Fiuba
Facultad de Ingeniería
Universidad de Buenos Aires

U.B.A. FACULTAD DE INGENIERÍA

Departamento de Computación

Taller de Programación 75-42
Informática

TRABAJO PRÁCTICO FINAL
Z

Documentación Técnica

Curso 2017 - 1er Cuatrimestre

GRUPO N° 2	
APELLIDO, Nombres	**N° PADRÓN**
MAITIA, Darius	95436
BOSCH, Martin	96749
TEJERINA, Luis	96629

Requerimientos de Software

Para la instalación:

- Cmake versión >= 2.7

Para la ejecución:

- Gtkmm v3.0
- SDL2:
    - SDL_Image
    - SDL_TTF
- gcc versión >=5.4.1
- GNU + Linux

Descripción general

El proyecto posee una arquitectura modular cliente-servidor. Sus tres módulos principales son: cliente, servidor y generador de mapas. La comunicación entre cliente y servidor se realiza por sockets y para el archivo de configuración y salida del generador de mapas se eligió el formato JSON. Además se utilizaron dos librerías externas del tipo header only para el manejo de los archivos JSON y para la serialización de los datos a enviar. Para el manejo de archivos se escogió la librería JSON for modern C++[1] y para la serialización de clases y estructuras se escogió la librería cereal[2].

Descripción por módulos

Clases, eventos, comandos y protocolo comunes a los módulos cliente y servidor

Clases

Las clases que comparten los módulos son en general las necesarias para poder establecer el protocolo de comunicación entre cliente y servidor, además del mapa que se vió conveniente que sea compartida también.

[1] Niels Lohmann, JSON for modern C++. https://nlohmann.github.io/json/
[2] Grant, W. Shane and Voorhies, Randolph. cereal - A C++11 library for serialization. http://uscilab.github.io/cereal/

Clases referidas al mapa.

El mapa fue modelado como un conjunto de Tiles, cada cual con un único tipo de terreno y tamaño fijo. Las posiciones dentro del mapa se manejaron como números enteros positivos encapsulados en una clase especial. De forma arbitraria se definió que el tamaño en pixeles de cada Tile sea de 100 y su forma cuadrada.

Map

Esta clase representa el mapa sobre el que se desarrolla la partida. Se encarga de contener el estado de los distintos objetos que hay sobre el terreno y actualizarlos para que otros objetos puedan consultarlos.

`std::vector<Tile> getNeighbors(const Tile &tile):`
Este método retorna un vector de los Tiles vecinos de otro Tile.

`void updateUnit(const UnitID &unitID, const UnitState &unitState):`
Actualiza el estado de una unidad.

`void updateBullet(const BulletID &bulletID, const BulletState &bulletState):`
Actualiza el estado de una bala.

`void updateBuild(const BuildID &buildID, const BuildState &buildState):`
Actualiza el estado de un edificio.

`void updateTerrainObject(const TerrainObjectID &id, const TerrainObjectState &newState):`
Actualiza el estado de un objeto sobre el terreno.

`void updateCapturable(const CapturableID &id, const CapturableState &state):`
Actualiza el estado de un capturable.

Tile

Esta clase representa un tile y se encarga de manejar su estado, pudiéndose convertir en un tile no atravesable por las unidades durante el transcurso del juego si así se requiere. Además contiene la información sobre el tipo de terreno que contiene.
`void makeNotPassable():`
Transforma el tile en no pasable.

`TerrainType getTerrainType():`
Retorna el tipo de terreno del tile.

Position

Esta clase encapsula el concepto de posición, internamente definido como un par de números positivos. Permite realizar operaciones entre ellos que son requeridas tanto por el cliente como el servidor.

`unsigned long chebyshevDistance(const Position &other):`
Retorna la distancia Chebyshev entre dos posiciones, esta es la mayor entre las distancias comparadas de cada componente de la posición.

`void move(Position target):`
Mueve la posición un paso hacia la posición destino.

Clases referidas a ID's

Para homogeneizar la forma en que se identifican las diferentes entidades del modelo que representan los elementos del juego, a los jugadores y a los equipos y cómo los clientes se refieren a estas en los mensajes del protocolo se crearon clases que funcionan como identificadores de instancias de estos elementos del juego. Cada una de estas clases contiene un número que identifica unívocamente a una instancia de la clase a identificar en cuestión. Estas clases se listan a continuación:

- `BuildID`
- `BulletID`
- `CapturableID`
- `PlayerID`
- `TeamID`
- `TerrainObjectID`
- `UnitID.`

Enumerados que identifican tipos

Para poder modelar algunos atributos de las entidades del juego se necesitó crear los enumerados que se listan a continuación:

- `BuildType`
- `CapturableType`
- `CommandType`
- `EventType`
- `PlayerColor`
- `UnitType`
- `TerrainObjectType`
- `TerrainType`
- `WeaponType`

Estructuras que modelan estados de entidades del juego

Para ciertas entidades del juego fue necesario crear estas estructuras que agrupan los atributos (salud, posición, etc) que representan el estado actual de ésta. Los estados creados fueron los siguientes:

- `BuildState`
- `BulletState`
- `CapturableState`
- `TerrainObjectState`
- `UnitState`

Datas

Estas estructuras son las que se serializan para la comunicación entre el cliente y el servidor, hay tres tipos: las de los eventos, que contienen los datos necesarios para instanciar los eventos del cliente; las de los comandos, que contienen los datos necesarios para invocar los métodos del modelo y las de los mensajes de conexión inicial de los jugadores que contienen datos como el equipo al que se quiere unir, el mapa que eligió, etc. Por lo que existe una de éstas por cada evento, comando y mensaje de conexión inicial.

Clases auxiliares

Para el manejo de sockets y threads se crearon dos clases auxiliares, Socket y Thread.

Eventos

Los eventos representan los diferentes sucesos que ocurren durante el transcurso del juego y deben ser informados a los clientes. Son generados por el servidor y enviados de forma simultánea a todos los clientes. Las clases que representan los eventos en el servidor difieren a las de los clientes porque su comportamiento es diferente, es por esto que para cada evento hay dos clases que lo representan: una en el cliente y otra en el servidor. La estructura de eventos que comparten cliente y servidor son los DataEvents que contienen los atributos que sirven de información del evento, cuyo origen está en el servidor y se trasmite hacia los clientes para que la procesen y muestren en pantalla. La lista de eventos utilizados es la siguiente:
(Se infiere que el nombre es descriptivo de su intención)

Eventos de edificios: `BuildDamage, BuildDestroyed, BuildUpdateTime.`

Eventos de balas: `BulletNew, BulletMove, BulletHit.`

Eventos de capturables: `Capture.`

Eventos del juego: `GamePlayerDefeat, GameEnd.`

Eventos de objetos sobre el terreno: `TerrainObjectDestroyed`.

Eventos de unidades: `UnitCreate, UnitMove, UnitStill, UnitAttack, UnitDeath`.

Comandos

Los comandos simbolizan las acciones que un cliente puede realizar sobre el juego, en el servidor estas están modeladas como métodos de la clase `GameController`. Igual que en el caso de los eventos las clases que los representan difieren en comportamiento en el cliente y en el servidor, pero ambos módulos comparten las estructuras `DataCommand` que contienen los atributos necesarios para la ejecución de los comandos. Los comandos utilizados se listan a continuación.

Comandos sobre unidades: `UnitAttackUnit, UnitAttackBuild, UnitAttackTerrainObject, UnitCapture, UnitMove`.

Comandos sobre edificios: `BuildChangeUnit`.

Protocolo

Para toda comunicación entre el cliente y el servidor se utiliza el protocolo TCP. Para la serialización de las estructuras transmitidas se utilizó la librería cereal, utilizando el modo binario de serialización para reducir el tamaño a enviar. Las estructuras serializadas son enviadas como strings con tamaño preconcatenado, es decir siempre que se envía una estructura serializada se espera recibir primero un entero sin signo de 32 bits. Esta lógica fue encapsulada dentro de la clase Socket en los métodos `send_str_preconcatenated()` y `rcv_str_preconcatenated()`.

Antes de iniciado el juego hay una primera instancia donde, luego de establecida la conexión, el cliente envía un `DataClientConnectedMessage` al servidor, recibida esta información el servidor envía al cliente un `DataServerClientAccepted`. A continuación el servidor envía al cliente el mapa que cargó de acuerdo a lo solicitado por los clientes en el paso anterior. Esto da comienzo al juego, por lo que luego de enviado el mapa el servidor quedará a la espera de recibir comandos y enviará los eventos generados a todos los clientes.

Tanto la transmisión de los comandos como la de los eventos se realiza de forma idéntica. Primero se envía como un número entero el tipo de evento o comando definido en sus correspondientes enumerados y a continuación se trasmite la estructura Data correspondiente a ese tipo de evento o comando como se describió anteriormente.

