FACULTAD DE INGENIERIA
Universidad de Buenos Aires

Taller de Programación I (75.42)
Facultad de Ingeniería
Universidad de Buenos Aires

Wolfenstein 3D

Ejercicio Final

Manual de Proyecto

2º Cuatrimestre 2020

Primera Entrega: 09/02/2021

Segunda Entrega: 23/02/2021

Alumnos	Padrón
Carretero Andres	101004
Montiel Ramiro	100870
Parente Gaston	101516
Roldan Maria Cecilia	101939

Índice

Enunciado

Al momento de realizar el trabajo se nos pidió, luego de una larga intriga, realizar un trabajo práctico final grupal que consistía en hacer una Remake del mítico juego Wolfenstein 92 con algunas variaciones.

Lo pedido fue que el juego tuviera los mismos gráficos, pero esta vez que fuera multijugador online (ahora sería un shooter en 1ra persona todos contra todos), como puntos fundamentales, cada jugador debía poder cambiar de arma (luego de agarrarla del piso) y este debía ser visto desde el punto de vista de otros jugadores con un skin distinto.

El resto del juego debía ser similar al original, se tuvo que generar los gráficos utilizando la técnica de ray casting, los jugadores deben poder agarrar armas del piso, abrir puertas, abrir pasadizos, curarse pasando por encima de botiquines, entre otras cosas. El juego debe contar con sonidos para brindar una mayor experiencia de usuario.

Como último punto también se pidió que se creara un programa aparte que debía ser un editor de mapas, para poder así cada uno editar el mapa que quisiera y poder jugarlo.

El enunciado completo se encuentra en el siguiente link:
https://09375403727218988247.googlegroups.com/attach/fcd76861df3c/enunciado.pdf?part=0.1&view=1&vt=ANaJVrHCt5y5uiOy5GM_9k_puD6r0bomW6aTGOTyL7ObPVCWpuSI6WFA0SM1jf3ma0rE8bbDZQveYybIXP6teO5ptkjifh_vupmnxtQ1jgcu38MmldPpxdQ

División de tareas

Al momento de dividir las tareas optamos por seguir la recomendación brindada por los docentes en el enunciado del trabajo práctico, adaptándolo al 4to integrante:

	Alumno 1 (Montiel)	Alumno 2 (Carretero)	Alumno 3 (Parente)	Alumno 4 (Roldán)
Semana 1 (24/11/2021)	Carga de mapas. Lógica de movimiento de los personajes (colisión con las paredes y otros objetos).	Mostrar una imagen. Mostrar una animación. Ecuación de los rayos (donde intersecciona con cada celda)	Reproducir sonidos, música. Mostrar texto en pantalla. Draft del editor.	No formaba parte del equipo.
Semana 2 (01/12/2020)	Modelo Proxy	Vista 3D las paredes usando ray casting. Sin texturas ni objetos ni otros jugadores	Editor básico	Lógica de las partidas. Lógica del ataque. Módulo IA básico.
Semana 3 (08/12/2020)	Sistema de comunicación (cliente -servidor)	Vista 3D incluyendo las texturas de las paredes, objetos y jugadores.	Editor completo	Módulo IA completo.
Semana 4 (15/12/2020)	Servidor completo. Configuración.	Cliente completo.	Pantalla de login y de partidas.	Integración con cliente y servidor
Semana 5(09/02/2021)	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar

Semana 6 (16/02/2021)	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar
Semana 7 (23/02/2021)	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar	- Pruebas y corrección sobre estabilidad del servidor.- Detalles finales y documentación preliminar

Evolución del proyecto

El cronograma propuesto fue el mismo que el planteado en la división de tareas.
A continuación veremos cómo resultó el cronograma real:

	Alumno 1 (Montiel)	Alumno 2 (Carretero)	Alumno 3 (Parente)	Alumno 4 (Roldán)
Semana 1 (24/11/2021)	Lógica de movimiento de los personajes (colisión con las paredes y otros objetos).	Mostrar una imagen. Mostrar una animación. Ecuación de los rayos (donde intersecciona con cada celda)	Reproducir sonidos, música. Mostrar texto en pantalla. Draft del editor.	No formaba parte del equipo.
Semana 2 (01/12/2020)	Modelo Proxy	Vista 3D las paredes usando ray casting. Sin texturas ni objetos ni otros jugadores	Editor básico	Lógica de las partidas.
Semana 3 (08/12/2020)	Sistema de comunicación (cliente -servidor)	Vista 3D incluyendo las texturas de las paredes, objetos y jugadores.	Editor avanzado	Lógica del ataque.
Semana 4 (15/12/2020)	Servidor y cliente avanzado.	Vista 3D incluyendo las texturas de las paredes, objetos y jugadores.	Editor avanzado	Módulo IA básico.
Semana 5(22/12/2021)	Servidor y cliente avanzado.	Vista 3D incluyendo las texturas de las paredes, objetos y jugadores.	Editor completo	Módulo IA completo.
Semana 6 (29/12/2021)	Vacaciones	Vacaciones	Vacaciones	Vacaciones

