UNIVERSIDAD DE GUADALAJARA

CENTRO UNIVERSITARIO DE CIENCIAS EXACTAS E INGENIERÍAS

DIVISIÓN DE ELECTRÓNICA Y COMPUTACIÓN

PROGRAMA 2 – SIMULAR EL PROCESAMIENTO POR LOTES CON MULTIPROGRAMACIÓN

SEMINARIO DE SOLUCIÓN DE PROBLEMAS DE SISTEMAS OPERATIVOS

SECCIÓN: D01

CÓDIGO: 215468696

PROFESOR: VIOLETA DEL ROCIO BECERRA VELAZQUEZ

ALUMNO: DOMÍNGUEZ DURAN OSCAR ALEJANDRO

CARRERA: INGENIERÍA EN COMPUTACIÓN

26 DE MARZO DE 2021

ACTIVIDAD DE APRENDIJAZE 4 – SIMULAR EL PROCESAMIENTO POR LOTES CON MULTIPROGRAMACIÓN

OBJETIVO

El objetivo de esta actividad es evolucionar el programa pasado para implementar nuevas funcionalidades tales como la posibilidad de detectar una interrupción (el proceso pasa de estar en ejecución a la "cola" del lote al que corresponde), un error (el proceso "falla" y pasa a estar en el apartado de terminados), una pausa (el proceso se detiene hasta que se indique) y solo cuando se indique se continuará con el programa en ejecución. Además, en este programa se depreciará la captura de los registros por el usuario y, por ende, se han de generar todos los lotes con sus respectivos procesos de manera aleatoria.

Al ser una evolución del programa anterior, se entiende que, a excepción de los puntos previamente mencionados, las demás funcionalidades y características del programa pasado se deben mantener en este, tales como la división de los procesos por lotes, el contador de lotes pendiente o el contador global, etc.

DESARROLLO

Lenguaje utilizado:

Al igual que en el programa pasada, decidí seguir utilizando el lenguaje de C++ con el *framework* de Qt debido a que los resultados obtenidos fueron bastante de mi agrado y considero que la presentación con interfaz permite que la simulación se vea mejor. Además, ya tenía toda la base con estas dos herramientas, no quería "desperdiciar" todo el código que tenía pese a los obstáculos que tuve en el desarrollo de esta actividad (los cuáles detallaré más adelante); ojo, hago mención que decidí conservarlo porque hubo un punto en el que me cuestioné regresar a hacer el programa en consola debido a una serie de inconvenientes relacionados a los requerimientos solicitados para esta actividad.

Funcionamiento del programa:

Bien, vamos describiendo todo el proceso que conllevó el desarrollo del programa y las estrategias y/o herramientas empleadas para este. Lo primero que me di a la tarea de realizar fue modificar la interfaz correspondiente a la pantalla de captura de datos, sin embargo, para no eliminar el *stackedwidget* empleado, decidí reutilizar dicha pantalla para mostrar el título del

programa y a la vez capturar la cantidad de procesos a ejecutar, debido a que solo pedía un dato al usuario, mantuve un tamaño acorde a la cantidad de elementos en pantalla, es decir, un tamaño relativamente pequeño. Anexo imagen correspondiente a la nueva primera pantalla del programa:

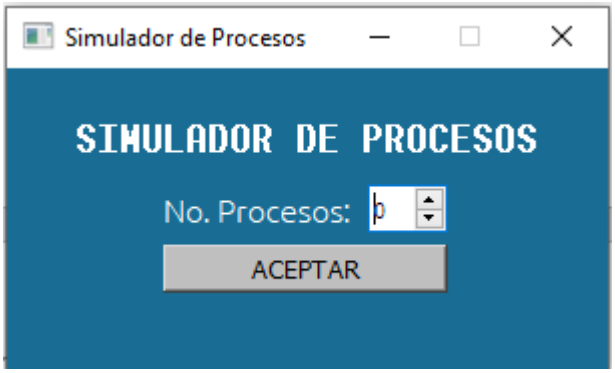

Ilustración 1 - Pantalla de captura de cantidad de procesos a ejecutar

En la pantalla anterior, se conservan algunas validaciones respecto a que no se puedan capturar 0 procesos y que el botón de aceptar solo pueda ser activado cuando sea diferente de 0, algo simple pero que merece la pena aclarar.

