|  |
| --- |
|  |
| Memoria Sistemas Informáticos |
| Terraform: juego de simulación y gestión espacial. |
|  |
| **Aris Goicoechea Lassaletta, Marcos Calleja Fernández, Pablo Pizarro Moleón** |
| **01/09/2012** |

|  |
| --- |
| En este documento se describe el proyecto al completo, intentando transmitir los conocimientos necesarios como para reproducir el desarrollo aquí descrito llegando al mismo resultado. Se comentará la arquitectura, el medio elegido, el sistema de trabajo usado y las razones de toda elección que se ha tomado durante su desarrollo. |

Tabla de contenido

[1. Visión 3](#_Toc332388824)

[1. Objetivos 4](#_Toc332388825)

[2. Requisitos 6](#_Toc332388826)

[2. El motor Unity 3D 8](#_Toc332388827)

[1. Introducción 8](#_Toc332388828)

[2. Estructura del proyecto 10](#_Toc332388829)

[3. Particularidades y problemas 11](#_Toc332388830)

[3. Arquitectura 12](#_Toc332388831)

[1. Introducción 12](#_Toc332388832)

[2. Creación del planeta 13](#_Toc332388833)

[3. Algoritmo de vida 16](#_Toc332388834)

[4. Efectos y parte gráfica 17](#_Toc332388835)

[5. Estructura y resto de funciones 18](#_Toc332388836)

[4. Pruebas 19](#_Toc332388837)

[5. Glosario 20](#_Toc332388838)

[6. Referencias 21](#_Toc332388839)

# Visión

El mundo de los videojuegos es un mundo apasionante, lleno de novedades, ideas y creatividad. Cuando a los integrantes del grupo “Terraform” se nos ofreció la posibilidad de elegir un tema libremente para el proyecto de Sistemas Informáticos, pronto tuvimos claro qué queríamos hacer, aunque el cómo y el cual no fueron tan claros.

Como muchos de los profesores nos comentaron, hacer un videojuego es complicado pues lleva una gran cantidad de trabajo, y además dicha cantidad de trabajo suele ser subestimada. Finalmente conseguimos sacar adelante el proyecto de hacer un videojuego, aunque inicialmente la idea era hacerlo para Android (desarrollo de videojuego para dispositivos móviles) e integrar sistemas de realidad aumentada. Finalmente se descartó la realidad aumentada por su complejidad, ya que nos limitaba mucho el desarrollo al obligarnos a dedicar mucho tiempo a investigación y aprendizaje de nuevas tecnologías. También se orientó el desarrollo hacia PC en lugar de Android por necesitarse una potencia de cálculo considerable para ejecutar el proyecto.

Una vez elegida la idea general comenzamos a tomar decisiones para orientar el desarrollo, eligiendo el motor Unity 3D para llevar a cabo el proyecto y decidiendo hacer un juego original que involucrase gestión, simulación y un entorno coherente. Este motor se elige por su versatilidad al ser multiplataforma y por ser 3D, elemento que creemos diferenciador e importante. Por otra parte el hecho de que el proyecto sea finalmente un juego de gestión y simulación fue fruto de una elección pragmática, pues son géneros que involucran menos interacción con el usuario y por ende más simples a priori.

Elegimos también llevar una planificación de tipo SCRUM, aunque adaptada a nuestra situación particular, llevando un desarrollo más ágil que nos permitiera reaccionar más rápidamente ante imprevistos, pues éramos un equipo inexperto en el desarrollo de juegos y tampoco sabíamos cómo funcionaría la realimentación con el profesor/tutor.

Con todo esto y llegado ya diciembre, por fin el grupo se encuentra en posición de empezar el desarrollo y una avalancha de preguntas comienzan a asaltarnos: ¿Cómo construimos lo que estamos pensando? ¿Dónde podemos solucionar nuestras dudas? ¿Cuánto tardaremos en terminar las tareas que vamos empezando? ¿Tendremos tiempo para todo o nos veremos obligados a recortar? ¿Merecerá la pena el sacrificio? ¿Reflejará finalmente nuestro proyecto el esfuerzo volcado en el de una forma digna?