Semana 7 (05/01/2021)	Vacaciones	Vacaciones	Vacaciones	Vacaciones
Semana 8 (12/01/2021)	Vacaciones	Vacaciones	Vacaciones	Vacaciones
Semana 9 (19/01/2021)	Integración con cliente y servidor	Integración con cliente y servidor	Pantalla de login y de partidas.	Configuración Integración cliente y servidor
Semana 10 (26/01/2021)	Agregado de audios al juego	Integración con cliente y servidor	Corrección de errores y re-factor de editor	Configuración Integración cliente y servidor
Semana 11 (02/02/2021)	Protocolos de comunicación en fin de juego	Pantalla de fin de juego	Corrección de errores y re-factor de editor	Animación de Lanzacohetes
Semana 12 (09/02/2021)	Ajustes finales y corrección de errores	Ajustes finales y corrección de errores	Ajustes finales y corrección de errores	Ajustes finales y corrección de errores
Semana 13 (16/02/2021)	Ajustes finales y corrección de errores	Ajustes finales y corrección de errores	Ajustes finales y corrección de errores	Ajustes finales y corrección de errores
Semana 14 (23/02/2021)	Ajustes finales, corrección de errores y Documentación	Ajustes finales, corrección de errores y Documentación	Ajustes finales, corrección de errores y Documentación	Ajustes finales, corrección de errores y Documentación

Inconvenientes encontrados

¿El server procesa el juego ó el cliente?

Al iniciar el proyecto nos topamos con esta problemática, al ser nuestro primer juego en red ninguno de los integrantes sabíamos cómo encarar esto. Primero pensamos en que el cliente realizará toda la lógica del juego, mientras que el servidor simplemente fuera un transmisor de mensajes entre todos los clientes, esto nunca lo implementamos ya que luego, pensándolo mejor, esa estrategias podría traernos problemas, más que nada enfocados en el tiempo de procesamiento de cada cliente en particular (en general corriendo en máquinas distintas) lo que podría llevar a una sincronización total entre todos los clientes.
La opción que tomamos fue darle el papel de procesamiento del juego al servidor, éste se encarga entonces de recibir los paquetes con protocolos enviados desde los cliente, actualiza el modelo interno del juego y responde a todos los clientes que se vieron afectados por ese cambio para que actualicen su propio modelo (el cual simplemente muestra gráficos). De esta forma obtuvimos un cliente con poca lógica, pero muy centrado en la parte gráfica y por otro lado un servidor 100% abstracto que les manda información a todos los clientes.

¿Qué formato le dabamos al protocolo?

Otro de los grandes problemas que nos encontramos fue a la hora de planificar el protocolo propietario que debíamos desarrollar para comunicar a los clientes con el servidor. Se plantearon muchas alternativas, entre ellas crear protocolos de longitud variable, enviar protocolos con todos los cambios ocurridos cada cierto tiempo desde el servidor, crear protocolos que tuvieran strings pero que al enviarlos los mismos fueran casteados para su posterior envío y que por parte del cliente fueran recibidos y regenerado a strings y viceversa. Ninguna de estas alternativas nos terminaba de convencer.
Luego de pensarlo nos determinamos por una solución, la que nosotros consideramos la más simple y eficiente: ideamos un protocolo que no utiliza strings y que trata de ser lo más minimalista posible.
Terminamos creando un protocolo con solamente 6 campos, 5 de los cuales fueron uints16 (que fueron muy fáciles de transmitir por red utilizando las funciones htons y ntohs) y un atributo float. Utilizando la directiva __attribute__((packed)) en total solamente utiliza 14 Bytes lo que lo vuelve muy ágil al momento de ser enviado y recibido.
La decisión tomada en cuanto al procesamiento de dichos protocolos fue que ni bien el servidor los recibiera los procesara y enviara un protocolo de respuesta a todos los clientes que fueran afectados por este para que de esta forma todos actualizan sus modelos y mostraran los cambios.
Gracias a esto logramos un juego con muy bajo lag.