Archivos

El formato de archivo utilizado para manejar el guardado de la configuración del juego y los mapas fue .json[3]. El formato Json (javascript object notation) proporciona una forma simple y sencilla de entender (y manejar) para incluso un usuario sin experiencia previa de programación, con lo que esperamos que el archivo data.json que almacena la configuración del juego sea fácilmente editable.
Los mapas por ejemplo siguen el formato siguiente:

```
   "81": {
        "bridge": false,
        "flag": false,
        "fort": true,
        "robot_factory": false,
        "rock": false,
        "terrain_type": 0,
        "territory": 1,
        "vehicle": false,
        "vehicle_factory": false,
        "x": 15,
        "y": 3
    },
```

dónde 81 es el número tile que contiene un fuerte, pertenece al territorio 1 y la ubicación en píxeles de la esquina superior izquierda del tile es de (15;3).
Además, al final del archivo se guarda la configuración del mapa:

```
   "base_terrain": 0,
    "bridge_type": 9,
    "factories_level": 2,
    "map_length": 30,
    "map_width": 22,
    "players": 2,
    "river_type": 3,
    "road_type": 6,
    "rock_type": 0,
    "territories_amount": 9
}
```

En cuanto al archivo data.json un ejemplo de lo que se encuentra en él es:

```
     "hcbullet.damage": 20,
    "hcbullet.isExplosive": true,
    "hcbullet.speed": 15,
    "hcbullet.type": 3,
    "iceRockObject.health": 1000,
    "iceRockObject.passable": false,
```

[3] "JSON." http://www.json.org/.

```
    "iceRockObject.size": 50,
    "iceRockObject.type": 1,
    "land.terrainFactor": 1,
    "land.type": 0,
    "laser.damage": 10,
    "laser.isExplosive": false,
    "laser.speed": 15,
    "laser.type": 4,
    "lava.terrainFactor": 0.1,
    "lava.type": 5,
```

En este ejemplo podemos ver y modificar fácilmente la velocidad del láser (por ejemplo), su daño, su tipo de armamento, etc...

Servidor

Descripción general

El servidor está compuesto por tres módulos principales:

- El modelo cuya responsabilidad es, iniciado el juego, manejar la lógica durante el transcurso de la partida, crear (no así emitir) los eventos que se vayan generando durante el transcurso del mismo y recibir las acciones que los jugadores realizan sobre él. Es importante destacar que el modelo no controla que el jugador que realice una determinada acción tenga el permiso de realizarla (ej: que le pertenezca la unidad), dejando la responsabilidad en el cliente. Esto elimina el envío de comandos inválidos. Éstos a su vez, no impactan de forma inmediata en el modelo ya que se avanza en el tiempo de forma discreta mediante pasos, los cuales tienen un tiempo de ejecución controlado para así poder manejar tiempos dentro de la lógica del juego. Para facilitar la modificación de los parámetros propios del juego (vida de las unidades, tiempos de fabricación, etc) el modelo utiliza una variable global de donde obtiene todas estas constantes necesarias para la lógica del mismo.

- Otro de los módulos del servidor es el que se encarga de la comunicación por sockets con los clientes, para recibir los comandos que estos envían y emitir los eventos que el modelo genera. La comunicación entre el modelo y este módulo se realiza, para los eventos, a través de una queue protegida donde el modelo coloca los eventos que luego son enviados por este módulo a todos los clientes simultáneamente y para los comandos directamente se ejecutan sobre el modelo protegido. Tanto el envío de eventos como la recepción de comandos se hacen siguiendo el protocolo establecido, cuya descripción se efectuará más adelante.

- El tercer módulo se encarga de generar, a partir de los parámetros provistos inicialmente por los clientes, las estructuras de datos necesarias para poner en funcionamiento el modelo y el módulo de comunicación cliente-servidor. Principalmente se encarga de cargar

el mapa y los parámetros de la variable global antes mencionada ambos a partir de sus correspondientes archivos JSON.

Clases

Ordenadas por orden alfabético se detallan a continuación las clases que componen el servidor, sus responsabilidades y métodos principales:

Clases del modelo:

AStar

Clase que encapsula el algoritmo AStar para el pathfinding en el mapa. El mapa, unidad y posición destino se ingresan al instanciar la clase.

`std::vector<Position> find():`
Retorna el resultado de la ejecución del algoritmo como un vector de posiciones.

Attackable

Interfaz que define los métodos necesarios para que una entidad del modelo pueda ser atacada.

`Position getAttackPosition(const Position &attacker):`
Retorna la posición en la cual el atacable tiene que ser atacado a partir de la posición del atacante. Esto permite que para ciertos tipos de atacables cuyo tamaño no es despreciable la posición a la cual el atacante ataca sea variable de acuerdo a su propia posición.

`Position nextMovePosition():`
Retorna la siguiente posición a la que se moverá el atacado, necesaria para la persecución.

`void receiveAttack(const Weapon &weapon):`
A partir de un Weapon recibe el daño procurado por este.

Build

Clase que modela un edificio, crea unidades encapsulando la cantidad de avances del modelo necesarios para que esto suceda y de acuerdo a su nivel los tipos de unidades que puede fabricar. Además implementa la interfaz Attackable, maneja los daños recibidos y si es destruido le informa a su dueño.

`void tick():`
Le informa al edificio que el modelo avanzó un paso, internamente actualiza el tiempo restante para la creación de las unidades.

`std::vector<Unit *> fabricateUnits(const Position &buildPos):`

Retorna un vector con las unidades fabricadas a partir de la posición inicial en la que se quiera fabricar, como el entorno del edificio puede variar durante el desarrollo del juego es necesario que reciba la posición donde se van a construir las unidades.

`void changeFabUnit(const UnitType &type):`
Cambia la unidad a fabricar por el edificio.

`void changePlayer(Player *player, const Team &team):`
Permite cambiarle el dueño al edificio.

+ Métodos de la interfaz Attackable

BuildData

Estructura necesaria para la variable global Data que contiene los parámetros iniciales para un tipo de edificio.

Bullet

Clase que representa el comportamiento de una bala, manejando su movimiento y informandole a su objetivo en caso de impacto.

`void move():`
Avanza un paso hacia el objetivo de acuerdo a la velocidad de la misma.

`void doHit():`
Si la bala llegó a su objetivo se lo informa.

Capturable

Clase abstracta que sirve como base para las entidades que deben poder ser capturadas.

`void capture(const UnitID &unitID, Player *newOwner, const Team &ownerTeam):`
Le informa al capturable que fue capturado por un jugador.

`std::map<BuildID, BuildState> getCapturedBuilds():`
Retorna la lista de edificios capturados.

`std::map<UnitID, UnitState> getCapturedUnits():`
Retorna la lista de unidades capturadas.

`bool capturerDissapear():`
Si el que captura tiene que desaparecer al capturar retorna true.

`bool isRecapturable():`
Si el capturable se puede volver a capturar retorna true.

`bool canBeCapturedBy(const UnitID &id):`
Si el capturable puede ser capturado por la unidad dada retorna true.

CapturableVehicle

Clase que representa un vehículo en el mapa inicialmente sin dueño que puede ser capturado por cualquier unidad robot. Implementa los métodos de la clase abstracta Capturable.

Data

Estructura que contiene toda los parámetros de configuración inicial para las unidades, edificios, objetos, etc. En general todo número y dato del enunciado está representado en esta estructura. Además de configuraciones internas del servidor.

GaiaPlayer

Esta clase representa al jugador que es dueño de los objetos del terreno y de los capturables nunca capturados. Implementa los mismos métodos que la clase madre Player pero adaptados a sus requerimientos.

GameController

Es una interfaz que define los métodos que debe tener un GameController, se utilizó para poder usar en algunos casos ambiguamente el RealGameController y el ProtectedGameController. La explicación de sus métodos se encuentra luego en la descripción de la clase RealGameController.

MovementState

Representa el estado de movimiento de una unidad.

Node

Esta clase es utilizada por el algoritmo AStar, representa un nodo posible de ser parte del camino resultado. Como tal sabe calcular el costo de moverse hacia él. Además de poder generar el camino de posiciones por el cual se llegó.

`std::vector<Position> makePath():`
Retorna el camino de posiciones por el cual se llegó al nodo.

Player

Representa un jugador en el modelo, controla si el jugador sigue vivo o no durante el transcurso del juego.

`void addTerritory():`
Le informa al jugador que capturó un nuevo territorio.

`void subTerritory():`
Le informa al jugador que dejó de ser el dueño de un territorio.

`void addUnit():`
Le informa al jugador que posee una nueva unidad.

`void subUnit():`
Le informa al jugador que ya no posee una unidad.

`void buildDestroyed(const BuildType &type):`
Le informa al jugador que un edificio que le pertenecía fue destruido.

`bool isAlive():`
Retorna true si el jugador está vivo.

RealGameController

Esta clase se creó para representar el modelo del juego, es decir, ser la receptora de los comandos a través de sus métodos e implementar el avance del modelo por pasos creando los eventos pertinentes. Para poder crear los eventos necesita controlar los otros elementos del modelo y la interacción entre ellos. Además obtiene de Data el tiempo que debe durar cada paso y controla que esto sea así.

`void move(const UnitID &unit, const Position &position):`
Le informa al modelo que un jugador quiere mover una unidad a una posición dada.

`void attack(const UnitID &attacker, const UnitID &attacked):`
Le informa al modelo que un jugador quiere atacar con una unidad a otra.

`void attack(const UnitID &attacker, const BuildID &attacked):`
Le informa al modelo que un jugador quiere atacar con una unidad a un edificio.

`void attack(const UnitID &attacker, const TerrainObjectID &attacked):`
Le informa al modelo que un jugador quiere atacar con una unidad a un objeto sobre el terreno.

`void capture(const UnitID &unit, const CapturableID &capturable):`
Le informa al modelo que un jugador quiere capturar un capturable.

`void changeUnitFab(const BuildID &buildId, const UnitType &type):`
Le informa al modelo un jugador quiere cambiar el tipo de unidad a construir de un edificio.

