Facultad de Informática

Memoria Sistemas Informáticos

Terraform: simulación de vida sobre planetas generados.

Proyecto de Sistemas Informáticos, Facultad de Informática, Universidad Complutense de Madrid
Curso académico 2011-2012

Autores: Aris Goicoechea Lassaletta, Marcos Calleja Fernández, Pablo Pizarro Moleón
Profesor director: Jorge Jesús Gómez Sanz

14/09/2012

En este documento se describe el proyecto al completo, intentando transmitir los conocimientos necesarios 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.

Licencia

Resumen

“Terraform: simulación de vida en planetas generados” es un proyecto en el que nos propusimos el aprender las nociones básicas sobre creación de videojuegos y superar las dificultades que ello supone. El videojuego en si es un simulador de vida que permite la generación de planetas de forma procedural mediante el uso de ruido perlin, permitiendo al usuario final modificar algunos de los parámetros. Durante el transcurso del juego el jugador manejara una nave en su empresa de convertir este planeta en uno más similar a la tierra, permitiendo a la vida asentarse, para permitir su posterior colonización. El objetivo del juego es construir ciertas estructuras que proporcionan recursos que luego usaremos para crear plantas y animales que formen un ecosistema en la superficie del planeta.

Como motor para la creación hemos usado Unity3D, que tiene la ventaja de ser un editor muy poderoso con una versión gratuita y una curva de aprendizaje suave. Todos los elementos del juego salvo alguna excepción han sido creados a mano en Blender, un editor de modelos 3D opensource y gratuito.

Palabras clave:

- Terraform
- Videojuego
- Simulador de vida artificial
- Generador procedural de planetas
- Unity3D
- Blender

Abstract

“Terraform: life simulation on generated planets” is a project in which we aimed to learn the basics of videogame creation and overcome the challenges that it takes to do so. The videogame itself is an artificial life simulator that allows the procedural generation of planets by using perlin noise, allowing the final user for modification of certain parameters. During gameplay the player will handle a starship in its quest of making planets more Earth-like, allowing life on them, to later on populate them with its civilization. The objective of the game is to build certain structures that provide resources we will later use to create plants and animals to sustain an ecosystem on the surface of the planet.

We used Unity 3D as game engine, which has the advantage of being a very powerful engine with a free version and a smooth learning curve. Every asset of the game save some exceptions is handicrafted by ourselves, such as the 3D modelsof Plants, Animals, the Starship and the structures, all made in the free, opensource 3D modelling software Blender.

Keywords:

- Terraform
- Videogame
- Artificial life simulator.
- Procedural planet generator.
- Unity3D
- Blender

Tabla de contenido

1. Visión

1. Introducción

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 que queríamos hacer un videojuego. Lo difícil era que tipo de juego hacer y sobre todo cómo hacerlo.

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 inicial, el proyecto “Terraform” se encuentra ya completado, y a lo largo de este documento detallaremos su construcción y explicaremos sus entresijos paso a paso.

2. Concepto del videojuego

Pero… ¿Qué es “Terraform”? Es un videojuego que se podría encuadrar dentro del género de la estrategia comercial. Algunos títulos de jugabilidad similar pero con una complejidad obviamente muy superior serían: Civilization, Anno, SimCity, Tropico, Populous, SimEarth. Se podrían nombrar decenas de títulos en los que nos hemos basado consciente y sobre todo inconscientemente.

Concretamente “Terraform” es un videojuego de simulación de vida artificial y gestión en el que el usuario controla una avanzada nave espacial con la que busca planetas que terraformar. Gracias a las increíbles posibilidades de la nave y a su pericia, el jugador puede convertir un planetoide inhabitado en un vergel rebosante de vida e industria en el que se pueda prosperar.

El juego empieza cuando la nave que controlas llega al planeta. A partir de ese momento podrás llevar a cabo una serie de acciones como construir edificios que te reporten recursos o como insertar vegetales y animales y hacer florecer la vida. Aparte la nave nos ofrecerá una serie de mejoras que permitirán obtener nuevos recursos y sobre todo realizar ciertas habilidades. Estas habilidades van desde filtrar visualmente ciertos elementos a matar a seres que te estén causando problemas o fomentar el crecimiento de otros. Evidentemente todas las mejoras y las habilidades consumirán parte de los recursos que nos proporcionan los edificios.

Es interesante comprobar cómo las plantas se van extendiendo por sus hábitats y como intentan acceder a algunos sin mucho éxito. Por otro lado los animales le dan un toque mucho más “vivo” al planeta, ya que su constante movimiento y sus simpáticas animaciones proporcionan la idea de que el planeta se está convirtiendo poco a poco en habitable.

Uno de los edificios más relevantes es la granja que permite aprovechar la nueva vida del planeta para recolectar un nuevo recurso que se utiliza en las mejoras y habilidades más avanzadas de la nave.

Finalmente el juego terminará cuando utilicemos una habilidad especial que creara un portal que comunicará con nuestro planeta natal y nuestra misión en ese planeta habrá terminado. No obstante, gracias al generador procedural de planetas cada partida será única y diferente.

3. 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 “UnrealDevelopment Kit” [r4], “CryEngine 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”, “BattlestarGallactica 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.[1]

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.

[1]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.

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 nVidiaGeForce 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.

2. El motor Unity 3D

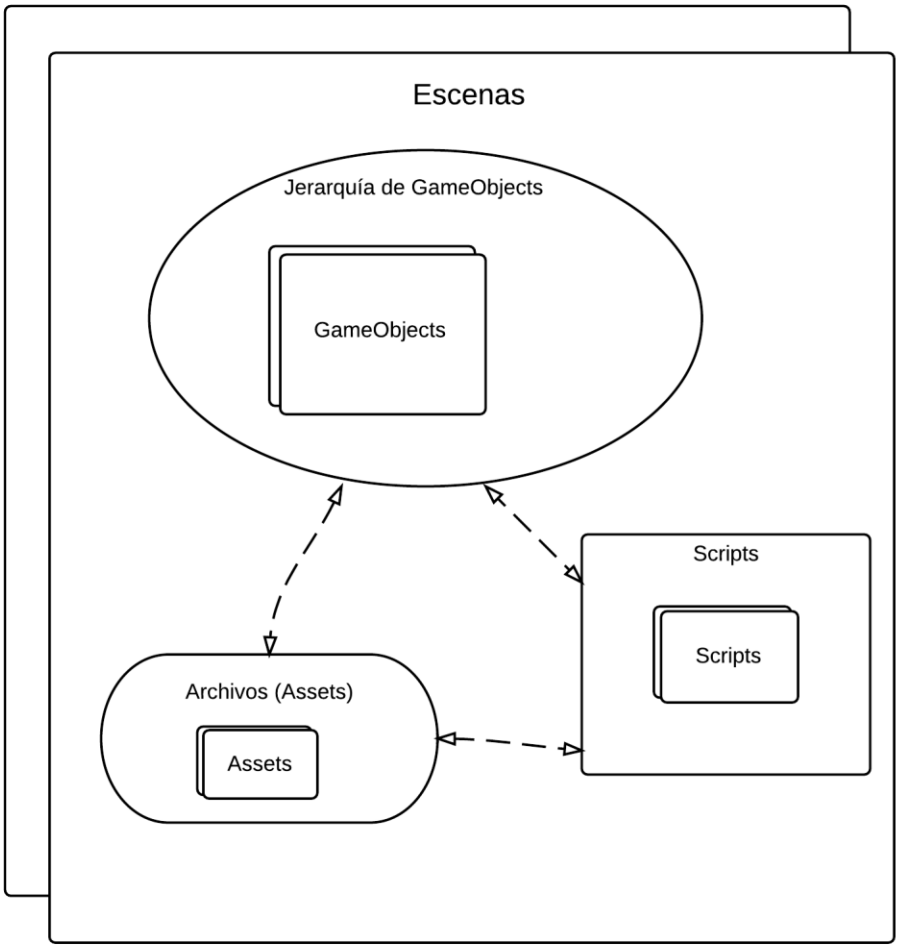
1. Introducción

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.

Una representación de como maneja Unity las escenas sería:



Esquema general de la estructura de Unity

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 ajenas a nosotros[2], los scripts se encuentran escritos en C#.

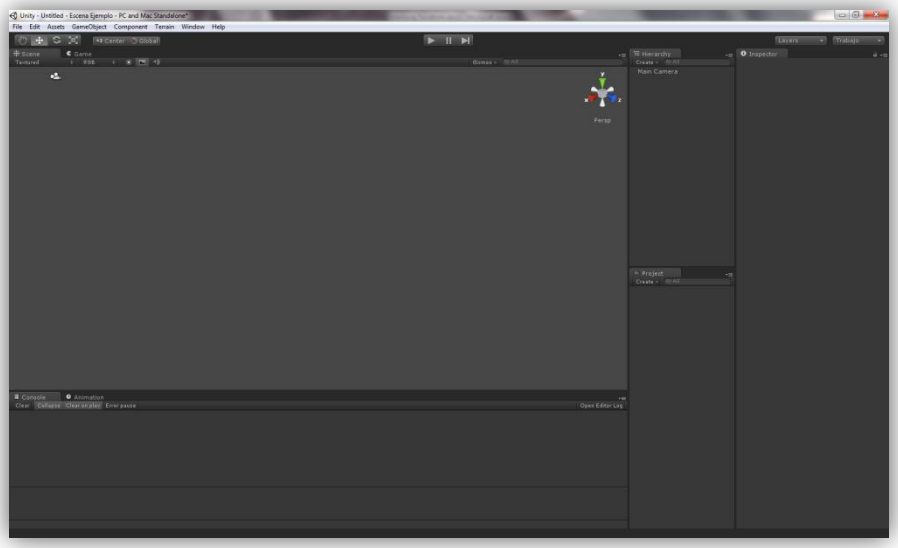
Para facilitar el desarrollo de proyectos grandes en los que trabajen diferentes personas simultáneamente, Unity también es compatible con sistemas de control de versiones. En nuestro caso usamos SVN para sincronizar nuestro trabajo, sustentando el sistema en un repositorio de código alojado en Google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

[2]Unity compila los scripts en un determinado orden, y solo permite ciertas cosas si se usa el lenguaje C# en ellos, permitiendo un control de las dependencias entre los scripts absoluto y manual en ese caso. Usando JavaScript este control es imposible de sonseguir.

2. 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.

El primer paso en Unity siempre es abrir un nuevo proyecto. En el asistente que se abre se nos da la opción de importar paquetes básicos de datos de ejemplo que podemos usar para acelerar el comienzo del proceso. Una vez elegido el nombre, el programa crea el nuevo proyecto y abre el programa dejándonos frente a la interfaz principal.



Pantalla principal de Unity

Podemos distinguir 5 áreas de trabajo diferentes: la ventana de escena (“Scene”) que comparte el mismo espacio que la ventana de juego (“Game”), las ventanas de jerarquía y de proyecto (“Hierarchy” y “Project” respectivamente), el inspector (“Inspector”) y la consola.

La ventana de escena, también llamada “viewport”, nos muestra el entorno tridimensional que estamos en proceso de crear. Por el podremos desplazarnos, seleccionar objetos, modificar sus posiciones, rotaciones o escalas y en definitiva modificar la parte visible. La ventana de juego ofrece una previsualización de lo que capta la cámara principal del juego y, cuando se activa el modo de juego con el botón de “Play” en la zona superior central, permite al usuario hacer funcionar la escena como si del juego final se tratara.

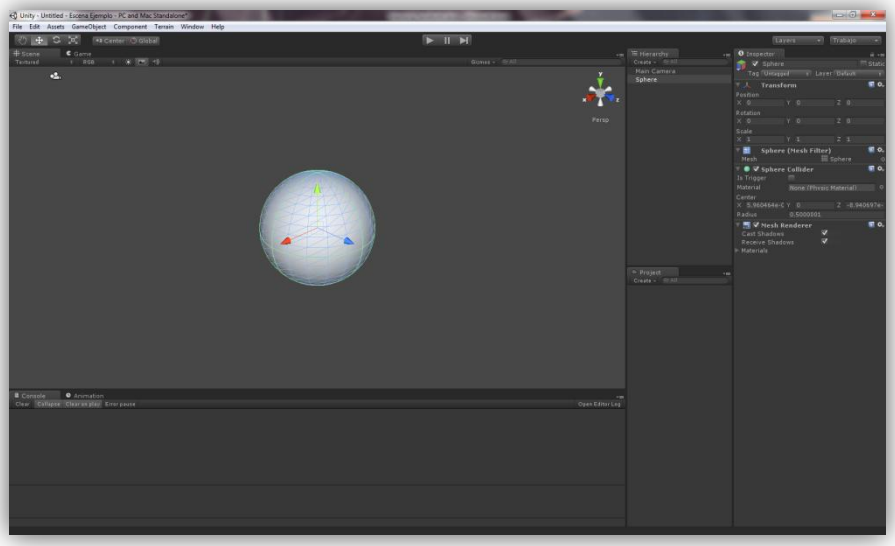
Las ventanas de jerarquía y proyecto son similares en su funcionamiento: mientras la de jerarquía guarda y organiza los “GameObjects” del juego, permitiendo la correcta navegación entre ellos, la ventana de proyecto hace exactamente la misma función, pero con los archivos o “assets” del proyecto, que representan las estructuras de datos que dan soporte real a los objetos del juego. Mientras que la ventana de jerarquía recoge su información de la escena, la ventana de proyecto representa un sistema de carpetas y archivos accesible desde el sistema operativo.

La ventana del inspector nos ofrece toda la información disponible del objeto seleccionado actualmente en la ventana de jerarquía o en la ventana de proyecto. En ella se visualizan los componentes actuales del objeto seleccionado así como todas sus propiedades. Ofrece la posibilidad de cambiar y editar la mayoría de los campos visibles de dichos objetos, teniendo todo cambio un efecto inmediato en el resto de ventanas.

Para comenzar la construcción de nuestra escena, primero planificamos de forma esquemática lo que queremos que se visualice y organizamos de forma general la escena. Para este ejemplo, vamos a situar una esfera con la textura de nuestro planeta (la Tierra) en el centro de la cámara, a dibujar un “skybox”[3] que muestre un cielo estrellado y un simple menú que nos permita iniciar el juego o salir de él.

[3]Un “skybox” es un cubo texturizado cuyas paredes se encuentran siempre a una distancia infinita de la cámara y que representan el límite del espacio que podemos renderizar. Suele usarse para dibujar el cielo y el horizonte visible, siempre en forma de una textura en dos dimensiones.

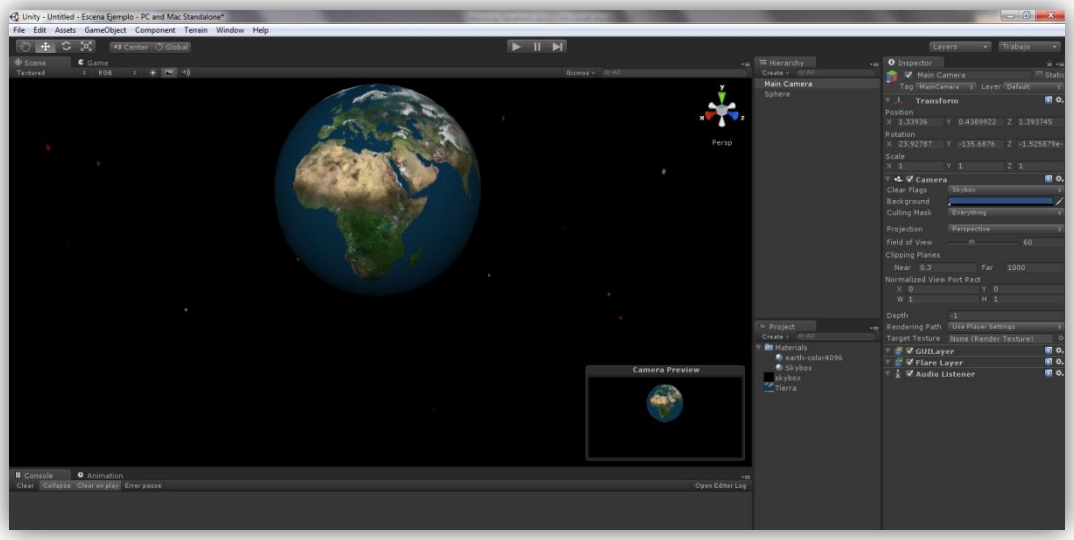
Primero colocamos una esfera en la escena. Para ello, seleccionamos el menú “GameObject”, ahí nos vamos a “CreateOther” y finalmente seleccionamos “Sphere”. Esto hará aparecer una esfera gris en el centro de coordenadas de la escena, el punto en el espacio (0, 0, 0). Para comprobarlo, seleccionamos la esfera recientemente creada y observamos como el inspector nos describe sus propiedades. Ahora mismo, la esfera recién creada tiene 4 componentes: un componente “Transform”, un “MeshFilter”, un “MeshCollider” y un “MeshRenderer”. No entraremos en ellos de momento pues no son relevantes por ahora.



Esfera recién insertada en la escena

Hecho esto, importamos a nuestra carpeta de proyecto una textura de la Tierra. Pulsamos el botón derecho del ratón encima de la ventana “Project” y seleccionamos “Import New Asset…”, seleccionando la imagen que deseemos para introducirla al proyecto. Una vez hecho esto, podemos pinchar y arrastrar esta nueva textura desde la ventana de proyecto hasta la esfera, y al soltarla encima Unity se encargará de crear un nuevo material con la textura deseada debidamente implementada y aplicárselo a nuestra esfera.

El siguiente paso será crear un “skybox” que represente el espacio exterior. Para ello y por simplicidad tomaremos una imagen negra con pequeños puntos blancos que importaremos del mismo modo que la anterior textura. En la ventana de proyecto creamos un nuevo material (botón derecho y “Create”, después “Material”) y en la ventana de inspector cambiamos la propiedad “Shader” del mismo de “Diffuse” a “Skybox”, que se encuentra dentro de la categoría “RenderFX”. En este nuevo material se nos permite usar un total de 6 texturas, una para cada cara del cubo. Nosotros ponemos la misma textura previamente importada arrastrando y soltándola en todos los lugares por igual. Por último, en el menú superior de Unity seleccionamos “Edit” y dentro “RenderSettings”, arrastrando nuestro nuevo material al hueco nombrado “Skybox Material”. De paso, en este mismo menú aprovechamos para aumentar un poco la luz ambiental, pulsando en la tira de color a la derecha de “Ambient Light”. Un selector de colores aparece y en el seleccionamos un color más cercano a blanco, para dar más iluminación a la escena (siendo negro la ausencia de luz de ambiente y blanco que cada objeto está completamente iluminado con luz blanca).



Escena con skybox, cámara y esfera con textura

Bien, ya solo nos falta la cámara por colocar. Para ello, primero colocaremos nuestra vista de la ventana de escena como queramos mediante los controles adecuados y después seleccionamos la cámara de la vista de jerarquía. Con ella seleccionada vamos al menú “GameObject” y seleccionamos la opción “AlignWith View”, moviendo la cámara a la misma posición en la que se encuentra nuestra vista y mirando en la misma dirección.

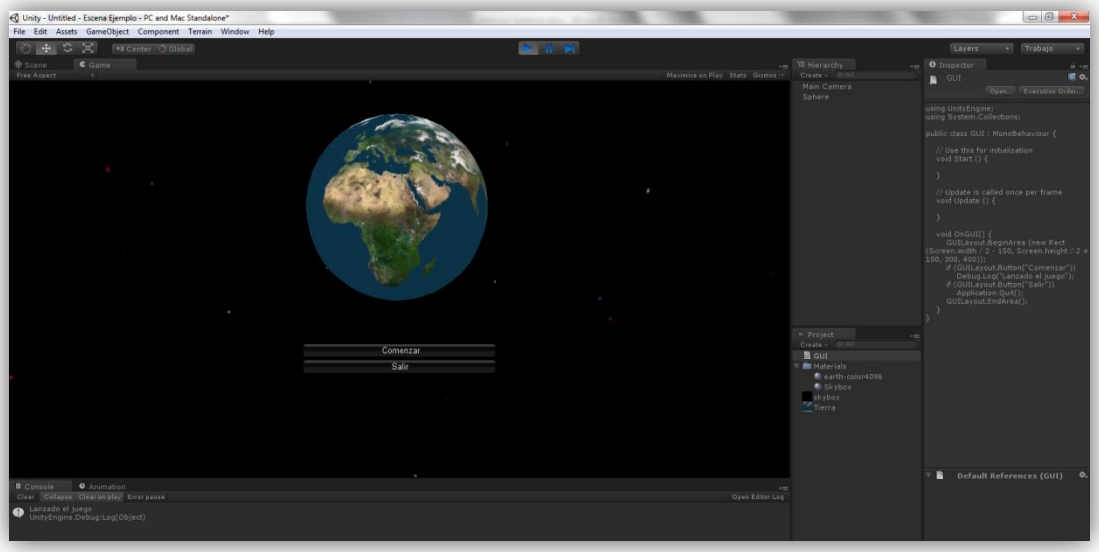
Con estos pasos ya tenemos creado el soporte básico para la escena, pero aun nos falta interactividad. La interactividad que necesitamos ahora mismo nos la debe proporcionar una interfaz gráfica de usuario, o como se llama esto en Unity, una “GUI”. Para crearla, necesitamos programarla en forma de script. Para ello, pulsamos el botón derecho nuevamente en la ventana de proyecto y creamos un nuevo script de C#, llamándolo “GUI”. Hacemos doble clic en él y se abre el programa de edición de scripts predeterminado, en este caso MonoDevelop, que nos permitirá escribir el código necesario para mostrar la interfaz que queramos.

