Evidencia de avance en el proyecto final de informática II.

Al comenzar creando una buena estructura en el diagrama de clases correspondiente al desarrollo del proyecto, creí que no habría mucha dificultad para implementar estas ideas estructuradas en la codificación, pero el tema de las interfaces gráficas está siendo un tipo de bloqueo para implementarlas, dado que, aunque es el mismo lenguaje y se usan los conceptos que se aprendieron este tiempo, es un “acento” diferente del lenguaje y cuesta implementarlo, pero no es ningún motivo de rendición, nada lo es.

Comienzo por el principio, creando una ventana donde hayan dos opciones: (1) Registrarse y (2) Iniciar sesión. Para lograr esto he creado una clase ventana que hereda de QmainWindow y representa lo que recién mencioné. Uso un Qlabel apuntando a un fondo para mostrar una imagen de inicio cargada de un archivo .qrc, esta imagen se adapta al tamaño de la ventana gracias a una función propia de la librería Qpixmap, y se va redimensionando automáticamente con otro método de la librería mencionada. En esta ventana se crean dos instancias de QPushButton, las dos instancias corresponden al botón de iniciar sesión y el de registro. Ambos están colocados en posiciones específicas con tamaños específicos, y estos dos botones son conectados a sus slots correspondientes, los cuales van a abrir interfaces de dialogo para inicio de sesión y registro, Esto último mencionado es importante porque tengo dos clases distintas, una que hace referencia al dialogo de registro y otra al dialogo de inicio de sesión, ambas interactúan con un archivo de texto que almacena el nombre del usuario y su contraseña correspondiente, la contraseña es almacenada de manera simple (no hay ningún método de codificación binaria). Dentro de la misma ventana, hay un pequeño audio de fondo, para esto debo usar QMediaPlayer configurado con un archivo de audio localizado en el .qrc, esta reproducción se realiza en una sola vez, una opción definida en setLoops () para que la música se mantenga activa durante toda la sesión en la ventana, y para este audio se crea un Objeto QAudioOutput que me va a permitir controlar aspectos de reproducción y salida de audio de la aplicación.

Luego de iniciar sesión correctamente llegamos a otra ventana, que corresponde a la clase base del nivel 1 y nivel 2, la cual la llamé NivelWindow. Esta clase que también hereda de QmainWindow, representa la interfaz ya mencionada, que va a dar a elegir al usuario para seleccionar el nivel 1 o el nivel 2, aquí también hay un fondo de una imagen de los Simpson con las mismas configuraciones de fondo que ya mencioné en la primer ventana, y también, como en la primer ventana, he instanciado dos botones como objetos de QPushButtom, cada botón está vinculado a un slot mediante connect (), además en este punto la música se ha detenido.

Luego de oprimir el botón para jugar el nivel 1 llego finalmente al primer nivel, donde al fondo se encuentra una imagen la calle de Springfield al frente de la casa de la familia Simpson, Bart, y la parca como personajes, donde hasta ahora Bart se puede mover y saltar libremente, pero debe evitar que la parca lance sus hachas a su dirección, debe hacer esto mientras intenta recolectar las gemas para destruir al villano.

Para lograr esto, construyo una clase base llamada personaje la cual hereda de objetos gráficos de Qt para ofrecer el movimiento, graficas de los personajes etc. En esta clase defino métodos públicos pertinentes a ser parte de un personaje, defino objetos de tipo QTimer, porque si Bart salta o es golpeado, las funciones que aplican a Bart un empuje son conectadas mediante slots a un par de timer´s que se encargan de dar un efecto de atracción gravitacional, así mismo una caída parabólica y ubicar finalmente a Bart en el suelo.

En la clase de Bart defino su sprite, la escala de su sprite, un método que lo mueve horizontalmente según sea el evento de teclado, y también la función de ser golpeado la cual hace un llamado a las funciones de la clase padre donde se aplica un empuje, se cambia la velocidad, se actualizan los retrocesos de caída, etc.