`void tick():`
Avanza un paso el modelo. El tiempo de ejecución de este método está controlado por el parámetro ticksPerSec de Data.

`void playerDisconnected(const PlayerID player):`
Le informa al modelo que un jugador se desconectó.

Robot

Esta clase es la especialización de Unit para modelar un robot. Re-implementa los métodos que se listan a continuación.

`bool canGoThrough(const TerrainData &terrainData):`

`unsigned short getMovementSpeed(float terrainFactor):`
Aunque es privado es importante remarcar que cada subclase de unidad calcula de diferente manera la velocidad de movimiento.

`UnitState getUnitState():`

Team

Esta clase representa un equipo conformado por jugadores. Permite saber si el equipo está vivo o no y si un determinado jugador es enemigo del equipo o no.

`bool isEnemy(const PlayerID &playerID):`
Devuelve true si el jugador es enemigo del equipo.

`bool isTeamAlive():`
Devuelve true si el equipo está vivo.

TerrainObject

Representa un objeto del terreno, es decir, engloba el concepto de puentes y rocas. Implementa la interfaz Attackable y recibe los daños para saber informar su estado.

TerrainObjectData

Estructura necesaria para la variable global Data que contiene los parámetros iniciales para un tipo de TerrainObject.

Territory

Esta clase representa un territorio, su principal responsabilidad es informar a los edificios pertenecientes a él y al anterior jugador cuando el territorio fue capturado. Su posición de captura es la de la bandera que lo identifica. Implementa los métodos de la clase abstracta Capturable.

Unit

Clase abstracta que modela una unidad, su estado y comportamiento. Recibe las acciones que cambian su estado de movimiento y las ejecuta separadamente cuando se llama a alguno de los métodos correspondiente. Estas acciones son ejecutadas en pasos y se controlan los avances necesarios para ejecutarlas. Además implementa la interfaz Attackable y recibe los daños modificando su estado e informando al dueño de su destrucción.

`void move(const std::vector<Position> &movementsPositions):`
Cambia el estado de la unidad a Moving y comienza a moverse de acuerdo al vector de posiciones dado.

`void capture(const std::vector<Position> &movementsPositions, Capturable *capturable):`
Cambia el estado de la unidad a Capturing y comienza a moverse de acuerdo al vector de posiciones dado.

`void hunt(const std::vector<Position> &movementsPositions, Attackable *other):`
Cambia el estado de la unidad a Hunting, comienza a moverse de acuerdo al vector de posiciones dado y conserva la referencia al atacable.

`void autoAttack(Attackable *hunted):`
Cambia el estado de la unidad a AutoAttacking y conserva la referencia al atacable.

`void doMoveWithSpeed(float terrainFactor):`
Ejecuta el siguiente movimiento con la velocidad correspondiente a la unidad y modificada por el factor del terreno.

`void addMove(const Position &position):`
Añade a la cola de movimientos una posición destino.

`void receiveDamages():`
Recibe los daños pendientes y actualiza su estado.

`void still():`
Cambia el estado de la unidad a STILL.

`void kill():`
Mata la unidad forzosamente.

`bool isTimeToAttack():`
Devuelve true si es momento de atacar, de lo contrario se informa que el modelo avanzó un paso y retorna false.

`bool canGoThrough(const TerrainData &terrainData):`
Retorna true si la unidad puede pasar por ese tipo de terreno.
+ Métodos de la interfaz Attackable

UnitData

Estructura necesaria para la variable global Data que contiene los parámetros iniciales para un tipo de unidad.

UnitFactory

Es una clase con métodos estáticos que encapsulan la creación de unidades.

`static Unit* createUnitDynamic(const Position &pos, const UnitType &type, Player& player, Team &team):`
Retorna un puntero a una unidad creada de acuerdo a los parámetros dados.

`static Vehicle* createVehicleDynamic(const Position &pos, const UnitType &type, Player& player, Team &team):`
Retorna un puntero a un vehículo creado de acuerdo a los parámetros dados.

Vehicle

Esta clase es la especialización de Unit para modelar un vehículo. Re-implementa los mismos métodos que la clase Robot. Pero además agrega la lógica para un conductor y el cambio de dueño.

`bool canGoThrough(const TerrainData &terrainData):`

`unsigned short getMovementSpeed(float terrainFactor):`

`UnitState getUnitState():`

`void changeOwner(Player *player, const Team &team, UnitType conductor):`
Cambia el dueño del vehículo y su conductor.

Weapon

Estructura que contiene los datos que representan un tipo de arma. Es utilizado por Data para los tipos de arma de las unidades y por Attackable para recibir los datos del arma del que se recibe el daño.

Clases del módulo de comunicación cliente-servidor

Command

Es la interfaz que implementan todos los comandos del servidor. Habrá una especialización por comando, es decir, una por método del GameController que pueda ser accedido por los clientes.

`void execute(GameController &gameController):`
Ejecuta el comando en el gameController pasado como parámetro.

CommandFactory

Esta clase define un método estático para la creación dinámica de los comandos a partir de la información proveniente de los clientes de acuerdo al protocolo.

`Command* createCommand(const CommandType &type, std::stringstream &ss):`
Dado el tipo y el stringstream que contiene al comando serializado retorna un puntero a un comando que puede ser ejecutado.

CommandReceiver

Esta clase es una especialización de Thread en donde al correr el hilo recibe según el protocolo establecido los comandos que envía un cliente para luego ejecutarlos inmediatamente, esto implica que existirá una instancia (y un thread) de esta clase por cliente conectado.

EventSender

Esta clase es una especialización de Thread en donde al correr el hilo saca de la cola de eventos un evento a la vez y lo envía simultáneamente a todos los clientes de acuerdo al protocolo.

PlayersManager

Esta clase se encarga de recibir a los jugadores previo al inicio del juego y crear a partir de los datos que estos le envían de acuerdo al protocolo las instancias de Player y Team necesarias.

`void receivePlayers():`
Implementa el protocolo para recibir a los jugadores a partir de la lista de clientes que recibió al instanciarse la clase.

ProtectedGameController

Es la versión protegida para concurrencia de GameController, implementa los métodos de esta interfaz.

ServerEvent

Esta clase abstracta representa un evento del servidor, por lo que habrá una especialización por cada tipo de evento que mencionaremos luego. Se encarga de encapsular la serialización de los eventos para ser enviados a los clientes. Las especializaciones de esta clase hacen uso de la librería cereal.

`std::stringstream getDataToSend():`
Retorna el evento serializado como un string stream.

Clases de carga del juego

GameLoader

Esta clase se encarga de crear a partir de un archivo JSON que representa un mapa las estructuras de datos y objetos necesarios para que se inicie el juego, principalmente crea una instancia de Map y otras estructuras de datos para que sean manejadas por la clase RealGameController.

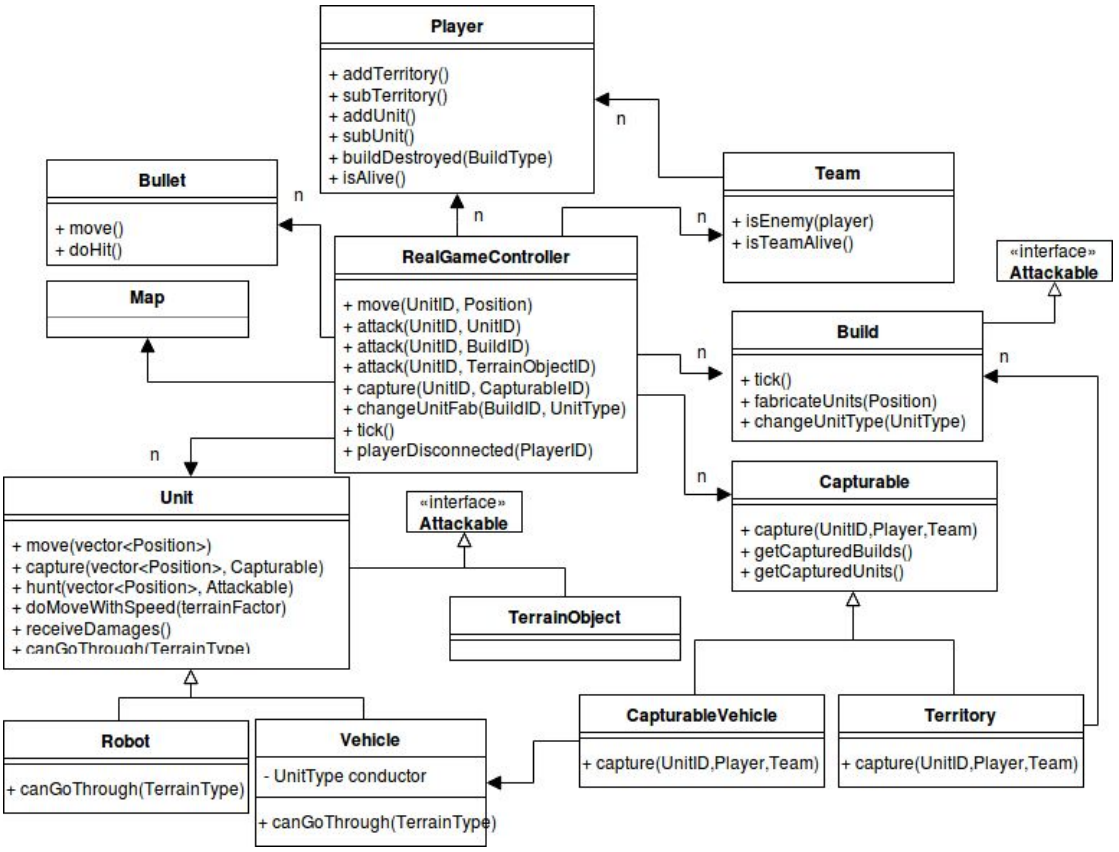
`Map run():`
Retorna un Map cargado a partir de los parametros pasados en la construcción del GameLoader.

