Taller de Programación I - 75.42
Micromachines

Manual de Proyecto

Índice

1. Integrantes y división de tareas

- **Alejo Acevedo**: Ventana de Qt. Conexión inicial con el servidor. Bot en Lua. Bibliotecas dinámicas. Archivo de configuraciones
- **Facundo Torraca**: Servidor. Manejo de Partidas. Box2D.
- **Javier Ferreyra**: Cliente en SDL. Grabación de partidas con ffmpeg.

2. Evolución del Proyecto

- **Semana 1**: Sistema multicliente multipartidas por consola. Diseño tentativo del cliente. Cliente por consola. Pruebas de concepto LUA, SDL y bibliotecas dinámicas.
- **Semana 2**: Ventana de entrada en Qt. Implementacion del esqueleto del cliente en SDL. Fisicas en el servidor con multiples vehículos. Comunicación cliente-servidor. Control del vehículo por parte del cliente. Cliente dibuja vehículos y la cámara sigue al vehiculo propio. Pista como imagen fija.
- **Semana 3**: Envío del mapa del servidor al cliente. Velocidad del coche dependiente del terreno. Cliente dibuja solo objetos visibles. Redimensionado de pantalla con reescalado. Cámara inteligente. Bot puede controlar el auto.
- **Semana 4**: Colisiones con objetos estaticos. Spawnpoints de vehículos en la meta. Línea de meta detecta el paso de autos. Hud en el cliente con velocimetro, contador de vida y minimapa. Inicio y fin de carrera con contador inicial y podio. Bot puede recorrer la pista.
- **Semana 5**: Modificadores funcionando en el servidor y en el cliente. Plugins en el servidor integrados. Lista de puestos de los jugadores a medida que van terminando la carrera. Reinicio de carrera a voluntad del creador. Menúes en el cliente: pausa, reinicio de carrera, desconexión. Vida de los coches en el servidor, respawneo en pista. Cálculo en tiempo real de la posición en la carrera.
- **Semana 6**: Grabación de partida con ffmpeg. Implementación de sonido en el cliente. Arreglos de bugs. Objetos dinámicos que se pueden chocar y mover en cliente y servidor.
- **Semana 7**: Mejora de performance del cliente. Posibilidad de intercambiar entre pantalla completa y ventana en cualquier momento. Arreglos de bugs. Mejora de la interfaz Qt.

3. Inconvenientes encontrados

3.1. Movimiento de Cámara

Durante el juego la cámara sigue al auto de manera ïnteligente", es decir que no sigue siempre al auto centrándose en él, sino que considera la velocidad, y los cambios de la misma, por ejemplo para mostrar más la pista por delante y menos por detrás, y alejar la cámara cuando se tiene mucha velocidad para dar mayor visión.

Inicialmente el problema con esto fue que si considerábamos sólo la velocidad actual, los movimientos de cámara iban a ser muy bruscos, por ejemplo si el jugador aceleraba y frenaba mucho (por ejemplo en curvas), la cámara estaría alejándose y acercándose todo el tiempo, lo que sería incómodo a la vista.

Lo que se implementó para solucionar esto fue guardar las velocidades durante el último segundo y tomar un promedio. De esta manera un cambio repentino no afecta tanto al movimiento de la cámara, y se consigue un un movimiento mucho más suave. Lo mismo se hizo con la rotación, ya que la misma se utiliza para calcular dónde está el "frente", y considerar sólo la rotación actual tendría el mismo problema ya mencionado al virar demasiadas veces.

3.2. Performance

Un problema que surgió durante la realización del juego fue la performance del cliente. Al ir agregando cada vez más elementos a dibujar la misma fue decayendo hasta que en un momento surgió la obligación de optimizar, ya que se registraba un consumo de al rededor del 90 % de CPU en una de las computadoras donde probamos el juego.

Para eso se utilizó una herramienta de Valgrind llamada **callgrind**, la misma genera un reporte de las llamadas a funciones que ocurren durante la ejecución del programa, luego con la herramienta **kcachegrind** se puede visualizar los resultados para conocer qué funciones están siendo llamadas más veces. Así logramos identificar las funciones que consumían más tiempo de CPU.