En la clase de la parca también defino su sprite y escala, tengo una función importante que retorna punteros a objetos de tipo Hacha, con ecuaciones de magnitud y normalizando el “vector” que apunta a la dirección de Bart, en esta función creo objetos de tipo hacha y los agrega a la escena solo si no ha terminado, debo hacer énfasis en que todas se guardan en una lista.

En la clase del Hacha también tengo unos atributos privados que muestran las velocidades de las componentes del Hacha, un booleano que me dice si el hacha está eliminado o no, un timer de movimiento claramente, un método que hace que el hacha se mueva conectado al timer de movimiento, y un getter para el booleano antes mencionado. Cuando la implemento defino el sprite y su escala, un método que conecta las salidas del timer con el movimiento del hacha, y en la función de mover trato de hacer que el hacha se elimine cuando se sale de las dimensiones de la pantalla si Bart la esquiva, aunque me esta crasheando la memoria porque parece que estoy eliminando objetos tipo hacha que no existen o que no se han creado y está siendo un problema bastante grande por lo que pienso añadir unos rectángulos que estén en los bordes de la imagen de fondo y que se verifique cuando el hacha los colisione, porque el problema no lo he podido resolver en un muy buen rato.

Aquí se muestra cómo se implementa el método que lanza hachas y es lanzada a la posición de Bart, este método pertenece a la clase Parca:

*float magnitud = std::sqrt(dx \* dx + dy \* dy);*

*float velocidadX = (dx / magnitud) \* 30;*

*float velocidadY = (dy / magnitud) \* 30;*

Donde se extrae un vector unitario con las velocidades (coordenadas) necesarias en X e Y para alcanzar las coordenadas del objetivo las cuales son indispensables para hallar la diferencia entre la posición de la parca y del objetivo que sería Bart, debo resaltar que se multiplica por 30 que sería un tipo de fuerza inicial con que es lanzada el hacha.

También tengo una clase sencilla llamada objeto, la cual solo implementa un constructor que settea la posición del objeto, de esta clase padre, hereda la clase gema, la cual es uno de los objetivos de Bart, esta gema desaparece cuando Bart la colisiona, y cuando las recolecta todas, se gana el juego, según esto, se puede ver que uso una lista de gemas para almacenarlas.

Finalmente implementando la clase de Nivel1, en la que comienzo añadiendo a la escena los ítems de los objetos creados con sus posiciones respectivas, añadiendo los timers para verificar las colisiones y para lanzar hachas cada cierto tiempo. También el método que inicializa la escena, con el suelo como rectángulo y con el fondo como imagen, aquí también está el método que maneja los eventos del teclado, cambiando las direcciones del Bart según la tecla, y el salto, y por ultimo verifico las colisiones, esta función verifica perfectamente si Bart colisiona con las gemas o con la parca, lo cual funciona excelente, el ciclo que hace que Bart retroceda si choca con el hacha, pero cuando el hacha sale de la pantalla estoy accediendo a una extraña posición de memoria a eliminar y el programa queda crashed, el error insiste de cualquier forma y es que quizá sigo sin verificar cuándo sale de la pantalla. Y por último, si se recolectan todas las gemas, sale un mensaje por pantalla de que se ganó.

Falta mucho por hacer realmente, pero en cada renglón redactado he tenido docenas de dificultades y dudas, espero llegar a un punto que el juego satisfaga, aunque se ha aplicado la mayoría del conocimiento teórico otorgado, y es lo que satisface. Así finalizo mi avance parcial para el 13/11/2024, con mucho más por implementar pero con el buen objeto de hacerlo.

Siguiendo con la implementación, la solución aplicada para poner rectángulos en los bordes de la escena ha permitido que el programa corra correctamente, las hachas se eliminan automáticamente una vez chocan con estos y el error dejo de persistir.

*“for (int j = 0; j < paredes.size(); { if (hacha->collidesWithItem(paredes[j])) {*

*scene->removeItem(hacha); delete hacha; parca->hachasLanzadas.removeAt(i); --i;*

*break; “*

De esa forma se verifica si algún objeto tipo hacha ha colisionado con algún elemento de la lista de las paredes, haciendo la verificación con un método perteneciente a QGraphicsItem, y así, el error dejó de persistir.

