

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

INGENIERÍA DE TELECOMUNICACIONES



"DESARROLLO DE UN JUEGO DE
SIMULACIÓN SOBRE LA
PLATAFORMA ANDROID"

TRABAJO FIN DE CARRERA

Enero -2017

AUTOR: Marco Antonio Chico Serrano

DIRECTOR/ES: Cesar Fernández Peris

UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

INGENIERÍA DE TELECOMUNICACIONES



"DESARROLLO DE UN JUEGO DE SIMULACIÓN SOBRE LA PLATAFORMA ANDROID"

TRABAJO FIN DE CARRERA

Enero -2017

AUTOR: Marco Antonio Chico Serrano

DIRECTOR/ES: Cesar Fernández Peris

VISTO BUENO Y CALIFICACIÓN DEL PROYECTO

Título proyecto:

Proyectante: _____

Director/es: _____

VºBº director/es del proyecto:

Fdo.:

Fdo.:

Lugar y fecha: _____

CALIFICACIÓN NUMÉRICA

MATRÍCULA DE HONOR ☐

Méritos justificativos en el caso de conceder Matrícula de Honor:

Conforme presidente:

Fdo.:

Conforme secretario:

Fdo.:

Conforme vocal:

Fdo.:

Lugar y fecha: _____



ÍNDICE

1.- Introducción	1
1.1 Visión general	1
1.2 Objetivos del proyecto	3
2.- Estado del arte	5
2.1 Smartphones y Android	5
2.2 Aplicaciones móviles y librería GDX	7
3.- Aplicación Dominations: Implementación general	12
3.1 Arquitectura de software.....	12
3.2 Lanzamiento e inicialización	14
3.3 Interfaz de usuario.....	32
3.4 Renderizado, capas y cámara	54
3.5 Inputs de usuario.....	59
3.6 Países, alianzas y territorios	70
3.7 Tecnologías.....	91
3.8 Edificios.....	105
3.9 Diplomacia	113
3.10 Tropas, batallas y conquistas	133
3.11 Eventos aleatorios.....	164
3.12 Ciclo del juego.....	167
3.13 Guardado y carga de partidas	175
3.14 Modo debug	182
4.- Aplicación Dominations: Inteligencia Artificial	186
4.1 Personalidades de IA	186
4.2 Gestión económica.....	198
4.3 Gestión defensiva.....	208
4.4 Gestión ofensiva.....	213
4.5 Gestión de tropas y tareas	220
4.6 Implementación del pathfinding	240
4.7 Futuro de la ia en dominations	251
5.- Rendimiento	252
5.1 Multithreading.....	252
5.2 Perfilado: Memoria, CPU, Red, GPU	260
6.- Capitalización y Marketing	263
7.- Conclusiones y planes de futuro	268

1.- INTRODUCCIÓN

1.1 VISIÓN GENERAL

Dominations es un juego de dominación geopolítica en tiempo real implementado en java para ser jugado desde smartphones con Android. El jugador tomará el control de un país cualquiera y lo gestionará, intentando alcanzar la victoria mediante la conquista del mundo, el resto de países son gestionados por IAs separadas con el mismo objetivo que el jugador, la victoria del juego mediante la conquista del resto de países.

Para lograr la victoria, cada país deberá planificar su economía y gestionar sus tropas, sin descuidar las relaciones diplomáticas con otros países, pues el comportamiento de las IAs pretende emular el comportamiento real de los distintos países del mundo, que responderán de forma realista frente a agresiones internacionales.

Si un país pierde todos sus territorios y tropas, será derrotado y perderá la capacidad de gestión.

El juego consta de los siguientes elementos diferenciados:

- Territorios: Cada país posee un numero de territorios, inicialmente cada país posee un territorio que coincide con el territorio real, pero esto se modificará si ocurren conquistas, si por ejemplo China conquistase Rusia, Rusia dejaría de existir como país independiente y su IA dejaría de ejecutarse y China (país) poseería dos territorios: Rusia y China.
- Tiempo del juego: El juego transcurre en tiempo real, aunque acelerado, de forma que cada 10 segundos en el juego transcurre un día (esta velocidad puede ser modificada por el usuario desde la interfaz) y cada día se produce un update del juego.
- Dinero: Cada país posee una cierta cantidad de dinero y cada territorio lo produce diariamente, dicha producción coincide con el PIB diario real del país real al que pertenece el territorio. Inicialmente cada país posee ahorrado un dinero equivalente a 10 días de producción.
- Tecnologías: Investigaciones tecnológicas que puede desarrollar cada país a cambio de invertir cierta cantidad de tiempo y dinero, aportarán beneficios permanentes que afectarán al país que las desarrolle.
- Edificios: Cada territorio posee cuatro edificios que puede construir y varios niveles para cada uno, las bonificaciones de los edificios son permanentes, pero sólo afectan al territorio donde se construyen.

- Tropas: Cada territorio puede producir tropas de hasta cinco tipos, pudiendo ser infantería, artillería y aviación. Las tropas tienen distintos valores ofensivos y defensivos, por lo que cada tropa será eficaz para determinadas tareas. Las tropas pueden moverse libremente por el mapa e interactuarán con el país sobre el que estén, defendiendo países aliados y atacando territorios hostiles.
- Batallas y daños: Cuando varios escuadrones de tropas se encuentran en un mismo territorio, se producirá un combate progresivo hasta que un bando venza. Durante la batalla se producirán daños en el territorio que reducirán su capacidad productiva, siendo las tropas áreas especialmente eficaces en ello. Los daños de los territorios se repararán progresivamente con el tiempo.
- Lealtad y conquista: Si unas tropas de un país se encuentran en un territorio enemigo, y el país defensor no tiene tropas que puedan defenderlo, la lealtad del territorio comenzará a bajar, si llega a cero, el país será conquistado por el atacante. Las tropas defensivas o aliadas provocarán el efecto contrario sobre el territorio: provocarán un aumento en la lealtad.
- Diplomacia: Cada país posee una relación diplomática con cada uno de los demás países del mundo, lo cual es representado por un valor que va de 0 a 100, siendo 0 hostilidad máxima y 100 alianza máxima. Dicho parámetro es empleado por las IAs para las decisiones defensivas y ofensivas además de otros parámetros del juego que se describirán en adelante.

1.2 OBJETIVOS DEL PROYECTO

El objetivo del proyecto es desarrollar una aplicación de entretenimiento capitalizable y que posea un cierto realismo a la hora de simular la interacción geopolítica de distintos países del mundo real a partir de la descripción de un modelo preciso que sea capaz de describir tanto sus relaciones como el estado inicial de los mismos.

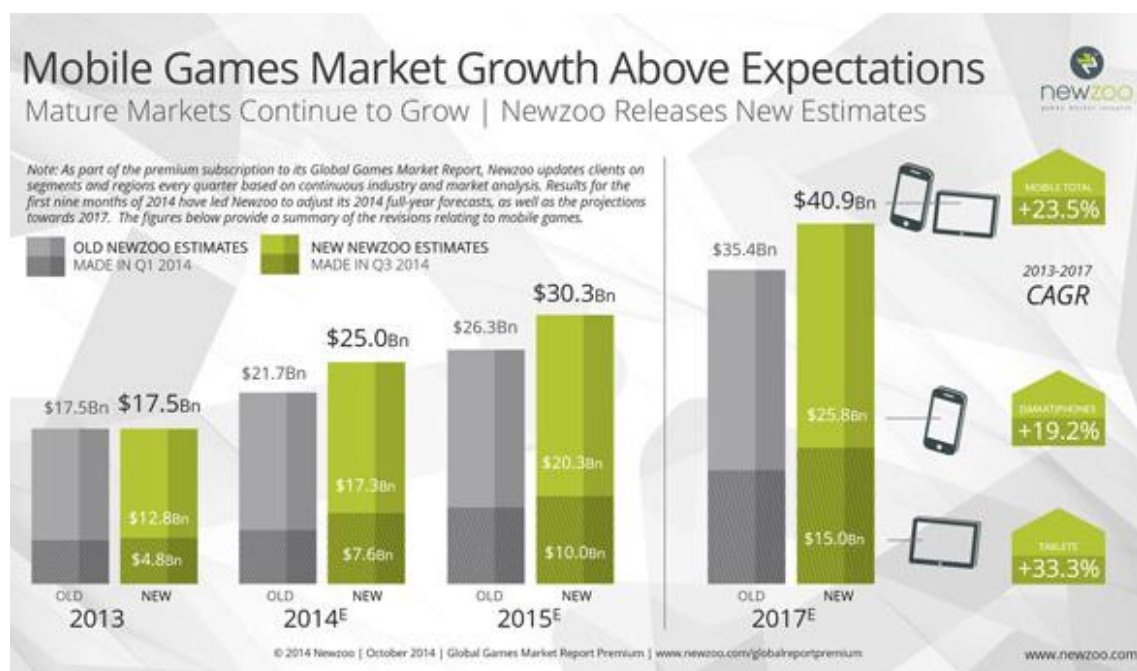
La aplicación intenta generar un entorno que se presente al usuario como un reto intelectual, obligándole a enfrentarse a múltiples IAs tomando decisiones complejas y delicadas para alcanzar el objetivo, convirtiéndose en un desafío para el jugador.

Esta lista muestra los 10 juegos de Android que más ingresos han generado:

1. Clash of Clans	Supercell	\$1.345M
2. Game of War: Fire Age	Machine Zone	\$799M
3. Puzzle & Dragons	Gungho Online Ent. Inc	\$729M
4. Candy Crush Saga	King Digital	\$682M
5. MonsterStrike	Mixi	\$674M
6. Candy Crush Soda Saga	King Digital	\$518M
7. Fantasy Westward Journey	NetEase	\$451M
8. Colopl Rune Story	COLOPL NI Inc	\$356M
9. Disney Tsum	LINE Corp	\$326M
10. Boom Beach	Supercell	\$297M

Si contextualizamos las cifras anteriores podemos apreciar que Clash of Clans es el segundo juego (de cualquier plataforma) que más beneficios ha facturado en 2015, sólo superado por League of Legends con \$1.620M y duplicando a productos AAA como World Of Warcraft con sus \$810M. La facturación de los diez juegos anteriores casi alcanza la de los 10 juegos más rentables para PC.

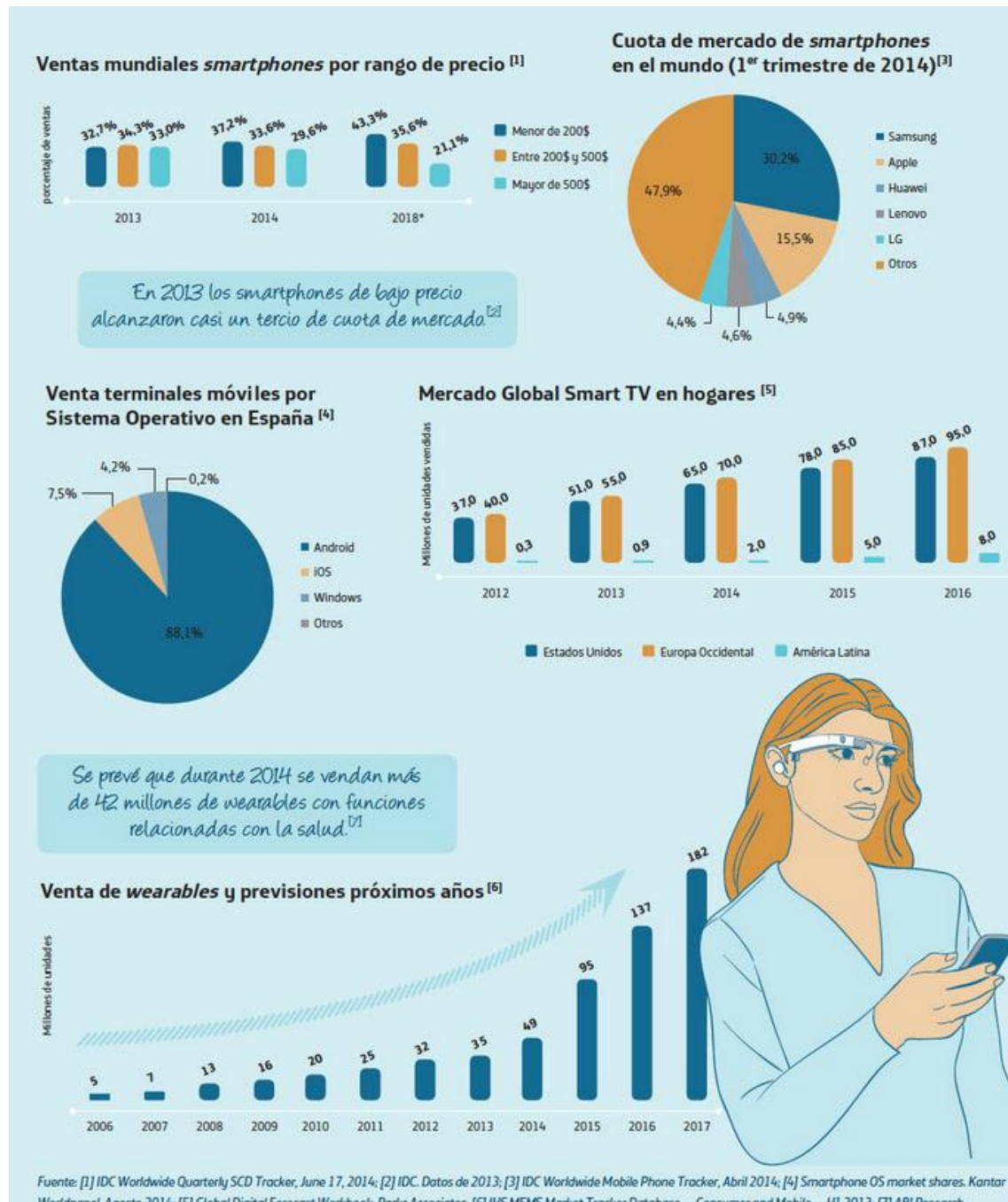
Con el auge de los smartphones y con los datos mostrados queda bastante patente que el entretenimiento mediante aplicaciones móviles representa una enorme oportunidad de mercado que puede ser fácilmente explotable con el marketing suficiente para captar la atención de los consumidores y un gameplay atractivo para mantenerlos.



2.- ESTADO DEL ARTE

2.1 SMARTPHONES Y ANDROID

El auge de los smartphones en la última década es absolutamente incuestionable, no obstante, vamos a contrastarlo con datos extraídos del informe SIE14 realizado por telefónica.



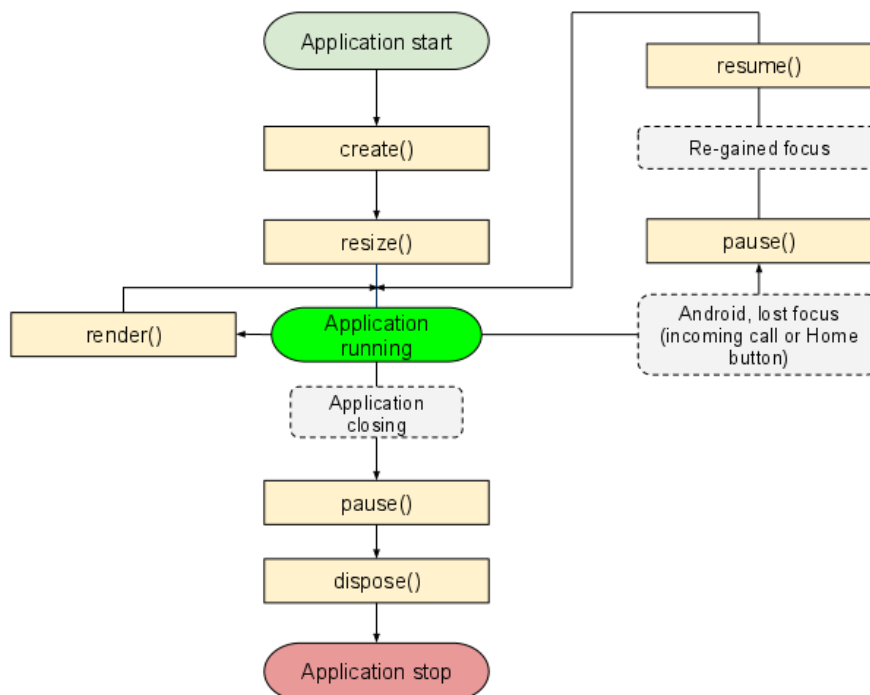
Los datos muestran que cada vez es más difícil encontrar personas que no tengan smartphone en los entornos industrializados, con un predominio casi absoluto del sistema operativo Android. Se espera que además las mejoras tecnológicas constantes en computación descritas por la ley de Moore sumados a los constantes proveedores de smartphones que quieren entrar en el negocio provocarán que cada vez más personas, incluso del tercer mundo, puedan disfrutar de las asombrosas capacidades de estos dispositivos, convirtiéndose en potenciales clientes para aplicaciones como la desarrollada en este proyecto.

El 88% de los smartphones utilizan Android, lo cual ya de por sí es suficiente justificación para implementar la aplicación para este SO, otra ventaja no tan intuitiva es que se implementa en JAVA, lo cual hace la aplicación fácilmente portable para otras plataformas y compatible con la librería gráfica que soporta el juego, lo cual es un plan de futuro que se detallará en adelante.

2.2 APLICACIONES MÓVILES Y LIBRERÍA GDX

Libgdx es un framework multiplataforma de código abierto que da soporte a la creación de videojuegos en java, siendo compatible con todas las plataformas capaces de ejecutar java y facilitando el desarrollo de las aplicaciones. Por este motivo la mayoría de métodos y abstracciones tienen también compatibilidad con PC, con soporte de inputs de ratón y demás.

Ciclo de vida



La función **create** es llamada al lanzar la aplicación y es el punto en el que se recomienda instanciar todos los elementos necesarios para la aplicación, así como realizar el cacheado de las imágenes y recursos de la misma.

La función **resize** es llamada cuando la ventana cambia de tamaño, cosa que en principio nunca ocurre si la librería se emplea en aplicaciones móviles.

El bucle principal se ejecuta en la función **render**, que es llamada desde un hilo OpenGL dedicado explícitamente a dibujar los elementos gráficos.

La función **pause** se llama cuando el juego es minimizado (por ejemplo, mediante el botón home de la mayoría de móviles) y al volver a lanzarla se llama a la función **resume**.

Cuando el juego se cierra definitivamente, se llama a la función **dispose**, siendo este un punto interesante para guardar el estado del juego si eso fuera relevante en el diseño del mismo, como en este caso.

Componente gráfico y animado

La librería implementa métodos de renderizado empleando openGL y una serie de objetos abstractos que facilitan la implementación de elementos gráficos:

- Sprite: Imagen que posee características de tamaño y posición.
- Actor: Objeto que se caracteriza por poseer un sprite y una serie de acciones.
- Stages: Representan capas gráficas donde se almacenarán distintos actores que tendrán una cámara común y que estarán dibujados a la misma profundidad.
- Camara: Objeto cámara que determina la parte de la escena que se muestra en pantalla, pudiendo desplazarse o hacer zoom por inputs de usuario.
- Input Listener: Se asignan a las distintas stages por prioridad, para que las stages respondan a dichos listener, como por ejemplo arrastrar y soltar o tocar la pantalla del Smartphone.
- Animaciones: Libgdx proporciona también herramientas para dibujar actores con sprites animados en base a una lista de fotogramas (sprites) que dibujados secuencialmente se muestran como una animación.

Componentes adicionales

El framework también da soporte a gestiones E/S como lectura y escritura de ficheros, reproducción de audio y sonidos y algunos algoritmos de IA entre los cuales destacaremos el algoritmo de pathing A* empleado en el proyecto. La librería además implementa un motor físico para gestionar colisiones que no se ha empleado en el desarrollo de esta aplicación.

Las siguientes tablas muestran un resumen de las características de esta librería extraídas directamente de su wiki en la siguiente dirección web: <http://libgdx.badlogicgames.com/features.html>

Cross-Platform

A single API to target:

- Windows
- Linux
- Mac OS X
- Android (2.2+)
- BlackBerry
- iOS
- Java Applet (requires JVM to be installed)
- Javascript/WebGL (Chrome, Safari, Opera, Firefox, IE via **Google Chrome Frame**)

3rd Party Support

Libgdx can be integrated with many 3rd party tools. We love:

- Spine - 2D Skeletal Animation
- Nextpeer - Mobile multiplayer made easy
- Saikoa - makers of ProGuard and DexGuard

Audio

- **Streaming music** and **sound effect** playback for WAV, MP3 and OGG
- Direct access to audio device for **PCM sample playback** and **recording** (unsupported in Javascript backend)

Input Handling

- Abstractions for **mouse and touch-screen, keyboard, accelerometer** and **compass**
- **Gesture detector**, detects taps, panning, flinging and pinch zooming

Math & Physics

- **Matrix, vector** and **quaternion** classes. Matrix and vector operations are accelerated via native C code where possible
- **Bounding shapes and volumes**
- **Frustum class**, for picking and culling
- **Catmull-Rom splines**
- **Common interpolators**
- **Concave polygon triangulator**
- **Intersection and overlap testing**
- 2D physics: JNI wrapper for **Box2D** physics. So awesome, other **engines** use it as well
- 3D physics: **JNI Wrapper** for **bullet physics**

File I/O & Storage

- **File system abstraction for all platforms**
- Read-only file system emulation for Javascript backend
- Binary file support for Javascript backend
- **Preferences** for lightweight setting storage

Utilities

- **Custom collections, with primitive support**
- Json **writer** and **reader**, with POJO (de-)serialization support
- Xml **writer** and **reader**

Tools

- **2D Particle editor**
- **3D Particle editor**
- **Texture packer**
- **Bitmap font generator**

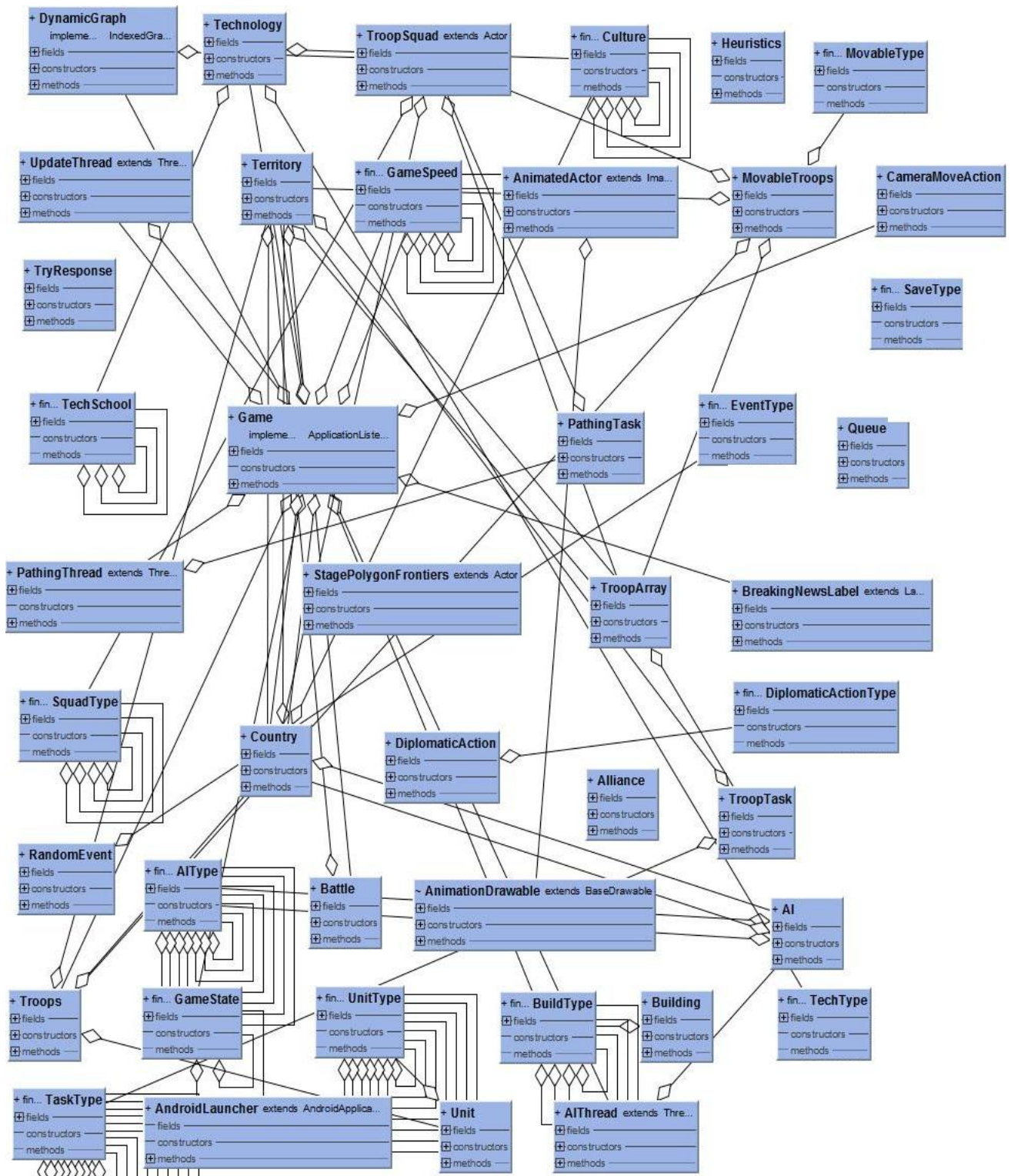
Graphics

- Rendering through OpenGL ES 2.0 on all platforms
- **Custom OpenGL ES 2.0 bindings** for Android 2.0 and above
- Low-Level OpenGL helpers:
 - **Vertex arrays** and **vertex buffer objects**
 - **Meshes**
 - **Textures**
 - **Framebuffer objects** (GL ES 2.0 only)
 - **Shaders**, integrating easily with meshes
 - **Immediate mode rendering emulation**
 - **Simple shape rendering**
 - Automatic software or hardware mipmap generation
 - **ETC1** support (not available in Javascript backend)
 - Automatic handling of **OpenGL ES context loss**. Restores all textures, shaders and other OpenGL resources
- High-level 2D APIs:
 - Custom CPU side **bitmap manipulation library**
 - **Orthographic camera**
 - High-performance **sprite batching** and **caching**
 - **Texture atlases**, with whitespace stripping support. Either generated **offline** or **online**
 - **Bitmap fonts** (does not support complex scripts like Arabic or Chinese). Either generated **offline** or **loaded from TTF files** (unsupported in Javascript backend)
 - **2D Particle system**
 - **TMX tile map** support
 - **2D scene-graph API**
 - **2D UI library**, based on scene-graph API, fully skinnable
- High-Level 3D APIs:
 - **Perspective camera**
 - **Decal batching**, for 3D billboards or particle systems
 - Basic loaders for **Wavefront OBJ** and **MD5**
 - 3D rendering API with materials and lighting system and support for loading FBX models via **fbx-conv**

