Diseño y desarrollo de un videojuego de simulación en 3D

Resumen

El objetivo de este TFG es documentar y realizar el diseño y desarrollo desde cero de un videojuego en 3D. El juego será una simulación de una colonia de hormigas, en la que el jugador tiene control indirecto sobre las acciones del nido. Se realiza en Unity, una plataforma de desarrollo de videojuegos.

Algunos de los objetivos principales del proyecto son:

- Creación de un terreno dinámico, que permitirá a las hormigas excavar en cualquier parte del mapa designado por el jugador.

- Que las hormigas sean agentes simples independientes, que compartiendo información y trabajando juntos simulen una inteligencia de colonia

- Que las hormigas exploren el mapa mediante un sistema de caminos de feromonas

- Crear animaciones 3D estilístico.

- Implementar un estilo artístico low poly

- Finalizar una demo jugable del juego

Table of Contents

Inspiraciones

Dos juegos que inspiraron este proyecto son Dwarf Fortress y Deep Rock Galactic.

En Dwarf Fortress, el jugador controla un pequeño asentamiento de enanos en una nueva parte del mundo a elegir por el jugador. El jugador puede mandar a que los enanos excaven en cualquier parte del mapa, asignando salas de dormir, de comer, de almacén, cárcel, etc. Estas zonas pueden coger cualquier forma y estar en cualquier parte, por lo que el jugador debe organizar la fortaleza de forma que los enanos lo puedan navegar eficientemente. Además, a pesar de ser un juego de vista dos dimensional, el mapa se divide en las capas horizontales del terreno, por lo que las fortalezas se pueden diseñar en tres dimensiones. Cada enano es un agente individual, que decide por su cuenta ir a beber, hablar con otros enanos, dormir, recolectar recursos o seguir las instrucciones de excavación o construcción del jugador.

Aunque las hormigas no serán tan complejas como los enanos, los cuales tienes personalidades, gustos y pasados distintos; si quise emular en ellas cómo siendo agentes individuales simples pueden juntos crear y gestionar un gran nido de hormigas. Otro aspecto de Dwarf Fortress que quiero emular es la libertad de dónde y cómo formar el nido en el mapa: cualquier parte debe ser una opción, en cualquier forma y combinación de túneles y salas.

En Deep Rock Galactic, el jugador forma parte de un escuadrón de hasta 3 otros jugadores en una misión en la subsuperfice de algún planeta aleígena. Hay varios tipos de misiónes, pero todos incluyen bajar al planeta y complir un recado de algun tipo. Gran parte del atractivo del juego es el sistema de cuevas en el que se adentran los jugadores: se trata de un mapa generado aleatoriamente, repleto de túneles y cámaras y las criaturas que contienen. El terreno es altamente interactivo, y los enanos son capaces de destruir cualquier parte de ella para excavar minerales o conectar distintas partes del mapa usando sus picos y taladros. El terreno y los personajes además se representan en un estilo de arte llamativo y low-poly, que hace juego con la naturaleza destruible del mapa. Son el terreno y el estilo de arte lo que inspiraron en gran parte este juego.

Figure 2: Pantallazo de sistema de cuevas y cuartos en Dwarf Fortress

Figure 1: Pantallazo de un sistema de cuevas en terreno dinámico en Deep Rock Galactic

Introducción

Estado del arte / software usado

Describir aquí como llegué a decidir usar Unity, VisualStudio y blender para el desarrollo del TFG.

Poner tambien el razonamiento del estilo de arte?

Creación del terreno dinámico: marching cubes

Razonamiento

Una de las mecánicas que quiero que destaquen de la demo es la capacidad del jugador para poder elegir cualquier parte del mapa como la localización del nido. Esta libertad crearía mucho replay value, ya que habría una infinidad de opciones incluso en el mismo mapa. Sin embargo, cuanta más libertad le concedes a un jugador, más situaciones imprevistas, glitches y exploits pueden ocurrir. Es necesario entonces encontrar una forma muy simple y flexible de representar el mapa del mundo, que puede ser editado y interactuar con las otras partes del juego fácilmente.

