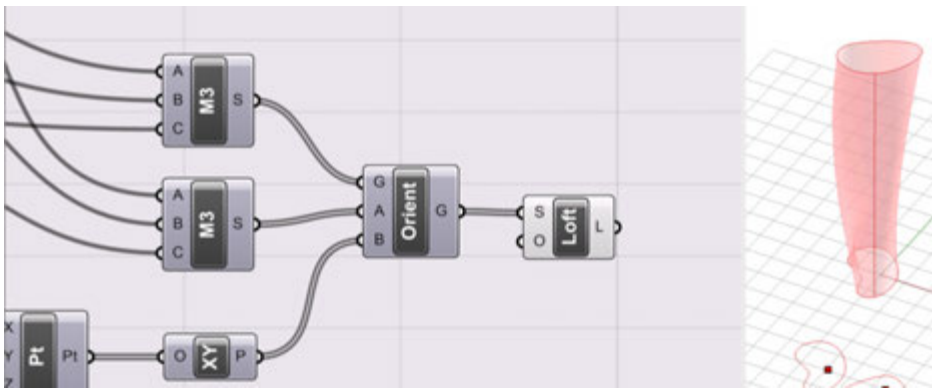


Figura 1.18

5. Twist and Shout:

Con nuestra otra columna de planos (la que depende del número de cortes) vamos a darle cortes a la superficie que hemos creado en el paso anterior. Será como obtener rebanadas de la superficie.

Las curvas resultantes, que deberían ser tantas como planos de corte tengamos definidos, serán cerradas, y las deformaremos y rotaremos a nuestro gusto para finalmente obtener una forma un poco más interesante.

Lo primero que hemos de hacer es cortar la superficie, para eso haremos doble click sobre el tapiz y buscaremos **BRep | Plane** que nos cargará un componente que hará la intersección entre un **BRep** (nuestro Loft) y cada uno de los planos.

A la salida de las secciones deberemos intercalar un componente **Flatten**, que simplifica la estructura de datos de GH.

Después insertaremos un componente **Scale NU** que nos permitirá definir factores de escala diferentes para **X Y** y **Z**, pero éste último no nos interesa porque las curvas son planas. En definitiva, nuestra definición deberá ser algo parecido a esto:

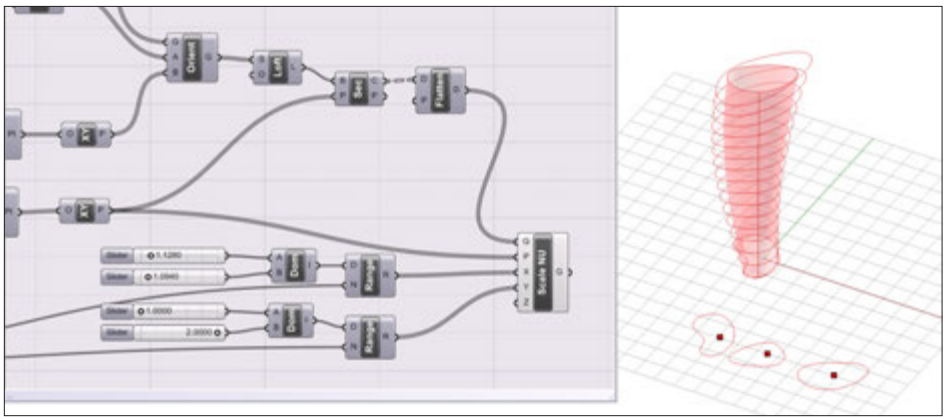


Figura 1.19

Si variamos los sliders que se ven en la figura, estaremos modificando en factor de escala para las dos direcciones en la parte baja y alta de la torre, con lo que nuestro control sobre la forma ahora es más directo.

Por otro lado, si modifico las curvas iniciales, que permanecen en el plano Z=0, sigo cambiando la forma de la torre, pero estoy separando topología de métrica. La forma y la medida.

El siguiente paso consiste en girar cada corte deformado un poquito. Girar es igual a ángulo, y como queremos girar cada sección un poquito necesitamos un nuevo rango. Esta vez irá de **0** a un factor de **Pi**, porque los programas serios tratan los ángulos en radianes, y así podremos variar cuánto se retuerce nuestra torre.

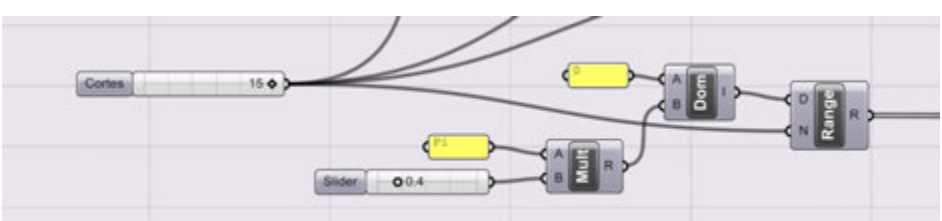


Figura 1.20

Una vez creado el rango de ángulos, que como imaginas tendrá tantos números como indique el valor de cortes, debemos rotar cada sección deformada según el plano de referencia. Podemos hacerlo de muchas maneras pero la forma más simple es volver a emplear el concepto de **Orient**.

En lugar de rotar la geometría directamente, rotamos el plano de referen-

cia de cada piso –rotate plane- y reorientamos la geometría tal y como hemos hecho antes. Como valor de la rotación usamos la lista de ángulos que hemos creado en la página anterior.

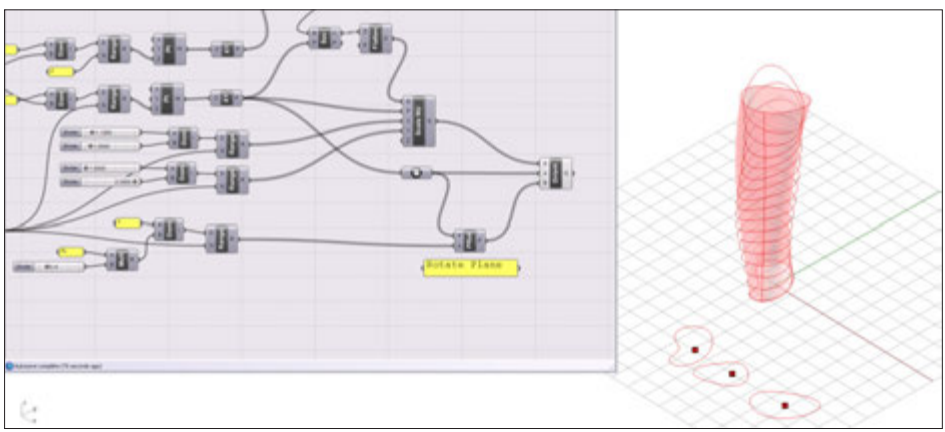


Figura 1.21

Finalmente, construiremos un Loft que pase por el conjunto de curvas deformadas y rotadas y así tendremos la piel de nuestra torre.

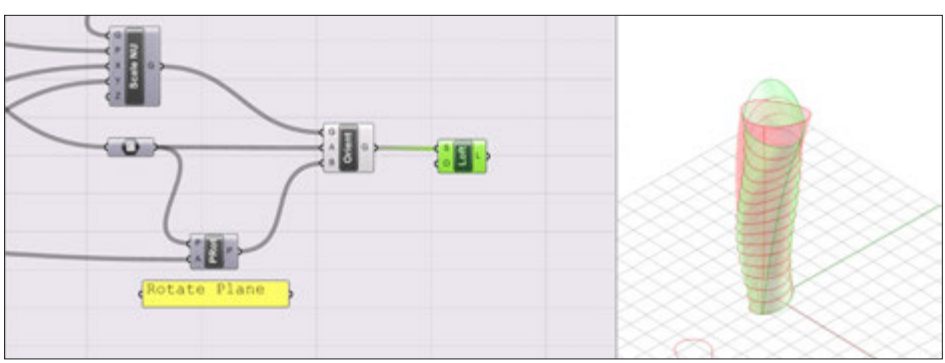


Figura 1.22

Un apunte. El parámetro que aparece con el icono es un parámetro de tipo Plano y que está funcionando como repetidor. A veces es una buena idea, si vamos a conectar muchos cables a un parámetro y desde muchas zonas distintas de la definición, utilizar repetidores de este tipo que hagan más fáciles los cambios y que mantengan la legibilidad del tapiz.

6. Cajas Mágicas:

A partir de aquí la cosa se vuelve un poco más complicada en cuanto a las operaciones que hay por detrás, pero todo resulta mucho más claro porque lo que hacemos tiene una representación directa.

Vamos a dividir la superficie Loft en regiones que serán, conceptualmente, rectangulares, y sobre esas regiones GH va a construir unas cajas, que serán una especie de extrusión de esas regiones.

Colocamos un componente **divide domain**, fijando la Superficie *Loft* como **I** y dando valores enteros a **U** y

Colocamos un componente **Surface Box**, tomando la superficie *Loft* como **S**, el dominio anterior como **D** y fijando la altura **H** en 10, por ejemplo.

A continuación, vamos hacer dos cosas con cada una de las cajas deformadas que tengamos.

Primero, vamos a calcular el centro de masas de la caja -**Brep Volume**-, por ser un punto que seguramente estará hacia lo que uno podría entender como el centro, y buscaremos su proyección -**Surface CP**- sobre la superficie *Loft* . La proyección será un nuevo punto N. Acto seguido, evaluaremos la superficie en cada uno de esos puntos P y obtendremos la normal a la superficie en esos puntos -**Evaluate Surface**- Calculamos la normal para después poder modificar cada cara de la torre en función del ángulo que formen con el sol.

Segundo, vamos a descomponer cada caja en sus partes -**Brep Components**-, que serán de tres tipos; caras, aristas y vértices. Nos interesa quedarnos sólo con los vértices. Como puedes imaginar, los vértices serán puntos que vendrán ordenados en listas, listas de 8 valores cada una, las 8 esquinas de una caja. De esos 8 puntos nos quedaremos sólo con los tres primeros. Esto lo haremos partiendo la lista -**Split List**- de puntos en el tercer elemento y quedándonos únicamente con lo que haya antes del corte (**A**)

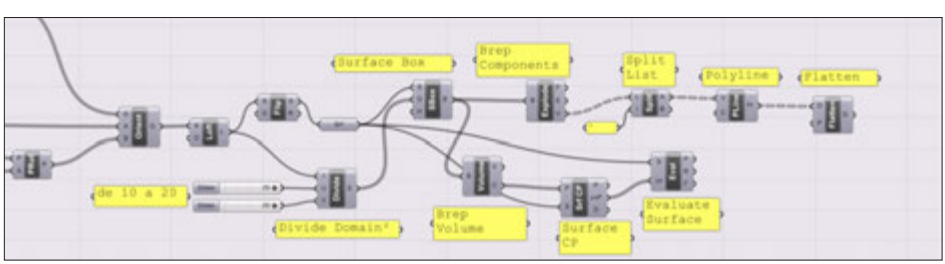


Figura 1.23

Finalmente, construiremos una polilínea que pase por los tres primeros puntos de cada caja –**Polyline**- y colocaremos un –**Flatten**- para hacer nuestra vida más fácil.

Este es el aspecto de nuestro modelo si fijamos la altura **H** de las cajas en 10:

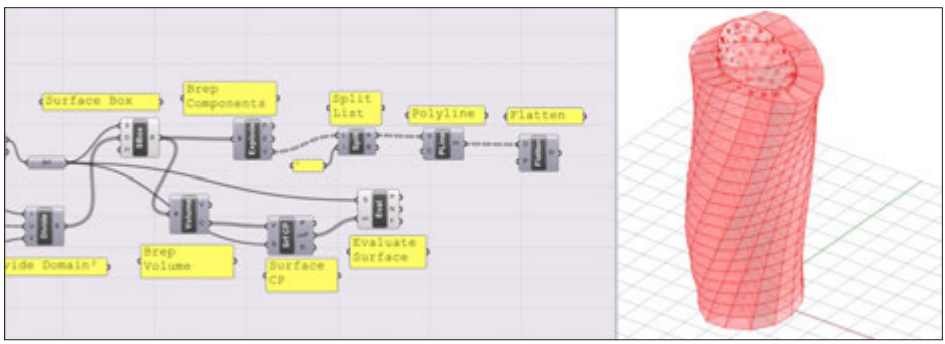


Figura 1.24

7. Diseño Reactivo:

A continuación vamos a cargar componentes de otra definición dentro de la nuestra.

Desde el menú *File* de GH, abre el archivo **Algomad_GH_Sol.ghx**, deberás ver algo así:

Vuelve a tu definición (puedes hacerlo con la lista desplegable que aparece al pulsar el botón en el margen superior derecho) y pega los componentes pulsando **Control+V**. Tu definición debería empezar a ser algo serio con una pinta parecida a esta:

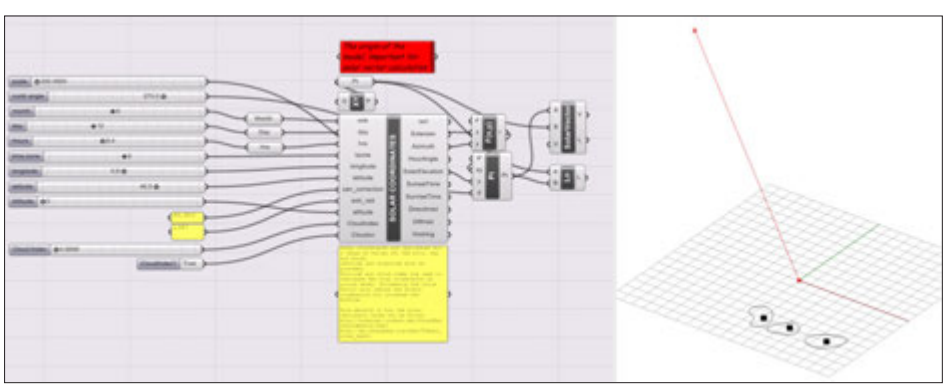


Figura 1.25

Selecciona todos los objetos de la definición y pulsa **Control+C** para copiarlos al portapapeles.

Vuelve a tu definición (puedes hacerlo con la lista desplegable que aparece al pulsar el botón en el margen superior derecho) y pega los componentes pulsando **Control + V**. Tu definición debería empezar a ser algo serio con una pinta

parecida a esta:

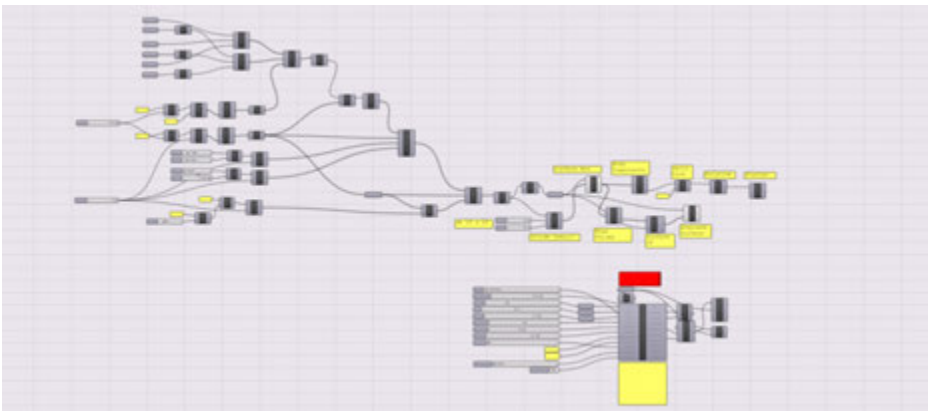


Figura 1.26

Lo que acabamos de pegar es una definición que sirve para calcular la posición del sol en cualquier momento. Antes hemos calculado la normal de cada región de *Loft* y ahora querremos compararla con la dirección del sol.

El comportamiento de la piel de nuestra torre variará en función de la incidencia del sol, si le da directo la piel se abrirá y si no lo hace la piel se cerrará.

Para evaluar el grado de paralelismo entre dos direcciones (vectores) hemos de utilizar el *producto escalar* de dos vectores (o “producto punto”). Si dos vectores unitarios son paralelos su producto escalar será 1. Si son perpendiculares será 0. Si son paralelos pero opuestos será -1.

El resultado del producto escalar lo usaremos para construir, mediante una función, un valor que utilizaremos como la altura de cada escama de la superficie de nuestra torre.

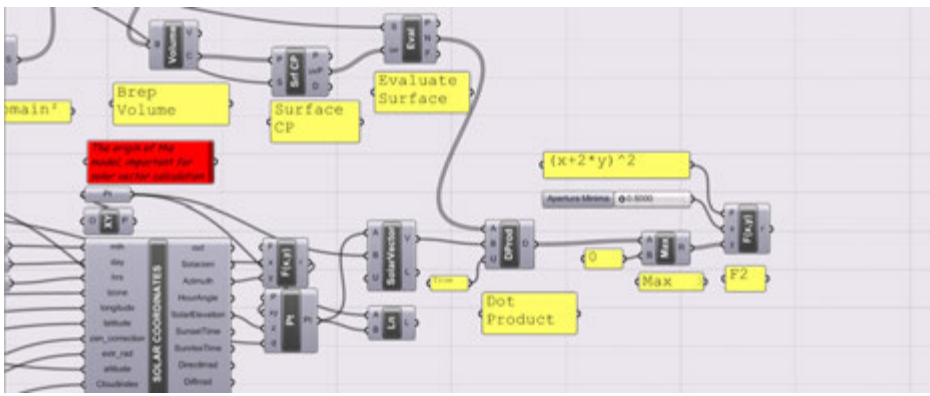


Figura 1.27

Calculamos el producto escalar entre el vector del sol y los normales de la superficies *Loft* -**dot product**- teniendo cuidado de pedirle que **normalice los vectores**.

Como vamos a tener valores positivos y negativos le pedimos que nos calcule el máximo –max- de cada valor y 0, de esta forma, si los vectores no tienen el mismo sentido (son opuestos o apuntan para cuadrantes distintos y por tanto el producto escalar es negativo) consideraremos que el producto escalar es 0.

Finalmente construiremos una función de dos variables -**F2**- que tendrá como ***X*** un factor que definamos con un slider y como ***Y*** el resultado del máximo previamente calculado.

La ecuación de la función se define en un panel aparte, de esta forma podemos cambiarla según nos convenga. Sin embargo, recomendamos que copies exactamente la que aparece en la figura y que luego la cambies poco a poco.

A continuación vamos a multiplicar cada normal por su factor correspondiente. Pero antes deberemos colocar un **reverse vector** para voltear las normales. Usaremos **vector multiply** para multiplicar cada vector por la salida de la función de dos variables.

El vector amplificado lo usaremos para desplazar -**move**- el punto evaluado sobre la superficie y así tener el vértice hacia el que extruiremos -**extrude point**- la polilínea creada anteriormente.

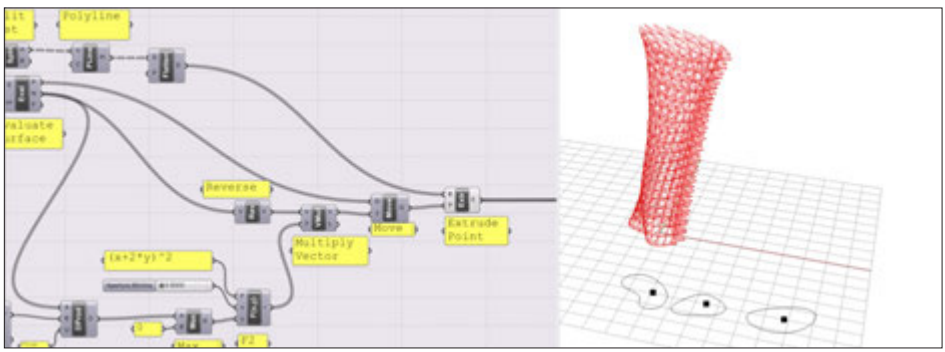


Figura 1.28

8. Un Toque de Color:

GH incorpora algunos componentes que permiten colorear los objetos en función de los parámetros que nosotros decidamos. El proceso de coloreado sigue una pauta que es casi siempre la misma. El color se suele atribuir a un objeto en función del valor de un parámetro asociado a ese objeto, puede ser el área, el volumen, alguna de las componentes de su centroide, la distancia a un punto, etc. El procedimiento podría definirse así:

Se mide el parámetro que se desea condicione el color, en nuestro caso vamos a medir el área de cada escama de la piel de la torre -**Brep Area**-. Este componente nos dará una lista de números reales que corresponderán al area de cada escama.

Ahora bien, para poder asignar un color a cada valor de área, debemos encontrar el valor mínimo y máximo de los aquellos a fin de establecer el dominio de mapeado de color.

Para hallar el mínimo y el máximo usamos el componente **sort** conectado a la lista de áreas, **sort** ordenará las áreas de menor a mayor. Si listamos o pedimos el primer elemento de la lista ordenada -**list ítem**- es decir, indicando el índice 0 con un panel, obtendremos el valor más pequeño de áreas. Si ahora volteamos la lista ordenada -**reverse list**- y listamos igualmente el primer elemento, obtendremos el valor máximo.

Colocamos ahora un componente **gradient** y haciendo botón derecho > **presets**, elegimos el esquema de colores que más nos guste. Le conectamos en **L0** el valor mínimo de áreas, en **L1** el valor máximo y en **t** conectamos la lista de áreas.

A cada valor de aérea le estará asignando un color determinado en función de los máximos y mínimos hallados.

Por último, y con esto terminamos, cargamos un componente - **custom pre-**

view – que nos permitirá ver la geometría del color calculado. Conectamos en G la superficie de extrusión y en **S** la salida de color del **gradient**.