Una vez damos clic en el botón de agregar, se procederá a generar todos los procesos solicitados de manera aleatoria. Recordemos que las capacidades de cómputo de una computadora no nos aseguran una aleatoriedad 100% pura, y eso lo podemos ver reflejado en algunas funciones tales como *rand()* la cual, en efecto te otorga un número pseudoaleatorio, aunque delimitado por una serie de parámetros; lo que generaba una serie de números repetidos. Probé utilizando la hora del sistema mediante el método de *srand(time(NULL))*, la cual, en teoría, toma la hora del sistema como un parámetro a considerar y si, genera números que pueden no repetirse tanto como solo usar el *rand()* normal pero aún así al utilizarlo dentro de los ciclos en los que creaba los procesos, seguían repitiéndose algunos números de manera consecutiva.

Es por eso por lo que decidí recurrir a algún método que me diera la garantía de que las probabilidades de que los números se repitieran fueran las más bajas posibles. Rebuscando un poco dentro de la página oficial del lenguaje C++, me topé con un par de funciones que podrían servirme y en efecto, me ayudaron a generar los números que necesitaba. En resumidas cuentas, estas funciones generan una distribución en un rango de números, los cuales dependen de la hora actual del sistema en segundos y a parte están dados por un motor de generación de números aleatorios. Para que quede un poco más claro lo anterior, mostraré una parte del código y explicaré lo que se hace:

```
//Generating random numbers with the system time
std::default_random_engine generator(std::chrono::system_clock::now().time_since_epoch().count());
std::uniform_int_distribution<int> distribution_numbers(-100,100);
std::uniform_int_distribution<int> distribution_sign(0,5);
auto randomNumber = bind(distribution_sign,generator);
```

Ilustración 2 - Generación de números aleatorios

En la imagen anterior podemos observar como se hace uso de un generador de números a partir de la hora actual del sistema en segundos, después de esto podemos crear una distribución de los números que, por lo que entendí, es generar los rangos que deseemos. Una vez tenemos el generador y la distribución, asignaremos a una función denominada "*randomNumber*" el resultado de la combinación de estos dos parámetros, esta función retornará el número deseado cada que la usemos. Honestamente hubo algunos aspectos de esta manera de generarlos que no terminé de entender al 100% pero lo he puesto a prueba y tal y como asegura la página del lenguaje, los resultados si aparentan ser aleatorios, así que terminé por utilizar esta manera de generar los números.

Se respetaron las validaciones necesarias para los casos en los que se presente alguna división entre 0, una validación simple pero sencilla, ya que en caso de que se generara un 0 en el divisor, checamos si estamos haciendo una división y de ser así, volvemos a generar un número aleatorio para el divisor hasta que este deje de ser 0, lo cual está expresado en un ciclo que se detiene al cumplir la condición previamente mostrada. También generamos el tipo de operación a realizar con este método, al igual que los segundos correspondientes al tiempo estimado de cada proceso. Respecto al ID, los genero de manera consecutiva por lo que nunca se van a crear ID's duplicados y, por ende, no requiere mayor validación.

Los lotes se siguen generando cada uno con 5 procesos máximo, aquí si decidí moverle un poco ya que en mi programa anterior primero capturaba todos los procesos y luego pasaba a separarlos en un vector de vectores, en los que el vector principal era para los lotes y el sub vector de este representaba los procesos de cada lote. Consideré bastante ineficiente esto ya que estoy reasignando los valores muchas veces, quizá en el programa anterior no me pareció mala idea debido a que la interacción con el usuario requería algo así, pero en esta actividad, al no requerir de una captura individual de cada proceso, considere apropiado que conforme generaba cada proceso lo almacenaba en su correspondiente lote, en donde cada 5 procesos (o ellos restantes) cambiaban de lote. De esta manera los procesos quedan asignados a su lote correspondiente justo después de ser creados. En este punto pensé en cambiar de estructura de datos y utilizar una lista en lugar del vector como tal, pero algo que me gusta de esta TDA es que podemos manipularla para hacerla funcionar según lo requiramos como una cola o una pila, en pocas palabras, es más flexible en su funcionamiento que una lista o cola.

Respecto a mi clase de proceso, solo realicé los cambios pertinentes para poder cumplir con los requerimientos de la actividad, eliminé de sus atributos el nombre, pero agregué dos más que me serían de utilidad: el tiempo transcurrido (para las interrupciones) y el estado de

finalización de este (es decir, si terminó de manera exitosa o mediante un error). Estos dos elementos de los procesos son de utilidad para poder controlar el como se verán mostrados en sus correspondientes apartados en la interfaz.