3.- APLICACIÓN DOMINATIONS: IMPLEMENTACIÓN GENERAL

3.1 ARQUITECTURA DE SOFTWARE

La siguiente imagen muestra un diagrama UML de las clases de la aplicación:



Del diagrama anterior las clases más importantes son las siguientes:

- AndroidLauncher: Clase que sirve de interfaz entre el lanzamiento de la aplicación por el SO de Android y la carga de la clase GDX que es game, evita tener que trabajar con Activities y demás estructuras de Android, este launcher puede ser particularizado para cada sistema operativo, de modo que tendrá una estructura diferente si se quiere lanzar la aplicación en el PC, también se definen aquí las configuraciones internas de la aplicación.
- Game: Clase principal del juego, contiene los métodos de inicialización y el update del juego, todo el soporte gráfico y de inputs, así como los métodos más importantes del juego.
- Country: Clase que contiene todos los métodos y variables relacionados con países, como dinero, valores diplomáticos, territorios que posee...
- Territory: Clase que contiene los métodos relacionados con los territorios, el polígono del mapa que le corresponde, la producción, lealtad, tropas...
- Building y BuildType: Clase relacionada con los tipos de edificios del juego y sus métodos.
- Technology, TechSchool y TechType: Clases relacionadas con las tecnologías que los países pueden investigar.
- AI y AIType: Clases que se encarga de la gestión de la IA de cada país, según sus tipos y demás.
- DynamicGraph, Heuristics y PathingTask: Clases que gestionan los cálculos de pathing de tropas en el mapa, empleado por las IAs para calcular rutas óptimas para alcanzar sus objetivos.
- TroopArray, Troops, TroopTask, Unit y UnitType: Clases relacionadas con las tropas y gestión de las mismas.
- TroopSquad, SquadType: Clases que gestionan la agrupación de tropas para moverlas por el mapa para atacar o defender objetivos.
- Battle: Clase que gestiona las batallas, efectúa los cálculos de bajas y demás.
- RandomEvent y EventType: Clases que gestionan los eventos aleatorios que ocurren en el juego (guerras, disputas diplomáticas o desastres meteorológicos)
- DiplomaticAction y DiplomaticActionType: Clases que gestionan los eventos diplomáticos voluntarios, como mejorar relaciones entre dos países, declarar guerras o solicitar la paz.
- AIThread, PathingThread y UpdateThread: Clases que extienden threads y que garantizan que las tareas concretas a las que se asigna cada clase se ejecuten en threads paralelos, explotando el multicore de los smartphones y realizando tareas en paralelo.

3.2 LANZAMIENTO E INICIALIZACIÓN

Aplicación

El nombre DomiNations ha sido elegido por tratarse de un juego de palabras: Domination + Nations. Dicha combinación representa completamente al juego: distintas naciones interactuando entre sí por la dominación del mundo. El nombre del juego que se puede ver en el splash combina estos dos conceptos adecuadamente con una combinación gradiente de colores haciendo especial hincapié en la N central, que representa la unificación de ambos conceptos. La posición del texto además recuerda al movimiento de una bandera con el viento:



El icono del juego está en consonancia con esta explicación, representando un mundo en guerra bajo la letra N de capital peso por lo expuesto:



La ruta del icono está establecida por el androidManifest, fichero del que hablaremos más adelante. Se guardan varios ficheros de imagen de distintas resoluciones del icono, el SO se encargará de mostrar la adecuada para el dispositivo del usuario.

Lógica de inicialización

La ruta lógica al lanzar la aplicación es la siguiente:

```
// Clase a la que se llama cuando arranca el juego (main) y esta
// inicializa la clase principal (Game) con la configuracion obtenida del
// AndroidManifest.xml que es parseada por
// AndroidApplicationConfiguration()
public class AndroidLauncher extends AndroidApplication {
    @Override
    protected void onCreate (Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        AndroidApplicationConfiguration config = new
            AndroidApplicationConfiguration();
        config.useAccelerometer = false;
        config.useCompass = false;
        config.numSamples = 5; // Numero de muestras del multimuestreo
            CSAA/MSAA
        config.touchSleepTime = 16; // Tiempo en ms entre inputs
            (ignorara si inferior)
        initialize(new Game(), config);
    }
}
```

Y la información del manifest:

```
<application
    android:allowBackup="true"
    android:icon="@drawable/ic_launcher"
    android:label="@string/app_name"
    android:theme="@style/GdxTheme" >
    <activity
        android:name="gameCore.AndroidLauncher"
        android:label="@string/app_name"
        android:screenOrientation="landscape"
        android:configChanges="keyboard|
            keyboardHidden|orientation|screenSize">
        <intent-filter>
            <action android:name="android.intent.action.MAIN" />
            <category android:name="android.intent.category.LAUNCHER" />
        </intent-filter>
    </activity>
</application>
```

La clase `AndroidApplicationConfiguration` que recibe el método `initialize` establece también el valor por defecto de determinadas variables. En el siguiente fragmento de código podemos apreciar el valor que se le da a dichas variables, esto es relevante ya que no hemos redefinido sus valores en el manifest:

```

/** Class defining the configuration of an {@link AndroidApplication}.
Allows you to disable the use of the accelerometer to save
* battery among other things.
* @author mzechner */
public class AndroidApplicationConfiguration
{
    /** number of bits per color channel */
    public int r = 5, g = 6, b = 5, a = 0;

    /** number of bits for depth and stencil buffer */
    public int depth = 16, stencil = 0;

    /** number of samples for CSAA/MSAA, 2 is a good value */
    public int numSamples = 0;

    /** whether to use the accelerometer. default: true */
    public boolean useAccelerometer = true;

    /** whether to use the compass. default: true */
    public boolean useCompass = true;

    /** the time in milliseconds to sleep after each event in the touch
        handler, set this to 16ms to get rid of touch flooding on
        * pre Android 2.0 devices. default: 0 */
    public int touchSleepTime = 0;

    /** whether to keep the screen on and at full brightness or not
        while running the application. default: false */
    public boolean useWakelock = false;

    /** hide status bar buttons on Android 4.x and higher (API 14+).
        Doesn't work if "android:targetSdkVersion" less 11 or if API
        * less 14. default: false */
    public boolean hideStatusBar = false;

    /** whether to disable Android audio support. default: false */
    public boolean disableAudio = false;

    /** the maximum number of {@link Sound} instances that can be
        played simultaneously, sets the corresponding {@link SoundPool}
        * constructor argument. */
    public int maxSimultaneousSounds = 16;

    /** the {@link ResolutionStrategy}. default: {@link
        FillResolutionStrategy} */
    public ResolutionStrategy = new FillResolutionStrategy();

    /** if the app is a livewallpaper, whether it should get full touch
        events */
    public boolean getTouchEventsForLiveWallpaper = false;

    /** set this to true to enable Android 4.4 KitKat's 'Immersive
        mode' */
    public boolean useImmersiveMode = false;
}

```

```

/** whether to use {@link
    com.badlogic.gdx.backends.android.surfaceview.GLSurfaceView20API18} in place of the classic
    * {@link
    com.badlogic.gdx.backends.android.surfaceview.GLSurfaceView20}
    on Android API 10 and lower.
    * In case this is true {@link
    com.badlogic.gdx.backends.android.surfaceview.GLSurfaceView20API18} will be used.
    * This implementation properly supports attach to and detach
    from window. default: false */
public boolean useGLSurfaceView20API18 = false;
}

```

La clase Game que se inicializa es la principal e implementa la interfaz del framework GDX ApplicationListener que define los siguientes métodos que se llaman ante eventos definidos por el framework:

```

// Called when the application is first created.
public void create ();
// Called when the application is resized.
public void resize (int width, int height);
// Called when the application should render itself.
public void render ();
// Called when the application is paused, usually when it's not active
or visible on
// screen. An Application is also paused before it is destroyed.
public void pause ();
// Called when the application is resumed from a paused state
public void resume ();
// Called when the application is destroyed. Preceded by a call to
pause()
public void dispose ();

```

Por lo que queda patente en ese diseño que nuestra inicialización se realiza en el método create, que implementamos en Game del siguiente modo:

```

@Override
public void create()
{
    // Comprobamos variables que fuerzan un reinicio del save y nodos
    (para debug)
    checkSavedFiles();

    // Inicializamos elementos basicos que se requeriran (fondo de
    pantalla, musica...)
    initializeBasicElements();

    // Inicializamos menu principal
    initializeMainMenu();
}

```

A continuación, copio algunos fragmentos de dichas inicializaciones:

```
private void initializeBasicElements()
{
    // Recursos que deben ser inicializados aqui para usarlos incluso
    // antes de cargar el juego
    m_buttonClick_sound =
        Gdx.audio.newSound(Gdx.files.internal("Sounds/button2Click.mp3"
            ));

    // Ancho y alto
    windowW = Gdx.graphics.getWidth();
    windowH = Gdx.graphics.getHeight();
    (...)

    // Fondo de menu
    m_mainMenuBackground = new
        Texture(Gdx.files.internal("mainMenu.png"));

    // Stage
    stageMainMenu = new Stage();

    // Boton de atras no kileara la app
    Gdx.input.setCatchBackKey(true);

    // Atlas
    atlas = new TextureAtlas(Gdx.files.internal("Atlas.pack"));

    // Estado en menu
    m_gameState = GameState.STATE_MENU;
}
```

En este método se inicializa un elemento llamado TextureAtlas, el atlas es una estructura que permite cargar todas las imágenes del juego juntas en un único fichero de imagen externo y cargar cada subimagen a partir de una definición de nombre, posición y tamaño, aquí el ejemplo de una imagen atlas y su fichero de definición asociado:



La generación de imágenes atlas la realiza automáticamente una herramienta programada en java que proporciona el framework GDX y que combina de forma óptima las imágenes del juego, generando un fichero externo de definición de posiciones.

La principal ventaja de emplear este sistema es que el sistema de renderizado de GDX está diseñado para iniciar una rutina de dibujado que se abre y se cierra para cada imagen cuando tiene que dibujarlas, de modo que incluso aunque la imagen esté cacheada en memoria evitando hacer una llamada I/O, presenta un consumo de CPU que en una función como la de renderizado que se ejecuta con una frecuencia de 60Hz puede afectar al rendimiento.

```

public void initializeMainMenu()
{
    (...)
    // Musica
    m_mainMenuMusic =
        Gdx.audio.newMusic(Gdx.files.internal("Sounds/mainMenuMusic.mp3
            "));
    playMusic(m_mainMenuMusic, 0.60f, true);
    (...)

    // Botones
    TextButton startButton = new TextButton("New Game",
        m_styleMainMenu);
    startButton.addListener(new InputListener() {
        @Override
        public boolean touchDown(InputEvent event, float x, float y,
            int pointer, int button) {
            // BOTON DE NUEVA PARTIDA

            // Sonido
            playSound(m_buttonClick sound, 0.5f);

            // Si hay partida guardada, confirmar si se quiere empezar
            nueva partida
            if (thereIsLoadableGame())
                ensureNewGameDialog("Are you sure you want to start a
                    new game? Any saved game will be lost.");
            else // Si no hay partida guardada, iniciar directamente.
                prepareToInitializeGame(false);

            return true;
        }
    });

    (...)

    // Tabla del menu principal.
    Table mainMenu = new Table();
    mainMenu.setFillParent(true);
    mainMenu.center().center();
    mainMenu.setBackground(new NinePatchDrawable(new
        NinePatch(m_mainMenuBackground, 10, 10, 10, 10)));
    mainMenu.add(startButton).size(reffLabelStart.getWidth() * 2f,
        reffLabelStart.getHeight() * 2f);
    mainMenu.row();
    mainMenu.add(loadButton).size(reffLabelStart.getWidth() * 2f,
        reffLabelStart.getHeight() * 2f);
    mainMenu.row();
    mainMenu.add(optButton).size(reffLabelStart.getWidth() * 2f,
        reffLabelStart.getHeight() * 2f);
    stageMainMenu.addActor(mainMenu);

    // Es el lector de inputs
    Gdx.input.setInputProcessor(stageMainMenu);
}

```

Desde estos métodos hemos instanciado todos los elementos que necesita la aplicación para generar el menú y hemos almacenado los componentes del entorno 2D en la variable **stageMainMenu**, que contiene actores que podrán ser dibujados desde el thread de renderizado, que llama recursivamente al método render ejecutando el siguiente fragmento:

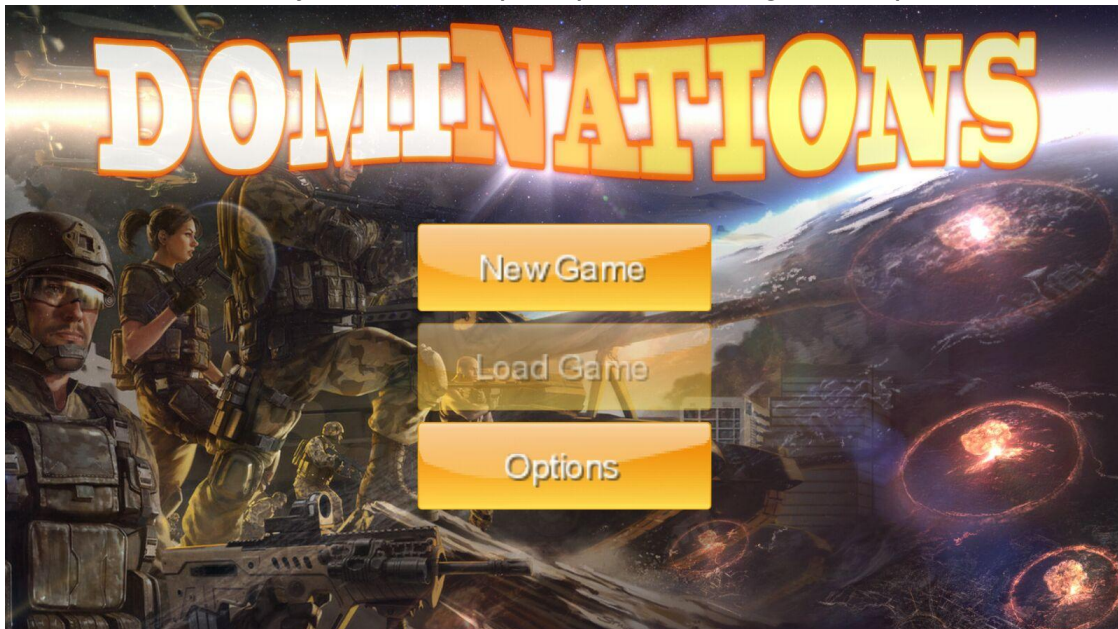
```
@Override
public void render()
{
    // Color de fondo Negro
    Gdx.gl.glClearColor(0, 0, 0, 0);

    // Limpiamos pantalla
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    switch (m_gameState)
    {
        case STATE_MENU:
            // Dibujado de Mapa
            stageMainMenu.act(Gdx.graphics.getDeltaTime());
            stageMainMenu.draw();
            break;
        // Estamos en el juego, update a la interfaz del juego
        case STATE_GAME:
            (...)
    }
}
```

(Obviamente en el estado STATE_GAME es donde se dibujará todo el juego cuando la aplicación se encuentre en esa fase)

Tras haber sido dibujado, el menú principal tiene la siguiente apariencia:



- New Game: Iniciamos una nueva partida, si existía una partida guardada previa, se avisará al usuario de que dicha partida será reemplazada si decide iniciar un nuevo juego, si confirma, se iniciará un nuevo juego y reemplazará al anterior guardado si lo había.

- Load Game: Si existía un juego guardado previo, se cargará y se podrá seguir jugando.
- Options: Opciones del juego, volumen de música y demás, no está implementado.

Una vez que hayamos decidido iniciar una nueva partida, se cargará una pantalla de carga para que permanezca de ese modo durante la inicialización del juego y se llamará al siguiente método:

```
public void initializeGame(boolean loadGame)
{
    // Initialize game
    initializeGameVariables();
    initializeSounds();
    initializeTerritories();
    initializeCountries();
    initializeDiplomacy();
    initializeCountriesFusion();
    initializeTechnologies();
    initializeDiplomaticActions();
    initializeBuildings();
    initializeUnits();
    initializeUI();
    initializeInputProcessors();
    initializeButtons();
    initializeNodePoints();
    initializeMovementMaps();
    initializeRandomEvents();
    initializeThreads();
    saveRelevantGameVariables();

    // Cargamos juego previo si ha de hacerse
    if (loadGame)
        loadGame();
    else // Si no tiene que cargar, mostrar start y borrar partida
        previa
    {
        clearSavedGame();
        showDialog("Welcome to DomiNations! select the country you
                    want to play with.");
    }

    // Musica del juego
    m_mainMenuMusic.stop();
    playMusic(m_backMusic, 0.20f, true);

    // Estado juego
    m_gameState = GameState.STATE_GAME;

    // Borramos menu inicial, ya no nos hara falta
    stageMainMenu.clear();

    // Limpieza del colector forzada
    System.gc();
}
```

InitializeGameVariables inicializa variables relevantes al propio juego, como los días a los que corresponde cada velocidad o la velocidad de regeneración de daño o lealtad en territorios, son parámetros importantes a la hora de balancear correctamente el gameplay:

```
public void initializeGameVariables()
{
    // Variables del juego
    m_debugButton = false;
    m_debugMode = false;
    m_debugMMapsMode = false;
    ai_active = true;
    m_tickCounter = 0;
    m_gameSpeed = GameSpeed.SPEED_STOP;
    m_secondsPerDay = 1;
    (...)

    // Guardado
    m_autoSave = true;
    m_saveTimer = 0;
    m_saveFrequency = 5; // Dias entre cada update

    // Variables del juego, valores base constantes
    m_baseTerritoryDamageRegenerationSpeed = 0.1f;
    m_baseTerritoryLoyaltyRegenerationSpeed = 0.2f;
    m_territoryMillitiaSpawningChance = 5;
    (...)
}
```

InitializeSounds inicializa los sonidos que se van a emplear en todo el juego:

```
public void initializeSounds()
{
    // Musica
    m_backMusic =
        Gdx.audio.newMusic(Gdx.files.internal("Sounds/backMusic.mp3"));

    // Sonidos
    m_zoom_sound =
        Gdx.audio.newSound(Gdx.files.internal("Sounds/zoom.mp3"));
    m_terryClick_sound =
        Gdx.audio.newSound(Gdx.files.internal("Sounds/buttonClick.mp3"));
    (...)
}
```

InitializeTerritories inicializa los territorios que van a existir en el juego, primero se inicializan los arrays de vértices de los polígonos y después cada territorio, que será instanciado y guardado en una HashMap de territorios que se utilizará después para obtener cualquier territorio por ID.

InitializeCountries inicializa los países de forma similar a los territorios, asignándole además el territorio inicial.

```
public void initializeTerritories()
{
    // Definimos poligonos que formaran los territorios
    short[][][] ae = new short[][][]
    {
        {{5995, 2942}, {6059, 2942}, {6115, 2890}, {6119, 2902}, {6119, 2922}, {
        6107, 2922}, {6107, 2946}, {6099, 2946}, {6087, 2986}, {6023, 2978}}};

    short[][][] cz = new short[][][]
    {
        {{5135, 2170}, {5087, 2198}, {5035, 2186}, {5027, 2202}, {5003, 2198}, {
        4963, 2146}, {5031, 2122}, {5127, 2158}}};

    (...)
    // instanciamos territorios ID, nombre, PIB, poligonos
    addTerritory(2, "Norway", 266, no);
    addTerritory(13, "Austria", 141, at);
    addTerritory(14, "Iceland", 141, is);
    (...)
}

public void initializeCountries()
{
    // instanciamos países ID, nombre, cultura, territorio base,
    // capital inicial, tipo de IA
    addCountry(2, "Norway", Culture.CULTURE_WESTERN,
        getTerritoryById(2), 2658, AIType.AI_DEFENSIVE);
    addCountry(4, "Switzerland", Culture.CULTURE_WESTERN,
        getTerritoryById(4), 2397, AIType.AI_PEACEFUL);
    (...)
    addCountry(10, "United States", Culture.CULTURE_WESTERN,
        getTerritoryById(10), 1496, AIType.AI_IMPERIALIST);
    (...)
}
```

initializeDiplomacy define listas de países que forman parte de alianzas y les define un valor diplomático que se aplicará entre todos los miembros de esa alianza, por ejemplo, la unión europea:

```
String[] european_union = new String[] {"Austria", "Belgium",
    "Bulgaria", "Croatia", "Cyprus", "Czech Republic", "Denmark",
    "Estonia", "Finland", "France", "Germany", "Greece", "Hungary",
    "Ireland", "Italy", "Latvia", "Lithuania", "Luxembourg",
    "Netherlands", "Poland", "Portugal", "Romania", "Slovakia",
    "Slovenia", "Spain", "Sweden", "United Kingdom"};

// La definimos con valor 74.
addAlliance(74, GetCountryListByNames(european_union));
```

initializeCountriesFusion define listas de países que se fusionan en un único país, combinando sus valores, por ejemplo, sumando su capital y promediando su diplomacia, se emplea para juntar países demasiado pequeños para ser usados ya desde el principio de la partida, un ejemplo son los países del sudeste asiático:

```
// Southeast Asia
ArrayList<Country> sAsia = new ArrayList<Country>();
sAsia.add(getCountryByName("Vietnam"));
sAsia.add(getCountryByName("Cambodia"));
sAsia.add(getCountryByName("Thailand"));
sAsia.add(getCountryByName("Laos"));
fusionateCountries("Southeast Asia", sAsia);
```

initializeTechnologies, initializeDiplomaticActions, initializeBuildings e initializeUnits inicializan elementos del juego y los guardan en un array, del que podrán ser obtenidos por tipo, se definen nombre, descripción y valor de forma que el valor se lea dinámicamente a partir de esta estancia y pueda ser modificado en cualquier momento cambiando solo el valor de instanciamiento.

```
public void initializeTechnologies()
{
    // Tipo, Nombre, descripción (%v se reemplaza en la interfaz por
    // el valor), valor, tecnologías previas requeridas, cultura
    // requerida, nombre del icono, coste de investigación, tiempo de
    // investigacion, escuela (usado por la IA)
    addTechnology(TechType.TECH_DIGITAL_EXCHANGE_I, "Digital
    Exchange I", "Increases the production of all of your
    territories by %v%", 5, null, null,
    "image_tech_digital_exchange_i", 1000, 20,
    TechSchool.TECH_SCHOOL_ECONOMY);

    addTechnology(TechType.TECH_DIGITAL_EXCHANGE_II, "Digital
    Exchange II", "Increases the production of all of your
    territories by %v%", 10, new
    ArrayList<Technology>(Arrays.asList(getTechnologyByType(TechType.
    TECH_DIGITAL_EXCHANGE_I))),
    null, "image_tech_digital_exchange_ii", 2000, 30,
    TechSchool.TECH_SCHOOL_ECONOMY);

    addTechnology(TechType.TECH_DIGITAL_EXCHANGE_III, "Digital
    Exchange III", "Increases the production of all of your
    territories by %v%", 15, new
    ArrayList<Technology>(Arrays.asList(getTechnologyByType(TechType.
    TECH_DIGITAL_EXCHANGE_II))),
    null, "image_tech_digital_exchange_iii", 4000, 40,
    TechSchool.TECH_SCHOOL_ECONOMY);
    (...)
}
```

```

public void initializeDiplomaticActions()
{
    // Tipo, nombre, descripcion, valor
    addDiplomaticAction(DiplomaticActionType.TYPE_IMPROVE_DIPLOMACY,
        "Money Injection", "Donates money to the target country,
        improving your relations by %v.", 5);

    addDiplomaticAction(DiplomaticActionType.TYPE_HARM_DIPLOMACY,
        "Public Impertinence", "Insults the people of the target
        country, harming your relations by %v.", 5);
    (...)
}

public void initializeBuildings()
{
    // Type, Name, Cost, prodTime, Level, description, value
    // Military Base (1-5)
    addBuilding(BuildType.BUILD_WORKSHOP, "Workshop", 600, 15, 1,
        "Increases the damage regeneration of this territory by %v%",
        30);
    addBuilding(BuildType.BUILD_WORKSHOP, "Workshop", 1200, 18, 2,
        "Increases the damage regeneration of this territory by %v%",
        60);
    (...)
    // Fortress
    addBuilding(BuildType.BUILD_FORTRESS, "Fortress", 800, 30, 1,
        "Increases the defense of your troops in this territory by
        %v%", 10);
    (...)
}

public void initializeUnits()
{
    // Type, Name, desc, time, Cost, Speed, attk, Infantrydefence,
    // ArtilleryDefence
    addUnit(UnitType.UNIT_INFANTRY_MORTAR, "Mortar Unit",
        "Disciplined defensive unit especially effective
        against artillery thanks to his weapon but their offensive
        capabilities are very low.",
        0.01f, 1.0f, 8, 20, 35, 60, getSpriteByName("mortarIcon"));
    (...)
}

```

initializeUI inicializa todos los elementos de la interfaz de usuario, el mapa, los poligonos de países...

```
private void initializeUI()
{
    (...)

    stageDialog = new Stage();
    stageUI = new Stage();
    stageUITech = new Stage();
    (...)

    // Inicializamos camara
    mainUICamera = new OrthographicCamera(windowW, windowH);
    (...)

    // Dibujamos mapamundi en sincronia con la camara
    stageMap.getViewPort().setCamera(mainUICamera);
    (...)

    // Dibujado de mapa
    Image mapLActor = new Image(new
    Texture(Gdx.files.internal("worldMapHDL.png")));
    Image mapRActor = new Image(new
    Texture(Gdx.files.internal("worldMapHDR.png")));
    (...)

    stageMap.addActor(mapLActor);
    stageMap.addActor(mapRActor);
    (...)
}
```

initializeInputProcessors inicializa los listeners de input de usuario, asignando las distintas capas (stages) a un elemento multiplexor de inputs que prioriza según orden en dicha lista, de modo que los primeros elementos de la misma tienen prioridad (si se recibe un input desde una capa prioritaria, lo procesará y no será recibido en capas inferiores)

Añadimos además una instancia de la clase `GestureDetector` que permite detectar movimientos complejos en pantalla, como por ejemplo hacer zoom con los dedos arrastrando.

```
private void initializeInputProcessors()
{
    InputMultiplexer = new InputMultiplexer();
    Gdx.input.setInputProcessor(inputMultiplexer);

    // Instanciamos clases de inputs
    MyInputProcessor backGroundInputProcessor = new
        MyInputProcessor();
    MyGestureHandler gestureInputProcessor = new MyGestureHandler();

    // Add los procesos a la cola de inputs
    inputMultiplexer.addProcessor(stageUIOptions);
    inputMultiplexer.addProcessor(stageDialog);
    inputMultiplexer.addProcessor(stageUIDiplo);
    inputMultiplexer.addProcessor(stageUITech);
    inputMultiplexer.addProcessor(stageUITrain);
    inputMultiplexer.addProcessor(dragSquadsStage);
    inputMultiplexer.addProcessor(stageUI);
    inputMultiplexer.addProcessor(stageSquadTables);
    inputMultiplexer.addProcessor(stageSquads);
    inputMultiplexer.addProcessor(stageMap);
    inputMultiplexer.addProcessor(new
        GestureDetector(gestureInputProcessor));
    inputMultiplexer.addProcessor(backGroundInputProcessor);
}
```

`initializeButtons` inicializa los botones como variables globales para que podamos guardar su estado de check, en otro diseño ese valor se perdería en cada update de interfaz.