Al correr la herramienta obtuvimos lo siguiente:

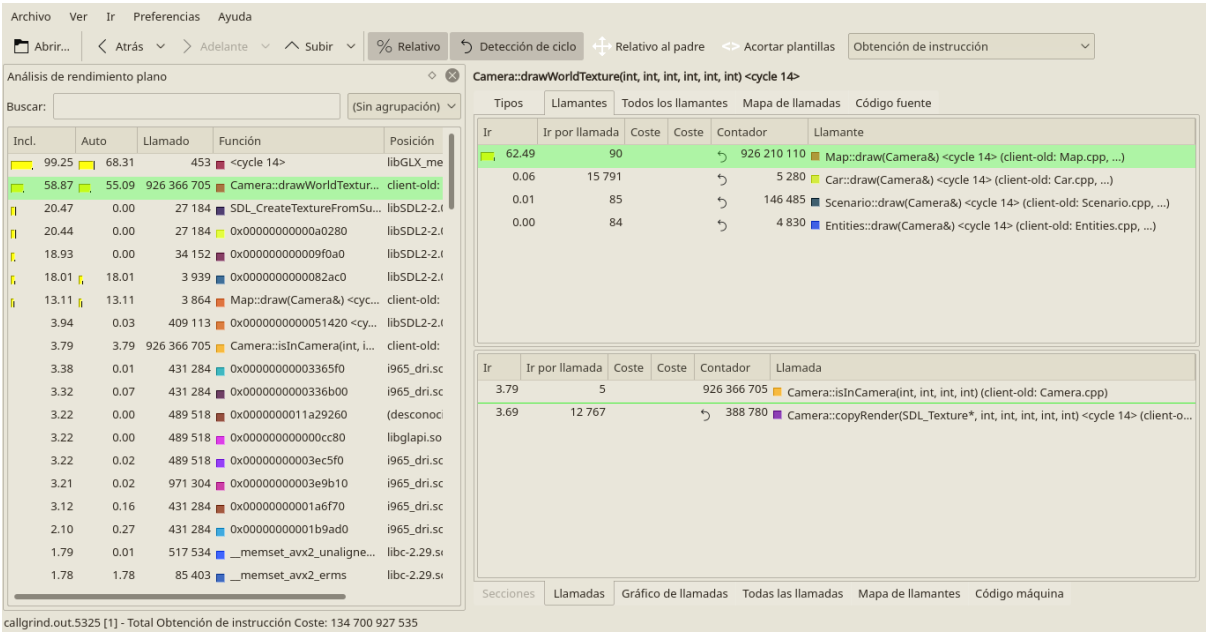

Figura 1: Vista del programa kcachegrind

Vemos que la función que está siendo llamada más veces es `Camera::drawWorldTexture()` por parte de `Map::draw()`, es decir, al hacer el dibujado del mapa.

Esto se debía a que al cargar el mapa, el servidor enviaba el tipo de textura que debía tener el fondo y el tamaño. El cliente se guardaba eso y en cada iteración del dibujado, hacía un doble *for* para dibujar la cuadrícula. Esto traía problemas de performance ya que un mapa en promedio del juego tendría una cuadrícula de al rededor de 150 x 150. Esto se traducía en que en cada iteración se llamaba al método mencionado unas 22.500 veces, y considerando que en cada llamado hace una copia de una textura al render de la pantalla, esto causaba el alto consumo de CPU.

El primer intento de solucionar esto fue que en el momento en el que el cliente reciba la información del fondo dibuje una textura enorme que abarque todo el mapa y se la guarde. Luego podría utilizar esa textura durante todo el juego haciendo un único llamado a la función por iteración, en lugar de dibujar cada cuadrado por separado.

El problema de eso fue que la textura que había que crear era extremadamente grande, y SDL2 fallaba al dibujarla. Y además el consumo de memoria RAM se incrementaba en al rededor de 1.5GB, demasiado en comparación de los 151MB que consumía antes.