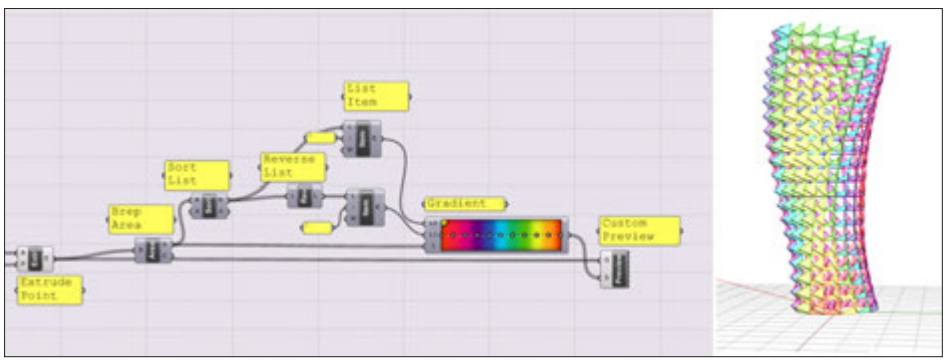


Figura 1.29

Con esto damos por finalizado el ejercicio guiado.

DIA 2. EXPERIENCIAS PARTICULARES

El segundo día del taller de Grasshopper se reservó para que cada asistente desarrollará, individualmente o en equipo, un tema de manera más personal, bien como continuación de algún aspecto tratado durante el ejercicio guiado, bien como experimento sobre alguna inquietud personal o bien como profundización sobre alguna de las ideas propuestas por los tutores. Esta estructura de taller partido en dos, con 4 de las 8 horas dedicadas al trabajo autónomo permitió que en el conjunto del taller se tratasen temas muy variados y problemáticas muy diversas.

La continuación lógica del ejercicio guiado del primer día consistió en que algunos de los equipos explorasen un poco más las posibilidades de construcción y población de superficies que Grasshopper ofrece. Grasshopper es tremendamente potente en las operaciones de deformación, replicación y adaptación de elementos predefinidos. En este sentido, equipos como el de Carles Ferrán, Leonardo Novelo y Luis Fernando Garcia Lara completaron pequeñas experiencias de población de superficies con componentes personales a modo de bloques.

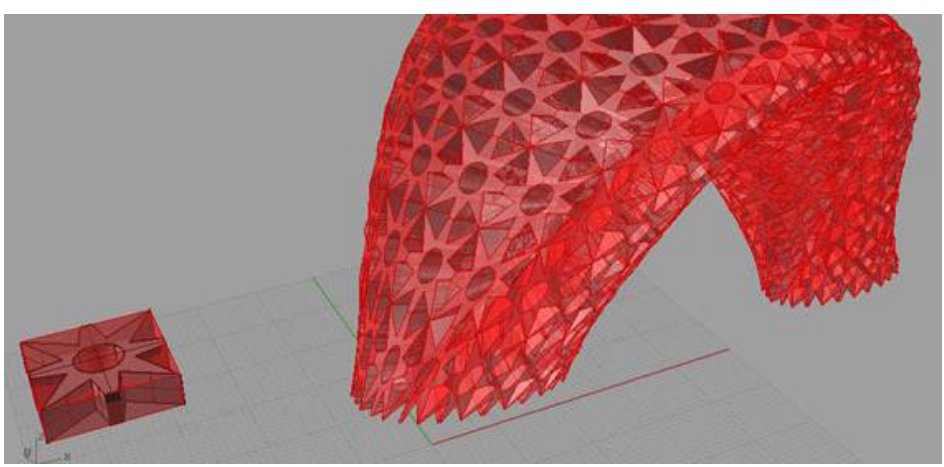


Figura 2.1

Este tipo de operaciones son fácilmente definibles en Grasshopper pero arrojan resultados normalmente espectaculares a base de gran cantidad de operaciones de deformación, operaciones que por otro lado requieren gran cantidad de potencia de cálculo y pueden llevar al desastre a casi cualquier sistema informático.

Otro equipo que profundizó en la generación y definición de superficies fue el

formado por Antonio Millán y Oleg Ogurstov, que propusieron distintas alternativas para cubrir un pabellón de proporciones alargadas.

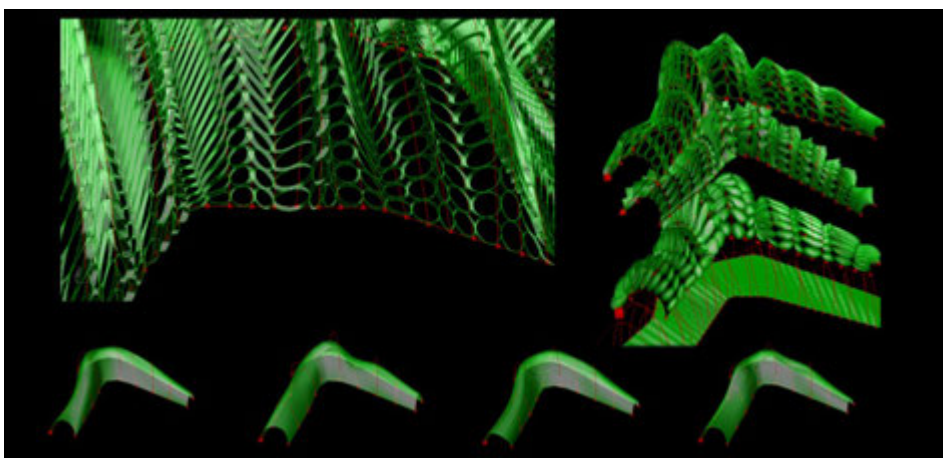


Figura 2.2

De entre las experiencias personales que algunos de los asistentes quisieron desarrollar durante el taller, destaca la de Pablo Delgado por conectar con las dos anteriores. Pablo planteó la necesidad de cubrir un estadio de atletismo, objeto de su proyecto final de carrera, con elementos constructivos repetitivos, para lo que utilizó las herramientas descritas en el ejercicio guiado.

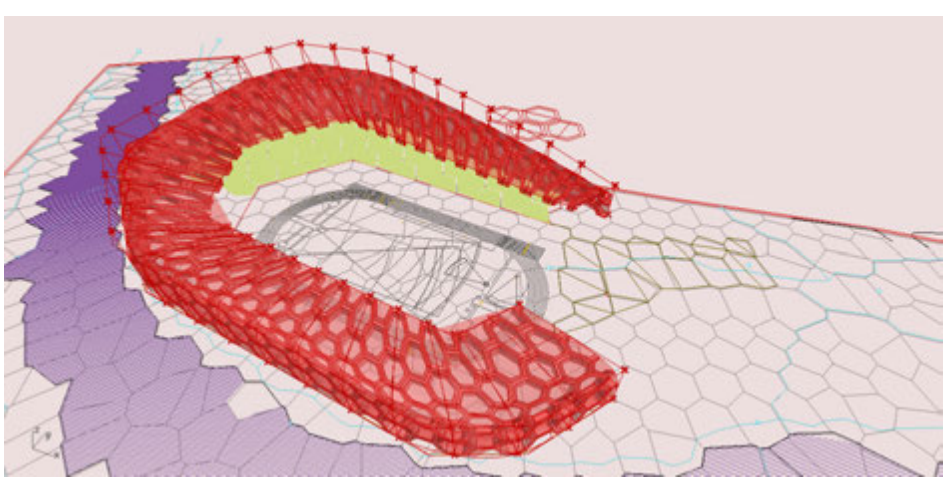


Figura 2.3

Tomi Sauquillo exploró las posibilidades de expresión formal de determinadas superficies pero en lugar de utilizarlas como elemento de referencia para colocar componentes aplicó operaciones de recorte sobre ellas obteniendo lo que podría entenderse como envolventes reactivas, en las que el tamaño de los huecos se relaciona con el valor de un parámetro determinado, como puede ser la proximidad de un elemento atractor.

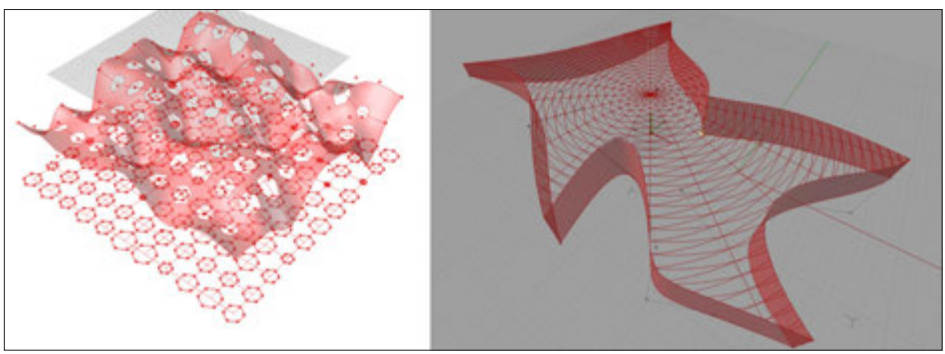


Figura 2.4

Simon Pierce y Adriá Bassaganyes quisieron probar las posibilidades de Grasshopper para definir geometrías de cubiertas acristaladas y poder emplearlas en su desempeño profesional. Grasshopper es muy potente en el trabajo con mallas pues permite tener un control sobre el grado de planeidad de aquellas a la vez que permite introducir restricciones geométricas de forma sencilla.

En cuanto a las líneas propuestas por los tutores del taller, varias de ellas tuvieron buena aceptación entre los asistentes y fueron desarrolladas con considerable exito.

Anna Pla y Mihai Brencea replicaron las experiencias de Gramazio y Kholer con la colocación robotizada de elementos de fábrica.

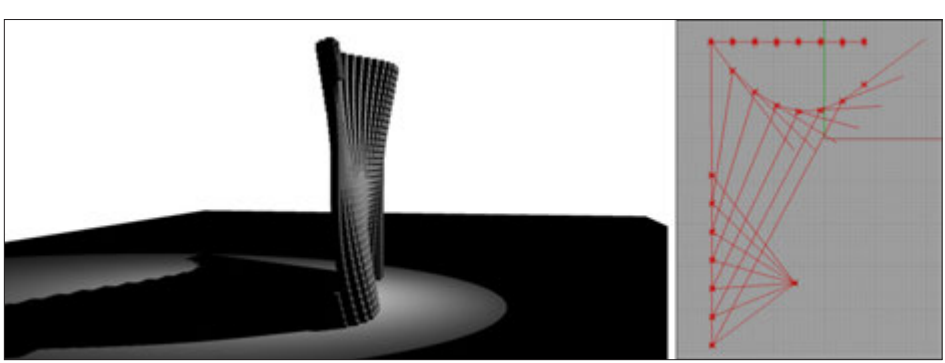


Figura 2.5

En este ejercicio, los ladrillos se acomodan a una superficie ideal variando el grado de rotación y solape entre cada uno. El resultado es una geometría compleja aproximada de una forma sencilla por elementos cotidianos.

Por su parte, Miguel Angel Lucas y Javier González aplicaron los principios de estática gráfica en una definición que construía el polígono funicular de fuerzas para hallar la catenaria entre dos puntos.

Por último, dos equipos profundizaron en ideas relacionadas con energía solar. Iván Pajares y Pablo Miranda estudiaron el asoleamiento de una fachada de lamas, un ejercicio no excesivamente novedoso pero que sirve para entender perfectamente las posibilidades de Grasshopper en cuanto al análisis en tiempo real de condiciones.

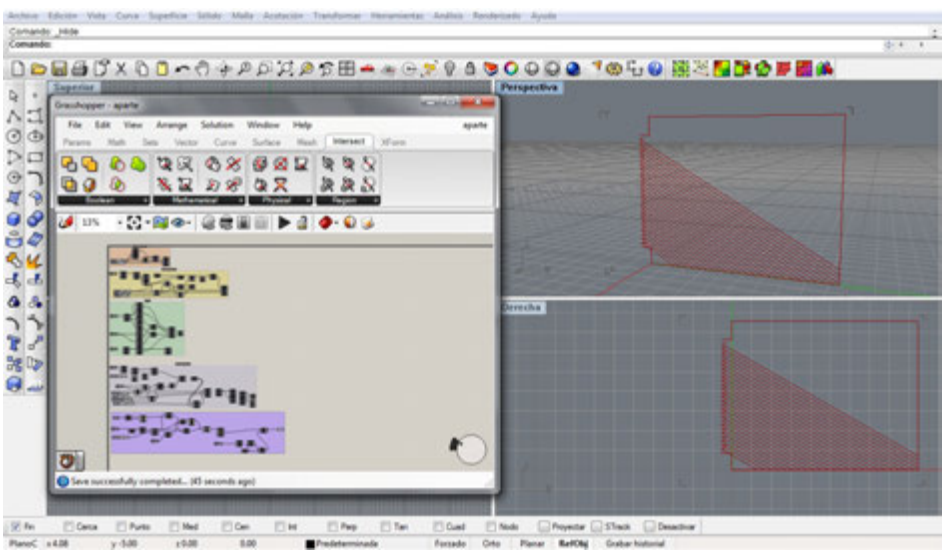


Figura 2.6

Por su parte, Gonzalo Besvievsky, Joan Pau Aragoneses y Carlos González aprovecharon la calculadora solar proporcionada en el ejercicio guiado para construir una definición que calculase la irradiancia sobre una superficie determinada.

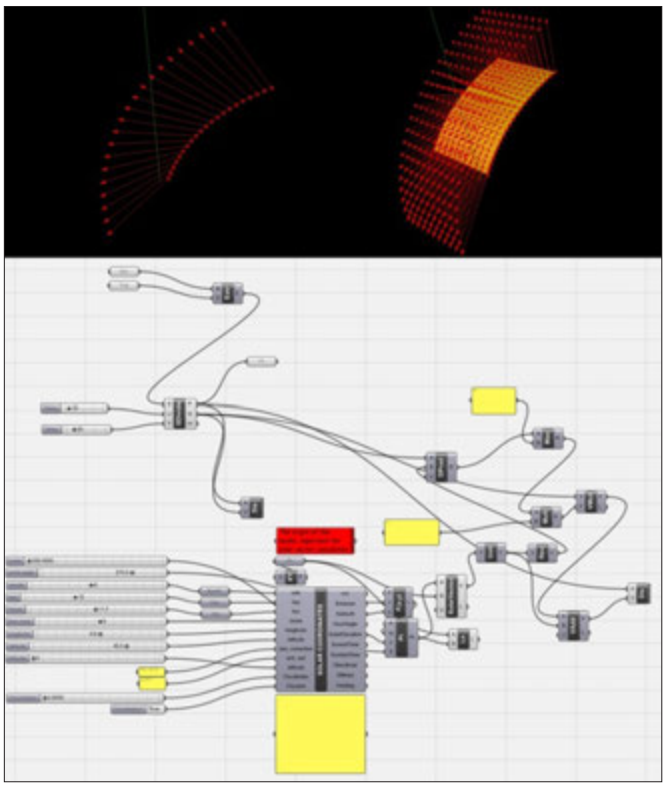


Figura 2.7

CONCLUSIONES

Los resultados de un taller de 8 horas de Grasshopper preparado para un público inexperto y con una temática intencionadamente difusa sólo dejan lugar para el éxito si los asistentes se marcharon de Algomad 2010 no con la sensación de saber más de lo que sabían al llegar sino con la sensación de querer saber más de lo que querían conocer al llegar.

Y sinceramente creemos que así fue.

Ni Grasshopper ni los fundamentos del diseño paramétrico pueden aprenderse ni enseñarse en 8 horas, el primero porque requiere un manejo constante y una atención permanente, ya que gracias a la activa comunidad que lo desarrolla, prueba y utiliza, está en continua mejora y evolución, y los segundos porque obligan a reestructurar en cierta modo la forma cómo las decisiones de diseño se ordenan en nuestra mente, obliga a priorizar variables sobre otras y a ordenar todo o parte del proceso en una serie de algoritmos que muchas veces constituyen la verdadera esencia del diseño, el manual de instrucciones de una idea genial.

Taller 3

RhinoScript

Quim Moya

Breve introducción a Rhinoscript

Rhinoscript es un lenguaje de programación basado en Visual Basic, y por lo tanto diseñado para ser asimilado con mas facilidad que otros lenguajes. Debido a que su sintaxis es sencilla no resulta difícil generar rutinas básicas con las que podemos automatizar todo tipo de procesos al dibujar con Rhino. Si ya contamos con Rhinoceros, para acceder al sistema de edición de Rhinoscript solo tenemos que ir al menú Herramientas/Rhinoscript. Sin embargo nos puede facilitar muchos las cosas descargarse un módulo adicional de edición llamado "Monkey", que podemos conseguir en la misma pagina Web de Rhino y que incorpora importantes ventajas.

Para entender un poco su funcionamiento podríamos empezar por establecer cuál es la diferencia entre generar geometrías a partir de un sistema de programación o generarlas por medios convencionales.

En la línea de comandos, o a través de los iconos, podemos realizar acciones puntuales sobre el dibujo como por ejemplo escribir el comando "_Line" y luego dar los valores de un par de puntos como por ejemplo: "0,0,0" y "0,10,15". Se puede repetir esta operación varias veces para generar una concatenación de órdenes que dibujen líneas. Este es un sistema rudimentario de empezar a automatizar procesos repetitivos pero no tenemos ninguna capacidad para establecer condiciones que regulen su ejecución. Tampoco tenemos capacidad para hacer cálculos matemáticos que nos permitan generar datos y almacenarlos. Aquí es donde la programación ofrece grandes ventajas respecto a sistemas de trabajo tradicionales pues no sólo podemos realizar acciones puntuales sino que podemos establecer en qué condiciones se realizarán y qué datos queremos extraer.

Un lenguaje de programación al igual que un lenguaje humano tiene una serie de reglas que se tienen que cumplir para que pueda transmitir información de manera correcta. Este conjunto de reglas conforman una sintaxis y en el caso de los sistemas digitales esta sintaxis suele ser muy estricta. Sin embargo, una vez que se han entendido los principios básicos, programar no es una tarea mucho más complicada que dar órdenes a alguien que no tiene sentido común; se ha de ser extremadamente prudente y prever que las posibles casuísticas que se puedan dar en su ejecución no conduzcan al desastre.

Hay tres elementos que tendremos que empezar a tener en cuenta si queremos utilizar la programación con Rhinoscript para generar rutinas, la primera es su sintaxis, la segunda sus posibilidades y la tercera su función.

La sintaxis de Rhino se basa en estructuras sencillas que incluyen expresiones simples (asignaciones, operaciones y llamadas a funciones) y expresiones agrupadas tales como condicionales "if a> b then c = d" o repeticiones tales como "for a = 0 to 10". Aun que Hay más variantes, comprendiendo el funcio-

namiento de estas dos y viendo cómo se disponen en una rutina resultara fácil extrapolar el funcionamiento a las demás. Se debe tener en cuenta que detrás del sistema de programación hay una manera de pensar en la que hay cosas que podemos decirle al ordenador que haga y otras que no.

Por ejemplo si nosotros queremos que Rhino nos haga 6 líneas rectas, cada una junto a la anterior, le diríamos en lenguaje humano “empezando en este punto de aquí haz una recta hasta allí, después comenzando en un punto junto al primer punto haz otra recta igual, y repite eso 6 veces “. Este proceso descrito en Rhinoscript sería exactamente así:

```
For a = 0 to 5
    Rhino.AddLine array(a, 0, 0) , array(a, 100, 0)
Next
```

En realidad lo que esta haciendo el ordenador no es otra cosa que repetir 6 veces la generación de una recta partir de unas coordenadas X, Y, Z que varían en cada repetición.

Este conjunto de funciones nos dan muchas posibilidades pero tiene límites, de hecho Rhinoscript, como lenguaje de programación, es muy limitado ya que esta pensado para actuar dentro de Rhinoceros y no de manera independiente. Por eso lo que podemos hacer con este lenguaje se limita a lo que podemos automatizar en Rhino sin cambiar su estructura interna. No podemos crear nuevas clases de objetos ni tampoco crear nuevos campos de información que se puedan visualizar. La mayor parte de limitaciones son precisamente de visualización ya que todo lo que no podemos dibujar con curvas, superficies o escribir en una cadena de texto permanecerá invisible para nosotros. Por ello conviene ver Rhinoscript como una herramienta de dibujo avanzado más que de programación ya que de hecho esta ha sido su finalidad.

Rhinoscript fue pensado como una manera de dar más versatilidad a Rhinoceros para facilitar que se adaptase a casuísticas muy específicas que no pueden implementarse en un programa de uso general. Precisamente por eso hacía falta una “meta-función” que permitiera al usuario generar nuevas funciones según sus necesidades y por eso implementaron un lenguaje de programación. Pero se hacía evidente que éste no podía ser muy complejo ya que la mayor parte de usuarios no disponían del tiempo necesario para adentrarse en la selva que supone un lenguaje como C o C++, especialmente a nivel gráfico. Así pues recurrieron a un sistema de programación que ya había demostrado que era muy sencillo y que era utilizado por millones de usuarios como es Visual Basic. Por otra parte, al utilizar un lenguaje tan simplificado, el control que ofrecía sobre aspectos complejos de la programación, como por ejemplo la visualización, era muy pobre por lo que se recurrió a las estructuras predeterminadas de Rhino como medio de ahorrar este trabajo al usuario. Así pues nace un lenguaje muy

simplificado, ideal para gente sin mucha experiencia pero con muchas ganas (o necesidad) de generar nuevas rutinas. La función principal de Rhinoscript es pues permitir que Rhinoceros sea más ambivalente y se pueda adaptar a las diferentes casuísticas profesionales del mercado.