A pesar de todas estas dudas e incertidumbre iniciales, el proyecto “Terraform” se encuentra ya completado, y a lo largo de este documento detallaremos sus construcción y explicaremos sus entresijos paso a paso.

## Objetivos y requisitos

El objetivo fundamental del proyecto “Terraform” es aprender a desarrollar un juego, pasando por todas las etapas necesarias. Por las restricciones intrínsecas a la naturaleza de la asignatura de Sistemas Informáticos (véase las limitaciones del equipo a 3 personas y a la duración de un curso lectivo) el juego a desarrollar será un juego modesto y de tamaño limitado, por lo que intentamos diferenciarnos del resto del mercado siendo innovadores u ofreciendo al menos un videojuego diferente y, esperamos, refrescante.

Dada nuestra voluntad de aprender, hicimos un pequeño estudio de los motores de videojuegos accesibles para nosotros en el mercado. Nuestros requisitos eran que fuera en tres dimensiones y que fuera gratuito. Entre las opciones que barajamos se encuentran el motor “Unreal Development Kit” [r4], “Cry Engine 3” [r5], “id Tech 4” [r6], “Unity 3D” o el motor “Source SDK” [r7]. Estos motores tienen como puntos fuertes el hecho de ser motores ya asentados en el mercado, que se han utilizado para desarrollos comerciales de éxito y el ser accesibles gratuitamente. Nos decantamos por Unity por su trayectoria hasta la fecha de hoy, pues ha mejorado exponencialmente desde su lanzamiento, por sus amplias posibilidades y también por el precio de su versión comercial, mucho más asequible que el de otras opciones.

Entre los juegos creados con Unity se encuentran entre otros títulos como “Shadowgun”, “Rochard”, “Battlestar Gallactica Online” o “Escape Plan”, juegos con un éxito bastante grande desarrollados con medios relativamente modestos.

Uno de los factores más importantes a la hora de desarrollar este proyecto ha sido la motivación de los miembros del grupo en todo momento por construir un juego atractivo y que nos gustara jugar a nosotros mismos. Nuestro tutor nos concedió una amplia libertad para que construyéramos el proyecto a nuestro gusto, y por tanto consideramos dos únicos grupos de interés para definir los requisitos, los usuarios finales y los desarrolladores (nosotros). Con esto en mente, inicialmente nos marcamos estos requisitos funcionales:

1. Generación aleatoria del terreno: El terreno esférico tridimensional se creará de forma aleatoria diferente cada vez, dando lugar además a terreno creíble con presencia de montañas, continentes, llanuras, mares, etc.
2. Controles del jugador: El jugador podrá interactuar con la nave de forma sencilla usando el teclado y el ratón. Además todas las funciones serán accesibles con la mayor facilidad posible.
3. Interacción con el planeta: El usuario podrá interactuar siempre de diferentes maneras con el planeta, haciendo uso de los recursos de la nave mediante los controles adecuados. Podrá construir edificios, liberar o destruir especies y otras acciones.
4. Autonomía de los seres vivos: Todo ser vivo que se encuentre en el planeta tendrá un comportamiento aleatorio, con diferentes acciones a realizar en cada momento. Podrán nacer, reproducirse, interactuar entre ellos o con el entorno o morir, pudiendo reaccionar de diversas maneras ante el mismo escenario. [Nota]
5. Equilibrio: El sistema de juego debe permitir crear y mantener equilibrios entre los edificios y los seres vivos. Así mismo, los propios seres vivos entre ellos podrán crear equilibrios naturales. No obstante igual que pueden crearse y mantenerse, una mala gestión, una decisión poco acertada o la propia aleatoriedad del sistema pueden desestabilizarlos y/o destruirlos.
6. Variedad de creaciones: Desde la nave se podrán crear diferentes seres vivos (plantas, herbívoros o carnívoros) y diferentes edificios, teniendo el jugador varias opciones. Además, dichas opciones se irán desbloqueando poco a poco a medida que el jugador consiga lo necesario para ello.
7. Recolección de recursos: Desde la interfaz principal deben ser visibles los recursos recolectados hasta la fecha, así como los costes de cualquier acción que los tenga o los beneficios de ellas.
8. Velocidad de juego: El juego debe ser lo suficientemente dinámico como para no aburrir al jugador mientras espera cierto acontecimiento. Para ayudar en esta tarea además existirán unos botones en la interfaz que permitan acelerar el tiempo en el juego.

