MODELADO DE CLASES EN QT PARA LA REALIZACIÓN DEL PROYECTO FINAL

MANUEL SANTIAGO VELÁSQUEZ LOPEZ

JOSÉ ALEJANDRO OLIVO PETIT

FACULTAD DE INGENIERÍA, UNIVERSIDAD DE ANTIOQUIA

2598521: INFORMÁTICA II

5 DE ABRIL DE 2022

INTRODUCCIÓN

Para el último proyecto del curso de Informática II, presentaremos una aproximación de los objetos que utilizaremos para su desarrollo, esto con el objetivo de presentar el camino a seguir para la realización de todo el código que es esencial para un trabajo eficiente y colaborativo. El juego el cual se va a crear debe constar de tres modelos físicos no relacionados sin tomar en cuenta el movimiento rectilíneo uniforme, un sistema de guardada y cargada de partidas, y documentación valida a través de repositorios para evaluar el correcto manejo de la programación orientada a objetos y como evaluación final del proceso transcurrido durante todo el curso.

Para esto se propone un juego en el cual se deba controlar un obús estático el cual dispara proyectiles a objetivos dados, este proyectil puede o no rebotar contra ciertos objetos para llegar a su destino, y debe evitar destruir obstáculos presentes a lo largo del mapa. El objetivo final puede estar en movimiento lo cual hace que se deba tener en cuenta su posición, además, en dificultades superiores, el viento puede afectar la trayectoria del proyectil. Cada vez que se dispara el juego calcula el costo del proyectil, la carga explosiva utilizada para su lanzamiento y la eficiencia del cañón utilizado, siendo posible realizar mejoras al obús que permiten una mejor eficiencia y proyectiles más pesados.

Todo esto con el objetivo de no solo presentar un desafío para el jugador sino también para incluir los modelos físicos necesarios para la realización del proyecto, estos se pueden apreciar en el movimiento parabólico del proyectil, el cálculo de su velocidad al inicio a través de ecuaciones de energía además del cálculo de su velocidad después de colisiones a través de choques, y el movimiento del objetivo u obstáculos a través de movimientos armónicos simples o movimientos circulares.

El costo variable del proyectil y la opción que el jugador tiene para utilizar sus fondos como crea que sea la forma más adecuada asegura que dos partidas nunca van a ser iguales, y por lo tanto genera la necesidad de guardar el progreso logrado en diferentes partidas.

Además, la relativa dificultad del juego hace necesaria la existencia del anteriormente mencionado programa, ya que el jugador puede decidir trabajar en el siguiente nivel después de un periodo de descanso y así preservar su salud física.

Teniendo esto en cuenta, se hace el planteamiento de las clases y métodos necesarios para la realización del juego de video en el IDE Qt, y una explicación del porqué de cada uno de los objetos y métodos a utilizar.

CLASES

Como fue anteriormente mencionado, es necesario el uso de objetos para la realización del juego de video, debido a su complejidad y además para una mayor facilidad de entendimiento y reutilización del código, siendo así más eficiente ver este problema a través del paradigma de la programación orientada a objetos. A continuación, se muestran las clases planteadas, divididas en clases gráficas y clases funcionales dependiendo de su utilización.

CLASES FUNCIONALES

Son clases funcionales aquellas que se proveen valores a las clases gráficas, son más robustas pero independientes de las clases graficas para facilitar su modificación. Requieren más argumentos que las clases gráficas y poseen los métodos necesarios para actualizar su posición a partir de variables dadas y eventos como colisiones o presión de botones.

howitzer(int px, int py, double joules, double shellMass, double tilt, bool spun, unsigned char efficiency)

Clase correspondiente al cañón, se le llama “howitzer” para distinguir entre el obús (pieza de artillería) y el obús (proyectil con explosivo) los cuales tienen palabras distintas en inglés. Se le ingresan todos sus atributos a través de argumentos y genera una clase shell a

la hora de llamar el método disparar, el cual calcula la velocidad en ambos ejes dada una carga explosiva, una masa del proyectil, una eficiencia (dada por mejoras) y un angulo de disparo. Las mejoras se podrán comprar en un menú extra que corresponderá a una tienda, y permiten mayor masa de proyectiles y mejor eficiencia a la hora de dispararlos, lo cual se transmitirá al cañón cada vez que se llame. Una vez puesto en escena llama a las clases graficas grafHowitzer Y grafTurret para su visualización.