Vemos que hay cierto código por defecto ya escrito en el script. Lo ignoramos y después del método “Update” ya escrito introducimos el siguiente fragmento de código:

```
    // Update is called once per frame
    void Update () {

    }

    voidOnGUI() {
        GUILayout.BeginArea (new Rect(Screen.width/2 - 150, Screen.height/2 + 100, 300, 400));
        if (GUILayout.Button("Comenzar"))
            Debug.Log("Lanzado el juego");
        if (GUILayout.Button("Salir"))
            Application.Quit();
        GUILayout.EndArea();
    }

}
```

Este fragmento de código crea un área en la pantalla delimitada por el rectángulo “Rect”, que ocupa 300 x 400 píxeles en la zona central-inferior de la pantalla. También crea dos botones y les asocia una pequeña funcionalidad. El primero, que mostrará las letras “Comenzar”, escribe un pequeño texto en la consola que dice “Lanzado el juego” cuando es pulsado. El segundo botón sale de la aplicación. La instrucción ‘Application.Quit()’ de este fragmento solo funciona una vez la escena ha sido construida y convertida a ejecutable, por lo que no podremos probarla por ahora. Una vez que nuestro juego disponga de más escenas y sea jugable, sustituiremos la línea ‘Debug.Log(“Lanzar el juego”)’ por algo como ‘Application.LoadLevel(“Escena 1”)’, pero de momento lo dejamos así.

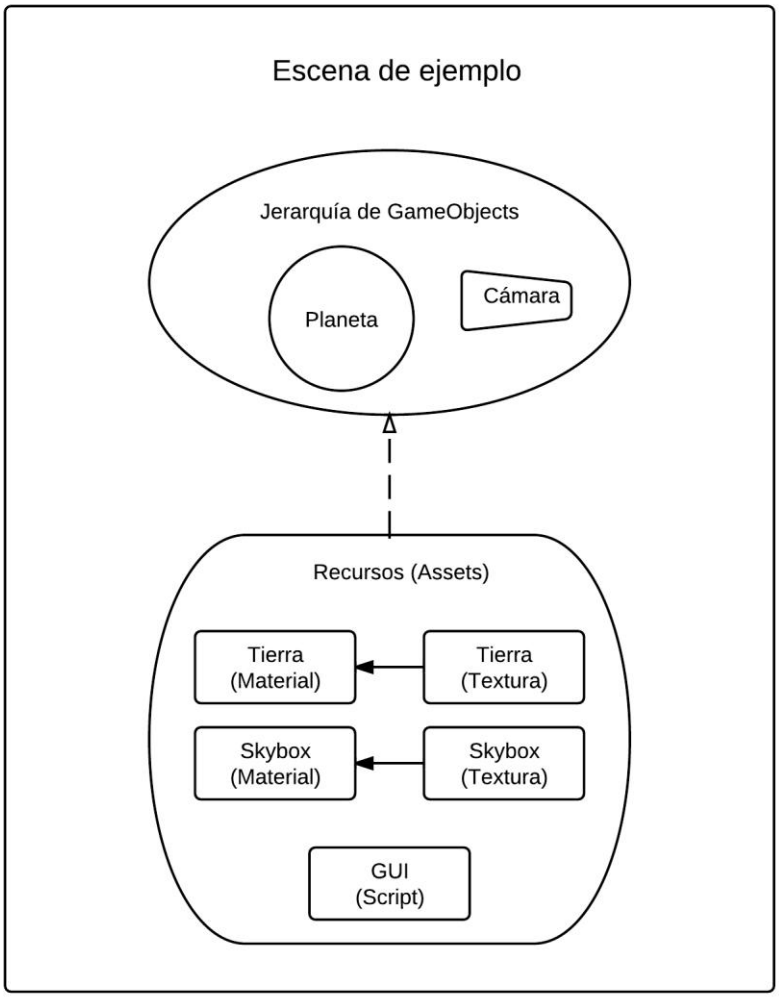
El último paso es agregar el script recién creado a un objeto de juego que esté en la escena, para que su código se ejecute cuando el objeto sea procesado. Lo más apropiado para nosotros es arrastrarlo desde la ventana de proyecto y soltarlo encima de la cámara principal en la ventana de jerarquía, añadiéndolo como componente a esta última.



Escena completada y ejecutándose dentro de Unity

Con esto tenemos terminada nuestra primera escena y una sencilla forma de iniciar el juego. Para poder usarla posteriormente, la guardamos pulsando el menú “File” y “SaveScene As…”, eligiendo el nombre que consideremos apropiado.

En el siguiente diagrama podemos ver como queda la escena recién creada:



Estructura de la escena de ejemplo

3. 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.

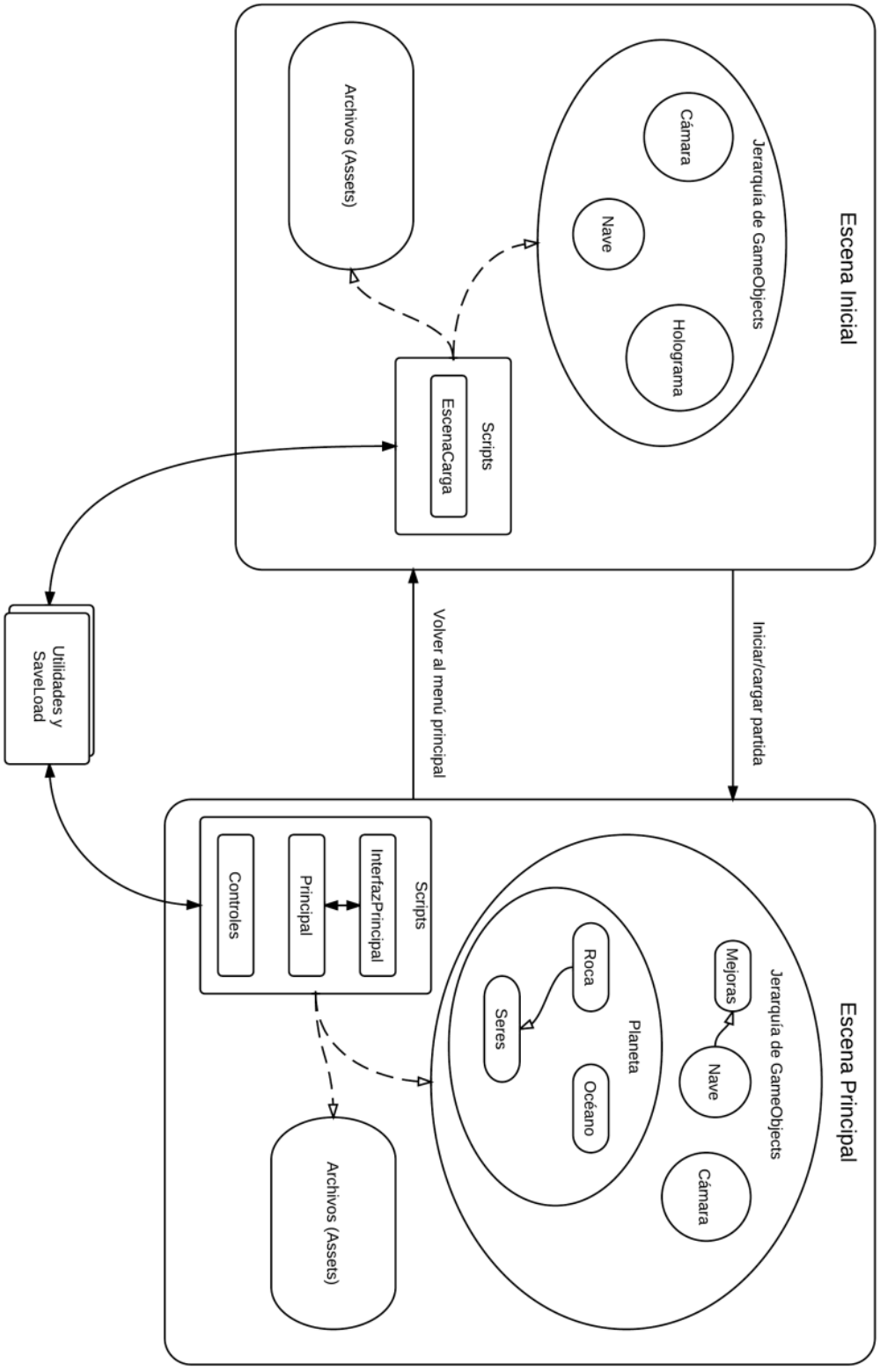
Dentro de la carpeta “Assets” tenemos multitud de carpetas que nos permiten organizar cada tipo de recurso en su lugar apropiado. Hay carpetas para guardar animaciones, sonidos, modelos, texturas, scripts, elementos de la interfaz de usuario, escenas, “shaders”, etc.

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 qué 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.

En el diagrama mostrado a continuación se puede ver la estructura general simplificada del proyecto Terraform:



Estructura general esquematizada del proyecto Terraform

4. 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.

3. Arquitectura

1. 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.

2. 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, que se representa y maneja en forma de recurso en la ventana de proyecto) 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:

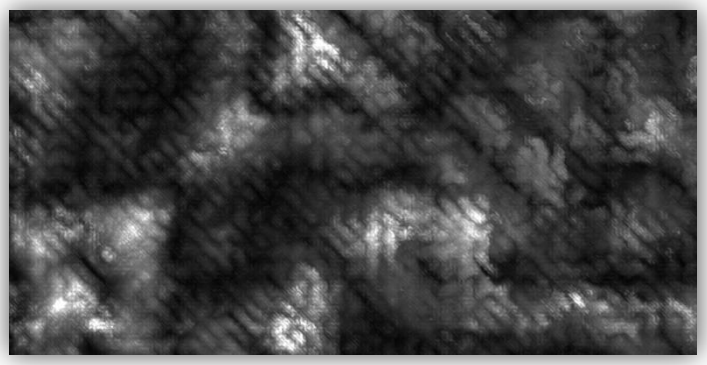
- Repetible: A pesar de ser aleatoria, queremos que dada una semilla y una entrada iguales, siempre nos devuelva el mismo resultado. Debe ser pues pseudo-aleatorio, pero con un patrón nada obvio.
- De rango conocido: Preferiblemente queremos un ruido que devuelva valores entre unos márgenes conocidos, 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 "fractionalBrownianmotion"), siendo esta variante en concreto llamada "ruido de turbulencia". Este ruido en particular hace un sumatorio de

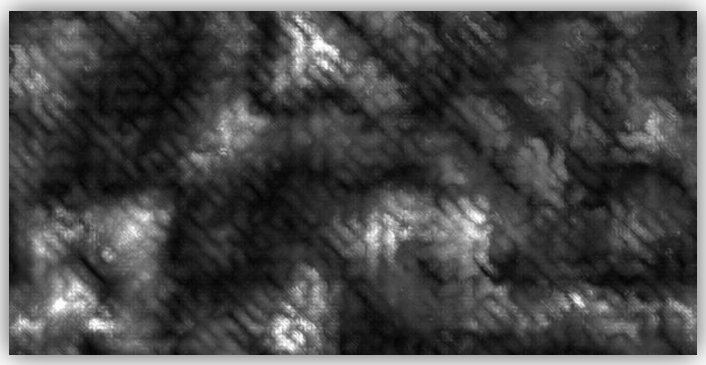
octavas de ruido Perlin con su valor absoluto, ofreciendo además la ventaja de no existir valores negativos.

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.



Textura de ruido recién creada

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.



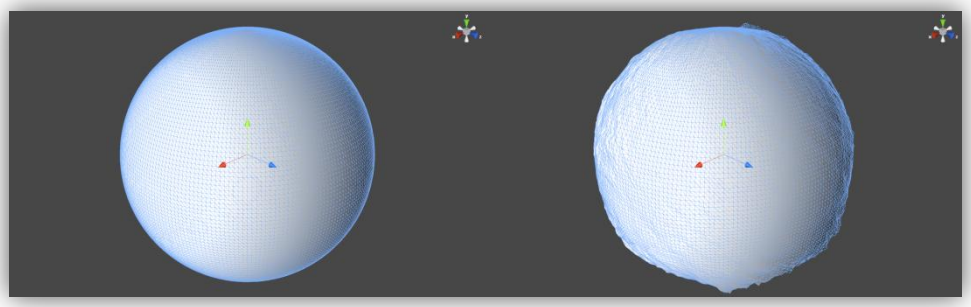
Textura de ruido con el borde izquierdo suavizado

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.

Textura de ruido “Heightmap” con polos y borde suavizados

A continuación se procede a extruirlos vértices de una esfera previamente acondicionada[4] 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.

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



Comparación de esferas antes y después de extruir los vértices

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 malla a partir del valor definido de la variable “nivelAgua” en la que representamos la presencia de agua en el planeta. Seguimos el mismo patrón que al extruir los vértices del planeta, desplazándolos en la dirección de la normal. De esta forma 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.

Finalizada esta fase y haciendo uso de esta información, procedemos a crear el soporte lógico de la vida en la escena. Para ello, creamos una matriz de objetos de tipo “Casilla”, cuyo objetivo es mapear los puntos de la textura previamente creada a un soporte lógico donde poder ejecutar los cálculos pertinentes al algoritmo de vida. Cada casilla contendrá información referida a la parcela de la textura que mapea, como puede ser los seres que en ella habitan, si hay edificios construidos, que tipo de hábitat representa, los elementos que contiene, etc. También guardan una referencia al punto en el espacio que ocupan en la superficie de la esfera.

Dado que inicialmente no hay ningún ser vivo ni edificio en el planeta, en la fase de creación debemos ocuparnos solamente de inicializar los datos referentes al hábitat de la casilla, los elementos que contiene y su posición en el espacio. Esta posición la calculamos a partir de las coordenadas UV de la textura, comprobando cada vértice de la esfera para ver si el UV asociado a él entra dentro de la casilla (es mayor o igual a las coordenadas UV del pixel de la esquina superior izquierda y menor o igual que las coordenadas UV del pixel de la esquina inferior derecha). Dado que este paso supone para cada casilla del tablero recorrer de principio a fin todo el array de vértices de la esfera, que son unos 34.000, hacemos un pre-cálculo de esta información. Este cálculo se ejecuta por primera vez con fuerza bruta (tres bucles anidados, dos para el tablero y uno para los vértices) y se guarda en un archivo en forma de índices que nos indican para cada casilla qué vértice de la esfera corresponde al centro. Este archivo es posteriormente leído, no teniendo que calcularse de nuevo. El pre-cálculo de esta información por tanto viene ya hecho cuando se instala el juego por primera vez y a menos que el tablero o la esfera cambien no es necesario rehacerlo.

El siguiente paso es calcular el hábitat de cada casilla. Hay nueve hábitats diferentes: el hábitat infranqueable, el mar, cuatro hábitats normales dependientes de la altura y tres hábitats especiales. Los hábitats dependientes de la altura son costa, llanura, colina y montaña, y como su nombre indica, su aparición depende de la altura de la casilla. Sirven como base del terreno. Por otra parte los hábitats especiales sustituyen a los hábitats dependientes de la altura y pueden estar presentes en diferentes alturas. Son el desierto, la tundra y el volcánico.

Para calcular el hábitat de cada casilla se tienen en cuenta los siguientes factores:

- La altura media de la casilla según la textura de ruido recién creada.
- La temperatura del planeta. Este es un factor elegido durante la fase de creación que tiene dos efectos: por una parte, divide el tablero en tres franjas, dos en los polos y una central, cuya proporción varía según la temperatura. A temperatura más alta, mas tamaño ocupa la franja central y menos las de los polos y viceversa. Por otra parte, este factor también afecta a las probabilidades de que un hábitat especial aparezca.
- La posición de la casilla respecto al planeta, es decir, en que franja se encuentra la casilla.
- La posición de la casilla dentro de su franja.
- La cercanía de otros hábitats especiales.

A raíz del parámetro elegido durante la creación aleatoria del planeta “nivelAgua”, se consideran todas las casillas por debajo de esa altura como hábitats marinos y no se les coloca ningún elemento. A medida que se sube del nivel del mar, dependiendo de la posición de la casilla hay diferentes posibilidades de que los hábitats sean unos u otros, y dependiendo de estos, la casilla puede contener elementos. Hay tres tipos de elementos que puede contener una casilla: ninguno, elementos comunes y elementos raros. Mientras que los elementos comunes se encuentran en diferentes hábitats, los elementos raros tienen posibilidades de aparecer únicamente en montañas o hábitats especiales.

Para que la distribución de hábitats no sea aleatoria, la cercanía de hábitats de un tipo favorece la aparición de hábitats iguales a continuación, de forma que sea más fácil dar lugar a zonas más amplias con agrupaciones de hábitats iguales.

Además, como se ha comentado, la temperatura afecta a la aparición de hábitats especiales, favoreciendo la aparición de desiertos y volcanes y disminuyendo las posibilidades de tundras a mayor temperatura.

Una vez completada la generación de las casillas, se procede al último paso de la creación del planeta: la creación de la clase “Vida”. Esta clase contiene el tablero, estructuras para manejar los seres vivos y los edificios y toda la funcionalidad asociada al algoritmo de vida.

Completado el proceso, se cede el control al método “FixedUpdate” del script “Principal”, que ejecuta el algoritmo de vida y al resto de scripts que permiten interacción, como son la interfaz principal o los controles.

En el siguiente diagrama podemos ver un resumen del flujo del proceso seguido para la creación aleatoria del planeta.

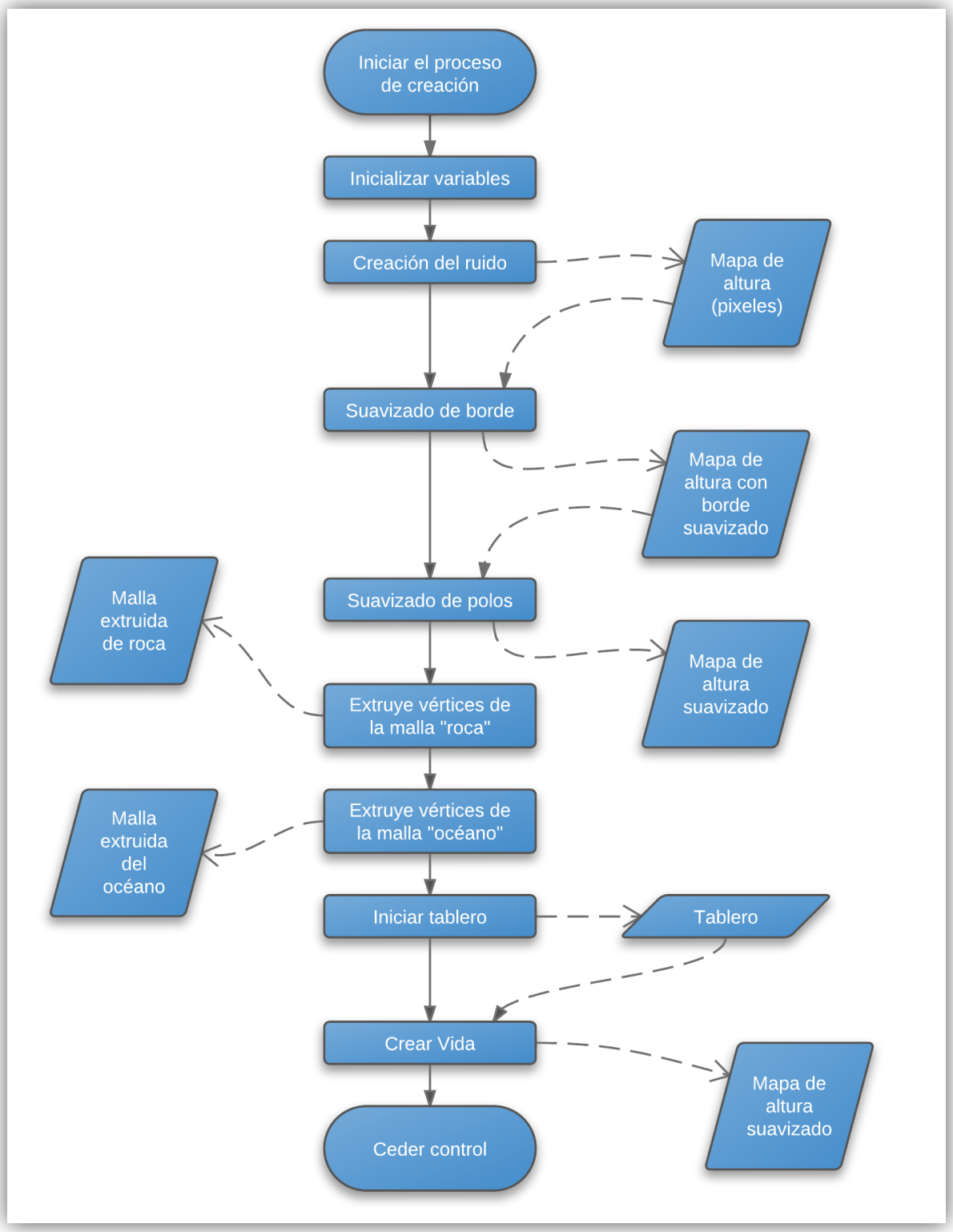


Diagrama con el flujo del proceso de creación del planeta

3. Algoritmo de vida

Introducción

Dada la naturaleza del proyecto, necesitábamos recrear una vida en el planeta que fuera dinámica y eficiente. No podía costar muchos recursos computarla y además debía ser lo suficientemente creíble y lo suficientemente aleatoria a la vez, dos características que chocan conceptualmente al menos por ahora.