Las siguientes inicializaciones corresponden con el sistema de nodos que se emplearán tanto para el pathing de tropas de las IAs como para leer territorios a partir de posiciones, handleando por ejemplo el click del usuario en pantalla.

`initializeNodePoints` inicializa las estructuras de posición que llamamos nodos y cuyas ventajas y aplicaciones detallaremos en otro apartado, en este apartado crearemos una estructura de datos que relacionará posiciones XY con el territorio ubicado en ellas en base los polígonos que lo forman, de esta forma tendremos en memoria información sobre a qué territorio corresponde cada pixel del mapa.

Dichos nodos tendrían idealmente un tamaño de un pixel, pero esto supone un enorme e innecesario coste computacional, por lo que se ha optado por establecer un número fijo de nodos (150.000), que serán de un tamaño variable en función de la resolución para intentar garantizar que este número sea constante con la resolución, en este diseño, el tamaño varía en torno a 4 pixeles por nodo.

Por el coste computacional que supone comprobar en que polígono se encuentra contenido cada nodo, esto sólo se realizará la primera vez que el usuario arranque el juego, generando un fichero externo codificado en base64 que contendrá estos datos y que será leído en posteriores lanzamientos del programa, reduciendo radicalmente el tiempo de inicialización en estos arranques posteriores. Si hubiera algún error al leer este archivo, sería descartado y se volvería a generar.

El algoritmo de generado de nodos está optimizado y es el siguiente:

```
// Numero ideal de nodos que intentaremos tener siempre
int nodeNumber = 150000;
m_nodeSize = (int) Math.round(Math.sqrt(windowW * windowH /
nodeNumber));
int arrayWidth = (int) windowW / m_nodeSize + 1;
int arrayHeight = (int) windowH / m_nodeSize + 1;
// El array es de tipo byte que es el tipo de variable más pequeña
capaz de representar todos los valores.
m_mapPoints = new byte[arrayWidth][arrayHeight];

// Inicializamos a -128 en vez de 0
for (int i = 0; i < arrayWidth; i++)
    for (int j = 0; j < arrayHeight; j++)
        m_mapPoints[i][j] -= 128;

for (Territory terry : m_territoryList.values())
    for (Polygon poly : terry.getTerritoryPolygonList())
    {
        // Calculamos el rectangulo del territorio y solo barremos los
        // nodos de su interior
        // Esta formula realiza un escalado y offset para transportar
        // las dimensiones de los nodos a las del
        // array de poligonos, ya que no comparten referencia 2D.
        Rectangle rect = poly.getBoundingRectangle();
        int startX = (int) ((rect.getX() + windowW / 2 + m_nodeSize /
            2) / m_nodeSize);
        int endX = (int) (((rect.getX() + rect.getWidth() + windowW / 2
            + m_nodeSize / 2) / m_nodeSize));
        int startY = (int) ((-rect.getY() + windowH / 2 + m_nodeSize /
            2) / m_nodeSize);
        int endY = (int) ((-rect.getY() + rect.getHeight() + windowH
            / 2 + m_nodeSize / 2) / m_nodeSize);
```

```

for (int i = startX; i <= endX; i++)
    for (int j = endY; j <= startY; j++)
    {
        int coordX = (int) ((i * m_nodeSize - m_nodeSize / 2 -
                               windowW / 2));
        int coordY = (int) (-(j * m_nodeSize - m_nodeSize / 2 -
                               windowHeight / 2));

        if (poly.contains(coordX, coordY))
        {
            // Add punto al array
            m_mapPoints[i][arrayHeight - j] =
                (byte) (terry.getId() - 128);

            // Add posicion al territorio en cuestion
            terry.addPosition(i, arrayHeight - j);
        }
    }
}

```

Una vez generado el array de nodos, se escribirá en el fichero externo existente en los recursos internos de la aplicación: Maps/worldMap.dat

Se pretende normalizar la generación de nodos en el futuro, de forma que los usuarios reciban en la aplicación el fichero externo de nodos generado bajo una resolución de referencia y adaptado a la suya, de modo que ningún usuario necesite calcular los nodos localmente.

Este fichero puede ser borrado mediante una variable de la inicialización, forzando una nueva generación de nodos y su fichero.

initializeMovementMaps inicializa el grafo que relaciona los nodos, creando conexiones entre ellos que constituirán los posibles caminos para la IA, caminos con un coste asociado a ellos dependiente de distintos factores, como si se cruza una frontera, si corresponde a agua o la diplomacia entre el país que mueve las tropas y el país atravesado por esos nodos, todo esto se detallará más adelante.

InitializeRandomEvents inicializa los eventos aleatorios que pueden ocurrir en el juego con su probabilidad (sobre 100)

```

public void initializeRandomEvents()
{
    addRandomEvent(EventType.TYPE_DIPLOMATIC_CONFUSION, 2);
    addRandomEvent(EventType.TYPE_DIPLOMATIC_IMPROVEMENT, 2);
    addRandomEvent(EventType.TYPE_NATURAL_DISASTER, 1);
}

```

initializeThreads inicializa los threads que se quedarán en pause a la espera de eventos relacionados con dichos threads, la modificación de ciertas variables bien sincronizadas desencadenará la ejecución de las tareas asociadas.

```
public void initializeThreads()
{
    // Thread de movimiento de tropas
    m_pathingThread = new PathingThread();
    m_pathingThread.start();
    m_pathingThread.setName("Pathing Thread");

    // Thread de updates
    m_updateThread = new UpdateThread(this);
    m_updateThread.start();
    m_updateThread.setName("Update Thread");

    // Thread de update IA
    m_aiThread = new AIThread(m_countryList);
    m_aiThread.start();
    m_aiThread.setName("IA Thread");
}
```

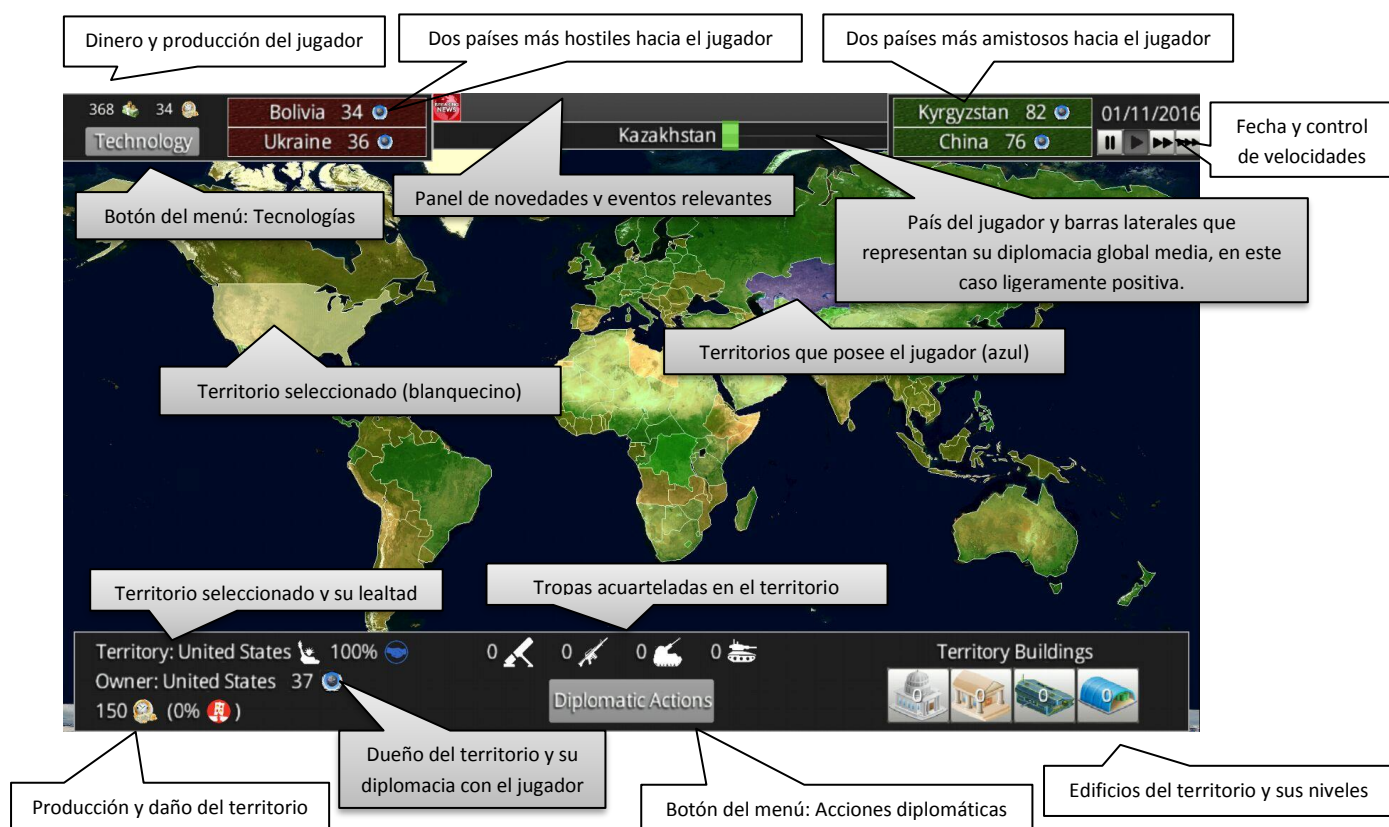
saveRelevantVariables ejecuta algoritmos que requieren que toda la inicialización haya terminado pero que son parte de la inicialización, por ejemplo, define el centro de cada territorio y se lo establece, esto no se puede ejecutar en la inicialización de territorios porque utiliza los nodos que aún no habían sido instanciados.

3.3 INTERFAZ DE USUARIO

La interfaz de usuario consiste en un mapamundi sobre el que se puede hacer zoom y moverse libremente arrastrando sobre la pantalla, se pueden clicar países para realizar acciones sobre ellos o simplemente ver sus valores. Consta también de barras de acción y botones para realizar todas las tareas necesarias para el desarrollo del juego.

La interfaz superior muestra todo lo referente al jugador y su país, mientras que la barra inferior muestra lo referente al territorio seleccionado, variando las opciones si el territorio es del jugador o no.

El color de los territorios en el mapa refleja el valor diplomático del país dueño de ese territorio con el jugador, siendo más verde cuanto mejor relación tengan y más rojo cuanto peor sean sus relaciones.



La siguiente imagen muestra la barra inferior cuando el territorio seleccionado es del jugador, observamos que aparece un botón para producir tropas, habilita la interfaz de construcción de edificios y la interfaz de gestión de tropas.



La interfaz de gestión de tropas se utiliza para desplazar las tropas por el mapa o acuartelarlas en su territorio origen, funciona del siguiente modo:

- Cuando el territorio produzca tropas, estas se colocarán en la barra superior de tropas, constituyendo las tropas acuarteladas en el territorio y sumándose a otras tropas previas que hubiera.
- Las flechas que hay debajo de cada tipo de unidad permiten desplazar las tropas entre la barra superior e inferior, el número de tropas que se mueven entre barras en cada click lo establece el botón de la izquierda y puede ser modificado pulsando dicho botón, moviéndose siempre en ordenes de magnitud 10.
- El icono azul que se encuentra a la izquierda de la barra inferior puede ser arrastrado sobre un punto del mapa, en el momento de hacer eso, las tropas que se encuentren asignadas en la barra inferior serán desplazadas a la posición del mapa elegido, constituyendo un escuadrón de tropas desplazable en el mapa que contendrá a las tropas arrastradas.
- Si un escuadrón del mapa es arrastrado sobre cualquier punto de la barra inferior de la interfaz de usuario, dichas tropas volverán a la barra inferior de la interfaz.
- Sólo se pueden arrastrar tropas desde la interfaz a un punto del territorio al que pertenecen, y sólo pueden devolverse si se encuentran también en dicho territorio.

Existen además tres menús que se muestran en pantalla y que se activan desde distintos botones de la interfaz corriente (acciones diplomáticas, tecnologías y entrenamiento de tropas), sólo puede abrirse uno simultaneo, y se reemplazarán entre ellos si se abren:





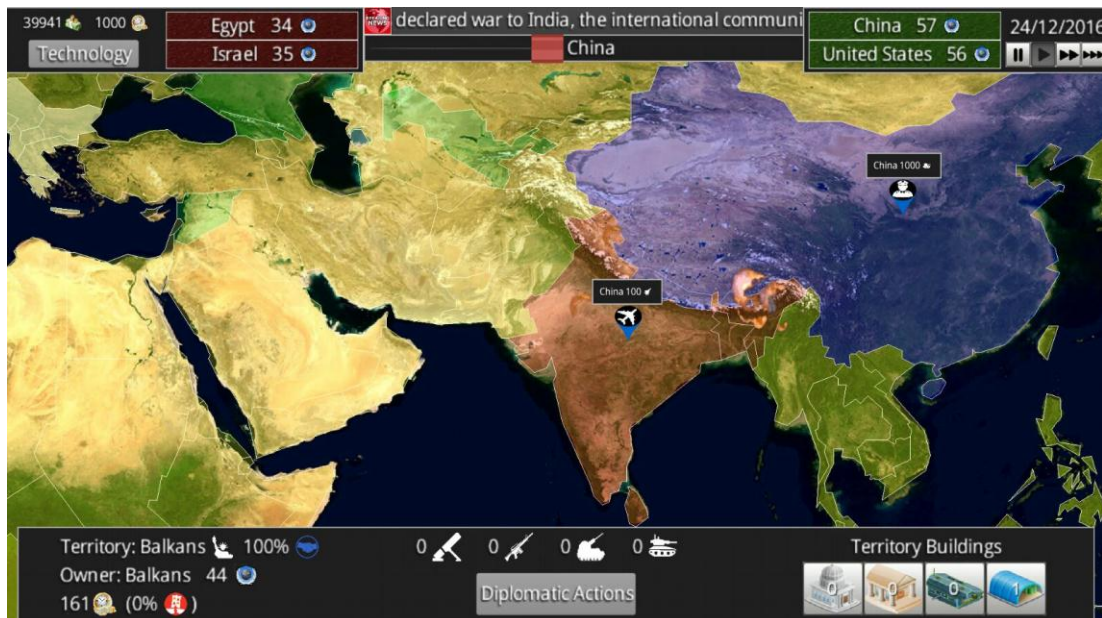
Las tecnologías ya investigadas aparecen con un fondo verde, aunque no se refleja en la imagen.

Cuando se inicia una investigación, un entrenamiento de tropas o una construcción de edificio, se mostrará además una barra de progreso de la producción concreta, generalmente a la derecha de la interfaz de producción. Como se muestra en la siguiente imagen.



Interfaz de tropas y ataques

Cuando un escuadrón de tropas es arrastrado al territorio origen, aparecerá en ese punto un icono que representa a la escuadra y que podrá arrastrarse sobre otro punto del mapa:



En la imagen anterior podemos observar que hay varios tipos de escuadras que se diferencian en el icono que las representa: aviación, tropas terrestres y milicias respectivamente. Mientras que el color de la escuadra representa la relación diplomática con el jugador, siendo azul del propio jugador, verde aliada, amarilla neutral y roja hostil:



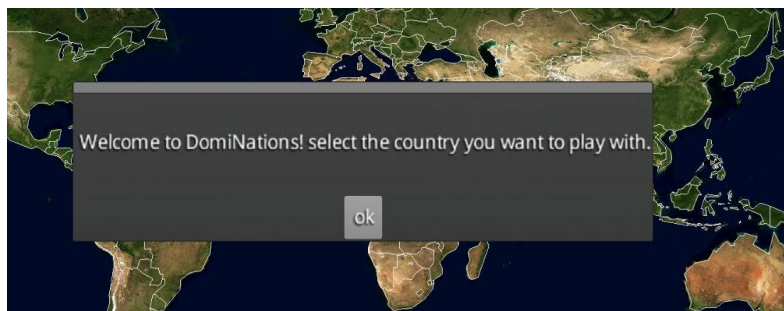
Las escuadras pueden ser movidas por su dueño a cualquier punto del mapa, en caso de ocurrir, la escuadra se desplazará por el mapa desde su ubicación actual hasta el destino, a la velocidad más lenta de las tropas que contenga y mostrando en la interfaz el camino van a seguir las tropas, el movimiento se puede interrumpir cuando se desee desplazando las tropas a otro punto, reemplazando el nuevo destino al previo.

Los territorios sobre los que se desarrolla una batalla serán indicados con un icono sobre el mismo, mientras que los territorios que están siendo bombardeados muestran animaciones de explosiones a lo largo de su área.



Alertas del juego

Durante el desarrollo del juego se muestran mensajes en pantalla para informar al usuario de diferentes eventos, la siguiente imagen muestra el alert que aparece al iniciar el juego y el que pide confirmación del país elegido inmediatamente después:



Para generar diálogos informativos basta con llamar al método:

```
public static void showDialog(String text)
{
    // Dialogo
    final Dialog dialog = new Dialog("", defaultSkin, "default");
    (...)
    dialog.text(text);

    // Boton
    TextButton but = new TextButton("ok", defaultSkin, "default");
    (...)
    but.addListener(new InputListener() {
        @Override
        public boolean touchDown(InputEvent event, float x, float y,
            int pointer, int button) {
            // Sonido
            Game.playSound(m_buttonClick_sound, 0.5f);

            // Limpiamos
            stageDialog.clear();

            // Cerramos dialogo
            dialog.hide();
            return true;
        }
    });
    dialog.button(but, true);

    // Mostramos dialogo
    (...)
    dialog.show(stageDialog);
}
```

Mientras que para el segundo diálogo que tiene dos posibles respuestas necesitaremos elaborar un método personalizado:

```
public void selectCountryAlert(final Territory terry)
{
    // Dialogo
    final Dialog dialog = new Dialog("", defaultSkin, "default");
    (...)
    dialog.text("Are you sure you want to play as " +
        terry.getOwner().getName() + "?");

    // Boton yes
    TextButton but = new TextButton("yes", defaultSkin, "default");
    (...)
}
```

```

but.addListener(new InputListener()
{
    @Override
    public boolean touchDown(InputEvent event, float x, float y,
        int pointer, int button)
    {
        // Sonido
        Game.playSound(m_buttonClick_sound, 0.5f);

        // Limpiamos
        stageDialog.clear();

        // Ya tenemos jugador, iniciamos juego y cerramos dialogo.
        m_player = terry.getOwner();
        startGame();
        dialog.hide();
        return true;
    }
});
dialog.button(but, true);

TextButton butNo = new TextButton("no", defaultSkin, "default");
(...)
butNo.addListener(new InputListener()
{
    @Override
    public boolean touchDown(InputEvent event, float x, float y,
        int pointer, int button)
    {
        // Sonido
        Game.playSound(m_buttonClick_sound, 0.5f);

        // Limpiamos
        stageDialog.clear();

        // Cerramos dialogo
        dialog.hide();
        return false;
    }
});
dialog.button(butNo, true);

// Mostramos dialogo
Gdx.app.postRunnable(new Runnable() {
    (...)
    dialog.show(stageDialog);
}

```

Iconos y su significado

Como hemos podido apreciar en las imágenes anteriores, existen muchos iconos que representan distintos elementos del juego, mejorando tanto el atractivo estético como la propia legibilidad de la interfaz, lo que representa cada icono es bastante intuitivo.

Música y sonidos

Dominations es una aplicación que posee tanto música de fondo como sonidos FX que se reproducen en distintos elementos del juego:

Nombre	Efecto
backMusic.mp3	Musica background durante el juego
mainMenuMusic.mp3	Musica background del menú principal.
zoom.mp3	Sonido FX cuando la cámara hace zoom forzado.
buttonClick.mp3	Sonido FX de click sobre territorio.
buttonToggleClick.mp3	Sonido FX de click sobre botón.
bomb.mp3 bomb2.mp3 bomb3.mp3 bomb4.mp3	Sonido FX para las explosiones de bombardeos, se elige aleatoriamente el sonido a reproducir para cada explosión que se produce.
sendTroops.mp3	Sonido FX cuando el jugador envía tropas terrestres.
sendAircrafts.mp3	Sonido FX cuando el jugador envía tropas aéreas.
battlefield.mp3	Sonido FX de fondo durante una batalla que afecte al jugador.
alertUnderAttack.mp3	Sonido FX que alerta al jugador de que algún territorio suyo está siendo atacado.
terryLost.mp3	Sonido FX que alerta al jugador de que algún territorio suyo ha sido perdido.
terryConquered.mp3	Sonido FX que alerta al jugador de que ha conquistado con éxito algún territorio.

La música y los sonidos se cachean en variables del juego durante la fase de la inicialización y se reproducen llamando a los siguientes métodos:

```
// Musica
m_mainMenuMusic =
    Gdx.audio.newMusic(Gdx.files.internal("Sounds/mainMenuMusic.mp3"));

playMusic(m_mainMenuMusic, 0.60f, true);
private void playMusic(Music mus, float vol, boolean loop)
{
    mus.play();
    mus.setLooping(loop);
    mus.setVolume(vol);
}
```

```

public void conquestTerritory(Country conqueror)
{
    (...)
    if (conqueror.equals(Game.m_player))
        Game.playSound(Game.m_terrorConquered_sound, 0.8f);
    else if (getOwner().equals(Game.m_player)) // Si lo pierde
        Game.playSound(Game.m_terrorLost_sound, 0.8f);
    (...)
}

public static void playSound(Sound sound, float volume)
{
    // Paramos el mismo sonido si ya sonaba
    sound.stop();

    // Reproducir sonido
    sound.play(volume);
}

```

Los sonidos se reproducen simultáneamente si se fuerza el inicio de un sonido mientras se reproduce otro. La capacidad de reproducción simultánea está limitada por una variable de configuración en la clase `AndroidApplicationConfiguration` que se carga en la inicialización de la aplicación y que para el caso de *Dominations* establece un límite de 16.

```

/** the maximum number of {@link Sound} instances that can be played
    simultaneously, sets the corresponding {@link SoundPool}
    * constructor argument. */
public int maxSimultaneousSounds = 16;

```

Implementación de animaciones

Las animaciones se implementan mediante una abstracción del framework llamada *Action*, que implementa distintas acciones sobre actores, como por ejemplo desplazarlos o escalarlos, mediante este sistema se ha implementado el icono de batalla en territorio (que simula el latido de un corazón) y el de los números de cambios de diplomacia (que se mueven hacia arriba mientras se desvanecen).