shell(int px, int py, double vx, double vy, double wind, double mass)

Clase correspondiente al proyectil que dispara el cañón, se le llama “shell” para distinguir entre el obús (proyectil con explosivo) y el obús (pieza de artillería) los cuales tienen palabras distintas en inglés. Recibe su velocidad de la clase cañón y calcula automáticamente su inclinación para su grafica y su marco de colisión. En vez de tener un rectángulo como cuadro de colisiones tendrá una forma definida como una elipse, esto con el objetivo de tener más precisión en cada rebote. Constantemente busca colisiones y, dependiendo de con que colisiona, llamará a un método que lo detonará o le hará cambiar su dirección dependiendo de su inclinación y la inclinación del objeto con el cual colisiona. Esto se logra a través de fórmulas de colisiones físicas las cuales tendrán en cuenta su velocidad y ángulo de entrada.

Cuando el proyectil deba explotar va a llamar un método de detonación el cual generara un círculo de área correspondiente a la masa del proyectil, el cual revisara con que ha colisionado y cambiara su estado respectivamente. Además, estará constantemente actualizando su posición antes de explotar a través de un timer, lo cual actualizara su inclinación y velocidad a través de una constante de gravedad.

wall(px, py, double tilt = 0, bool bounce)

Clase correspondiente a las paredes presentes en cada nivel, estas pueden ser de acero y no permitir rebotes, o de goma y si permitirlos. Ambas tendrán un brush diferente pero su comportamiento no amerita crear dos clases diferentes. Con cada colisión de un projectil le dan sus valores de bounce, siendo True para las paredes de goma y False para las paredes

de acero, y si puede rebotar entonces le dará su valor de inclinación. Contienen un método para llamar a su respectiva clase grafica grafWall y su Rect de colisión estará modificado dependiendo de su inclinación.

target(int px, int py, double tilt, bool enemy, int movePattern, bool destroyed)

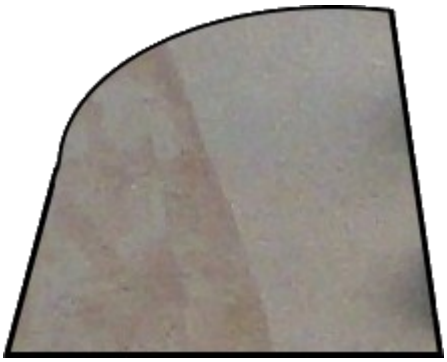
Clase correspondiente a los objetivos y obstáculos, tendrán un Sprite diferente dependiendo de cual sean. No poseen colisiones precisas pero los enemigos pueden moverse con patrones predefinidos dados por el atributo movePattern, los cuales ya estarán definidos y solamente deberán ser llamados. Esto se hace ya que esta es la única clase con movimientos de este tipo y para simplificar el código. Contienen además un atributo bool que me permite saber si ya han sido destruidos, si todos los objetivos son destruidos se pasara de nivel, pero si un obstáculo es destruido hará que se deba reiniciar el nivel.

CLASES GRAFICAS

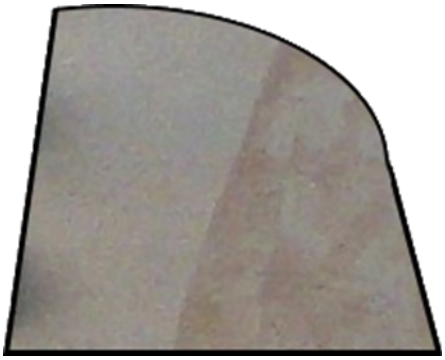
Se definen clases graficas a aquellas que permiten la visualización de los objetos funcionales, son más simples que sus contrapartes funcionales y dependen de ellas, pero son separadas para facilitar la modificación de las clases funcionales, especialmente a la hora de modelar colisiones precisas.

grafTurret(int px, int py, bool spun)

Clase grafica correspondiente a la torreta, contiene dos argumentos de variables enteras para su posición (int px & int py) y un argumento para determinar si la torreta mira a la derecha o a la izquierda. Estos son extraídos de la clase cañón de la cual depende, también posee un método que actualiza su posición cada vez que se coloque. Utiliza herencia de la clase QPixmap para graficar el Sprite .png de la torreta, el cual estará alojado en una carpeta de recursos para un más fácil acceso.