Existen varios tipos de algoritmos de vida artificial ya existentes: basados en programa (con representaciones muy complejas de ADN), basadas en módulos vitales, en redes neuronales o en parámetros. Nosotros decidimos dirigir nuestros esfuerzos en esta última dirección por ser la que computacionalmente tenía un coste más bajo, creando nuestro propio algoritmo desde cero.

El algoritmo de vida artificial resultante se compone de diferentes parámetros y componentes que detallamos a lo largo de esta sección.

Etapas

Cuando nos propusimos representar la vida de un planeta nos encontramos con el principal problema de que cada uno tenía una visión muy distinta y muy irreal de lo que quería. Cada idea era perfecta en nuestras cabezas y nos costó bastante tiempo concretar una idea común y que fuera realizable.

La representación de la vida ha tenido varias etapas hasta llegar a lo que es actualmente. En un principio teníamos claro que la evolución de una serie de especies iba a ser el trasfondo de todo el juego. Con el paso del tiempo empezamos a ver que para que los animales y las plantas se comportasen de una manera menos inesperada tendríamos que recortar muchos de los conceptos que nos habíamos marcado como importantes. Así pues a lo largo del proyecto se pueden definir 3 etapas muy diferenciadas:

1. Etapa conceptual. En esta etapa después de largas discusiones que no llegaron a buen término, decidimos representar cada uno por separado la idea de cómo tenía que ser la vida de un planeta. Descubrimos que había conceptos tan dispares como permitir o no organismos unicelulares y permitir agrupaciones de varios seres en el mismo espacio lógico o tratarlos como seres independientes.

Llegamos a la conclusión de que meter seres que no se iban a ver y que por tanto no iban a interactuar realmente con el jugador era algo que nos iba a dar muchos quebraderos de cabeza y pocas satisfacciones.

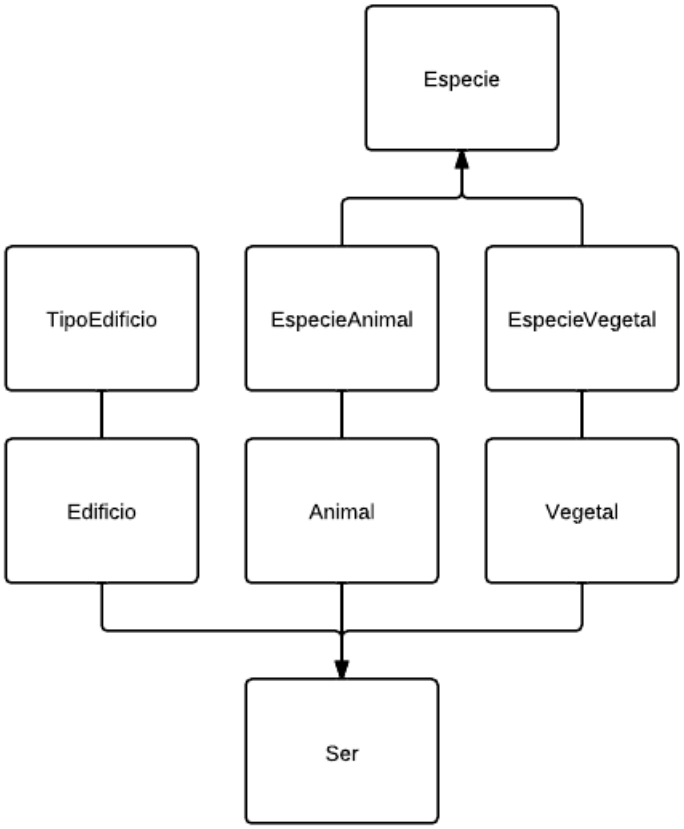
2. Primera implementación. Para la primera implementación recortamos mucho sobre las ideas que teníamos en la cabeza con la idea de hacer algo funcional básico y luego terminar de implementar las ideas que habíamos dejado fuera. De todo lo que recortamos lo más relevante fue la idea de la evolución. Obviamente no tenía ningún sentido enfrentarse directamente a ella si los propios seres sin evolucionar no se comportaban de manera correcta.
   Al implementarlo y ver que el comportamiento sobre todo de los animales no era el esperado, empezamos a darnos cuenta de que probablemente tendríamos que renunciar al concepto de la evolución porque planteaba problemas que no se podían resolver fácilmente.
   El principal obstáculo que nos encontramos es que había una limitación del tablero lógico a 128x64 casillas, lo que limitaba mucho la interacción entre animales. Si además le añadíamos que hubiese diferentes especies evolucionadas que pudieran cambiar su alimentación, se convertía en una situación muy difícil de balancear. Así que decidimos replantearnos toda la representación modificando aquellas cosas que nos estaban dando problemas y eliminando las que ya no tenían cabida.

3. Implementación final. Llegados a este punto, lo importante era que las cosas funcionasen no sólo en el tablero lógico sino visualmente. En esta etapa pudimos comprobar cómo se veían realmente los cambios que producía el algoritmo y en función de eso decidimos hacer pequeños cambios para que la relación entre el aspecto visual y el funcionamiento lógico fuera más sencilla.
   Por ejemplo tuvimos que introducir una serie de estados a los animales para identificar visualmente lo que estaban haciendo en el algoritmo y poder acoplarles animaciones que lo representasen. Otro cambio relevante fue que los animales pasaron de moverse varias casillas por turno (lo que producía colisiones visualmente) a moverse sólo una. Esto simplificaba los cálculos pero eliminaba un atributo tan básico como la velocidad. En cualquier caso lo solventamos introduciendo nuevos atributos y perfilando el comportamiento de los animales.

Estructura

La estructura del algoritmo se basa en una serie de listas de clases que o bien definen el comportamiento de un ser o bien representan a un ser en un momento concreto. Las encargadas de definir los valores predeterminados son TipoEdificio, EspecieVegetal y EspecieAnimal. Y las encargadas de contener a los propios seres en sí son: Edificio, Vegetal y Animal.

A continuación vemos en un primer vistazo como es la organización de las clases que componen los edificios, vegetales y animales:



Estructura de clases de Vida

A la hora de inicializar los valores principales del videojuego se cargan definiciones (tipos o especies) de estos seres para tener unos valores iniciales que usarán al instanciarse. Así podemos diferenciar 4 listas que definirían respectivamente a estos seres:

1. Tipos de edificios: donde se definen los distintos edificios utilizados en el juego. Sus atributos son:
    - idTipoEdificio: identifica inequívocamente a un tipo de edificio.
    - Nombre.
    - Hábitats: lista de hábitats en los que se puede insertar el edificio.
    - Energía, componentes básicos, avanzados y material biológico que se consumen al crear el edificio.
    - Energía, componentes básicos, avanzados y material biológico que consume como máximo el edificio por turno.
    - Energía, componentes básicos, avanzados y material biológico que produce como máximo el edificio por turno.
    - Metales que usa el edificio para funcionar. Pueden ser comunes, raros o ninguno.

2. Especies: donde se definen atributos comunes que se utilizarán en especies vegetales y especies animales. Sus atributos son:
    - idEspecie: identifica a una especie.
    - Nombre.
    - Nº máximo de seres por especie: indica cuantos individuos puede haber de la misma especie en un instante dado.
    - Nº actual de seres por especie: indica el nº actual de individuos que hay de esa especie que lógicamente es menor que el máximo.

3. Especies vegetales: hereda de especies y extiende sus atributos para que identifiquen sólo a los vegetales. Sus atributos son:
    - Nº máximo de vegetales: indica el valor máximo de un vegetal en una misma casilla.
    - Nº inicial de vegetales: indica el valor inicial de un vegetal de una casilla al crearse.
    - Migración local: representa la posibilidad en tanto por uno de que un vegetal pueda expandirse a una casilla colindante.

- Migración global: representa la posibilidad en tanto por uno de que un vegetal pueda expandirse a una casilla de distancia entre 1 y el radio de migración.
- Radio migración: representa la distancia máxima a la que puede migrar globalmente un vegetal.
- Habitabilidad inicial: define para cada hábitat como de habitable es esta especie en ese hábitat. Si el valor es de -1.0 la especie no podrá migrar a ese hábitat. Si el valor está entre -1.0 y 0.0 (sin incluir ninguno), la especie podrá migrar temporalmente a ese hábitat pero acabará muriendo. Y si el valor está entre 0.0 y 1.0, la especie podrá migrar y crecerá en ese hábitat a no ser que otro ser le impida hacerlo.

4. Especies animales: hereda de especies y extiende sus atributos para que identifiquen sólo a los animales. Sus atributos son:
    - Tipo de alimentación: indica si el animal es herbívoro o carnívoro.
    - Reserva máxima: cantidad de alimento del que dispone como máximo un animal. Viene a ser algo como sus reservas de energía.
    - Consumo: cantidad de esas reservas que se consumen por turno.
    - Alimento máximo por turno: indica la cantidad máxima de alimento que puede añadir a la reserva.
    - Aguante inicial: indica la cantidad máxima de turnos que puede moverse una especie animal sin necesitar descansar.
    - Reproductibilidad: indica el nº de turnos que tienen que pasar en unas determinadas condiciones para que el animal se reproduzca.

Cuando instanciamos un elemento básicamente lo que hacemos es recurrir a estas estructuras para tomar unos valores iniciales y un comportamiento predeterminado para que todos los seres del mismo tipo se comporten de una forma similar. Luego cada ser tiene sus distintos atributos que lo diferencian de sus semejantes y que se modifican a lo largo del tiempo. Las estructuras que se encargan de administrar a estos seres son:

1. Seres: donde se encuentra listado la totalidad de los seres. Es la estructura que más se utiliza ya que es recorrida en cada iteración del algoritmo. Tiene como atributos propios:
    - idSer: identifica a cada ser de manera única.
    - Posición x e y: indica la posición en el tablero lógico del ser.

2. Edificios: hereda de seres y expande sus atributos para identificar y controlar el comportamiento de los edificios. Sus atributos característicos son:
    - Tipo edificio: donde guardamos la referencia al tipo de edificio que define a este edificio. Así nos evitamos duplicar los atributos que ya contiene tipo edificio y que no varían con el tiempo.
    - Energía, componentes básicos, avanzados y material biológico que consume el edificio por turno. Se calculan en función de los valores introducidos en el tipo de edificio y un valor de eficiencia.
    - Energía, componentes básicos, avanzados y material biológico que produce el edificio por turno. Se calculan en función de los valores introducidos en el tipo de edificio, la eficiencia y el número de metales del tipo que utiliza el edificio que se encuentren dentro del radio indicado.
    - Eficiencia: valor que indica a que potencia va a estar funcionando el edificio. Los valores que utilizamos son 0%, 25%, 50%, 75%, 100%. Esto es porque al aumentar la eficiencia aumenta el radio del edificio para detectar más o menos metales y decidimos dar esas 5 opciones.
    - Radio de acción: distancia a la que se comprueba el número de metales a usar desde la posición del edificio. Los valores van en función de la eficiencia y son 0, 2, 3, 4 y 5.
    - Nº de metales: número de metales que se han obtenido al buscar en el tablero con el radio indicado por la eficiencia.
    - Matriz de acción: lista de posiciones pre calculadas al crear el edificio o modificar su eficiencia.
    - Material biológico sin procesar: uno de los edificios tiene la característica especial de recolectar por turno en su radio parte de la vida y convertirla en material biológico compactándola. Este atributo es necesario para que no se pierda entre turno y turno el material sobrante al procesar el material biológico recolectado en material biológico útil para la nave.

3. Vegetales: hereda de seres y expande sus atributos para identificar y controlar el comportamiento de los vegetales. Sus atributos característicos son:
    - Especie vegetal: donde guardamos la referencia a la especie vegetal a la que pertenece.
    - Nº de vegetales: indica cuantos vegetales hay en la casilla a la que pertenece. Se inicializa con nº inicial de vegetales al que accedemos mediante la especie y tiene como valor máximo el nº máximo de vegetales indicado también en la especie.
    - Índice del hábitat: hace referencia al hábitat en el que se encuentra el vegetal.

4. Animales: hereda de seres y expande sus atributos para identificar y controlar el comportamiento de los animales. Sus atributos característicos son:
    - Especie animal: donde guardamos la referencia a la especie animal a la que pertenece.
    - Reserva: representa la cantidad de alimento de la que dispone el animal. Si esta cantidad llega a 0 el animal muere. Igualmente tiene un valor máximo definido en su especie del que no puede pasar.
    - Aguante: número de turnos seguidos que le quedan al animal para seguir moviéndose sin necesidad de descansar.
    - Turnos para reproducción: número de turnos que le quedan al animal para reproducirse. Este contador sólo disminuye si el animal está bien alimentado (cuando su reserva está por encima del 75% de su reserva máxima). Además la reproducción sólo se producirá si el número de animales de esta especie no ha alcanzado su máximo.
    - Estado: identifica el estado actual del animal que define su comportamiento y su situación actual. Los estados posibles son los siguientes:
        - Nacer: es un estado que sólo dura un turno y que representa la creación del animal. La animación mostrará como el animal va creciendo desde pequeño hasta alcanzar el tamaño normal.

- Descansar: se produce cuando un animal se queda sin aguante durando un turno o cuando un animal ha ingerido la suficiente cantidad de comida como para sentirse lleno (más del 75% de la reserva máxima) y descansa los turnos necesarios hasta volver a tener hambre. La animación muestra como duermen los animales plácidamente.
- Buscar alimento: es en esencia la animación de mover ya que el animal sólo se desplaza en busca de alimento. La animación desplaza al animal a su nueva posición mientras se balancea para dar sensación de movimiento.
- Comer: la acción de comer conlleva haber encontrado comida al alcance (los herbívoros se alimentan en su posición actual y los carnívoros a una posición colindante). En esta animación se muestra como los herbívoros devoran plantas y como los carnívoros acechan y atacan a otros animales.
- Morir: al igual que nacer sólo se produce una vez y da con la muerte del animal. La animación muestra como el animal se cae de lado y disminuye su tamaño hasta desaparecer.

La estructura general de vida puede verse en el siguiente diagrama:

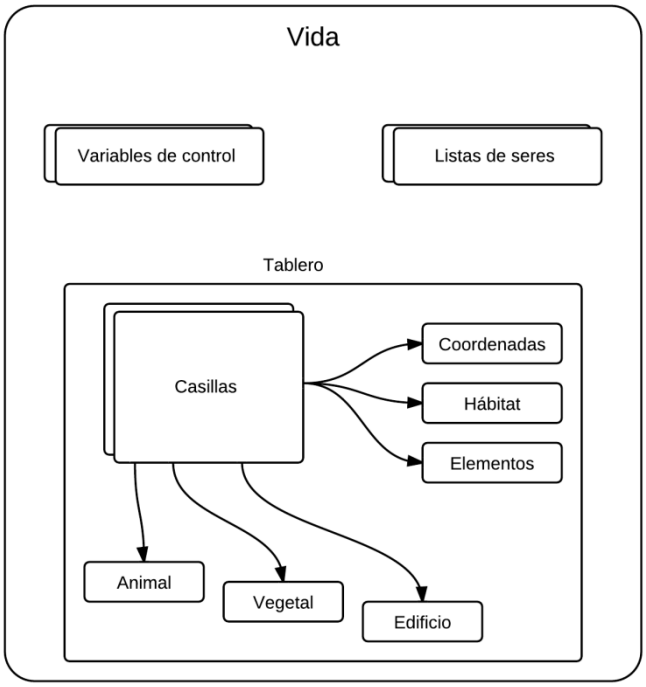


Diagrama general de la estructura de la clase Vida

Funcionamiento del algoritmo

El algoritmo se ejecuta cada 3segundos a velocidad normal. En un principio lo teníamos a 1 segundo por turno, pero en cuanto añadimos las animaciones vimos que era extremadamente rápido. Con pocas pruebas nos dimos cuenta de que 3 segundos era un valor ideal. Demás el juego cuenta con varias velocidades pudiendo ponerlo a 2x (1.5 segundos cada turno) y 5x (0.6 segundos por turno).

En cada paso se recorre toda la lista de seres y uno por uno realiza su algoritmo dependiendo del tipo de ser que sea. Para que todos tengan las mismas oportunidades reordenamos la lista de seres cada cierto número de turnos, de forma que no siempre se ejecuten en el mismo orden. Esta reordenación la realizamos mediante el algoritmo de Fisher-Yates shuffle[r8]. Según el origen de cada ser el algoritmo hará lo siguiente:

- Vegetal: consta de los siguientes pasos:
    - Reproducción o muerte: dada la habitabilidad que tenga el vegetal en el hábitat de la casilla en la que se encuentra, la variación en el número de vegetales puede ser positiva o negativa. Esto es porque como hemos comentando anteriormente con una habitabilidad negativa (distinta de -1.0) el vegetal no puede sobrevivir y va muriendo paulinamente. Si al final de este paso el vegetal tiene un nº de vegetales <= 0 el vegetal desaparece.
    - Migración local: se calcula mediante un número aleatorio y los atributos del vegetal si se va a producir una migración local, que consiste en extender el vegetal a una casilla colindante. En caso de que de positivo se intenta esta migración que sólo se producirá si la casilla elegida está libre (no puede haber otro vegetal ni un edificio) y si es habitable para la especie (habitabilidad distinto de -1.0).
    - Migración global: es similar a la migración local, sólo que se calcula una posición aleatoria con radio entre 1 y el radio de migración que tenga la especie.
    - Actualizar modelos del vegetal: a partir del nuevo valor del vegetal se calcula el nº de modelos que deberían representarlo. Y se añaden o eliminan de 1 en 1 cada turno para evitar cambios visualmente muy bruscos.

- Animal: consta de los siguientes pasos:
    - Consumo de alimento: el animal consume parte de su reserva y en caso de que esta baje o iguale 0 el animal muere.
    - Reproducción: si el animal está bien alimentado (reserva mayor que el 75% de la reserva máxima) pasa a estado descansando y disminuye sus turnos de reproducción. Si estos turnos llegan a 0 el animal intenta reproducirse escogiendo una casilla colindante. Si esta casilla está libre (no hay ningún otro animal y no hay edificio) y puede habitar en esa casilla, aparece un nuevo animal.
    - Cambio de estado: el animal modifica su estado actual por el siguiente. Básicamente si el estado es morir el animal desaparece. Si el animal está buscando comida y se queda sin aguante pasa a descansar y en el resto de los casos el siguiente estado es buscar comida.
- Edificio: en el caso de los edificios el algoritmo recopila el consumo y la producción de cada edificio y los va sumando a unos valores netos de cada recurso que se devolverán al final de cada paso del algoritmo. Como caso especial está la granja que recolecta en un radio de acción y procesa la recolección para convertirla en material biológico.

4. Efectos y parte gráfica

Introducción

"Unity 3D, (También Unity) es un Motor 3D para el Desarrollo de Videojuegos creado por Unity Technologies. ..."

Un Motor3D, ¿qué diablos significa esto?

"Un Motor 3D o GameEngine es un sistema diseñado para la creación y desarrollo de videojuegos. Los motores más populares proporcionan un framework a los desarrolladores para crearlos juegos de ordenadores y consolas.La funcionalidad primaria típicamente proporcionada se basa en varios sistemas: El motor de renderizado o Renderer, que representa gráficamente los elementos 2D y 3D, el motor físico o detector de colisiones y controladores para animación, sonido, inteligencia artificial, scripting, redes..."

En esta sección nos vamos a centrar en el Renderer. ¿Pero qué es un Renderer?

Modelos

La representación gráfica en un ordenador se basa en modelos internos de las "Escenas", es decir, representaciones matemáticas aptas para ser manejadas y operadas gráficamente. Este modelo describe tanto las formas como la posición en el espacio y los materiales de los objetos.

Para convertir este modelo en una imagen de dos dimensiones apta para ser presentada en una pantalla de ordenador lo primero es especificar un punto de vista. Es aquí donde entra en juego el Renderer. Su trabajo consiste en convertir ese modelo matemático, esos datos, en una imagen bidimensional, controlando que partes son visibles en pantalla, que objetos se ocultan entre ellos, su perspectiva, su iluminación, su apariencia y propiedades relacionadas con la luz y la interacción con dichos cuerpos.

