Documentación técnica

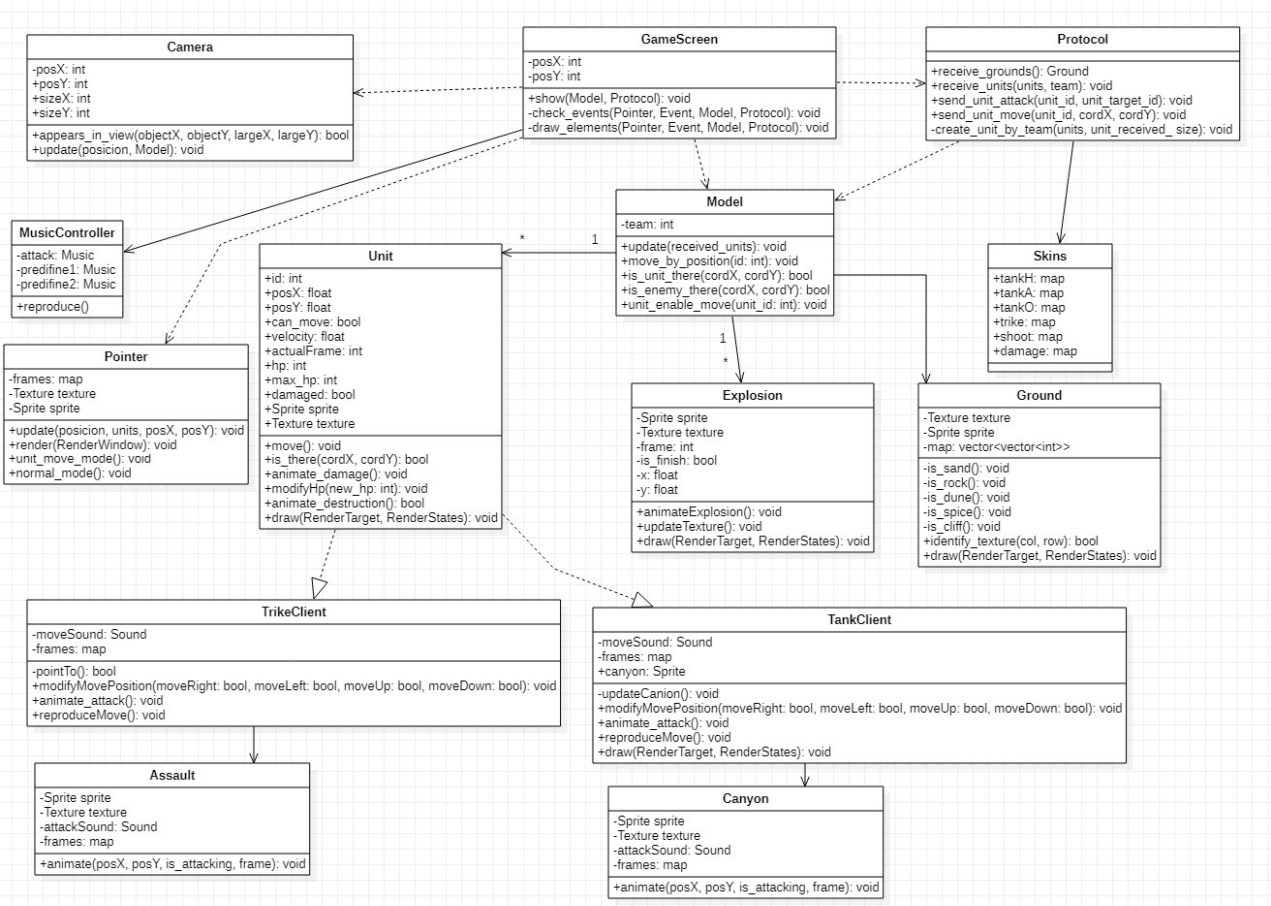
Cliente

El funcionamiento del programa desde el lado del cliente se basa en la clase GameScreen, esta es la que se encargará del loop principal del juego, mostrando por pantalla y chequeando todos los eventos que ingrese el jugador mediante la utilización del mouse, esta clase GameScreen será dueña de una clase Model, que será la que contendrá toda la lógica de la parte del cliente, entiéndase esto como dos map, uno para todos los Build y otro para todas las Unit, una clase Ground, encargada de dibujar el terreno, así como un vector que contendrá otros scripts a ejecutar como Explosión.
GameScreen también será dueño de una clase Protocol, ya que en cada loop del juego será con este Protocol con el que recibirá el status actual del juego y pasando este a Model actualizará la pantalla, también cuando revise los eventos del mouse y el modelo certifica que, por ejemplo, se seleccionó una unidad y luego se hizo click en un lugar del mapa, le envía al servidor una orden de movimiento.
La clase Protocol instanciara un struct Skins dueño de todos los map que contienen los frames de los elementos animables del juego, ya que cuando cree una unidad usara este struct para pasarle por referencia ese map con las posiciones de cada frame, esto lo hará para crear las animaciones de movimientos y rotación. Este sistema implementado funciona de la siguiente manera, para explicarlo tomare de ejemplo un Trike:

Cuando se crea un trike se le selecciona por única vez esta textura, y se le pasa por referencia el ya mencionado map de frames, este map contendrá como clave el numero del frame (siendo en este caso 32) y como valores dos posiciones x,y siendo estas la posición de ese frame respecto a la textura. Entonces cuando se cree el Trike, se le instanciara primero un frame actual, en este caso el 17, por lo tanto cuando se realice una animación, por ejemplo la orden de moverse hacia arriba a la derecha, se vera que el frame destino es el 5, se preguntara si el frame actual es mas chico o mas grande que el frame destino, si es mas chico aumentara el frame actual y en caso contrario lo reducirá, finalmente se accede a ese frame en el map y se obtiene el próximo “cuadrado de recortar” que seria el del frame 16, y asi se van formando las animaciones de rotación y movimiento de todas las unidades.

También hay otras clases aunque son más de detalles como la clase Pointer, que se encarga de actualizar el puntero del mouse cambiándole la skin de acuerdo a la acción realizada por el jugador y la posición en la que se encuentre. O la clase MusicController que setea la música en todo momento, es decir si esta debe ser una predefinida o de ataque.

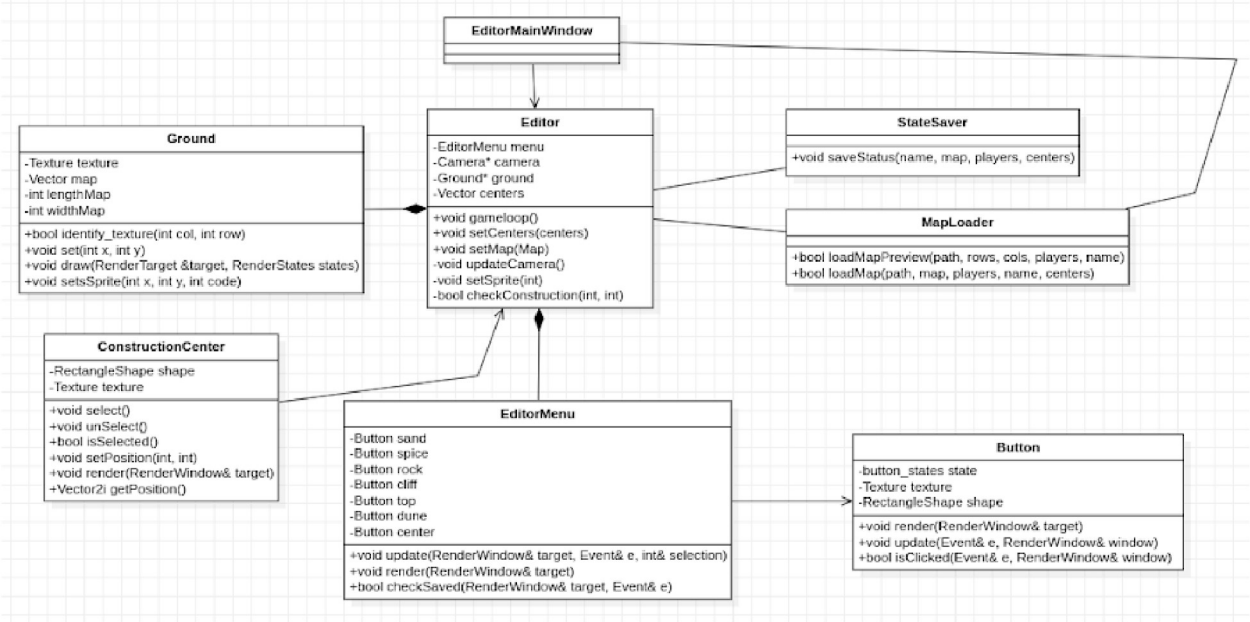
Diagrama de clases (se obviaron atributos y métodos para facilitar la legibilidad):