Es necesario implementar otro tipo de física, y la implementé en una clase llamada fuego, en esta clase fuego, se está trabajando con un objeto que representa una entidad, que será un tipo de proyectil lanzado por Bart en el escenario gráfico. La clase hereda de dos clases: QObject y QGraphicsPixmapItem la primera, QObject, como ya lo sé, es la clase base para todos los objetos que requieren características como señales y ranuras en Qt, mientras que QGraphicsPixmapItem permite que el objeto sea representado en una escena gráfica, mostrando imágenes, explico esto dado que es usado con frecuencia en el desarrollo y no había dado una explicación certera.

Esta clase fuego solo tiene un método, además del constructor, y es el que hace que ese “proyectil” se mueva, pero a diferencia de las hachas de la parca, se moverá en forma de onda, el movimiento en las coordenadas X es simple, solo se suma la velocidad horizontal a la posición en x. Pero en el eje Y, el movimiento es un poco más elaborado, ya que incorporo una pequeña función seno: (std::sin(x()/30)) Dando un movimiento oscilante emitiendo la apariencia de que el fuego se mueve hacia arriba y hacia abajo, mientras se desplaza horizontalmente hacia la posición de la parca, debe quedar claro que este movimiento es modulado por la velocidad en Y que es un parámetro y un factor de aceleración adicional que es importante para un movimiento con más forma de onda a efectos de nuestra vista, en este caso si multiplico por un 20 a la función seno implementada, el valor máximo y mínimo no será [-1,1] sino [-20,20] lo cual se verá mejor por pantalla. “velocidadY + 20 \* std::sin(x()/30)”

Es importante saber cuándo va a finalizar el juego. Para ello se trabajó bastante en una buena forma de saber 4 cosas: cuál es el estado de la vida de Bart, cuál es el estado de vida de la parca, cuántas gemas se deben recoger, y cuántas gemas se han recogido. Estos son los cuatro eventos que si se cumplen hacen que el nivel termine y se gane o se pierda, es claro insinuar cuándo se gana o pierde.

Debo explicar entonces como sé qué vida le queda a Bart o a la parca, para esto uso implementaciones similares dado que siguen siendo personajes.

Para esto lo que hago es crear un QGraphicsRectItem (un rectángulo). Cada clase, la de Bart y la Parca van a tener un método que se encargara de reducir la vida del personaje y consecuentemente reducir el tamaño del rectángulo que es el que la representa, este método toma como parámetro la cantidad de vida a reducir después de la colisión, tendrá un porcentaje de vida que simplemente será la vida divida entre cien: *“float porcentajeVida = vida / 100.0”*

Entonces ¿para qué me va a servir ese porcentaje? Pues, cada que vaya disminuyendo ese porcentaje, va a disminuir el largo de la barra de vida:

*“barraVida->setRect(0, -10, 100 \* porcentajeVida, 5)”*

De esto se puede apreciar, que esa barra quedará fijada diez pixeles por encima del personaje y se moverá a donde éste se mueva, además cada que vaya disminuyendo el porcentaje va a ir cambiando de color para una mejor experiencia gráfica y así saber que tanta vida queda solo intuyendo con el color:

*“if (porcentajeVida > 0.7) {barraVida->setBrush(Qt::green); } else if (porcentajeVida > 0.3 barraVida->setBrush(Qt::yellow);} else {barraVida->setBrush(Qt::red);}”*

Y una vez la vida sea cero, doy finalización al nivel mostrando si se perdió o se ganó.



Ahora procedo a explicar ¿cómo sé cuántas gemas debo recoger? y además de eso ¿cuántas gemas he recogido? Para esto tuve que agregar 3 nuevos atributos privados a la clase nivel1:

*QGraphicsTextItem\* puntajeTexto; int puntaje;*

*QVector<QGraphicsPixmapItem\*> gemasRecolectadasImagenes;*

Como se puede observar se mostrará un texto, este es el que me va a decir cuántas gemas he recogido y cuántas debo recoger, además de eso defino un vector en el que los elementos apuntan a objetos gráficos, puede ser una lista dado que no necesito acceder a la posición de memoria de cada elemento rápidamente, solo los iré agregando y al final elimino todos, pero uso el vector para no usar solo listas.