ORGANIZACIÓN DEL CURSO

Las clases se organizaron en dos sesiones de 4 horas cada una. En la primera se dieron conceptos básicos de programación y se facilitó unas rutinas sencillas

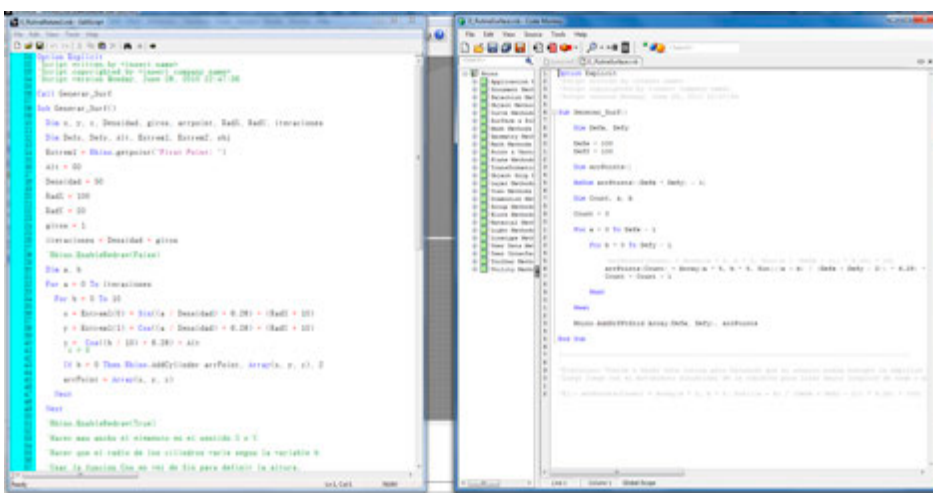


Figura 1. Comparativa entre el editor para Rhinoscript de Rhino (izquierda) y el módulo MONKEY (derecha)

para que hicieran a la vez de ejemplo y taller práctico. En la segunda sesión se facilitaron rutinas un poco más elaboradas que ya permitían hacer más modificaciones y obtener resultados más interesantes o cercanos a la arquitectura. Una idea que considero esencial es comprender que hacer una aplicación práctica suele ser más complicado de lo que parece por lo que de entrada no suele dar buen resultado intentarlo directamente sin caer en casos demasiado específicos o anecdóticos que no dan una idea lo bastante general de qué es programar. Hay por tanto una fase inicial en que es importante que la gente "juegue" (y a poder ser se divierta) con la programación de manera que interiorice los principios en profundidad y con ganas. La razón de este procedimiento radica en el hecho de cómo planteamos un problema en programación; Cuando uno no conoce el funcionamiento de un lenguaje suele plantear las rutinas de una manera mucho menos eficaz que cuando se conocen los recursos de los

que se dispone. Por eso es necesario que primero se tengan las herramientas y luego se planteen los problemas. Por otra parte la experiencia hace cambiar mucho este planteamiento de manera que incluso es recomendable atacar el mismo problema varias veces y desde diferentes perspectivas para explorar todas las posibilidades de resolución. Debido a esta idea el primer día se dieron los conceptos básicos de funcionamiento de Rhinoscript y se trabajó con un ejemplo muy abstracto mientras que el segundo día se plantearon directamente rutinas más complejas y más concretas.

PRIMER DÍA:

Se comenzó por introducir a nivel teórico el funcionamiento del lenguaje Rhinoscript de manera que en la medida de lo posible los participantes pudieran seguir el código de las rutinas facilitadas. Los conceptos iniciales comenzaron con una

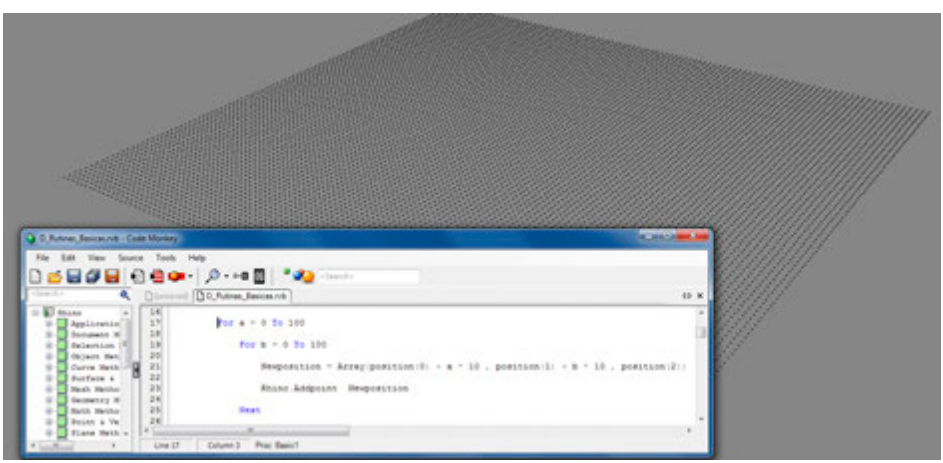


Figura 2. Ejemplo del estado inicial de la función para generar mallas de puntos

descripción más o menos detallada de los posibles editores que nos permiten programar en RhinoScript, como el editor propio de Rhino y otro editor llamado Monkey que ya hemos mencionado al comienzo. Las diferencias entre ellos son esenciales para alguien que no tenga nociones de programación ya que empezar con un sistema que obliga a tener buenas costumbres y al mismo tiempo facilite el proceso de escritura no es lo mismo que empezar con un programa que acepta casi cualquier cosa . Por esta razón Monkey es un editor más adecuado ya que además de ser más estricto con la escritura y no permitir mucho “faltas”, también permite, entre otras cosas, parar la ejecución de una rutina para comprobar su código en funcionamiento. Esta posibilidad es muy importante cuando

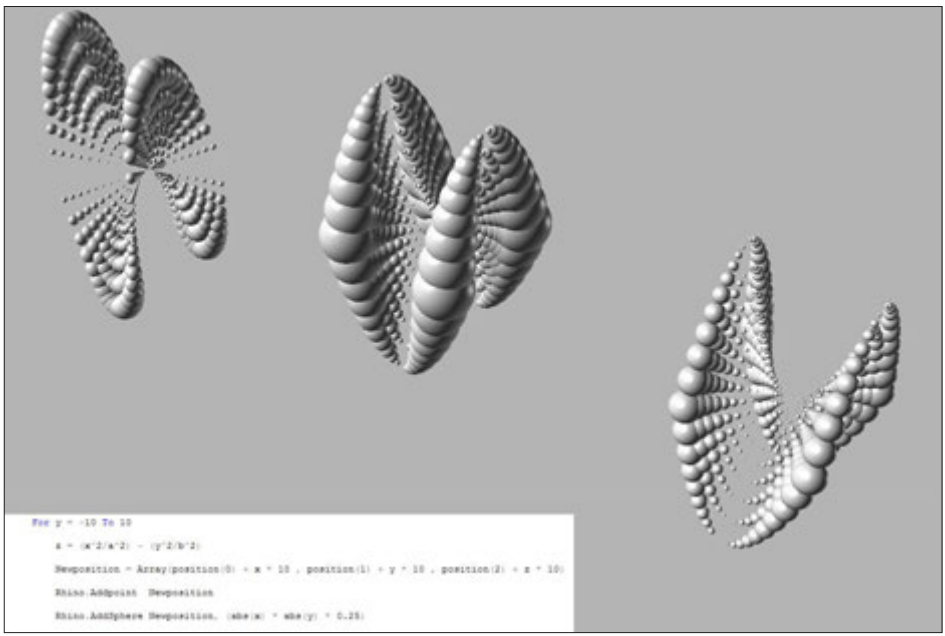


Figura 3. Ejemplo del trabajo realizado a partir de una malla de puntos (Juan Gonzalo)

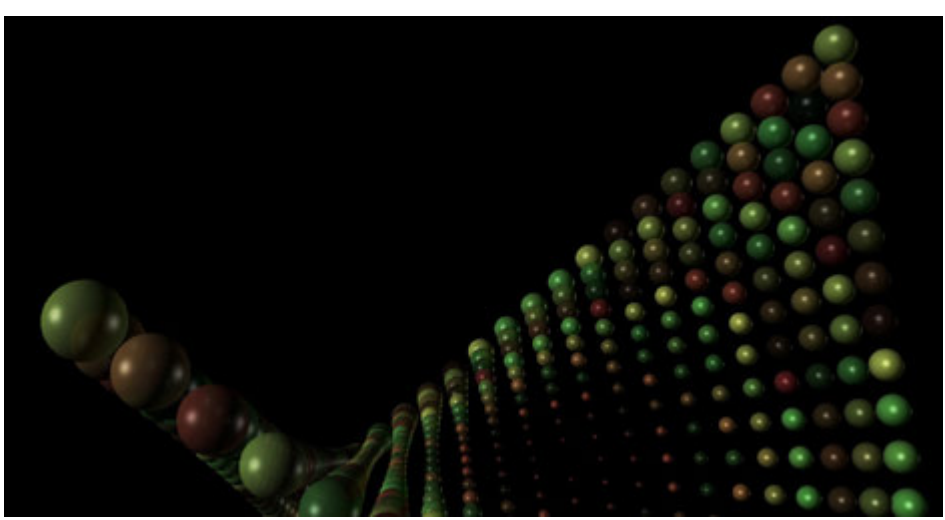


Figura 4. Otra derivación sobre el mismo tema (de Carlos gonzalez)

se programa ya que permite entender con más facilidad lo que está haciendo el ordenador mientras ejecutamos nuestra rutina. Sin esto hay muchos errores de programa que serían muy difíciles de detectar o de rastrear su origen.
El siguiente concepto que se explicó fue el de variable. Las variables son las encargadas de retener datos durante la ejecución del programa y se clasifican según el tipo de información que pueden contener (numérica, textos o valores booleanos). Su funcionamiento tiene que estar muy claro y por lo tanto se debe comprender su "sintaxis" y su papel en el código del programa. Por último se describieron las funciones más básicas que se utilizarían en las aplicaciones que íbamos a ver a continuación, como el "if", el "for" y el "while" y la estructuración de éstas en una rutina. Las funciones tienen una gramática propia que les permite generar "frases" que el ordenador puede entender, básicamente o bien son sistemas que establecen condiciones para la ejecución de las diferentes líneas de código, o bien modifican los valores contenidos en las variables según una serie de patrones. Cada apartado tenía unas prácticas propias con las que se podían ensayar las explicaciones teóricas, pero por cuestiones de tiempo estas no se pudieron realizar de manera que se pasó a trabajar directamente sobre una rutina con la que se tenían que generar los primeros resultados de aquel día.

La idea detrás de la generación de la rutina consistía en facilitar un código sencillo y polivalente que el alumno pudiera entender con facilidad para modificarlo a su gusto. Una vez que se empieza a modificar el código y se obtienen resultados se va adquiriendo experiencia en el funcionamiento general de las funciones dentro de la programación. Este proceso permite romper el hielo con más facilidad que empezar con una rutina escrita desde cero y por otro lado permite obtener unos primeros resultados concretos para trabajar con ellos. El planteamiento consistió en generar una malla de puntos a partir de los cuales se podría experimentar tanto con la sustitución de estos por otros elementos como en la modificación de sus coordenadas. Ambas operaciones son muy sencillas y no requieren de muchos conocimientos ni experiencia para poder realizarse.
Generar una malla de puntos no es una decisión arbitraria ya que lo primero que se debe establecer si queremos generar objetos en Rhino es un punto de inserción en el que poder insertar elementos, por lo tanto la manera más básica de empezar a trabajar sobre una rutina de este tipo consiste en generar posiciones en el espacio o puntos. Por otra parte las estructuras de repetición son una de las maneras más básicas de generar posiciones en el espacio o repetir procesos en una rutina y por lo tanto son una de las herramientas más utilizadas en programación. Precisamente por todo ello la idea de la primera rutina recurrió a estos dos planteamiento tan esenciales. A partir de dos estructuras de repetición de tipo "for" se generó una matriz de posiciones. Lo primero fue entender cómo se generaban los valores de las X, Y y las Z a partir de estructuras de repetición.

Una vez entendido esto se pasó a alterar estas fórmulas para obtener distintos tipos de resultados como por ejemplo superficies hiperbólicas. Realmente la idea era jugar con modificaciones que afectaran exclusivamente al valor de Z de manera que las superficies se obtendrían por las diferentes formulaciones sobre el valor de Z según los valores de X e Y. Poco a poco fueron apareciendo variantes en las que también la X y la Y se modificaban de manera que las superficies no eran ya sólo hiperbólicas o parabólicas sino que eran híbridas. El siguiente paso fue precisamente adentrarse un paso más en este proceso alterando las diferentes variables que definían la formulación hiperbólica para descubrir nuevos aspectos y potencialidades de esta, ya sea modificando las constantes, multiplicando los valores de X y de Y por diferentes cifras o alterando los exponentes de modo que fueran apareciendo toda una serie de nuevos elementos inclasificables pero que dado el caso pueden ser igualmente útiles. Lo más importante es que el alumno integre la idea de que las posibles superficies se limitan a todo lo que puede ser formulado con cifras y no a los tipos ya conocidos, de esta manera se rompe la idea de seguir un objetivo fijado (obtener una superficies a partir de esta o aquella fórmula) para intentar abrir el campo de exploración con la idea de combinar diferentes fórmulas o incluso alterar de manera libre para obtener un objeto “propio”.

Una vez el tema de generar diferentes posiciones en el espacio a partir de

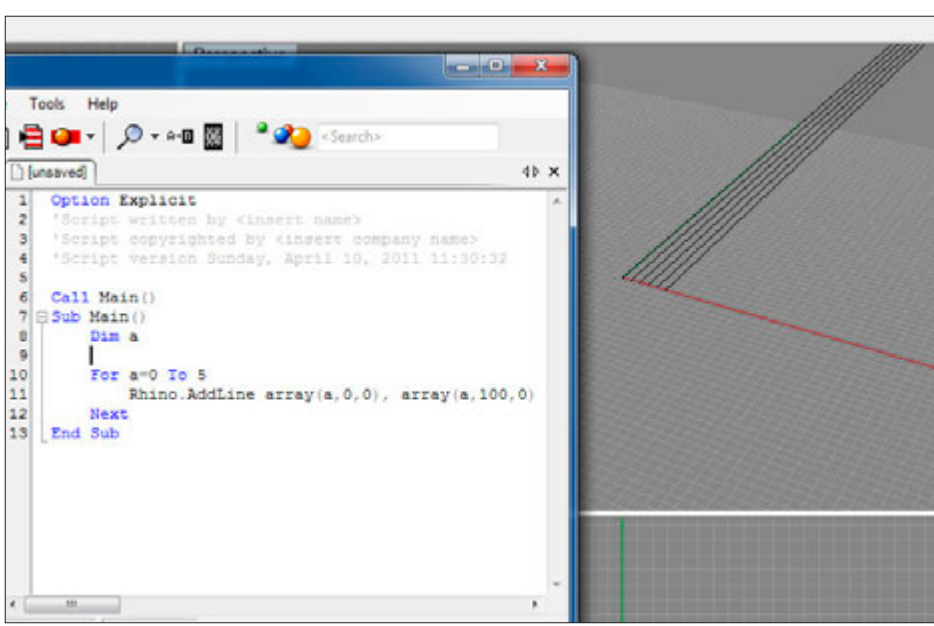


Figura 5. Imagen con el código y el resultado de la rutina

una malla de elementos estuvo bastante trabajado se pasó a sustituir lo que hasta ahora eran sólo puntos por esferas. El punto esencial de este cambio fue introducir una nueva variable en el juego, pues ahora además de la posición X, Y y Z aparece también el radio.

Siguiendo pues el planteamiento anterior este radio podía responder a varios factores y el primero y más sencillo fue que su valor se derivara de la altura.

SEGUNDO DÍA:

Para la siguiente sesión se introdujeron nuevas funciones de Rhinoscript específicas para generar objetos con Rhino como por ejemplo superficies, curvas, cilindros, etc. Para ponerlas en práctica se prepararon tres rutinas más que iban añadiendo gradualmente mayor complejidad.

En el primer caso se recogían todos los puntos de la rutina de la sesión anterior y se transformaban en una superficie de manera que uno podía jugar con ella y generar elementos que podían empezar a parecerse a objetos arquitectónicos. En la segunda rutina ya iba un paso más allá y además de generar una superficie, se añadía una barra en cada arista y un conector en cada vértice. El resultado era que se generaba una fachada compuesta de una estructura que sostenía unos paneles. Quedó planteada la posibilidad de crear varias mallas paralelas con las que crear un muro cortina o una fachada ventilada. Finalmente se planteó una estrategia que no se había practicado hasta ese momento que era la de plantear una estructura a partir de la rotación de un elemento a alrededor de un eje. Aunque la idea puede parecer muy sencilla los resultados pueden ser mucho más ricos de lo que se prevé en un primer momento.

Análisis de una rutina básica

Para empezar con el proceso de descripción de una rutina comenzaremos por una muy básica y la analizaremos, para pasar después a ver una más compleja. En la introducción he descrito una rutina que haría algo parecido a lo que sigue: “Empezando en este punto de aquí hago una recta hasta allí, después comenzando en un punto junto al primer haz otra recta igual, y repite eso 6 veces”. Este proceso descrito en Rhinoscript sería exactamente así:

```
For a = 0 To 5
    Rhino.AddLine array(a, 0, 0), array(a, 100, 0)
Next
```

El primer elemento que hemos de entender en este proceso es “a”. Es una variable y su función es almacenar un dato. Este dato puede ser tanto una cifra como una cadena de texto, en este caso lo que guarda es una cifra. Lo segundo

que hay que descifrar es qué papel juega el “For ... To ...” y “Next”. Esta es una función interna de Rhinoscript y lo que hace es coger una variable y sumarle 1 (el incremento por defecto) hasta que el valor inicial de la variable supere un valor final que hayamos especificado. Estos dos valor son 0 y 5 en nuestra función y en total describen 6 pasos (0,1,2,3,4,5) ya que el primero es siempre 0. Esta función contiene un espacio que permite especificar una serie de acciones a realizar en cada paso. Delimitar este espacio es precisamente la función de “Next” que nos indica el límite hasta donde podemos poner líneas de código que se repitan. Todo lo que esté contenido entre “For ... to ...” y “next” se repetirá en cada paso. La variable que utilizará la función “For” para que haga de contador (e ir incrementando sucesivamente su valor en una unidad) es “a” y la línea de código contenida entre “For ... to ...” y “next” que se repetirá en cada paso es:

“Rhino.AddLine array (a, 0,0), array (a, 100, 0)”.

Fijémonos en que esta línea de código contiene la variable “a” en lo que debería ser la posición X del punto inicial y final. Lo que pasará es que se asignará el valor de “a” en X de manera que cuando se vaya repitiendo la función y “a” vaya aumentado su valor, también se irá incrementado el valor de X, de manera que la primera vez será 0 y el último 5.

Sin entrar mucho en detalles podríamos decir que “Rhino.AddLine” es una función que genera líneas y que necesita dos coordenadas para poder ejecutar (un punto inicial y un punto final), sin estos datos es imposible crear una línea. ¿Para qué especificamos los puntos con la palabra “Array” delante? Esta es una función un poco más complicada pero nos podemos quedar con la idea de que lo que hace es empaquetar 3 cifras y transformarlas en una coordenada tridimensional del tipo X, Y, Z que Rhinoscript puede interpretar. Por lo tanto si nosotros le queremos dar una posición a una función de Rhinoscript muchas veces usaremos la función “array”.

Los datos empaquetadas en la función “array” se corresponderán con la X, Y, Z de un punto en el espacio, y por tanto la “a” (que irá variando a cada paso) dará el valor de X. Las funciones de Rhinoscript que ejecutan tareas específicas de Rhino suelen ir precedidas del prefijo “Rhino.”, Hay que tener en cuenta que estas casi siempre necesitan datos para poderse ejecutar, y como hemos visto en el ejemplo anterior estos datos suelen necesitar que las empaquetamos de manera que el ordenador entienda que aquellas tres cifras que le estamos dando son una posición. En general lo que pasa es que el ordenador espera recibir los datos en un formato concreto, y si las damos de otra manera no entiende lo que le estamos diciendo.

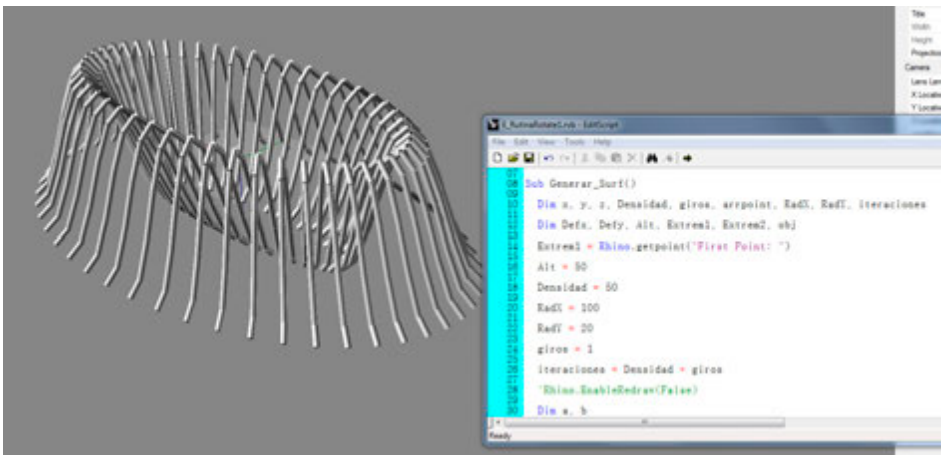


Figura 6. Estadio inicial donde solo se generan los arcos sin conexiones entre ellos

Análisis de una rutina algo más compleja

Como hemos visto con esta primera rutina, a pesar de ser muy corta están actuando muchos conceptos. Ahora pasaremos a ver una rutina más larga con la que podremos obtener resultados como los mostrados en la imagen adjunta al final del código. Esta rutina no es demasiado extensa pero conceptualmente es mucho más compleja que la anterior ya que relacionar cada acción con cada elemento del código es bastante más difícil. El código completo sería este:

```
Sub Generar_Surf()
Dim x, y, z, Densidad, giros, iteraciones, arrpoint(), RadX, RadY
Dim Defx, Defy, Alt, Extrem1, Extrem2, obj
Extrem1 = Rhino.getpoint("First Point: ") '11
Alt = 50
Densidad = 70
RadX = 100
RadY = 20
giros = 1
Rhino.EnableRedraw(False) '23
Dim a, b '25
iteraciones = Densidad * giros
ReDim arrPoint(iteraciones, 10) '29
For a = 0 To iteraciones '31
For b = 0 To 10 '33
x = Extrem1(0) + sin((a / Densidad) * 6.28) * (RadX + b * 10)
y = Extrem1(1) + Cos((a / Densidad) * 6.28) * (RadY + b * 10)
z = sin((b / 20) * 6.28) * Alt
If b > 0 Then Rhino.AddCylinder arrPoint(a, b - 1), Array(x, y, z), b / 10 '41
```

```
algomad 2010 76
If a > 0 Then Rhino.AddCylinder arrPoint(a - 1, b), Array(x, y, z), b / 10 '43
arrPoint(a, b) = Array(x, y, z) '45
Next
Next
Rhino.EnableRedraw(True)
End Sub
```

El código de la rutina empieza con un punto especificado con el Mouse (P). Luego a partir de una serie de propiedades en forma de variables genera una serie de arcos, dispuestos en una cuadricula radial, alrededor de un eje vertical centrado en el punto P. Para crear estos arcos se han de considerar dos rotaciones, una horizontal que va barriendo todos los arcos alrededor del eje central, y otra rotación que recorre todos los vértices que formaran cada arco. La primera nos indica en qué arco estamos y la segunda permite recorrer cada vértice del arco en sí. Para establecer el número de arcos hemos definido una variable llamada "densidad" con la que podemos especificar cuántos arcos queremos crear. Así pues tendremos que crear dos estructuras de repetición, una que recorra todos los arcos y otra que recorra todos los puntos de cada arco y para esto utilizaremos la función de repetición "For ... to ... next".