El método más usado para modelar la geometría 3D son los polígonos. Un objeto se aproxima mediante una malla poligonal o Mesh, una colección de polígonos interconectados entre sí. Por lo general el polígono mas usado es el triangulo dada su simplicidad.

Cada triángulo se describe por las tres coordenadas de sus vértices. La desventaja de esta aproximación es que producen una superficie plana y lleva a tomar decisiones que conllevan sacrificar realismo por eficiencia o viceversa.

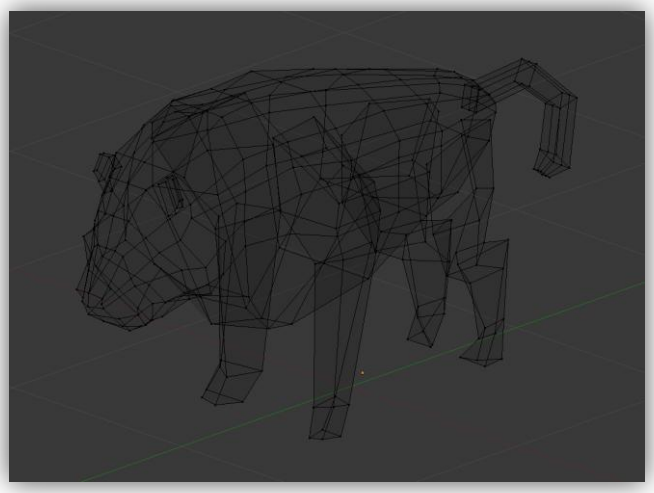


Imagen de un Mesh en Blender, un programa de diseño 3D.

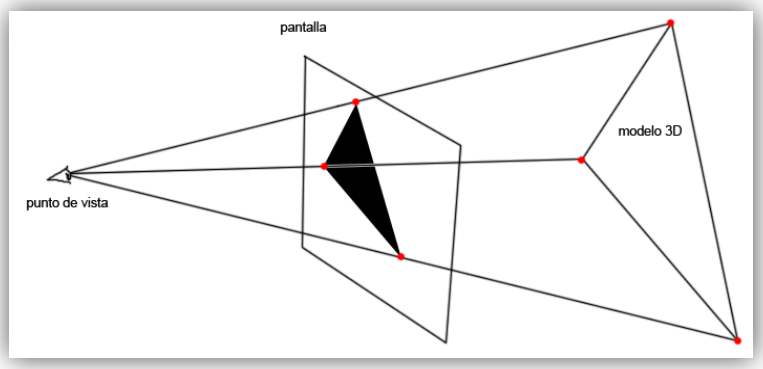
Renderizado

El proceso de convertir estos datos manejables por el ordenador a imágenes manejables por nuestro cerebro se llama renderizado, y de ello se encarga el renderizador o Renderer. Se basa en varios procesos:

Proyección y rasterización

La proyección de los objetos 3D se calcula usando perspectiva lineal. Dada la posición del punto de vista y algunos parámetros de la cámara como la perspectiva, el volumen de vista, el campo de visión, etc., es sencillo calcular la proyección de un punto en un espacio tridimensional en un plano.

Una vez calculados los vértices se "rasterizan" -se rellenan- los pixeles interiores para darle el aspecto de superficie:

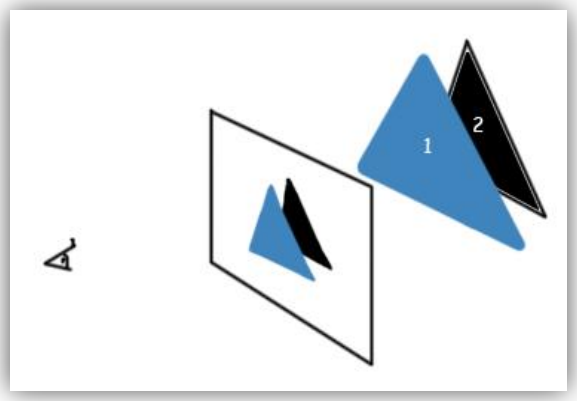


En este ejemplo se han proyectado los 3 puntos rojos usando perspectiva lineal y se ha rasterizado el triangulo rellenándolo de negro. Para mejorar la imagen cada pixel rasterizado debería tener en cuenta las propiedades del objeto en sí tales como su iluminación, reflectividad, etc.

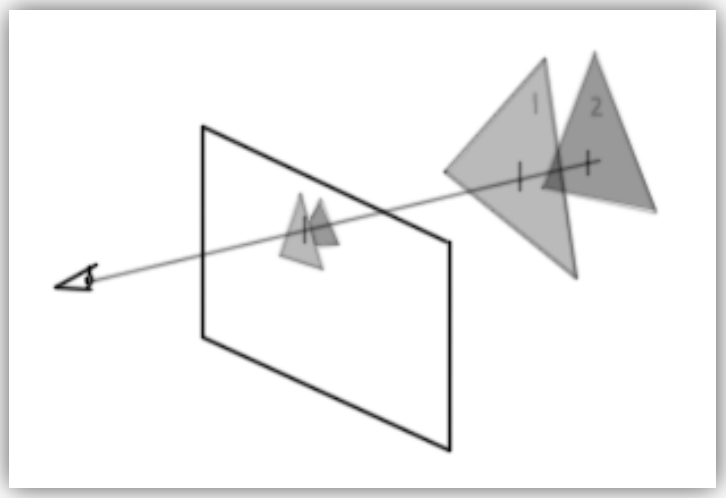
Visibilidad

Si la escena contiene más de un objeto puede ocurrir que se oculten unos a otros. Lo ideal es que se representen solo los que serian visibles, y esto se consigue gracias a ciertos algoritmos y técnicas de visibilidad. Uno de ellos es el llamado "Algoritmo del Pintor", que consiste en ordenar los polígonos de los objetos de más a menos lejano al punto de vista y rasterizarlos en ese orden. De este modo los polígonos frontales al ser rasterizados ocultan las partes de los polígonos más alejados.

Algoritmo del pintor: el polígono 2 se rasteriza antes que el polígono 1. Al rasterizarse este último la sección no visible del polígono 2 queda oculta al punto de vista y se produce la oclusión correcta.

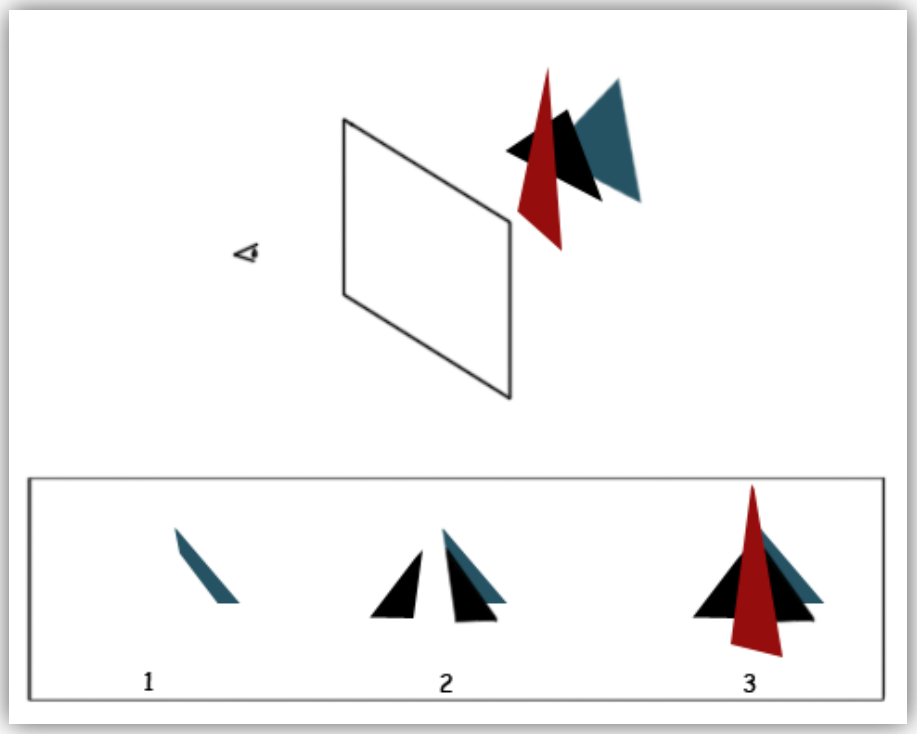


Otro algoritmo común en diseño 3D es el llamado Ray-tracing. Este método se basa en lanzar un rayo desde el punto de vista hasta cada pixel de la escena. Después se calcula la intersección entre cada rayo con cada uno de los objetos y se selecciona la primera de ellas, la más cercana al punto de vista. Este método es más costoso en cuanto a rendimiento, pero permite su aplicación a otras áreas como el sombreado y modelos de iluminación más complejos y realistas que incluyen propiedades de los materiales como la reflexión, la absorción o la emisión de luz, iluminación especular, etc.



Se manda un rayo desde el punto de vista y atravesando el pixel de la imagen a calcular. La intersección de este rayo se produce con 2 polígonos. Nos quedamos con el pixel claro ya que se produjo primero la del polígono 1.

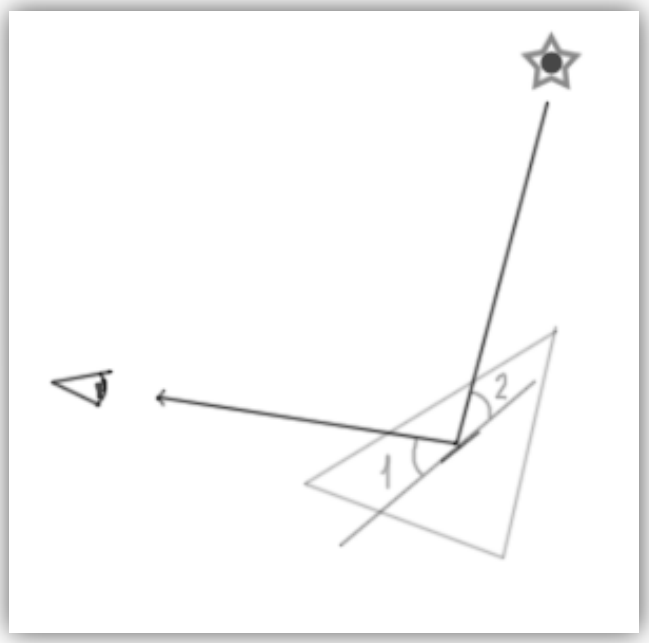
El método del z-buffer es el más utilizado y el que nos atañe, pues se usa en las aplicaciones de renderizado en tiempo real dada su velocidad. Se basa en almacenar la profundidad (z) de cada pixel. Cuando se va a rasterizar un polígono el algoritmo calcula para cada pixel la profundidad de lo ya dibujado con la del pixel del polígono en sí. De esta forma si la profundidad actual es más lejana que la del pixel del polígono actual se actualizara el pixel de la imagen con su color y profundidad. En el caso contrario tenemos que para este pixel un polígono dibujado anteriormente lo habrá ocultado y por lo tanto queda oculto y no ha de pintarse.



El método del z-buffer

Shading y materiales

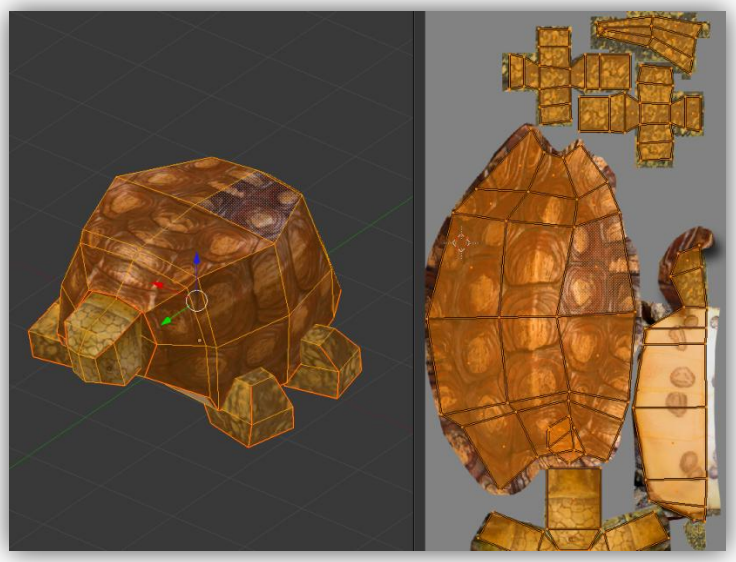
Introducir puntos de luz en la escena aumenta el realismo de ésta. Los objetos pueden ser sombreados o iluminados de acuerdo a su interacción con la luz recibida. Hay varios modelos de iluminación, a cada cual más complejo, propuestos en las diferentes publicaciones de cada autor. Todos ellos tratan de resolver con distintas prioridades la llamada Ecuación del Renderizado[r9]. Describen como se refleja la luz en el objeto en el que incide, dependiendo de las orientaciones relativas de la superficie, el origen de la luz y el punto de vista:



Modelo de iluminación simple: la iluminación/intensidad del color del pixel dependerá tanto del ángulo de incidencia de la luz en el objeto (1) como el ángulo de visión desde el que se observa (2).

UVs y texturas

Para dar aun mayor realismo existe la posibilidad de utilizar texturas en cada uno de estos polígonos del objeto. A cada una de ellos se le hace corresponder una sección de dicha textura, una imagen 2D dicha textura, y mediante cálculos se distorsiona para cuadrar con la perspectiva usada. Este "mapa" de vértices relacionados con la textura se denomina UV, y su nombre proviene de las letras U y V que hacen referencia a las coordenadas del plano que forma la textura -coordenadas normalizadas, en un intervalo de 0 a 1- en "homenaje" a las letras “X", "Y" y "Z" que hacen referencia a las coordenadas de los vértices en el espacio 3D.



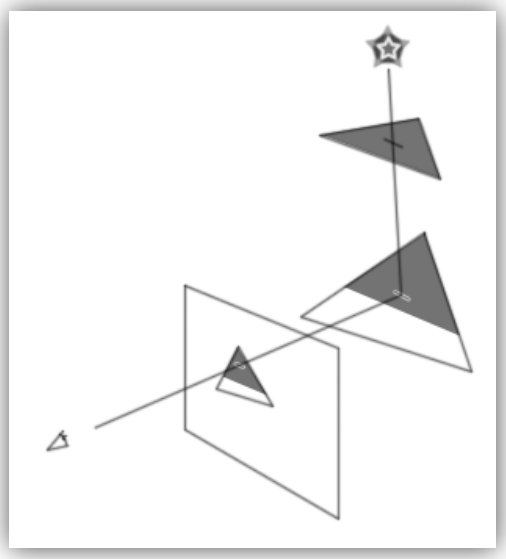
UVs del modelo de una tortuga. Cada cara queda mapeada a unas coordenadas de la textura, que se rasteriza en el espacio 3D teniendo en cuenta todas las propiedades anteriormente descritas de iluminación y sombreado, oclusiones, etc.

Luces y sombras

Los modelos de iluminación y sombreado solo toman en consideración las interacciones locales de un objeto con la luz. No simulan por ejemplo las sombras que producen o reciben, que son más complejas dado su carácter no local, ya que una sombra se produce cuando un objeto bloquea la fuente de luz en cierta medida.

El Ray-tracing, por ejemplo, puede manejar estas características de las sombras, pero requiere un cálculo de sombra por cada pixel y cada fuente de luz. El método consiste en lanzar un rayo desde el punto visible o iluminado hacia la

fuente de luz, y comprobar si interseca con algún otro objeto. De hacerlo, se puede concluir que el pixel desde el que se lanzo el rayo esta bajo una sombra.



Calculo de sombras mediante Ray-tracing. La zona oscura esta en sombra ya que el objeto superior se encuentra en el camino del rayo lanzado desde cada pixel a la fuente de luz.

Existen más técnicas para añadir aun mayor realismo a la escena, casi todas ellas relacionadas con aspectos físicos de la luz como por ejemplo la iluminación especular, o como objetos que reflejan la luz pueden actuar como fuentes de luz en sí mismo para otros objetos, o fuentes de iluminación ambiente, dispersión de la luz en la atmosfera, etc. pero su alto coste computacional los limita de momento a aplicaciones de diseño 3D y no para aplicaciones más enfocadas al tiempo real como es un juego.

Construyendo un modelo 3D

Para crear los modelos tridimensionales usados en el proyecto hemos valorado varias opciones. En un principio usamos Maya 2011 y Maya 2012 de Autodesk, aprovechando el convenio que permite a estudiantes de universidad adquirir una licencia de uso gratuito.

En un principio fue difícil adaptarse a manejarse en su entorno, no tanto por su usabilidad sino por la abstracción propia de modelar un objeto tridimensional desde un punto de vista bidimensional. Por otra parte Maya tenía varias peculiaridades que resultaban molestas, la más preocupante era que al cabo de un tiempo usándolo el programa se volvía lento y le costaba funcionar. Tras investigar descubrimos que se debía a que la memoria que el programa requería crecía y crecía sin parar, y desde los foros y tutoriales recomendaban usar una función del programa para vaciar el registro histórico y liberar la memoria. Esto hacía que el programa volviera a la normalidad, pero resultaba incomodo ver como el rendimiento iba cayendo durante el tiempo.

Por otra parte la interfaz introduce tantísimos botones que resulta fácil perderse para encontrar algo, y el funcionamiento de algunas opciones como el mapeado de texturas no funcionaba 2 de cada 3 veces. Puede que se debiera a nuestra inexperiencia, pero daba la sensación de que Maya era un programa viejo, el cual habían ido parcheando desde la antigüedad hasta llegar al Maya 2012 que teníamos entre manos, y se notaba en su fluidez a la hora de manejarse.

Buscando otras alternativas, encontramos Blender, un software de modelado 3D open source basado en Phyton. Puede que influenciados por los prejuicios del Maya, o puede que realmente fuera así, nos pareció que era mucho más amigable al uso, y todo iba mucho más fluido. Nada de parones ni trompicones, ni necesidad de liberar la historia, creación de UVs mucho más amigables… Por otra parte el bagaje de sufrir con Maya vino muy bien para coger soltura a la hora de desenvolverse con Blender. Además se ha actualizado varias veces desde que empezamos el proyecto, pero siempre con mejoras que no solo no comprometían el desarrollo sino que mejoraban el flujo de trabajo.

Tomando como referencia algunos de los múltiples tutoriales que hay en la red sobre creación de modelos y mapeado de texturas creamos la mayoría de los assets 3D de los que disponemos ahora mismo.

No todo fueron maravillas por parte del Blender. En teoría según la documentación de Unity, los archivos .blend se importan a Unity directamente, convirtiéndolos internamente a .fbx en el importer, la utilidad para modificar los parámetros de los assets externos. No fue nuestro caso, teníamos la versión 3.4.2f3 de Unity y la 2.62 de Blender y el sistema de importado automático no funcionaba. Tuvimos que usar la función de exportar a .fbx manualmente de Blender, teniendo en cuenta que los sistemas de coordenadas de Blender y Unity eran distintos: Unity usa Y como dirección para la altura y Z como dirección para la profundidad, mientras que Blender usa Z para la altura e Y para la profundidad. A consecuencia de esto todos los modelos tienen una rotación en el eje X de 270 grados, pero teniéndolo en cuenta no es problema. El problema fue descubrirlo.

Shaders

Se llama comúnmente Shader a un programa escrito en un lenguaje de alto nivel, pensado para ser independiente del hardware donde se ejecuta, que opera en las GPUs para representar los efectos gráficos. Esta tecnología es relativamente reciente y permite a los programadores utilizar las funciones que ofrecen las tarjetas gráficas recientemente y que antes estaban limitadas a la CPU. Al ser las GPU arquitecturas especialmente diseñadas para estas funciones su rapidez y eficiencia supera en gran medida a las CPUs en este dominio, dándonos una gran versatilidad.

Los shaders describen las características de pixeles y vértices. se dividen en varias categorías, y cada una de ellas hace uso de ciertas partes de la arquitectura de las GPUs. Los shaders están divididos en varias subrutinas. Los Vertex Shader especifican la representación de los vértices del modelo sobre el que se aplican (posición, coordenadas de texturas, colores, etc. mientras que los relativos a los pixeles, los Pixel Shader, concretan el color, la profundidad y la transparencia u opacidad de cada uno. Se llama a Vertex Shader una vez por cada vértice del modelo, de forma que por cada vértice que entra en la GPU sale un vértice en la escena modificado. Estos vértices forman superficies que se renderizan a continuación como una serie de texels (bloques de memoria) que se mandaran a la pantalla convirtiéndose en pixeles, las unidades de representación gráfica.

Principalmente hay tres tipos de shader que se usan con normalidad. Aunque algunas tarjetas gráficas antiguas tenían módulos dedicados especialmente a cada una de estas funciones, en la actualidad cada unidad de procesamiento puede operar cualquiera de estos tipos de shader, permitiendo hacer un uso más eficiente de la potencia de cálculo disponible.