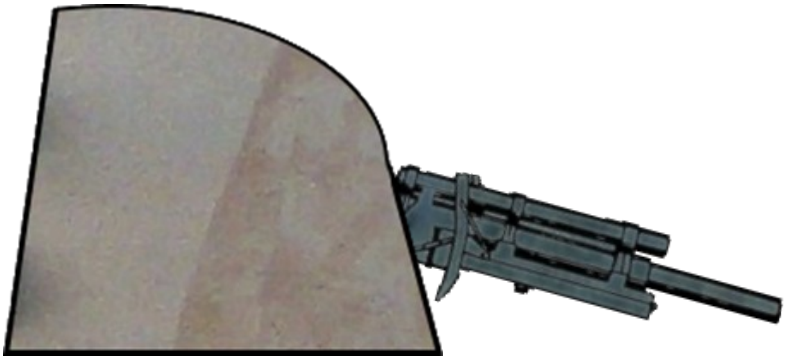
Torreta girada a la izquierda (bool spun = True)



Torreta girada a la izquierda (bool spun = False)

grafHowitzer(int px, int py, double tilt, bool spun)

Clase grafica correspondiente a el cañón, contiene dos argumentos de variables enteras para su posición (int px, int py), un argumento que define si está mirando a la derecha o izquierda (bool spun) y un argumento de variable tipo double para su inclinación (double tilt) con valor máximo 90 y mínimo -15. Estos son extraídos de la clase cañón a la cual depende, también posee un método que actualiza su posición cada vez que se apunte, sus coordenadas van a ser diferentes a las cuales se ingresen ya que no se encuentra en exactamente el mismo punto que la torreta. Utiliza herencia de la clase QPixmap para graficar el Sprite .png del cañón, y además utiliza transformaciones para girar el Sprite, el cual estará alojado en una carpeta de recursos para un más fácil acceso.



Torreta con el cañón mirando a la derecha (bool spun = False) y cañón apuntando a -15.0 grados (double tilt = -15.0)

grafShell(int px, int py, double tilt, bool spun)

Clase grafica correspondiente a el proyectil lanzado por el obús, contiene dos argumentos de variables enteras para su posición (int px, int py), una variable double para su inclinación con valores entre 90 y -90 (double tilt), y una variable que determina a que lugar está mirando el proyectil (bool spun). Su posición e inclinación son dados por la clase “shell” de la cual depende, y es actualizado por este a través de un método de movimiento. Utiliza herencia de la clase QPixmap para graficar el Sprite .png del proyectil, y además utiliza transformaciones para girar el Sprite, el cual estará alojado en una carpeta de recursos para un más fácil acceso.

Proyectil viajando en trayectoria parabólica, siendo actualizado cada t(x) tiempo.

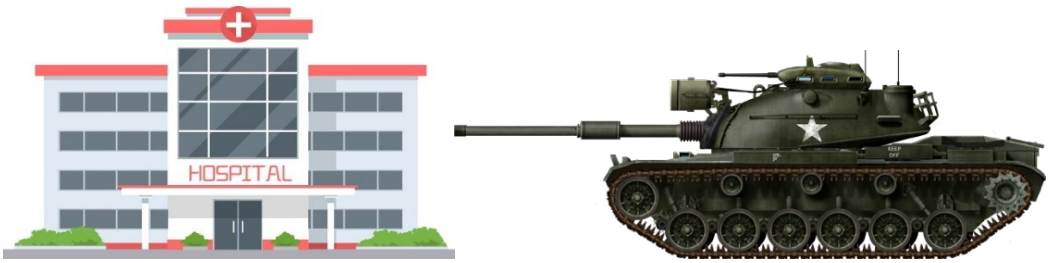
grafWall(int px, int py, double tilt = 0, bool bounce)

Clase grafica correspondiente a las paredes presentes en cada nivel, puede ser una pared de caucho que permite al proyectil rebotar, o una pared de acero que lo detona. Contiene dos argumentos de posición tipo int (int px, int py), un argumento double opcional para su inclinación (double tilt = 0) y un argumento bool que me dice si la pared permite rebotes (bounce). Su posición e inclinación son dados por la clase “wall” de la cual depende, y es actualizado por este a través de un método que lo llama una sola vez. Utiliza herencia de la clase QPixmap para graficar el Sprite .png del proyectil, y además utiliza transformaciones para girar el rectángulo que posee un brush distinto para las paredes de acero que no permiten rebotes y las paredes de caucho que si lo permiten, los cuales estarán alojados en una carpeta de recursos para un más fácil acceso ya que será un brush de Sprite.