Una vez tenemos los procesos listos, el programa cambia a la pantalla en la cual se mostrará la "animación" correspondiente la simulación. Esta ventana no recibió ningún cambio significativo, de hecho, se podría decir que se ha quedado intacta visualmente respecto al programa pasado, con la excepción de que en el apartado del lote en ejecución ahora mostramos el tiempo transcurrido de cada proceso y que en el apartado del proceso en ejecución se quitó el renglón correspondiente al nombre del programador. De todas maneras, procederé a mostrarla para que en futuras capturas se pueda apreciar su estado antes de comenzar a ejecutar los procesos:

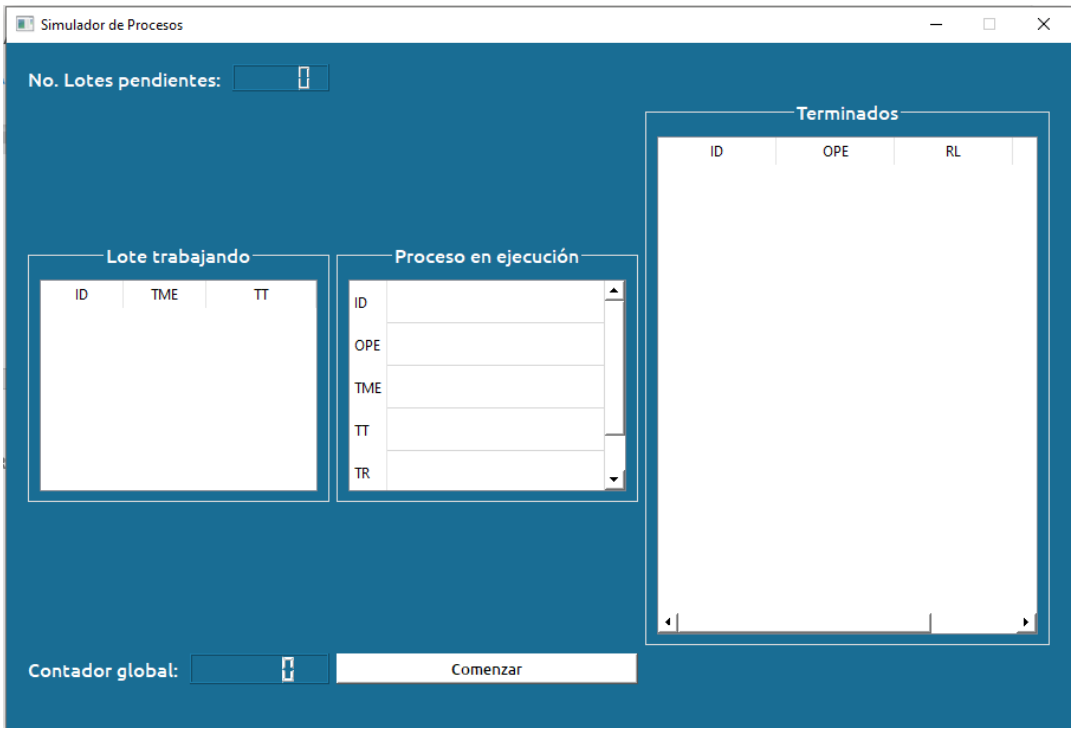

Ilustración 3 - Pantalla para mostrar la ejecución de los procesos

Hasta este momento, no se me había presentado algún problema que realmente representará un impedimento de llevar a cabo el programa. No fue hasta el momento de querer utilizar el *kbhit()* cuando la cosa se complicó. Como ya se nos había explicado en clase, el uso de esta función está ligado al manejo de un *buffer* en el cual se busca la tecla que deseemos detectar; lo que quiero decir es que, no funciona como una detección de un evento como tal y esto, en términos de desarrollo de GUI's supone un gran problema, un de que por desgracia me di cuenta muy tarde. Para no hacer el cuento muy largo, cuando yo ponía esta función dentro del bucle correspondiente, lo que hacía era congelar a lo que en Qt se le conoce como "EventLoop", es decir, la manera en la que yo estaba mostrando la información corresponde a

una serie de eventos que el programa procesa de manera secuencial, cuando yo trataba de poner el *kbhit()* lo que hacía era interrumpir este flujo y por ende el programa se volvía un tanto lento hasta llegar al punto en el que no podía retomar el flujo principal y se este *crasheaba*. En consolas funciona sin problema alguno debido a que en las interfaces gráficas los eventos son acciones especificas que suceden a la par de la ejecución del "MainLoop" o el flujo principal del programa. A decir verdad, si es algo que no termino de entender al 100%, pero leyendo la documentación de Qt, pude comprender que todo lo que desee implementar respecto a los eventos que maneje deben de estar "sincronizados" con el flujo principal.