Descripción del código

El código comienza con la función "Sub" que marca el principio de la rutina y sirve para indicarle al ordenador que todo lo que viene a continuación forma parte de esta. El ámbito en el que podemos poner funciones termina en "End Sub" que está al final de todo. Por lo tanto todo lo que ponemos entre "Sub" y "End Sub" formará parte de la rutina. Después de "Sub" aparece "Generar_Surf ()" que es la etiqueta con la que llamaremos a la rutina en Rhino y que lleva dos paréntesis al final para indicar que es una función (aunque no incluya argumentos entre los paréntesis). A continuación del nombre de la rutina tenemos dos líneas que comienzan con "Dim" a lo que sigue una lista de nombres. De este modo estamos "declarando" las variables que vamos a usar en nuestra función pues "Dim" le indica al ordenador que los nombres que pondremos a continuación son identificadores que utilizaremos durante el programa para almacenar datos. Se han de declarar las variables antes de usarlas para que el ordenador sepa a que nos referimos cuando ponemos "Alt" o "longitud". Más abajo, en la línea 11 (los números se refieren al código original) aparece algo nuevo:

```
Extrem1 = Rhino.getpoint("First Point: ") '11
```

Esta línea asigna a la variable "Extrem1" el resultado de una función interna de Rhino, "GetPoint ('First point: ')". El operador "=" nos permite guardar lo que

hemos obtenido con la función “Rhino.GetPoint ()” en la variable “Extrem1”. Esta función nos devuelve un punto (o sea un listado de tres coordenadas) que se van a guardar en forma de listado. Al ser una función interna de Rhino va precedida de “Rhino.”. Este identificador indica que lo que viene a continuación es una función interna de la clase “Rhino” que a diferencia de otras funciones como “dim” o “For” pertenece solo al módulo Rhino. La clase es un concepto característico de la programación orientada a objetos en el que no vamos a entrar pues con RhinoScript en principio no se trabaja con clases aunque sea posible invocarlas. Finalmente, la cadena de texto que encontramos dentro del paréntesis (“First Point:”) es el texto que podrá leer el usuario cuando se le pida un punto con el mouse. Este punto será el centro de rotación en torno al cual pondremos los arcos. Por tanto, a partir de la línea 11, el programa tendrá almacenado en la variable “Extrem1” una posición en el espacio que podremos utilizar para nuestros cálculos. En las siguientes líneas, hasta la 23, lo único que el programa hace es asignar el valor correspondiente a las variables del programa que definen la cantidad de arcos, la altura de estos, etc.

```
Alt = 50    <- Altura
Densidad = 70                    <- Cantidad de arcos
RadX = 100    <- Radio de giro general
RadY = 20     <- Radio de los arcos
giros = 1   <- Numero de giros
```

En la línea 23 se activa una función que ayuda a que el programa se ejecute más rápido impidiendo que la pantalla se redibuje. Seguidamente en la línea 25 declaro dos variables más, “a” y “b” que serán los contadores o índices de las funciones de repetición que usaré más adelante. En la siguiente línea asignamos a la variable “iteraciones” el producto del número de giros por el número de elementos en cada giro, obteniendo así el número total de elementos que guardaremos en esta variable. Con esto termina la asignación de datos.

Una vez que tenemos todas las variables preparadas queda aún una operación previa que consiste en declarar un listado de valores que puedan contener todos los puntos necesarios a partir de los cuales se definirá el objeto. Como ya he mencionado anteriormente, para poder crear un objeto en Rhino se nos pide dar posiciones en el espacio como dato de entrada y por lo tanto debemos tener esos datos almacenados en variables. Estas posiciones se guardarán en un listado de valores que contendrán, en cada caso, una X, una Y, y una Z. Pero debemos tener en cuenta que si pretendemos generar 70 arcos y hacer 10 tramos de recta para cada arco, la cantidad de puntos que tenemos que guardar son 700. Es evidente que no podemos generar 700 variables a mano. Por esta razón se generaron las matrices o listas de valores, con los que una misma etiqueta puede contener múltiples datos. Para declarar este tipo de variables se

debe utilizar la función “ReDim” que se aplicara igual que la función “Dim”. Entre paréntesis se introducen las dimensiones de la matriz y la cantidad de casillas que contendrá. Para entender este concepto podemos imaginarnos un archivador que tenga cajones y otro que tenga cajones y carpetas. El primer caso de matriz sería como un archivador sencillo donde los datos se guardan en cada cajón y sería de una dimensión, el segundo tipo sería como un archivador con carpetas en cada cajón dentro de las cuales dejamos los datos y sería de dos dimensiones. Si por ejemplo ponemos “ReDim archivador (NumeroDeCajones)” hemos creado un listado de datos donde se guarda un dato en cada cajón, pero si ponemos “ReDim archivador (NumDeCajones, NumDeCarpetas)” hemos creado una matriz de dos dimensiones donde podemos dejar datos en cada una de las carpetas de cada cajón. Esto lo podemos hacer crecer indefinidamente “ReDim archivador (cajones,carpetas,separadores)” o “ReDim archivador (cajones,carpetas, separadores,apartados)”, etc. Se ha de tener en cuenta que con cada nueva dimensión se multiplica el número de datos de la matriz y aunque el número de dimensiones no tiene límite, la memoria del ordenador sí es limitada.

En nuestro caso nos interesa una matriz de dos dimensiones, ya que por cada arco queremos guardar 10 puntos que lo definan por lo tanto queremos generar una variable del tipo “Redim datos (arcos, puntos)”. Lógicamente en el código de programación “arcos” corresponderá a una cifra que dará el número de arcos que queremos generar y puntos será el número de puntos por cada arco. Sabiendo esto estamos en condiciones de interpretar lo que quiere decir la línea de código 29:

```
ReDim arrPoint(iteraciones, 10)
```

En primer lugar hemos especificado la función “ReDim” que nos permite crear una matriz. Después hemos indicado el nombre de ésta lista, “arrPoint”, y he-

Figura 7. Ejemplos de arcos obtenidos por la iteración de “b”, que corresponden directamente a curvas sinusoidales de distinta longitud de onda

mos especificado las dimensiones poniendo el número de arcos (iteraciones) y, tras la coma, el número de puntos por cada arco “10”. Pero ahora se han de llenar todas las casillas de la matriz con listados de tres datos X, Y y Z. En las líneas 31 y 33 podemos observar como he declarado una función de repetición que utilizan como índice “a” y “b”. Las funciones de repetición han sido descritas anteriormente y sabemos que disponen de un valor inicial del índice y un valor final. El valor inicial de la primera repetición es 0 “For a = 0 ...” y el valor final se corresponderá con el número de elementos (el número de arcos) “... To iteraciones”. Esto quiere decir que la variable “a” recorrerá todos los valores entre 0 y el número total de elementos y por tanto pasará por cada uno de los “cajones” de la variable “arrPoint”. Pero necesitamos un dato más para recorrer todas las casillas de “arrPoint” ya que es de dos dimensiones y además de cajones tiene carpetas. Es por ello que generamos otra función de repetición con un nuevo índice “b” que recorrerá todas las carpetas de cada cajón. Como cada arco consta de 10 puntos, si hacemos que “b” pase por todos los valores entre 0 y 10 recorreremos todas las carpetas (“b”) de cada cajón (“a”). Todo esto se condensa en estas dos líneas:

```
For a = 0 To iteraciones ‘31
For b = 0 To 10 ‘33
```

Primero se asigna a “a” todos los valores entre 0 y el número total de elementos. En cada paso de “a” hacemos que “b” recorra todos los valores entre 0 y 10, de manera que entre a y b se recorrerán todas las casillas de la variable “arrPoint”. Ahora, por cada una de las casillas se le asigna un valor de X, de Y y de Z. En las líneas que siguen generamos el valor de X, Y y Z con la siguiente formulación:

```
x = Extrem1(0) + sin((a / Densidad) * 6.28) * (RadX + b * 10)
y = Extrem1(1) + Cos((a / Densidad) * 6.28) * (RadY + b * 10)
z = sin((b / 20) * 6.28) * Alt
```

Antes he declarado unas variables x, y, z que hemos servirían para almacenar temporalmente las coordenadas de cada punto. Ahora se les asigna un valor antes de empaquetarlas con la función “array”. Unas cuantas páginas atrás he descrito como he almacenado en la variable “Extrem1” una posición facilitada por el usuario. Ahora usaré esta posición como punto inicial a partir del cual situar los puntos. Como esta posición se proyecta sobre el plano de trabajo, el valor Z no cambia por lo tanto sólo podré recoger un valor X y un valor Y. Es por esta razón por lo que la variable “Extrem1” sólo actúa cuando defino X e Y y no aparece en el valor de Z. Ahora pasamos a analizar la formulación con la que asigno valor a la X:

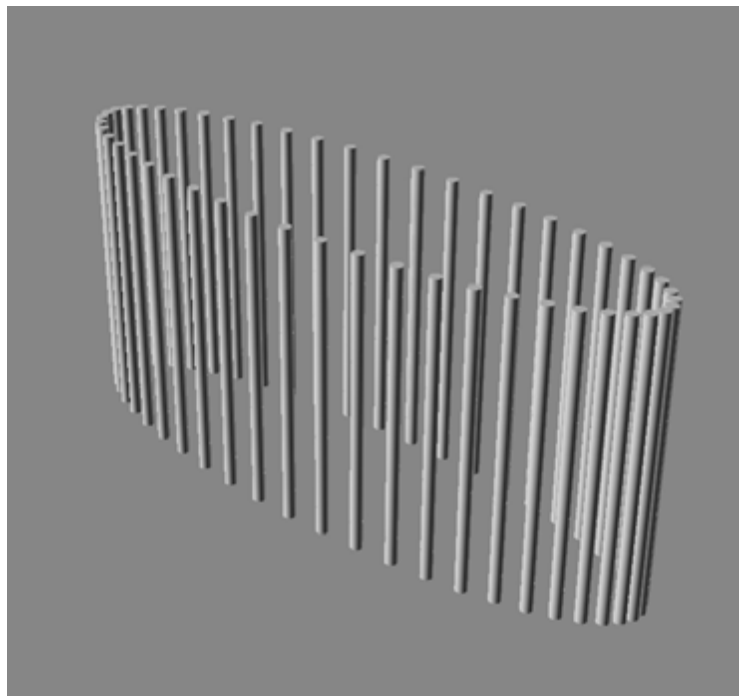


Figura 8. Ejemplo de lo que ocurre si el radio es constante

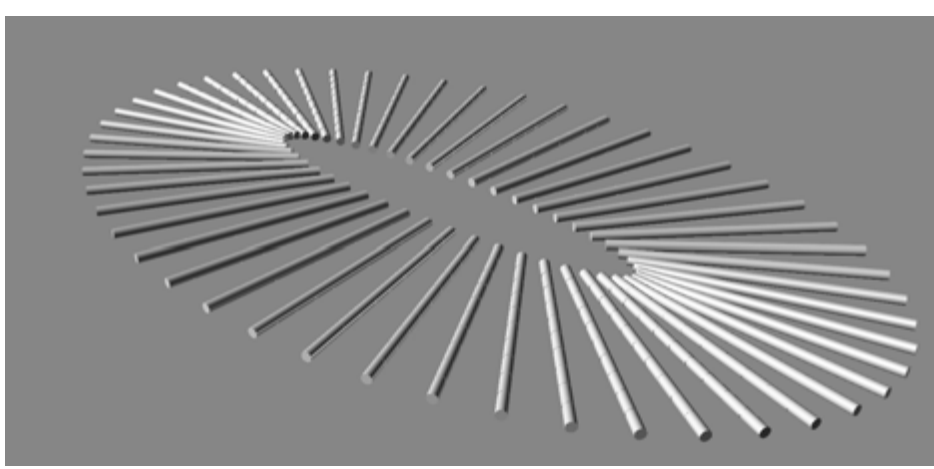


Figura 9. ejemplo de lo que ocurre si le damos valor 0 a la Z

x = Extrem1(0) + sin((a / Densidad) * 6.28) * (RadX + b * 10)

Como ya sabemos el operador “=” asigna un valor a una variable, en este caso la “x”. Lo primero que encontramos tras el “=” es “Extrem1(0)”. Como ya he dicho “Extrem” guarda un punto que el usuario ha marcado sobre el plano de trabajo, y este punto esta dado en coordenadas (x, y), de manera que el elemento 0 de la lista es la X y el elemento primero la Y. Debido a esto el primer valor que le asignamos a la X es “Extrem1(0)”. A este valor le sumamos el seno del ángulo que le corresponde (0º si es el primer elemento o 360º si es el último) multiplicado por el radio. El ángulo lo hemos de facilitar en radianes (de 0 a 2pi para una vuelta completa de 360º). El valor 2pi es igual a 6.28 de modo que este número ya está fijado y se multiplica por una relación entre “a” y el número de arcos por cada vuelta. Recordemos que “a” recorrerá todos los arcos, uno por uno. Si en cada vuelta tenemos un cierto número de arcos, al dividir el valor de “a” por el número de arcos en cada vuelta podemos saber en qué fase del giro estamos. Recordemos también que hemos dicho que en cada vuelta habría 70 arcos. Por lo tanto cuando “a” sea 0, la relación será 0/70, es decir 0 veces. Cuando valga 1 será 1/70, y por tanto corresponderá a 1/70 de vuelta. Así seguiremos sucesivamente hasta llegar a 70/70, que dará 1 (o sea una vuelta completa).

x = Extrem1(0) + sin((a / Densidad) * 6.28) * (RadX + b * 10)

Fijémonos en que si damos más de una vuelta obtendremos valores mayores que 1, y por lo tanto al elemento le corresponderá más de una vuelta. Con este sistema a cada elemento le corresponde un giro concreto según su posición en la lista y el número de arcos en cada vuelta. Multiplicando la relación obtenida por 2pi (6.28) que corresponde a una vuelta entera obtenemos el ángulo en radiantes.

x = Extrem1(0) + sin((a / Densidad) * 6.28) * (RadX + b * 10)

Una vez tenemos el seno necesitamos multiplicarlo por el radio pero también necesitamos que este radio aumente según se trate del primer punto del arco o del último. Si utilizáramos siempre el mismo radio, entonces todos los puntos de un mismo arco caerían en la misma vertical y no formarían un arco (fig-2).

Pero nosotros queremos que formen un arco y por lo tanto necesitamos variar el radio según si se trate del elemento 0 o 10 del arco. Este concepto se traduce en la formula en sumarle al radio el valor del contador “b” que nos indica en qué punto del arco estamos. De esta manera si estamos en el punto 0 del arco, el radio será “ RadX + 0 * 10”, y si estamos en el último punto del arco el radio será “ RadX + 10 * 10”.

Finalmente, sólo indicar que uso un radio diferente en X y en Y, para poder

generar formas elípticas, lo cual es más interesante desde el punto de vista del diseño.

En el caso de la Y las operaciones son iguales:

```
x = Extrem1(0) + sin((a / Densidad) * 6.28) * (RadX + b * 10)
y = Extrem1(1) + Cos((a / Densidad) * 6.28) * (RadY + b * 10)
```

Únicamente varía que, en lugar de buscar el seno, buscamos el coseno. Y el radio es el radio de la componente Y “RadY”.

Finalmente necesitamos asignar una altura a la Z para completar la posición de cada punto.

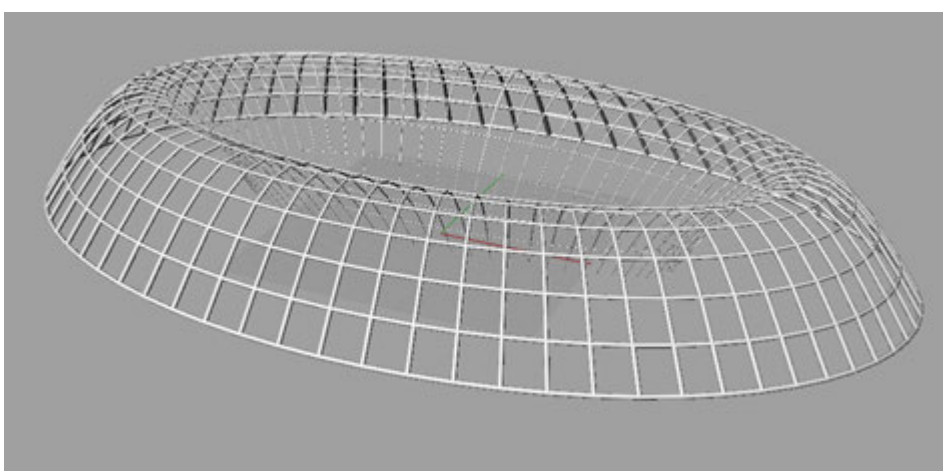


Figura 10. Script “GenerarSurf1”

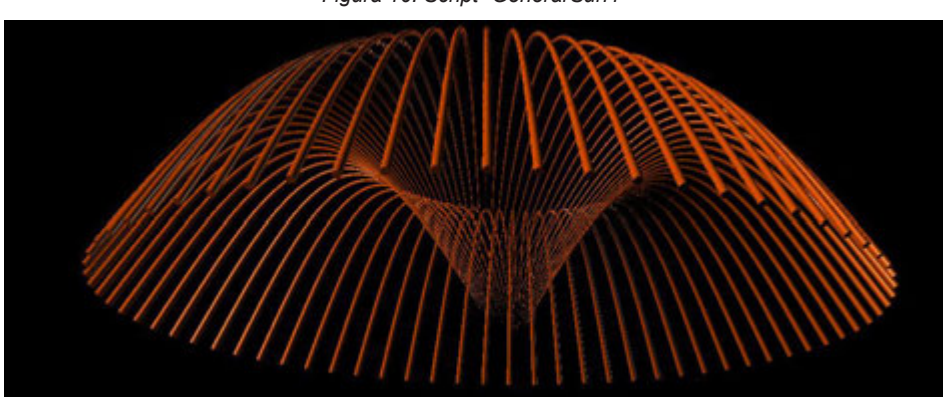


Figura 11. Variación sobre el tema obtenida durante la clase por Joan Pau

La operación que nos asigna valores a Z es:

```
z = sin((b / 20) * 6.28) * Alt
```

Esta formulación utiliza de nuevo una relación para obtener el ángulo, pero ahora lo hace a partir de “b”. Recordemos que no queremos dar una vuelta completa si no media vuelta que forme un arco, por lo tanto si los valores de “ b “ van de 0 a 10, necesitamos que la relación nos dé un valor máximo de 0.5 (media vuelta). Como “b” tendrá un valor máximo de 10 será necesario dividirlo por 20 para que nos dé 0.5 como máximo. Este valor se multiplica por 2pi al igual que en los casos anteriores de forma que obtenemos el ángulo, y a este se le aplica una función seno. El resultado se multiplica por la altura del arco.