`Json_to_Data`
Esta clase encapsula la lógica necesaria para cargar el archivo de configuraciones.

`void load_file():`
Carga el archivo JSON en el objeto Data pasado como parámetro en la construcción del objeto.

Diagramas UML

El siguiente es un diagrama de clases principales del modelo.



Cliente

Descripción general

Para la implementación del cliente se utilizó el patrón MVC[4] por lo que este consta de tres módulos principales:

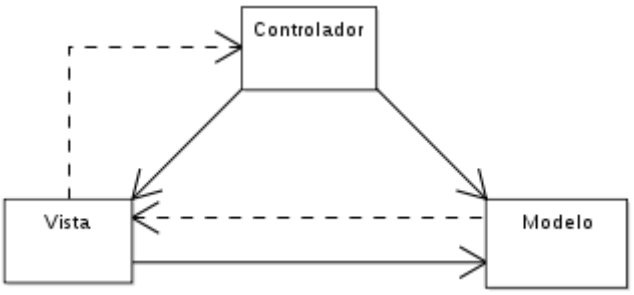
- Controlador: su responsabilidad es manejar las entradas del jugador, estos son eventos producidos por la librería SDL. De estos eventos, los que interesan para el juego son traducidos a eventos propios que indican la acción realizada por el jugador, que luego serán procesados y manejados por el modelo o la vista.

- Modelo: su responsabilidad es manejar las acciones del jugador (ej click derecho e izquierdo) las cuales son informadas por el controlador, identificar qué acción quiere

[4] Modelo–vista–controlador - Wikipedia, la enciclopedia libre

realizar (ej seleccionar, capturar, etc.) y validarlas (ej el jugador solo pueda seleccionar sus unidades).

- Vista: su responsabilidad es procesar los eventos, tanto del modelo como del controlador, actualizar las distintas vistas y finalmente dibujarlas, es decir, mostrar adecuadamente en imágenes el estado actual del modelo.



Relación entre los componentes del patrón MVC[5]

Además de los tres módulos anteriores es importante mencionar los eventos. Estos son generados por el controlador o por el server, y su responsabilidad es:

- actualizar el modelo: para que el modelo del cliente esté sincronizado/actualizado con el del servidor, esto permite que las validaciones y las acciones realizadas por el cliente sean válidas.
- actualizar la vista: para que esta dibuje el verdadero estado actual del modelo, ya que esta no lee el modelo al dibujar sino que los eventos se encargan de mantenerla sincronizada con el modelo.

Clases

Se detallan a continuación las clases que componen el cliente, sus responsabilidades y métodos principales.

Clases del modelo

Model

Su responsabilidad es manejar las acciones de click izquierdo y derecho realizadas por el jugador, y validarlas.

```
void leftClick(int x, int y):
```

Identifica qué acción quiere realizar el jugador, estas son:

- seleccionar una unidad, validando que esta le pertenezca.
- seleccionar un edificio, para ver el menu de producción, también validando que este le pertenezca.

[5] https://es.wikipedia.org/wiki/Modelo%E2%80%93vista%E2%80%93controlador#/media/File:ModelViewControllerDiagram_es.svg

- si no es ninguna de las anteriores, lo único que se realiza es no mostrar ninguna unidad seleccionada en el panel lateral

`void rightClick(int x, int y):`
Identifica qué acción quiere realizar el jugador, para esto debió de haber seleccionado una unidad anteriormente, estas son:

- capturar un vehículo o territorio.
- atacar una unidad, edificio, roca o puente. Que esta sea enemiga se valida en el gameController del servidor.
- mover la unidad seleccionada.

GameControllerProxy

Esta clase es un proxy para la clase GameController del servidor, es decir implementa la misma interfaz y encapsula el hecho de que las clases que la utilicen no se están comunicando directamente con el servidor sino que cada acción realizada se encola como un comando en una queue de comandos.

Clases del módulo de comunicación cliente-servidor

ClientCommand

Es la interfaz que implementan los comandos del cliente. Habrá una especialización por comando, es decir, una por método del GameController.

ClientCommandSender

Esta clase es una especialización de Thread, saca de la cola de comandos un comando, de uno por vez, y lo envía al server de acuerdo al protocolo.

ClientEventReceiver

Esta clase es una especialización de Thread, recibe según el protocolo establecido los eventos que envía el server, los cuales encola en la cola de eventos para que luego sean procesados por la vista.

EventFactory

Esta clase define un método estático para la creación dinámica de los eventos a partir de la información proveniente del servidor de acuerdo al protocolo.

`Event* createEvent(const EventType &type, std::stringstream &ss):`
Dado el tipo y el stringstream que contiene al evento serializado retorna un puntero a un evento que puede ser procesado.

EventHandler

Su responsabilidad es guardar los eventos en el orden en que estos ocurrieron y son agregados, por esto se utiliza una cola para guardarlos, para que luego sean procesados por la vista en el orden que ocurrieron.

`void add(Event *event):`
agrega el evento a la cola y le agrega la vista y el modelo, para que luego al ser procesado el evento este los pueda modificar.

`Event* get() :`
retorna el evento más antiguo.

Event

Es una clase abstracta, de la cual deben heredar todos los eventos del juego.

`void process() :`
método que debe implementar cada especialización de la clase Event. Este debe actualizar tanto la vista como el modelo del cliente.

Clases del controlador

Controller

Su responsabilidad es manejar los eventos de SDL. Estos son convertidos a eventos propios, en caso que sean eventos que interesan, para luego ser procesados.

`void handle(SDL_Event *e) :`
maneja los eventos de SDL, producidos por acciones del jugador, convirtiéndolos a eventos propios para que luego sean procesados por la vista.

Handler

Interfaz que se utiliza para convertir los eventos de SDL que interesan en eventos propios.

`void handle(SDL_Event *e, EventHandler &eventHandler) :`
método que debe implementar cada especialización de la clase Handler. Este debe insertar en el eventHandler el evento propio que corresponda de acuerdo al tipo de Handler.

HandlerFactory

Su responsabilidad es convertir los eventos de SDL que interesan en eventos propios. Los eventos que interesan son:

- click izquierdo
- click derecho
- tecla escape

`static Handler* get(SDL_Event *e) :`
en caso que el evento interese devuelve un Handler que agregara el evento que corresponda al eventHandler, en otro caso devuelve un nullptr.

Clases de la vista

View

Clase principal que contiene todos los objetos a ser dibujados. Sus responsabilidades son:

- procesar todos los eventos, estos provienen del controlador y del server.
- actualizar los objetos que requieren ser actualizados, que son las unidades y balas para moverse, y las explosiones que una vez que realizan todo un ciclo de dibujo son eliminadas.
- dibujar todos los objetos del juego.

`void tick() :`
Avanza un paso la vista, la cantidad de veces que lo hace por segundo lo define la variable fps.

`void update() :`
Por cada tick() que realiza la vista actualiza aquellos objetos que lo requieren. Estos son:

- las unidades y balas para moverse.
- las explosiones, ya que una vez que dibujan todas sus imágenes son borradas.

ObjectView

Clase abstracta que representan las imágenes del juego, tanto animaciones como imágenes fijas, agrupando los atributos que tienen en común estas.

`void draw(SDL_Renderer *render, Camera &camera) :`
método que debe ser redefinido por las imágenes y animaciones. Este método es llamado a la hora de mostrar las imágenes.

Image

Esta clase modela una sola imagen y se utiliza para encapsular la clase SDL_Texture de SDL haciendo uso del patrón de diseño RAII[6]

`void draw(SDL_Renderer *render, Camera &camera) :`
método encargado de dibujar la imagen en cuestión. Esta solo es dibujada si se encuentra dentro de la cámara.

[6] RAII - Wikipedia, la enciclopedia libre

Sprite

Clase que representa una animación. Está compuesta por varias Image's.

`ViewPosition getDrawPosition() const :`
retorna la posición donde se dibuja la imagen, que es distinta de la posición en el modelo, ya que estas se dibujan de la esquina superior izquierda y para que coincida el medio de la imagen con la posición del modelo, estas se deben desplazar.

`void draw(SDL_Renderer *render, Camera &camera) :`
se encarga de manejar la lógica de que imagen mostrar, en base al frame actual y la velocidad de la animación, y de si la animación ya realizó todo un ciclo, es decir, ya dibujó todas sus imágenes (esto es útil para las explosiones que deben dibujarse hasta que realizan solo un ciclo).

ViewFactory

Clase que encapsula la creación de cualquier imagen del juego. Esto es, determina de acuerdo al tipo pasado por parámetro la cantidad de frames, la velocidad, el desplazamiento, etc de la animación, devolviendo un Sprite o Image válido que represente a ese tipo.

ViewPosition

Clase que representa a la posición en la vista, que cambia respecto al modelo ya que en este caso el x,y son flotantes, lo que permite mayor precisión en la ubicación y rotación.