Además de estos requisitos funcionales también extraemos unos requisitos no funcionales:

1. Rendimiento: El juego debe ser capaz de ejecutarse en un computador de gama media (procesador Core 2 Duo o AMD equivalente, tarjeta gráfica nVidia GeForce 9800 GT o AMD Radeon equivalente) a una tasa de imágenes por segundo superior a 30.
2. Estética diferenciadora: Todo componente tridimensional o bidimensional que el usuario perciba en la pantalla estará hecho a mano por nosotros, siendo lo más atractivo posible. Además se cuidará la inmersión y la ambientación.
3. Credibilidad del entorno: La historia y los componentes del juego estarán lo mejor documentados posible, teniendo numerosas referencias a precursores existentes en la actualidad o a acontecimientos históricos de nuestra historia.

A estos requisitos hay que añadir como se ha comentado anteriormente que el proyecto de Sistemas Informáticos tiene sus limitaciones intrínsecas, una de las cuales teníamos todos por igual. Esta limitación no era otra que la carga lectiva del curso académico, por lo que nos era imposible dedicarle al proyecto todo el tiempo que queríamos.

Una vez tuvimos los requisitos marcados el primer problema que tuvimos fue el de saber que la carga de trabajo iba a ser superior a la esperada. No obstante nuestra motivación nos llevó a seguir adelante sin rebajarlos, pues no queríamos recortar ninguno de ellos y decidimos pues poner toda nuestra ilusión en marcha para poder sacar el proyecto adelante.

Por otra parte también tenemos que tener en cuenta los requisitos externos marcados por nuestro tutor y por el marco de desarrollo que conforma la asignatura de Sistemas Informáticos en sí. Estos se pueden resumir con dos conceptos un tanto abstractos y subjetivos: calidad y esfuerzo.

El primero implica que el desarrollo debe ser útil o interesante para alguien y estar bien construido siguiendo los estándares aprendidos durante la enseñanza previa en la carrera.

El segundo implica que a lo largo de un año lectivo se debe cubrir un mínimo de horas por persona equivalente a 15 créditos, lo que equivale a unas 8 horas semanales, y a lo largo del curso a más de 150 horas. Esto debe reflejarse en la calidad y tamaño del trabajo entregado finalmente, y será sujeto a evaluación por parte del tutor a lo largo del curso.

Con todo, los requisitos nos marcan las primeras pautas a seguir, intentando nosotros ir completándolos poco a poco a lo largo del curso.

[Nota] = En un principio nuestro objetivo era también hacer que mutaran aleatoriamente, generando nuevas especies con nuevos comportamientos y aspectos, pero este objetivo se descartó en una fase temprana del desarrollo por su complejidad. Aun así seguimos pensando que sería un añadido genial al proyecto.

# El motor Unity 3D

## Introducción

El proyecto Terraform se sustenta en un motor de creación de videojuegos llamado Unity 3D. Este poderoso motor nos ofrece una API propia, muchas facilidades para la gestión de las diferentes partes del videojuego en sí y en general una capa de abstracción importante entre el funcionamiento interno del juego y el código de los scripts.