Ray Casting

Si bien en la teoría el Ray Casting es sencillo, ponerlo en práctica no lo fue tanto, dado que constaba de pequeños pero múltiples detalles, todos importantes.
Por ejemplo:

Al crear cada rayo, éste tiene un ángulo relativo al jugador, dado que este tiene una vista de 90 grados, los ángulos van desde -45° a 45° respecto de la dirección relativa hacia la que está 'apuntando' el jugador, y éstos luego hay que rotarlos a la dirección real a la que apuntan en el mapa.

Cuando un rayo choca contra un casillero nuevo y tengo que fijarme que hay en él, obtengo una coordenada por ejemplo (4.5; 5), pero éste delimita 2 casilleros, el casillero (4;5) y (4;4), y para determinar en cuál de ellos estaba y en cual me tengo que fijar qué hay ahora necesito saber la dirección del rayo. Por lo que no importa solo las coordenadas del casillero con el que choqué, sino también la dirección de mi rayo para determinar en qué casillero me tengo que fijar. siguiendo el ejemplo, si la dirección del rayo es (0;1), quiere decir que estaba en la posición (4;4), por lo que ahora tengo que ver que hay en la (4;5), y caso contrario si la dirección es (0;-1).

Al chocar contra una puerta, el rayo tiene que avanzar 0.5 unidades más en la dirección con la que chocó para determinar exactamente contra qué pixel choca, dado que las puertas están "hundidas" en el casillero en lugar de mostrarse en todo el borde de los casilleros como las paredes, esto, además es necesario comprobar que tanto de la puerta es visible (si se está abriendo o cerrando), y en caso de que se quiera mostrar una parte de la puerta que ya no está, el rayo tiene que seguir en busca de otros objetos del mapa.

Además, ésta manera de ver la puerta presenta problemas cuando se la vé de costado, la solución a esto fue restringir el creador de mapas para que no se pueda crear una puerta a menos que esté entre 2 paredes, y más aún, cuando el jugador muere y entra en "modo fantasma" (capaz de atravesar objetos), se decidió directamente que deje de ver las puertas para impedir que ocurra este problema.

Sprites

Si bien fue más complejo desarrollar la parte de raycasting, para éste se tenía muchas referencias, ejemplos, y demás fuentes que permitían saber con precisión cómo hacerlo para que funcione correctamente. No fue este el caso para dibujar los sprites que, si bien fue poco el código que requirió, fue difícil saber qué codear. Tras muchos intentos fallidos se determinó que la mejor manera era determinar el rayo "central", es decir qué número de rayo del raycasting era el que pasaba más cerca del centro del sprite (su coordenada específica en punto flotante), y a partir de ese rayo central, se vuelve a tirar una cantidad de rayos proporcional a la distancia del sprite al jugador para cada lado del sprite, entre más cerca esté el jugador del sprite, más rayos tiro(donde cada rayo representará una franja de pixeles en el eje X). Y al final termino graficando el sprite desde el primer al ultimo rayo que tiré cuya distancia del jugador al sprite no fue mayor a la distancia del jugador al objeto (pared o puerta), con el que choqué durante el raycasting. Estas distancias se cargan en un vector durante el mismo raycasting.

Textures

Fue necesario setear la posición y tamaño de cada textura en valores relativos al ancho y al alto del tamaño de la pantalla, para que estos se mantengan al modificar el tamaño de la pantalla o entrar en modo fullscreen en medio de la partida.

¿Quién controla los objetos de los eventos?

Hay pocos eventos en el juego, uno para la finalización del juego por el tiempo, dos eventos para controlar las puertas y uno para mover los misiles del lanzacohetes. Sin embargo al momento de controlar los objetos contenidos en el mapa nos dimos cuenta que era peligroso cambiar el estado o moverlos desde los eventos, ya que se estarían ejecutando desde otro thread y eso podría terminar en race conditions. La forma que encontramos de solucionarlos fue, en vez de controlar los objetos desde el thread de eventos, enviar protocolos a la cola del modelo de juego indicando de alguna forma el objeto específico a modificar. De esta forma el juego los controla desde un solo thread.

SDL

SDL fue una gran herramienta gráfica, sin embargo dado la enorme cantidad de errores que daba con valgrind, aun usando un supressor, dificultó el control del uso de memoria dinámica. Ante esto se puso especial atención en el manejo de la misma, ademas de reducir su uso al minimo, utilizandola nada mas para 3 tipos de objetos: enemigos, sprites, y misiles. Cada uno de estos almacenado en su respectivo vector de punteros en GameModelClient y teniendo muy pocos metodos que liberen su memoria además del destructor.