`double getRotation(const ViewPosition &pos) :`
devuelve el ángulo entre ambas posiciones. Esta es utilizada a la hora de dibujar las imágenes, por ej, en el caso de las balas ya que la imagen no cambia con la rotación. Por lo tanto es ángulo devuelto no es el convencional de 0 a 360°, sino que la relación con estos es:

Ángulo convencional(°)	Ángulo devuelto(°)
0	0
90	-90
180	-180 o 180
270	90

Este ángulo devuelto es utilizado en la función `SDL_RenderCopyEx()` para rotar las imágenes.

`int getDrawRotation(const ViewPosition &pos) :`
devuelve ángulos múltiplos de 45°. Este es utilizado para saber qué imagen utilizar al dibujar por ej las imágenes de las unidades, que cambian de acuerdo a la rotación.

MainWindow

Esta clase modela una ventana y se utiliza para encapsular la clase SDL_Window de SDL haciendo uso del patrón de diseño RAII. No tiene comportamiento ya que solo se utiliza el render de la ventana para dibujar sobre ella en la clase View.

Camera

Esta clase modela la cámara del juego, permite no tener que dibujar todo el mapa sino solo los objetos del mapa que se encuentran dentro de las coordenadas de la cámara, dadas por el x, y, ancho y alto de esta. Esta cámara se mueve con el mouse.

`void move(int x, int y) :`
Mueve la cámara en la dirección que corresponda de acuerdo a los valores de x, y pasados por parámetro.

ObjectViewMove

Esta clase encapsula la lógica de movimiento en la vista de aquellos objetos que se muevan, como las unidades y las balas, por lo que todo objeto que se mueva debe heredar de esta clase.

Para el movimiento se calcula la distancia entre la posición actual y a la que indica el server que se debe mover, se divide esta distancia por la velocidad de la vista respecto al server (ya que en el próximo tick del server vendrá una nueva posición) y se va sumando este resultado empezando por la posición actual, obteniendo posiciones que se van encolando en esta clase (este cálculo se realiza en el evento UnitMoveEvent y BulletMoveEvent), que serán a las que se irá moviendo la unidad en cada tick de la vista.

`void update() :`
Este método se encarga de mover el objeto. En caso que el objeto tenga posiciones a las que moverse, se desencola esta posición y se mueve el objeto hacia ella teniendo en cuenta si debe cambiar su rotación.

`void walk(int rotation, const Position &posTo) :`
Este método debe ser redefinido en las clases que desean moverse en la vista, realizando la acción que corresponda, por ej, cambiar la imagen a la nueva rotación o setear esta nueva rotación.

UnitView

Esta clase representa a la vista de los robots y los vehículos, con sus estados y comportamientos en relación a la vista. Sus métodos reflejan las acciones que pueden realizar estos, mostrando la imagen que corresponda en cada caso.

`void walk(int rotation, const Position &posTo) :`
Redefine el método de la clase anterior. En este caso el método cambia la imagen de “walk” a la nueva rotación.

RobotView

Esta clase modela a los robots en particular. Casi todo su comportamiento se encuentra definido en la clase UnitView de la cual hereda.

VehicleView

Esta clase modela a los vehículos en particular. Lo que agrega nuevo respecto a UnitView es el top (arma) de los vehículos (con toda su complejidad), que se comporta de manera independientemente al vehículo en sí, por ej, al moverse éste a cualquier dirección el top sigue girando de la misma forma o al disparar solo se gira el top y no el vehículo.

`void fire(const Position &huntedPos) :`
En este caso no se mueve la unidad hacia la dirección del “hunted”, sino que el vehículo sigue apuntando en la dirección de antes de empezar a atacar y lo que se mueve es el top del vehículo.

BulletView

Esta clase modela a las balas, en particular, su movimiento y rotación.

`void walk(int rotation, const Position &posTo) :`
Redefine el método de la clase ObjectViewMove. En este caso el método no cambia la imagen a la nueva rotación ya que la imagen de la bala es la misma para cualquier rotación, entonces se utiliza el método getRotation() de ViewPosition para obtener una rotación más precisa y setearsela a la bala para que se dibuje rotada.

ExplosionView

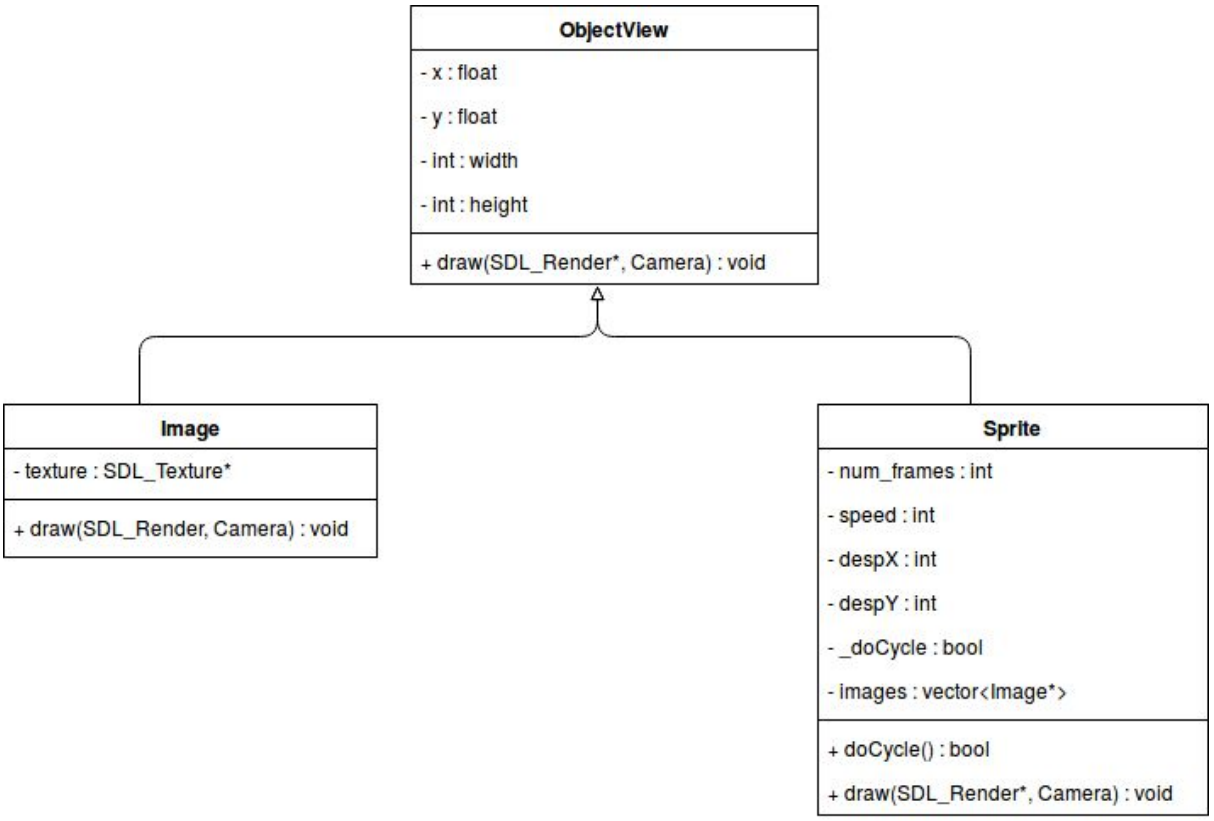
Interfaz que deben implementar las explosiones, que pueden tener 1 o mas sprites que las representen y dibujarlos en el orden que corresponda.