Unity entiende los videojuegos como un conjunto de escenas, scripts, objetos de juego y recursos interconectados entre sí:

* Las escenas representan conjuntos de objetos de juego ordenados en una jerarquía específica. Son la unidad de datos más grande que puede manejar Unity dentro de un proyecto.
* Los objetos de juego a su vez representan los objetos que son representados gráficamente en la escena, teniendo cada uno diferentes componentes en su interior. Dichos componentes otorgan funcionalidades al objeto al que están unidos. Como mínimo, cada objeto de juego tiene un componente llamado “Transform” que otorga espacialidad al objeto (una posición tridimensional, una rotación y una escala).
* Los recursos son todos los demás datos necesarios para el funcionamiento del juego: texturas, modelos tridimensionales, sonidos, archivos de datos, etc. Son almacenados separados de la jerarquía de objetos de juego e independientes de las escenas.
* Los scripts son los conjuntos de código que usará el juego durante su funcionamiento, pudiendo usarse como componentes que se pueden unir a los objetos de juego si se cumplen ciertas condiciones. Son considerados recursos a todos los efectos, y por tanto son guardados al margen de las escenas.

La estructura que sigue Unity comprende una jerarquía de objetos de juego (llamados GameObjects) que son la representación de los objetos que tienen influencia directa sobre la escena de juego. Los emisores de iluminación, los objetos tridimensionales, los personajes del juego, los emisores de sonido, sistemas de partículas o el propio terreno son objetos de juego o GameObjects, y por tanto pueden formar parte de la jerarquía de una escena. Esto significa que pueden ser guardados y cargados junto con la escena, formando parte de un único sistema dentro de Unity orientado a proporcionar un fragmento de jugabilidad autónomo.

Para otorgar a cada GameObject de funcionalidad y propiedades existen los componentes (“Component” en la nomenclatura de Unity). Estos componentes se añaden a los objetos de juego y cambian sus propiedades o su funcionalidad en diferentes situaciones. Por ejemplo, añadir un componente emisor de luz a un objeto de juego cualquiera hará que este objeto ilumine a los que se encuentren a su alrededor, mientras que añadirle un componente de tipo malla tridimensional y otro de tipo “renderer” hará que este objeto de juego sea visible dentro del juego con la forma definida por la malla.

Los scripts por si mismos no son componentes válidos y por tanto no pueden añadirse como componente a un objeto de juego, pero Unity ofrece con su API la posibilidad de transformarlos en componentes mediante el siguiente procedimiento: Unity ofrece una clase llamada “MonoBehaviour” de la que se puede heredar al construir una clase. Una vez hecho esto el objeto gana una serie de propiedades especiales y se convierte en un componente válido. Entre estas propiedades se encuentran la posibilidad de modificar los valores de las variables públicas desde la ventana de editor, funciones para controlar su funcionamiento y ejecución u otras propiedades que lo interconectan con el sistema de Unity.

Unity 3D ofrece la posibilidad de desarrollo multiplataforma con un solo clic, permitiendo desarrollos para PC, para Mac, para navegadores Web y para dispositivos móviles con Android o con iOS. Además, puede compilar código de scripts escrito en C#, JavaScript o Boo, ofreciendo para todas estas opciones su propia API que gestiona los elementos nativos. Para nuestro proyecto y por causas que se comentarán más adelante, los scripts se encuentran escritos en C#.

Para facilitar el desarrollo de proyectos grandes en los que trabajen diferentes personas simultáneamente, Unity también es compatible con sistemas de control de versiones. En nuestro caso usamos SVN para sincronizar nuestro trabajo, sustentando el sistema en un repositorio de código alojado en Google Code. Como herramienta de sincronización usamos el software gratuito TortoiseSVN [r2].

TODO

## Escena de ejemplo

Para clarificar mejor el comportamiento y uso de Unity, explicaremos como construir una escena simple desde cero. Esta escena conformará un menú inicial a través del cual podremos ejecutar ciertas acciones simples o movernos a la siguiente escena.

TODO

## Estructura del proyecto

Este proyecto cuenta con muchos recursos diferentes organizados dentro de los esquemas de Unity. Dicho esquema obedece fundamentalmente a la jerarquía de carpetas creada por Unity, que consiste en una carpeta general del proyecto con 2 sub-carpetas: la carpeta “Assets” y la carpeta “Library”. En la segunda se guardan, entre otros, los archivos necesarios para la configuración interna del motor, mientras que en la primera se guardan todos los recursos necesarios para el juego en sí.