La solución final al problema fue crear una textura de 50x50 cuadrados, la cual nos dio buenos resultados siendo lo suficientemente grande para que la cantidad de veces que haya que dibujarla para cubrir el mapa sea baja, y a su vez lo suficientemente pequeña para no disparar el consumo de RAM a niveles exorbitantes.

Con esta solución se logró reducir el consumo de CPU de 90 % a un 50 % en la computadora de prueba.

3.3. Puestos en tiempo real

Consideramos que un aspecto fundamental que debe tener un juego de carreras es mostrar el puesto de los jugadores en la carrera en tiempo real. Inicialmente creíamos que esto iba a ser fácil, ya que existen técnicas que se pueden utilizar para obtener esta información.

Una de ellas era contar la distancia que recorre un jugador desde que cruza la linea de meta, y ordenar los puestos en base a ese número. Ésta es una de las maneras más sencillas, pero el inconveniente que tiene es que los jugadores podrían sencillamente comenzar a dar vueltas en círculos en el lugar para sumar a su

contador de distancia. Esto no les serviría para ganar la carrera ya que eso se contabiliza con la cantidad de vueltas, pero haría que el puesto que se muestre sea erróneo, entonces descartamos esta posibilidad.

Otra técnica era contar la cantidad de cuadrados que pisó el jugador desde que cruzó la línea de meta. De esta forma se podrían marcar los cuadrados que el jugador ya pisó y no volver a contarlos. Esto podría haber funcionado de no ser que nuestro diseño de la pista permite que la misma tenga más de un cuadrado de ancho, por lo que el jugador podría pisar los que tenga a los lados sin avanzar realmente en el circuito y al igual que la técnica anterior el número que se mostraría sería erróneo.

Debido a estos inconvenientes debimos utilizar una técnica algo más sofisticada. Ya que nuestra pista estaba diseñada en forma de cuadrícula, la técnica utilizada consiste en generar un grafo donde cada nodo es uno de los cuadrados que conforman el circuito, y donde cada uno de ellos está conectado por una arista de peso 1 si son adyacentes en la cuadrícula. Luego, utilizando el algoritmo de Dijkstra se calcula la distancia a partir de la meta a cada uno de los nodos (nota: los cuadrados de la meta no se consideran conectados a los que se encuentran justo detrás, ya que se espera que los últimos cuadrados del circuito estén a distancia máxima). Luego, se ordena los puestos de los jugadores en base a la distancia a la meta del cuadrado sobre el que se encuentran, en tiempo real. Si el jugador se encuentra fuera de la pista (por ejemplo en pasto) su posición va a depender de el ultimo cuadrado de la pista que pisó, por lo tanto va a conservar su puesto en la carrera hasta que otro jugador pase por ese lugar.