Instancias de grafWall con diferentes valores de bounce y tilt, siendo la negra la pared que permite rebotes (bounce = True).

grafTarget(int px, int py, int image, double tilt)

Clase grafica correspondiente a los objetivos y obstáculos presentes en cada nivel, puede ser un objetivo al cual destruir o preservar. Contiene dos argumentos de posición (int px, int py) y un argumento que define el Sprite a usar (int image). Su posición es dada por la clase "target" de la cual depende, y es actualizado por la misma una vez si es estático o dinámicamente si se mueve. Una vez destruido cambia su Sprite. Utiliza herencia de la clase QPixmap para graficar el Sprite .png del objetivo u obstáculo y transformaciones de Qt para cambiar su orientación, el Sprite estará alojado en una carpeta de recursos.

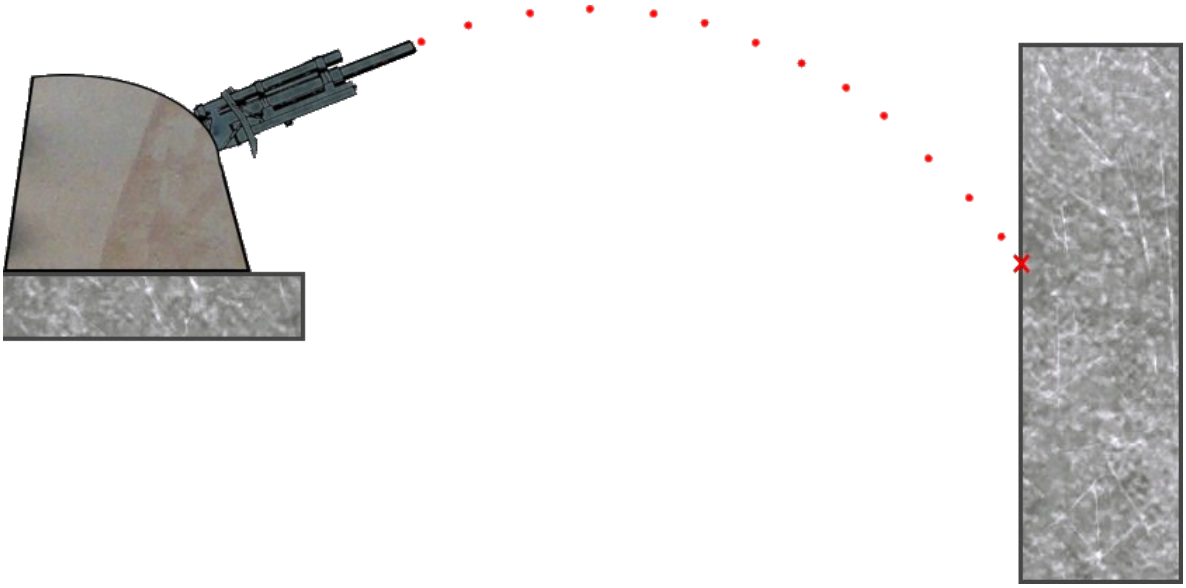


Objetivos con diferentes Sprites gracias a la clase grafTarget

scope(int px, int py, double vx, double vy, double wind = 0)

Clase grafica correspondiente al predictor de trayectoria visible para dificultades mas fáciles. Permite ver donde colisionara por primera vez el proyectil, dejando puntos rojos mientras vuela y una "X" roja al colisionar. Recibe su posición como variables enteras (int

px, int py) del cañón a través de un método similar a disparar, y calcula su trayectoria con una gravedad fija, velocidades iniciales (vx, vy), y además tiene en cuenta velocidad del viento si la hay (double wind = 0). Para generar los círculos simplemente pinta una bola con QPainter y una X con un Sprite, estos colocados en sus posiciones respectivas respecto a la posición del predictor balístico, el cual genera una partícula invisible que es disparada muy rápido para facilitar el apuntar con el cañón.



Predictor balístico siento utilizado para predecir donde caerá el proyectil