Una vez hecho todo esto ya tenemos las coordenadas de cada uno de los puntos de cada uno de los arcos. Ahora nos falta dibujar los cilindros que definirán la estructura. El procedimiento para dibujar estos cilindros consistirá en tomar cada punto y hacer un cilindro que vaya del punto en cuestión hasta el punto anterior, con este procedimiento se puede dibujar todo el arco. Pero hay una excepción y es que cuando estamos en el primer punto, no hay ningún punto anterior, por lo tanto esta función se debe aplicar sólo si el punto no es el punto inicial (si “b” no es igual al valor inicial 0). Esto es exactamente lo que se prevé en la línea de código 41:

```
If b > 0 Then Rhino.AddCylinder arrPoint(a, b - 1), Array(x, y, z), b / 10
```

Esta línea 41 empieza con un condicional que evalúa si “b” es mayor que 0 (o sea si no estamos en el punto inicial); en caso de que no sea así, ejecutará la acción de generar un cilindro a partir de dos puntos, el presente y el anterior. Estos puntos deben ser facilitados en forma de listado de tres coordenadas y por eso uso la orden “Array” para empaquetar la X, Y y Z del punto presente. En el caso del punto anterior “arrPoint (a, b - 1)”, las coordenadas ya están “empaquetadas” y se ha de especificar el índice anterior (b - 1). “Rhino.AddCylinder” es una función interna de Rhino que genera un cilindro a partir de dos puntos y un radio, la estructura básica de la función responde al esquema “Rhino.AddCylinder Punt1, Punt2, Radio”, los dos puntos ya hemos visto como se han facilitado, pero el radio será variable y lo obtendremos dividiendo el índice que nos indica en qué punto del arco estamos entre 10. Todavía nos falta definir otro cilindro que une transversalmente cada punto del arco con el punto correspondiente del arco anterior, dando lugar a una cuadricula. Esto se hace en la siguiente línea, la 43:

```
If a > 0 Then Rhino.AddCylinder arrPoint(a - 1, b), Array(x, y, z), b / 10
```

Esta vez lo que buscamos es el arco anterior y por lo tanto cuando estemos en el primer arco no tendremos que ejecutar esta función por que no habrá ningún arco anterior. Por esta razón ahora la condición es que “a” no sea 0. El resto

del código es similar al anterior y se puede interpretar de la misma manera. La última línea que requiere una explicación sería la 45:

```
arrPoint(a, b) = Array(x, y, z)
```

En esta línea se asigna al listado de puntos el valor de la posición actual. Terminada esta última función se cierran los dos procesos de repetición con un “next” cada uno y se reactiva el redibujar pantalla para que muestre los resultados.

Función de Retícula

El código que viene a continuación es una muestra de una función en que a partir de dos puntos dados por el usuario se genera una retícula de puntos en la que se pone una malla y se coloca un pilar en cada vértice. Con este sistema se pueden obtener resultados sencillos como el mostrado en la imagen adjunta. Como vemos son estructuras que se asemejan en gran medida a elementos reales, pero que todavía son muy sencillas y abstractas y carecen de definición. Sin embargo son más concretos que los ejemplos presentados el primer día que eran completamente abstractos. Con estas rutinas se puede practicar fácilmente la programación ya que tienen un código sencillo que se puede codificar con facilidad. El código de la rutina que vamos a analizar es el siguiente:

```
Sub Generar_Surf()
Dim x, y, z, LngX, LngY, giros ‘7
Dim Defx, Defy, Alt, Extremo1, Extremo2, obj
Defx = 15
DefY = 15
Extremo1 = Rhino.getpoint(“First Point: “) ‘14
Extremo2 = Rhino.getpoint(“Second Point: “)
LngX = (Extremo2(0) - Extremo1(0)) / Defy ‘18
LngY = (Extremo2(1) - Extremo1(1)) / Defx
Alt = 20
giros = 0.25
Dim arrPoints() ‘26
ReDim arrPoints((Defx + 1) * (Defy + 1) - 1) ‘28
Dim Count, a, b ‘30
Count = 0
For a = 0 To Defx ‘34
For b = 0 To Defy
x = Extremo1(0) + a * LngX ‘38
y = Extremo1(1) + b * LngY
z = Sin((a / Defx) * 6.28 * giros) * cos((b / Defy) * 6.28 * giros) * (Alt)
arrPoints(Count) = Array(x, y, z) ‘44
Count = Count + 1
Rhino.AddCylinder Array(x, y, z), array(x, y, 0), 1
Next
```

```
Next
Rhino.AddSrfControlPtGrid Array(Defx + 1, Defy + 1), arrPoints
End Sub
```

Este código es muy similar al que hemos visto en la rutina anterior, pero conceptualmente diferente. Si bien en el caso anterior el sistema usaba los índices de las funciones iterativas para obtener los ángulos correspondientes a cada arco, en este caso se utilizan para determinar la posición X e Y de los puntos de una malla. Por otra parte, el sistema que almacenará las posiciones de todos los puntos ya no será una matriz bidimensional si no unidimensional y por lo tanto, para declararla usaremos un sistema distinto al de la rutina anterior. Sin embargo la estructura general del sistema es similar. En un primer momento se recogen los datos iniciales, después se utilizan estos para operar y obtener los listados de puntos necesarios para definir la malla y una vez se han encontrado los puntos se dibujan los objetos. En la primera línea encontramos la función "Sub" que nos indica que la rutina empieza en este punto y que todo lo que viene a continuación son funciones internas de la misma. A continuación, como antes, se pone el nombre de la rutina acompañado de unos paréntesis.

En las líneas que siguen, a partir de la 7, se declaran las variables que se utilizarán en todo el proceso. Lo primero que debemos tener en cuenta es que nosotros queremos definir una malla de puntos y para ello necesitamos una resolución en X y en Y que nos indique la densidad de ésta. Las variables "DefX" y "DefY" serán las encargadas de almacenar estos valores de definición de manera que una vez declaradas se les asigna valor "15", lo que significa que la malla estará compuesta de 15 puntos en la dirección X por 15 puntos en la dirección Y. En total serán 225 puntos con sus correspondientes pilares.

```
Dim x, y, z, LngX, LngY, giros
Dim Defx, Defy, Alt, Extremo1, Extremo2, obj
Defx = 15
DefY = 15
```

En la línea 14, recogemos dos posiciones facilitadas por el usuario que serán la posición inicial y la posición final de nuestra malla. La malla comenzará y terminará según las indicaciones del usuario y al igual que en la rutina anterior los puntos recogidos por el mouse se almacenarán en dos variables destinadas a tal efecto (Extremo1 y Extremo2).

```
Extremo1 = Rhino.getpoint("First Point: ")
Extremo2 = Rhino.getpoint("Second Point: ")
```

Este proceso es exactamente igual que en la rutina anterior: asigna mediante el operador "=" un listado de coordenadas facilitadas por la función "Getpoint()" que al ser una función interna de Rhino va precedida del prefijo "Rhino" y un ".".

La cadena de texto que encontramos entre paréntesis corresponde al mensaje que leerá el usuario cuando ejecute la función. Para generar la matriz necesitaremos también saber cuánto medirá la distancia entre los puntos de la malla en la dirección X e Y y así situarlos a las distancias correspondientes. Estas medidas dependerán de las posiciones que haya facilitado el usuario. La formulación (línea 18 y siguientes), queda de la siguiente manera:

```
LngX = (Extremo2(0) - Extremo1(0)) / Defy '18
LngY = (Extremo2(1) - Extremo1(1)) / Defx
Alt = 20
giros = 0.25
```

Las variables LngX y LngY almacenan el desplazamiento en cada sentido (x

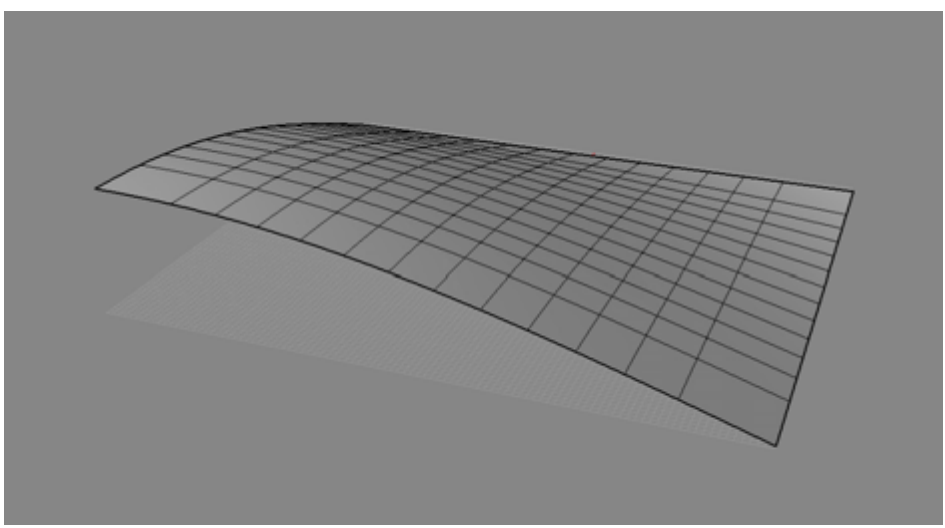


Figura 12. Ejemplo de superficie formada por los puntos de control

e y) que tendrán los segmentos de la matriz. Estos se obtienen midiendo la diferencia de distancias entre los dos puntos recogidos por el usuario, y dividiéndolos por el número de segmentos que tendremos a la malla. Por ejemplo, si tenemos dos posiciones en el espacio (10, 10) y (-5, 0) en primer lugar calculamos el desplazamiento total en dirección X como la diferencia entre la X final y la inicial ((-5) - 10 = -15). Si nuestra malla tuviera una definición de 10 puntos por lado, el desplazamiento de cada punto en la dirección X será el desplazamiento total dividido por el número de puntos, o sea, (-15 / 10 = 1,5). Si repetimos esta operación para la Y tenemos también el desplazamiento en la dirección Y por cada punto, de manera que el punto inicial será igual al Extrem1 y el punto final

será igual al Extrem2, los puntos intermedios irán desplazándose del Extrem1 al Extrem2 a saltos iguales a los desplazamientos obtenidos en LngX y LngY.

La variable “Alt” y la variable “giros” son valores iniciales que afectan a la altura media y a la forma general de la malla. A continuación encontramos que el código declara una variable llamada “arrPoints” con dos paréntesis al final y después le aplica una función “ReDim” para declarar sus dimensiones. Cualquier variable de matriz debe ser declarada inicialmente de manera convencional con la palabra clave “Dim” y dos paréntesis al final para poder más tarde aplicarle una función “ReDim” para darle la dimensión que nos interese.

```
Dim arrPoints() ‘26
ReDim arrPoints((Defx + 1) * (Defy + 1) - 1) ‘28
```

En la línea 28 lo que hacemos es encontrar el número total de puntos que generaremos y asignarles como extensión de la matriz unidimensional. (Fijémonos que a pesar de ser una malla con X e Y utilizo una matriz unidimensional y no bidimensional). La operación con la que obtengo el número total de elementos no se obtiene directamente multiplicando el número de elementos en X para los elementos en Y. Si recordamos que los contadores empiezan en 0 entenderemos por que, al multiplicar el número de elementos en X por el número en Y les sumo 1 a cada uno. Si nosotros decidimos que una matriz debe tener 15 puntos por lado debemos tener en cuenta que el ordenador empezará a contar desde 0 y por tanto en realidad habrá 16a pasos. Así pues el número real de elementos si la DefX vale 15 y la DefY también vale 15 será 16 x 16. Pero aún debemos

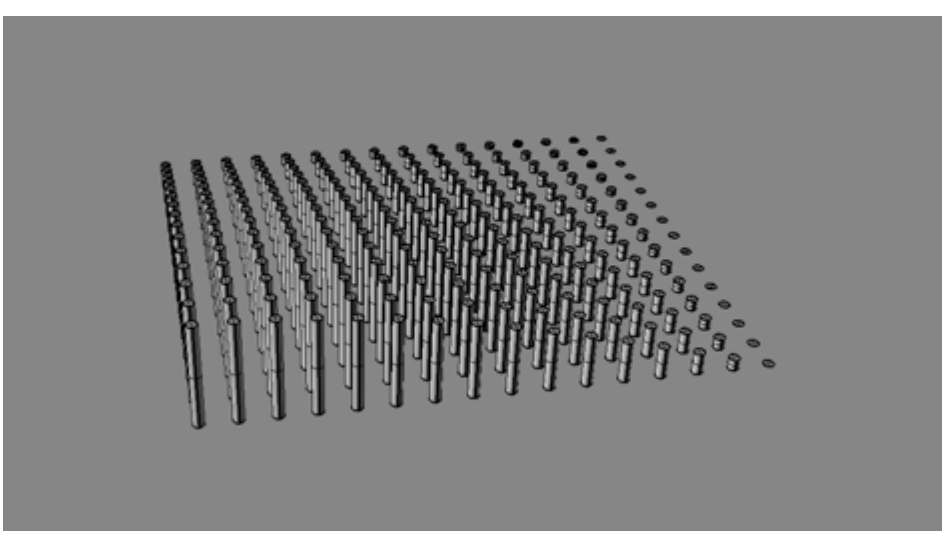


Figura 13. Ejemplo de los pilares formados por el conjunto de puntos

tener en cuenta que estamos declarando uno más ya que se empieza a contar desde 0 y por tanto hay que restarle 1.

```
Dim Count, a, b '30
Count = 0
For a = 0 To Defx '34
For b = 0 To Defy
```

Al igual que en el caso anterior, en las líneas 34 y siguiente, aplicaré dos funciones de repetición para asignar el valor de X, Y y Z en cada punto, pero esta vez, además de generar un índice "a" y un índice "b" también generamos una variable llamada "Count" que será un contador de la matriz unidimensional "arrPoints". A diferencia del caso anterior, la matriz que almacena los puntos "arrPoints" es unidimensional, esto quiere decir que no puedo usar "a" y "b" para indicar una posición en este listado. Por eso es necesario generar una nueva variable llamada "Count" que irá contando cuántos elementos han sido generados para poder indicar en qué posición nos encontramos de la matriz lineal. Fuera de esta diferencia el resto de la estructura es igual. Una vez hemos hecho todo esto pasamos a asignar los valores de X, Y y Z de cada punto. Esto se resuelve con las siguientes funciones:

```
x = Extremo1(0) + a * LngX '38
y = Extremo1(1) + b * LngY
z = Sin((a / Defx) * 6.28 * giros) * cos((b / Defy) * 6.28 * giros) * (Alt)
```

La función para asignar valor a X y Y es extremadamente sencilla. En primer lugar asignamos el valor X o Y del extremo superior, el valor inicial (Extremo1 (0) y Extremo1 (1)). A continuación le sumamos el desplazamiento en X y en Y que antes hemos calculado por "a" o "b" según su posición dentro de la matriz. De esta manera si bien el punto inicial será igual al punto inicial recogido por el usuario, el punto final será igual al segundo punto recogido por el usuario. La definición de Z será más compleja de lo que lo era en la función anterior. El valor de Z variará según su posición dentro de la matriz de puntos, para asignar un valor de Z según la posición en la matriz uso los contadores "a" y "b" de manera que a partir de ellos pueda obtener una altura . Al igual que en la rutina anterior se utiliza una relación entre "a" y "DefX" que dará valores entre 0 y 1 según si a es igual a su valor inicial (0) o a su valor final (DefX). Estos se multiplicarán por un ángulo de 360 º en radianes con el que se hará la operación seno. La misma operación se utiliza con la coordenada Y para obtener el coseno del correspondiente ángulo y aplicarlo también a la Z. Una vez obtenidos estos dos valores se multiplican entre ellos y con el valor de la altura media de manera que obtenemos el valor de Z. De esta manera si bien la posición X y la Y corresponderán a una cuadricula, su altura corresponderá a una función sinusoidal que depende

de su posición en la matriz.

Hecho esto tenemos la posición de cada punto de la malla, y sólo nos queda generar el pilar, y generar la malla que pasa por todos los puntos. Estas son funciones internas de Rhino que están muy automatizadas y que nos ocuparán pocas líneas de código:

```
arrPoints(Count) = Array(x, y, z) '44
Count = Count + 1
algomad 2010 88
Rhino.AddCylinder Array(x, y, z), array(x, y, 0), 1
Next
Next
Rhino.AddSrfControlPtGrid Array(Defx + 1, Defy + 1), arrPoints
```

Lo primero que vemos en la línea 44 es como se almacena el paquete con las posiciones X, Y y Z en la matriz “arrPoints”. Fijémonos en que al contador “Count” se le suma 1 justo después de esta operación. Volvemos a encontrar la función “AddCylinder punt1, Punt2, radio” que ya he explicado en la rutina anterior, como ya he dicho el punto inicial es “array (X, Y, Z)”, o sea el punto de la malla, pero el punto final es “array (X, Y, 0)” o sea un punto que se corresponde con la X y la Y del punto de la malla pero esta altura del suelo, así obtenemos un “pilar” que va del punto de la malla hasta el suelo en vertical. Al estar incluido dentro del proceso de iteraciones se generará un pilar por cada punto.

En las últimas líneas cerramos las dos funciones de repetición o iteración y generamos la malla a partir de una función interna de Rhino llamada “AddSrfControlPtGrid()”. Esta función necesita por un lado dos valores empaquetados en una función “Array” que indiquen cuantos puntos en la dirección X y en cuanto en la dirección Y tiene la malla. Recordemos que esta cifra la tenemos almacenada en las variables “Defx” y “Defy” pero hemos de sumarle 1 debido a

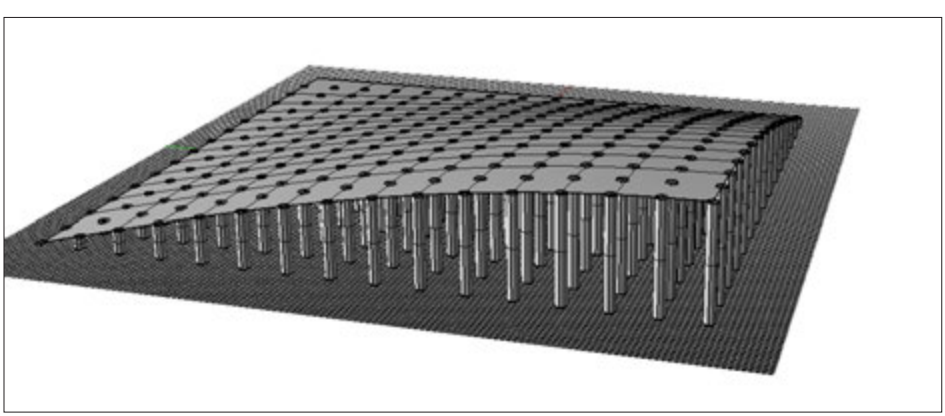


Figura 14. Resultado del Script Función de Retícula

las razones que antes se han expuesto con los contadores cuando empiezan a contar desde 0. Después de haber facilitado estos dos datos, nos pide el listado de puntos en una lista unidimensional que hemos almacenado en la variable “arrPoints”.

Fijémonos que en esta rutina no hemos utilizado las líneas de código “Rhino. EnableRedraw ()” que en cambio sí que estaban presentes en la rutina anterior.

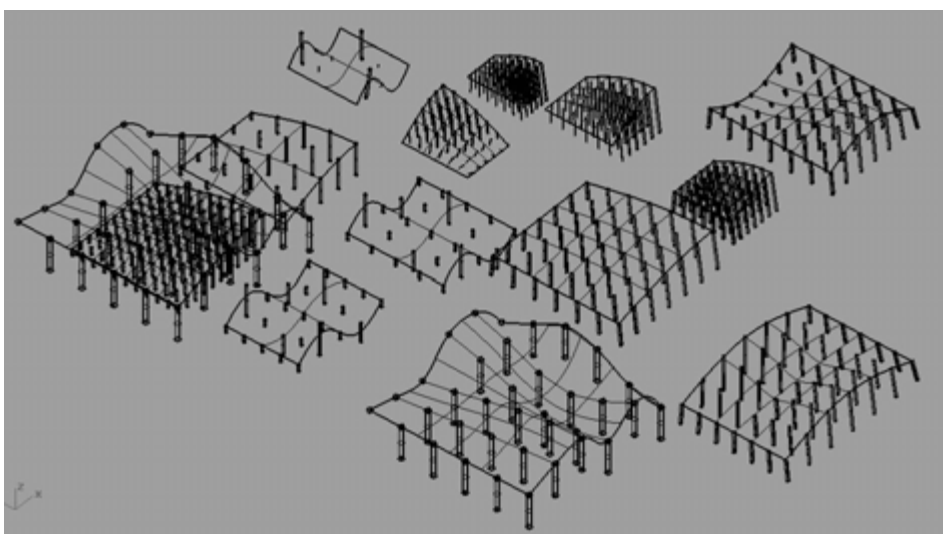


Figura 15 Variaciones sobre el tema de Carles Ferran

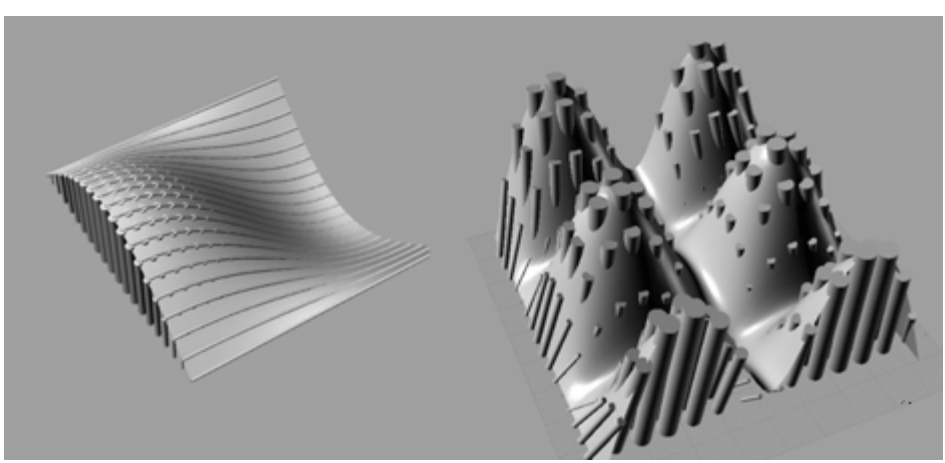


Figura 16.Variaciones sobre el tema de la superficie generadas por Javier González durante las clases

Esto permite que mientras la rutina se ejecuta podemos ir viendo todo lo que se va dibujando. El problema es que esto ralentiza mucho el proceso, por lo que si quisiéramos ir que el programa dibujase más rápido sólo deberíamos poner “Rhino. EnableRedraw (false)” antes de las iteraciones y “Rhino.EnableRedraw (true)” justo antes de “End Sub “.