En nuestro caso, el repositorio usado contiene la carpeta general del proyecto en su totalidad, aunque se excluyen aquellos archivos temporales y generados automáticamente por Unity que no son necesarios para ahorrar tiempo y espacio.

Dentro de la carpeta “Assets” tenemos multitud de carpetas que nos permiten organizar cada tipo de recurso en su lugar apropiado. Hay carpetas para guardar animaciones, sonidos, modelos, texturas, scripts, elementos de la interfaz de usuario, escenas, “shaders”, etc.

Además de la estructura del proyecto en cuanto a carpetas y recursos, el proyecto sigue otra estructuración diferente una vez dentro del motor. Como se ha comentado antes, las escenas y la jerarquía de objetos de juego constituyen los fragmentos jugables del proyecto y por tanto también tienen una estructuración muy relevante.

“Terraform” está organizado en dos escenas diferentes o niveles de juego. En la primera escena, llamada “Escena inicial”, se presenta el menú principal de opciones, desde el que el jugador puede comenzar una nueva partida, cargar una previamente guardada, modificar las opciones del juego o salir del mismo. Si se elige la opción de iniciar una partida nueva se guiará al usuario por una serie de fases durante las que elegirá en que tipo de planeta quiere jugar, modificando multitud de parámetros durante la creación que le permitirán adaptar a su gusto personal el planeta.

En la segunda escena transcurre toda la acción del juego, pues en ella se nos presenta nuestra nave alrededor del planeta elegido, junto con la interfaz principal que pone a nuestro alcance todas las formas de interacción que tenemos con el planeta y su contenido.

TODO

## Particularidades y problemas

A pesar de las facilidades que nos otorga el motor Unity, su uso no está exento de problemas y limitaciones. Las principales limitaciones a las que nos enfrentamos son debidas a la propia API de Unity, como pueden ser la imposibilidad de hacer uso de hilos de ejecución separados que usen dicha API (pues no es segura en cuanto a hilos), o la práctica imposibilidad de usar otras APIs o “frameworks” no nativos.

El hecho de crear cosas originales y nuevas siempre es problemático, y en el caso de Unity existe la dificultad de que es un software pensado para hacer juegos siguiendo ciertos estándares. En nuestro caso el problema más grave derivado de esto estaba relacionado con el terreno. Usualmente el terreno base en los videojuegos es plano, representando una región cuadrada de terreno transitable. En nuestro proyecto el terreno es una esfera, y al no estar soportado de forma nativa por Unity con la herramienta de terreno integrada, tuvimos que crear de cero una forma de representar y usar un terreno de esta forma. Esto retrasó en gran medida el desarrollo, pues crear un terreno esférico y manipularlo requiere altas dosis de geometría y muchos “trucos” para hacer que las funciones originalmente pensadas para terrenos planos funcionen de forma adecuada.

Otro de los problemas encontrados tiene que ver con el rendimiento. Unity ejecuta de forma nativa sobre un solo hilo de ejecución como se ha comentado previamente. Por tanto, para que ciertos procesos, como puede ser el proceso de creación del planeta, se ejecutaran más rápidamente hemos tenido que renunciar a ciertas florituras y a aplicar infinidad de trucos para acelerar el código. Por ejemplo el uso de archivos con cómputos previamente calculados, el uso de vectores nativos y otras muchas modificaciones en pos de la velocidad.

TODO

# Arquitectura

## Introducción

Como ya se ha comentado previamente, la arquitectura del proyecto está fuertemente ligada a las restricciones y normas del motor Unity. Por claridad vamos a separar la arquitectura inicialmente en 3 partes:

* La creación del planeta de forma aleatoria mediante el uso de ruido.
* El algoritmo Vida que permite la simulación de seres vivos y edificios.
* La parte gráfica del proyecto, que incluye “shaders”, materiales y otros objetos.