Me tomó bastante tiempo comprender el porqué no podía utilizar una función cómo *kbhit()* en una GUI, por lo que cuando empecé buscar otras opciones me topé con que Qt posee su propia librería para detectar eventos de tecleo, y que por fortuna, podemos añadirle este evento a cada *Widget* de Qt, este evento sería colocado en la ventana principal. De esta manera, siempre estará atento a lo que tecleemos siempre y cuando el foco del programa no se encuentre en algún *Widget* que a su vez requiera de la intervención del teclado como lo puede ser una caja de texto.

```
protected:
    void keyPressEvent(QKeyEvent *event);
```

Ilustración 4 - Sobrecarga de la función para detectar un tecleo

Una ventaja que tiene esta función es que, al ser un "listenner", siempre está en espera de que se realice la acción solicitada, en este caso, el tecleo del usuario. Cuando esto sucede, el flujo principal del programa no se detiene de manera abrupta, sino que pasa directamente a ejecutar lo que declaremos en dicha función y al hacerlo vuelve a donde se quedó dentro del flujo principal, hasta dónde sé no se maneja ningún hilo con esta opción ni nada parecido, si no que el mismo Qt busca la manera de manejar el evento o los eventos, sin interrumpir o alterar el famoso "Main Loop".

Es por eso que pese a que *kbhit()* pertenece a una librería del sistema, no me fue posible utilizarla sin alterar el funcionamiento del programa, así que decidí atenerme a la documentación del *framework* y utilizar la librería diseñada específicamente para este. De esta manera, respeto todas las cuestiones relacionadas al flujo principal del programa; el manejo de eventos me resultó más sencillo de comprender y de implementar para la detección de las teclas. Traté de relacionar el como funcionaba todo esto con el uso de botones en la interfaz, es cierto que, al presionarlos, estos no detienen lo que el programa esté realizando en ese momento, si no que lanzan una señal que es percibida para realzar una función, manejada como SLOT. De hecho, ahora que lo pongo así, los eventos tienen una manera de comportarse similar a los SIGNALS/SLOTS, pero con diferencias delimitadas por los factores externos a la interfaz como lo puede ser el uso del teclado.

Una vez aclarado todos los problemas que me generó el querer detectar las teclas pulsadas por el usuario, puedo describir lo que se realiza cada que se presiona una tecla durante la ejecución del programa. Como ya se observó, esta función recibe como parámetro un puntero a un *QKeyEvent*, mediante este evento, podremos checar la tecla que fue pulsada mediante su correspondiente método. Por fortuna, Qt tiene sus propias maneras de verificar los valores de las teclas, por lo que es indiferente a si pulsamos una letra mayúscula o minúscula.

```
void MainWindow::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_I and state == RUNNING){
        qDebug() << "Interrupcion";
        interrupted = true;
        tT = ACTION_CODE;
        interruptProcesss();
    }
    else if (event->key() == Qt::Key_E and state == RUNNING){
        qDebug() << "Error";
        tT = ACTION_CODE;
        showFinishedProcesses(ERROR_FINISH);
    }
    else if (event->key() == Qt::Key_P and state == RUNNING){
        qDebug() << "Pause";
        state = PAUSED;
        pause.exec();
    }
    else if (event->key() == Qt::Key_C and state == PAUSED){
        qDebug() << "Continue";
        state = RUNNING;
        pause.quit();
    }
    else{
        qDebug() << "Nothing";
    }
}
```

Ilustración 5 - Detección de la tecla pulsada

En el caso de que se pulse una tecla, muestro (mediante un mensaje de *debugeo* propio del *framework*) lo que se debe hacer en la consola para posteriormente realizar una de las acciones especificadas al inicio de la actividad (interrupción de la ejecución del proceso, terminar el proceso en ejecución mediante un error, la pausa del proceso que se esté ejecuta y por supuesto la manera de continuar con la ejecución de un proceso pausado). Para el desarrollo de esta característica, decidí empezar con lo que, en principio, consideré que sería lo más complicado de llevar a cabo, es decir, la interrupción de un proceso para continuar con otro y postergar la ejecución de este. La lógica por utilizar fue bastante sencilla: si se detecta que la tecla pulsada fue una “I”, procedo a activar una bandera la cual me servirá para saber si se trata de en efecto una interrupción o, por el contrario, el proceso se detuvo por error. Además de