Los Pixel shaders, también conocidos como Fragment shaders, calculan el color y otros atributos de los pixeles. Programándolos se puede representar un color uniforme, aplicar iluminaciones, reflejos y sombras, variar la intensidad o saturación del color, aplicar Bump mapping, opacidad y otros efectos. También pueden alterar la profundidad de los pixeles a la hora de aplicar el z-buffering, o cambiar todos estos valores dependiendo del Angulo de visión. Aun así no se consiguen efectos muy complejos porque operan en un rango muy local, el pixel, sin conocimiento de la geometría de la escena circundante.

Los Vertex shaders se aplican una vez por cada vértice del modelo. Su propósito es transformar la posición 3D del vértice en el espacio virtual a la coordenada 2D en la que aparecen en pantalla, teniendo en cuenta el valor de la profundidad de cada pixel. Pueden manipular propiedades como la posición, el color y la coordenadas del mapeado en la textura, pero no crear nuevos vértices.

Los Geometry shaders son relativamente modernos, introducidos en Direct3D 10 y OpenGL 3.2; Este tipo de shader puede generar geometría dinámicamente que no existe en el modelo original, tales como puntos, líneas o triángulos. El Geometry shader se ejecuta después de la etapa Vertex y toman como entrada lospolígonos iniciales y pueden admitir geometría cercana en su entorno como polígonos adyacentes.

Algunos usos de este shader incluyen la teselación, el cálculo de volúmenes de sombras y el cálculo de reflejos.

¿Cómo se hace? la Tubería Gráfica

A la hora de representar un a escena, la CPU manda instrucciones del lenguaje de programación de shaders y los datos del modelo a renderizar a la GPU de la tarjeta grafica.

Una vez allí, en el vertex shader se calcula la geometría y la iluminación. Si la GPU dispone de un GeometryShader, se modifica el modelo añadiendo o reduciendo los vértices del modelo.

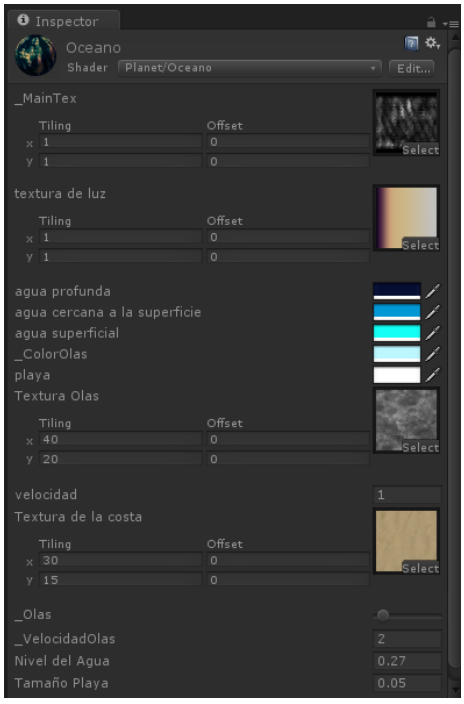
Se triangula la geometría (se subdivide en triángulos).

Se dividen los triángulos en Quads de pixeles (cuadrados de 2x2 pixeles).

La tubería aplica estos pasos para convertir datos tridimensionales abstractos en datos bidimensionales representables en los soportes informáticos. Esta representación 2D es conocida como matriz de píxeles o "frame buffer".

Materiales

Los shaders en Unity se implementan por medio de los llamados Materiales. Todo GameObject en Unity que se renderice tiene asociado a un material. Mediante éste se ajustan las propiedades del shader que este asociado al material en la ventana del inspector.



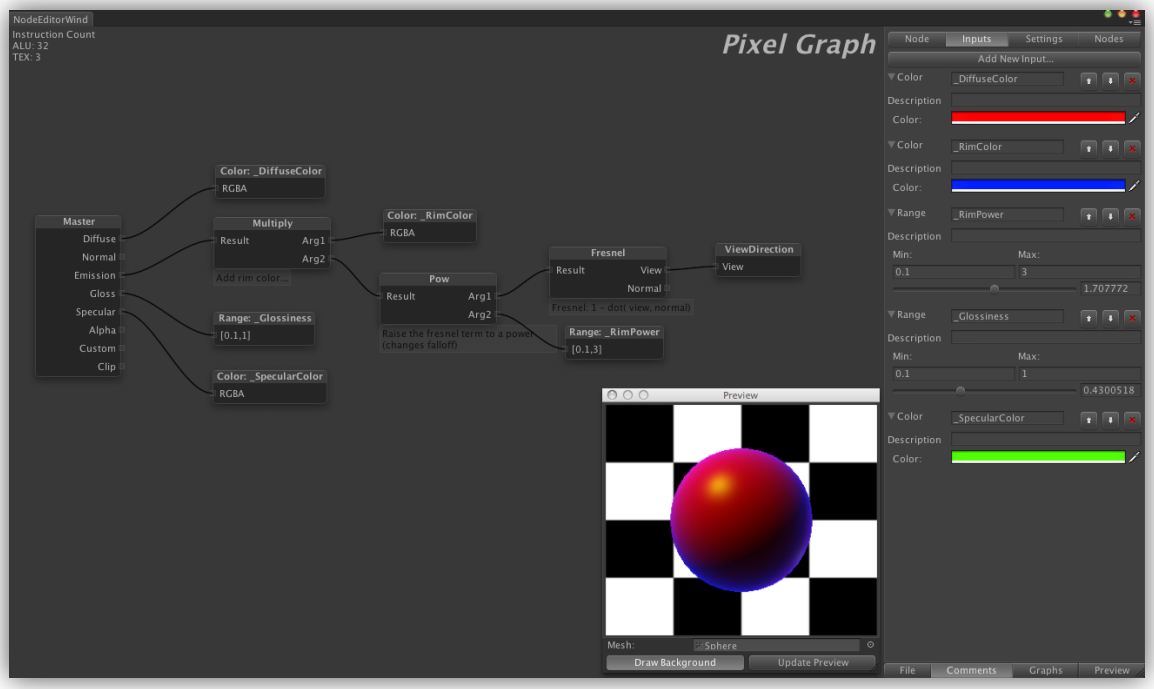
Un Material se asocia a un objeto renderizable y sirve para controlar las variables del shaderéste utiliza.

Estas propiedades se pueden modificar tanto en edición como en ejecución. Una propiedad de estos materiales que hemos usado a nuestro favor es que al crear varios GameObject con el mismo material, podemos modificar las propiedades de ambos objetos modificando el material que comparten. Esto ha sido muy útil a la hora de plantear los filtros.

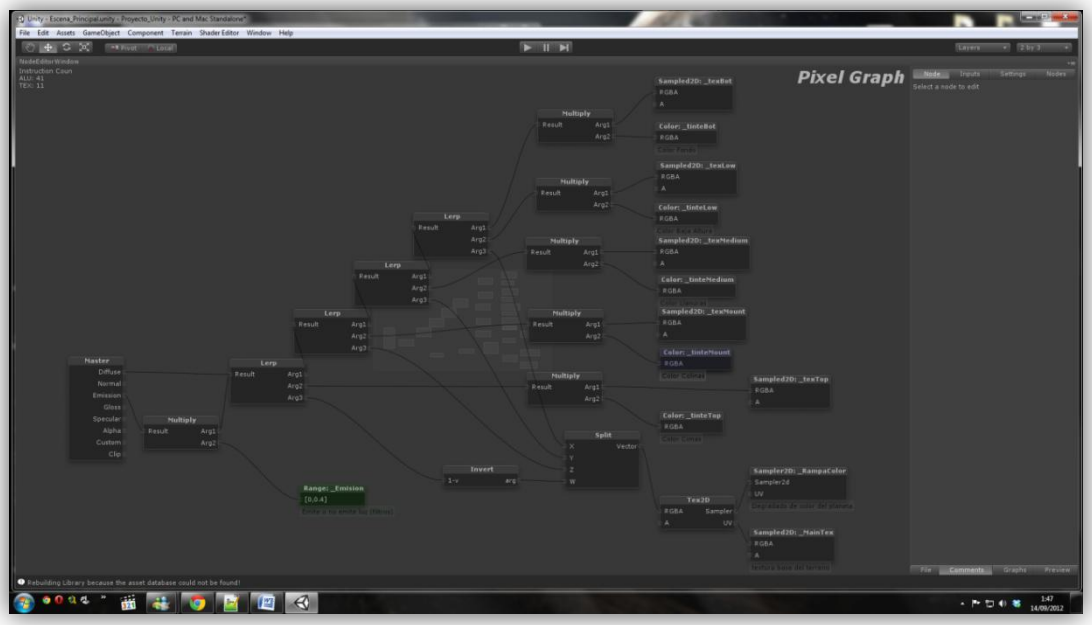
Construcción

Para programar los shaders del proyecto hemos usado un script/plugin que encontramos en los foros de Unity y hacia el trabajo muchísimo más fácil. El programa, Strumpy's Shader Editor[r12], es un generador de código automático que usa nodos y enlaces entre ellos para esquematizar el shader en lenguaje gráfico.

El plugin adjunta un manual de uso y archivos ejemplo que pueden servir de gran ayuda para aprender el manejo de la herramienta y el funcionamiento de la tubería grafica en sí misma.



Una captura de ejemplo extraída del foro del plugin.



Uno de los diagramas que codifican los shaders del proyecto.

Shaders del proyecto

Escena inicial

La escena inicial es una muestra del poder y la simplicidad de los shaders. El efecto conseguido es muy vistoso y se basa en una idea muy antigua. Consiste en un foco de luz con un "Flare", una imagen 2D que representa una fuente de iluminación, y una esfera que envuelve a la nave, distorsionada para dar efecto de perspectiva, con un shader que combina 2 texturas mediante una interpolación. Cada una de ellas se anima mediante una función UVPan, que va moviendo las coordenadas de los UVs por cada una de ellas, creando un efecto de animación. El truco está en que una se mueve más lentamente que la otra, lo que da una sensación de profundidad. Este método se inspira en los juegos 2D antiguos donde para crear la ilusión de profundidad en una escena se creaban varias capas de assets, por ejemplo unas montañas lejanas, y una colina más cercana, y el efecto se conseguía moviendo ambas capas a diferente velocidad según el usuario cambiaba su punto de vista para recrear la perspectiva.

Capturas estáticas de distintas combinaciones de texturas y valores en el shader animado de la escena inicial.

El shader soportaría por ejemplo cambiar paulatinamente el valor de interpolación con el tiempo, o cambiar las velocidades de las texturas en tiempo real. También podrían cambiarse aleatoriamente las texturas actuales por algunas de las otras texturas que hay aptas para esta escena para que, como mejora, cada vez que se inicie el juego diera la sensación de que la nave va viajando por otra región del espacio completamente diferente.

Hay dos shaders más en la escena principal, y aparecen a la hora de buscar un planeta a "terraformar". Su carácter es informativo y esquemático y no son muy elaborados. El shader de la roca toma la textura de relieve y extruye el Mesh (a diferencia de la escena principal, se hace por shader en la etapa vertex) y le aplica una rampa de color para darle un aspecto más representativo a las diferentes alturas. La rampa de color interpola el valor de color de un pixel con el valor de color correspondiente en la textura proporcionada como rampa. De esta forma se obtienen imágenes como las siguientes:

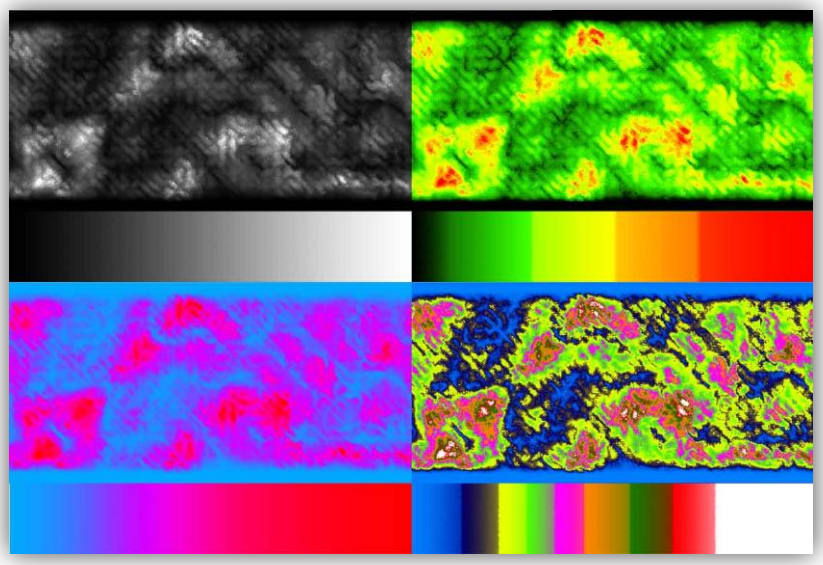
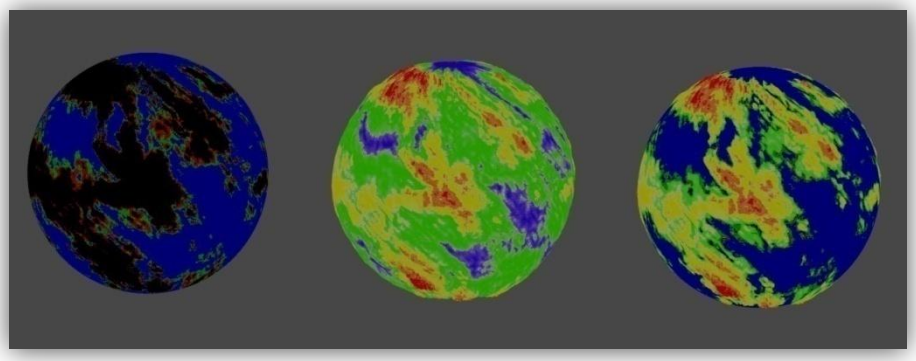


Imagen en escala de grises y con rampa de color aplicada

El shader del Océano de esta etapa inicial es una simplificación del shader del Océano de la escena principal, del cual obviamos los detalles estéticos y nos centramos en la separación de las secciones del océano en función de la altura y los valores del tamaño de las playas y la altura del nivel del mar. Al estar extruido por shader al cambiar el parámetro de nivel del agua en las opciones de creación del planeta los cambios se pueden visualizar en tiempo real, ya que la modificación ocurre en la GPU.



El shader del agua y el shader de la roca en escena inicial, y ambos combinados

Escena Principal

En la escena principal nos encontramos con varios elementos. Como componentes básicos de una escena tenemos una o más fuentes de luz y el Skybox:

La fuente de luz es una luz direccional (un plano invisible que emite rayos de luz paralelos en una única dirección) y va rotando y orientándose hacia el planeta. Inicialmente se planteo como una luz posicional (un punto de luz en el espacio que emite rayos de luz de forma omnidireccional) y la idea era que el planeta rotara alrededor de este punto, que hacía las veces de estrella, emulando el comportamiento heliocéntrico que se da en la realidad. Este comportamiento llevaba consigo múltiples quebraderos de cabeza desde el punto de vista técnico ya que al estar moviéndose el planeta en todo momento las posiciones de los objetos que en el iban a posicionarse también habrían de moverse, con lo que nos decantamos por un modelo geocéntrico para simplificar cálculos: el planeta es el origen y la luz va rotando a su alrededor.

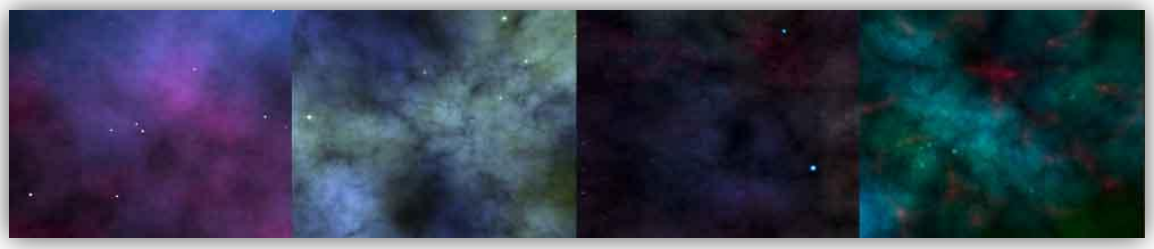
Los skyboxes en Unity

Un skybox es un componente esencial de la escena. Como su nombre sugiere, es como una caja vista desde dentro, que cubre la escena a modo de cielo y hace su función. Cuando la cámara se orienta el skybox rota con ella y su perspectiva cambia, dando la sensación de estar envueltos por él.

En cada cara de este cubo hay mapeada una textura que encaja perfectamente con las adyacentes, de forma que la transición de una a otra resulta inapreciable.

Texturas de un Skybox desplegadas.

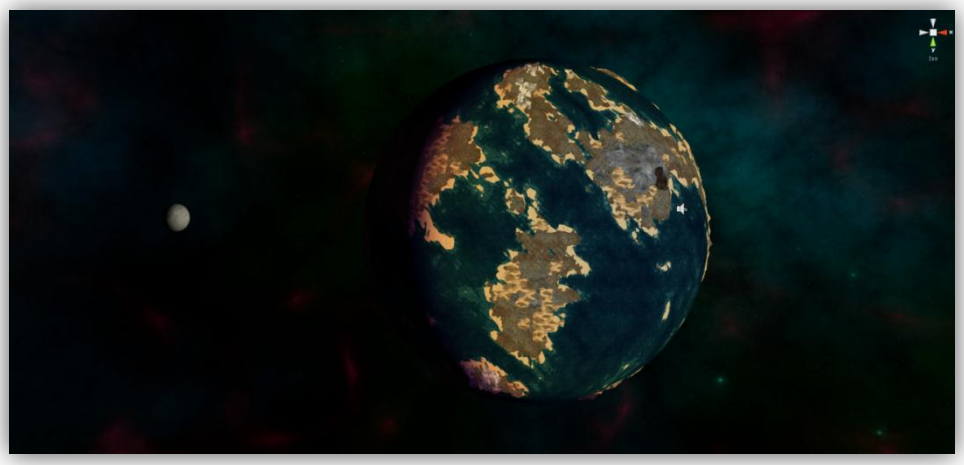
Para construir estos skyboxes primero utilizamos programas de diseño grafico como Photoshop creando cada una de las 6 texturas de forma que cumplieran estos requisitos, pero comprobamos que el trabajo era costoso en cuanto a tiempo y los resultados eran mejorables. Tras buscar concienzudamente encontramos un programa que las generaba aleatoriamente mediante capas de ruido Perlin y billboards de imágenes. Dicho programa nos resulto de gran utilidad y en una fracción del tiempo que nos llevo construir uno "a mano" creamos 4 skyboxes mucho mejores. Desde aquí un agradecimiento a Alex Peterson, el creador de Spacescape[r11]:



Los 4 skyboxes disponibles. Creados con Spacescape.

El Planeta

El planeta es el protagonista de la escena principal del juego y su aspecto grafico ha de reflejar las características estéticas e informativas de forma que hagan su papel de tablero jugable.



El planeta.

El planeta se compone de dos esferas: el GameObject Roca y el GameObject Océano. Ambos objetos son hijos de un contenedor "earth-moon" que hace que compartan ciertas propiedades como la posición, rotación y escala en el espacio 3D además de servir de organización jerárquica.

La Roca

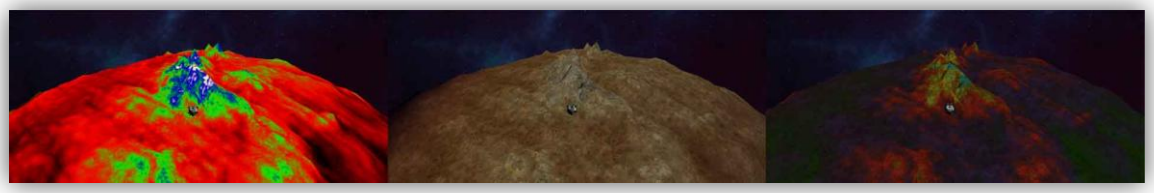
El GameObject Roca es básicamente una esfera de 32753 vértices. Este número viene dado por la limitación de Unity de no permitir Meshes poligonales de más de 65000 polígonos. Al parecer este límite viene dado por el motor para que a la hora de construir un juego los usuarios no abusen de meshes poligonales excesivamente detallados que pudieran dar lugar a perdidas de rendimiento. Como veremos más adelante, hay formas de superar estos límites, pero se nos ocurrieron en una etapa tardía del proyecto en la cual había cuestiones más relevantes que atender.

Unity tiene una característica interesante que permite la inclusión de varios shaders en el mismo Mesh haciendo uso de los llamados submeshes. El GameObject renderiza a modo de capas cada uno de los materiales de su lista de materiales disponibles de forma que materiales superiores ocultan a los inferiores si se da el caso. Haciendo uso de esta característica el GameObject Roca está construido en 3 capas:

Shader de Roca

El shader de la base de roca utiliza la textura de relieve combinada con una rampa de color con los 4 canales básicos RGBA y el negro. Utilizando cada uno de estos canales y asignándoles una altura obtenemos una textura diferenciada por niveles con transiciones suaves. Después se interpola una textura estética por cada canal, y otra como base para la ausencia de color.

De esta forma hay 5 texturas interpoladas: Negro (ausencia de color), Rojo, Verde, Azul y Alfa (representado como blanco en la imagen). A su vez todas estas texturas tienen un tinte, para colorearlas en caso de necesidad.