Las acciones permiten además ser anidadas en acciones superiores, que permiten por ejemplo la implementación de secuencias o la repetición de las mismas. En los siguientes fragmentos se muestra la implementación de las dos animaciones a las que he hecho referencia antes:

```
// Acciones del icono de batalla
// Crecer y decrecer
Action actionUp = Actions.scaleBy(0.10f, 0.10f, 0.1f);
Action actionDown = Actions.scaleBy(-0.10f, -0.10f, 0.1f);

// Secuencia de acciones, ejecutará una después de la otra (crecer,
decrecer)
SequenceAction seqAction = new SequenceAction();
seqAction.addAction(actionUp);
seqAction.addAction(actionDown);

// Repetición de la secuencia de acciones, cuando acabe la secuencia,
la volverá a hacer.
RepeatAction repeat = new RepeatAction();
repeat.setCount(repeat.FOREVER);
repeat.setAction(seqAction);

// Esta es la acción final, que contiene todas las demás, de modo que
la asociamos al actor (el icono)
m_icon.addAction(repeat);

(...)
// Añadimos el actor a la stage, que será dibujado en el siguiente
ciclo del thread de renderizado.
Game.stageAnimations.addActor(m_icon);

// Acciones de los textos de cambio de diplomacia
// Accion que hace que el label se desvanezca
Action fadeOut = Actions.fadeOut(0.3f);
// La encapsulamos en otra acción que le implementa un retardo
temporal a su ejecución
DelayAction fadeOutDelay = new DelayAction();
fadeOutDelay.setAction(fadeOut);
fadeOutDelay.setDuration(0.7f);

// Accion que hace que el actor empiece a subir (en 50 px en este
caso)
Action actionUp = Actions.moveBy(0,
Formulas.convertScaleToResolution(50), 1);

// Accion que eliminara el actor cuando ya sea totalmente transparente
Action actionRemove = Actions.removeActor();

// Secuencia de acciones, lo subimos y eliminamos cuando ya haya
recorrido los 50 px
SequenceAction seqAction = new SequenceAction();
seqAction.addAction(actionUp);
seqAction.addAction(actionRemove);
```

```

// Añadimos dos acciones al label, la de desvanecimiento y la
// secuencia de movimiento + eliminado (la idea es
// que así hagan ambas cosas en paralelo)
diploLabel.addAction(fadeOutDelay);
diploLabel.addAction(seqAction);

(...)

// Color segun valores
if (value < 0)
    diploLabel.setColor(1, 0, 0, 1);
else
{
    diploLabel.setText("+" + diploLabel.getText()); // Add tambien
    simbolo si es positivo
    diploLabel.setColor(0, 1, 0, 1);
}

// Añadimos el actor a la stage, que será dibujado en el siguiente
// ciclo del thread de renderizado.
stageLabelAnimations.addActor(diploLabel);

```

Las explosiones de los bombardeos sin embargo se implementan mediante la clase Animation, que se emplea mediante la adición de imágenes a una lista que juntas forman la animación:

```

m_explosion1 = new Array<>();
m_explosion1.add(new TextureRegion(getSpriteByName("expAa")));
m_explosion1.add(new TextureRegion(getSpriteByName("expAb")));
m_explosion1.add(new TextureRegion(getSpriteByName("expAc")));
m_explosion1.add(new TextureRegion(getSpriteByName("expAd")));
m_explosion1.add(new TextureRegion(getSpriteByName("expAe")));
anim = new Animation(1f / duration, Game.m_explosion1);

```

Donde expAa, expAb, etc, son imágenes de explosiones (fotogramas de una única explosión) y generaremos varios de estos arrays que corresponden con diferentes explosiones, y el siguiente algoritmo los hará aparecer en puntos aleatorios del territorio:

```
private void spawnExplosionsAnimations(float bomberDamage, boolean
playerIsBombing)
{
    // Numero de explosiones aleatorias
    int blastsPerDamage = (int)(10 * bomberDamage);
    int maxBlast = Utils.getRandomValue(0, 5 > blastsPerDamage ? 5 :
    blastsPerDamage);
    for (int i = 0; i < maxBlast; i++)
    {
        // Duracion, size y explosion a usar aleatoria
        int size = Utils.getRandomValue(20, 40);
        int duration = Utils.getRandomValue(15, 25);
        int sprite = Utils.getRandomValue(1, 3);
        Animation anim = null;
        switch (sprite)
        {
            case 1:
                anim = new Animation(1f / duration,
                    Game.m_explosion1);
                break;
            case 2:
                anim = new Animation(1f / duration,
                    Game.m_explosion2);
                break;
            case 3:
                anim = new Animation(1f / duration,
                    Game.m_explosion3);
                break;
        }

        if (anim != null)
        {
            // Posicion de la explosion tambien aleatoria
            Vector2 pos = getRandomPoint();
            AnimatedActor actorAnim = new AnimatedActor(new
                AnimationDrawable(anim));
            actorAnim.setBounds(pos.x, pos.y, size, size);
            Game.stageAnimations.addActor(actorAnim);
        }
    }
}
```

```

// Sonido solo si el jugador esta implicado
if (getOwner().equals(Game.m_player) || playerIsBombing)
{
    // Sonido aleatorio
    Sound sound = null;
    switch (Utils.getRandomValue(1, 4))
    {
        case 1:
            sound = Game.m_bomb_sound;
            break;
        case 2:
            sound = Game.m_bomb2_sound;
            break;
        case 3:
            sound = Game.m_bomb3_sound;
            break;
        case 4:
            sound = Game.m_bomb4_sound;
            break;
    }
    if (sound != null)
        Game.playSound(sound, 0.05f);
    }
}
}

```

Interfaces de usuario

Refiriéndonos específicamente a interfaz de usuario como las tablas que contienen valores y/o botones con los que el usuario puede interactuar, se implementan mediante una clase del framework que es Table y que es de tipo Actor, por lo que puede ser renderizada fácilmente y sobre la que se pueden implementar Actions.

Todos los valores de la interfaz, tablas y demás se construyen en un único método: `updateUI()`, al que llamamos en cada update del juego y siempre que ocurran ciertos eventos, como un input de usuario en algún botón. En el caso de la siguiente imagen, un click de usuario en una tecnología forzará una llamada al `updateUI` que actualizará la parte inferior que muestra la tecnología seleccionada y sus costes.



Los siguientes fragmentos muestran trozos relevantes del método `updateUI`:

```
private void updateUI()
{
    (...)

    // Menu de tecnologias, solo actualizar si es visible
    if (m_outTableTech.isVisible())
        updateUITech();

    // Menu de training, solo actualizar si es viable
    if (m_outTableTrainTroops.isVisible())
        updateUITrain();

    // Menu de diplomacia, solo actualizar si es viable
    if (m_outTableDiplo.isVisible())
        updateUIDiplo();

    if (m_outTableOptions.isVisible())
        updateUIOptions();

    (...)

    // PANEL ECONOMICO+TECNOLOGICO (dinero + produccion + tecnologia)
    Label money = new Label(String.valueOf(m_player.getMoney()),
        smallSizeSkin);
    Table tableResourcesAndTech = new Table();
    tableResourcesAndTech.add(money).padRight(smallPad);
    tableResourcesAndTech.add(new Image(getSpriteByName("money-icon-13"))).size(money.getHeight() - smallPad, money.getHeight() - smallPad).padRight(medPad);
```

```

tableResourcesAndTech.add(new
Label(String.valueOf(m_player.getTotalMoneyProduction()),
smallSizeSkin)).padRight(smallPad);
tableResourcesAndTech.add(new
Image(getSpriteByName("moneyProd"))).size(money.getHeight() -
smallPad, money.getHeight() - smallPad);
tableResourcesAndTech.row();
tableResourcesAndTech.add(techButton).padTop(smallPad).size(new
Label(techButton.getText(),
(...)

// Panel superior+derecha (fecha y botones de speed)
DateFormat formatter = new SimpleDateFormat("dd/MM/yyyy");
Label dateText = new Label(formatter.format(m_gameDate), defaultSkin);
Table tableDate = new Table();
tableDate.add(dateText).colspan(4);
tableDate.row();
tableDate.add(buttonStop).size(dateText.getWidth() / 4,
dateText.getWidth() / 4);
tableDate.add(buttonLow).size(dateText.getWidth() / 4,
dateText.getWidth() / 4);
tableDate.add(buttonMedium).size(dateText.getWidth() / 4,
dateText.getWidth() / 4);
tableDate.add(buttonHigh).size(dateText.getWidth() / 4,
dateText.getWidth() / 4);
(...)

// Tabla izquierda
Table leftTable = new Table();
leftTable.add(tableResourcesAndTech).size(resPortion *
topMoneyTechBlockPortions, resPortion * topHeightPortions);
leftTable.add(diploHostileTable).size(resPortion *
topDiploBlockPortions, resPortion * topHeightPortions);
leftTable.setBackground(new NinePatchDrawable(new
NinePatch(getSpriteByName("tableBackGround"), 10, 10, 10, 10)));

// Tabla derecha
Table rightTable = new Table();
rightTable.add(diploFriendsTable).size(resPortion *
topDiploBlockPortions, resPortion * topHeightPortions);
rightTable.add(tableDate).size(resPortion * topDateBlockPortions,
resPortion * topHeightPortions);
rightTable.setBackground(new NinePatchDrawable(new
NinePatch(getSpriteByName("tableBackGround"), 10, 10, 10, 10)));

// Tabla TOP FINAL
m_topTable.add(leftTable);
m_topTable.add(tableNews).size(resPortion *
topInternationalDiplomacyBlockPortions, resPortion *
topHeightPortions).top();
m_topTable.add(rightTable);
m_topTable.row();
m_outTopTable.add(m_topTable);
m_outTopTable.row();
(...)

stageUI.addActor(m_outTopTable);

```

```

(...)
// DATOS TERRITORIO SELECCIONADO
if (m_selectedTerritory != null)
{
    float diploE =
    m_selectedTerritory.getOwner().getDiplomacyValueWithCountry(m_pl
    ayer);
    // INFO TERRITORIO
    Label territoryLoyalty = new
    Label((int)m_selectedTerritory.getLoyalty() + "%", defaultSkin);
    Label ownerDiplo = new Label(String.valueOf((int)diploE),
    defaultSkin);
    Label terryMoneyProd = new
    Label(String.valueOf(m_selectedTerritory.getMoneyProduction()),
    defaultSkin);
    int terryDamages = (int)m_selectedTerritory.getDamage();
    Label damages = new Label("(" + (terryDamages == 0f ? "" : "-")
    + String.valueOf(terryDamages) + "%", defaultSkin);
    // Fila de titulo
    Table terryName = new Table();
    terryName.add(new Label("Territory: " +
    m_selectedTerritory.getName(), defaultSkin)).padRight(smallPad);
    Sprite cultureIcon = m_selectedTerritory.getOwner().getIcon();
    if (cultureIcon != null)
        terryName.add(new
        Image(cultureIcon)).padRight(smallPad).height(territoryLoyalty.g
        etHeight() - smallPad).width(territoryLoyalty.getHeight() -
        smallPad);
    terryName.add(territoryLoyalty).padRight(smallPad);
    terryName.add(new
    Image(getSpriteByName("loyaltyIcon")).height(territoryLoyalty.g
    etHeight() - smallPad).width(territoryLoyalty.getHeight() -
    smallPad);
    // Fila de owner
    Table ownerInfo = new Table();
    ownerInfo.add(new Label("Owner: " +
    m_selectedTerritory.getOwner().getName(),
    defaultSkin)).padRight(medPad);
    if (!m_selectedTerritory.getOwner().equals(m_player)) // Solo
    mostrar diplomacia si no es nuestro territorio (evitar ver
    diplomacia con nosotros mismos.
    {
        ownerInfo.add(ownerDiplo).right().padRight(smallPad);
        ownerInfo.add(new
        Image(getSpriteByName("diplomacyIcon")))
        height(ownerDiplo.getHeight() -
        smallPad).width(ownerDiplo.getHeight() - smallPad);
    }
    // Fila de prod
    Table terryProd = new Table();
    terryProd.add(terryMoneyProd).padRight(smallPad);
    terryProd.add(new
    Image(getSpriteByName("moneyProd")).height(terryMoneyProd.getHe
    ight() - smallPad).width(terryMoneyProd.getHeight() -
    smallPad).left().padRight(smallPad);
    terryProd.add(damages).padRight(smallPad);
    terryProd.add(new
    Image(getSpriteByName("damageIcon")).height(damages.getHeight()
    - smallPad).width(damages.getHeight() - smallPad);
    terryProd.add(new Label(")", defaultSkin));
}

```

```

// Tabla final
Table terryInfo = new Table();
terryInfo.add(terryName).left();
terryInfo.row();
terryInfo.add(ownerInfo).left();
terryInfo.row();
terryInfo.add(terryProd).left();

(...)

// Tabla BOT FINAL
// Info si seleccionamos territorios propios
if (m_selectedTerritory.getOwner().equals(m_player))
{
    m_botTable.add(terryInfo).left().top().width(resPortion *
        botTerryInfoBlockPortions);

    m_botTable.add(territoryTroops).left().top().padRight (medPad) .
        width(resPortion * botTroopsDeployBlockPortions);

    m_botTable.add(tableBuildings).left().top().width(resPortion *
        botBuildingsBlockPortions);
}
else // Info si seleccionamos territorios no propios
{

    m_botTable.add(terryInfo).left().top().padRight (medPad) .width(
        resPortion * botTerryInfoBlockPortions); // Tamaño fijo
        es mas comodo para la vista

    m_botTable.add(territoryTroops).left().top().padRight (medPad)
        .width(resPortion * botTroopsInfoPortions);

    m_botTable.add(tableBuildings).left().top().width(resPortion *
        botTerryInfoBlockPortions);
}

m_outBotTable.add(m_botTable).center().bottom();
stageUI.addActor(m_outBotTable);

```

Los siguientes fragmentos de código corresponden con los métodos de las tablas que se abren con botones y cuyo update se llama también desde updateUI si dichas tablas son visibles, variable que se establecerá en true al pulsar los botones asociados y false cuando se cierren con el icono de la cruz de las ventanas o se abra otra diferente:

```
private void updateUIOptions()
{
    // Limpiamos tablas
    m_outTableOptions.clearChildren();
    m_tableOptions.clearChildren();
    (...)

    Label reffLabelStart = new Label(saveAndExitButton.getText(),
        defaultBigSkin);
    Label title = new Label("Game options", defaultBigSkin);
    m_tableOptions.add(title).left();
    m_tableOptions.add(optionsCrossButton).right().top().size(title
        .getHeight(), title.getHeight());
    m_tableOptions.row();
    m_tableOptions.add(saveAndExitButton).expand().colspan(2)
        .padBottom(medPad);
    m_tableOptions.row();

    m_tableOptions.add(optionsButton).expand().colspan(2).padBottom
        (medPad );

    // Add Tabla final
    m_outTableOptions.add(m_tableOptions);
    stageUIOptions.addActor(m_outTableOptions);
}
```

```

public void updateUIDiplo()
{
    (...)

    m_tableDiplo.add(new Label("Diplomatic actions to " +
        m_selectedTerritory.getName(), defaultSkin)).left();

    m_tableDiplo.add(diploCrossButton).right().top().size(troopButtonWidth / 1.5f, troopButtonHeight / 1.5f);
    m_tableDiplo.row();

    Table inTable = new Table();
    for (final DiplomaticAction action : m_diplomaticActions)
    {
        // Add boton
        TextButton button = new TextButton(action.getName(),
            defaultSkin);
        button.addListener(new InputListener()
        {
            @Override
            public boolean touchDown(InputEvent event, float x,
                float y, int pointer, int button)
            {
                (...)
                String response = action.tryExecuteAction(m_player,
                    target, true);
                if (response != null)
                    showDialog(response);
                (...)
            }
        });
        inTable.add(button).left().padRight(smallPad);

        // Coste
        int cost = action.getCost(m_player, target);
        if (cost > 0)
        {
            Label labelCost = new Label(String.valueOf(cost),
                defaultSkin);
            inTable.add(labelCost).padRight(smallPad);
            inTable.add(new Image(getSpriteByName("money-icon-13"))).padRight(medPad);
        }
        else
        {
            Label labelCost = new Label(String.valueOf(cost),
                defaultSkin);
            inTable.add().size(labelCost.getWidth(),
                labelCost.getHeight()).padRight(smallPad);
            inTable.add().padRight(medPad).size(labelCost.getHeight(),
                labelCost.getHeight());
        }
    }
}

```

```

Label description = new Label(action.getDisplayDescription(),
                                defaultSkin);

description.setWrap(true);
inTable.add(description).width(descriptWidth);
inTable.row();

// Transparencia del boton si no es pulsable
if (!diplomaticActionCanBeExecuted(action, m_player,
                                     target).getResult())
    button.setColor(1, 1, 1, 0.5f);
else
    button.setColor(1, 1, 1, 1);
}

// Tabla interior
m_tableDiplo.add(inTable).left().colspan(2).padBottom(smallPad);

// Add Tabla final
m_outTableDiplo.add(m_tableDiplo);
stageUIDiplo.addActor(m_outTableDiplo);
}

```

Podemos extraer varias observaciones en el diseño del sistema:

- El diseño por tablas y la definición de sus propiedades permite una adaptación dinámica de la interfaz a la resolución del dispositivo, escalando los tamaños de ciertas celdas para garantizar que los elementos de las tablas se mantengan dónde deben.
- Las tablas se generan en cada llamada al método, borrando las anteriores; este diseño es redundante pero más simple, tras realizar los perfilados concluí que no tenía ningún impacto en el rendimiento.
- Cuando las tablas han sido generadas en función del contexto del juego, se guardan en una stage como un actor corriente.
- Las stages que contienen las tablas de la interfaz serán dibujadas en el siguiente ciclo de renderización, en este caso además la función `updateUI` se mantiene sincronizada con el thread de renderizado, lo que garantiza que no ocurran problemas de ningún tipo.
- Hay ciertos elementos de las tablas que deben ser establecidos como variables globales para evitar que el dato se pierda en una nueva llamada a `updateUI`, por ejemplo, el edificio seleccionado en la interfaz de construcción.

3.4 RENDERIZADO, CAPAS Y CÁMARA

Como ya hemos comentado en muchos de los apartados anteriores, el renderizado es ejecutado por el método `render()` que se ejecuta desde un thread que emplea `OpenGL` para renderizar mediante el método `draw`. Una llamada de `draw` sobre una `stage` llamará recursivamente a `draw` para los actores que componen la `stage`, dibujándolos en la ubicación del mismo y con su tamaño.

Como la función `render` es única para todo el juego, si se desean dibujar distintos menús del juego, necesitaremos organizar el flujo de programa, en este caso mediante un `switch` que separa el renderizado del menú principal y el del juego en sí mismo.

En `render` también ubicamos la ejecución de las acciones de los actores, al igual que con `draw`, una llamada a `act()` sobre una `stage`, ejecutará un `act()` sobre todos los actores que la componen, recibiendo como parámetro el tiempo transcurrido desde el anterior `render`, que emplea para el cálculo de la velocidad de la acción (como por ejemplo moverse) y que se puede obtener llamando al método `Gdx.graphics.getDeltaTime()`

Los siguientes fragmentos pertenecen al método `render` y se puede ver fácilmente lo que hace cada parte en base a sus comentarios sobre el código:

```
public void render()
{
    // Color de fondo Negro
    Gdx.gl.glClearColor(0, 0, 0, 0);

    // Limpiamos pantalla en cada render, para evitar que se acumule
    // lo nuevo con lo previo
    Gdx.gl.glClear(GL20.GL_COLOR_BUFFER_BIT);

    switch (m_gameState)
    {
        case STATE_MENU:
            // Dibujado del menú
            stageMainMenu.act(Gdx.graphics.getDeltaTime());
            stageMainMenu.draw();
            break;
        // Estamos en el juego, update a la interfaz del juego
        case STATE_GAME:
            // Trigger de update de interfaz para cuando se llama
            // desde un método estático
            if (m_forceUpdate)
            {
                updateUI();
                m_forceUpdate = false;
            }
    }
}
```

```

// Actualizar Camara: actualizaremos su campo de visión en
    función de su posición y zoom que se
define por inputs de usuario.
if (m_cameraAction != null)
    m_cameraAction.update(Gdx.graphics.getDeltaTime());
if (m_zoom < m_maxZoom)
    m_zoom = m_maxZoom;
mainUICamera.zoom = m_zoom;
// este metodo reposiciona la camara si se desplaza fuera
    de los limites
displaceCameraIfNeeded();
mainUICamera.update();

// Dibujado de Mapa: Dibujamos la stage que contiene el
    mapa de fondo y también las fronteras de los
poligonos
stageMap.act(Gdx.graphics.getDeltaTime());
stageMap.draw();

// DIBUJADO DE POLIGONOS: Barremos todos los territorios
    del juego y dibujamos su area, del color
    acorde a la diplomacia y si esta seleccionado, etc.
    Añadir que cada territorio tiene definido el sprite de
    los poligonos que le corresponden desde la
    inicialización, ahorrando carga computacional en este
    punto, que solo tendrá que obtenerlo y dibujarlo.
m_polyBatch.setProjectionMatrix(mainUICamera.combined);
m_polyBatch.begin();
// Poligono seleccionado: Blanco
if (m_selectedTerritory != null)
    for (Polygon poly :
        m_selectedTerritory.getTerritoryPolygonList())
    {

        m_selectedTerritory.getPolygonSprite(poly)
            .setColor(1, 1, 1, 0.5f);
        m_selectedTerritory.getPolygonSprite(poly)
            .draw(m_polyBatch);
    }

```

```

(...)
if (m_renderTargetCountry != null)
{
    for (Territory terry : m_territoryList.values())
    for (Polygon poly:terry.getTerritoryPolygonList())
    {
        // PLAYER: AZUL
        if (terry.getOwner()
            .equals(m_renderTargetCountry))
        {
            terry.getPolygonSprite(poly).setColor(0,
                0, 1, 0.3f);
            terry.getPolygonSprite(poly)
                .draw(m_polyBatch);
        }
        else if (terry.getOwner()
            .isAlliedWith(m_renderTargetCountry))
            // ALIADO: VERDE FUERTE
        {
            terry.getPolygonSprite(poly).setColor(0,
                1, 0, 0.4f);
            terry.getPolygonSprite(poly)
                .draw(m_polyBatch);
        }
        else if (terry.getOwner()
            .isFriendlyWith(m_renderTargetCountry))
            // AMISTOSO: VERDE
        {
            terry.getPolygonSprite(poly).setColor(0,
                1, 0, 0.2f);

            terry.getPolygonSprite(poly)
                .draw(m_polyBatch);
        }
        else if (terry.getOwner()
            .isNeutralWith(m_renderTargetCountry))
            // NEUTRAL: AMARILLO
        {
            terry.getPolygonSprite(poly).setColor(1,
                1, 0, 0.2f);

            terry.getPolygonSprite(poly)
                .draw(m_polyBatch);
        }
        else if
        (terry.getOwner()
            .isHostileWith(m_renderTargetCountry))
            // HOSTIL: ROJO
        {
            terry.getPolygonSprite(poly).setColor(1,
                0, 0, 0.2f);

            terry.getPolygonSprite(poly)
                .draw(m_polyBatch);
        }
    }
}

```

```

m_polyBatch.end();

// Dibujamos path de tropas del jugador
for (int i = 0; i < m_troopSquads.size(); i++)
{
    TroopSquad squad = m_troopSquads.get(i);
    (...)
    squad.getStagePath().act(Gdx.graphics.getDeltaTime());
    squad.getStagePath().draw();
}

// Dibujamos stage de animaciones
stageAnimations.act(Gdx.graphics.getDeltaTime());
stageAnimations.draw();

// Acciones de las tropas, mover si no esta parado el
// juego a una velocidad dependiente de la
// velocidad del juego
if (m_gameSpeed != GameSpeed.SPEED_STOP)
    stageSquads.act(Gdx.graphics.getDeltaTime() *
        (m_secondsPerDayHigh / m_secondsPerDay));
// Pequeño hack para mover las tropas a la velocidad
// adecuada segun speed
stageSquads.draw();
(...)

// Dibujado de animaciones de texto
stageLabelAnimations.act(Gdx.graphics.getDeltaTime());
stageLabelAnimations.draw();

// Update de breaking news (es un actor custom)
// Breaking news
if (m_player != null)
{
    m_spriteBatch.begin();
    m_breakingNewsTextField.draw(m_spriteBatch, 1);
    if (m_gameSpeed != GameSpeed.SPEED_STOP) // No mover
                                                si pausado

        m_breakingNews.update(Gdx.graphics.getDeltaTime());
        m_breakingNews.draw(m_spriteBatch, 1);
    m_spriteBatch.end();
}

// Dibujado de UI global
stageUI.act(Gdx.graphics.getDeltaTime());
stageUI.draw();

// Drag de stage
dragSquadsStage.act(Gdx.graphics.getDeltaTime());
dragSquadsStage.draw();

// Dibujado de UI train
stageUITrain.act(Gdx.graphics.getDeltaTime());
stageUITrain.draw();

// Dibujado de UI tech
stageUITech.act(Gdx.graphics.getDeltaTime());
stageUITech.draw();

```

```

// Dibujado de UI diplo
stageUIDiplo.act(Gdx.graphics.getDeltaTime());
stageUIDiplo.draw();

// Dibujado de dialogos
stageDialog.act(Gdx.graphics.getDeltaTime());
stageDialog.draw();

// Dibujado de opciones
stageUIOptions.act(Gdx.graphics.getDeltaTime());
stageUIOptions.draw();
(...)
Funciones de dibujo de elementos de debug
(...)
// Update del juego, incrementa un Contador y cuando llega
// a un valor que depende de la velocidad llama al update
// No actualizar si esta parado
if (m_secondsPerDay != -1)
{
    if (m_tickCounter >= m_secondsPerDay)
    {
        // Restablecemos y actualizamos
        m_tickCounter = 0;
        m_updateThread.updateGame();
    }
    else
        m_tickCounter += Gdx.graphics.getDeltaTime();
}
break;
}
}

```

Observaciones del diseño:

- El dibujo es secuencial, de modo que las capas dibujadas anteriores son superpuestas por las ultimas si ocupan las mismas posiciones.
- El movimiento de escuadrones de tropas no se ejecuta en los updates, sino que debe procesarse en cada render, porque en otro caso, si parasesmos el juego, las tropas seguirían moviéndose al haber sido iniciado el movimiento en el update previo.
- Los dibujados de debug no han sido copiados en el fragmento porque no son necesarios, básicamente consisten en el dibujo de nodos y valores de pathing para comprobar que el algoritmo funciona correctamente y que se comporta de forma realista ante modificaciones de los costes.
- Esta función se llama desde un thread independiente con una frecuencia equivalente a los fotogramas por segundo del juego, el framework adapta este valor al consumo de CPU asumible. Esto convierte a esta función en una de las que más capacidad computacional requieren y por tanto simplificarla y optimizarla ha sido uno de los objetivos principales de todo el proyecto, alcanzando un estado final bastante aceptable en este sentido.