activar la bandera correspondiente modificó el valor del contador de tiempo transcurrido global, de esta manera me aseguro que el ciclo en dónde verifico que el tiempo transcurrido sea menor al tiempo estimado se detenga y proceda a realizar la función denominada *interruptProcess()*, la cual genera una copia temporal del registro en ejecución, elimina el proceso que se está ejecutando y procede a insertar al final del vector la copia que hicimos del proceso. Para ilustrar de mejor manera lo que acabo de mencionar, mostraré la parte del código descrita:

```
void MainWindow::interruptProcesss()
{
    Proceso p = lots[0][0];;
    lots[0].push_back(p);
}
```

Ilustración 6 - Interrupción del proceso

Como ya se mencionó el cambió del valor del contador de tiempo transcurrido es importante para salir del ciclo de ejecución del proceso, por lo que como ya se cambió el valor, cuando el flujo del programa regrese a ese ciclo, notará que el valor de dicho contador ha cambiado, saldrá del ciclo y eliminará el proceso, pero como ya se hizo un respaldo insertando al final el proceso, se mantendrá en la cola de espera del lote de ejecución.

```
while (tT < p.getTiempoEstimado() and tT != ACTION_CODE){
    tR--;
    tT++;
    p.setTiempoTranscurrido(tT);
    ui->processRuningTB->setItem(3,0,new QTableWidgetItem(QString::number(p.getTiempoTranscurrido())));
    ui->processRuningTB->setItem(4,0,new QTableWidgetItem(QString::number(tR)));
    globalCounter++;
    ui->globalCountLCD->display(globalCounter);
    ui->pendientLotsLCD->display(pendientLots);
    delay();
}
if (p.getFinalizacion() != ERROR_FINISH and !interrupted){
    showFinishedProcesses(SUCCESFUL_FINISH);
}
lots[0].erase(lots[0].begin());
ui->processRuningTB->clearContents();
```

Ilustración 7 - Ciclo de ejecución del proceso

Bueno, en el bloque de código mostrado previamente podemos apreciar lo mencionado, la macro *ACTION_CODE* posee un valor de -1, dicho valor, me permite salir del ciclo cuando ocurra un error o una interrupción. Posterior a la salida del ciclo, verificamos si la bandera de finalización del proceso no es un error y que tampoco la bandera de interrupción esté encendida para mostrar el proceso terminado, de esta manera, en caso de saltar ese *if* disparador se trataría de una interrupción y, por ende, procedemos a eliminar el proceso actual porqué ya lo habremos insertado al final del lote.

En el caso de que la tecla presionada sea la referente a que el proceso debe terminar por un error, también cambiamos el valor del contador de tiempo transcurrido al valor de *ACTION_CODE*, posteriormente mostramos el proceso terminado mediante la función de

showFinishedProcess(bool typeFinish), a la cual le pasaremos el valor de 0/false (valor que guardamos en el proceso) ya que se trata de un error e internamente, esta función mostrará el proceso con los valores que poseía antes de presionar la tecla correspondiente en la tabla de los procesos terminados con la diferencia de que en ligar de mostrar el resultado de dicha operación, nos mostrará el mensaje de "ERROR". Anexo una imagen de la función descrita previamente:

```
void MainWindow::showFinishedProcesses(bool finishedType)
{
    Proceso& p = lots[0][0];
    int rowsFinished(ui->finishedTB->rowCount());
    p.setFinalizacion(finishedType);
    ui->finishedTB->insertRow(rowsFinished);
    ui->finishedTB->setItem(rowsFinished,0,new QTableWidgetItem(QString::number(p.getId())));
    ui->finishedTB->setItem(rowsFinished,1,new QTableWidgetItem(QString(p.getOperacion().c_str())));
    ui->finishedTB->setItem(rowsFinished,3,new QTableWidgetItem(QString::number(currentLot)));
    if (finishedType == ERROR_FINISH){
        ui->finishedTB->setItem(rowsFinished,2,new QTableWidgetItem(QString("ERROR")));
    }
    else{
        ui->finishedTB->setItem(rowsFinished,2,new QTableWidgetItem(QString::number(p.getResultadoOperacion())));
    }
}
```