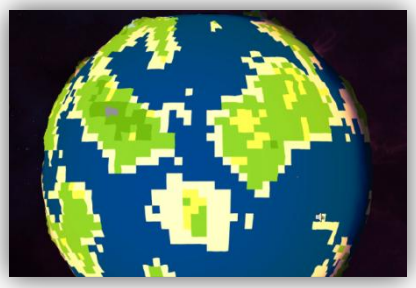
Shader Roca sin texturas, con texturas y texturas tintadas.

Shader de Hábitats

El shader de hábitats utiliza dos texturas: la textura de hábitats estética y la textura de hábitats esquemática. Estas texturas se calculan en la creación del planeta en el mismo método y de forma similar, pero su resultado es distinto. En el caso de la textura de los hábitats esquemáticos, se pinta la casilla correspondiente a cada celda del tablero en la textura con un color que corresponde a un hábitat especifico.

La textura estética en cambio se crea de forma que, usando un conjunto de pinceles de los cuales elegimos uno al azar cada vez que pintamos una casilla para atenuar la sensación de cuadriculado, solo pintamos las "mutaciones" de hábitats especiales, es decir, desierto, tundra, volcánico y costa. Así sólo necesitamos pintar cada una de ellas con un canal, y utilizando el mismo método de interpolación de texturas, asignar una textura en mosaico por canal.

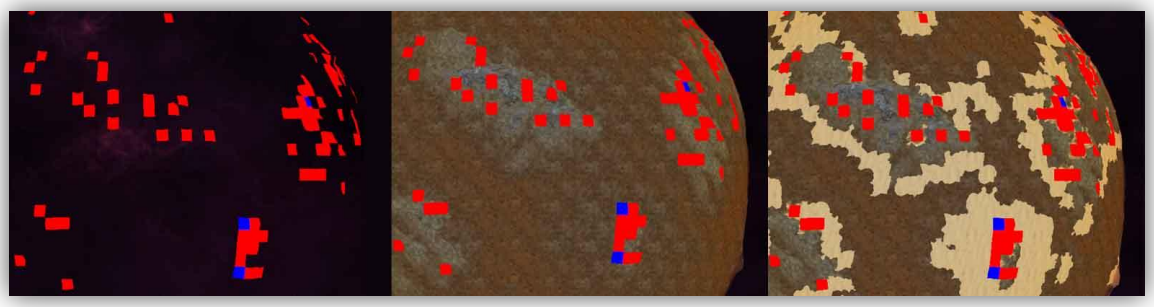
El shader tiene un interruptor FiltroOn que se utiliza a modo de filtro, de forma que con una o-exclusiva podemos alternar entre las 2 texturas, estética y esquemática, e iluminando suavemente la esquemática para que se represente aún en ausencia de fuentes de luz conectándola al Emission. Además de la interpolación, cuando el filtro está inactivo el color negro se recorta con un nodo clip (que también depende de FiltroOn), de forma que las partes no pintadas de ningún color no se rasterizan, con lo cual al superponer esta capa sobre la capa del terreno base solo ocultamos las zonas donde los hábitats han "mutado" de su variante por defecto a los especiales.



Textura de hábitats Esquemática y textura de hábitats Estética.
Emission = 0 + FiltroOn * Esquemática , Difuso = FiltroOn * Esquemática + ¬FiltroOn * Estética

Shader de Recursos

El shader de recursos utiliza una textura creada a la par que las relacionadas con la capa hábitats, de forma que se pinta la zona de la textura correspondiente a una celda del tablero de Rojo si hay metales comunes en la casilla y de Azul si hay metales raros. Cada canal tiene asociado un interruptor, de forma que se pueden visualizar los canales independientemente unos de otros, esto es, solo metales comunes, solo metales raros, o ambos. La zona de la textura sin color también se recorta con un nodo clip relacionado con cada uno de los interruptores de filtros. Existe la posibilidad de representar y filtrar el canal verde y el alfa, en caso de que quisiéramos añadir mas información de recursos, añadiendo otros tipos, etc. Todos los filtros van también al Emission para que se aprecie la información que transmiten aun en ausencia de iluminación.



ShaderRecursos en el espacio, sobre Roca y sobre Hábitats.

Combinando estas 3 capas obtenemos la Roca del planeta, sobre la que luego superpondremos el Océano, de forma que las partes por debajo del nivel del mar quedaran cubiertas y las partes que sobresalgan mostraran el aspecto que le hemos dado con este shader.

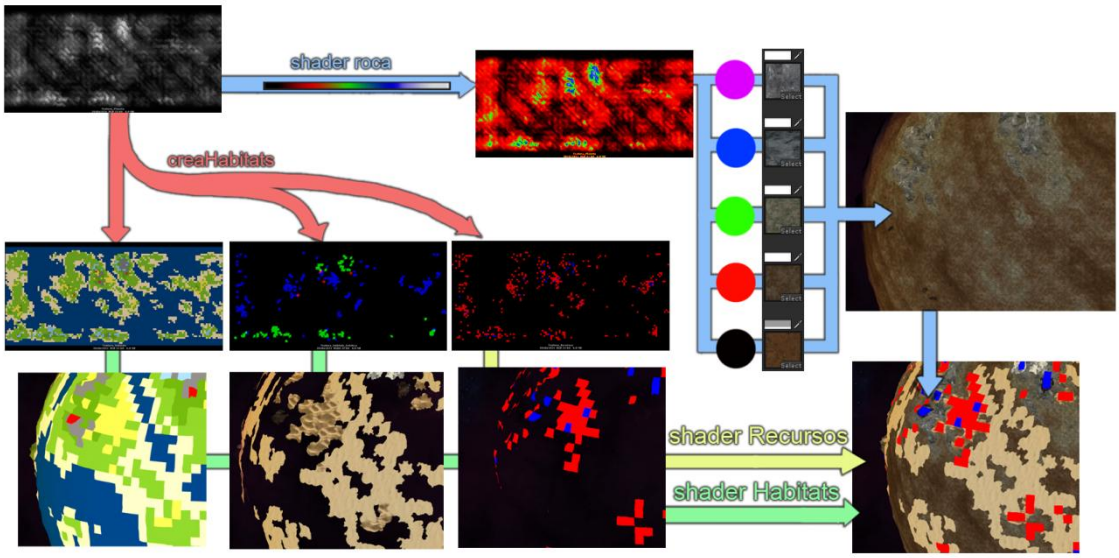


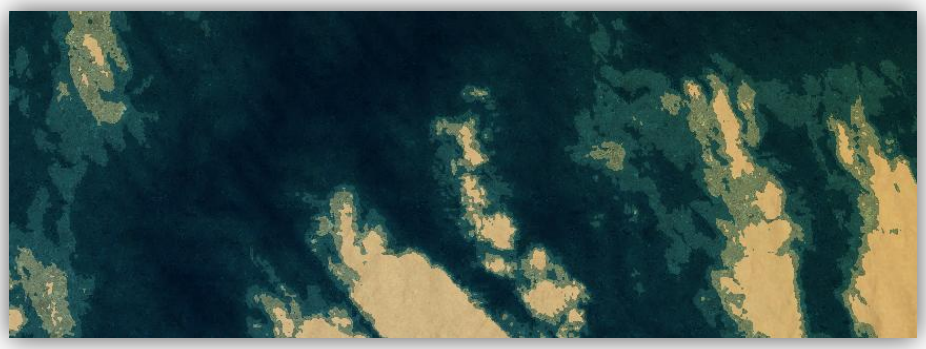
Diagrama de construcción de Roca.

El Océano

El Océano es otra esfera de 32753 vértices situada en la misma posición que Roca. Podría parecer excesivo y no sería raro pensar que no requiere en principio el mismo nivel de detalle que Roca al no extruirse de forma tan irregular, pero comprobamos que al intersecar entre ambas esferas las líneas de la costa eran más suaves a mayores niveles de detalle, como es lógico.

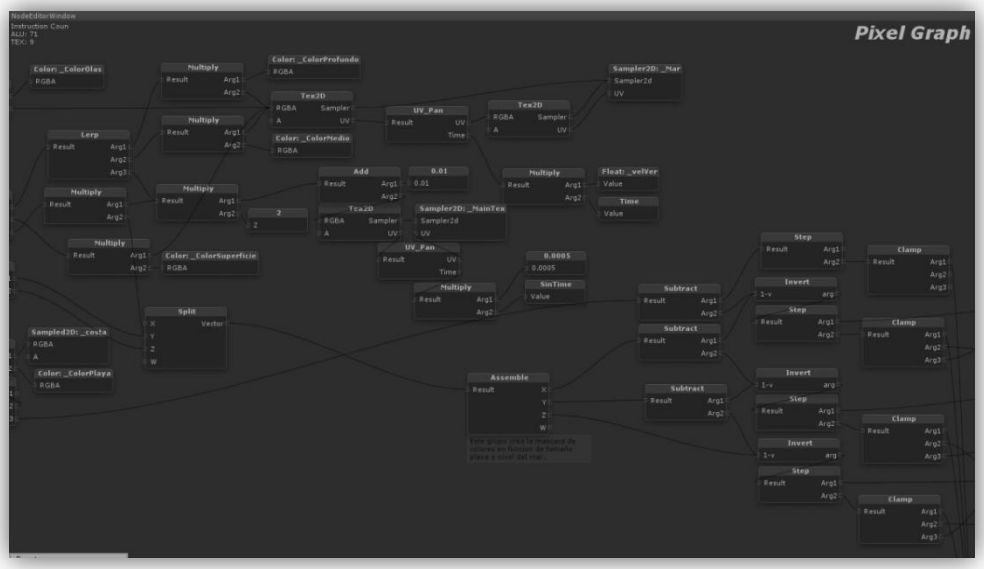
El Océano se construye a partir de la textura de relieve y dos valores: el tamaño de las playas y el nivel del agua. Mediante una serie de funciones de escalón y sustituciones de colores se separan las diferentes secciones del océano: el agua

profunda, las por llamarlas de alguna forma "plataformas continentales", la línea de playa y la tierra. A cada una de estas secciones codificadas por colores se le aplica un efecto particular: Al agua profunda se le aplican dos colores y se interpolan con referencia el mapa de alturas, lo que produce un efecto de profundidad directamente relacionado con la profundidad del planeta. A su vez se anima esta zona con un nodo UVPan para dar la sensación de que esa masa de agua se mueve sutilmente bajo la superficie y crea efectos de refracción de la luz.



Detalle de aguas profundas, costa y playas. .

Al agua poco profunda se le aplica otro color a elección, interpolando desde el color más cercano a la plataforma continental de la sección de agua profunda al color elegido como agua superficial. A esta zona se le aplica una subdivisión animada que crea un efecto de olas, de las cuales se puede elegir también el color. Este efecto se consigue animando mediante el shader utilizando como entrada el tiempo de juego (Time) y aplicándole una función seno (SinTime) para que oscile. Todas estas secciones son consideradas agua, y como tales se les aplica una textura de relieve a modo de olas, la cual se anima a su vez con otro nodo UVPan pero esta vez animando los colores de dicha textura, de forma que oscilan entre oscuro y claro, dándole un aspecto cristalino. La zona donde empieza la arena de la playa en cambio carece de esta animación y es mate, lo que la hace parecer seca en contraste con el agua. A esta zona se le interpola una textura de arena de playa.



Una sección de la etapa superficie del shader de Océano. Es complejo.

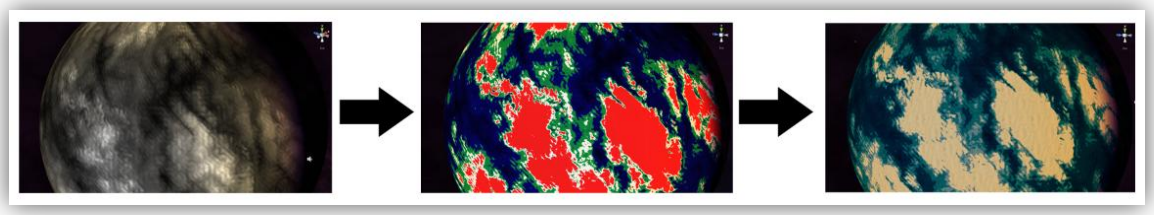
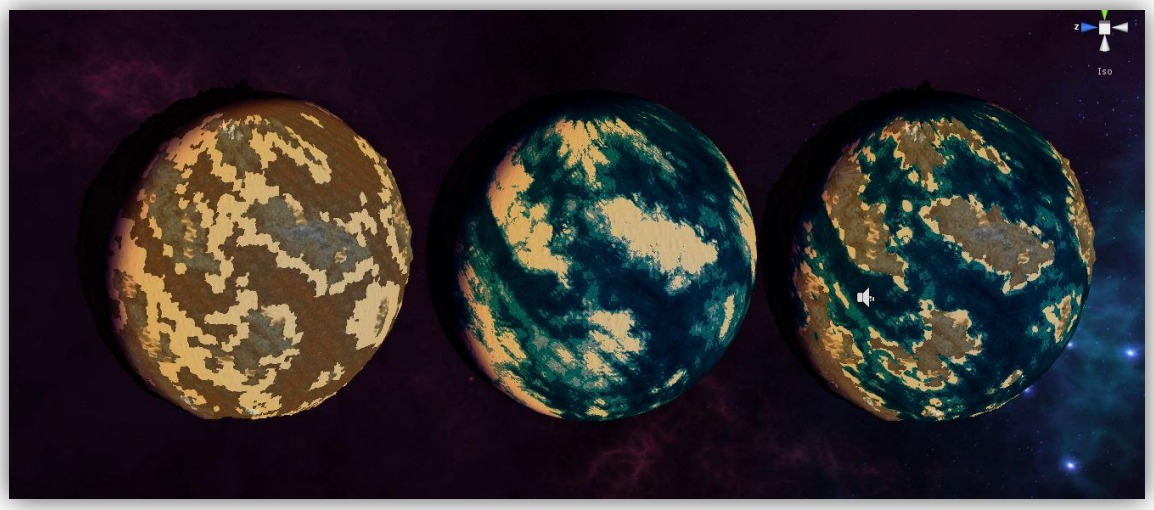


Diagrama de construcción de Océano.

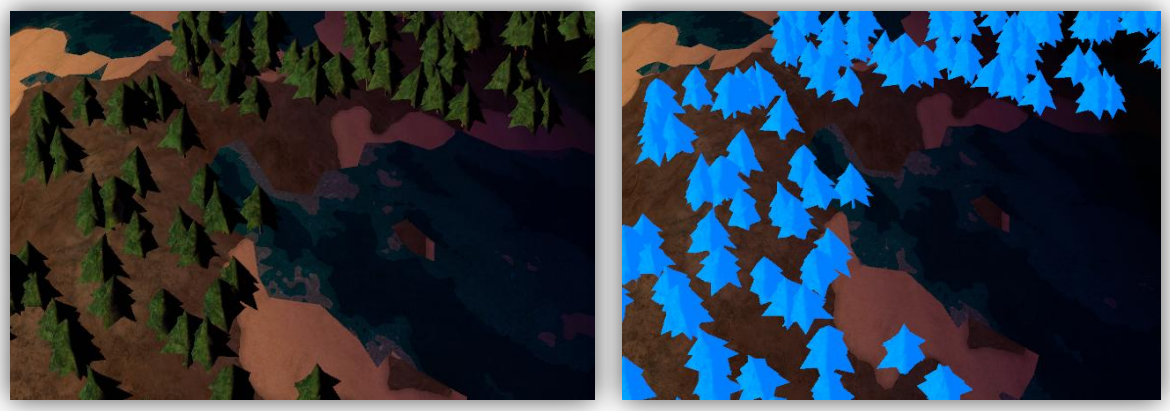
Combinando estas dos esferas y gracias a la extrusión de ambas en la misma escala obtenemos el planeta de la escena principal:



Roca, Océano y Ambos en la escena principal

Objetos con filtro y otros objetos

Las piezas de la nave, los animales, las plantas, y los edificios tienen todos un shader similar. La base para todos estos es una textura base que da color a los objetos, y una textura para añadir efectos de iluminación en la propiedad Emission. Por otra parte, para permitir que el usuario destacara los animales o plantas que le interesara encontrar con mayor facilidad mediante filtros, este shader tiene una propiedad numérica float que funciona a modo de "interruptor", y un color a modo de tinte. Si el interruptor esta encendido, la textura se sustituye por el color del tinte, y se ilumina el objeto con este color, para visualizar el objeto elegido aun en ausencia de luz. Esta mecánica se basa en una simple o-exclusiva:



Difuso, Emisión = FiltroOn * Tinte + FiltroOff * Textura

Para modelar las plantas fue suficiente con este shader. Los animales tienen, aparte de lo mencionado, una animación por shader en la sección vertex, que oscila -con SinTime- modificando la posición en el espacio siguiendo como dirección el vector normal de cada vértice, lo que crea un efecto de respiración. Dos valores de números reales controlan esta animación: la frecuencia de la "respiración", que afecta a la multiplicación del nodo Time antes de entrar en la función seno y la "corpulencia" que no es más que el intervalo en el que oscilan los vértices dentro de la normal, y da la sensación de que el animal toma más o menos aire al respirar. Con esta sencilla animación podemos por ejemplo recrear un conejo que respira a gran velocidad pero bocanadas cortas, o un oso que respira lentamente pero aspira y expulsa gran cantidad de aire con cada bocanada.

Los edificios parten del shader de las plantas, aunque en un principio iban a ser filtrados del mismo modo, al haber pocos y ser bastante distintivos en su estética se descarto la necesidad. Aún así el shader actual de los edificios permite esta posibilidad y podría ser implementada en un futuro si se añadiesen más edificios o interesara filtrarlos. Como añadido este shader tiene un hueco para texturas de NormalMap para añadir relieve a estos objetos. Así mismo, la textura de iluminación anima su valor de Emission para dar la sensación de un parpadeo en las luces, a velocidad configurable.

Las piezas de la nave tienen las mismas características que un edificio pero carecen de la opción de ser filtradas, ya que cada pieza iba a ser única y no hay necesidad de hacerlo. Añaden además una segunda textura de iluminación, que se combina con la inicial en el nodo Emission. Esta segunda textura se creó pensando en la animación de la nave de parpadeo de luces, de la cual hablaremos más adelante.

Iluminación

Para mejorar la iluminación general del planeta introdujimos una modificación en la etapa lighting de todos estos shaders añadiendo una textura a modo de rampa de color. Esta textura funciona de forma similar a cuando se usa para colorear una textura en el shader de Roca en la escena inicial, pero en lugar de actuar sobre la textura base actúa sobre el valor de iluminación de cada pixel de los objetos, de forma que si el objeto está completamente iluminado se le aplica un tinte del color más a la derecha de la textura de iluminación, y si está completamente a oscuras se le aplica de la izquierda. Los valores de iluminación medios se interpolan y se obtiene el efecto deseado. Es una modificación sencilla que aporta un cambio de atmosfera bastante grande con un coste ínfimo.

Mejoras Propuestas

Efecto de Construcción

No sería muy difícil hacer un efecto de construcción tanto para edificios como para piezas de la nave. Por ejemplo, introduciendo una textura a modo de "ladrillos" de diferentes tonalidades de gris, insertándola en el nodo Clip y animando el valor que controle cuanto de dicha textura se recorta podríamos obtener un efecto de construcción controlando un único valor en una animación.

Animaciones y Efectos de Partículas

El aspecto gráfico del juego aportaba mucho a la hora de ver el juego, pero a la hora de jugarlo se veía todo muy parado… muy estático. Los seres vivos se beneficiarían ampliamente de las animaciones, e incluso las plantas, que a escala de tiempo normal no parecen moverse, se mueven cuando aumentamos la escala de tiempo.

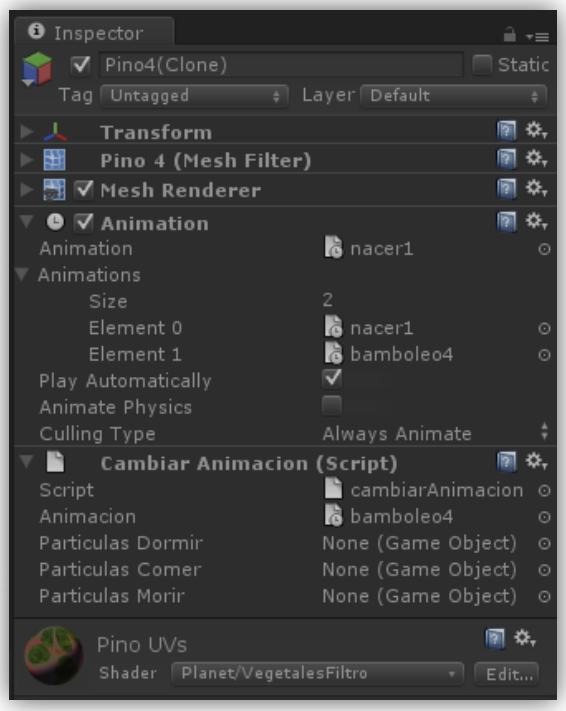
Animaciones básicas, animaciones con Armatures

Para crear la ilusión de movimiento, el ordenador renderiza en pantalla una imagen que es reemplazada repetidas veces por nuevas imágenes similares, pero modificadas. Estas modificaciones se hacen en varias de las propiedades de los objetos representados, principalmente su posición en el espacio, su rotación y su escala. Estos tres atributos se conocen como LocRotScale.

