Geonosis
Memoria

URJC
2º Diseño y Desarrollo de Videojuegos
Programación Avanzada

Rodrigo Martínez Sánchez
Manuel Pérez Ramil

ÍNDICE

1) Introducción

Esta memoria contiene información acerca de las decisiones que hemos tomado para desarrollar la práctica final de Programación Avanzada, así como una explicación de los cambios realizados sobre el motor original. El diagrama de clases UML está incluído como un archivo adjunto debido a su tamaño.

Geonosis es un endless runner con temática espacial. El juego termina al colisionar con un obstáculo o al sobrevivir durante un tiempo determinado. Los principales problemas que nos hemos encontrado a la hora de plantear el desarrollo han sido el sistema de generación de obstáculos y el sistema de colisiones. En el siguiente apartado se explica cómo lo hemos implementado. Finalmente se incluye un pequeño apartado de conclusiones.

2) Implementación

Para desarrollar Geonosis hemos aprovechado el motor proporcionado en el aula virtual. Hemos intentado mantener la idea de las funciones que realizaba cada clase (Scene, Game, GameObject…) ya que estaba dividido de forma que era modular y fácil de entender. Hemos aplicado el concepto de polimorfismo para las colisiones, delegando esta función a los propios objetos para que cada uno lo compruebe dependiendo de su forma. En una partida, la nave del jugador se desplaza lateralmente mientras que los objetos se dirigen hacia ella. Al salirse de los límites del escenario (por detrás de la cámara), los obstáculos son reubicados delante del jugador.

Las principales modificaciones han sido las siguientes:

- Para crear objetos aleatorios que se dirijan hacia el jugador hemos decidido modificar el update de Scene. Siguiendo la idea enseñada en clase de delegar el control de la posición de los sólidos a la escena (sólidos rebotan al chocar con los bordes), programamos a Scene para que cada vez que un sólido llegue a una posición detrás de nuestro jugador y de la cámara, se posicione delante de nuevo en una posición aleatoria (dentro de un rango). Con esto únicamente necesitamos añadir a la hora de crear una Scene una serie de obstáculos ya que estos se reutilizarán.

- Para comprobar si ha habido una colisión hemos decidido utilizar polimorfismo. Las colisiones son calculadas en tiempo real teniendo en cuenta un espacio 2D y las áreas simplificadas de la nave y de los obstáculos. Existen 2 formas con las que se puede colisionar: cuadrados y círculos. La nave está representada por 5 puntos en el espacio. Para implementar esto creamos la función abstracta checkCollision(float z, float x) en GameObject (del cual heredan los obstáculos). Esta función recibe como parámetro un punto en el espacio 2D y devuelve un booleano que representa si ese punto se encuentra dentro del área del obstáculo. La clase Cube y la clase Cylinder implementan esta función con sus particulares formas (cube comprueba colisión con un área cuadrada y cylinder comprueba colisión con un área circular). Gracias al polimorfismo, en el update de Scene podemos almacenar un vector de GameObject para llamar a esta función de la misma manera para cada obstáculo y que cada uno ejecute la suya dependiendo de si es un Cylinder o un Cube.

A continuación se incluye un listado de las clases usadas, modificadas e incluídas:

- **Engine**: *sin modificar*
- **Game** *modificado*
  Hemos modificado este archivo pero guardando la idea original de utilizarlo únicamente para contener la escena a ejecutar, asignándole los objetos que que queremos que contenga y llamando a sus métodos render(), update() y processKeyPressed().
- **GameObject** *modificado*
  Esta clase se modificó para poder utilizar el polimorfismo a la hora de comprobar las colisiones. Contiene un atributo booleano collidable que determina si el objeto puede comprobar colisiones o no. Contiene también el método abstracto checkCollision(float z, float x) comentado anteriormente, que comprueba si un punto está dentro del área del objeto. También se modificaron los constructores en consecuencia y se añadieron getters & setters para el atributo collidable. Además, se añadio una funcion LoadModel() la cual obtiene el nombre de un archivo .obj y te lo renderiza en el objeto que lo cargue
- **Camera** *modificado*
  Implementa el método checkCollision() para devolver siempre false.
- **FlyingCamera** *sin modificar*
- **Color, Vector3D** *sin modificar*
- **Cube, Cylinder** *modificados*
  Ambos objetos implementan ahora cada uno con sus particularidades la función abstracta checkCollision() heredada de GameObject. Cube comprueba si las coordenadas z, x pasadas como parámetro se encuentran dentro de un área cuadrada (con el tamaño de lado dependiente del tamaño del cubo y el centro dependiente de la posición del cubo). Cylinder comprueba si las coordenadas pasadas como parámetro se encuentran dentro de un área definida por el radio del cilindro y su posición x, z.

- **Player** *añadido*
  Esta clase que hereda de GameObject representa a un jugador y tiene dos particularidades. La primera es que guarda un array de floats, que almacena un número de puntos 2D (primero coordenada x, segundo coordenada z) que representa a la nave para simplificar las colisiones. Contiene también sus getters & setters.
  La segunda particularidad es que contiene un método processKeyPressed() similar al de FlyingCamera que actualiza la posición Z del objeto, para permitir el movimiento.
- **Scene** *modificado*
  Esta clase ha sufrido numerosas modificaciones. En cuanto a atributos se ha añadido un puntero a Player y se ha añadido un nuevo vector de GameObject, utilizando este nuevo únicamente para los obstáculos. Esto es para aplicarle una velocidad creciente únicamente a los obstáculos con el paso del tiempo, además de para optimizar al no tener que llamar a la función checkCollision() en GameObjects que sabemos que no harán colisión (entorno).
  Otra modificación es que, a la hora del update, se llama para todos los GameObject del array de obstáculos su función checkCollision(), tantas veces como puntos tenga Player. Esta clase también se ocupa si ha habido colisión con un obstáculo y se ha perdido, permitiendo el reintento; y se encarga de terminar el juego cuando se termina el tiempo, ofreciendo también un reintento.
  Finalmente, la función processKeyPressed() llama a la función processKeyPressed de la instancia de Player.
- **Color, Vector3D** *sin modificar*
- **Sphere, Teapot, Torus** *eliminados*

3) Conclusiones

Tras esta práctica hemos podido comprobar la utilidad de la herencia y del polimorfismo. Gracias a la herencia reutilizamos y hacemos más fácil de entender muchas partes del código. Además, también se facilita la modificación de las clases ya que con una modificación a GameObject, estábamos modificando también Camera, FlyingCamera, Cube, Cuboid, Cylinder y Player.

El polimorfismo también nos ha valido para simplificar nuestro código. Al poder tratar a diferentes clases de una misma manera utilizando funciones virtuales, simplificamos tanto la escritura como la lectura del código eliminando en nuestro caso el tratamiento por separado de objetos Cube y Cylinder.

Incluímos el enlace a nuestro github https://github.com/Prothoky/PA-2019 ya que no hemos podido subir el proyecto entero al aula virtual debido al espacio que ocupa. En el aula virtual están los archivos .cpp y .h, .obj y los pdf.

4) Bibliografía

Tanto como para el desarrollo de la práctica como para el de la memoria y el GCD hemos utilizado el material proporcionado en el aula virtual: las diapositivas y el código del motor.