3.5 INPUTS DE USUARIO

Como ya explicamos en la sección de la inicialización, los inputs de usuario se asignan durante la misma a un multiplexor de inputs, del siguiente modo:

```
private void initializeInputProcessors()
{
    InputMultiplexer = new InputMultiplexer();
    Gdx.input.setInputProcessor(inputMultiplexer);

    // Instanciamos clases de inputs
    MyInputProcessor backGroundInputProcessor = new
        MyInputProcessor();
    MyGestureHandler gestureInputProcessor = new MyGestureHandler();

    // Add los procesos a la cola de inputs
    inputMultiplexer.addProcessor(stageUIOptions);
    inputMultiplexer.addProcessor(stageDialog);
    inputMultiplexer.addProcessor(stageUIDiplo);
    inputMultiplexer.addProcessor(stageUITech);
    inputMultiplexer.addProcessor(stageUITrain);
    inputMultiplexer.addProcessor(dragSquadsStage);
    inputMultiplexer.addProcessor(stageUI);
    inputMultiplexer.addProcessor(stageSquadTables);
    inputMultiplexer.addProcessor(stageSquads);
    inputMultiplexer.addProcessor(stageMap);
    inputMultiplexer.addProcessor(new
        GestureDetector(gestureInputProcessor));
    inputMultiplexer.addProcessor(backGroundInputProcessor);
}
```

El orden de adición al multiplexor determina la prioridad del evento, a continuación, se muestran dos fragmentos de listeners de distintos actores para ejemplificar lo dicho:

```
saveAndExitButton = new TextButton("Save/Exit", m_styleMainMenu);
saveAndExitButton.addListener(new InputListener() {
    @Override
    public boolean touchDown(InputEvent event, float x, float y, int
        pointer, int button)
    {
        // BOTON DE NUEVA PARTIDA

        // Sonido
        playSound(m_buttonClick_sound, 0.5f);

        // Guardar y salir
        saveGame(true);

        return true;
    }
});
```

```

// Boton
TextButton okBut = new TextButton("ok", defaultSkin, "default");
okBut.addListener(new InputListener() {
    @Override
    public boolean touchDown(InputEvent event, float x, float y, int
        pointer, int button) {
        // Sonido
        Game.playSound(m_buttonClick_sound, 0.5f);

        // Limpiamos
        stageDialog.clear();

        // Cerramos dialogo
        dialog.hide();
        return true;
    }
});

```

El botón saveAndExitButton se asignará a la stage stageUIOptions, mientras que el botón okBut se asignará a la stage stageDialog, podemos observar en la asignación multiplexada que stageUIOptions se añade primero, esto implica que si en el juego, dos capas stage dibujadas tuvieran los dos botones superpuestos, la capa superior (stageUIOptions) procesaría el evento touchDown y nunca llegaría a activarse el listener de la capa inferior. Todos los inputs de elementos de interfaz en stages como botones se han implementado siguiendo el esquema anterior.

Por otra parte, la clase inputListener implementa los siguientes métodos:

```

/** Called when a mouse button or a finger touch goes down on the
    actor. If true is returned, this listener will receive all
    * touchDragged and touchUp events, even those not over this actor,
    until touchUp is received. Also when true is returned, the
    * event is {@link Event#handle() handled}.
    * @see InputEvent */
public boolean touchDown (InputEvent event, float x, float y, int
    pointer, int button) {
    return false;
}

/** Called when a mouse button or a finger touch goes up anywhere, but
    only if touchDown previously returned true for the mouse
    * button or touch. The touchUp event is always {@link Event#handle()
    handled}.
    * @see InputEvent */
public void touchUp (InputEvent event, float x, float y, int pointer,
    int button) {
}

```

```

/** Called when a mouse button or a finger touch is moved anywhere,
but only if touchDown previously returned true for the mouse
* button or touch. The touchDragged event is always {@link
Event#handle() handled}.
* @see InputEvent */
public void touchDragged (InputEvent event, float x, float y, int
pointer) {
}

/** Called any time the mouse is moved when a button is not down. This
event only occurs on the desktop. When true is returned,
* the event is {@link Event#handle() handled}.
* @see InputEvent */
public boolean mouseMoved (InputEvent event, float x, float y) {
    return false;
}

/** Called any time the mouse cursor or a finger touch is moved over
an actor. On the desktop, this event occurs even when no
* mouse buttons are pressed (pointer will be -1).
* @param fromActor May be null.
* @see InputEvent */
public void enter (InputEvent event, float x, float y, int pointer,
Actor fromActor) {
}

/** Called any time the mouse cursor or a finger touch is moved out of
an actor. On the desktop, this event occurs even when no
* mouse buttons are pressed (pointer will be -1).
* @param toActor May be null.
* @see InputEvent */
public void exit (InputEvent event, float x, float y, int pointer,
Actor toActor) {
}

/** Called when the mouse wheel has been scrolled. When true is
returned, the event is {@link Event#handle() handled}. */
public boolean scrolled (InputEvent event, float x, float y, int
amount) {
    return false;
}

/** Called when a key goes down. When true is returned, the event is
{@link Event#handle() handled}. */
public boolean keyDown (InputEvent event, int keycode) {
    return false;
}

/** Called when a key goes up. When true is returned, the event is
{@link Event#handle() handled}. */
public boolean keyUp (InputEvent event, int keycode) {
    return false;
}

/** Called when a key is typed. When true is returned, the event is
{@link Event#handle() handled}. */
public boolean keyTyped (InputEvent event, char character) {
    return false;
}

```

Dichos métodos serán llamados desde distintos elementos de interfaz que requieran acciones asignadas a esos inputs, como botones.

No obstante, esta clase no nos bastará para implementar todos los inputs de usuario, ya que no soporta gestos sobre la pantalla táctil, para ello deberemos emplear la clase `GestureListener`, que hemos observado que es la última en prioridad y que implementará el movimiento de arrastre sobre el mapa (zoom de la cámara).

La clase `GestureListener` implementa los siguientes métodos:

```
/** @see InputProcessor#touchDown(int, int, int, int) */
public boolean touchDown (float x, float y, int pointer, int button);

/** Called when a tap occurred. A tap happens if a touch went down on
    the screen and was lifted again without moving outside
    * of the tap square. The tap square is a rectangular area around the
    initial touch position as specified on construction
    * time of the {@link GestureDetector}.
    * @param count the number of taps. */
public boolean tap (float x, float y, int count, int button);

public boolean longPress (float x, float y);

/** Called when the user dragged a finger over the screen and lifted
    it. Reports the last known velocity of the finger in
    * pixels per second.
    * @param velocityX velocity on x in seconds
    * @param velocityY velocity on y in seconds */
public boolean fling (float velocityX, float velocityY, int button);

/** Called when the user drags a finger over the screen.
    * @param deltaX the difference in pixels to the last drag event on x.
    * @param deltaY the difference in pixels to the last drag event on y.
    */
public boolean pan (float x, float y, float deltaX, float deltaY);

/** Called when no longer panning. */
public boolean panStop (float x, float y, int pointer, int button);

/** Called when the user performs a pinch zoom gesture. The original
    distance is the distance in pixels when the gesture
    * started.
    * @param initialDistance distance between fingers when the gesture
    started.
    * @param distance current distance between fingers. */
public boolean zoom (float initialDistance, float distance);

/** Called when a user performs a pinch zoom gesture. Reports the
    initial positions of the two involved fingers and their
    * current positions.
    * @param initialPointer1
    * @param initialPointer2
    * @param pointer1
    * @param pointer2 */
```

```
public boolean pinch (Vector2 initialPointer1, Vector2
initialPointer2, Vector2 pointer1, Vector2 pointer2);
```

Implementaremos dos de estos métodos para dar soporte al zoom de la cámara:

```
class MyGestureHandler implements GestureDetector.GestureListener
{
    public float initialScale = 1.0f;

    @Override
    public boolean touchDown(float x, float y, int pointer, int
        button)
    {
        initialScale = m_zoom;
        return false;
    }

    @Override
    public boolean zoom(float initialDistance, float distance)
    {
        //Calculate pinch to m_zoom
        float ratio = initialDistance / distance;

        //Clamp range and set m_zoom
        m_zoom = MathUtils.clamp(initialScale * ratio, m_maxZoom,
        1.0f);
        return true;
    }
    (...)
}
```

Además de esto implementaremos otra clase adicional llamada InputProcessor, que se asigna al multiplexer también ocupando el último nivel de prioridad de input, está hecho de este modo porque no se puede asignar a una stage en concreto, ya que trabaja con ubicaciones definidas por los nodos que no tienen un diseño de scene2D (stages/actores) y se emplea fundamentalmente para la traslación de la cámara y para detectar el país pulsado en base al nodo que haya en esa posición.

La clase `inputProcessor` define los siguientes métodos:

```
/** Called when a key was pressed
 *
 * @param keycode one of the constants in {@link Input.Keys}
 * @return whether the input was processed */
public boolean keyDown (int keycode);

/** Called when a key was released
 *
 * @param keycode one of the constants in {@link Input.Keys}
 * @return whether the input was processed */
public boolean keyUp (int keycode);

/** Called when a key was typed
 *
 * @param character The character
 * @return whether the input was processed */
public boolean keyTyped (char character);

/** Called when the screen was touched or a mouse button was pressed.
The button parameter will be {@link Buttons#LEFT} on iOS.
 * @param screenX The x coordinate, origin is in the upper left corner
 * @param screenY The y coordinate, origin is in the upper left corner
 * @param pointer the pointer for the event.
 * @param button the button
 * @return whether the input was processed */
public boolean touchDown (int screenX, int screenY, int pointer, int
button);

/** Called when a finger was lifted or a mouse button was released.
The button parameter will be {@link Buttons#LEFT} on iOS.
 * @param pointer the pointer for the event.
 * @param button the button
 * @return whether the input was processed */
public boolean touchUp (int screenX, int screenY, int pointer, int
button);

/** Called when a finger or the mouse was dragged.
 * @param pointer the pointer for the event.
 * @return whether the input was processed */
public boolean touchDragged (int screenX, int screenY, int pointer);

/** Called when the mouse was moved without any buttons being pressed.
Will not be called on iOS.
 * @return whether the input was processed */
public boolean mouseMoved (int screenX, int screenY);

/** Called when the mouse wheel was scrolled. Will not be called on
iOS.
 * @param amount the scroll amount, -1 or 1 depending on the direction
the wheel was scrolled.
 * @return whether the input was processed. */
public boolean scrolled (int amount);
```

Este sistema de inputs es general y no trabaja con actores, por lo que debemos leer directamente posiciones y operar con dichos valores, implementando los siguientes métodos:

```
class MyInputProcessor implements InputProcessor
{
    // Ultima posicion pulsada (de referencia para el drag)
    private Vector3 lastTouchPos = new Vector3(0, 0, 0);

    @Override
    // Usamos este evento para seleccionar territorio al clicar en
    // cualquier posición del mapa, si clickeasemos en la interfaz, el
    // evento input seria detectado desde una stage superior en el
    // multiplexer y estos eventos no se ejecutarían.
    public boolean touchDown(int screenX, int screenY, int pointer,
                             int button)
    {
        // Guardamos posicion del primer toque
        lastTouchPos.set(screenX, screenY, 0);

        // Detectamos el país pulsado
        Territory terry =
            getTerritoryByPositionFromTopLeftReference(new Vector2(screenX,
                                                                    screenY));

        if (terry != null) // Seleccionamos el territorio
            onTerritorySelect(terry);
        else
        {
            m_selectedTerritory = null;
            updateUI();
        }
        return true;
    }

    @Override
    // Usamos este evento para la traslación de la cámara, así como
    // comprobar si la sacamos de los límites del
    // mapa para reposicionarla.
    public boolean touchDragged(int screenX, int screenY, int pointer)
    {
        // Nueva posicion de la camara en funcion del punto pulsado
        Vector3 destPos = getNewCameraPosition(screenX, screenY);

        // Zoom de la camara, afectara a la velocidad de traslacion 4
        // para mantener la proporcion
        float cameraZoom = mainUICamera.zoom;

        // Si esta fuera del limite X pero NO del Y, desplazar solo en
        // el eje Y.
        if (isCameraOutOfLimitInDestPos(destPos)[0] &&
            !isCameraOutOfLimitInDestPos(destPos)[1])
            mainUICamera.translate(0, (destPos.y -
                                         mainUICamera.position.y) * cameraZoom, destPos.z -
                                     mainUICamera.position.z);

        // Si esta fuera del limite Y pero NO del X, desplazar solo en
        // el eje X.
        if (!isCameraOutOfLimitInDestPos(destPos)[0] &&
```

```

isCameraOutOfLimitInDestPos(destPos)[1])
    mainUICamera.translate((destPos.x -
        mainUICamera.position.x) * cameraZoom,
        0, destPos.z - mainUICamera.position.z);
// Si no esta fuera de ningun limite, desplazar en todos los
ejes.
if (!isCameraOutOfLimitInDestPos(destPos)[0] &&
!isCameraOutOfLimitInDestPos(destPos)[1])
    mainUICamera.translate((destPos.x -
        mainUICamera.position.x) * cameraZoom, (destPos.y -
        mainUICamera.position.y) * cameraZoom,
        destPos.z - mainUICamera.position.z);

// Establecer toque en ultima posicion
lastTouchPos.set(screenX, screenY, 0);

// No activar evento de drag
return false;
}

private Vector3 getNewCameraPosition(int x, int y)
{
    if (lastTouchPos == null)
        lastTouchPos = new Vector3(x, y, 0);
    Vector3 destPos = lastTouchPos;
    destPos.sub(x, y, 0);
    destPos.y = -destPos.y;
    destPos.add(mainUICamera.position);
    return destPos;
}

@Override
// Boton back del smartphone, mostrará el menu de opciones en
// lugar de abandonar el juego, lo cual ocurrirá
// si se pulsa dos veces.
public boolean keyDown(int keycode)
{
    if(keycode == Input.Keys.BACK)
    {
        setOptionsMenuVisibility(true);
    }
    return false;
}

// Devuelve true en cada coordenada si esta fuera X = 0; Y = 1
private boolean[] isCameraOutOfLimitInDestPos(Vector3 position)
{
    // Variables de salida
    boolean isXOutOfLimit = false;
    boolean isYOutOfLimit = false;

    // COMPROBACION PARA X
    // Posiciones limite
    int xLeftLimit = (int)(mainUICamera.zoom *
        (mainUICamera.viewportWidth / 2));
    int xRightLimit = (Gdx.graphics.getWidth()) - xLeftLimit;

```

```

// Posicion tomada desde la esquina suroeste
int positionX = (int)((Gdx.graphics.getWidth()) / 2 +
    position.x);

// Logica de comprobacion
if(positionX < xLeftLimit || positionX > xRightLimit)
    isXOutOfLimit = true;

// COMPROBACION PARA Y
// Posiciones limite
int yBottomLimit = (int)(mainUICamera.zoom *
    (mainUICamera.viewportHeight / 2));
int yTopLimit = Gdx.graphics.getHeight() - yBottomLimit;

// Posicion tomada desde la esquina suroeste
int positionY = (int)(Gdx.graphics.getHeight() / 2 +
    position.y);

// Logica de comprobacion
if( positionY < yBottomLimit || positionY > yTopLimit )
    isYOutOfLimit = true;

return new boolean[]{isXOutOfLimit,isYOutOfLimit};
}
(...)
}

```

Además de todo lo anterior empleamos la clase DragAndDrop para implementar actores que puedan ser arrastrados sobre distintas stages y ejecutar eventos cuando esto ocurre, como se muestra en los siguientes ejemplos relacionados con el arrastre de tropas desde la interfaz de deploy al mapa y el arrastre de las propias tropas en el mapa de un punto a otro para iniciar un movimiento.

```

// Drag de la squad de la interfaz deploy al mapa
DragAndDrop dragAndDrop = new DragAndDrop();
dragAndDrop.addSource(new Source(m_squadPinImage)
{
    // Evento que se llama cuando se inicia el drag.
    public Payload dragStart(InputEvent event, float x, float y, int
        pointer)
    {
        // Si no tiene tropas, no permitimos el drag
        TroopArray troops = m_selectedTerritory.getDeployedTroops();
        if (!troops.hasTroops()) {
            showDialog("There are no troops to deploy.");
            return null;
        }

        Payload payload = new Payload();
        Image squadIcon = new Image(getSpriteByName("squadPinMove"));
        squadIcon.setSize(m_squadPinImage.getWidth() / 2,
            m_squadPinImage.getHeight() / 2);
    }
}

```

```

        squadIcon.setColor(1, 1, 1, 0.5f);
        payload.setDragActor(squadIcon);
        return payload;
    }

    // Evento que se llama cuando se suelta el drag y las tropas se
    // dropean.
    public void dragStop(InputEvent event, float x, float y, int
        pointer, Payload payload, Target target)
    {
        // Territorio destino
        Territory terryDest =
            getTerritoryByPositionFromTopLeftReference(new
                Vector2(x + m_squadPinImage.getX(), windowH - (y +
                    m_squadPinImage.getY())));

        // Cubrimos excepciones
        if (terryDest == null) {
            showDialog("You must deploy your troops on its own
                territory.");
            return;
        }
        if (!terryDest.equals(m_selectedTerritory)) {
            showDialog("You must deploy your troops on its own
                territory.");
            return;
        }

        (...)

        // Deploy troops (solo si hay)
        TroopArray troops = terryDest.getDeployedTroops();
        if (!troops.hasTroops())
            return;
        deployTroops(troops, terryDest, coordX, coordY,
            SquadType.SQUAD_LAND);

        // Actualizamos interfaz
        updateUI();

        return;
    }
});

// DragAndDrop de las squads sobre otros puntos del mapa
DragAndDrop dragAndDrop = new DragAndDrop();
(...)
dragAndDrop.addSource(new Source(squad)
{
    // Evento que se llama cuando se inicia el drag.
    public Payload dragStart(InputEvent event, float x, float y, int
        pointer)
    {
        // El player solo puede arrastrar sus tropas
        if (squad.getOwner() != m_player)
            return null;
    }
});

```

```

    Payload payload = new Payload();
    Image squadIcon = new Image(getSpriteByName("squadPinMove"));
    squadIcon.setSize(squad.getWidth() / 2, squad.getHeight() /
                                                              2);

    payload.setDragActor(squadIcon);
    return payload;
}

// Evento que se llama cuando se suelta el drag y las tropas se
// devuelven a la interfaz si se han arrastrado
// sobre ella o se desplazan a otro punto del mapa en caso
// contrario.
public void dragStop(InputEvent event, float x, float y, int
pointer, Payload payload, Target target)
{
    // El player solo puede arrastrar sus tropas
    if (squad.getOwner() != m_player)
        return;

    // Posicion devuelta es relativa al actor original, calculamos
    // relativa al 0,0 (referencia de poligonos)
    float destX = squad.getX() + x;
    float destY = squad.getY() + y;

    (...)

    // Draw sobre UI
    // El territorio seleccionado es mio, asi que su interfaz ha
    // de ser visible
    if (m_selectedTerritory != null)
        if (m_selectedTerritory.getOwner().equals(m_player))
            if (destY < mainUICamera.position.y -
                    mainUICamera.viewportHeight / 2 *
                    mainUICamera.zoom +
                    m_botTable.getHeight() *
                    mainUICamera.zoom)
            {
                if (squad.getTerritory() == null)
                    showDialog("The squad you want to retract its
                                not at a territory.");
                else if (!squad.getTerritory()
                        .equals(m_selectedTerritory))
                    showDialog("The squad you want to retract its
                                not at the selected territory.");
                else
                {
                    // Llegados a este punto, replegamos tropas
                    squad.retractSquadToTerritory
                        (m_selectedTerritory, true);

                    // Hack: Forzamos un update de la UI desde
                    // entorno estático.
                    m_forceUpdate = true;
                    return;
                }
            }
}

```

```

// El player mueve troop squad
if (squad.getType() == SquadType.SQUAD_LAND)
    playSound(m_sendTroops_sound, 0.3f);
else if (squad.getType() == SquadType.SQUAD_AIR)
    playSound(m_sendAirCraft_sound, 0.3f);

// Desde la interfaz humana siempre en linea recta
squad.sendSquad(new Vector2(destX, destY), true);
return;
}
});

```

3.6 PAISES, ALIANZAS Y TERRITORIOS

Como hemos visto anteriormente, tanto países como territorios se inicializan en `initializeCountries` e `initializeTerritories` respectivamente, en las siguientes tablas se pueden encontrar los datos de todos los territorios y países instanciados en el juego:

Territorios:

Id	Nombre	Producción	Polígono
2	Norway	266	no
4	Switzerland	240	ch
5	Australia	168	au
6	Denmark	166	dk
7	Sweden	161	se
10	United States	150	us
11	Ireland	147	ie
12	Netherlands	141	nl
13	Austria	141	at
14	Iceland	141	is
15	Canada	139	ca
16	Finland	136	fi
17	Belgium	131	be
18	Germany	14	de
19	United Kingdom	126	gb
20	France	123	fr
21	New Zealand	121	nz
22	UA Emirates	12	ae
23	Kuwait	119	kw
26	Japan	100	jp
27	Italy	99	it
28	Spain	83	es
30	South Korea	77	kr

31	Cyprus	72	cy
33	Saudi Arabia	67	sa
35	Slovenia	66	si
36	Taiwan	62	tw
37	Portugal	7	pt
38	Greece	60	gr
39	Trinidad Tobago	59	tt
40	Estonia	54	ee
41	Czech Republic	54	cz
42	Oman	53	om
43	Slovakia	51	sk
44	E. Guinea	51	gq
45	Lithuania	45	lt
46	Uruguay	45	uy
47	Latvia	44	lv
51	Chile	40	cl
52	Poland	40	pl
55	Hungary	39	hu
56	Croatia	37	hr
57	Russia	36	ru
58	Argentina	36	ar
59	Kazakhstan	34	kz
60	Brazil	32	br
61	Panama	31	pa
62	Lebanon	31	lb
63	Gabon	30	ga
64	Malaysia	30	my
65	Mexico	30	mx
67	Turkey	29	tr
68	Costa Rica	28	cr
69	Romania	28	ro
70	Suriname	27	sr
72	Turkmenistan	23	tm
74	Colombia	23	co
75	Belarus	23	by
77	Azerbaijan	22	az
78	Bulgaria	22	bg
79	China	21	cn
80	Botswana	21	bw
83	Venezuela	19	ve
85	Libya	19	ly

86	South Africa	18	za
87	Peru	18	pe
88	Ecuador	18	ec
89	Iraq	17	iq
90	Serbia	17	rs
91	Namibia	17	na
92	Dominican R	17	doo
93	Algeria	16	dz
94	Macedonia	16	mk
95	Thailand	15	th
96	Jordan	15	jo
97	Angola	15	ao
98	Iran	15	ir
99	Jamaica	14	jm
100	Albania	14	al
102	Belize	13	bz
103	Bosnia	13	ba
105	Tunisia	13	tn
107	Paraguay	12	py
108	Mongolia	12	mn
109	El Salvador	11	sv
111	Guatemala	11	gt
112	Guyana	11	gy
113	Georgia	11	ge
116	Sri Lanka	10	lk
118	Indonesia	10	id
120	Swaziland	10	sz
121	Egypt	10	eg
122	Nigeria	10	ng
123	Morocco	10	ma
124	Congo A	9	cd
125	Armenia	9	am
127	Bolivia	9	bo
129	Philippines	8	ph
130	Bhutan	1	bt
131	Honduras	7	hn
132	Moldova	7	md
133	New Guinea	6	pg
134	Vietnam	6	vn
135	Uzbekistan	6	uz
137	Sudan	1	sd
138	Nicaragua	6	ni
139	Zambia	5	zm