Partiendo de esta rutina se generaron múltiples variantes del mismo tema a partir de las experiencias de cada alumno. El objetivo principal era romper la rigidez de la forma para entender que se trata de un proceso y no de un objeto acabado. Así se entiende mejor como usar la programación como herramienta de trabajo durante el modelado y de que manera el concepto de diseño, modelo o forma cambian sustancialmente a partir de esta reflexión.

TRABAJOS

Este trabajo de José Ignacio Gálvez, fue capaz de generar geometrías interesantes a partir de una práctica relativamente sencilla del primer dia. A pesar de que los recursos eran muy limitados se supo jugar con los criterios de programación para generar procesos muy diferentes cambiando aspectos claves de la formula. La imagen se compone de dos mitades, una superior y una inferior. En la mitad superior se modificaron los exponentes de las fórmulas de manera que las alturas aumentaban considerablemente. Lo más importante es que también las proporciones cambiaron de manera se generaron nuevas relaciones geométricas entre los elementos. Este planteamiento es capaz de jugar con el proceso en sí de forma libre, y por tanto se está adquiriendo la idea de que en temas de programación lo más importante es dominar el proceso y no los objetos concretos. En la mitad inferior se encuentra un nuevo ejemplo de este planteamiento al añadir una nueva condición puntual pero clave. Muchas veces, cuando hacemos una forma libre en un sistema digital, lo primero que sorprende es que no se respete el nivel del suelo de manera que las cosas “flotan”. Añadiendo una condición tan sencilla como decir que cuando la altura de un punto sea negativa, que cambie el signo, este efecto desaparece e inmediatamente se forma una estructura más estable visualmente.

Otros trabajos tenían problemas mucho más concretos y trabajaron el código hasta solucionarlos. En el ejemplo siguiente, un trabajo de Francisco Javier Gonzalez, puede parecer muy similar a la función radial descrita anteriormente, pero tiene una diferencia fundamental y es que todas las barras están entrecruzadas en diagonal y no de manera ortogonal. Este cambio no fue tan inmediato como podría parecer ya que ha sido necesario cambiar dos cifras y añadir algunas condiciones e interpretar el código para poderlo modificar correctamente. Normalmente esto representa un grado de dificultad considerable, especialmente si no tienes práctica en programación. Pero en este caso, al tener un

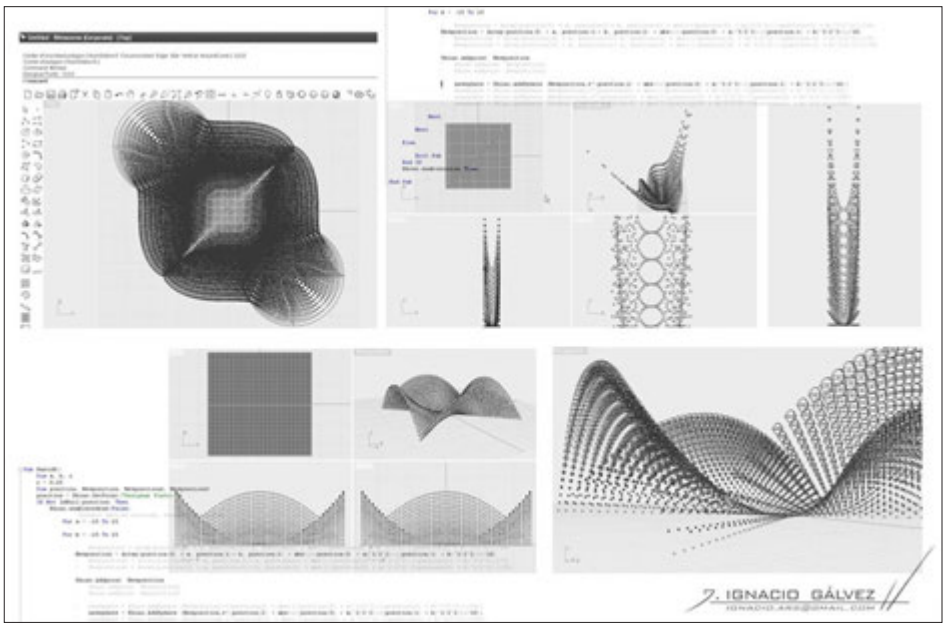


Figura 17. Trabajo de Ignacio Gálvez

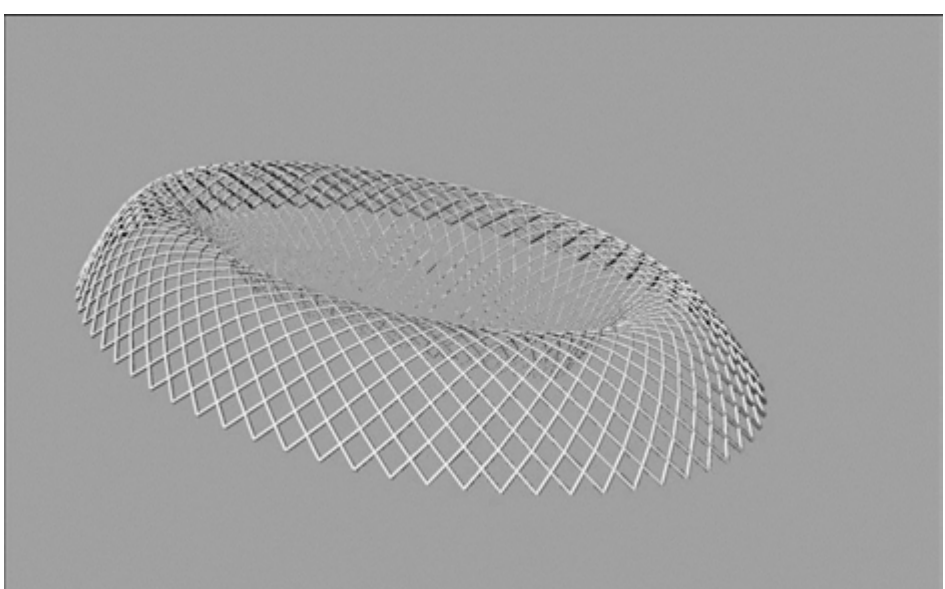


Figura 18. Trabajo Francisco Javier González

planteamiento claro de lo que se quería alcanzar, se fue capaz de superar las dificultades y cambiar el código de manera que se solucionara el problema. Lo más importante fue que una vez que se planteó un objetivo claro, las dificultades fueron mucho más fáciles de superar.

CONCLUSIONES

Los cursos de RhinoScript tenían la intención de hacer que el alumno "jugase" con diferentes rutinas. Estas pretendían generar geometrías más o menos complejas con códigos muy sencillos. Aunque el código sea corto, ciertamente la base conceptual puede ser muy compleja y eso ha sido un problema importante para facilitar el acceso a la programación del público profano. Por eso intenté que las rutinas tuvieran el máximo de posibilidades con el mínimo de líneas de código y fueran muy explicativas de los conceptos con los que se trabaja en programación. Muchos alumnos respondieron positivamente a este planteamiento, experimentando y asumiendo el sistema de programación como una herramienta de gran potencia pero de uso muy abstracto. La idea de que a todos les queda es que programando se pueden obtener todo tipo de resultados, pero hay que ser capaz de romper el hielo con estos sistemas y adentrarse en la lógica de la programación.

Programar es como aprender una nueva lengua, frecuentar su uso es la mejor manera para acabar hablando con fluidez; esto quiere decir que si se pudiera programar cada día un poco se podría acabar haciendo todo tipo de aplicaciones. Obviamente en estos dos días, a pesar de dar las bases necesarias para programar, la intención primera era demostrar las potencialidades que estos sistemas ofrecen que son realmente muy extensas.

A modo de reflexión debo decir que el futuro se está moviendo en la dirección de variar la relación que mantenemos con los sistemas digitales y precisamente la programación es la forma en que un humano se comunica naturalmente con un sistema digital, del mismo modo que el lápiz es el instrumento natural con el que un humano dibuja sobre un papel. Programar es difícil debido al alto grado de abstracción que implica y es por eso que se están desarrollando nuevas maneras de intentar acercar este campo a los usuarios, Rhinoscript ha sido el primer paso, Grasshopper ha sido el siguiente, pero aún falta conseguir un sistema más completo y más concreto para comunicar con los ordenadores. El objetivo final es aprovechar completamente la capacidad de computación que en gran medida sigue limitada por la pobre comunicación que tenemos con estos sistemas .. La pregunta que esta al aire ahora mismo es ¿cómo debe ser nuestra relación con los sistemas digitales? y cómo podríamos relacionarnos con el ordenador de manera que retengamos el grado de control de un sistema de programación con la facilidad de uso de un CAD?

La mayor dificultad se encuentra en que cuanto más abstracto es un lenguaje, más capacidad de generar procesos concretos puede ofrecer. Esta relación hace que un sistema intuitivo limite en gran medida la capacidad de acción del usuario. Quizá el conflicto principal está en que las personas agrupamos los elementos conceptualmente de manera muy distinta a como lo hace la máquina debido a nuestra extraordinaria capacidad de interpretación, y cuando queremos que la máquina mantenga nuestros conceptos nos damos cuenta que tenemos que limitar en gran medida lo que podemos hacer con ellos. Es pues fundamental que analicemos nuestro propio funcionamiento interno, la forma en que interpretamos y modificamos nuestras propias ideas, para aproximar los sistemas de programación a los propios procesos mentales y hacerlos intuitivos y eficientes al mismo tiempo. En este aspecto es necesario aprender muchas cosas de uno mismo que, sin fijarse, pasarían desapercibidas y que son las que marcan la diferencia entre un buen o un mal planteamiento informático.

Taller 4

Processing

Marcel Bilurbina

QUÉ ES PROCESSING

Processing es un lenguaje de programación y un entorno de desarrollo Open Source, es decir, de código abierto. Fue creado por Casey Reas y Ben Fry en el MIT para mostrar los fundamentos de programación en un contexto visual y está pensado para artistas, diseñadores y programadores que quieran expresarse con el lenguaje digital.

El programa está basado en el lenguaje de programación Java, con lo que hereda todas sus funcionalidades, convirtiéndose en una herramienta poderosa a la hora de crear diferente tipos de proyectos, aplicaciones locales o para la web.

Las principales ventajas de Processing son las siguientes:

— Permite introducirse al mundo de la programación de forma fácil y amena y a la vez tener una herramienta suficientemente potente para desarrollar proyectos complejos.
— No hay que compilar de forma externa y explícita. Se hace automaticamente al ejecutar nuestra aplicación.
— Es un proyecto Open Source en continuo desarrollo y crecimiento con un claro comportamiento emergente. Cualquier usuario se puede convertir en desarrollador y hacer crecer el proyecto.
— La documentación y la ayuda es abierta y excelente.

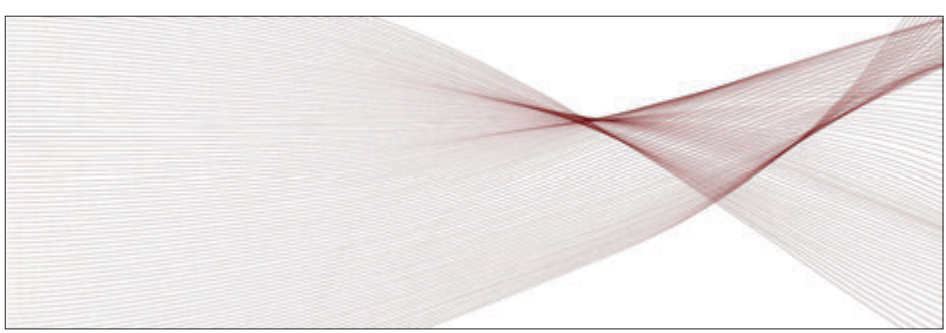


Figura 1

TUTORIALES DE PROCESSING

Processing: A Programming Handbook for Visual Designers and Artists
Casey Reas and Ben Fry (Foreword by John Maeda).
Published 24 August 2007, MIT Press. 736 pages. Hardcover.

Learning Processing: A Beginner's Guide to Programming Images, Animation, and Interaction
Daniel Shiffman.
Published August 2008, Morgan Kaufmann. 450 pages. Paperback

Visualizing Data
Ben Fry.
Published December 2007, O'Reilly. 384 pages. Paperback

INSTALACION

Processing se puede descargar desde la página oficial: http://processing.org/download/.

Para los usuarios de Windows es recomendable instalar la versión completa con Java.

NO instalar la versión Windows(without Java).

Se puede instalar en cualquier carpeta del ordenador. Simplemente hay que descomprimir el fichero descargado. Para ejecutar Processing hay que clicar en el archivo processing.exe ubicado en el directorio raiz.

INTERFICIE DE PROCESSING

La interficie de Processing basicamente consta de los siguientes elementos:

1. MENU
2. EDITOR DE CODIGO
3. CONSOLA
4. VENTANA GRAFICA

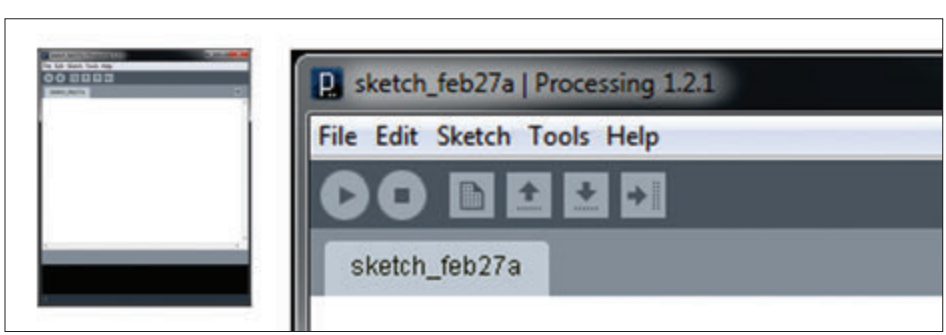


Figura 2

MENU:

Encontramos los siguientes elementos:

RUN	Compila el código y lo ejecuta en la ventana gráfica
STOP	Para el programa que se está ejecutando en la ventana gráfica
NEW	Abre un archivo nuevo
OPEN	Abre un archivo existente
SAVE	Guarda un archivo
EXPORT	Genera un archivo .html con el sketch ejecutable mediante cualquier navegador
EXPORT APPLICATION	Genera un archivo ejecutable .exe ejecutable fuera de la interficie de Processing

EDITOR DE CODIGO:

Es el lugar donde escribiremos nuestro código.

CONSOLA:

Basicamente nos sirve para 2 cosas:

1 Es el sitio donde Processing escribe los mensajes de error cuando no puede compilar
2 Permite al usuario imprimir valores de variables y así obtener detalles sobre la ejecucion del código

VENTANA GRAFICA:

Es el sitio donde visualizaremos nuestro código una vez compilado y ejecutado. Se trata de una ventana externa que solamente es visible cuando se ejecuta el código.

CARACTERISTICAS BASICAS

REFERENCIA DE PROCESSING

Es el listado de todas la funciones y métodos de Porcessing. Permite acceder a ejemplos sobre cada una de las funciones existentes (http://processing.org/reference/).

COMENTARIOS

Permite escribir texto en editor sin que Processing lo tenga en cuenta a la hora de compilarlo y ejecutarlo. Se utiliza para hacer comentarios sobre el código. Es muy importante comentar el código bien para poder encontrar errores o rescatar viejos códigos al cabo del tiempo y poderlos entender.

Hay de 2 tipos:

. Comentarios simples o de 1 línea

```
// Esto es un comentario simple y no se compila
```

. Comentarios de 2 líneas o más

```
/*
Esto es un comentario
de 2 lineas y tampoco se compila
*/
```

PUNTO Y COMA “ ; ”

Al final de casi todas las líneas de código deberemos escribir un punto y coma “;” Hay algunas excepciones que veremos más adelante. Se trata de cambios de línea que no representan un final de “sentencia”

```
rect(100, 100, 10, 10) ;
```

MAYUSCULAS Y MINUSCULAS

Processing es sensible a minúsculas y mayúsculas. No es la misma variable

```
int posicionX ;
```

que la variable

```
int PosicionX ;
```

HELLO WORLD!

HelloWorld! Es el primer programa que se realiza habitualmente cuando se aprende un lenguaje de programación. Se trata simplemente de escribir en la consola el texto “Hello World!”.

Así lo haremos con Processing.

print() println()

Para imprimir en la consola usaremos la función **print()** o **println()**. La segunda indica que después de imprimir el texto en la consola cambia de línea.

```
print(“texto  a imprimir”)
```

Así, el código del “hello world!” en Processing será

```
print(“hello world !”)
```

Una vez compilado y ejecutado el resultado se verá en la consola.

```
HelloWorld! Grafico
```

También podemos realizar un helloWorld gráfico, pues ese es el lenguaje que nos interesa de Processing.

Como se trata de hacer un ejercicio lo más sencillo posible dibujaremos un rectangulo escribiendo la siguiente línea de código.

```
rect( 10, 10, 80, 80);
```

El resultado será un rectangulo de 80 x 80 con origen en el punto 10,10

GRID. EJES DE COORDENADAS

Pixels y ejes de coordenadas

La ventana gráfica, donde visualizamos nuestro código, está formada por una retícula ortogonal de pixels.

El origen de coordenadas (0,0) se encuentra en el extremo superior izquierdo. El eje X es horizontal y el eje Y vertical.

Cada unidad corresponde a un pixel.

Más explicaciones. Ver: http://processing.org/learning/drawing/

FUNCIONES BASICAS DE DIBUJO

size(x,y)

Permite fijar el tamaño de la ventana grafica de nuestro applet.
Las unidades son pixeles.

La primera coordenada (x) sirve para indicar el tamaño en direccion del eje X, horizontal.
La segunda variable (y) sirve para indicar el tamaño en la direccion del eje Y, vertical.

El siguiente codigo crea una ventana de 400 pixels en horizontal y 300 pixels en vertical:

```
size ( 400, 300 );
```

line(x1, y1, x2, y2)

Dibuja una línea con origen a las coordenada x1, y1 y final a la coordenada x2,y2.
En el siguiente ejemplo el punto origen (x1,y1) es (0,0) y el final (x2,y2) es (100,100)

```
line ( 0, 0, 100, 100);
```

rect(x1, y1, x2, y2)

Dibuja un rectangulo. Se define mediante 4 coordenadas correspondientes al

origen y al tamaño del rectángulo.
En el siguiente ejemplo el punto origen (x1,y1) es (10,10) y el tamaño es (80,80).

```
rect( 10, 10, 80, 80);
```

ellipse(x1, y1, rH, rV)

Dibuja una elipse. Se define mediante 4 coordenadas correspondientes al centro de la elipse (x1,y1), el radio en direccion X (rX) y el radio en dirección Y (rY) . Cuando rX sea igual a rY se dibujará un círculo.
En el siguiente ejemplo el punto origen de la elipse (x1,y1) es (50,50) y el radio X e Y es 10.

```
ellipse( 50, 50, 10, 10);
```

Otras funciones

Otras funciones basicas de dibujo son:

```
arc()
point()
quad()
triangle()
```

Ver referencia de Processing en http://processing.org/reference/

FUNCIONES BASICAS DE COLOR

color(R,G,B,alpha)

El color se especifica mediante 3 variables que nos dan el color (R,G,B) y otra que nos da la transparencia (alpha). Cada una de estas variables puede tener un valor de 0 a 255, correspondientes a 8 bits de informacion: 2^8 = 256

Por consiguiente el color se define como una cadena de bits tal como esta:
RRRRRRRRGGGGGGGGBBBBBBBBAAAAAAAA

R es el valor de rojo, G el valor de verde y B el valor de azul.
A corresponde a la transparencia

Programando, podemos especificar el color de distintas formas:

. Con 4 variables. Indicando el valor de R, G, B y alpha

```
color( 255, 0, 0, 126);          // color rojo y 50% de transparencia
```

. Con 3 variables. Indicando el valor de R, G, B. Se considera alpha 0, es decir, sin transparencia.

```
color( 255, 0, 0);               // color rojo y opaco
```

. Con 2 variables. Equivale a trabajar en escala de grises . 0 corresponde a negro y 255 a blanco

```
color( 0, 126);                  // color negro y 50% de transparencia
```

. Con 1 variable. Equivale a trabajar en escala de grises sin transparencia . 0 corresponde a negro y 255 a blanco

```
color( 255);                     // color blanco
```

background(color)

Permite definir el color de fondo de la ventana

```
background( 0,255,0 );           // fondo ventana de color verde

background( 0 );                 // fondo ventana de color negro
```

fill(color) noFill()

Permite definir el color de relleno de una forma
Afecta a las funciones de dibujo que se llamen después.

```
fill( 0 );                       // rellenar de color negro
rect(0,0,10,100);                // el rectangulo se dibujará relleno de negro
```

Sirve para indicar NO relleno. Transparente

```
noFill( );                       // sirve para indicar NO relleno
rect(0,0,10,100);                // el rectangulo se dibujará sin relleno
```

stroke(color) noStroke()

Permite definir el color del contorno de una forma.
Afecta a las funciones de dibujo que se llamen después.

```
stroke( 255 );          // contorno de color negro
rect(0,0,10,100);       // el rectangulo se dibujará  con líneas de color blanco
```

Sirve para indicar que NO dibuje la línea de contorno

```
noSstroke( );           // sirve para indicar que NO se dibujen líneas de contorno
```

VARIABLES

Son elementos básicos de cualquier lenguaje de programación.
Sirven para almacenar valores que pueden cambiar o no durante la ejecución del programa (posiciones, colores, tamaños, etc). Hay de distintos tipos y tamaño.

Para usar una variable hay que seguir los siguientes pasos:

. **Declaración** de la variable
. **Asignación** de un valor inicial

Se declaran, en general, al principio del programa.