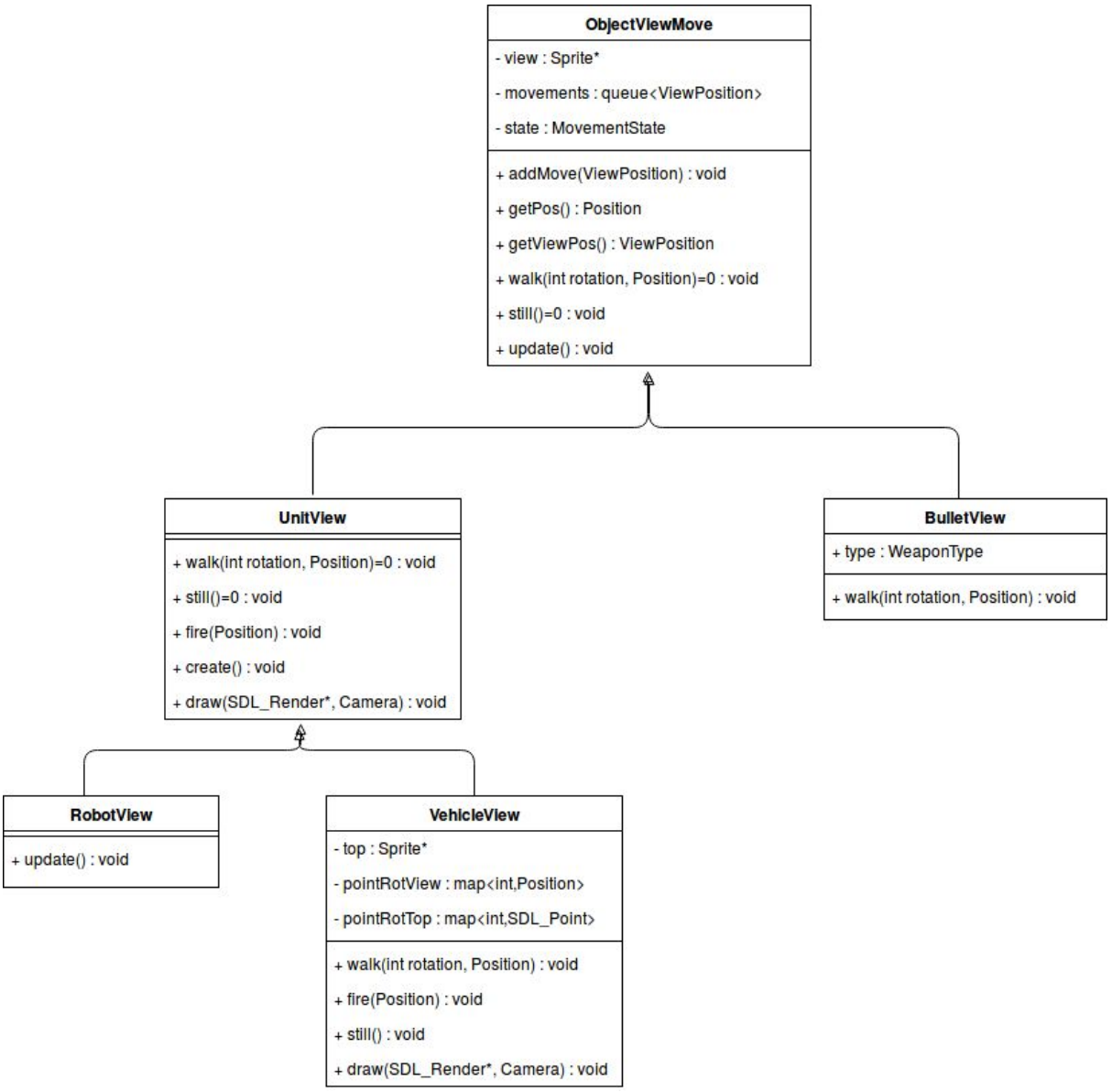
BuildingView

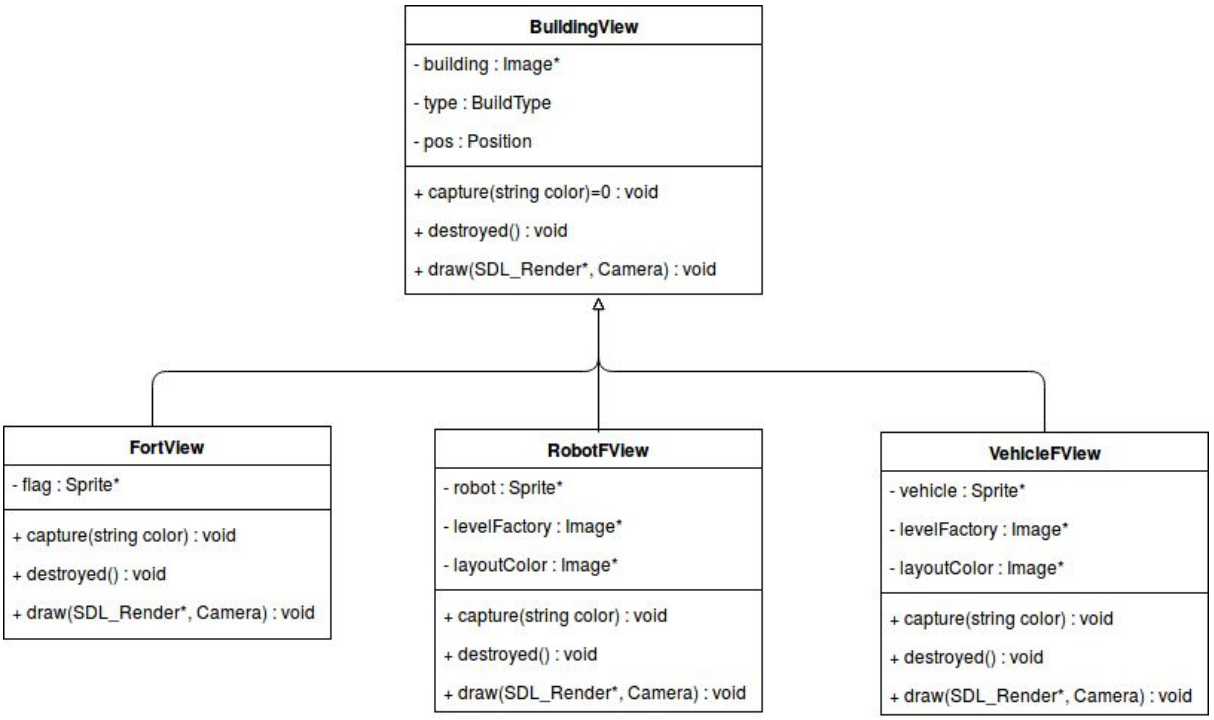
Esta clase modela a los edificios, que se componen de una imagen del edificio y distintos efectos de acuerdo al estado en que se encuentren (sin dueño, capturado o destruido).

`void capture(const std::string &color) :`
Este método agrega los efectos/animaciones (humo, luces, etc) de acuerdo al tipo de edificio que fue capturado.

Diagramas UML




ObjectViewMove
- view : Sprite*
- movements : queue<ViewPosition>
- state : MovementState
+ addMove(ViewPosition) : void
+ getPos() : Position
+ getViewPos() : ViewPosition
+ walk(int rotation, Position)=0 : void
+ still()=0 : void
+ update() : void
UnitView
+ walk(int rotation, Position)=0 : void
+ still()=0 : void
+ fire(Position) : void
+ create() : void
+ draw(SDL_Render*, Camera) : void
BulletView
+ type : WeaponType
+ walk(int rotation, Position) : void
RobotView
+ update() : void
VehicleView
- top : Sprite*
- pointRotView : map<int,Position>
- pointRotTop : map<int,SDL_Point>
+ walk(int rotation, Position) : void
+ fire(Position) : void
+ still() : void
+ draw(SDL_Render*, Camera) : void



Generador

Descripción general

El generador se compone de 3 clases sin contar la librería de manejo de archivos .json de Niels Lohmann[7] utilizada en este trabajo práctico. Tales clases son:

- Interface: constituye una pequeña y simple interfaz de usuario vía terminal que va solicitando al usuario el ingreso de los parámetros tales como tipo de mapa, porcentaje de ríos, cantidad de territorios, cantidad de jugadores, etc… con los que el generador va a construir el mapa. Además la interfaz valida los valores numéricos ingresados por el usuario debido a que valores muy extremos (e.g. muchos territorios para un mapa muy pequeño) pueden provocar la falla del programa, por lo que se establecen en la interfaz valores mínimos y máximos que garantizan el buen funcionamiento del generador.
- Generator: constituye el generador automático de mapas. Requiere que se le indique el ancho, el largo del mapa, la cantidad de territorios, de jugadores, de vehículos, el

[7] "GitHub - nlohmann/json: JSON for Modern C++...." https://github.com/nlohmann/json. Fecha de acceso 22 jun.. 2017.

tipo de mapa (invierno, pradera, con lava…), el nivel de producción de las fábricas, el porcentaje de ríos y de rocas.

- Store_map: se basa en la librería de manejo de archivos .json previamente mencionada para guardar el mapa generado por el generador en un archivo .json capaz de ser levantado por el servidor.

Funcionamiento del generador

Distribución de territorios

El generador de mapas automáticamente distribuirá los tiles[8] de modo tal de generar un mapa con territorios “lo más cuadrados posible”.
Sin embargo surgieron varias dificultades para lograr esto, las explicamos aquí y posteriormente detallamos las soluciones encontradas.

Determinar el espacio que ocupa un territorio:
El usuario ingresa el ancho y largo del mapa y la cantidad de territorios.
La dificultad de este paso radica en que en función del ancho, largo del mapa y la cantidad de territorios ingresados por el usuario, la mayoría de las veces la cantidad de tiles por territorio hace que los territorios no puedan tener todos una misma forma determinada. Sin embargo en aras de obtener mapas que sean lo más jugables posible, se estima necesario que un territorio deba ser “lo más cuadrado posible” siempre que pueda, osea siempre que los bordes del mapa lo permitan.

Por ejemplo si tuviéramos un mapa de 10x12 con 7 territorios, una buena dimensión para un territorio sería 4x4. Sin embargo el ancho del mapa (10) no es múltiplo de 4, por lo que el territorio 3 no logra ser un cuadrado.

6	6	6	6	7	7	7	7		
6	6	6	6	7	7	7	7		
6	6	6	6	7	7	7	7		
6	6	6	6	7	7	7	7		
5	5	5	5	4	4	4	4	3	3
5	5	5	5	4	4	4	4	3	3
5	5	5	5	4	4	4	4	3	3
5	5	5	5	4	4	4	4	3	3
1	1	1	1	2	2	2	2	3	3
1	1	1	1	2	2	2	2	3	3

[8] Conviene destacar de que es equivalente el término “tiles” del término “posiciones” por la forma en cómo elaboramos el juego; una posición en los diagramas que se muestran termina siendo un tile en el juego.

1	1	1	1	2	2	2	2	3	3
1	1	1	1	2	2	2	2	3	3

La forma mediante la cual este generador encara esta dificultad es haciendo que cuando un bloque (territorio) alcanza un borde prosigue arriba en sentido inverso tal y como se muestra en la figura.