Estas modificaciones se pueden aplicar a cada objeto (como en nuestro caso) o por secciones del objeto, usando Armatures (Armazones).No hemos usado esta técnica, pero se basa en delimitar regiones del modelo y animar estas regiones usando la misma técnica.

Animaciones en GameObjects

Para animar cualquier GameObject en Unity se le inserta un componente Animation. Este componente tiene una lista de animaciones de la cual podemos elegir el tamaño, donde se insertan los diferentes archivos ".take". Uno de ellos se puede incluir en el campo Animation, el cual será considerado como animación por defecto. Con Play Automatically activado la animación empezará a reproducirse cuando se cree el objeto. No hemos hecho uso de AnimatePhysics. CullingType permite que de no estar el objeto en el campo de visión se ignore la reproducción de la animación.



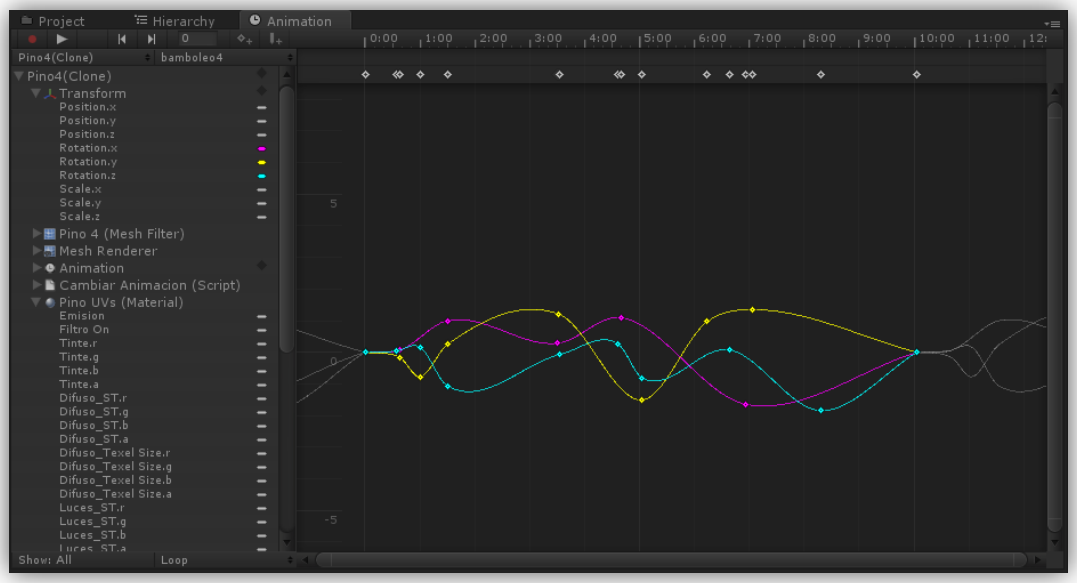
Añadiendo un componente Animation al GameObject.

Animation View Unity, eventos

Para crear o modificar una animación disponemos de la Animation View en Unity. En esta ventana podemos editar con todo detalle todos los componentes del GameObject, tanto de su Transform (posición, rotación y escala), su MeshFilter (la visibilidad del objeto), los scripts que tenga incrustados, sus Materiales...

Su uso se basa en los Keyframes o fotogramas clave. Consiste en marcar como interesante una posición en el tiempo de forma que tenga ciertas propiedades, y marcar otra en otro momento temporal con otras propiedades. El propio programa interpola los valores modificados con una curva, de la cual podemos

elegir distintos parámetros como su suavidad, su pendiente, etc. Así mismo podemos previsualizar la animación y navegar por cada fotograma con comodidad. En la animación también podemos incrustar funciones de scripts propios del GameObject y llamarlas mediante Eventos, de gran utilidad.



Animation View en Unity. Editando estas curvas podemos modificar en el tiempo cualquier propiedad del GameObject.

En la parte inferior tenemos el WrappingMode de la animación, su comportamiento en cuanto a reproducción se refiere, de forma que podemos elegir entre repetir la animación una sola vez, que se repita en forma de bucle y diversas opciones más. Este valor también puede cambiarse desde el inspector seleccionando un asset de tipo animación, junto con otras propiedades como el framerate (fotogramas por segundo).

Hemos creado varias animaciones, algunas para piezas de la nave que rotan sobre ejes y parpadean usando el atributo de Emission de cada uno de sus shaders, otras para manejar transiciones de cámara, pero las más importantes vendrían a ser las de los seres vivos.

Las plantas tienen 2 animaciones: nacer y bamboleo. Al crear el GameObject nacer es su animación por defecto y esta activado el Play Automatically. Es una animación muy larga que recrea como la planta nace y va creciendo durante el tiempo hasta que llega a una edad adulta. Cuando eso ocurre, en la propia animación hay un keyframe con un evento, es decir, una llamada a una función de un script adjunto al GameObject. Este script cambia la animación por defecto

de nacer por la de bamboleo, de forma que al terminar de "nacer", la planta empieza a menearse simplemente con cambiar la animación, sin necesidad de invocar el play de cada GameObject que hubiera llegado a una edad adulta.

Por otra parte los animales tienen una animación para cada uno de sus comportamientos, nacer, comer, mover, descansar y morir. Cada una de ellas se reproduce al cambiar de estado el animal, y duran un turno o una fracción de turno, en cuyo caso se repiten en bucle hasta que se cambien por otra en el siguiente turno. Todas ellas empiezan y acaban en una posición de "quietud" por lo que enlazan suavemente.

Partículas

Las animaciones dotaron de bastante vida a los elementos del juego, pero decidimos ir más allá e investigar los efectos de partículas para crear algún que otro efecto que hiciera las animaciones interesantes.

El sistema de partículas de Unity[r10] es bastante sencillo, y tiene múltiples aplicaciones. En nuestro caso las hemos utilizado para dotar a las animaciones de algún efecto que les dieran un toque más depurado. Se añaden varios componentes a un GameObject para crear el efecto, aunque el Animator es opcional:

ParticleEmitter: Crea las partículas según diferentes parámetros, los mas importantes en nuestro caso son el tamaño de las partículas, la cantidad de partículas emitidas (max y min emission) y el tiempo de vida (energy).

ParticleRenderer: aplica el material, el shader, a las partículas. Aquí elegimos el aspecto gráfico de cada sistema de partículas.

ParticleAnimator: Anima estas partículas con diferentes opciones, dándoles direcciones, rotaciones, cambios de color, etc. En nuestro caso los hemos usado para difuminar o desvanecer las partículas con el fondo, de forma que no desaparecieran abruptamente.

Hemos creado 3 sistemas de partículas asociados a sus respectivas animaciones: un emisor de "Zs" para representar que el animal descansa, un emisor de hojas asociado a los herbívoros comiendo, y un emisor de sangre y huesos para cuando los carnívoros devoran a sus presas.

Cada sistema se añade a un GameObject vacio y se crea un prefab con él. Cuando la animación llega a un punto deseado, se lanza un evento que instancia el prefab requerido. El sistema de partículas se autodestruye cuando termina de emitir las partículas que hemos configurado gracias a una propiedad de autodestrucción pensada para estos casos.

Mejoras propuestas

Animaciones animales:

A pesar de toda la vida que ganan los elementos de la escena al animarse, el hecho de haber una única animación para cada estado de todos los animales hace que muy frecuentemente parece que asistamos a una coreografía mundial. Sería conveniente crear animaciones propias de cada animal, e incluso distintas animaciones con variaciones para cada evento de cada animal, del mismo modo que se elige un modelo random de una lista de modelos a la hora de insertarlo en el tablero, podría elegirse una de estas variaciones de entre las posibles de forma aleatoria, y le daría un aspecto menos uniforme a la visión general.

Animación de construcción de piezas de la nave o edificios:

Tal como se comento en el tema de shaders, modificando el shader de los edificios y las piezas de la nave como allí se describe, podemos crear una animación que usando uno de sus valores al iniciarse la pieza seria invisible y según pasara el tiempo los "ladrillos" de la textura de construcción del shader con colores más oscuros irían apareciendo, dando paso a los más claros hasta formar la pieza en su totalidad. Cuando la animación de construcción se completara cambiaríamos de animación a la original de la pieza con el script de cambiar animación.

Mensajes de información por partículas

Otra posible aplicación de las partículas sería informar al usuario de eventos sin crear ventanas de interfaz que requieran una participación más activa que pueda resultar molesta, como por ejemplo, no disponer de recursos a la hora de insertar un edificio. En lugar de mostrar una ventana que necesite de una confirmación, deteniendo la acción del juego, podría crearse un prefab de un emisor de partículas con emisión 1 y con un mensaje de "No hay suficientes recursos" que dure el tiempo suficiente como para leerse. En caso de no tener recursos suficientes al intentar insertar el edificio, se instanciaría el sistema en la posición donde el usuario haya pinchado para mostrar el mensaje justo donde tiene lugar la acción y de forma no intrusiva.

Interfaz

Como toda aplicación informática que se precie, el juego dispone de una interfaz. El método para crear interfaces en Unity es bastante primitivo en esta versión. Su funcionamiento se basa en la función OnGUI(), que se llama en múltiples ocasiones cada frame, de la misma forma que Update(). En ella se crean o actualizan los componentes gráficos 2D de la interfaz, que recogen las interacciones con el usuario.

Estos componentes se categorizan por "estilos", de los cuales se describen unas características como funcionamiento especifico a la hora de interaccionar o dimensiones y propiedades visuales. Unity facilita ciertos estilos por defecto como botones, labels, sliders y scrollbars, muy comunes en su uso pero bastante anodinos. Para cubrir nuestras necesidades tenemos también la posibilidad de crear estilos propios, los llamados "CustomStyles". Prácticamente todos los botones de nuestro proyecto son estilos propios, pero la única diferencia en la mayoría de ellos es estética.

A la hora de construir las interfaces se hace pesada la falta de facilidades por parte de Unity en esta versión. Todos los componentes han de ser codificados manualmente, tanto posicionamiento como dimensiones y por supuesto eventos invocados. Se echa en falta tanto un editor gráfico para lo primero como un editor de eventos para lo segundo.

A pesar de todo esto, creemos que hemos hecho un buen trabajo y ha quedado una interfaz muy profesional. Esperamos con ansias que en las nuevas actualizaciones de Unity mejoren este aspecto!

Un mapa de la interfaz final del proyecto.

Las interacciones del usuario con el juego también se basan en gran medida en los Raycasts. Los Raycasts son vectores o líneas que van de un punto del espacio en una dirección, y podemos controlar ambas. Su utilidad viene dada por la colisión o intersección de este rayo con un objeto que tenga un componente Collider. Al chocar el rayo con el componente obtenemos todo tipo de información interesante como las coordenadas del espacio del vértice del Mesh asociado al collider, el triangulo, la normal, la coordenada de la textura en los UVs, etc. La más frecuente en nuestro caso es el posicionamiento de objetos a insertar en las coordenadas adecuadas del planeta.

4. Pruebas

Introducción

Una de las ventajas de Unity es que nos permite probar el funcionamiento de las cosas implementadas con solo pulsar un botón. Gracias a esta facilidad, hemos podido hacer multitud de pruebas durante el periodo de desarrollo del proyecto, corrigiendo y adaptando el mismo a raíz de los resultados que obteníamos. Podemos decir que nuestro plan de pruebas ha sido una "evaluación continua", en la que cada característica nueva incorporada era sometida a un test inmediato con el fin de detectar fallos y defectos.

Casi todo nuestro plan de pruebas se ha incorporado al desarrollo desde el primer momento, en forma de pruebas de caja blanca y pruebas funcionales. Todo el equipo ha sido responsable de probar tanto sus aportaciones al código como el comportamiento general del proyecto y de secciones individuales. Cuando el equipo encontraba un fallo o un defecto, se depura el código, se arregla y se vuelve a ejecutar la prueba para comprobar que ya no está, o en caso de ser un defecto de una sección que no le correspondía, se notificaba pertinentemente al responsable de esa sección de código.

Pruebas realizadas

Como se ha dicho anteriormente, las pruebas realizadas son demasiado numerosas para enumerarlas todas, así que a continuación enumeraremos algunas de las más representativas:

1. Pruebas iniciales de la GUI de la escena inicial:
    a. Elementos a comprobar:
       En la primera prueba de la interfaz de usuario planteada, la intención es probar el correcto funcionamiento de todos los elementos de la interfaz en la primera escena.
       Se probarán que todos los elementos estáticos aparezcan en el lugar adecuado y con el aspecto esperado y se probará así mismo que todos y cada uno de los controles con posibilidad de interacción de la interfaz funcionan y desempeñan correctamente su cometido. No se probará nada más.
    b. Procedimiento a seguir:
       El usuario encargado de la prueba hace un recorrido por la interfaz, interactuando del mayor número de maneras posibles y comprobando que cada uno de los controles existentes se comporta de la manera esperada.
    c. Criterios de fallo/paso de la prueba:
       Si todos los controles cumplen su función y la interactividad con ellos funciona de la manera esperada, se considera la prueba superada. En caso de que al menos uno de los controles no se comporte según lo esperado, se considera fallo y se procede a depurar el defecto y a volver a comenzar la prueba.

2. Pruebas de la generación de textura de altura aleatoria
    a. Elementos a comprobar:
       Dada la complejidad del proceso de generación aleatoria del terreno, debemos comprobar dos cosas: que la generación produce una salida en forma de textura y que esa salida es satisfactoria y adecuada a nuestros intereses. La textura además debe cumplir ciertas condiciones, como que su tamaño es el adecuado, que sus píxeles se encuentran en un rango aceptable de valores para el resto del proceso, etc.
    b. Procedimiento a seguir:
       Esta prueba puede considerarse tanto de caja negra como de caja blanca, pues aunque el código interno y su estructura nos es conocida, una vez puesta en marcha se debe esperar a los resultados (o a un fallo) para su evaluación.
    c. Criterios de fallo/paso de la prueba:
       Al concluir se valorará, primero, el formato de la salida: que sea una textura de las dimensiones dictadas, que los valores de los pixeles entren dentro del rango determinado y que son adecuados para servir de entrada al resto del proceso.
       También se valorará que estos pixeles formen una representación lo más adecuada posible, según nuestro punto de vista personal, de un terreno.

3. Pruebas de la GUI de la escena principal
    a. Elementos a comprobar:
    Dado que es imposible probar todos los elementos en una sola prueba, esta prueba se divide en sub-pruebas más específicas que prueban uno a uno cada uno de los apartados, y luego una sucesión de pruebas aleatorias completas en las que ese prueba el sistema completo como si de un usuario final se tratase.
    Durante las sub-pruebas se comprueba el funcionamiento de los elementos de la interfaz, su correcto funcionamiento, colocación y aspecto. Durante las pruebas generales se comprueba el funcionamiento en conjunto de la interfaz, sobre todo intercalando acciones de diferentes secciones e interactuando entre ellas.
    b. Procedimiento a seguir:
    El usuario encargado de la prueba hace un recorrido por la interfaz, interactuando del mayor número de maneras posibles y comprobando que cada uno de los controles a comprobar se comporta de la manera esperada.
    c. Criterios de fallo/paso de la prueba:
    Si todos los controles comprobados cumplen su función y la interactividad con ellos funciona de la manera esperada, se considera la prueba superada. En caso de que al menos uno de los controles no se comporte según lo esperado, se considera fallo y se procede a depurar el defecto y a volver a comenzar la prueba.

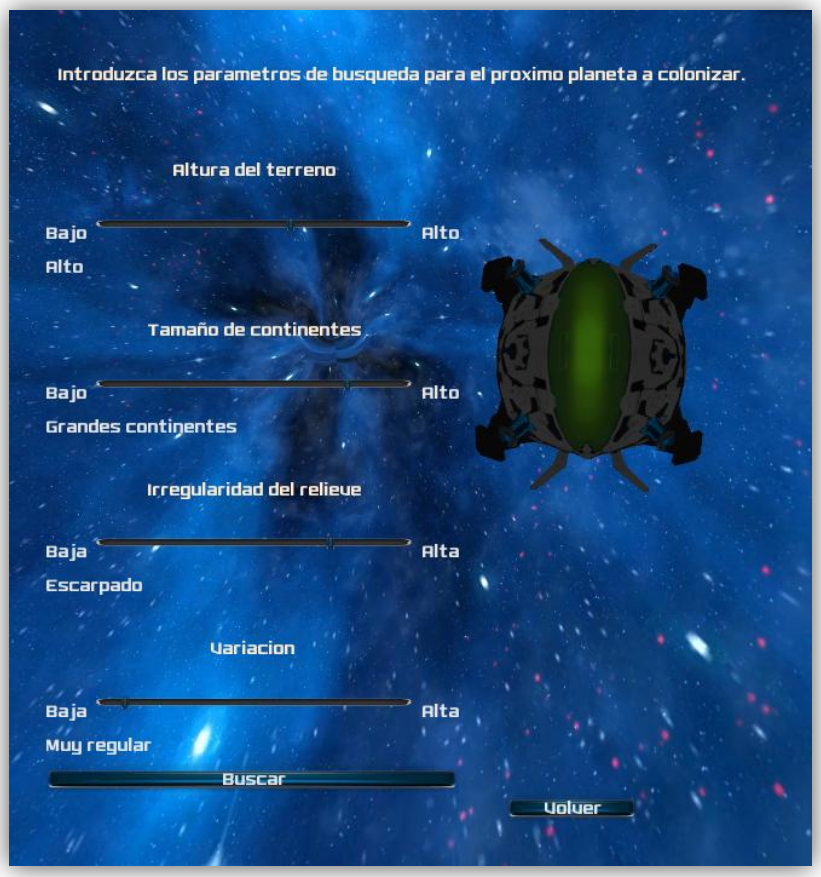
4. Pruebas del algoritmo de vida:
    a. Elementos a comprobar:
       Esta prueba la podemos separar en secciones también, pues el algoritmo de vida contiene objetos de muy diferentes propiedades y funcionamientos y normalmente se comprueban agrupándolos por clases: herbívoros, carnívoros, plantas y edificios.
       En cada una de las secciones se comprueba que el ser funciona como es debido, que atraviesa su ciclo de vida completo y que interactúa de forma correcta con su entorno.
    b. Procedimiento a seguir:
       Se introduce un ser del tipo elegido en el terreno de juego, con el tablero y el planeta correctamente generados y se observa su ciclo de vida y su interacción con otros seres o con el tablero. Si es necesario se introducen en el tablero otros seres para poder probar todas sus interacciones. Se prueba repetidas veces porque, al estar el comportamiento de los seres (solo de los vivos) gobernado por un factor aleatorio importante, debemos cubrir el máximo posible de su comportamiento.
    c. Criterios de fallo/paso de la prueba:
       Si el ser completa su ciclo de vida entero y sus interacciones de forma correcta, mientras monitorizamos sus atributos básicos, consideramos la prueba como superada. En caso contrario depuramos el fallo y volvemos a repetirla desde el principio.

5. Pruebas generales estéticas o funcionales:
    a. Elementos a comprobar:
       Cada una de las nuevas incorporaciones estéticas o funcionales de pequeño tamaño que se introducen, una a una. Por ejemplo los efectos de partículas, las animaciones, los efectos de luz, un cambio de texturas o modelo tridimensional, pequeños cambios en las texturas de la interfaz, etc.
       Estas pruebas están planteadas de una manera muy atómica, y su objetivo es probar exclusivamente la incorporación de estos pequeños añadidos.
    b. Procedimiento a seguir:
       Una vez comenzado el procedimiento, se realiza una prueba aislada del elemento introducido, comprobando su interactividad, aspecto, funcionalidad o lo que sea pertinente dependiendo del caso. Si es pertinente se comprueban las funciones que deben interactuar con dicho elemento.
    c. Criterios de fallo/paso de la prueba:
       Si el nuevo elemento añadido cumple su función y el proyecto no arroja ningún fallo por su introducción, se considera la prueba superada. En cualquier otro caso, se depura el defecto y se vuelve a repetir desde el principio.