Si Bart choca con una gema, se modifica el texto con el puntaje actualizado, además se agrega un objeto grafico a la escena y también al vector que va a contener estos punteros:

*“ puntaje += 1 puntajeTexto->setPlainText("Gemas recolectadas: " + QString::number(puntaje) + " de 10") ”*



En este punto no he hablado de un par de métodos importantes para el videojuego, uno que pertenece a la clase parca, el cual hace que ésta se mueva de manera aleatoria dentro de los límites de la escena, y la otra pertenece a la clase Bart que es la que permite que Bart no atraviese los puentes y se vea más realista, hay otros más como el cambio de sprites cuando se golpean, o pequeños audios cuando sucede algo pero omitiré estos y explicaré los que mencioné al principio.

Para que la parca se mueva aleatoriamente quiero que se mueva en una sola dirección por un tiempo, y que esa dirección sea aleatoria y este dentro de los limites de la pantalla, entonces para esto doy uso de ranuras y señales de esta forma: *connect(movimientoTimer, &QTimer::timeout, this, &Parca::moverAleatoriamente);*

Donde la función “moverAleatoriamente” es el slot o la acción que se usa cada que el Timer me da una salida, cada que llamo esta función calculo de manera aleatoria nuevas posiciones en X e Y:

*if (contador % 10 == 0)*

*{ direccionX = (std::rand() % 41) - 20; direccionY = (std::rand() % 41) – 20;} “*

No está de más aclarar que esto se hace solo cuando la función se ha llamado unas 10 veces, y dado que el Timer me da una salida cada 100 milisegundos, entonces se concluye que la parca cambia de dirección cada segundo. Esto lo logro también usando variables estáticas, que evitan que los valores del contador se reinicien cada que la función se llama.

Ahora, la función que evita que Bart atraviese los puentes usa un casteo dinámico para asegurarse que los objetos con los que está colisionando Bart sean de tipo QGraphicsRectItem, si lo son, entonces se pasa a extraer la caja de colisiones o delimitadora del ítem con el que colisiono (puente), así mismo la caja delimitadora o de colisiones de Bart para hacer las comparaciones y ajustes a las nuevas posiciones según el caso.

*QRectF puenteRect = puente->boundingRect().translated(puente->pos()) QRectF bartRect = boundingRect().translated(pos());*

Aquí, el método boundingRect() pertenece a la clase QGraphicsItem y devuelve el rectángulo delimitador (QRectF) el cual es el área mínima que rodea el objeto que apunta al método, pero dado que no tiene en cuenta su posición real entonces debo usar el método *translated()* perteneciente a QRectF dando como argumento la posición del objeto del contexto. Luego de tener estas delimitaciones aplicadas, procedo a aplicar la lógica de las posiciones:  
 *“ If {bartRect.bottom() <= puenteRect.top()} setPos(x(), puenteRect.top() - bartRect.height()) “*

Y así sucesivamente con todos los casos de colisiones que puedan haber entre Bart y algún puente, lo cual requirió de una amplia codificación.

**

Como mencioné, tengo más métodos de decoración como algunas aplicaciones del sonido, sprites etc. pero dado que ya di a conocer cómo hice algunos de ellos, está de más decir cómo funcionan los demás.

Gracias a que me costó bastante codificar cada línea escrita, trabajos de laboratorio y demás, solo tuve la posibilidad de hacer un solo nivel, además de lo relacionado a la ventana de registro e inicio de sesión y la ventana donde aparece el nivel disponible, todo esto también fue un aprendizaje amplio y por ello se hizo algo lento el avance del proyecto, pero con esfuerzo. Digo posibilidad dado que fue bastante satisfactorio hacer lo que se hizo, aplicar los conceptos que se dictaron durante todo el año, y además dimensionar tanto los fallos hasta el punto de encontrar la solución que se piensa menos trivial, es el mayor desafío que se me ha presentado hasta ahora en la vida académica, y aunque no se hizo al pie de las instrucciones o de los requerimientos, complace decirse a uno mismo que no me presté a la rendición.

Le agradezco maestro Augusto.