140	Laos	5	la
141	Djibouti	5	dj
143	India	5	in
144	Yemen	5	ye
145	Ivory Coast	5	ci
146	Ghana	5	gh
147	Kenya	4	ke
148	Cameroon	4	cm
149	Mauritania	4	mr
150	Pakistan	4	pk
151	Kyrgyzstan	4	kg
152	Chad	4	td
154	Bangladesh	4	bd
155	Lesotho	1	ls
157	Tajikistan	4	tj
158	Cambodia	3	kh
159	Senegal	3	sn
160	Zimbabwe	3	zw
161	Tanzania	3	tz
163	Haiti	3	ht
164	Benin	3	bj
165	Sierra Leone	3	sl
166	Mali	3	ml
167	Uganda	2	ug
169	Burkina Faso	2	bf
170	Nepal	2	np
171	Togo	2	tg
172	Afghanistan	2	af
173	Mozambique	2	mz
174	Eritrea	2	er
175	Guinea-Bissau	2	gw
176	North Korea	2	kp
177	Ethiopia	2	et
178	Guinea	2	gn
179	Liberia	2	lr
180	Niger	2	ne
181	Madagascar	2	mg
182	Congo B	2	cd
183	Gambia	2	gm
184	African R	2	cf
185	Burundi	1	bi
186	Malawi	1	mw

187	Greenland	62	gl
193	Congo C	17	cg
194	Cuba	28	cu
196	Western Sahara	7	eh
204	Syria	14	sy
205	Somalia	1	so
207	Puerto Rico	78	pr
211	Israel	12	il
213	Myanmar	15	mm
218	Ruanda	5	rw
219	French Guiana	23	gf
220	Montenegro	29	me
221	Ukraine	9	ua

Países:

Id	Nombre	Cultura	Territorio Base (ID)	Dinero inicial	Tipo de IA
2	Norway	WESTERN	2	2658	DEFENSIVE
4	Switzerland	WESTERN	4	2397	PEACEFUL
5	Australia	WESTERN	5	1678	DEFENSIVE
6	Denmark	WESTERN	6	1660	PEACEFUL
7	Sweden	WESTERN	7	1603	DEFENSIVE
10	United States	WESTERN	10	1496	IMPERIALIST
11	Ireland	WESTERN	11	1465	DEFENSIVE
12	Netherlands	WESTERN	12	1408	DEFENSIVE
13	Austria	WESTERN	13	1406	DEFENSIVE
14	Iceland	WESTERN	14	1405	DEFENSIVE
15	Canada	WESTERN	15	1381	DEFENSIVE
16	Finland	WESTERN	16	1357	DEFENSIVE
17	Belgium	WESTERN	17	1308	DEFENSIVE
18	Germany	WESTERN	18	131	IMPERIALIST
19	United Kingdom	WESTERN	19	1251	IMPERIALIST
20	France	WESTERN	20	1221	IMPERIALIST
21	New Zealand	WESTERN	21	1202	PEACEFUL
22	UA Emirates	ARABIC	22	119	DEFENSIVE
23	Kuwait	ARABIC	23	1181	DEFENSIVE
26	Japan	EASTERN	26	996	IMPERIALIST
27	Italy	WESTERN	27	982	DEFENSIVE
28	Spain	WESTERN	28	830	DEFENSIVE

30	South Korea	WESTERN	30	770	DEFENSIVE
31	Cyprus	ARABIC	31	716	PEACEFUL
33	Saudi Arabia	ARABIC	33	670	DEFENSIVE
35	Slovenia	WESTERN	35	659	DEFENSIVE
36	Taiwan	EASTERN	36	620	DEFENSIVE
37	Portugal	WESTERN	37	61	DEFENSIVE
38	Greece	WESTERN	38	594	DEFENSIVE
39	Trinidad Tobago	WESTERN	39	584	DEFENSIVE
40	Estonia	WESTERN	40	539	DEFENSIVE
41	Czech Republic	WESTERN	41	536	DEFENSIVE
42	Oman	ARABIC	42	521	DEFENSIVE
43	Slovakia	WESTERN	43	506	DEFENSIVE
44	E. Guinea	AFRICAN	44	504	VARIABLE
45	Lithuania	WESTERN	45	449	DEFENSIVE
46	Uruguay	WESTERN	46	444	DEFENSIVE
47	Latvia	WESTERN	47	433	DEFENSIVE
51	Chile	WESTERN	51	397	DEFENSIVE
52	Poland	WESTERN	52	394	DEFENSIVE
55	Hungary	WESTERN	55	381	DEFENSIVE
56	Croatia	WESTERN	56	370	DEFENSIVE
57	Russia	WESTERN	57	355	IMPERIALIST
58	Argentina	WESTERN	58	353	DEFENSIVE
59	Kazakhstan	ARABIC	59	334	DEFENSIVE
60	Brazil	WESTERN	60	318	DEFENSIVE
61	Panama	WESTERN	61	306	PEACEFUL
62	Lebanon	ARABIC	62	304	AGGRESSIVE
63	Gabon	AFRICAN	63	297	VARIABLE
64	Malaysia	EASTERN	64	296	DEFENSIVE
65	Mexico	WESTERN	65	294	DEFENSIVE
67	Turkey	ARABIC	67	288	AGGRESSIVE
68	Costa Rica	WESTERN	68	277	DEFENSIVE
69	Romania	WESTERN	69	275	DEFENSIVE
70	Suriname	WESTERN	70	263	DEFENSIVE
72	Turkmenistan	ARABIC	72	227	DEFENSIVE
74	Colombia	WESTERN	74	222	DEFENSIVE
75	Belarus	WESTERN	75	221	DEFENSIVE
77	Azerbaijan	ARABIC	77	217	DEFENSIVE
78	Bulgaria	WESTERN	78	213	DEFENSIVE
79	China	EASTERN	79	208	IMPERIALIST

80	Botswana	AFRICAN	80	206	VARIABLE
83	Venezuela	WESTERN	83	186	DEFENSIVE
85	Libya	ARABIC	85	182	DEFENSIVE
86	South Africa	AFRICAN	86	178	DEFENSIVE
87	Peru	WESTERN	87	177	DEFENSIVE
88	Ecuador	WESTERN	88	173	DEFENSIVE
89	Iraq	ARABIC	89	169	AGGRESSIVE
90	Serbia	WESTERN	90	168	DEFENSIVE
91	Namibia	AFRICAN	91	167	DEFENSIVE
92	Dominican R	WESTERN	92	166	DEFENSIVE
93	Algeria	ARABIC	93	152	DEFENSIVE
94	Macedonia	WESTERN	94	151	DEFENSIVE
95	Thailand	EASTERN	95	150	DEFENSIVE
96	Jordan	ARABIC	96	147	DEFENSIVE
97	Angola	AFRICAN	97	145	VARIABLE
98	Iran	ARABIC	98	142	AGGRESSIVE
99	Jamaica	WESTERN	99	135	PEACEFUL
100	Albania	WESTERN	100	131	DEFENSIVE
102	Belize	WESTERN	102	130	DEFENSIVE
103	Bosnia	WESTERN	103	128	DEFENSIVE
105	Tunisia	ARABIC	105	121	DEFENSIVE
107	Paraguay	WESTERN	107	118	DEFENSIVE
108	Mongolia	EASTERN	108	113	DEFENSIVE
109	El Salvador	WESTERN	109	110	DEFENSIVE
111	Guatemala	WESTERN	111	105	DEFENSIVE
112	Guyana	WESTERN	112	103	DEFENSIVE
113	Georgia	ARABIC	113	102	DEFENSIVE
116	Sri Lanka	EASTERN	116	98	DEFENSIVE
118	Indonesia	EASTERN	118	97	DEFENSIVE
120	Swaziland	AFRICAN	120	92	DEFENSIVE
121	Egypt	ARABIC	121	91	DEFENSIVE
122	Nigeria	AFRICAN	122	91	DEFENSIVE
123	Morocco	ARABIC	123	91	DEFENSIVE
124	Congo A	AFRICAN	124	87	AGGRESSIVE
125	Armenia	ARABIC	125	86	DEFENSIVE
127	Bolivia	WESTERN	127	84	DEFENSIVE
129	Philippines	EASTERN	129	79	PEACEFUL
130	Bhutan	EASTERN	130	8	DEFENSIVE
131	Honduras	WESTERN	131	65	DEFENSIVE
132	Moldova	WESTERN	132	62	DEFENSIVE
133	New Guinea	WESTERN	133	59	DEFENSIVE

134	Vietnam	EASTERN	134	57	DEFENSIVE
135	Uzbekistan	ARABIC	135	57	DEFENSIVE
137	Sudan	AFRICAN	137	6	AGGRESSIVE
138	Nicaragua	WESTERN	138	52	DEFENSIVE
139	Zambia	AFRICAN	139	49	VARIABLE
140	Laos	EASTERN	140	47	DEFENSIVE
141	Djibouti	AFRICAN	141	47	VARIABLE
143	India	EASTERN	143	45	DEFENSIVE
144	Yemen	ARABIC	144	44	DEFENSIVE
145	Ivory Coast	AFRICAN	145	41	VARIABLE
146	Ghana	AFRICAN	146	41	VARIABLE
147	Kenya	AFRICAN	147	39	AGGRESSIVE
148	Cameroon	AFRICAN	148	39	VARIABLE
149	Mauritania	AFRICAN	149	39	VARIABLE
150	Pakistan	ARABIC	150	37	DEFENSIVE
151	Kyrgyzstan	ARABIC	151	36	DEFENSIVE
152	Chad	AFRICAN	152	34	DEFENSIVE
154	Bangladesh	EASTERN	154	33	DEFENSIVE
155	Lesotho	AFRICAN	155	4	DEFENSIVE
157	Tajikistan	ARABIC	157	31	DEFENSIVE
158	Cambodia	EASTERN	158	30	DEFENSIVE
159	Senegal	AFRICAN	159	30	VARIABLE
160	Zimbabwe	AFRICAN	160	29	VARIABLE
161	Tanzania	AFRICAN	161	28	DEFENSIVE
163	Haiti	WESTERN	163	23	DEFENSIVE
164	Benin	AFRICAN	164	23	DEFENSIVE
165	Sierra Leone	AFRICAN	165	23	AGGRESSIVE
166	Mali	AFRICAN	166	21	AGGRESSIVE
167	Uganda	AFRICAN	167	20	VARIABLE
169	Burkina Faso	AFRICAN	169	20	DEFENSIVE
170	Nepal	EASTERN	170	20	DEFENSIVE
171	Togo	AFRICAN	171	19	VARIABLE
172	Afghanistan	ARABIC	172	18	DEFENSIVE
173	Mozambique	AFRICAN	173	18	DEFENSIVE
174	Eritrea	AFRICAN	174	17	DEFENSIVE
175	Guinea-Bissau	AFRICAN	175	17	DEFENSIVE
176	North Korea	EASTERN	176	16	AGGRESSIVE
177	Ethiopia	AFRICAN	177	16	DEFENSIVE

178	Guinea	AFRICAN	178	16	DEFENSIVE
179	Liberia	AFRICAN	179	14	AGGRESSIVE
180	Niger	AFRICAN	180	13	DEFENSIVE
181	Madagascar	AFRICAN	181	13	DEFENSIVE
182	Congo B	AFRICAN	182	12	AGGRESSIVE
183	Gambia	AFRICAN	183	12	DEFENSIVE
184	African R	AFRICAN	184	11	VARIABLE
185	Burundi	AFRICAN	185	10	DEFENSIVE
186	Malawi	AFRICAN	186	7	DEFENSIVE
187	Greenland	WESTERN	187	617	PEACEFUL
193	Congo C	AFRICAN	193	170	DEFENSIVE
194	Cuba	WESTERN	194	280	DEFENSIVE
196	Western Sahara	AFRICAN	196	70	DEFENSIVE
204	Syria	ARABIC	204	140	AGGRESSIVE
205	Somalia	AFRICAN	205	10	DEFENSIVE
207	Puerto Rico	WESTERN	207	780	DEFENSIVE
211	Israel	ARABIC	211	120	AGGRESSIVE
213	Myanmar	EASTERN	213	150	DEFENSIVE
218	Ruanda	AFRICAN	218	50	DEFENSIVE
219	French Guiana	WESTERN	219	230	DEFENSIVE
220	Montenegro	WESTERN	220	290	DEFENSIVE
221	Ukraine	WESTERN	221	84	DEFENSIVE

Sobre estos países además se establecerán relaciones diplomáticas iniciales basadas en alianzas territoriales reales que se definirán como una lista de países y un valor diplomático que los relacionará a todos, las alianzas creadas son las siguientes:

Valor	Alianza	Países miembros
80	otan	Belgium, Canada, Denmark, France, Iceland, Italy, Luxembourg, Netherlands, Norway, Portugal, United Kingdom, United States, Greece, Turkey, Germany, Spain, Czech Republic, Hungary, Poland, Bulgaria, Estonia, Latvia, Lithuania, Romania, Slovakia, Slovenia, Albania, Croatia
70	abca	United States, United Kingdom, Canada, Australia
75	zpcas	Angola, Argentina, Benin, Brazil, Cameroon, Cape Verde, Congo, Congo, Ivory Coast, Equatorial Guinea, Gabon, Gambia, Ghana, Guinea, Guinea-Bissau, Liberia, Namibia, Nigeria, Senegal, Sierra Leone, South Africa, Togo, Uruguay
70	tiar	Argentina, Bahamas, Brazil, Chile, Colombia, Costa Rica, Dominican R, El Salvador, Guatemala, Haiti, Honduras, Panama, Paraguay, Peru, Trinidad Tobago, United States, Uruguay, Bolivia, Ecuador, Nicaragua, Mexico, Venezuela
65	anzus	United States, New Zealand, Australia
65	tra	United States, Taiwan
65	sk_us	United States, South Korea
65	ph_us	United States, Philippines
65	anpo	United States, Japan
62	matf	United States, Morocco
80	arab_league	Algeria, Comoros, Djibouti, Egypt, Iraq, Jordan, Kuwait, Lebanon, Libya, Mauritania, Morocco, Oman, Saudi Arabia, Somalia, Sudan, Tunisia, UA Emirates, Yemen
74	european_union	Austria, Belgium, Bulgaria, Croatia, Cyprus, Czech Republic, Denmark, Estonia, Finland, France, Germany, Greece, Hungary, Ireland, Italy, Latvia, Lithuania, Luxembourg, Netherlands, Poland, Portugal, Romania, Slovakia, Slovenia, Spain, Sweden, United Kingdom
65	otsc	Armenia, Belarus, Kazakhstan, Kyrgyzstan, Russia, Tajikistan
70	ocs	China, Kazakhstan, Kyrgyzstan, Russia, Tajikistan, Uzbekistan, Pakistan, India
70	takm	Turkey, Azerbaijan, Kyrgyzstan, Mongolia

Algunos de los países anteriores se fusionarán en un nuevo país para simplificar el juego y eliminar microestados con un tamaño demasiado pequeño para el juego, los países que se fusionan en un nuevo país pueden verse rápidamente en el siguiente fragmento de código:

```
public void initializeCountriesFusion()
{
    // America central
    ArrayList<Country> americaCentral = new ArrayList<Country>();
    americaCentral.add(getCountryByName("Guatemala"));
    americaCentral.add(getCountryByName("Belize"));
    americaCentral.add(getCountryByName("El Salvador"));
    americaCentral.add(getCountryByName("Honduras"));
    americaCentral.add(getCountryByName("Nicaragua"));
    americaCentral.add(getCountryByName("Costa Rica"));
    americaCentral.add(getCountryByName("Panama"));
    fusionateCountries("Middle America", americaCentral);

    // Cuba e islas cercanas
    ArrayList<Country> cuba = new ArrayList<Country>();
    cuba.add(getCountryByName("Cuba"));
    cuba.add(getCountryByName("Haiti"));
    cuba.add(getCountryByName("Dominican R"));
    cuba.add(getCountryByName("Puerto rico"));
    cuba.add(getCountryByName("Jamaica"));
    fusionateCountries("Cuba", cuba);

    // Venezuela e islas
    ArrayList<Country> ven = new ArrayList<Country>();
    ven.add(getCountryByName("Venezuela"));
    ven.add(getCountryByName("Trinidad Tobago"));
    fusionateCountries("Venezuela", ven);

    // Laterales de venezuela
    ArrayList<Country> guianas = new ArrayList<Country>();
    guianas.add(getCountryByName("Guyana"));
    guianas.add(getCountryByName("Suriname"));
    guianas.add(getCountryByName("French Guiana"));
    fusionateCountries("Guyana", guianas);

    // Southeast Asia
    ArrayList<Country> sAsia = new ArrayList<Country>();
    sAsia.add(getCountryByName("Vietnam"));
    sAsia.add(getCountryByName("Cambodia"));
    sAsia.add(getCountryByName("Thailand"));
    sAsia.add(getCountryByName("Laos"));
    fusionateCountries("Southeast Asia", sAsia);

    // India
    ArrayList<Country> india = new ArrayList<Country>();
    india.add(getCountryByName("India"));
    india.add(getCountryByName("Sri Lanka"));
    india.add(getCountryByName("Bangladesh"));
    fusionateCountries("India", india);
}
```

```

// Balcanes
ArrayList<Country> cEur = new ArrayList<Country>();
cEur.add(getCountryByName("Greece"));
cEur.add(getCountryByName("Bulgaria"));
cEur.add(getCountryByName("Macedonia"));
cEur.add(getCountryByName("Albania"));
cEur.add(getCountryByName("Croatia"));
cEur.add(getCountryByName("Bosnia"));
cEur.add(getCountryByName("Slovenia"));
cEur.add(getCountryByName("Serbia"));
cEur.add(getCountryByName("Montenegro"));
fusionateCountries("Balkans", cEur);

// Sudafrica
ArrayList<Country> sAfr = new ArrayList<Country>();
sAfr.add(getCountryByName("South Africa"));
sAfr.add(getCountryByName("Swaziland"));
sAfr.add(getCountryByName("Lesotho"));
fusionateCountries("South Africa", sAfr);

// Eastern africa
ArrayList<Country> eAfr = new ArrayList<Country>();
eAfr.add(getCountryByName("Gambia"));
eAfr.add(getCountryByName("Senegal"));
eAfr.add(getCountryByName("Guinea-Bissau"));
eAfr.add(getCountryByName("Guinea"));
eAfr.add(getCountryByName("Sierra Leone"));
eAfr.add(getCountryByName("Liberia"));
eAfr.add(getCountryByName("Ivory Coast"));
eAfr.add(getCountryByName("Ghana"));
eAfr.add(getCountryByName("Togo"));
eAfr.add(getCountryByName("Benin"));
fusionateCountries("Eastern Africa", eAfr);

// Gabon
ArrayList<Country> gab = new ArrayList<Country>();
gab.add(getCountryByName("Gabon"));
gab.add(getCountryByName("E. Guinea"));
fusionateCountries("Gabon", gab);

// Ethiopia
ArrayList<Country> eth = new ArrayList<Country>();
eth.add(getCountryByName("Ethiopia"));
eth.add(getCountryByName("Eritrea"));
eth.add(getCountryByName("Djibouti"));
fusionateCountries("Ethiopia", eth);

// Congo
ArrayList<Country> con = new ArrayList<Country>();
con.add(getCountryByName("Congo A"));
con.add(getCountryByName("Congo B"));
con.add(getCountryByName("Congo C"));
con.add(getCountryByName("Burundi"));
con.add(getCountryByName("Ruanda"));
fusionateCountries("Congo", con);

```

```

// China
ArrayList<Country> chin = new ArrayList<Country>();
chin.add(getCountryByName("China"));
chin.add(getCountryByName("Nepal"));
chin.add(getCountryByName("Bhutan"));
fusionateCountries("China", chin);
}

```

Las reglas de dichas fusiones se pueden resumir en las siguientes, y serán realizadas por el método `fusionateCountries`:

- El territorio base del nuevo país es la suma de los territorios base de los países fusionados, apareciendo un nuevo territorio formado por todos los polígonos que formasen los territorios de los países originales.
- La producción del nuevo país es la suma de las producciones de todos los países originales multiplicada por un factor de escala para evitar que acumulen demasiada producción, actualmente ese factor es de un 60%.
- El dinero inicial del nuevo país es la suma del dinero de todos los países fusionados multiplicado por el mismo factor de escala que en el caso de la producción, así la cantidad seguirá siendo proporcional.
- Las relaciones diplomáticas del nuevo país con el resto de países será la media de los valores de los países originales con cada país objetivo.

Respecto a lo anterior, hay una serie de consideraciones a tener en cuenta:

- Toda la información anterior se ha extraído de Wikipedia, de distintas listas de países agrupados por alianzas territoriales, por PIB, etc.
- La producción de cada territorio se ha extraído de su PIB anual dividido entre 365, de forma que la producción de cada territorio es proporcional al PIB de su país base.
- El dinero que posee cada país inicialmente es el equivalente a 10 días de producción.
- Las alianzas de países también se han extraído de listas y sus valores se han diseñado de forma coherente con el modelo geopolítico real.
- Las culturas de los países se han extraído también de listas que valoran distintos parámetros, como continente de pertenencia o religión.
- El tipo de IA de los países ha sido establecido manualmente de forma que encaje con el modelo geopolítico real.

Polígonos de territorios

Hemos visto en la lista de territorios que se le asignaba una variable a cada territorio, dicha variable no es más que un array tridimensional de puntos que conforman polígonos, siendo la primera dimensión las coordenadas XY, la segunda dimensión los puntos que conforman un polígono, y la tercera cada polígono en sí mismo, como podemos apreciar en los siguientes ejemplos de variables:

```
// Definimos poligonos que formaran los territorios
short[][][] ae = new short[][][]
{{{5995,2942},{6059,2942},{6115,2890},{6119,2902},{6119,2922},{6107,2922},{6107,2946},{6099,2946},{6087,2986},{6023,2978}}};

short[][][] at = new short[][][]
{{{4895,2238},{4979,2238},{4975,2218},{5003,2198},{5027,2202},{5035,2186},{5087,2198},{5087,2230},{5067,2258},{5023,2274},{4999,2270},{4959,2254},{4915,2262},{4891,2250}}};

short[][][] tt = new short[][][]
{{{3016,3320},{3025,3316},{3021,3304},{3039,3301},{3036,3321}},{{3041,3292},{3048,3288},{3049,3291}}};
(...)
```

Construir manualmente esta información de puntos sería un trabajo titánico, así que se han extraído y adaptado de una fuente ya existente: <https://github.com/maralik/Interactive-HTML5-world-map>

Dicho repositorio aloja la implementación de una web en HTML5 que genera un mapa bidimensional del mundo que reacciona al evento mouseover, mostrando las propiedades del país que hay bajo el ratón.



