Manual de Proyecto Wolfenstein 3D

[75.42-95.08] Taller de Programación I
Cátedra Veiga
Segundo cuatrimestre de 2020

Rosenblatt, Kovnat, Fontela, Betz
104105, 104429, 103924, 104348

Índice

1. Enunciado

El presente trabajo práctico consistió en hacer un remake del Wolfenstein 3D, desarrollado en 1992 por ID Software. El mismo es un juego del genero *First Person Shooter* y fue el precursor de la tecnología 3D moderna. Utilizando el algoritmo de Raycasting, se traslada un mapa 2D a una ilusion de 3D tomando la distancia que viaja cada rayo hasta chocar con una pared en el plano 2D. El juego originalmente era para un unico jugador de forma offline, sin embargo en este trabajo se recreo el juego tomando al juego como multiplayer online, utilizando sockets TCP bloqueantes.

El trabajo fue realizando utilizando el lenguaje C++ y el paradigma de la programación orientada a objetos, así como la programación concurrente utilizando multiples hilos de ejecución.

2. División de Tareas

Durante el trabajo no hubo una gran división de tareas ya que los límites de lo que hacía cada uno resultó ser bastante difuso. Rosenblatt desarrolló la mayoría del cliente y dio una mano con el debugging del mismo, la pantalla de login (preparada con Qt) y el servidor. Kovnat aportó mucho código para el servidor y ayudó al desarrollo con el cliente (tanto el Login como el resto del mismo). El editor fue escrito completamente por Betz y Fontela quienes aportaron valiosa ayuda en el cliente y servidor y parseo de archivos yaml respectivamente. Cada uno además documentó la parte del código con la que se siente mas cómodo y más desarrolló. Mucho del trabajo se realizó siguiendo la modalidad de Pair-Programming, diviendonos en grupos de dos cada uno con su tarea particular para ese dia. De esta manera, evitamos varios bugs teniendo una supervision constante del codigo que escribíamos.

3. Evolución del proyecto

Durante la primera semana, tuvimos varias reuniones pensando en como resolver el problema, planteando un modelo que sea eficiente y facilmente ampliable, ya que en un futuro se podría agregar funcionalidad extra, como, por ejemplo, armas nuevas. Ademas, intentamos familiarizarnos con las liberias gráficas SDL y Qt que son esenciales para el funcionamiento del cliente y del editor.

3.1. Cliente

Mucho del tiempo inicial utilizado para el cliente fue usado en resolver el algoritmo de Raycaster. Esto nos tomo bastante tiempo dado que se nos presentaron varias formas de hacerlo, pero ninguna terminaba de explicar por completo el funcionamiento del sistema, ni como acoplarle muchas de las cosas que luego necesitariamos (Animaciones, Sprites, etc) Sin embargo, luego de unas semanas de trabajo pudimos resolver las ecuaciones y plantear un sistema que se acople a nuestras necesidades.

Una vez que resolvimos esos problemas, fuimos agregando las funcionalidades mas simples para luego pasar a las mas complejas. Comenzamos agregando sprites basicos sin posibilidad de movimiento (Un barril por ejemplo) para luego pasar a sprites que podian moverse (Un jugador) y finalmente terminar agregando las animaciones.

Una vez resuelto los temas gráficos, terminamos agregando los sonidos a las acciones que ocurrian en el juego, modularizando su volumen por la distancia al jugador que controla el cliente.

Finalmente, hicimos un paso de optimización para aumentar la cantidad de frames a los que podia correr el juego. Investigamos sobre flags de compilación, costos temporales de operaciones sobre estructuras de datos de STL y como optimizar su uso. De esta manera pudimos aumentar fuertemente la cantidad de frames a los que puede correr el cliente.

3.2. Integración con Lua

Una vez que teniamos terminado el modelo de funcionamiento de la comunicacion entre servidor y cliente, pasamos a implementar un sistema con el cual se puedan correr scripts de Lua y que el servidor lo tome como un jugador mas. Comenzamos implementando un modulo en C++ que tome un input aleatorio que provee un script de Lua y este se lo envie al servidor como haria un cliente normal. Luego, le implementamos un algoritmo de raycasting al modulo de Lua que modifica un objeto GameState que tiene una representacion de lo que el jugador controlado por Lua podria ver por pantalla. Finalmente, creamos una API para que el script de Lua pueda extraer información necesaria para tomar decisiones sobre el input a enviar. Con esto tuvimos una integración completa y decidimos crear un script que simplemente se mueva en posiciones aleatorias y que cuando vea a un jugador en su pantalla le dispare hasta que este muera o bien este se vaya de su pantalla.