Todas estas partes están repartidas entre las escenas y por ello se añadirá una cuarta parte para clarificar la relación entre todo ello. En esta cuarta parte también se hablará del resto de funciones que sustenta la arquitectura.

Entre estas funciones se encuentran la posibilidad de guardar y cargar el progreso del juego, la interfaz de usuario, las mejoras de la nave, la gestión de recursos, etc.

TODO

TODO

## Creación del planeta

Para la creación aleatoria del planeta nos enfrentábamos a varios retos. El primero era que soporte utilizar, teniendo en cuenta nuestras necesidades y las facilidades que nos otorgaría cada aproximación. La opción elegida fue usar una textura (Texture2D en términos de la API de Unity) sobre la que pintaríamos la representación de la superficie del planeta. Esta aproximación nos permite reducir la lógica a una representación bidimensional y posteriormente portarla a un objeto tridimensional, lo cual es mucho más sencillo programáticamente. Llamamos a esta textura “Textura\_Planeta”.

Para cumplir el requisito de generación aleatoria de terreno necesitábamos una función de ruido de calidad [r1], entendiendo calidad como la conjunción de las siguientes cualidades:

* Repetible: A pesar de ser aleatoria, queremos que dada una semilla y una entrada iguales, siempre nos devuelva el mismo resultado. Debe ser pues pseudo-aleatorio, pero con un patrón nada obvio.
* De rango conocido: Preferiblemente queremos un ruido que devuelva valores entre unos márgenes conocidos, como [0, 1] o entre [-1, 1] y cuya media sea conocida también.
* Rápido de evaluar.
* Resolución ilimitada: La entrada de la función de ruido debería tener el rango más amplio posible, obteniendo diferentes salidas para cada entrada.

Además de estas cualidades era vital que el ruido elegido tuviera una más: debía dar lugar a patrones que pudieran ser interpretados como terreno, evitando el efecto de “nieve” en el que cada punto generado aleatoriamente es completamente diferente de los circundantes y no guarda ninguna relación.

Por todo ello, la elección natural era el uso de ruido basado en ruido Perlin. Nuestra aproximación hace uso de varias octavas de ruido para lograr un mayor detalle. Este tipo de ruido en el que se suman varias octavas está basado en el ruido “fBm” (o “fractional Brownian motion”), siendo esta variante en concreto llamada “ruido de turbulencia”. Este ruido en particular hace un sumatorio de octavas de ruido Perlin con su valor absoluto, ofreciendo además la ventaja de no existir valores negativos.

Una vez conseguido el ruido para cada pixel de la textura, se pintan en ella valores en escala de grises (mismo valor para los tres componentes del color del pixel) que representan la altura de dicho pixel en un rango [0, 1] (en punto flotante). La textura “Textura\_Planeta” tiene unas dimensiones de 2048 x 1024 pixeles, por lo que rellenar los valores apropiados de ruido dentro de la textura tiene un coste bastante elevado.

Hecho esto se aplican unas funciones sobre la textura, concretamente dos. La primera hace que coincidan el borde derecho y el izquierdo de nuestra textura rectangular, aplicando una serie de transformaciones a ambos extremos y teniendo en cuenta los valores del ruido de ambas partes para interpolar el valor final del pixel entre ellos dependiendo de la cercanía a un extremo del punto en cuestión.

La segunda función suaviza la textura en la zona de los polos, haciendo que los valores de ruido vayan disminuyendo suavemente al acercarse a los polos, de tal forma que siempre haya altura mínima en ellos y los valores de altura cercanos vayan en descenso.

A continuación se procede a extruir los vértices de una esfera previamente acondicionada [Nota] atendiendo a “Textura\_Planeta”. Para ello, accedemos a cada vértice de la esfera y consultamos su coordenada UV. Esta coordenada representa el punto de proyección sobre la textura, lo que significa que cuando la textura se proyecte sobre el objeto ese punto de la misma coincidirá con el punto en el espacio del vértice. Una vez obtenido su coordenada UV, accedemos a ese punto en concreto de la textura y consultamos el valor del vértice, consiguiendo un valor en el rango [0, 1] generado previamente por el ruido. Con este valor ya en nuestro poder, desplazamos el vértice en el espacio una cantidad proporcional al mismo y a una variable de control (el factor de extrusión) siguiendo la dirección de la normal del propio vértice. Esto consigue que los vértices se “alejen” del centro de la esfera de forma proporcional al valor de la textura creada.