Para declarar una variable hay que definir el tipo de variable y su nombre:

```
tipoDeVariable  nombreDeVariable ;
```

Tipos de variables:

boolean	Boleanos . Variables con 2 posibles valores “true” o “false”. Verdadero o falso
int	Enteros . Valores posibles: 1,2, 182, - 9283, etc
float	Decimales . Valores posibles: 1.02 , - 326.045, etc
char	Letras . Valores posibles: ‘A’, ‘d’
String	Palabras . Valores posibles: “processing”

Ejemplo de declaración de variables:

```
int posicionX;
float velocidad;
boolean dentro;
```

Asignación de valores a variables si se han declarado previamente:

```
posicionX = 10;
velocidad =  0.5;
dentro = false;
```

También podemos declarar y asignar un valor inicial directamente:

```
int posicionY = 20;
```

Variables del sistema

width height

Son variables que no tenemos que declarar porque Processing las reconoce por defecto.
Cuando asignamos un tamaño de ventana mediante la función **size(x,y)** Processing reconoce automaticamente las variables **width** y **height** de manera que **: width = x** y **height = y**

Ejemplo:
Dibuja una elipse con centro en el centro de la ventana y de radio 10

```
size(400, 300);
ellipse( width/2, height/2, 10,10);          // elipse con centro 200,150 y radio 10
```

mouseX mouseY

Su uso permite conocer la posición del mouse dentro de la ventana. Nos devuelve la posición en forma de entero.

Ejemplo:

Dibuja una elipse con centro en la posición del mouse

```
int posicionX;
int posicionY;
```

```
posicionX = mouseX;
posicionY = mouseY;

ellipse(posicionX, posicionY, 5, 5);
```

BLOQUES DE PROGRAMACION

La programación en Processing se estructura en distintos bloques:

Bloque 1 **Declaración de variables genéricas, objetos y librerías**

Bloque 2 **void setup() { }**

Bloque 3 **void draw () { }**

Bloque 4 **Funciones y clases propias**, definidas por el usuario

BLOQUE 1:

Esta parte de código se lee y ejecuta 1 única vez.
Se declaran las variables que se usarán posteriormente. También se pueden inicializar las variables, es decir, darles su valor inicial.

```
int posicionX;
int posiciony;
int radio = 5;
```

BLOQUE 2:

```
void setup( ){ }
```

Esta parte de código se lee y ejecuta 1 única vez.
Se definen los valores que afectan al carácter general de nuestro programa como **size()** y se pueden inicializar las variables.

En el siguiente ejemplo se dibuja 1 ÚNICA VEZ un rectangulo de color blanco sobre fondo negro, se inicializa la variable posicionX con el valor 0 y se determina el tamaño de la ventana (400x300)

```
void setup(){

    size(400, 300);
    posicionX = 0;
```

```
    background(0);          // fondo ventana de color negro
    fill(255);              // relleno color blanco
    rect(10,10,80,80);      // dibuja un rectangulo

}
```

BLOQUE 3:

```
void draw( ){ }
```

Esta parte de código se lee y ejecuta **CONTINUAMENTE**.
Aquí se escribirá todo el código referente a acciones que deban realizarse continuamente y mientras se ejecute el programa.
Permite trabajar con el factor tiempo, haciendo modificaciones de variables por cada loop que se ejecuta.

Ejemplo:
En el siguiente ejemplo se dibuja continuamente un rectangulo de color blanco sobre fondo negro

```
void draw(){

    background(0);          // fondo ventana de color negro
    fill(255);              // relleno color blanco
    rect(10,10,80,80);      // dibuja un rectangulo

}
```

BLOQUE 4:

Se definirán otras estructuras como funciones y clases propias que se verán más adelante.

OPERADORES MATEMATICOS

Se pueden realizar simplemente utilizando los operadores necesarios.

+ - * /

suma, resta, producto, division, ...

Ejemplos:

```
int x1 = 2;
int x2 = 10;
int suma;
int producto;

suma = x1 + x2;
producto = x1 * x2;
println(suma);
println(producto);
```

Ver otros operadores . http://processing.org/reference/

++ – += -=

Se trata de abreviaciones muy utlizadas para realizar **contadores**. Significan lo siguiente:

Contador positivo y negativo

x++ → x=x+1
x-- → x=x-1
x+=3 → x=x+3
x -=3 → x=x -3

Ejemplo de contador:

```
int x= 0;
void setup(){
size(200,200);
}
void draw(){
println(x);     // imprime el valor de x
x++             // siginifica x=x+1. Por cada loop incremente al valor de x en 1.
}
```

Por cada loop incrementa el valor de x en una unidad.
Si imprimimos el valor en cada loop nos dará el siguiente resultado:

1er loop	x=0
2º loop	x=1
3er loop	x=2
4º loop	x=3
… loop	x=…

```
int x = 0;
void draw(){
println(x);     // imprime el valor de x
x++;            // significa x = x + 1. Por cada loop incrmenta el valor de x en 1
}
```

Número aleatorio . Random

float random(min, max)

Sirve para generar números aleatorios entre 2 valores: el minimo (min) y el máximo (max)
Devuelve un número de tipo **float**.

Ejemplo:
Nos da valores entre 2 y 10.

```
float x;
x= random(2, 10)
```

También se puede utilizar pasando una sola variable. En este caso nos dará valores entre 0 y el valor que le pasamos.

Ejemplo:
Nos da valores entre 0 y 10

```
float x;
x= random(10)
```

Módulo

%

Nos permite crear un bucle de numeros que vayan de 0 al valor que queramos de forma infinita

Ejemplo:
El valor de X se incrementará de 0 a **width** de uno en uno y cuando sea igual a **width** (el valor del ancho de nuestra ventana) volverá a 0. Se dibuja una elipse con tamaño = a X para visualizarlo.

```
int x=0;                    // declaramos X . La inicializamos con valor = 0
```

```
void setup(){
    size(400,400);                    // tamaño de la ventana
}

void draw(){

    x++;                              // podria ser x++
    x=x%width;
    ellipse (width/2, width/2, x, x);

}
```

MOVIMIENTO

Mediante el uso de contadores podremos facilmente controlar el movimiento. Para ello se crearan 2 variables: una que recuerde la posición en la que está nuestro elemento que movemos y la otra que nos indique el valor de desplazamiento por cada loop.

Ejemplo:

```
int posicionX = 0;                    // declaramos posicionX . La inicializamos con valor = 0
int step = 2;                         // declaramos la variable step
```

Si por cada loop queremos que la posiciónX se incremente en el valor de step haremos lo siguiente:

```
void draw(){

    x = x + step;
    ellipse (x, 100, 10, 10);       // dibujamos elipse para comprobarlo

}
```

ESTRUCTURAS DE CONTROL

Nos sirven para comparar valores y tomar decisiones.

Operadores condicionales y booleanos

x == y	x **igual** a y
x != y	x **no igual** a y

x < y	x **menor** que y
x > y	x **mayor** que y
x <= y	x **menor o igual** que y
x >= y	x **mayor o igual** que y
&&	(**y**)
| |	(**o**)
!	(**no**)

Estructuras condicionales

if if…else

```
if ( CONDICION ) {                    // si se cumple la condición
    ACCION A REALIZAR;
}
else{
ACCION A REALIZAR;                    // si NO se cumple la condición
}
```

Ejemplo:

Cambiar el signo de la variable **V** si la variable **posicionX** es mayor al tamaño horizaontal de la pantalla.
Este ejemplo se usará, por ejemplo para controlar que un objeto no pase de un límite determinado

```
if ( posicionX > width ) {        // si se cumple la condición
    v = -v;                       // invertir signo de v
}
else{                             // si NO se cumple la condición
v = v;                            // seguir igual
}
```

Otras estructuras condicionales:

switch case
for
while
do...while
break
continue

EJERCICIOS

1. Formas basicas y variables del sistema

Objetivo: Copiar el dibujo 1 controlando el límite blanco/negro con **mouseX**
Objetivo: Copiar el dibujo controlando la posicion Y de la línea con **mouseY**
Pistas: mouseX, mouseY

Figura 3-4

2. Condicionales y operadores condicionales

Objetivo: Hacer rebotar un rectangulo con los límites de la pantalla y que no salga de ella en ningún momento

Pistas: variables:

```
int x        // posición X
int y        // posición Y
int vx       // velocidad en la direccion X
int vy       // velocidad en la direccion Y
int sizeX    // tamaño rectangulo en el eje X
int sizeY    // tamaño rectangulo en el eje Y
```

variables del sistema:

width, height

ecuaciones:

```
x = x +vx
y = y +vy
```

ESTRUCTURAS DE REPETICION

FOR LOOP

Las estructuras de repetición aprovechan la capacidad de cálculo de los ordenadores.
Se utilizan para ejecutar procesos repetitivos. Nos permiten realizar rutinas economizando código.

La estructura de repetición más habitual es la siguiente:

```
for ( int i=0; i<max ; i++ )  {  }
```

Nos permite repetir una acción un cierto número de veces mientras se cumpla una condición que definiremos nosotros. Se puede entender como un loop interno que permite ejecutar una acción varias veces.
El esquema de esta estructura es el siguiente:

```
for ( inicio ; test o condicion ; actualizacion) {

    CODIGO A REPETIR MIENTRAS SE CUMPLA LA CONDICION

}
```

La estructura se inicializa con la instrucción **for**
Entre paréntesis encontramos 3 elementos distintos:

Inicio

Se declara la variable a utilizar en la condición y se inicializa con un valor

```
for ( int i = 0 ; test o condicion ; actualizacion) {    }
```

Se declara la variable i y se inicializa con valor 0

Test o condicion

Condicion o test que se debe cumplir para ir generando repeticiones

```
for ( int i = 0 ; i < 10 ; actualizacion) { }
```

Equivale a decir “si i es menor a 10 sigue haciendo loops”

Actualizacion

Valor que la variable adopta por cada nuevo **loop** . Siempre se trata de algún tipo de contador.

```
for ( int i = 0 ; i < 10 ; i ++) {          }
```

Equivale a decir “por cada loop incrementa el valor de i en 1” . Podríamos escribir también i=i+1 o podríamos sumar de 3 en 3 escribiendo i=i+3

Entre llaves se define la accion a repetir mientras la variable declarada en primer lugar, y que se actualiza en cada loop, cumpla la condicion definida.

Ejemplo:

```
for ( int i=0 ; i < 3 ; i++ ){
    println(“valor de i : ” + i );
}
```

Realiza la acción “imprimir el valor de i” 3 veces --> realiza 3 loops
Por cada loop se incrementa el valor de i en 1

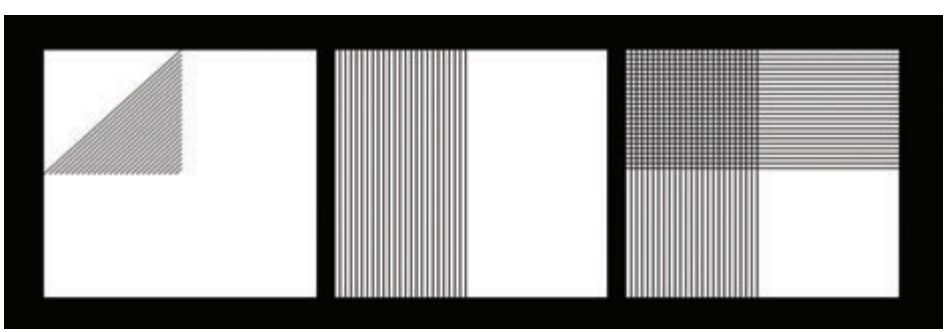


Figura 5

En el	1er loop	valor de	**i = 0**
En el	2o loop	valor de	**i = 1**
En el	3er loop	valor de	**i = 2**

Lo interesante de los loops es utlizar el cambio de valor de la variable para generar variaciones del código a repetir. Aprovechando el cambio de valor de i podemos hacer, por ejemplo, lo siguiente:

```
for(int i = 0 ; i<100 ; i+=4){

    line( i, 0, i, height);

}
```

Este código nos permite dibujar 100/4 líneas verticales (ver la figura adjunta). Una cada 4 pixels.
. La variable es “i” y se inicializa con valor 0
. La condición es que “i” sea menor que 100
. Por cada loop el valor de “i” se incrementa en 4.

FOR LOOP “doble”

Un recurso habitual es utilizar dos o más loops, uno dentro de otro.

Ejemplo:

```
for (int i = 0 ; i < 100 ; i+=20 ) {          // primer for loop
    for (int j = 0 ; j < 100 ; j+=20 ) {      // segundo for loop

        fill ( i+j ) ;
        rect ( i, j, 20, 20);

    }                                         // se cierra el primer for loop
}                                             // se cierra el segundo for loop
```

Este código nos permite dibujar una cuadrícula de 5 x 5 cuadrados.
El desarrollo del código es el siguiente:

Primero se toma el valor de la variable “i” del primer “for” :
i = 0
Seguidamente se entra en el segundo “for” y se toma el valor de j:
j = 0
El siguiente paso es definir el color de relleno y dibujar la ellipse para
i=0 j=0
Se repite el paso para el segundo “for” hasta que se deja de cumplir la condición:
“i” y “j” van tomado los siguientes valores:
i = 0 j = 20
i = 0 j = 40
i = 0 j = 60
i = 0 j = 80

Se repite la misma secuencia para el valor de “i” en su segundo loop:

i = 20 j = 0
i = 20 j = 20
i = 20 j = 40
i = 20 j = 60
i = 20 j = 80

Se repite la misma secuencia para el valor de “i” en su tercer loop:

i = 40 j = 0
i = 40 j = 20
i = 40 j = 40
i = 40 j = 60
i = 40 j = 80

Se repite la misma secuencia para el valor de “i” en su cuarto loop:

i = 60 j = 0
i = 60 j = 20

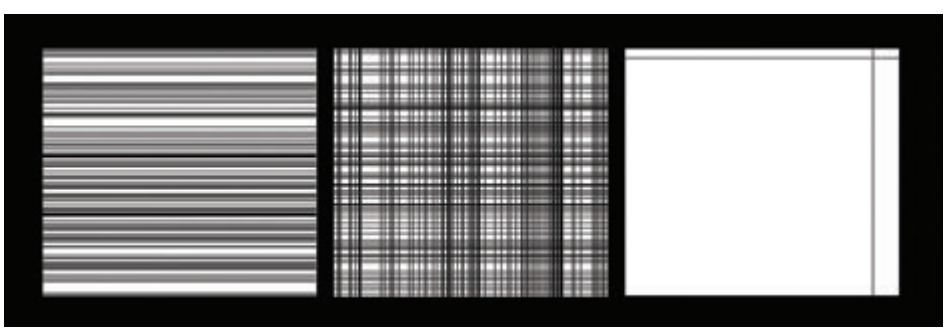


Figura 6

i = 60 j = 40
i = 60 j = 60
i = 60 j = 80

Y se repite la misma secuencia para el valor de “i” en su quinto loop:

i = 80 j = 0
i = 80 j = 20
i = 80 j = 40
i = 80 j = 60
i = 80 j = 80

EJERCICIOS

1. Doble loop

Objetivo: Dibujar cuadrados usando doble loop y rellenarlos con colores aleatorios BN.

Pistas:

```
for(int i=0; i<...; i=i+...){
        for(int j=0; j<...; j=j+...){
                rect( . . . )
        }
}
```

Variables: int lado

2. Doble loop

Objetivo: Dibujar cuadrados separados usando doble loop y rellenarlos con colores aleatorios.

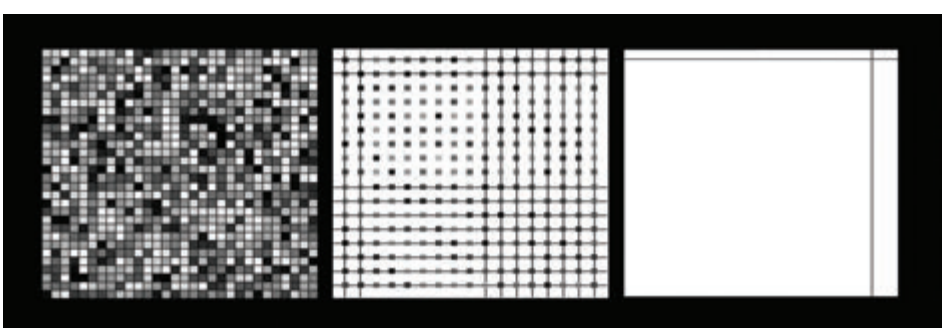


Figura 7

Pistas:

```
for(int i=0; i<...; i=i+...){
        for(int j=0; j<...; j=j+...){
                rect( . . . )
                line( . . . )
        }
}
```

Variables: int separacion
int lado

ARRAYS

Se trata de estructuras para **almacenar** un **conjunto de variables** de **forma ordenada** y bajo un mismo nombre. Dichas variables tienen que ser del mismo tipo. Por ejemplo: int, float, boolean, etc.

DECLARACIÓN DE UN ARRAY :

Al igual que las variables, hay que declarar los arrays. Se hace de la siguiente forma:

tipoDeVariable + [ ] + nombreDelArray ;

Ejemplo:

```
int [ ] posicionesX ;
```

Declaración de un array de tipo entero (**int**) llamado "**posicionesX**"
Este array nos permite almacenar variables de tipo entero

CAPACIDAD DE UN ARRAY :

Los arrays pueden almacenar un cierto número de variables en concreto.
(Más adelante se verá la forma de ampliar y reducir la capacidad de un array)
La capacidad de un array hay que definirla. Lo podemos hacer de distintas maneras:

```
tipoDeVariable + [ ] + nombreDelArray = new + tipoDeVariable + [ capacidad ] ;
```

Ejemplo:

```
int [ ] posicionesX = new int [10] ;
```

Significa que el array llamado "posicionesX" tiene capacidad para almacenar 10 variables de tipo entero.

VALORES DE UN ARRAY :

Los valores que guarda un array se almacenan de forma ordenada. A cada valor del array le corresponde una posición. **Las posiciones se enumeran empezando por 0**.

nombreDelArray + [posicion] = valorDeLaVariable;

Ejemplo:

```
posicionesX[0] = 23 ;
posicionesX[1] = 12 ;
posicionesX[2] = 59 ;
```

Significa que el array llamado “posicionesX” almacena 3 valores ordenados de la siguiente forma:

. En el lugar 0 del array se almacena la variable de tipo entero de valor igual a 23.
. En el lugar 1 del array se almacena la variable de tipo entero de valor igual a 12.
. En el lugar 2 del array se almacena la variable de tipo entero de valor igual a 59.

ACCESO A LOS VALORES DE UN ARRAY :

Una vez se tiene un array con ciertos valores se puede acceder a éstos de la manera siguiente:

valor = nombreDelArray + [posicion] ;

Ejemplo:

```
int [] posicionesX = new int[3];          // declaracion del array de enteros y capacidad 3 valores

void setup(){

    size(400,300);                        // tamaño ventana

    posicionesX[0] = 23;                  // valor [0] del array posicionesX = 23
    posicionesX[1] = 97;                  // valor [1] del array posicionesX = 97
    posicionesX[2] = 64;                  // valor [2] del array posicionesX = 64

}

void draw(){

    ellipse (posicionesX[0], 150, 10, 10);
    ellipse (posicionesX[1], 150, 10, 10);
    ellipse (posicionesX[2], 150, 10, 10);

}
```

Se dibujan 3 elipses. Cada elipse tiene la coordenada x del centro ubicado en

las posicionesX[] correspondientes:

. La primera ellipse dibujada tiene el centro en:
(posicionesX[0], 150) equivalente a (23, 150)
. La segunda ellipse dibujada tiene el centro en:
(posicionesX[1], 150) equivalente a (97, 150)
. La tercera ellipse dibujada tiene el centro en:
(posicionesX[2], 150) equivalente a (64, 150)

ARRAYS + for LOOP

Arrays y loops se utilizan habitualmente de forma conjunta.
Permite empezar a trabajar con cantidades grandes de elementos de forma controlada.

La manera de hacerlo es la siguiente:
Se recorren los valores del array de forma consecutiva utilizando el valor de la variable interna del loop que se incrementa de 1 en 1 (i=i+1) o (i++)

Ejemplo:

```
int [] posicionesX = new int[3];          // declaracion del array de enteros y capacidad 3 valores

void setup(){

    size(400,300);                        // tamaño ventana

    posicionesX[0] = 23;                  // valor [0] del array posicionesX = 23
    posicionesX[1] = 97;                  // valor [1] del array posicionesX = 97
    posicionesX[2] = 64;                  // valor [2] del array posicionesX = 64

}

void draw(){

    for(int i=0; i< 3; i++){

        ellipse (posicionesX[i], 150, 10, 10);

    }
}
```

Como en el ejemplo anterior se dibujan las mismas 3 elipses
Se aprovecha que el valor de “ i “ va tomando los valores **0**, **1** y **2** de forma con-

secutiva para tomar los valores posicionesX[**0**], posciones X[**1**] y posciones X[**2**]

En el primer loop	**i = 0**	por tanto **posicionesX[i]** correponde a **posicionesX[0]**
En el segundo loop	**i = 1**	por tanto **posicionesX[i]** correponde a **posicionesX[1]**
En el tercer loop	**i = 2**	por tanto **posicionesX[i]** correponde a **posicionesX[2]**

. length

Se utiliza para saber la capacidad del array que se quiere recorrer.

nombreDelArray . length

Ejemplo:
Para recorrer el array del ejemplo anterior se puede utilizar:

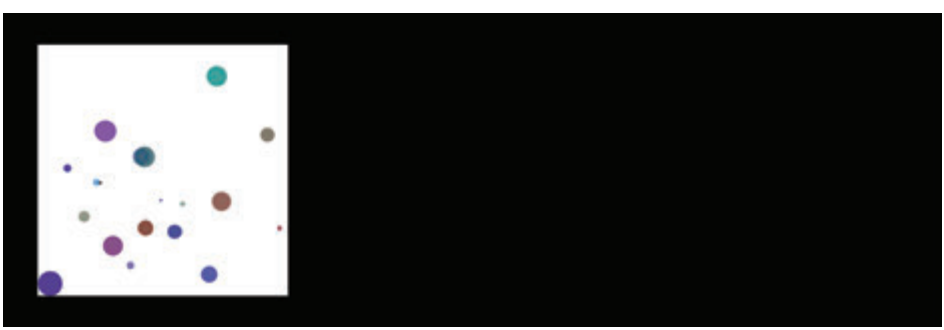


Figura 8

```
void draw(){

    for(int i=0; i < posciones X.length; i++){
        ellipse (posicionesX[i], 150, 10, 10);
    }

}
```

FUNCIONES

Las funciones sirven para “empaquetar” o encapsular partes de código que realizan rutinas concretas.