Dentro del código podemos encontrar la variable javascript oWorldMap que guarda la información de los polígonos. Dicha información ha sido traducida a java y procesada por una función de adaptación al formato de resolución del smartphone que devuelve directamente una lista que contiene objetos Polygon, que es una clase del framework y que será lo que recibirá el constructor de Territory. Las fórmulas de conversión de valores han sido obtenidas comparando las referencias que se han usado en la web con respecto a las que usa libgdx para renderizar, realizando además un escalado dependiente de la resolución que hemos supuesto lineal y cuya recta se ha inferido mediante regresión.

```
public static ArrayList<Polygon> getPolygonListFromVortex(short[][][]
territoryPolygonVortex)
{
    ArrayList<Polygon> outputPolygonList = new ArrayList<Polygon>();

    // Bucle for que vaya barriendo el territoryPolygonVortex y vaya
    // anadiendo poligonos a la lista output.
    int i;
    int j;
    for (i = 0; i < territoryPolygonVortex.length; i++)
    {
        // Leemos poligono (array de coordenadas)
        short[][] polygon = territoryPolygonVortex[i];

        // Contenedor de coordenadas para este poligono
        ArrayList<Float> floatCoordinates = new ArrayList<Float>();

        // Barremos poligono acumulando coordenadas que en el formato
        // de entrada estan separadas en subarrays
        for (j = 0; j < polygon.length; j++)
        {
            // Variables de ajuste de representacion
            float scalingFactor = 5;
            float offsetX = 20.0f; // Cte para las resoluciones
            // OffsetY -20 en 800x600 y 100 en 1920x1080, lo suponemos
            // lineal. (esto puede dar problemas)
            float offsetY = 0.2f * Gdx.graphics.getHeight() - 116;
            float resFactorX = Gdx.graphics.getWidth() / 1920f;
            float resFactorY = Gdx.graphics.getHeight() / 1080f;

            // Ajustamos formato de las coordenadas a la libreria
            // grafica GDX
            float finalXCoordinate = (((float)polygon[j][0]) /
            scalingFactor - offsetX) * resFactorX -
            Gdx.graphics.getWidth() / 2;
            float finalYCoordinate = ((Gdx.graphics.getHeight() -
            (float)polygon[j][1]) / scalingFactor -
            offsetY) * resFactorY +
            Gdx.graphics.getHeight() / 2;

            // Establecemos valor
            floatCoordinates.add(finalXCoordinate);
            floatCoordinates.add(finalYCoordinate);
        }
    }
}
```

```

        // Creamos el nuevo poligono con las coordenadas acumuladas y
        // lo add a la lista de poligonos
        outputPolygonList.add(new
            Polygon(getFloatArrayFromFloatList(floatCoordinates)));
    }

    // Devolvemos lista con todos los poligonos que forman el país
    // acumulados
    return outputPolygonList;
}

```

Una vez que el constructor del territorio ya posee la lista de polígonos generará una lista de sprites (imágenes de tamaño y posición determinadas) para cada uno de sus polígonos, formando el dibujo geométrico que será dibujado sobre el mapa en la llamada a render:

```

// SELECCIONADO: GRIS BLANQUECINO
if (m_selectedTerritory != null)
    for (Polygon poly : m_selectedTerritory.getTerritoryPolygonList())
    {
        m_selectedTerritory.getPolygonSprite(poly).setColor(1, 1, 1,
            0.5f);
        m_selectedTerritory.getPolygonSprite(poly).draw(m_polyBatch);
    }

for (Territory terry : m_territoryList.values())
    for (Polygon poly : terry.getTerritoryPolygonList())
    {
        // PLAYER: AZUL
        if (terry.getOwner().equals(m_renderTargetCountry))
        {
            terry.getPolygonSprite(poly).setColor(0, 0, 1, 0.3f);
            terry.getPolygonSprite(poly).draw(m_polyBatch);
        }
        else if (terry.getOwner().isAlliedWith(m_renderTargetCountry))
        // ALIADO: VERDE
        {
            terry.getPolygonSprite(poly).setColor(0, 1, 0, 0.4f);
            terry.getPolygonSprite(poly).draw(m_polyBatch);
        }
        else if (terry.getOwner().
            isFriendlyWith(m_renderTargetCountry))
        // AMISTOSO: VERDE MENOS INTENSO
        {
            terry.getPolygonSprite(poly).setColor(0, 1, 0, 0.2f);
            terry.getPolygonSprite(poly).draw(m_polyBatch);
        }
        else if (terry.getOwner().
            isNeutralWith(m_renderTargetCountry))
    }

```

```

// NEUTRAL: AMARILLO
{
    terry.getPolygonSprite(poly).setColor(1, 1, 0, 0.2f);
    terry.getPolygonSprite(poly).draw(m_polyBatch);
}
else if (terry.getOwner()
        .isHostileWith(m_renderTargetCountry))
// HOSTIL: ROJO
{
    terry.getPolygonSprite(poly).setColor(1, 0, 0, 0.2f);
    terry.getPolygonSprite(poly).draw(m_polyBatch);
}
}

```

Como los polígonos no están implementados en el framework como una extensión de la clase Actor, no podemos asignarle eventos touchDown a los propios polígonos, por ese motivo debemos implementar el input mediante la detección del nodo pulsado, dando lugar a un diseño considerablemente más complejo.

Sistema de nodos

Como hemos visto en la inicialización, se guarda en un array de posiciones la id del territorio que corresponde a cada posición, con lo que el juego almacena desde ese instante en memoria la id de cada territorio que hay en cada posición, esto por ejemplo permite que si queremos comprobar el territorio que ha clickado un usuario hagamos lo siguiente:

```

// Detectamos el país pulsado
Territory terry = getTerritoryByPositionFromTopLeftReference(new
Vector2(screenX, screenY));

private Territory getTerritoryByPositionFromTopLeftReference(Vector2
pos)
{
    // Detectamos el país pulsado
    float coordX = pos.x * Game.mainUICamera.zoom +
        Game.mainUICamera.position.x + windowW / 2 - ((windowW / 2)
            * Game.mainUICamera.zoom);
    float coordY = (windowH - pos.y) * Game.mainUICamera.zoom +
        Game.mainUICamera.position.y + windowH / 2 -
            ((windowH / 2) * Game.mainUICamera.zoom);

    return getTerritoryById(m_mapPoints[(int) (coordX /
        Game.m_nodeSize)][(int) (coordY / Game.m_nodeSize)] + 128);
}

```

Este sistema también es eficiente a la hora de obtener un punto aleatorio del territorio o su centro, cuando los territorios se instancian, guardan en una variable de clase un array de las posiciones que forman el territorio, facilitando la obtención de un punto aleatorio del territorio que usarán tanto las IAs, por ejemplo:

```
// Se llama a este método al generar el array de posiciones, guardando
la posición en el territorio al que pertenece
if (poly.contains(coordX, coordY))
{
    // Add punto al array
    m_mapPoints[i][arrayHeight - j] = (byte)(terry.getId() - 128);

    // Add posicion al territorio en cuestion
    terry.addPosition(i, arrayHeight - j);
}

public void addPosition(int x, int y)
{
    ArrayList<Integer> point = new ArrayList<>();
    point.add(x);
    point.add(y);
    m_positions.add(point);
}

// Y aquí obtenemos un punto aleatorio del territorio
public Vector2 getRandomPoint()
{
    (...)
    // Devolvemos un punto aleatorio del array de puntos
    ArrayList<Integer> ranPos =
        m_positions.get(Utils.getRandomValue(0, m_positions.size() -
                                                1));

    return Utils.getMapPositionByNodePosition(new
        Vector2(ranPos.get(0), ranPos.get(1)));
}
```

Podemos hacer las siguientes apreciaciones al sistema:

- Sobre la ID leída del array de posiciones tendremos que sumar 128, ya que se almacena en bytes para ahorrar memoria al ser el tipo primitivo de menor número de bits capaz de representar todos los territorios que tenemos, pero al ser signed necesitaremos desplazarlo en -128 al guardarlo para no perder los valores altos y volverlo a desplazar en +128 al leer sus valores.
- Debemos realizar una conversión entre la posición pulsada y la posición del nodo, ya que se encuentran en planos de referencia diferentes al no estar los nodos afectados por los movimientos de cámara o por el zoom, teniendo posiciones absolutas respecto a la pantalla que no compartirá con otros parámetros que reciben los métodos que implementa el framework. (como el listener touchDown que representa el ejemplo anterior)
- La principal ventaja del sistema es el rendimiento que ofrece, cuando se genera el array de posiciones lo hace comprobando si una posición está contenida en un polígono, esta operación resulta extremadamente ineficiente si se realiza un elevado número de veces, ya que ha de comprobar la pertenencia a un polígono irregular que puede tener muchísimas aristas, en los ejemplos anteriores vemos que una vez generado el array de posiciones basta con obtener la ID del territorio que se encuentra en esa posición y operar con ella.
- Existen más ejemplos de utilización de este sistema para leer posiciones, pero no veo necesario comentarlos, ya que todos funcionan del mismo modo.

Lo que hemos comentado anteriormente hace referencia a la generación del array de posiciones, que no es exactamente lo mismo que el array de nodos. Un nodo es un objeto definido por una posición y por sus conexiones con otros nodos, y dichas conexiones tienen un coste, de modo que un nodo se puede relacionar directamente con un territorio debido a la id a la que corresponde la misma posición del array de posiciones. El uso principal de los nodos es aplicar sobre ellos el algoritmo de pathing que emplearán las IAs para hallar caminos óptimos para mover sus tropas. Todos estos elementos se realizan en la inicialización del juego, y en este apartado vamos a profundizar a ese respecto:

```
// Aquí inicializamos el grafo de nodos
public void initializeMovementMaps()
{
    // Inicializamos m_path
    m_graph = new DynamicGraph(m_mapPoints);
    m_path = new DefaultGraphPath<DynamicGraph.Node>();
}
```

```

// Instanciamos todos los nodos, uno por cada posición leída del array
de posiciones
// Conectamos todos los nodos entre si, cada nodo se conecta con los 8
que le rodean (
public DynamicGraph(byte[][] mapPoints)
{
    (...)

    // Construimos
    m_nodePoints = new Node[mapPoints.length][mapPoints[0].length];
    for (int i = 0; i < mapPoints.length; i++)
        for (int j = 0; j < mapPoints[0].length; j++)
            m_nodePoints[i][j] = new Node(getIndexByPosition(new
                Vector2(i, j)));

    // Conectamos
    for (int i = 0; i < m_nodePoints.length; i++)
        for (int j = 0; j < m_nodePoints[0].length; j++)
            m_nodePoints[i][j].createConnections();
}

```

Ejemplo de creación de las posiciones de cada nodo que se muestra en el siguiente método:

-1,1	0,1	1,1
-1,0	0,0	1,0
-1,-1	0,-1	1,-1

```

public void createConnections()
{
    (...)
    // LATERALES
    // +1, 0
    int destPosX = basePosX + 1;
    int destPosY = basePosY;
    if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
        <= maxY)
        m_connections.add(new NodesConnection(this, getNode(destPosX,
            destPosY)));

    // 0, +1
    destPosX = basePosX;
    destPosY = basePosY + 1;
    if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
        <= maxY)
        m_connections.add(new NodesConnection(this, getNode(destPosX,
            destPosY)));
}

```

```

// -1, 0
destPosX = basePosX - 1;
destPosY = basePosY;
if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
                                     <= maxY)
    m_connections.add(new NodesConnection(this, getNode(destPosX,
                                                         destPosY)));

// 0, -1
destPosX = basePosX;
destPosY = basePosY - 1;
if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
                                     <= maxY)
    m_connections.add(new NodesConnection(this, getNode(destPosX,
                                                         destPosY)));

// DIAGONALES
// 1, 1
destPosX = basePosX + 1;
destPosY = basePosY + 1;
if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
                                     <= maxY)
    m_connections.add(new NodesConnection(this, getNode(destPosX,
                                                         destPosY)));

// 1, -1
destPosX = basePosX + 1;
destPosY = basePosY - 1;
if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
                                     <= maxY)
    m_connections.add(new NodesConnection(this, getNode(destPosX,
                                                         destPosY)));

// -1, 1
destPosX = basePosX - 1;
destPosY = basePosY + 1;
if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
                                     <= maxY)
    m_connections.add(new NodesConnection(this, getNode(destPosX,
                                                         destPosY)));

// -1, -1
destPosX = basePosX - 1;
destPosY = basePosY - 1;
if (destPosX >= 0 && destPosY >= 0 && destPosX <= maxX && destPosY
                                     <= maxY)
    m_connections.add(new NodesConnection(this, getNode(destPosX,
                                                         destPosY)));
}

```

Dichas conexiones tienen además un coste que se lee dinámicamente y que empleará el algoritmo de pathing, profundizaremos más en la función de costes, el algoritmo de ruta y las heurísticas geométricas en el apartado dedicado al pathing de las IAs.

3.7 TECNOLOGÍAS

Las tecnologías son elementos que pueden ser investigados por cada país, pagando un coste y esperando un tiempo de investigación, una vez desarrolladas, su bonificación se aplicará permanentemente al país que las desarrolla. Se implementa mediante árboles tecnológicos, de forma que cada tecnología requiere otras previas para ser investigada y algunas sólo existen para una determinada cultura, las tecnologías acumulan sus efectos una vez investigadas. En las siguientes tablas se muestran todas las tecnologías existentes y sus atributos:

Tipo	Nombre	Descripción	Valor
TECH_DIGITAL_EXCHANGE_I	Digital Exchange I	Increases the production of all of your territories by %v%.	5
TECH_DIGITAL_EXCHANGE_II	Digital Exchange II	Increases the production of all of your territories by %v%.	10
TECH_DIGITAL_EXCHANGE_III	Digital Exchange III	Increases the production of all of your territories by %v%.	15
TECH_GLOBAL_ARCHITECTURE_I	Global Architecture I	Allow your country to upgrade its buildings to level %v.	2
TECH_GLOBAL_ARCHITECTURE_II	Global Architecture II	Allow your country to upgrade its buildings to level %v.	3
TECH_GLOBAL_ARCHITECTURE_III	Global Architecture III	Allow your country to upgrade its buildings to their maximum level.	4
TECH_ROBOTIC_MACHINERY_I	Robotic Machinery I	Increase the damage regeneration on your territories by %v%.	10
TECH_ROBOTIC_MACHINERY_II	Robotic Machinery II	Increase the damage regeneration on your territories by %v%.	20
TECH_ROBOTIC_MACHINERY_III	Robotic Machinery III	Increase the damage regeneration on your territories by %v%.	30
TECH_INTELLIGENT_BUILDINGS_I	Intelligent Buildings I	Increases the speed of all your constructions by %v%.	10
TECH_INTELLIGENT_BUILDINGS_II	Intelligent Buildings II	Increases the speed of all your constructions by %v%.	20
TECH_INTELLIGENT_BUILDINGS_III	Intelligent Buildings III	Increases the speed of all your constructions by %v%.	30
TECH_NATIONAL_EXPLOITATION_I	National Exploitation I	Increases the production of your initial territory by %v% with a maximum of 50.	500

TECH_NATIONAL_EXPLOITATION_II	National Exploitation II	Increases the production of your initial territory by %v% with a maximum of 50.	1000
TECH_NATIONAL_EXPLOITATION_III	National Exploitation III	Increases the production of your initial territory by %v% with a maximum of 50.	2000
TECH_DRONES_RESEARCHING_I	Drones Researching I	Allows you to produce aircrafts.	0
TECH_DRONES_RESEARCHING_II	Drones Researching II	Increases damage and speed of all your aircrafts by %v%.	15
TECH_DRONES_RESEARCHING_III	Drones Researching III	Increases damage and speed of all your aircrafts by %v%.	30
TECH_GENETIC_REDESIGN_I	Genetic Redesign I	Increases damage of all your infantry units by %v%.	10
TECH_GENETIC_REDESIGN_II	Genetic Redesign II	Increases damage of all your infantry units by %v%.	15
TECH_GENETIC_REDESIGN_III	Genetic Redesign III	Increases damage of all your infantry units by %v%.	20
TECH_CONSCRIPTION_I	Conscription I	Increases the chance of spawning militias by %v%.	15
TECH_CONSCRIPTION_II	Conscription II	Increases the chance of spawning militias by %v%.	30
TECH_CONSCRIPTION_III	Conscription III	Increases the chance of spawning militias by %v%.	45
TECH_ARMORED_TANKS_I	Armored Tanks I	Increases the defence of all artillery units by %v%.	10
TECH_ARMORED_TANKS_II	Armored Tanks II	Increases the defence of all artillery units by %v%.	15
TECH_ARMORED_TANKS_III	Armored Tanks III	Increases the defence of all artillery units by %v%.	20
TECH_MILITARY_TRAINING_I	Military Training I	Increases the speed of all your military trainings by %v%.	10
TECH_MILITARY_TRAINING_II	Military Training II	Increases the speed of all your military trainings by %v%.	20
TECH_MILITARY_TRAINING_III	Military Training III	Increases the speed of all your military trainings by %v%.	30
TECH_MILITARY_IMPERIALISM_I	Military Imperialism I	Increases damage done by all your troops to non western enemies by %v%.	2

TECH_MILITARY_IMPERIALISM_II	Military Imperialism II	Increases damage done by all your troops to non western enemies by %v%.	4
TECH_MILITARY_IMPERIALISM_III	Military Imperialism III	Increases damage done by all your troops to non western enemies by %v%.	6
TECH_POLITICAL_TRAINING_I	Political Training I	Strengthen the loyalty on your territories by %v%.	5
TECH_POLITICAL_TRAINING_II	Political Training II	Strengthen the loyalty on your territories by %v%.	10
TECH_POLITICAL_TRAINING_III	Political Training III	Strengthen the loyalty on your territories by %v%.	15
TECH_SOLIDARY_DONATIONS_I	Solidary Donations I	Improves your international relations will all foreign countries by %v%.	5
TECH_SOLIDARY_DONATIONS_II	Solidary Donations II	Improves your international relations will all foreign countries by %v%.	5
TECH_SOLIDARY_DONATIONS_III	Solidary Donations III	Improves your international relations will all foreign countries by %v%.	5
TECH_DIPLOMATIC_REINFORCEMENT_I	Diplomatic Reinforcement I	Strengthen your relations with all foreign countries by %v%.	15
TECH_DIPLOMATIC_REINFORCEMENT_II	Diplomatic Reinforcement II	Strengthen your relations with all foreign countries by %v%.	15
TECH_DIPLOMATIC_REINFORCEMENT_III	Diplomatic Reinforcement III	Strengthen your relations with all foreign countries by %v%.	30
TECH_CULTURAL_SUMMIT_I	Cultural Summit I	Strengthen your relations with all foreign countries by %v%.	10
TECH_CULTURAL_SUMMIT_II	Cultural Summit II	Strengthen your relations with all foreign countries by %v%.	20
TECH_CULTURAL_SUMMIT_III	Cultural Summit III	Strengthen your relations with all foreign countries by %v%.	30
TECH_UNBEATABLE_LOYALTY_I	Unbeatable Loyalty I	Strengthen the loyalty on your territories by %v%.	10
TECH_UNBEATABLE_LOYALTY_II	Unbeatable Loyalty II	Strengthen the loyalty on your territories by %v%.	20
TECH_UNBEATABLE_LOYALTY_III	Unbeatable Loyalty III	Strengthen the loyalty on your territories by %v%.	30

TECH_LOCAL_TAXES	Local Taxes	Increases the production of each territory you own by %v% for each building it has at its maximum level.	10
TECH_TIRELESS_WORKERS	Tireless Workers	Increases the damage regeneration of your territories damaged above 50% by %v%.	300
TECH_WAR_HEROES	War Heroes	Increases damage done by all your troops by %v%.	10
TECH_FIERCE_DEFENCE	Fierce Defense	Increases defence of all your troops by %v%.	20
TECH_GLOBAL_SUMMIT	Global Summit	Improves your international relations will all foreign countries by %v%.	20
TECH_INTERNATIONAL_SUPPORT	International Support	Your territories spawns %v% more millitians when being conquered.	50

Tipo	Tecnologia previa	Cultura
TECH_DIGITAL_EXCHANGE_I		
TECH_DIGITAL_EXCHANGE_II	TECH_DIGITAL_EXCHANGE_I	
TECH_DIGITAL_EXCHANGE_III	TECH_DIGITAL_EXCHANGE_II	
TECH_GLOBAL_ARCHITECTURE_I		
TECH_GLOBAL_ARCHITECTURE_II	TECH_GLOBAL_ARCHITECTURE_I	
TECH_GLOBAL_ARCHITECTURE_III	TECH_GLOBAL_ARCHITECTURE_II	
TECH_ROBOTIC_MACHINERY_I		
TECH_ROBOTIC_MACHINERY_II	TECH_ROBOTIC_MACHINERY_I	
TECH_ROBOTIC_MACHINERY_III	TECH_ROBOTIC_MACHINERY_II	
TECH_INTELLIGENT_BUILDINGS_I		
TECH_INTELLIGENT_BUILDINGS_II	TECH_INTELLIGENT_BUILDINGS_I	
TECH_INTELLIGENT_BUILDINGS_III	TECH_INTELLIGENT_BUILDINGS_II	
TECH_NATIONAL_EXPLOITATION_I		AFRICAN
TECH_NATIONAL_EXPLOITATION_II	TECH_NATIONAL_EXPLOITATION_I	AFRICAN
TECH_NATIONAL_EXPLOITATION_III	TECH_NATIONAL_EXPLOITATION_II	AFRICAN
TECH_DRONES_RESEARCHING_I		
TECH_DRONES_RESEARCHING_II	TECH_DRONES_RESEARCHING_I	
TECH_DRONES_RESEARCHING_III	TECH_DRONES_RESEARCHING_II	
TECH_GENETIC_REDESIGN_I		
TECH_GENETIC_REDESIGN_II	TECH_GENETIC_REDESIGN_I	
TECH_GENETIC_REDESIGN_III	TECH_GENETIC_REDESIGN_II	
TECH_CONSCRIPTION_I		
TECH_CONSCRIPTION_II	TECH_CONSCRIPTION_I	
TECH_CONSCRIPTION_III	TECH_CONSCRIPTION_II	
TECH_ARMORED_TANKS_I		

TECH_ARMORED_TANKS_II	TECH_ARMORED_TANKS_I	
TECH_ARMORED_TANKS_III	TECH_ARMORED_TANKS_II	
TECH_MILITARY_TRAINING_I		
TECH_MILITARY_TRAINING_II	TECH_MILITARY_TRAINING_I	
TECH_MILITARY_TRAINING_III	TECH_MILITARY_TRAINING_II	
TECH_MILITARY_IMPERIALISM_I		WESTERN
TECH_MILITARY_IMPERIALISM_II	TECH_MILITARY_IMPERIALISM_I	WESTERN
TECH_MILITARY_IMPERIALISM_III	TECH_MILITARY_IMPERIALISM_II	WESTERN
TECH_POLITICAL_TRAINING_I		
TECH_POLITICAL_TRAINING_II	TECH_POLITICAL_TRAINING_I	
TECH_POLITICAL_TRAINING_III	TECH_POLITICAL_TRAINING_II	
TECH_SOLIDARY_DONATIONS_I		
TECH_SOLIDARY_DONATIONS_II	TECH_SOLIDARY_DONATIONS_I	
TECH_SOLIDARY_DONATIONS_III	TECH_SOLIDARY_DONATIONS_II	
TECH_DIPLOMATIC_REINFORCEMENT_I		
TECH_DIPLOMATIC_REINFORCEMENT_II	TECH_DIPLOMATIC_REINFORCEMENT_I	
TECH_DIPLOMATIC_REINFORCEMENT_III	TECH_DIPLOMATIC_REINFORCEMENT_II	
TECH_CULTURAL_SUMMIT_I		EASTERN
TECH_CULTURAL_SUMMIT_II	TECH_CULTURAL_SUMMIT_I	EASTERN
TECH_CULTURAL_SUMMIT_III	TECH_CULTURAL_SUMMIT_II	EASTERN
TECH_UNBEATABLE_LOYALTY_I		ARABIC
TECH_UNBEATABLE_LOYALTY_II	TECH_UNBEATABLE_LOYALTY_I	ARABIC
TECH_UNBEATABLE_LOYALTY_III	TECH_UNBEATABLE_LOYALTY_II	ARABIC
TECH_LOCAL_TAXES	TECH_DIGITAL_EXCHANGE_III, TECH_GLOBAL_ARCHITECTURE_III	
TECH_TIRELESS_WORKERS	TECH_ROBOTIC_MACHINERY_III, TECH_INTELLIGENT_BUILDINGS_III	
TECH_WAR_HEROES	TECH_DRONES_RESEARCHING_III, TECH_GENETIC_REDESIGN_III	
TECH_FIERCE_DEFENCE	TECH_ARMORED_TANKS_III, TECH_MILITARY_TRAINING_III	
TECH_GLOBAL_SUMMIT	TECH_POLITICAL_TRAINING_III, TECH_SOLIDARY_DONATIONS_III	
TECH_INTERNATIONAL_SUPPORT	TECH_DIPLOMATIC_REINFORCEMENT_III, TECH_CONSCRIPTION_III	

Tipo	Icono	Cost	Time	Escuela
TECH_DIGITAL_EXCHANGE_I	image_tech_digital_exchange_i	1000	20	ECONOMY
TECH_DIGITAL_EXCHANGE_II	image_tech_digital_exchange_ii	2000	30	ECONOMY

TECH_DIGITAL_EXCHANGE_III	image_tech_digital_exchange_iii	4000	40	ECONOMY
TECH_GLOBAL_ARCHITECTURE_I	image_tech_global_architecture_i	1000	20	ECONOMY
TECH_GLOBAL_ARCHITECTURE_II	image_tech_global_architecture_ii	2000	30	ECONOMY
TECH_GLOBAL_ARCHITECTURE_III	image_tech_global_architecture_iii	4000	40	ECONOMY
TECH_ROBOTIC_MACHINERY_I	image_tech_robotic_machinery_i	1000	20	ECONOMY
TECH_ROBOTIC_MACHINERY_II	image_tech_robotic_machinery_ii	2000	30	ECONOMY
TECH_ROBOTIC_MACHINERY_III	image_tech_robotic_machinery_iii	4000	40	ECONOMY
TECH_INTELLIGENT_BUILDINGS_I	image_tech_intelligent_buildings_i	1000	20	ECONOMY
TECH_INTELLIGENT_BUILDINGS_II	image_tech_intelligent_buildings_ii	2000	30	ECONOMY
TECH_INTELLIGENT_BUILDINGS_III	image_tech_intelligent_buildings_iii	4000	40	ECONOMY
TECH_NATIONAL_EXPLOITATION_I	image_tech_national_exploitation_i	100	20	ECONOMY
TECH_NATIONAL_EXPLOITATION_II	image_tech_national_exploitation_ii	500	50	ECONOMY
TECH_NATIONAL_EXPLOITATION_III	image_tech_national_exploitation_iii	2500	150	ECONOMY
TECH_DRONES_RESEARCHING_I	image_tech_drones_researching_i	1000	20	MILITARY
TECH_DRONES_RESEARCHING_II	image_tech_drones_researching_ii	2000	30	MILITARY
TECH_DRONES_RESEARCHING_III	image_tech_drones_researching_iii	4000	40	MILITARY
TECH_GENETIC_REDESIGN_I	image_tech_genetic_redesign_i	1000	20	MILITARY
TECH_GENETIC_REDESIGN_II	image_tech_genetic_redesign_ii	2000	30	MILITARY
TECH_GENETIC_REDESIGN_III	image_tech_genetic_redesign_iii	4000	40	MILITARY
TECH_CONSCRIPTION_I	image_tech_conscription_i	1000	20	MILITARY
TECH_CONSCRIPTION_II	image_tech_conscription_ii	2000	30	MILITARY
TECH_CONSCRIPTION_III	image_tech_conscription_iii	4000	40	MILITARY
TECH_ARMORED_TANKS_I	image_tech_armored_tanks_i	1000	20	MILITARY
TECH_ARMORED_TANKS_II	image_tech_armored_tanks_ii	2000	30	MILITARY
TECH_ARMORED_TANKS_III	image_tech_armored_tanks_iii	4000	40	MILITARY
TECH_MILITARY_TRAINING_I	image_tech_military_training_i	1000	20	MILITARY