Editor

En primer lugar se incluye un diagrama de clases representando las partes más importantes del editor:

```
 void  Editor::gameloop() {  int  TSIZE =
    ConstantGetter::getTsize();
    StateSaver saver;  int  selection =  1
    ;  bool  dragMode =  false;
    sf::Music music;  if
(!           music.openFromFile(ConstantGetter::getResourcePath()+
"music.ogg"
 ))  throw  EditorException( "Error abriendo archivos") ;
```



Para la realización del editor se utilizaron principalmente dos herramientas. Las librerías SFML Y QT, a su vez para las partes con QT se utilizó su IDE (qt designer, con el que se simplificó significativamente el trabajo).

El flujo del editor comienza en EditorMainWindow, un objeto programado en su mayoría a través de las librerías de qt y cuyo objetivo es el de establecer qué tipo de partida quiere el usuario (si nueva o cargar una). Por otro lado, una vez realizado este paso, se procede a la ventana principal del editor, vale aclarar que el loop principal del juego se encuentra en el objeto Editor, este contiene la lógica del editor, capturando a través de gameloop() todos los eventos que van sucediendo en el programa e interactuar con los demás objetos para actualizarlo consecuentemente.

```
     music.play();
    while (! game_over)
    {
loop
}
```

Por otro lado, una de las principales clases es la de EditorMenu, la cual es básicamente la botonera a través de la cual el usuario selecciona los distintos tipos de terrenos y los centros de construcción. La misma se encarga de renderizar todos los botones y servir como contenedor de los mismos. En cuanto a los botones, estos están implementados a través de la clase Button, los botones no realizan acciones, simplemente reaccionan ante los eventos del mouse y al realizar un “update” a través de la función is_Clicked() estos comunican si fueron o no clickeados y el editor actúa en consecuencia.

Adentrándonos en el mapa propiamente dicho, este, es un vector de vectores de enteros, cuyos valores indican cada uno de los distintos terrenos del juego, para

representarlo se utiliza la clase Ground, que no es más que un sprite sfml que dibuja los terrenos según corresponda. Por otra parte tenemos la clase constructionCenter, que sirve para representar los centros de construcción colocados y que guarda la posición de los mismos. Vale aclarar que el chequeo para colocar los centros de construcción en una posición específica del mapa se realiza a través de la clase Editor en su funcion checkConstruction().

Otros dos objetos que vale la pena mencionar son StateSaver y MapLoader, cuya función es la persistencia dentro del editor, es decir, el guardado y la carga de los mapas. Para el guardado de archivos del tipo yaml, se utilizó la librería yaml-cpp (referencia: https://yaml-cpp.docsforge.com)

Por ejemplo, para la carga de archivos:

```
 bool  MapLoader::loadMap( std :: string & path,  std ::
vector < std :: vector < int>> &  map,   int & players,  std
:: string &  name,  std :: vector< ConstructionCenter*>&
centers) {  YAML::Node config = YAML::LoadFile(path);
     map =  config[ "map"] .as< std :: vector < std :: vector<
    int>>>() ;  players = config[ "players"] .as< int>() ;  name
    = config[ "name"] .as< std :: string>() ;  for  ( int  i =  0
    ;  i  <  players  ;  ++i)  {   std  ::  string   number  (  std::
    to_string(i));  auto  *c =  new  ConstructionCenter();  auto
    n= config[ "players"+  number].as< std :: vector < int>>() ;
    c->setPosition(n[ 0 ],n[1  ]);
```

```
         centers.push_back(c);
     }  return
    true ;
 }
```