Editor:

Para desarrollar el editor, optamos por utilizar QT, teniendo en cuenta que es ideal para realizar aplicaciones de escritorio.
Sin embargo, tuvimos inconvenientes para decidir qué estructuras utilizar para generar el mapa y para cada caso de uso, ya que QT ofrece distintas variantes para crear “grids” o “tables”.

- Comenzamos utilizando QTableView, que ofrecía las facilidades de otorgarnos una vista de tabla muy fácil, pero sin embargo, la posibilidad de implementar el drag & drop y point & click se veían perjudicadas por el funcionamiento del mismo. Entonces, finalmente luego de varias pruebas, optamos por utilizar QGridLayout. El mismo, funciona como una grilla, donde se pueden insertar elementos. Lo que hicimos para poder almacenar la información de los objetos con lo que se edita el mapa, es generar 2 QLabel por cada posición, uno encargado de guardar la imagen, y otro encargado de guardar que objeto es.

- La implementación del Drag & Drop también nos generó varios inconvenientes. Hacer viajar la data desde un QWidget hacia otro, no nos resultó sencillo y requirió de muchas pruebas diferentes.
- La implementación del Point & Click, también fue difícil teniendo en cuenta que comenzamos trabajando con la idea de guardar el recurso que era clickeado en la clase, y luego copiarlo en los lugares donde se clickeaba. Esta solución, no era del todo eficiente y era complejo de implementar puesto que había que tener varias opciones en cuenta. Lo que terminamos utilizando, fue el Clipboard de QT, que permite almacenar información durante un tiempo hasta que se limpie. De esa manera, al clickear, lo único que hacemos es guardar la información en el Clipboard y cambiar el cursor.
- La implementación del ScrollBar, comenzó siendo un dolor de cabeza cuando utilizamos QTableView. Una vez que reemplazamos esta tecnología por QGridLayout, se solucionó el problema.

Los problemas que tuvimos, principalmente, se basaron en que clases de QT utilizar para caso de uso, lo que nos llevó mucho tiempo porque tuvimos que probar las diferentes posibilidades y optar por la que consideramos más ajustable a nuestro caso.

Lua

Lua es un lenguaje fácil de entender, por lo tanto se aprende rápido. Sin embargo la búsqueda de información para implementar el módulo fue más larga. Lo más difícil fue encontrar la manera en la que intercambian datos entre los archivos lua y C++.
Otro inconveniente es el manejo de errores de lua, a veces fallaba al cargar el archivo y eran causados por errores en la sintaxis, sólo se podían encontrar al imprimir la salida de la función de C++ que lo cargaba. Otras veces fallaba cuando llamaba a funciones propias del módulo, siempre retornando el mismo resultado, eran errores de código que saltaban imprimiendo en cada paso los valores de las variables, por no poder debuggearlo.
La última cuestión es que para actualizar la información de los jugadores en el módulo lua se producían race conditions. El módulo necesita las posiciones de los jugadores, entre otras cosas, y al pedir esa información desde otro thread las race conditions eran inevitables. Se agregaron variables atómicas para evitar esto.

Análisis de puntos pendientes

Nos quedaron pendientes algunas animaciones, sonidos, mayor claridad en los errores del lado del cliente y mejorar la fuente utilizada en el cliente, pero se cumplió con el objetivo pedido por el enunciado.

Herramientas

Al momento de desarrollar, algunos de los integrantes utilizamos como IDE Visual studio Code, otros CLion…
Para debuguear el programa utilizamos la herramienta que brinda VS Code con la cual fue muy fácil encontrar segmentation faults y todo tipo de errores.
También utilizamos Valgrind para debuguear y principalmente para verificar que nuestros programas (tanto el cliente como el servidor y el editor) no tuvieran errores de memoria que sean de nuestra responsabilidad.
Como programa para el control de cambios utilizamos Github en donde fuimos creando distintas ramas y luego las fuimos mergeando en la rama principal y cuando los programas ya estaban más cerca de finalizarse comenzamos a trabajar todos sobre la rama main.
Para facilitar los diseños de QT, utilizamos QTDesigner.

Conclusiones

Al momento de realizar el trabajo práctico nos dimos cuenta lo complejo que puede llegar a ser un proyecto y lo importante que es mantener las buenas prácticas de programación, como por ejemplo crear apis claras y concisas que fueran fáciles de utilizar por otros desarrolladores (nuestros compañeros de grupo) y que permitieran escalar en el trabajo.