3.3. Servidor

El primer paso del servidor fue resolver los temas de conectividad, es decir, que varios jugadores puedan conectarse al servidor que estara escuchando en un puerto especificado por el usuario, y que dichos usuarios puedan especificar a que partida se querian unir, dado que el servidor es capaz de gestionar multiples partidas.

Una vez resuelto eso, pudimos pasar a una pequeña prueba de concepto en la que teniamos dos jugadores representandos por cuadrados 2D en la pantalla del cliente que podian moverse y el otro jugador podria ver el movimiento del jugador rival.

La tercera semana la ocupamos pensando en un sistema en el que el servidor pueda notificar apropiadamente a los clientes de todas las acciones que se tomaban en el modelo. Nos terminamos decidiendo a hacer un sistema en el que el input del usuario cree un comando que el servidor podria ejecutar sobre el modelo del juego, y de ser necesario el servidor crearia una notificación que es enviada a todos los clientes. Para esto creamos una clase Engine que seria el encargado de ejecutar estos comandos sobre el modelo y de enviar las notificaciones que se crean, ademas de simular el paso del tiempo.

Una vez que tuvimos la base del motor del juego, fuimos agregando con el paso de las semanas mas funcionalidades que eran requeridas en el modelo. Empezamos con las cosas mas basicas, como el chequeo del movimiento del jugador, para luego agregar los items que el jugador podria agarrar del piso y finalmente agregar las cosas mas complejas como son el chequeo de los disparos del jugador, los cohetes que estos pueden lanzar y los tiempos de las puertas.

Finalmente, agregamos la funcionalidad de poder importas las constantes numericas de un archivo de configuración YAML para poder balancear mas facilmente el juego sin la necesidad de recompilar el trabajo.

3.4. Editor

Lo primero que hicimos fue aprender como usar qt para esto pasamos un tiempo viendo turoriales y practicando con proyectos menores para aprender a usar features particulares.

Despues de tener una idea general del funcionamiento de qt y qt designer empezamos con el editor. El primer paso fue crear un editor base que solo funcionaba con un solo tipo de tile para asegurarnos de que la logica por la cual se creaba la matriz y se imprimiesen las cosas en la pantalla sea correcta.

El segundo paso fue generalizarlo agregando todos los tileset, los puntos de respawns y el borrador.

El tercer paso fue crear las ventanas de nuevo, guardar y abrir con toda su logica. Este paso permitio tener una primer version funcional del editor.

Para poder crear mapas grandes con mas facilidad y para poder ver bien algunos items pequeños le agregamos un zoom in y un zoom out.

Para terminar solucionamos algunos problemas de memoria que nos habian quedado pendientes.

4. Análisis de puntos pendientes

A pesar de que pudimos completar la funcionalidad pedida por el enunciado, hay algunas cosas que nos gustaria poder haber agregado si disponiamos del tiempo para concretarlo. Algunas de estas ideas son:

- **Mejorar la colisión del jugador con las esquinas del mapa.** Inicialmente, esto era un error mas frecuente dado que el movimiento se calculaba como como un paso gigante entre dos posiciones que despues se optimizo haciendo quarter-steps entre la posición inicial y la final. Sin embargo, como el jugador en su representación 2D es un simple punto en un plano, al pasar por ciertas esquinas en puntos muy particulares, en el cliente da la impresión de que un costado de la camara atraviesa una pared. Esto lo podriamos haber resuelto tomando al jugador como un segmento perpendicular al vector dirección y al moverse chequear los extremos y el centro de ese segmento para colisiones.
- **Agregar sprites direccionales al cliente.** Esto fue un item al que le dedicamos bastante tiempo pero tuvimos que seguir adelante porque no pudimos encontrarle una solución. Esto le daria una sensacion mas realista al cliente a la hora de jugar, aunque no tenerlos no afecta gravemente la jugabilidad del mismo.
- **Identificar a los jugador por un nombre que ellos indiquen.** Por el momento, los jugadores estan diferenciados por un ID unico que les genera el servidor, pero hubiese estado bueno poder tener el tiempo para agregarles una identificación mas customizable como seria un ID propio.
- **Hacer un script de Lua que sea mas inteligente.** A pesar de que solo se pedía la integración, podriamos haber agregado algunos chequeos a la API que le permitan al script de Lua tener una especie de auto-aim ya que podria ajustar su punteria dependiendo de en que columna de la pantalla el jugador es dibujado. Por el momento, el modulo simplemente dispara a la vista de un jugador, y no tiene en cuenta su posición relativa en la pantalla.