Tiles remanentes:
Además, debido al hecho de que dada la dimensión de un territorio puede darse de que al final queden posiciones sin ser cubiertos por ningún territorio y que juntas esas posiciones no son suficientes como para constituir un territorio de nxm tiles, cabe destacar también que los tiles remanentes que completan el rectángulo general del mapa pasan a ser propiedad de el último territorio. En el ejemplo anterior, el territorio 7 se adjudica los tiles remanentes que quedaron sin cubrir, por lo que este único territorio termina siendo más grande.

Entonces en definitiva el algoritmo elaborado consta de los siguientes pasos:

Paso 1: en base al largo, al ancho del mapa y la cantidad de territorios ingresado por el usuario, se obtiene un valor entero con el número de tiles por territorio:

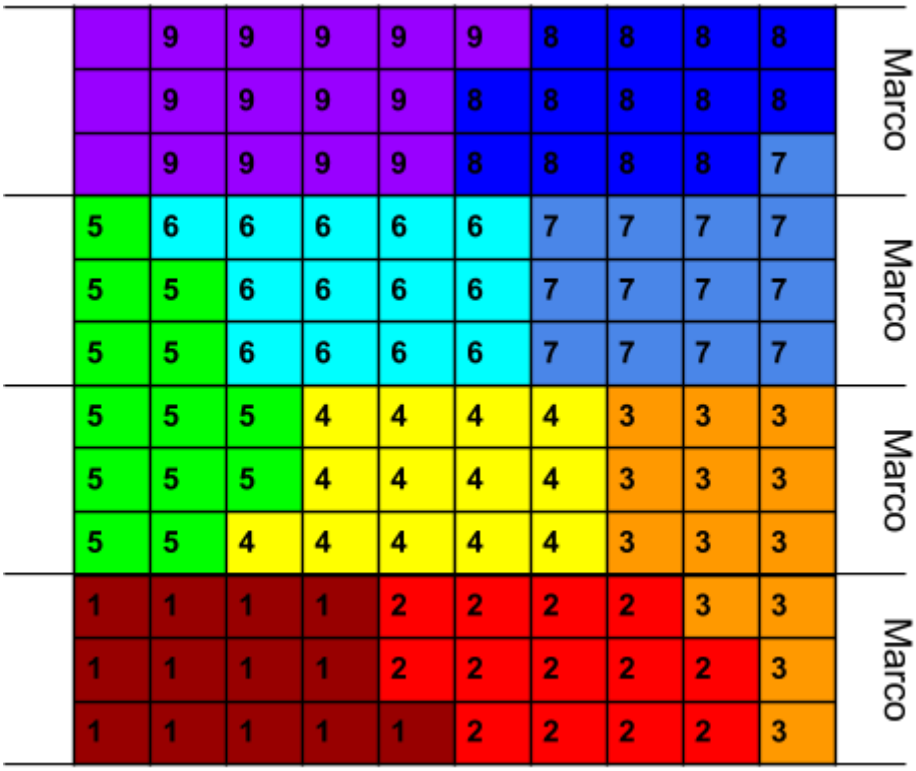
$$tiles\ por\ territorio = \left\lfloor \frac{cantidad\ de\ tiles\ en\ el\ mapa}{cantidad\ de\ territorios\ en\ el\ mapa} \right\rfloor$$

Por ejemplo si se ingresa que el mapa ha de ser de 10*12 y con 7 territorios, la cantidad de tiles por territorio será de 17 tiles.

Paso 2: Calcular el largo del territorio.

La forma mediante la cual el generador logra distribuir los tiles por territorios de forma que estos queden con una superficie más o menos rectangular es mediante la utilización de lo que denominamos marcos (frames). Un marco se define como un conjunto de filas sobre las que se acomoda un territorio. El número de filas que componen un marco se calcula como el mayor número entero que elevado al cuadrado es menor al número de tiles por territorio.

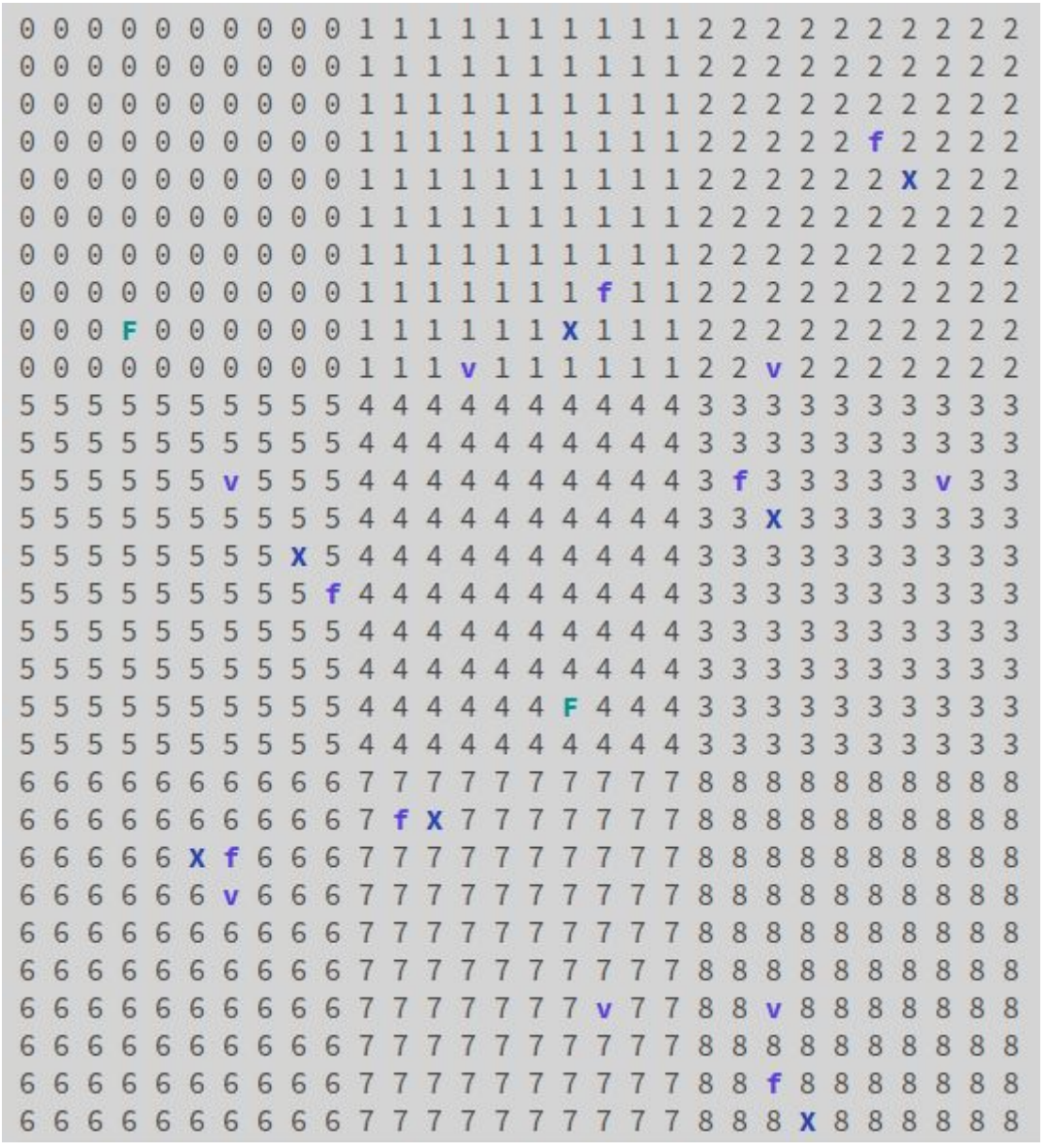
En el ejemplo siguiente disponemos de un mapa de 10x12 tiles con 9 territorios.
La cantidad de tiles por territorios calculado mediante la forma explicada mediante la anterior fórmula es de 13.
El mayor número entero que elevado al cuadrado da un valor inferior a 13 es 3 (3*3 = 9 < 13 y 4*4 = 16 > 13) por lo que vemos encuadrar los territorios en 3 filas:



Paso 3: rellenar el mapa con los territorios.

Vamos rellenando cada marco con territorios, serpenteando los territorios por el mapa completo tal y como se muestra en la figura; cada fila encuadrada en un marco se va rellenando de abajo hacia arriba hasta que ocurre que se alcanza la cantidad de tiles por territorio. En ese caso se prosigue con el territorio siguiente desde la posición en donde se dejó. Al alcanzar un borde se prosigue con el marco siguiente en sentido inverso.

Paso 4: Distribución de edificios y banderas:
En los territorios en los que no hay fuerte se distribuye al azar una fábrica de robots, una fábrica de vehículos y una bandera (adyacente a la fábrica de robots). Si hay fuerte, solo se dispone del fuerte dentro del territorio, sin bandera, siendo que emulando al juego original el territorio (destruido el fuerte o no) no puede ser capturado.
Se establece también un “padding” de uno, es decir que sobre los tiles que bordean el mapa no se ubicará ningún edificio y esto está diseñado para facilitar la forma en cómo aparecen unidades nuevas en el mapa.
Por ejemplo el siguiente mapa de 30x30 con 9 territorios:



Los fuertes son representados por las F mayúsculas, las banderas por las X, las fábricas de robots por las f minúsculas y las fábricas de vehículos por las v minúsculas.

Paso 5: Distribución de caminos:
Los caminos son aristas que unen vértices (i.e. fábricas y fuertes). Para esto se toma al azar un vértice, se lo conecta con su vértice más cercano, se lo descarta, se sigue con un siguiente vértice, se busca el vértice remanente más cercano, se lo conecta a él, se lo descarta, etc… hasta que no quedan más vértices. La distancia utilizada es la distancia Manhattan y los caminos son o verticales u horizontales.