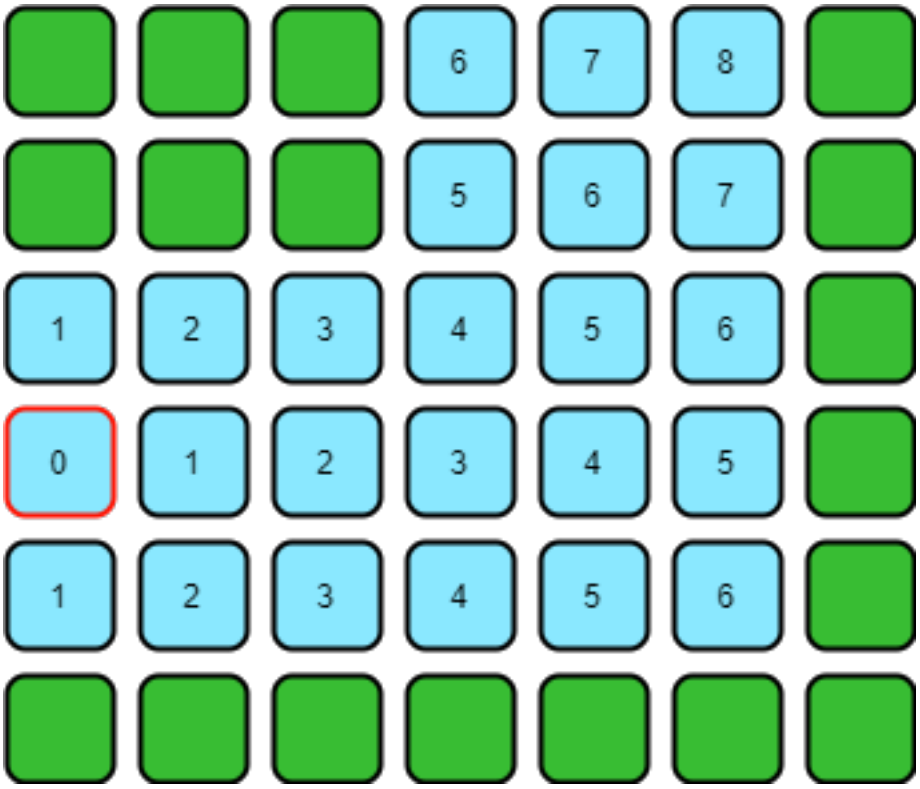

Figura 2: Ejemplo de como quedan marcados los cuadrados utilizando este algoritmo

El cuadrado marcado en rojo es el cuadrado inicial, los cuadrados marcados en celeste son los de la pista, y los cuadrados verdes son los externos (ejemplo: pasto). Los números indican la distancia del cuadrado al cuadrado inicial. Los puestos de los jugadores en la carrera se ordenan de acuerdo al número del cuadrado que están pisando en cada instante, o del último que pisaron que sea parte de la pista.

4. Análisis de puntos pendientes

4.1. Creación de mapas

Actualmente para crear los mapas se utiliza un programa llamado Tiled, y en el manual de usuario se detallan las instrucciones correspondientes a como crear con ese programa un mapa compatible con nuestro

juego. Consideramos que los pasos a seguir para crear un mapa 100 % compatible pueden llegar a ser algo confusos, por lo que un objetivo a futuro sería el de simplificar esta tarea.

5. Herramientas

5.1. QtCreator

Es la herramienta oficial que provee Qt para crear las interfaces de usuario. Nosotros la utilizamos para crear el menú principal del juego donde se debe introducir nombre de usuario, nombre de partida, entre otras cosas.

5.2. Tiled

Este programa permite crear de manera sencilla un mapa en forma de cuadrícula. Permite exportar los mapas en formato JSON, por lo tanto utilizamos esa funcion para luego desde nuestro juego parsearlo y generar la pista de carreras.

5.3. CLion

Todos los integrantes del grupo decidimos utilizar este IDE ya que contiene muchas otras herramientas que existen por separado, pero aquí se encuentran totalmente integradas, entre ellas:

- Análisis estático de código
- Integración completa con CMake
- Debugger con interfaz gráfica y totalmente integrado al código
- Generacion automática de headers y declaraciones de funciones o métodos

5.4. Git

Durante el desarrollo utilizamos un repositorio Git alojado en GitHub, el cual está planeado hacerse público una vez esté terminado el proyecto.

6. Conclusiones

Consideramos que la realización de este trabajo nos aportó muchos conocimientos en cuanto a desarrollar proyectos a mucha mayor escala que los que se suelen realizar en materias anteriores de la carrera. Una cosa que aprendimos es que es de mucha importancia sentarse a pensar bien los modelos que vamos a proponer para modelar cada sección del programa. Muchas veces es fácil escribir código sin pensar, ya que podemos creer que si nos tomamos más tiempo no vamos a llegar, pero eso a la larga puede traer problemas, ya que más tarde podemos cometer el error de empezar a poner parches en el código para arreglar un diseño que no fue pensado con tranquilidad. Muchas veces es parte de esto pensar un diseño que siga las buenas practicas de programación. Este fue uno de los pilares a la hora de encarar este proyecto y creemos que nos dio muy buenos resultados, ya que si bien no llegamos a presentar un trabajo completo el día de la primera entrega, logramos tener la mayor parte funcionando para que en las dos semanas restantes consistan en programar partes pequeñas que no presentaron mayores dificultades.