5. Manual de usuario

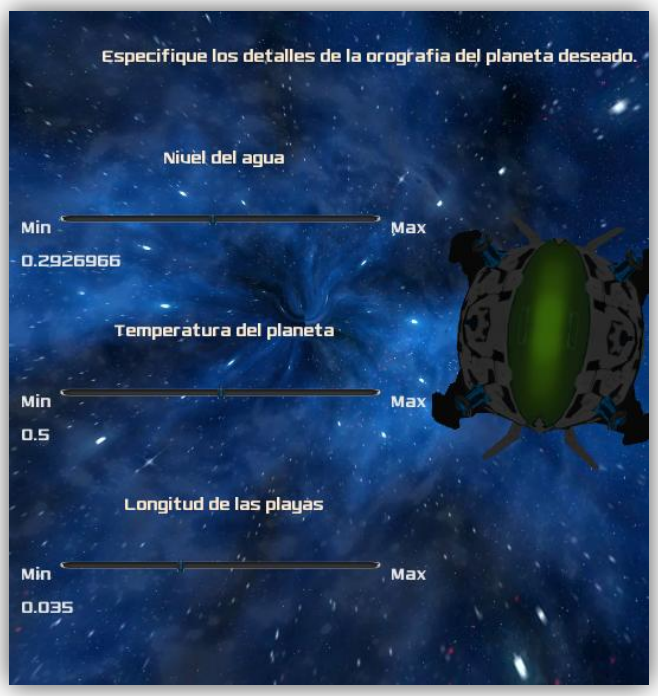
Creación/búsqueda del planeta

El primer paso en Terraform una vez arrancado el juego es comenzar una nueva partida. Durante los menús posteriores, el usuario puede elegir entre varios parámetros en los que se delimita el planeta que estamos buscando. Entre estos factores podemos encontrar el tamaño de los continentes, la altura de sus montañas, la proporción entre terreno firme y mares, etc.



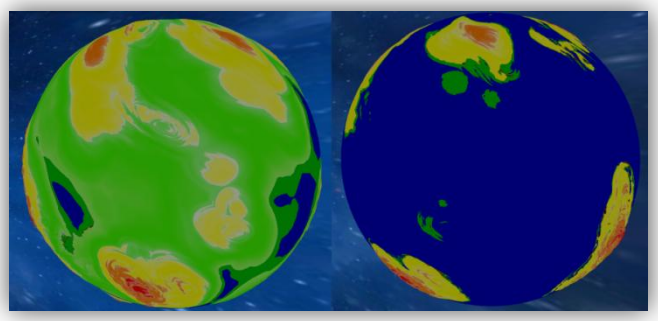
Especificando unos parámetros de búsqueda de planeta

Mediante el uso de unas barras horizontales podemos ajustar los parámetros del proceso de búsqueda de un planeta con las características especificadas. Incluso la propia nave nos puede ofrecer una previsualización del planeta que estamos buscando.



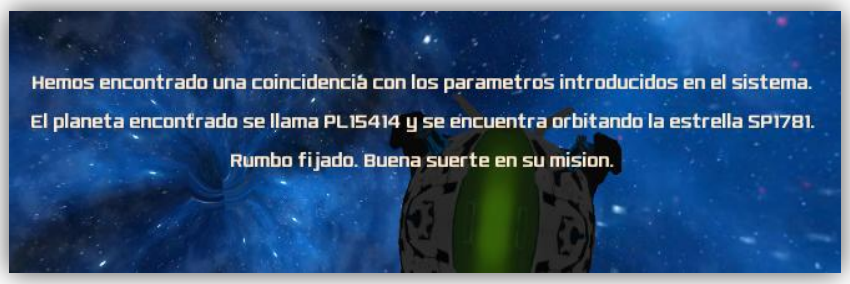
Más parámetros de búsqueda.

Una vez elegidos todos los parámetros de búsqueda y obtenida la previsualización del planeta que queremos colonizar, continuamos con el proceso pulsando en los botones correspondientes hasta que el sistema encuentre una coincidencia en el universo conocido.



Planeta seco y planeta con mucho líquido

Si estás de acuerdo con el planeta asignado, entonces estás listo para comenzar la terraformación. ¡Mucha suerte!



Mensaje de Apoyo desde el equipo técnico.

Jugabilidad y objetivos del juego

Nuestra nave ha llegado a un nuevo planeta que deberemos explotar añadiendo edificios y seres vivos para ir consiguiendo recursos con los que poder mejorarla. Esto a su vez nos permitirá acceder a nuevos edificios más complejos y nuevas mejoras.

El objetivo final del juego consiste en activar una habilidad de la nave a la que sólo podremos acceder después de haber completado todas las mejoras y de haber construido e insertado todos los edificios y seres disponibles. Este habilidad llamada “Vórtice Espaciotemporal” permitirá a tu especie acceder al planeta y colonizarlo completamente.

Pero para llegar a tener los recursos necesarios para poder abrir el vórtice necesitaremos construir fábricas de distintos componentes, centrales de energía y granjas que produzcan material biológico.

En las siguientes páginas podemos encontrar una descripción de la interfaz del juego y de las distintas opciones que ofrece:

Interfaz

Interfaz General.

Hábitats

Tanto los seres vivos como los edificios están ubicados en una serie de hábitats. Para saber si un vegetal, un animal o un edificio se puede insertar en ese hábitat usaremos esta leyenda:

Icono			
Significado	Habitable	Poco habitable	Inhabitable

Vegetales

A continuación una lista de los vegetales de los que dispondremos en el juego:

Vegetal	Nombre	Coste Energía	Coste Comp Básicos	Coste Comp Avanzados	Coste Material Biológico
	Seta	200	110	0	0
	Flor	230	100	0	0
	Caña	450	150	0	0
	Arbusto	420	170	0	0
	Estromatolito	620	190	0	5
	Cactus	600	180	0	5
	Palmera	1500	280	10	10
	Pino	1800	300	10	10
	Ciprés	2500	420	50	20
	Pino alto	3000	500	50	25

Los hábitats para cada vegetal son los siguientes:

Nombre	Montaña	Llanura	Colina	Desierto	Volcánico	Mar	Costa	Tundra
Seta	✓	✓	✓	✗	✗	✗	✓	✗
Flor	✓	✓	✓	✗	✗	✗	✗	✗
Caña	✗	✓	✓	✓	✗	✗	✓	✗
Arbusto	✓	✓	✓	✓	✗	✗	✗	✗
Estromatolito	✗	✓	✗	✓	✓	✗	✓	✓
Cactus	✓	✓	✗	✓	✓	✗	✗	✗
Palmera	✗	✓	✗	✓	✗	✗	✓	✗
Pino	✓	✓	✓	✗	✗	✗	✓	✗
Ciprés	✓	✓	✓	✓	✓	✗	✓	✓
Pino alto	✓	✓	✓	✗	✗	✗	✓	✓

Animales

Igualmente los animales usados en el juego son:

Animal	Nombre	Coste Energía	Coste Comp Básicos	Coste Comp Avanzados	Coste Material Biológico	Tipo de Alimentación
	Caracol	1000	250	0	10	Herbívoro
	Conejo	1500	350	0	15	Herbívoro
	Vaca	2200	450	30	25	Herbívoro
	Jirafa	3200	570	100	35	Herbívoro
	Tortuga	4000	750	500	50	Herbívoro
	Zorro	2000	350	20	15	Carnívoro
	Lobo	3300	450	80	25	Carnívoro
	Tigre	4500	750	250	40	Carnívoro
	Oso	5600	830	320	60	Carnívoro
	Tiranosaurio	7500	1200	430	90	Carnívoro

Los hábitats para cada animal son los siguientes:

Nombre	Montaña	Llanura	Colina	Desierto	Volcánico	Mar	Costa	Tundra
Caracol	✗	✓	✓	✗	✗	✗	✓	✗
Conejo	✓	✓	✓	✗	✗	✗	✗	✓
Vaca	✗	✓	✓	✗	✗	✗	✓	✗
Jirafa	✗	✓	✓	✓	✗	✗	✗	✗
Tortuga	✓	✓	✓	✗	✓	✗	✗	✗
Rata	✗	✓	✓	✗	✗	✗	✗	✓
Lobo	✓	✓	✓	✗	✗	✗	✗	✓
Tigre	✓	✓	✓	✗	✗	✗	✓	✗
Oso	✓	✓	✓	✗	✗	✗	✗	✓
Tiranosaurio	✗	✓	✓	✓	✓	✗	✗	✓

Edificios

Los edificios que podemos utilizar en el videojuego son los siguientes:

Edificio	Nombre	Coste inicial del edificio				Metales necesarios	Coste/producción máximo del edificio			
		Energía	Comp Bas	Comp Avz	Mat Bio		Energía	Comp Bas	Comp Avz	Mat Bio
	Central de energía	150	30	0	0	comunes	40	0	0	0
	Fab. Comp. Básicos	180	20	0	0	comunes	-10	20	0	0
	Granja	800	280	0	0	ninguno	-75	0	0	0
	Fab. Comp. Avanzados	1000	350	0	10	raros	-100	0	40	0
	Central de energía avanzada	2500	250	50	20	raros	1000	0	0	0

Y se podrán construir en los siguientes hábitats:

Nombre	Montaña	Llanura	Colina	Desierto	Volcánico	Mar	Costa	Tundra
Central de energía	✗	✓	✓	✗	✗	✗	✗	✗
Fab. Comp. Básicos	✗	✓	✓	✗	✗	✗	✗	✗
Granja	✗	✓	✓	✗	✗	✗	✗	✗
Fab. Comp. Avanzados	✗	✓	✓	✓	✗	✗	✓	✓
Central de energía avanzada	✗	✓	✓	✓	✗	✗	✓	✓

Mejoras

Las mejoras de la nave nos permiten realizar nuevas acciones, acceder a nueva información e incrementar la cantidad de recursos que podemos almacenar:

	Mejora	Descripción
Sensores		Usando unos sensores ópticos de lo más rudimentario la nave tendrá acceso a información relevante en una zona limitada. Algo es algo. (Muestra Información Básica de las casillas)
		Un modulo de sensores de altura, temperatura, humedad, condiciones de viento y varios factores mas que recopila información y elabora un mapa de los diferentes ecosistemas que presenta el planeta. (Habilita el mostrar el Mapa de Hábitats en Habilidades)"
		Utilizando un preciso medidor de radiación estos sensores localizan los minerales más valiosos del subsuelo tras analizar las longitudes de onda más exóticas.(Habilita la detección de Recursos en Habilidades)
		Utilizando técnicas de predicción, visión térmica e introduciendo a los seres vivos marcadores a base de isotopos radioactivos (inocuos!) tendremos controlada a la fauna y la flora local con este sensor. (Habilita la Visualización de Vida en menú Habilidades)
Motores		Unos motores convencionales para la navegación espacial. No son lo más óptimo. No son lo más eficiente. ¿Que si vas más rápido? Si. (Aumenta la velocidad de la nave)
		Motores especialmente diseñados para la navegación en orbitas geoestacionarias bajas. Cancelación inercial, corrección de deriva gravitacional y un montón de tecnicismos. Este si que si. (Mejora notablemente la velocidad de la nave)
		Un escudo magnético propio protege de los rayos cósmicos que azotan las zonas más inhóspitas del planeta.(Permite transitar cerca de los polos)
		Aumenta la potencia del repulsor gravitacional y eleva la nave a una orbita superior. (Eleva la nave sobre el planeta)
Energía		Añade un condensador con receptores de microondas para almacenar el excedente de energía producido por las centrales planetarias. (Aumenta la Energía Máxima disponible)
		Añade un condensador en forma de anillo que aumenta la capacidad energética de forma considerable. Además sirve de soporte para otras piezas. (Aumenta la Energía Máxima disponible y desbloquea otras mejoras)
		Un array de paneles solares capta la energía de la estrella durante el día, y la almacena para su posterior uso. (Habilita el Foco Solar)
		Habilita las armas y los sistemas más avanzados de la nave, incluyendo el portal para traer a nuestra especie al planeta.(Habilita Fertilizante Ecoquímico, Bomba de Implosión Controlada, Virus SelectivoPoblacionaly Portal Espacio/temporal)
Almacén		Investiga la tecnología avanzada. Contenedores de condiciones controladas adaptados a material delicado permiten almacenar materiales. (Permite almacenar Componentes Avanzados y construir fabricas y centrales avanzadas)
		Investiga la tecnología genética. Para almacenar el material obtenido de los seres vivos en condiciones seguras, mimetizando la gravedad, temperatura y humedad de un planeta en un contenedor. (Permite almacenar Material Biológico y construir las granjas)
		Mejorando los algoritmos de ordenación de los contenedores de la nave se obtiene un incremento en la capacidad de almacenaje. Viene con un contenedor gratis. (Mejora la Capacidad de Almacenamiento)
		Añadiendo contenedores de cada clase se amplia enormemente el espacio de almacenaje. (Mejora la Capacidad de Carga)

Habilidades

Las habilidades nos permiten realizar acciones especiales como identificar visualmente algunos elementos mediante los filtros (5 primeras habilidades) y eliminar elementos de distintas maneras. Para ganar una partida tendremos también que usar una habilidad. Son las siguientes:

Habilidad	Descripción
	Desactiva todos los filtros que tengamos activos en el momento.
	Filtro metales: permite ver de forma resaltada en que zonas hay metales comunes y/o metales raros.
	Filtro hábitats: muestra en el planeta los diferentes hábitats que hay mediante colores.
	Filtro vegetales: permite identificar en el planeta a los diferentes tipos de especies vegetales que hay.
	Filtro animales: permite identificar en el planeta a los diferentes tipos de especies animales que hay.
	El Foco Solar es totalmente ecológico. Utiliza la energía captada cuando la nave esta bajo la acción directa de la radiación solar y la redirige a las zonas donde es interesante. Aun no se ha comprobado si este fenómeno es perjudicial para los animales, pero de momento no hemos recibido ninguna queja.
	El Fertilizante Ecoquímico estimula biológicamente a los seres vivos de una zona incrementando asombrosamente su capacidad reproductiva.
	La Bomba de Implosión Controlada destruye todo ser y edificio de una determinada zona. ¡Utilizar con mucha precaución!
	El Virus Selectivo Poblacional elimina selectivamente a una especie de animal o vegetal en una posición determinada.
	El Portal Espaciotemporal se habilita cuando tu tarea ha terminado y al usarlo abre un portal que comunica con tu planeta. Una vez desplegado tu misión en este planeta habrá terminado.

Menú

Al acceder al menú podremos guardar la partida, modificar el volumen tanto de la música como de los sonidos, retroceder al menú principal y abandonar el juego.

Menú Principal

Menú de Sonido

Atajos de teclado

TECLA	ACCIÓN
W, ↑	Mueve la nave hacia arriba
A, ←	Mueve la nave hacia la izquierda
S, ↓	Mueve la nave hacia abajo
D, →	Mueve la nave hacia la derecha
1	Pausa el juego
2	Cambia la velocidad del juego a 1x
3	Cambia la velocidad del juego a 2x
4	Cambia la velocidad del juego a 5x
ESC	Abre o cierra el menú del juego

6. Conclusiones

Reflexión Final

La principal conclusión después de todos estos meses de arduo y desesperante trabajo es que es posible hacer un videojuego desde cero y, aunque es algo realmente frustrante, motiva en gran medida ver cada progreso en funcionamiento. Si algo hemos sacado en claro es la dificultad de planificar los objetivos. La mayoría de las cosas parece que se pueden hacer en unas pocas horas y luego tardas semanas, y hay otras cosas que parecía que iban a ser inviables y sin embargo se resuelven rápidamente.

La creación de un videojuego es una labor muy difícil de valorar y creemos que sólo merece la pena llevarla a cabo como proyecto fin de carrera si tienes una enorme motivación y un gran interés en el mundo de los videojuegos. Nosotros los tenemos y por tanto estamos contentos y muy orgullosos del trabajo realizado.

En cuanto al motor gráfico utilizado (Unity3D) estamos muy satisfechos con su funcionamiento. Si bien es cierto que tiene muchas cosas que mejorar, es un motor muy potente y muy bien pensado que sobre todo destaca por ser accesible. Le auguramos un futuro muy prometedor y la intención que tenemos es de seguir trabajando con él y pagar llegado un momento dado la versión Premium que incluye ciertas mejoras. Por tanto lo recomendamos totalmente para introducirse en el mundo del desarrollo de videojuegos.

Con Blender nos pasa un poco lo mismo. Aunque su papel no sea tan relevante en el proyecto como el del motor gráfico estamos también muy contentos con su uso. No empezamos a usarlo por el hecho de que fuera gratuito ya que disponíamos de una licencia de estudiante de Maya, sino porque es mucho más sencillo e intuitivo y es algo que se comenta mucho en internet. Sinceramente si costase 100€ los pagaríamos gustosamente porque es un software pensado 100% para la labor que desempeña y se nota que no está parcheado como pasa con otros. Sin duda recomendado completamente.

Como análisis post-mortem autocrítico, creemos que hemos hecho mal en no ceñirnos a una captura inicial de requisitos, dado que hemos ido introduciendo nuevas ideas casi constantemente según adquiríamos conocimientos o nos inspirábamos, pero a la larga creemos que el no haberlo hecho ha dado resultados muy positivos.

7. Glosario

Términos

- Unity 3D:
  Motor y editor gráfico utilizado para realizar este proyecto.
- GameObject:
  Componente básico del motor Unity3D con el que se interactúa mediante el editor.
- Shader:
  Programa en lenguaje de alto nivel que describe las operaciones necesarias que ha de realizar la GPU para renderizar un modelo.
- Transform:
  Representa la posición, rotación y escala de un GameObject en un momento dado.
- Mesh (o Malla):
  Malla poligonal que almacena la información referente a la representación tridimensional de un objeto.
- Collider:
  Representación del espacio ocupado por un objeto tridimensional que representa sus límites físicos.
- Texture2D:
  Almacén donde se guarda información de una imagen preparada para colocarse en un objeto de juego.
- Component:
  Módulo que se añade a un GameObject para agregarle funcionalidad.
- Renderer:
  Renderizado (render en inglés) es un término usado en jerga informática para referirse al proceso de generar una imagen desde un modelo. Este término técnico es utilizado por los animadores o productores audiovisuales y en programas de diseño en 3D.
- MonoBehaviour:
  Clase de la que pueden heredar los scripts y que les otorga funcionalidad para convertirse en componentes, pudiendo agregarse a un GameObject.
- Viewport (o punto de vista)
  Punto del espacio virtual que representa la cámara o el teatro cartesiano del jugador o espectador en un entorno de gráficos 3D.

- MonoDevelop:
  IDE utilizado en Unity3Dpara crear y editar scripts en C#, UnityScript y Boo.
- Skybox:
  Conjunto de texturas mapeadas en forma de cubo que envuelven la cámara y crean la sensación de bóveda celeste.
- Rasterización:
  La rasterización es el proceso por el cual una imagen descrita en un formato gráfico vectorial se convierte en un conjunto de píxeles o puntos para ser desplegados en un medio de salida digital.
- Pixel:
  Un pixel es la mínima unidad de medida del soporte físico donde se representan las imágenes digitales.
- Texel:
  Texel o Fragmento es una unidad abstracta de medida que representa la mínima fracción de un modelo de datos gráfico.
- Filtrado de texturas:
  Las texturas son creadas a resoluciones específicas, pero ya que la superficie en donde están aplicadas puede estar a cualquier distancia del observador, estas pueden mostrarse en tamaños arbitrarios en la imagen final. Como resultado, un píxel en la pantalla usualmente no corresponde directamente a un texel. Alguna técnica de filtrado debe ser aplicada para lograr imágenes claras a cualquier distancia. Hay varios métodos, con diferentes relaciones entre calidad de imagen y complejidad computacional.
- Environment mapping (o mapeado del entorno)
  Es una técnica de mapeado de texturas en la cual las coordenadas de la textura son dependientes de la vista. Una aplicación común, por ejemplo, es simular un objeto altamente reflexivo como metal cromado, espejos, etc. Por ejemplo, se puede mapear el entorno de un cuarto a una copa de metal y cuando el observador se mueva alrededor de la copa las coordenadas de la textura en los vértices de la copa se mueven proporcionalmente, dando la ilusión de reflejar la escena en la superficie del objeto.
- Bump mapping (o Normal mapping)
  Es otra forma de mapeado de textura que provee a los píxeles de profundidad. Especialmente con pixel shaders modernos, los bump maps

crean efectos como mapeado de imperfecciones, golpes y rugosidad que dependen de la luz y el punto de vista en una superficie para aumentar el realismo.

- Nivel de detalle (LOD)
  En muchas aplicaciones modernas el número de polígonos en una escena puede ser impresionante. Sin embargo, un observador en una escena sólo podrá discernir detalles de objetos cercanos. Los algoritmos de nivel de detalle varían la complejidad de la geometría en función de la distancia al observador, de forma que los objetos justo enfrente del observador pueden ser mostrados en su completa complejidad mientras que los objetos que están más lejos pueden ser simplificados dinámicamente o incluso ser reemplazados por imágenes 2D (billboards).

8. Referencias

- [r1] “The Importance of Being Noisy: Fast, High Quality Noise”, Natalya Tatarchukhttp://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/
- [r7] “Source SDK”, http://source.valvesoftware.com/
- [r8] “Fisher–Yates shuffle”, http://en.wikipedia.org/wiki/Knuth_shuffle
- [r9] “Ecuación del Renderizado” http://en.wikipedia.org/wiki/Rendering_equation
- [r10] "Sistemas de Partículas" http://docs.unity3d.com/Documentation/Components/comp-ParticlesLegacy.html
- [r11] Spacescape versión 0.3 - 10/12/2010 Copyright 2010 Alex Peterson, Todos los Derechos Reservados. http://alexcpeterson.com/spacescape
- [r12] "Strumpy Shader Editor" http://forum.unity3d.com/threads/56180-Strumpy-Shader-Editor-4.0a-Massive-Improvements