Como comentado antes, quise emular la libertad de interacción con el terreno de Deep Rock Galactic, por lo que intenté descubrir qué sistema usaron para crear el terreno. Aunque nunca publicaron ninguna parte de su código (obviamente), en una entrevista con los desarrolladores del juego estos confirmaron que usaron una modificación del algoritmo marching cubes para generar su terreno, y para la destrucción de esta usaron CSG (Constructive Solid Geometry): a los meshes del terreno se le sustraen otros meshes para formar las excavaciones de los jugadores. Decidí entonces investigar el algoritmo de cubos de marcha.

Una de las otras opciones que investigué fue el sistema de terreno por defecto de Unity. El sistema de terreno de unity consiste en un mesh simple con valores de altura en cada coordenada del plano horizontal. Cada punto del mesh en dichas coordenadas entonces se encuentra en la altura definida. Esto significa que no puede formar túneles, ya que tienen un techo y un suelo. Además de que no es apto para la creación de un terreno altamente tres dimensional, no permite crear aperturas para cuevas en su superficie fácilmente. Por estas razones, rápidamente descarté la idea de usar los objetos terreno nativos de Unity y me centré en el algoritmo de cubos de marcha.

Marching cubes, ¿Qué es?

Cubos de marcha es el algoritmo que finalmente se implementó para representar la superficie del terreno. Es un algoritmo de gráficos por computadora originalmente publicado en SIGGRAPH en 1987 por Lorensen y Cline. Se usa para crear superficies poligonales como representación de una isosuperficie de un campo escalar 3D. En concreto, dado un campo tridimensional escalar de valores numéricos en el que se encuentra un volumen, cada punto dentro del volumen teniendo un valor mayor o igual que la constante isolevel y cada punto fuera del volumen teniendo un valor de menos de isolevel, el algoritmo puede crear la superficie aproximada entre el volumen y las afueras colocando una malla de triángulos entre ellos. Este proceso es simplificado dividiendo el espacio en cubos que comparten caras, donde un lado de un cubo será cortado por la superficie si de los dos vertices que lo forman 1 se encuentra dentro del volumen y el otro fuera. Así, dado los 8 vértices del cubo, hay 256 (2^8) posibles combinaciones de polígonos dentro del cubo que corten los lados.

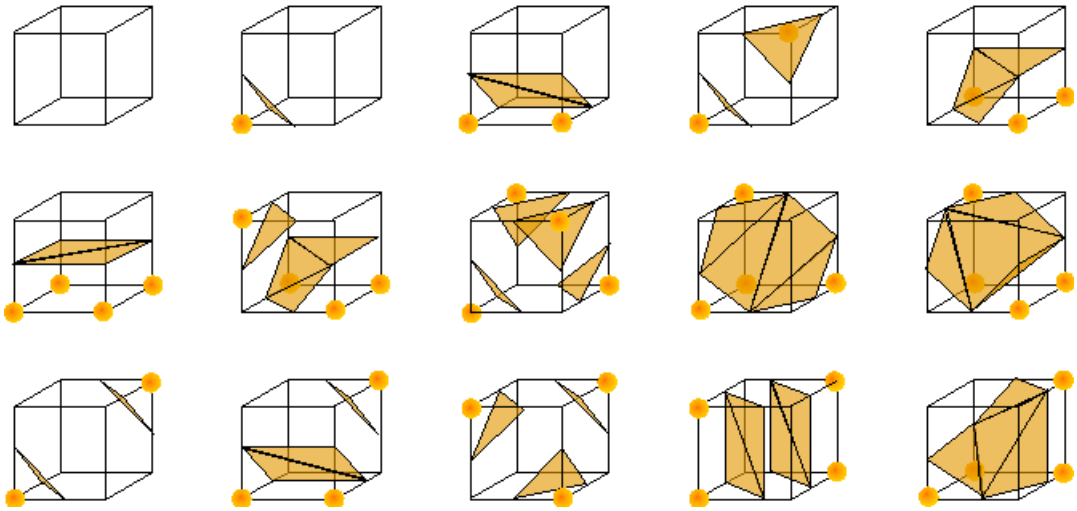

Figure 3: Los distintas casos posibles en un cubo (sin tener en cuenta orientación)