Ilustración 8 - Función para mostrar los procesos finalizados

Después de haber realizado la parte de la interrupción y la parte del error, creí que el hacer la pausa sería mucho más sencillo, pero no, me llevé una desafortunada sorpresa al ver que no existía un método como tal dentro de Qt que hiciera una pausa, por lo menos no como en el caso de un programa de consola. Recordemos que gran parte de los problemas que he tenido han sido por el manejo de los eventos del "Main Loop", pues bien, para realizar una pausa dentro de este ciclo tuve que leer bastante más documentación de Qt y por lo que leí, no existe una manera 100% orgánica de generar una pausa por lo que tuve que buscar algunas alternativas a esto y simular una pausa; es decir, bloquear el flujo del programa con la ejecución de un *QEventLoop* sin bloquear los eventos que se estén procesando. Sé que suena confuso, pero una vez instancié un objeto de este tipo, al cual llamé *pause,* fue bastante fácil de manipular ya que solo tenía que ejecutarlo cuando se detectará la pausa y en efecto, se bloqueaba de alguna manera el flujo del programa y, además, cambiábamos el estado del programa a *PAUSED*. Para desbloquear el flujo, bastaría con validar que la tecla "C" es presionada solo cuando el estado del programa es pausado, de esta manera solo cambiamos el estado del programa a *RUNNING* y detenemos el *QEventLoop* denominado *pause*, de esta manera el "Main Loop" quita de su foco al ciclo que creamos y continua con el resto del código.

La solución terminó siendo más simple de lo que esperaba, pero como ya he mencionada en varias ocasiones a lo largo de este reporte, me tomó bastante tiempo encontrar dicha solución, leyendo documentación oficial o foros exclusivos de Qt. Definitivamente no esperaba que la pausa del programa fuera de las cosas que más me generó problemas en esta actividad. Una vez concluidas las funcionalidades de cada tecla lo demás fue solo coordinar algunas cosas para que la información mostrada en las tablas y contadores fuera la correcta. Para finalizar, me gustaría mostrar una captura de la ejecución del programa con algunas pruebas para la interrupción y el error en un proceso, al igual que la pausa de este:

Ilustración 9 - Ejecución del programa con 11 procesos

Bien, en la imagen podemos apreciar como existen algunos procesos que finalizaron en error, de igual manera se puede observar que el proceso con el ID 6 fue mandado al final del lote mediante una interrupción y finalmente, en la parte izquierda podemos apreciar la secuencia de teclas pulsadas hasta el momento en la ejecución del programa (mediante los *QDebugs* mostrados en el código fuente previamente.

CONCLUSIÓN

Esta actividad realmente supuso un reto quizá no tan grande como en el primer programa ya que, a diferencia de ese entonces, ya tengo las bases del programa y las modificaciones o evoluciones de algo establecido siempre son de alguna manera más sencillas que empezar a realizar algo nuevo desde 0. Por lo que la dificultad de este vino de la mano con mi desconocimiento sobre algunos aspectos importantes de las aplicaciones con interfaz de usuario, quizá si lo hubiera realizado desde un inicio en consola, esta actividad no me hubiera supuesto mayor problema. No considero que ahora sea un experto en interfaces gráficas ni mucho menos, con este programa me di cuenta de que muchos de los conocimientos que poseía sobre Qt y las interfaces eran bastante bajo y que aún me quedan muchas cosas por aprender por lo que, de alguna manera u otra y aunque no lo quisiera, terminé aprendiendo aún más cosas sobre la herramienta que estoy utilizando y también el lenguaje.

Respecto al programa funcionando, he de decir que estoy bastante satisfecho con el resultado obtenido y el como la animación ahora es más fluida, al igual que todas las detecciones de teclas funcionan bien, sin generar conflictos con el "Main Loop" del programa y sin hacerlo tronar. Quizá para esta actividad me falló un poco más la constancia ya que se me juntaron algunas tareas y avances de otros proyectos con fechas más cerradas y, por ende, terminé dedicando menos tiempo por día de lo que me hubiese gustado a este programa; por lo que sigo agradeciendo la flexibilidad de los tiempos de entrega de esta materia. Para terminar, me gustaría mencionar que aún pese a todos los problemas que se me presentaron, el poder haber sacado adelante este programa como lo tenía planeado representa una gran satisfacción para mi y me motiva a seguir utilizando Qt y C++ como mis herramientas para los futuros programas.

Link del vídeo del programa:

- https://youtu.be/qmXXT1q3frI