5. Incovenientes Encontrados

Thiago Kovnat: Personalmente, al ser el primer proyecto de gran tamaño en el que trabajé, me mareaba un poco el volumen de los archivos con los que había que trabajar y como funcionaban todos como un conjunto. Principalmente el tener una imagen de como funciona el sistema en general y como un cambio chico en alguna parte te fuerza a cambiar muchas cosas en el sistema cuando despues quieras o necesites implementarle algun feature nuevo. Luego de unas semanas me fui acostumbrando a esto y a pensar en adelantado para contemplar cambios a futuro en el modelo.

Jonathan Rosenblatt: Mi gran inconveniente con este trabajo es el tiempo que consume. Al ser un trabajo tan grande un pequeño glitch puede desencadenar un problemón que puede tardarse horas en resolver. Entre eso y el gran volumen de archivos que hay que manejar, hace que se deba dedicar muchísimo tiempo para realizar el proyecto. También saltan muchas complicaciones secundarias, como entender la matemática involucrada en la construcción del sistema (álgebra vectorial y mucha geometría en el caso del motor gráfico) o aprender a utilizar herramientas de edición de imagen para obtener y modificar los sprites requeridos, entre otros. Esto, por complicado que sea de adquirir en un principio, resulta ser un beneficio por todas las herramientas y experiencias adquiridas.

Joaquin Fontela: Mis mayores inconvenientes al momento de realizar el trabajo, habiendome ocupado casi totalmente del parseo de los archivos '.yaml' (desde conseguir un parser en internet, incluir las lineas necesarias en el CMake y crear las clases correspondientes que accedieran al parser), estuvieron por este lado. Una de las mayores dificultades la tuve preparando el parser de los archivos '.yaml' para "mergearlo.ªl proyecto, ya que requeria incluir algunas lineas de comandos en

el CMake que me costo bastante encontrar en internet. Por otro lado, se hizo tedioso el debugging de las clases que parsean los archivos ya que muchas veces no podia probar algunas implementaciones hasta escribir un gran pedazo de codigo que, evidentemente, casi siempre presentaba algun error. Luego tuve que encargarme de enviarle al editor la informacion correspondiente para la lectura de los archivos '.yaml' que representaban un mapa, y viceversa (es decir, capturar la informacion del editor para convertirla en un archivo de la extension en cuestion). Esto generaba, nuevamente, errores tediosos, ya que no eran errores graves de concepto, pero al ser gran cantidad de cosas a tener en cuenta, se pasaba mucho tiempo haciendo debugging, agregando y quitando 'std::cout', etc.

Joaquin Betz: Mi mayor inconveniente a la hora de realizar el trabajo fue mi falta de experiencia haciendo interfaces graficas ya que en los anteriores trabajos practicos de otras materias que lo requerian yo me encargue de otros temas. Esto me trajo algunos problemas ya que fui aprendiendo sobre la marcha y tuve que cambiar el modelo varias veces hasta encontrar uno que funcione correctamente.

6. Herramientas

Para el desarrollo del trabajo utilizamos las siguientes herramientas:

- Como alojamiento del proyecto y control de versiones, utilizamos Git (en GitHub)
- Para editar el codigo fuente, utilizamos Visual Studio Code para documentar (dado que nos provee buenos plug-ins que facilitan) y Atom para la edicion del codigo en si (Dado que tiene una funcionalidad de pair-programming en el cual varias personas pueden modificar el codigo que presente un usuario)
- Para generar archivos de compilación utilizamos CMake.
- Para depurar, utilizamos GDB.
- Para encontrar leaks de memoria, utilizamos la herramiente MemCheck de Valgrind.
- Para librerias graficas, utilizamos SDL2 y Qt5.
- Para librerias de parsing de YAML, utilizamos YAML-CPP.
- Para manejo de sprites utilizamos Photoshop y www.spriters-resource.com

7. Conclusiones

En conclusión, fue un muy buen trabajo práctico que nos enseñó muchas cosas, desde como trabajar como un conjunto y dividir eficientemente las tareas hasta como manejar el versionado de un proyecto con un volumen mayor al que estamos acostumbrados con el resto de los trabajos prácticos con los que veniamos trabajando. Terminamos teniendo un trabajo muy completo para presentar a terceros. Sentimos que en general el trabajo quedó muy bien y disfrutamos haciendolo.