Con los vértices extruidos se crea un objeto tipo “Mesh” o “malla tridimensional” que se añade al objeto de juego que representa el planeta. El mismo objeto se usa como base para crear el objeto “Collider”, que será necesario posteriormente para detectar colisiones y para ciertas interacciones requeridas por el código.

El siguiente paso es crear una nueva textura a partir de “Textura\_Planeta” en la que representamos la presencia de agua en el planeta. Se pintan los píxeles que quedan por debajo de un cierto nivel de un color diferente (en este caso, el color en escala de grises es el valor de la variable “nivelAgua” que tiene un rango entre [0, 1], el mismo que los píxeles de “Textura\_Planeta”) y luego seguimos el mismo patrón que al extruir los vértices del planeta. A esta textura la llamamos “Textura\_Agua”. Al extruir los vértices de la misma esfera pero con esta nueva textura obtenemos un nuevo “Mesh” que al ser colocado en un objeto de juego que se encuentra en la misma posición que el anterior cubre las partes de tierra que quedan por debajo del nivel del agua elegido.

Una vez completados estos pasos disponemos de lo siguiente:

* Un objeto llamado “roca” con una malla tridimensional extruida mediante el uso de una textura creada con ruido aleatorio y posteriores transformaciones (“Textura\_Planeta”).
* Un objeto llamado “océano” con una malla tridimensional extruida mediante el uso de otra textura creada a partir de la primera, coloreando solo las zonas por debajo de un valor previamente elegido (“Textura\_Agua”).
* Al poner ambos objetos en la misma posición espacial, se superponen creando la ilusión de un único objeto en el que las partes de terreno por debajo del valor elegido se encuentran cubiertas de agua.
* Todo lo anterior se genera de forma aleatoria cada ejecución.

[Nota] = Esta esfera es creada con un software de edición de modelos tridimensionales llamado “Blender” [r3].

TODO

## Algoritmo de vida

TODO

## Efectos y parte gráfica

TODO

(Introduccion) Para representar gráficamente todos estos datos y operar con ellos hemos usado ...

(Temas) Interfaz, Shaders, Animaciones

(Interfaz) Interfaz Unity, guiskins, estilos, mapa de la interfaz

(Shaders) intro shaders, materiales (y hablar de filtros), herramientas, construccion y "materias primas" (shaders esteticos e informativos), relacion de shaders con gameobjects,

(animaciones) utilidades, eventos, efectos de particulas (?)

## Estructura y resto de funciones

TODO

# Pruebas

TODO

# Glosario

* Terraform:
* Unity 3D:
* GameObject:
* Shader:
* Transform:
* Mesh:
* Collider:
* Texture2D:
* Component:
* Renderer:
* MonoBehaviour

TODO

# Referencias

* [r1] “The Importance of Being Noisy: Fast, High Quality Noise”, Natalya Tatarchuk <http://developer.amd.com/assets/Tatarchuk-Noise(GDC07-D3D_Day).pdf>
* [r2] “TortoiseSVN : the coolest interface to (Sub)version control ”, <http://tortoisesvn.net/>
* [r3] “Blender: free open source 3D content creation suite”, <http://www.blender.org/>
* [r4] “Unreal Development Kit”, <http://www.unrealengine.com/en/udk/>
* [r5] “Cry Engine 3”, <http://mycryengine.com/>
* [r6] “id Tech 4”, [http://web.archive.org/web/20081026115600/http://www.idsoftware.com/business/idtech4/](http://web.archive.org/web/20081026115600/http:/www.idsoftware.com/business/idtech4/)
* [r7] “Source SDK”, <http://source.valvesoftware.com/>