Nos sirven para organizar el código de forma eficiente y para que este se

pueda interpretar y reutilizar facilmente.

Se definen fuera del draw() y fuera del setup(), al final del código.

Hay 2 tipos de funciones:

. funciones que NO devuelven valores
. funciones que devuelven valores

FUNCIONES QUE NO DEVUELVEN VALORES

void nombreFuncion ( ) { }

```
void + nombreFuncion(){

    CÓDIGO QUE EJECUTA LA FUNCIÓN

}
```

Ejemplo:
Funcion que nos hace una cuadrícula en toda la pantalla en las direcciones X e Y.

```
void cuadricula(){

    stroke(150,0,0);

    for(int i=0; i<width; i+=10){
        line(i, 0, i, height);
    }

    for(int j=0; j<height; j+=10){
        line(0, j, width, j);
    }

}
```

Llamada de la funcion:
Al llamar la funcion ésta ejecutará el código interno que contiene

```
void draw(){
    cuadricula();
}
```

Otras funciones son más flexibles y nos permiten ejecutar la misma función modificando variables contenidas en el código que encierran: los **parámetros**. Se pueden pasar tantos parámetros como sean necesarios.

void nombreFuncion (tipoVariables nombreVariable, ...) { }

```
void + nombreFuncion( tipoVariable nombreParámetro, ...){

    CÓDIGO QUE EJECUTA LA FUNCIÓN

}
```

Ejemplo:

Función que dibuja una elipse del tamaño que le pasamos a través del parámetro _x

```
void circulo(int _x){          // definicion de la funcion y tipo y nombre de variable

    stroke(150,0,0);
    noFill();
    ellipse(10, 10 , _x, _x);  // ejecutamos codigo usando la variable pasada
}
```

Llamada de la la funcion:

```
void draw(){

    circulo(20);               // nos dibujará un circulo de 20 pixeles de diametro
}

void circulo(int _x){          // definicion de la funcion y tipo y nombre del parámetro stroke(150,0,0);
    noFill();
    ellipse(10, 10 , _x, _x);  // ejecutamos codigo usando la variable pasada
}
```

Ejemplo:

La función que nos dibuja la cuadrícula la podríamos hacer más flexible mediante 2 parámetros que nos permitan definir la separación entre líneas horizontales y líneas verticales.

```
void cuadricula(int _sepX, int _sepY){
```

```
    stroke(150,0,0);

    for(int i=0; i<width; i+= _sepX){
        line(i, 0, i, height);
    }

    for(int j=0; j<height; j+=_sepY){
        line(0, j, width, j);
    }

}
```

FUNCIONES QUE DEVUELVEN VALORES

Otras funciones permiten ejecutar un código y devolver el resultado del cálculo ejecutado.

tipoDeVariable nombreFuncion (tipoVariables nombreVariable, ...) { }

Hay que anunciar la función de forma distinta. No seusará “void” sino el tipo de variable que nos devuelve la función.
Al final hay que usar “**return**” para devolver el resultado del código ejecutado.

```
tipoDeVariable + nombreFuncion( tipoVariable nombreVariable, ...){

    tipoDeVariable resultado;

    resultado = CÓDIGO QUE EJECUTA LA FUNCIÓN

    return resultado;

}
```

Ejemplo:

Función que calcula la hipotenusa de un triangulo rectangulo.

Definimos la función con el tipo de variable que devuelve (**float**) y los parámetros que requiere (**float a**, **float b**) , que corresponden a los 2 lados del triangulo rectangulo

```
float calcularHipotenusa( float a, float b){

    float c;                                    // declara la variable que se devolverá al final
```

```
    c = sqrt ( a * a + b * b);      // calcula la hipotenusa y pasa el resultado a la variable a devolver
    return c ;                      // devuelve la variable c , que toma el valor del calculo previo

}
```

A continuación se utiliza la función calcularHipotenusa.
La variable “hipo” toma el resultado de la función.

```
//declaramos variable del mismo tipo que devuelve la funcion
float hipo;

// llamamos la funcion pasando los 2 parámetros requeridos
hipo = calcularHipotenusa(5, 7);
```

OOP. PROGRAMACION ORIENTADA A OBJETOS

Se trata de un paradigma de programación que utiliza “objetos” y las relaciones entre ellos.

Estos objetos tienen unos comportamientos y unas características propias que se definen mediante las “clases”. Se puede definir una “clase” como una familia de objetos con funcionalidades propias.

Un ejemplo para verlo mejor:

Imaginemos una clase “Pelota”. Esta clase nos permite definir objetos Pelota con ciertas características como el tamaño, el color, y si se mueven, se paran,etc.

CLASES

Una CLASE se define mediante 3 tipos de elementos:

CAMPOS o **VARIABLES**:	Permite definir las características de los objetos creados
CONSTRUCTOR:	Permite crear objetos de una clase
MÉTODOS:	Funcionalidades de los objetos creados (moverse, cambiar de color)

DECLARACION

Siempre se comineza por mayúscula el nombre de la clase y por minúscula el

nombre de los objetos

```
class + NombreClase {

    VARIABLES
    CONSTRUCTOR
    METODOS

}
```

Ejemplo:

CLASE PELOTA

Esta clase tendrá las siguientes características:

a. creará un objeto circular (ellipse) que tendrá las siguientes características:
    1. una posicion inicial
    2. una velocidad inicial
    3. un tamaño concreto
    4. un color concreto

b. se visualizará en pantalla
c. se moverá a una determinada velocidad (se actualizará su posición por cada frame)
d. temblará

Los puntos **b**, **c** y **d** responden al comportamiento del objeto mientras que el punto **a** resume las características "físicas" concretas del objeto

Se declara la clase de la siguiente forma:

```
class Pelota {

    VARIABLES
    CONSTRUCTOR
    METODOS

}
```

CAMPOS o VARIABLES

Sirven para definir las características de los objetos de la clase.
Todas éstas variables serán accesibles desde el exterior de la clase como se

verá más adelante. Es decir, se puede acceder a su valor.

Ejemplo:

Se declaran las variables que definiran los objetos Pelota:

. posiciones iniciales **X** e **Y**
. velocidades iniciales **X** e **Y**
. tamaño
. color

De ésta forma, podremos saber en cualquier momento cuál es la posicion, la velocidad, etc. de cualquier objeto que hayamos creado.

```
class Pelota {

    int posX, posY;          // posiciones iniciales
    int vX, vY;              // velocidades iniciales
    int tam;                 // tamaño
    int col;                 // color B/N

    CONSTRUCTOR
    METODOS

}
```

CONSTRUCTOR:

Permite definir como se crean los objetos de la clase.
Precisa de una serie de parámetros que corresponden a las características variables de los objetos a definir.
Se define con el mismo nombre de la clase + los parámetros que precisa
Los parámetros requeridos por el constructor se suelen escribir precedidos por un guion bajo (por ejemplo: _posicionX) para no confundirlos con las variables propias de la clase.

Ejemplo:

En la clase pelota

```
Pelota( posiciones iniciales, velocidades iniciales, tamaño, color ){... }
```

Los valores de los parámetros se pasan a las variables declaradas previamente.

```
class Pelota {

    int posX, posY;                    // posiciones iniciales
    int vX, vY;                        // velocidades iniciales
    int tam;                           // tamaño
    int col;                                   // color B/N

    Pelota(int _posX, int _posY, int _vX, int _vY, int _tam, int _col){

    posX = _posX;
    posY = _posY;
    vX = _vX;
    vY = _vY;
    tam = _tam;
    col = _col;

    }

    METODOS

}
```

METODOS:

Definen las funcionalidades de los objetos creados. Se trata de funciones internas de la clase y se definen como tales.

Ejemplo:

Para la clase Pelota se definen 3 comportamientos:

- dibujarse
- actualizar su posicion en funcion de la velocidad y cambiar de sentido cuando esté al límite de la pantalla
- temblar

```
 void render(){                        // método dibujar pelota
  stroke(col,0,0);
  noFill();
  ellipse(posX, posY, tam, tam);
 }

void update(){                         // método actualizar posicion pelota
  posX = posX+vX;
  posY = posY+vY;
```

```
    if((posX > width-tam/2)||(posX < tam/2)){
      vX = -vX;
    }

    if((posY > height-tam/2)||(posY < tam/2)){
      vY = -vY;
    }
  }

  void temblar(){                                    // método temblar
    posX = posX + int(random(8))-4;
  }
}
```

A continuación se muestran algunos de los trabajos analizados y otros desarrollados por los alumnos:

Figura 9. Interaccion mouse y paleta de colores

Figura 10. Interaccion mouse. “pong” Realizado por Francisco Tabanera

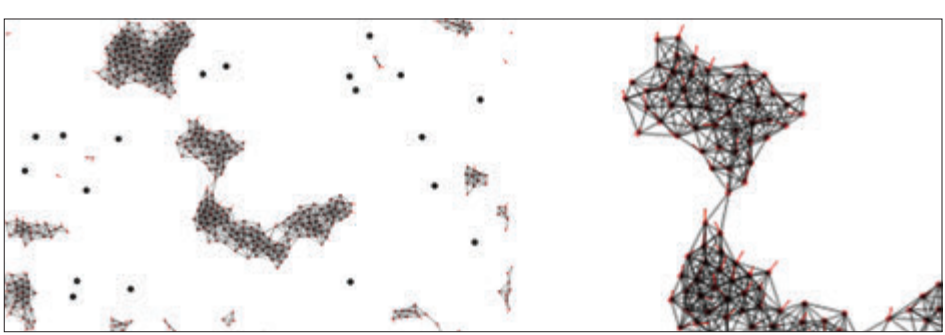


Figura 11. Interaccion mouse y agentes autonomos

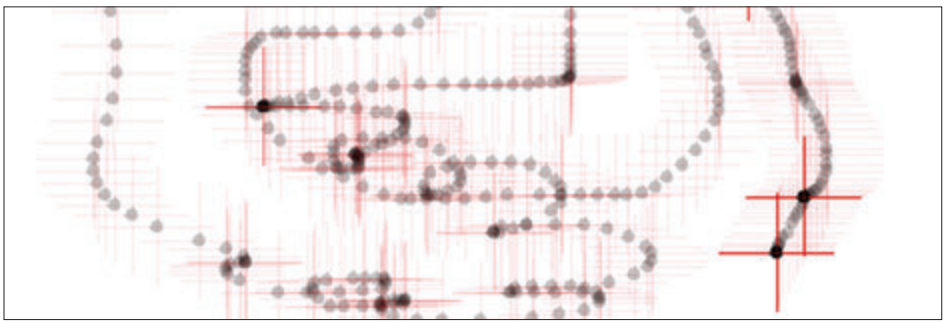


fig. 12.tif Interaccion mouse como pincel

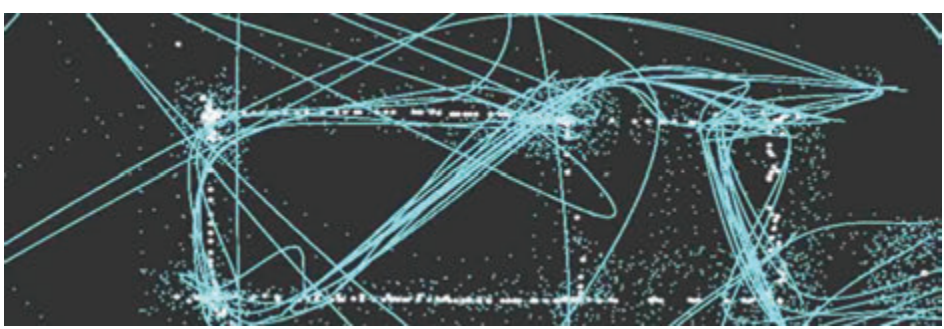


fig. 13.tif Interaccion mouse como pincel. Realizado por Ivan Pajaes

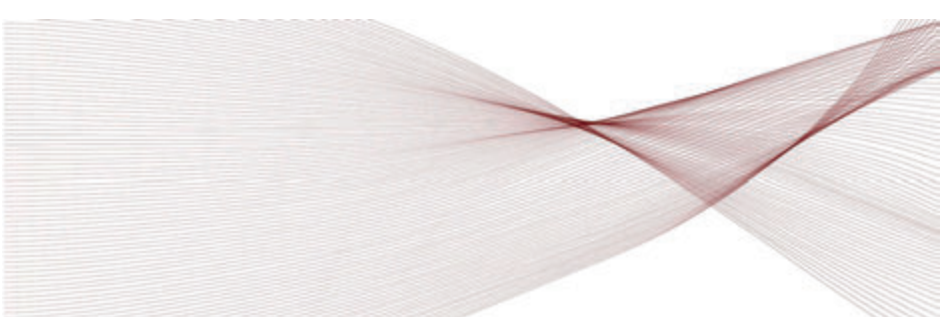


fig. 14 (1).tif Sistemas de particulas y atractores

Conferencias

Durante la sesión final del taller se presentaron las siguientes conferencias:

Ramón Gonzalez - SENER
Cubierta de la Estación de Logroño - SENER+Abalos&Sentkiewicz

Magdalena Ostornol - IDOM / ACXT
Envolvente de Centro de Proceso de Datos en Cerdanyola

Anna Pla Catalá - IE School of Architecture
Cultura Digital en la Escuela de Arquitectura

José Luis Uribe - Universidad de Talca
La Contemporanización de lo Vernáculo

CODA - UPC / ETSAV
Reflexiones sobre el curso Introducción a la Arquitectura Paramétrica

Ruther Paullo - IMAR / FRE
Diseño de Fachadas de Metal Expandido

Pepe Ballesteros - UPM / ETSAM
Al Borde del Cambio de Paradigma

Jerónimo Buxareu & Andrés de Mesa
Sagrada Familia / EtsaB - Herramientas Paramétricas en la Sagrada Familia

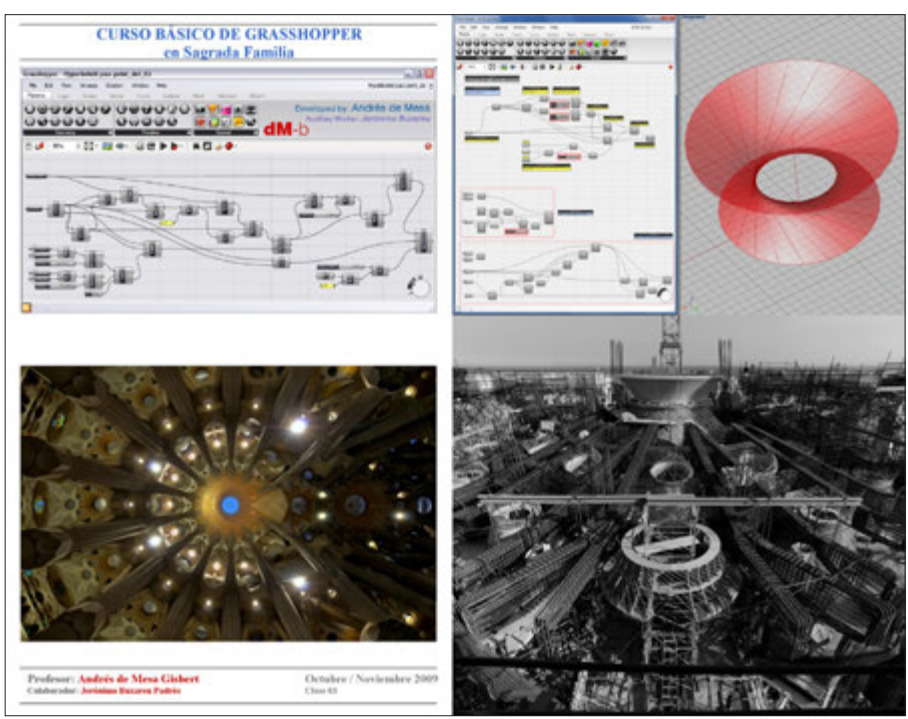
Algunas de esas conferencias se resumen en las siguientes páginas.

Jerónimo Buxareu y Andrés de Mesa
Utilización de Herramientas Paramétricas en La Sagrada Familia

El proyecto de la construcción del Templo de la Sagrada Familia se ha ido diseñando, durante más de 125 años, mediante distintos sistemas de representación. Desde el uso del lápiz y la tinta, hasta métodos de CAD con distintos programas (Autocad, CADD5, Mechanical Desktop, Rhinoceros, Catia v5). En los últimos dos años se ha implementado el plug-in de diseño paramétrico de Rhino, Grasshopper. Este sistema permite resolver con gran eficiencia problemas locales aislándolos del conjunto al que pertenecen.

Se han resuelto una serie de superficies básicas utilizadas por Gaudí como los hiperboloides, paraboloides hiperbólicos y helicoides, organizando una biblioteca básica con distintos métodos paramétricos y dinámicos. El ejemplo que mostramos es el más avanzado. Se trata de la resolución formal de la cimbra de madera que sirvió de base para la colocación de las armaduras del hiperboloide central del ábside (15 x 9.5 x 3.75 metros).

Jerónimo Buxareu y Andrés de Mesa son arquitectos y forman parte del equipo de diseño de la Oficina Técnica de la Sagrada Familia



Jerónimo Buxareu y Andrés de Mesa
Utilización de Herramientas Paramétricas en La Sagrada Familia

José Luis Uribe
La Contemporanizacion de lo Vernáculo en la Arquitectura: El caso del Valle Central de Chile

La conferencia tuvo como objetivo principal reconocer las narrativas del paisaje arquitectónico del Valle Central de Chile. Por un lado, la antigua narrativa caracterizada por las viejas construcciones vernáculas que se distribuyen por el territorio, y por otro lado la nueva narrativa arquitectónica del paisaje que a surgido a través de recientes obras de pequeña escala que toman como partida de proyecto matrices complejas del antiguo habitar vernáculo. Esta nueva narrativa arquitectónica es consecuencia de la práctica académica de la Escuela de Arquitectura de la Universidad de Talca, en la cual el Proyecto de Final de Carrera es una obra construida que debe aportar a lo público y que debe ser diseñada, gestionada y construida por el alumno.

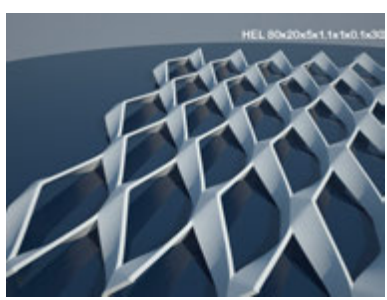
José Luis Uribe Ortiz es arquitecto por la Universidad de Talca, Master en Arquitectura ETSAB UPC y Profesor Escuela de Arquitectura Universidad de Talca.

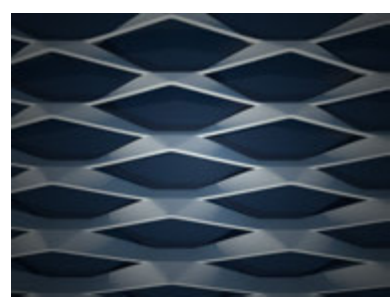
José Luis Uribe. La Contemporanizacion de lo Vernáculo en la Arquitectura: El caso del Valle Central de Chile

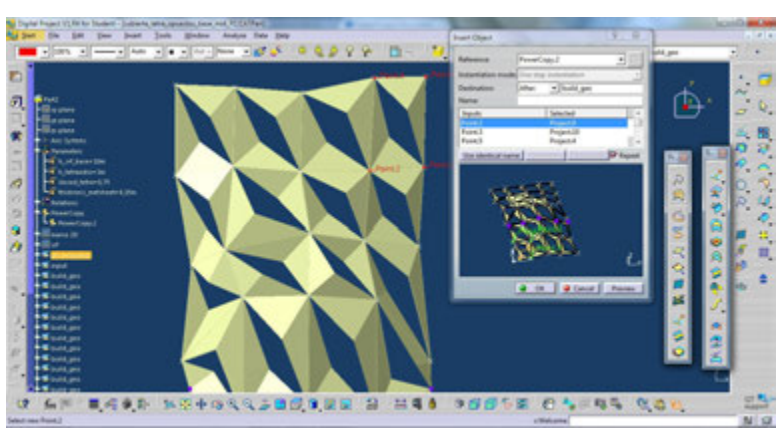
Ruther Paullo Chirinos
Aplicación en Rhinoscripting para Modelado y Paramétrización de Chapas de Metal Expandido

Esta investigación desarrolla una aplicación que controla con precisión los parámetros geométricos de la chapa de metal expandido. Se ha diseñado en rhinoscripting de la plataforma CAD Rhinoceros, basado en VisualBasic. Con esta automatización de procesos se obtiene una infinidad de modelos geométricos válidos que sirve de punto de partida para la materialización final del producto y la innovación en nuevos diseños. Además, tener el control paramétrico de la geometría de una fachada favorece la interacción entre espacio exterior e interior, logrando un mejor control lumínico y de confort térmico que contribuirá al ahorro energético del edificio.

Ruther Paullo Chirinos es arquitecto y su trabajo está patrocinado por la Fundación Rafael Escolá e Industrias IMAR S.A.







Ruther Paullo Chirinos
Aplicación en Rhinoscripting para Modelado y Paramétrización de Chapas de Metal Expandido