La dificultad a la que se enfrenta representar así un volumen es la gran cantidad de combinaciones posibles y la necesidad de que aquellas combinaciones conecten correctamente entre los cubos para formar la malla. Esto requeriría además bastante computación para generar los triángulos de cada cubo individual. Sin embargo, el algoritmo de cubos de marcha toma ventaja del numero limitado de combinaciones posibles en cada cubo, guardando todas estas posibilidades en lookup tables. Estos lookup tables guardan las 256 combinaciones posibles de triángulos y ahorran mucho tiempo de cómputo, acelerando la generación de las mallas.

Funcionamiento del algoritmo de marching cubes

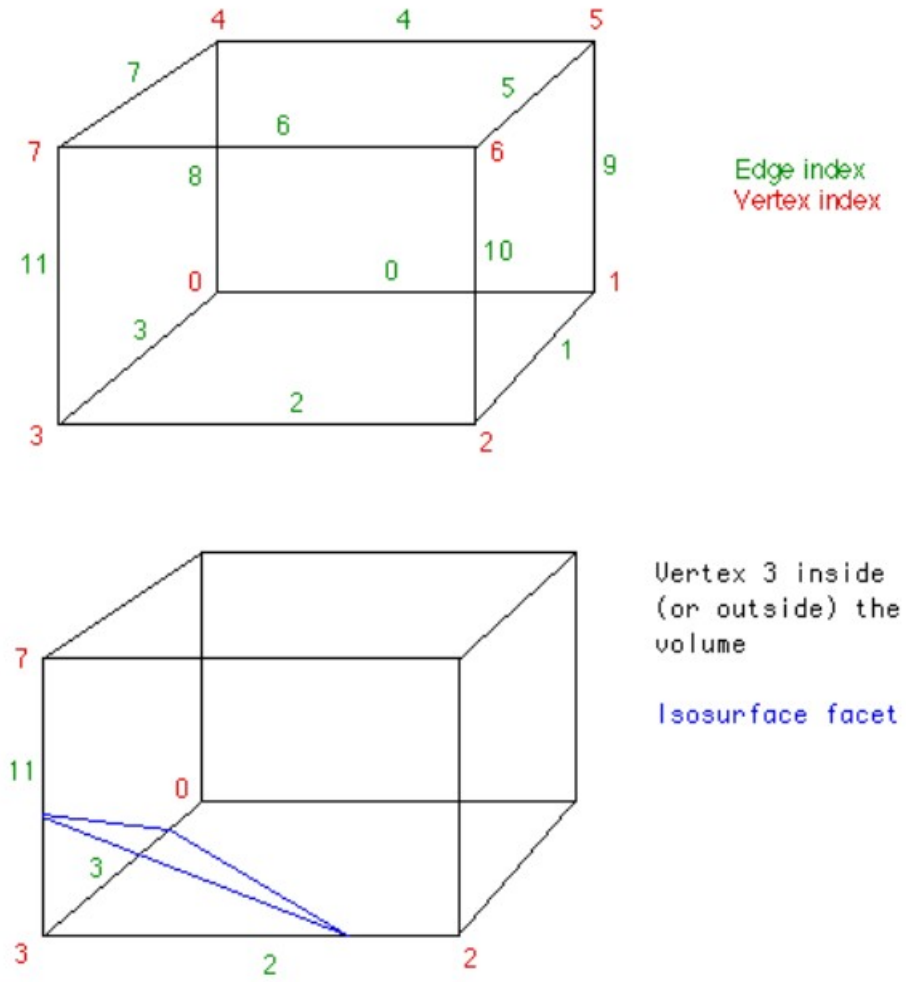

Figure 4: Ejemplo de un cubo en el que la superficie corta los lados 2, 3 y 11

Los pasos que sigue el algoritmo para formar la malla son los siguientes:

1. Para cada cubo, se obtiene los ejes cortados por la isosuperficie usando los valores de los vértices del cubo. Dado los vértices que se encuentran debajo del volumen, se obtiene un índice que se usa en la tabla de ejes para conseguir los ejes que la superficie del volumen corta. Por ejemplo, dado la figura 4, el índice correspondiente sería 0000 1000 o 8 en binario. Se obtiene de la siguiente forma:

```
cubeindex = 0;
if (cube.corner[0] < isolevel) cubeindex |= 1;
if (cube.corner[1] < isolevel) cubeindex |= 2;
if (cube.corner[2] < isolevel) cubeindex |= 4;
if (cube.corner[3] < isolevel) cubeindex |= 8;
if (cube.corner[4] < isolevel) cubeindex |= 16;
if (cube.corner[5] < isolevel) cubeindex |= 32;
if (cube.corner[6] < isolevel) cubeindex |= 64;
if (cube.corner[7] < isolevel) cubeindex |= 128;
```

CodeBlock 1: Código que obtiene el índice para un cubo

La tabla de ejes devuelve un número de 12 bits que representa los ejes del cubo. Para cada bit, si su valor es 1, significa que el eje respectivo es cortado por la superficie. Si su valor es 0, es que no es cortado. Volviendo al ejemplo de la imagen 1 [CHECK VALIDITY], al buscar en la tabla de ejes usando el índice 8, se obtendría el número 1000 0000 1100. Esto significa que los ejes cortados son el 2, el 3 y el 11 (el primer bit es el eje 0).

2. Se calculan los puntos de intersección en los ejes cortados. Esto se hace mediante interpolación linear a base de los valores de los dos vértices que forman el eje. Permite representar con más precisión el volumen, ya que no limita la malla a solo cortar en la parte media del eje de los cubos. Dado los puntos P1 y P2 que forman el eje cortado y sus valores escalares V1 y V2 respectivamente, el punto de intersección en el eje viene dado por:
$P = P1 + (isolevel - V1)(P2 - P1)/(V2 – V1)$.

3. Por último, se crean las facetas formadas por las posiciones por las que la isosuperficie corta a los cubos. Se usa una segunda tabla, llamada tabla de triángulos, que contiene un array de números para cada posible combinación de facetas en un cubo. Cada array contiene para cada triángulo de esa combinación los ejes en los que se encuentran los puntos que forman dicho triángulo. Cada array contiene a lo sumo 5 triángulos, por lo que puede tener una longitud máxima de 15 + 1 números (el último se usa para indicar cuando acaba el array). Los primeros 3 números del array indican el primero triangulo, los segundo 3 el segundo triángulo, etc. El algoritmo sigue creando facetas de triangulo dado un array hasta que se encuentra con un valor -1. El índice ya obtenido en el primer paso del algoritmo se usa también en esta tabla.

   Siguiendo el ejemplo anterior, el índice 8 en la tabla de triángulos apunta al array {3, 11, 2, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1}.

```
MallaTriangulos <- empty
Para cada cubo:
        índice <- obtenerIndice(cubo.vértices)
        ejes <- tablaEjes[índice]
        Para cada eje en ejes:
                P1 <- eje.P1; V1 <- cubo.valor(P1)
                P2 <- eje.P2; V2 <- cubo.valor(P2)
                eje.puntoCorte <- P1 + (isolevel - V1)(P2 - P1)/(V2 - V1)
        T <- tablaTriángulos[índice]
        Para i en [0..4]
                si T[i*3] == -1
                        BREAK
                P1 <- eje[T[i*3]]
                P2 <- eje[T[i*3 + 1]]
                P3 <- eje[T[i*3 + 2]]
                mallaTriangulo.añadirTriangulo({P1, P2, P3})
```

CodeBlock 2: Pseudocódigo del algoritmo de creación del terreno

Implementación del algoritmo de marching cubes