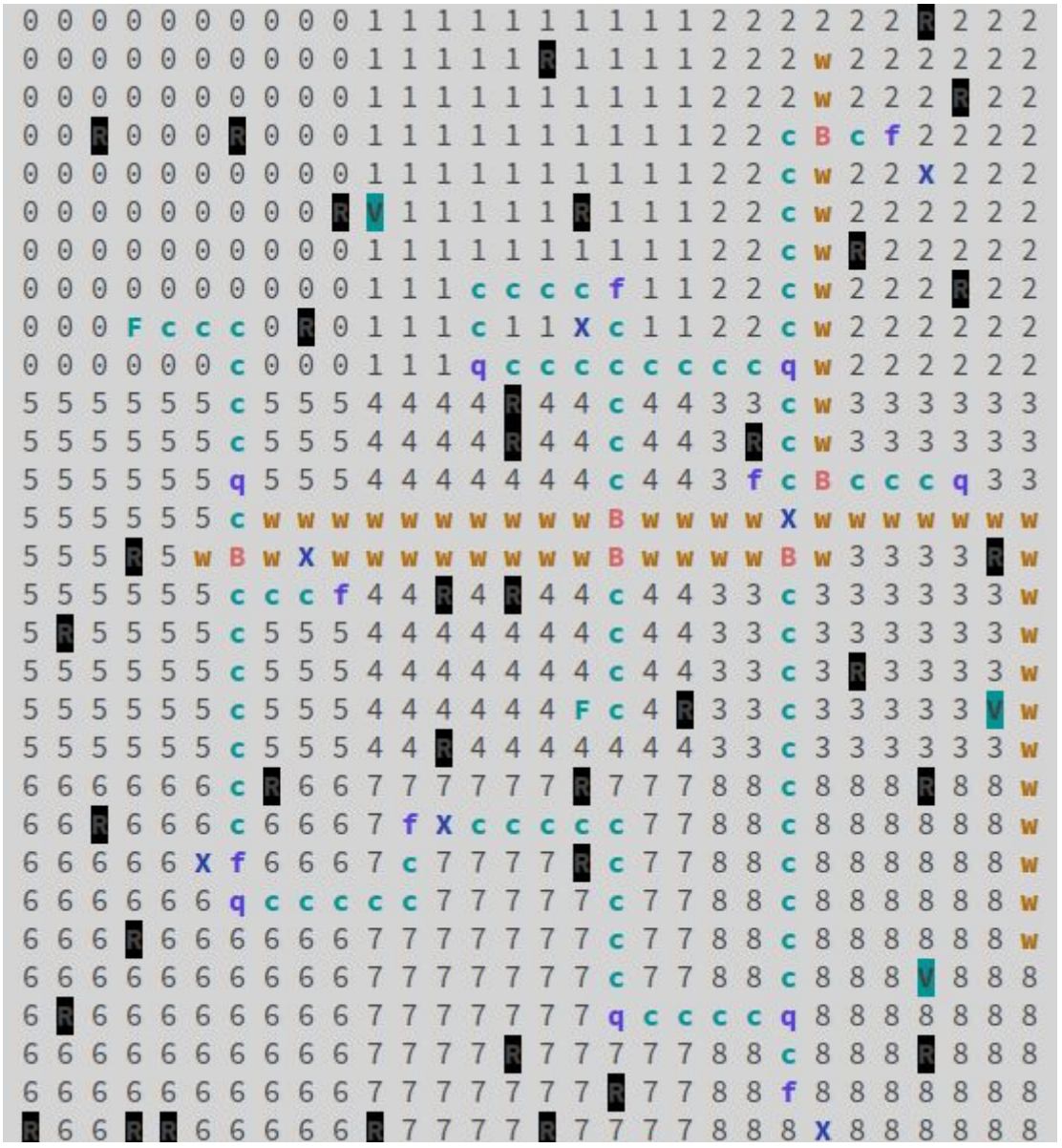
Paso 6: Distribución de terrenos (agua, pantano o lava)
Ha de ingresarse por parámetro el porcentaje de terrenos de agua/pantano y lava.
El ingreso de un valor de 100% no se corresponde con que 100% de los tiles serán agua/pantano y lava, sino el mapa será injugable. Por el contrario, el porcentaje ingresado

será un subporcentaje sobre un porcentaje preestablecido de 20% de tal modo que un 100% de lava, agua o pantano será equivalente como mucho a 20% del mapa.

Por el enunciado se requiere que se formen ríos, esto se logra trazando “caminos” de agua, pantano o lava, cuidando que los puntos de partida y fin de los caminos no se ubiquen en una misma fila o columna que contenga un vértice, esto evita la superposición de estos caminos con los caminos reales. A lo sumo habrá intersecciones perpendiculares que se convertirán en puentes (que dicho sea de paso esto permite que el mapa sea múltiplemente conexo con una única componente conexa, siendo los vértices los edificios del mapa).

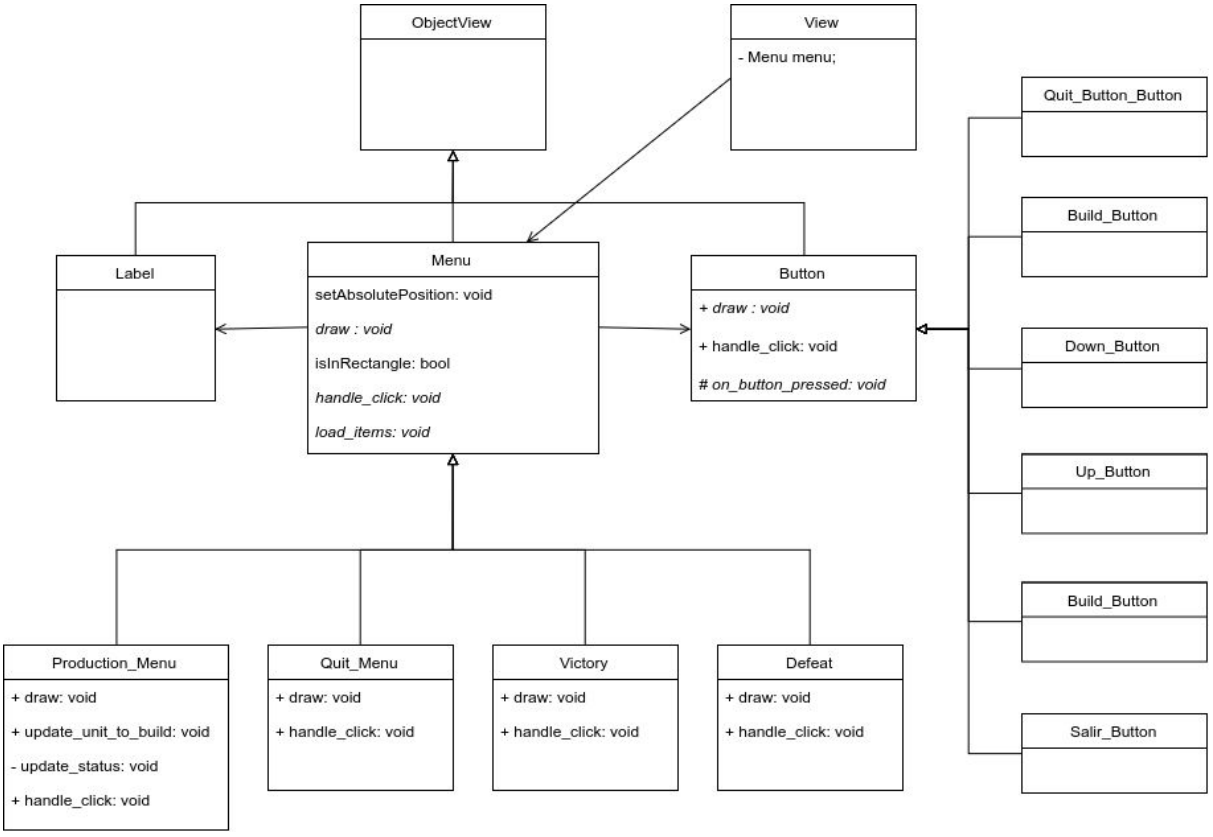
Paso 7: Distribución de rocas y vehículos:
Al igual que los terrenos de agua/pantano y lava, el usuario ingresa un porcentaje sobre un porcentaje preestablecido de 20%, salvo que la distribución de las rocas es aleatoria (sin formarse “murallas” de rocas).
La cantidad de vehículos se define mediante un número discreto que a su vez queda normalizado entre 0 y 30 por la interfaz y se dejó como que los vehículos a distribuir por el territorio son únicamente jeeps.

Ejemplo:
Mapa de 30 x 30, 9 territorios, 2 jugadores, 3 vehículos distribuidos, 20 % de ríos, 20% de rocas.
R : rocas
C: caminos
B : puentes (bridge)
W: agua (water)
V : vehículo
f : fábrica de robots
q: fábrica de vehículos
F: fuerte
X: bandera
Los números indican la pertenencia del tile al número de territorio.



Interfaz

Menús y botones:



Elementos externos utilizados:

- nlohmann json library (https://github.com/nlohmann/json)
- cereal (http://uscilab.github.io/cereal/)
- Dibujos de The Zod Engine (http://zod.sourceforge.net/)