TECH_MILITARY_TRAINING_II	image_tech_military_training_ii	2000	30	MILITARY
TECH_MILITARY_TRAINING_III	image_tech_military_training_iii	4000	40	MILITARY
TECH_MILITARY_IMPERIALISM_I	image_tech_military_imperialism_i	1000	20	MILITARY
TECH_MILITARY_IMPERIALISM_II	image_tech_military_imperialism_ii	2000	30	MILITARY
TECH_MILITARY_IMPERIALISM_III	image_tech_military_imperialism_iii	4000	40	MILITARY
TECH_POLITICAL_TRAINING_I	image_tech_political_training_i	1000	20	DIPLOMACY
TECH_POLITICAL_TRAINING_II	image_tech_political_training_ii	2000	30	DIPLOMACY
TECH_POLITICAL_TRAINING_III	image_tech_political_training_iii	4000	40	DIPLOMACY
TECH_SOLIDARY_DONATIONS_I	image_tech_solidary_donations_i	1000	20	DIPLOMACY
TECH_SOLIDARY_DONATIONS_II	image_tech_solidary_donations_ii	2000	30	DIPLOMACY
TECH_SOLIDARY_DONATIONS_III	image_tech_solidary_donations_iii	4000	40	DIPLOMACY
TECH_DIPLOMATIC_REINFORCEMENT_I	image_tech_diplomatic_reinforcement_i	1000	20	DIPLOMACY
TECH_DIPLOMATIC_REINFORCEMENT_II	image_tech_diplomatic_reinforcement_ii	2000	30	DIPLOMACY
TECH_DIPLOMATIC_REINFORCEMENT_III	image_tech_diplomatic_reinforcement_iii	4000	40	DIPLOMACY
TECH_CULTURAL_SUMMIT_I	image_tech_cultural_summit_i	1000	20	DIPLOMACY
TECH_CULTURAL_SUMMIT_II	image_tech_cultural_summit_ii	2000	30	DIPLOMACY
TECH_CULTURAL_SUMMIT_III	image_tech_cultural_summit_iii	4000	40	DIPLOMACY
TECH_UNBEATABLE_LOYALTY_I	image_tech_unbeatable_loyalty_i	1000	20	DIPLOMACY
TECH_UNBEATABLE_LOYALTY_II	image_tech_unbeatable_loyalty_ii	2000	30	DIPLOMACY
TECH_UNBEATABLE_LOYALTY_III	image_tech_unbeatable_loyalty_iii	4000	40	DIPLOMACY
TECH_LOCAL_TAXES	image_tech_smart_saving	8000	50	ECONOMY
TECH_TIRELESS_WORKERS	image_tech_tireless_workers	8000	50	ECONOMY
TECH_WAR_HEROES	image_tech_war_heroes	8000	50	MILITARY
TECH_FIERCE_DEFENCE	image_tech_fierce_defense	8000	50	MILITARY
TECH_GLOBAL_SUMMIT	image_tech_global_summit	8000	50	DIPLOMACY
TECH_INTERNATIONAL_SUPPORT	image_tech_international_support	8000	50	DIPLOMACY

El campo valor representa el efecto que tendrá esa tecnología, la descripción que se muestra en la interfaz reemplazará la cadena "%v" por el valor de la instancia como se puede ver en el siguiente fragmento:

```
public String getDisplayDescription()
{
    return m_description.replace("%v", Integer.toString(getValue()));
}
```

La implementación del efecto leerá el valor desde la instancia de la tecnología, por lo que modificarlo en la instancia es suficiente, sin necesitar tocar el código de la implementación del efecto. En el siguiente ejemplo de código vemos como se implementa el efecto de las tecnologías, en este caso, en este método de Troops que devuelve el total del poder de ataque:

```
public int getTotalAttackPower(float troopNumberPct)
{
    // Bono por distintos efectos
    float bonusAttack = 0f;

    // Ataque de cada unidad por su numero y el % de tropas que luchan
    int baseAttackPower = (int)(getNumber() * (troopNumberPct / 100f)
        * getUnit().getAttack());

    // Add bonos de tecnologias
    // Drone researching 2&3: "Increases damage and speed of all your
    // aircrafts by %v%."
    if (getUnit().getType() == UnitType.UNIT_AIRCRAFT_BOMBER)
    {
        Technology droneResearching1 =
            getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_DR
                ONES_RESEARCHING_II);
        if (droneResearching1 != null)
            bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
                droneResearching1.getValue());

        Technology droneResearching2 =
            getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_DR
                ONES_RESEARCHING_III);
        if (droneResearching2 != null)
            bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
                droneResearching2.getValue());
    }
}
```

```

// Genetic redesign "Increases damage of all your infantry units
// by %v%"
if (getUnit().getType() == UnitType.UNIT_INFANTRY_ASSAULT
    || getUnit().getType() == UnitType.UNIT_INFANTRY_MORTAR)
{
    Technology geneticRedesign1 =
    getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_GENETIC_REDESIGN_I);

    if (geneticRedesign1 != null)
        bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
            geneticRedesign1.getValue());

    Technology geneticRedesign2 =
    getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_GENETIC_REDESIGN_II);

    if (geneticRedesign2 != null)
        bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
            geneticRedesign2.getValue());

    Technology geneticRedesign3 =
    getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_GENETIC_REDESIGN_III);

    if (geneticRedesign3 != null)
        bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
            geneticRedesign3.getValue());
}

// Military Imperialism "Increases damage done by all your troops
// to non western enemies by %v%"
if (getOwner().getCulture() != Culture.CULTURE_WESTERN)
{
    Technology militaryImperialism1 =
    getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_MILITARY_IMPERIALISM_I);

    if (militaryImperialism1 != null)
        bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
            militaryImperialism1.getValue());

    Technology militaryImperialism2 =
    getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_MILITARY_IMPERIALISM_II);

    if (militaryImperialism2 != null)
        bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
            militaryImperialism2.getValue());

    Technology militaryImperialism3 =
    getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_MILITARY_IMPERIALISM_III);

    if (militaryImperialism3 != null)
        bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
            militaryImperialism3.getValue());
}

```

```

// War Heroes "Increases damage done by all your troops by %v"
Technology warHeroes1 =
    getOwner().getCountryDevelopedTechnologyByType(TechType.TECH_WA
                                                    R_HEROES);

if (warHeroes1 != null)
    bonusAttack += Utils.ApplyPctToValue(baseAttackPower,
                                         warHeroes1.getValue());

// Ataque total que aportan estas tropas
return baseAttackPower + (int)bonusAttack;
}

```

La implementación de los efectos de todo el juego se realiza con el método `ApplyPctToValue` para acumular en una variable adicional (`bonusAttack`) el incremento porcentual que provoca la bonificación para después añadir dicha suma a la cantidad base y así obtener el resultado final (`baseAttackPower + bonusAttack`)

```

public static float ApplyPctToValue(float value, int pct)
{
    return (value * (float)pct / 100.0f);
}

```

El cálculo se hace de este modo para asegurar que los porcentajes aumentados se apliquen de forma aditiva en lugar de multiplicativa, respondiendo conforme la siguiente expresión:

$$valor_{final} = valor_{inicial} \cdot \left(1 + \frac{suma_bonos_{pct}}{100}\right)$$

Si hiciéramos el cálculo porcentual de la forma más intuitiva acumulando incrementos, el valor final se comportaría según la siguiente expresión:

$$valor_{final} = valor_{inicial} \prod \left(1 + \frac{bono_pct_1}{100}\right)$$

La diferencia de resultado puede apreciarse perfectamente con un ejemplo, imaginemos un valor base de 100, sobre el que se aplican 3 bonificaciones de 25%, 35% y 40% respectivamente:

- En la forma de calculo que hemos empleado mediante acumulación aditiva el resultado sería:

$$valor_{final} = 100 \cdot \left(1 + \frac{25+35+40=100}{100}\right) = 200$$

- En la forma de cálculo basada en simplemente aplicar los porcentajes incrementados de forma secuencial el resultado final sería: $valor_{final} =$

$$100 \cdot \left(1 + \frac{25}{100}\right) \cdot \left(1 + \frac{35}{100}\right) \cdot \left(1 + \frac{40}{100}\right) = 236$$

Podemos observar que el primer resultado es mucho más coherente con el diseño lógico de un sistema de bonificaciones porcentual, en el segundo caso cada incremento porcentual se aplica sobre la cantidad ya incrementada por el incremento que se aplica anteriormente, esto provoca un aumento final superior al deseado y con una dispersión mayor cuantas más bonificaciones actúen simultáneamente.

Interfaz e iconos

En el instanciamiento de las tecnologías definimos el nombre de su icono, y en el siguiente fragmento de código definimos para cada tecnología el nombre de los tres iconos que posee cada tecnología (normal, pulsada y aprendida).

```
// Imagenes de fondo de cada tecnologia
for (Technology tech : m_technologyList)
{
    tech.setDrawableOn(new NinePatchDrawable(new
        NinePatch(getSpriteByName(tech.getIconName() + "_on"), 5, 5, 6,
            6)));
    tech.setDrawableOff(new NinePatchDrawable(new
        NinePatch(getSpriteByName(tech.getIconName() + "_off"), 5, 5,
            6, 6)));
    tech.setDrawableLearnt(new NinePatchDrawable(new
        NinePatch(getSpriteByName(tech.getIconName() + "_learnt"),
            5, 5, 6, 6)));
}
```

La generación de la interfaz de tecnologías se realiza desde el método general de generación de interfaz de usuario updateUI, que llama al método updateUITech si la interfaz tecnológica es visible, el cual en primer lugar asigna a cada botón de tecnología la imagen de fondo que debería mostrar en función de su estado, esto se implementa de este modo ya que los botones de tecnología no deben mostrar los iconos habituales (on, off, check) al seguir una lógica completamente diferente.

```
public void updateUITech()
{
    (...)
    // Actualizamos botones y sus fondos segun estado tecnologico
    for (Technology tech : m_technologyList)
    {
        Image but = m_techButtonList.get(tech);

        // Si no tiene botón, abortar iteracion.
        if (but == null)
            continue;
    }
}
```

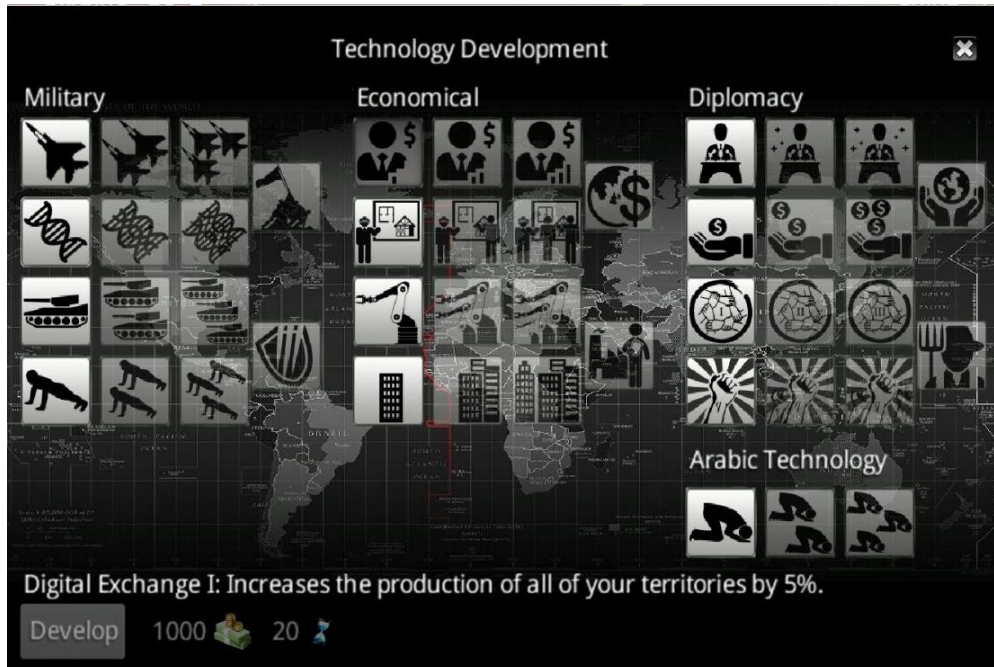
```

// Si es la elegida, oscurecida.
if (m_selectedTech != null)
    if (m_selectedTech.equals(tech))
    {
        but.setDrawable(tech.getDrawableOff());
        continue;
    }
// Si se esta investigando ya, oscurecida.
if (m_player.hasAnyTechnologyInQueue())
    if (m_player.getInvestigationQueue().get(0).equals(tech))
    {
        but.setDrawable(tech.getDrawableOff());
        continue;
    }
// Si la tiene desarrollada, verde.
if (m_player.getCountryDevelopedTechnologyByType
        (tech.getType()) != null)
{
    but.setDrawable(tech.getDrawableLearnt());
    continue;
}
else // si no, si es desarrollable, visible, si no,
        transparente
{
    // Es desarrollable?
    boolean canDevelop = true;
    if (tech.getRequiredTechnologies() != null)
    {
        for (Technology reqTech :
            tech.getRequiredTechnologies())
            if (m_player.getCountryDevelopedTechnologyByType
                (reqTech.getType()) == null)
                canDevelop = false;
    }
    if (!canDevelop) // No se puede desarrollar, transparente
    {
        but.setDrawable(tech.getDrawableOn());
        but.setColor(1, 1, 1, 0.5f);
        continue;
    }
}

// Llegados a este punto, imagen por defecto y sin
        transparencia
but.setColor(1, 1, 1, 1);
but.setDrawable(tech.getDrawableOn());
}

```

Tras el establecimiento del drawable, generamos la interfaz de tecnologías mediante el ya conocido sistema de tablas, la siguiente imagen muestra un ejemplo de árbol tecnológico y a continuación el fragmento de código que lo generaría:



Los árboles tecnológicos se interpretan de izquierda a derecha, de forma que cada tecnología requiere haber investigado las de su izquierda para poder investigarse.

```
(...)
// Panel de tecnologías (por defecto oculto)
m_techTable.add(new Label("Technology Development",
defaultSkin)).colspan(5);
m_techTable.add(techCrossButton).right().top().size(techButtonWidth
/ 1.2f, techButtonWidth / 1.2f);
m_techTable.row();
m_techTable.add(new Label("Military",
defaultSkin)).left().colspan(2);
m_techTable.add(new Label("Economical",
defaultSkin)).padLeft(techSpace * 3).left().colspan(2);
m_techTable.add(new Label("Diplomacy",
defaultSkin)).padLeft(techSpace * 3).left().colspan(2);
m_techTable.row();

// Tabla tecnologías Militar
Table militaryTable = new Table();
Table militarySecondTierTable = new Table();
militaryTable.add(getButtonByTechType(TechType.TECH_DRONES_RESEARCHI
NG_I)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_DRONES_RESEARCHI
NG_II)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_DRONES_RESEARCHI
NG_III)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.row();
```

```

militarySecondTierTable.add(getButtonByTechType(TechType.TECH_WAR_HE
ROES)).padTop(rowheight / 2).padBottom(rowheight /
2).size(techButtonWidth, techButtonWidth);
militarySecondTierTable.row();
militaryTable.row();
militaryTable.add(getButtonByTechType(TechType.TECH_GENETIC_REDESIGN
_I)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_GENETIC_REDESIGN
_II)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_GENETIC_REDESIGN
_III)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.row();
    militarySecondTierTable.add(getButtonByTechType(TechType.TECH_FIER
CE_DEFENCE)).padTop(rowheight / 2).padBottom(rowheight /
2).size(techButtonWidth, techButtonWidth);
militaryTable.row();
militaryTable.add(getButtonByTechType(TechType.TECH_ARMORED_TANKS_I
)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_ARMORED_TANKS_II
)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_ARMORED_TANKS_II
I)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.row();
militaryTable.add(getButtonByTechType(TechType.TECH_MILITARY_TRAININ
G_I)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_MILITARY_TRAININ
G_II)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.add(getButtonByTechType(TechType.TECH_MILITARY_TRAININ
G_III)).space(techSpace).size(techButtonWidth, techButtonWidth);
militaryTable.row();

    if (m_player.getCulture().equals(Culture.CULTURE_WESTERN))
    {
        militaryTable.add(new Label("Western Technology",
defaultSkin)).colspan(3).left().padTop(techSpace);
        militaryTable.row();

        militaryTable.add(getButtonByTechType(TechType.TECH_MILITARY_IMPER
IALISM_I)).space(techSpace).size(techButtonWidth, techButtonWidth);

militaryTable.add(getButtonByTechType(TechType.TECH_MILITARY_IMPERIA
LISM_II)).space(techSpace).size(techButtonWidth, techButtonWidth);

militaryTable.add(getButtonByTechType(TechType.TECH_MILITARY_IMPERIA
LISM_III)).space(techSpace).size(techButtonWidth, techButtonWidth);
        militaryTable.row();
    }
    m_techTable.add(militaryTable).left().top();

    m_techTable.add(militarySecondTierTable)
                .padRight(10).left().top();

    (...)

m_techTable.add(developTable).colspan(6).left();
m_outTableTech.add(m_techTable);
stageUITech.addActor(m_outTableTech);

```

Con el código anterior habríamos generado el primer árbol tecnológico (militar), los demás se han saltado porque son muy similares y no aportan nueva información. Se ha evitado también la parte de la interfaz de desarrollo, ya que está relacionada con la cola de otras tecnologías y esto será tratado explícitamente en el apartado de colas.

3.8 EDIFICIOS

Los edificios son un elemento del juego asociado a cada territorio, pudiendo ser construidos por el dueño del territorio y proporcionando bonificaciones sobre distintos parámetros de dichos territorios, están implementados de forma muy similar a las tecnologías con la salvedad de estar asociados a territorios en lugar de al país completo. Cada edificio tiene varios niveles de actualización, cada nivel requerirá haber construido el anterior antes de poder mejorarse y las ventajas que proporciona cada nivel no se acumulan con las de los niveles previos, reemplazándolas.

Tipo	Nombre	Cost	Time	Nivel	Descripción	Valor
WORKSHOP	Workshop	600	15	1	Increases the damage regeneration of this territory by %v%	30
WORKSHOP	Workshop	1200	18	2	Increases the damage regeneration of this territory by %v%	60
WORKSHOP	Workshop	2400	24	3	Increases the damage regeneration of this territory by %v%	100
WORKSHOP	Workshop	4800	24	4	Increases the damage regeneration of this territory by %v%	150
FORTRESS	Fortress	800	30	1	Increases the defense of your troops in this territory by %v%	10
FORTRESS	Fortress	1600	60	2	Increases the defense of your troops in this territory by %v%	20
FORTRESS	Fortress	3200	90	3	Increases the defense of your troops in this territory by %v%	40
FORTRESS	Fortress	6400	90	4	Increases the defense of your troops in this territory by %v%	70
BANK	Bank	1000	15	1	Increases the production of this territory by %v%	30

BANK	Bank	2000	30	2	Increases the production of this territory by %v%	60
BANK	Bank	4000	45	3	Increases the production of this territory by %v%	100
BANK	Bank	8000	45	4	Increases the production of this territory by %v%	150
PARLIAMENT	Parliament	500	15	1	Strengthen the loyalty of this territory by %v%	30
PARLIAMENT	Parliament	1500	45	2	Strengthen the loyalty of this territory by %v%	60
PARLIAMENT	Parliament	3000	90	3	Strengthen the loyalty of this territory by %v%	100
PARLIAMENT	Parliament	6000	90	4	Strengthen the loyalty of this territory by %v%	150

La implementación del efecto es muy similar al caso de las tecnologías, con la salvedad de que en este caso debe buscar edificio por tipo y nivel, para leer su valor.

```

public int getTotalInfantryDefencePower(float troopNumberPct,
Territory battleTerritory)
{
    // Bono por distintos efectos
    float bonusDefenseInfantry = 0f;

    // Defensa de cada unidad por su numero
    int baseDefense = (int)(getNumber() * (troopNumberPct / 100f) *
        getUnit().getDefInfantry());

    (...)

    // Fortress increases the defence
    if (battleTerritory != null)
    {
        Building deffBuilding =
            battleTerritory.getBuildingByType(BuildType.BUILD_FORTRESS);
        if (defensiveBuilding != null)
            bonusDefenseInfantry += Utils.ApplyPctToValue(baseDefense,
                deffBuilding.getValue());
    }
    (...)

    // Defensas totales que aportan las tropas
    return baseDefenseInfantryPower + (int)bonusDefenseInfantry;
}

```

La implementación de la interfaz se realiza en el método updateUI, añadiendo los datos relevantes e interfaz asociada a la tabla inferior.

Verificación de producciones

Se ha realizado una implementación que soporta la comprobación de la posibilidad de una producción (tropas, tecnologías y edificios) y reporta al usuario la causa si hay algún error. Funciona del siguiente modo:

- Si un edificio/tecnología/tropa no puede ser producido por cualquier causa, el botón de construir/entrenar/desarrollar se mostrará disabled, pero será pulsable.
- Si un botón desactivado es pulsado, el usuario recibirá un mensaje de error explicando por qué no puede realizar la producción.
- Si el botón no está desactivado, al pulsarlo encolará la producción.
- Todo lo anterior se implementa mediante la clase TryResponse, que devuelve un campo bool (true si se puede construir y false si no) y un String (error en caso de false) en caso de ser true, el valor del String se ignora.

El siguiente ejemplo muestra el código de verificación de desarrollo tecnológico:

Se pulsa el botón de la interfaz de desarrollar, que llama a tryAddTechnologyToQueue

```
public boolean touchDown(InputEvent event, float x, float y, int
pointer, int button)
{
    // Sonido
    playSound(m_buttonClick_sound, 0.5f);

    // Intentamos investigar y reportamos si fracaso
    String response =
        m_player.tryAddTechnologyToQueue(m_selectedTech);
    if (response != null)
        showDialog(response);
    // Actualizar UI
    updateUI();

    return true;
}
```

tryAddTechnologyToQueue realiza la comprobación de si se puede desarrollar llamando a canDevelopTechnology, si esta comprobación es incorrecta, devolverá el error al método anterior que se encargará de mostrarlo en un dialog, si no es incorrecta, gastará el dinero del país y encolará la tecnología

```

public String tryAddTechnologyToQueue(Technology tech)
{
    // Comprobamos si se puede desarrollar
    TryResponse tryDevelop = canDevelopTechnology(tech);

    // No se puede desarrollar, alertar si es el jugador
    if (!tryDevelop.getResult())
        return tryDevelop.getResponse();

    // Gastamos el dinero
    wasteMoney(tech.getCost());

    // Iniciamos construccion
    addTechnologyToQueue(new Queue(tech, tech.getProdTime()));

    // Log debug IA
    if (!equals(Game.m_player))
        Log.d("IA Debug", getName() + " developed " + tech.getName());

    // Actualizamos interfaz
    return null;
}

public TryResponse canDevelopTechnology(Technology tech)
{
    // La tecnologia no existe
    if (tech == null)
        return new TryResponse("Technology does not exist.", false);

    // El que la intenta desarrollar ya la tiene
    if (getCountryDevelopedTechnologyByType(tech.getType()) != null)
        return new TryResponse("You already know that technology",
                                false);

    // Comprobamos si ya tiene tecnologias en produccion, abortar.
    if (hasAnyTechnologyInQueue())
        return new TryResponse("There is another technology in
                                queue.", false);

    // Comprobamos cultura
    if (tech.getRequiredCulture() != null && tech.getRequiredCulture()
        != getCulture())
        return new TryResponse("This technology is not available to
                                your culture.", false);

    // Comprobamos que tiene las tecnologias requeridas
    boolean canDevelop = true;
    if (tech.getRequiredTechnologies() != null)
    {
        for (Technology reqTech : tech.getRequiredTechnologies())
            if (getCountryDevelopedTechnologyByType(reqTech.getType())
                == null)
                canDevelop = false;
    }
    if (!canDevelop)
        return new TryResponse("You have not the required technology
                                to develop that.", false);
}

```

```

    if (!hasEnoughMoney(tech.getCost()))
        return new TryResponse("You dont have enough money to develop
                                that.", false);

    // Llegados a este punto, all correcto
    return new TryResponse("", true);
}

```

Si desde la interfaz queremos saber si una tecnologia se puede desarrollar para establecer el visual de su botón de construcción bastará con hacer un `canDevelopTechnology.getResult()` desde el método de actualización de interfaz `updateUI()` (`updateUITech()` en este caso particular). Este sistema se aplica exactamente del mismo modo para las otras dos producciones que se implementan en el juego (tropas y edificio)

Se trata de un sistema muy elegante, ya que se adapta perfectamente a todas las producciones del juego y permite al usuario ver de forma rápida cuando no puede producir algo a la vez que le informa de la causa.

Sistema de colas

El sistema de colas, al igual que el de verificación previamente expuesto, se trata de una implementación general que sirve para gestionar las colas de las tres principales producciones, para ello implementamos la clase `Queue` que gestiona las colas de forma genérica:

```

public class Queue <T>
{
    T m_item;
    int m_remainingTime;
    int m_totalTime;

    public Queue(T item, int remainingTime)
    {
        m_item = item;
        m_remainingTime = remainingTime;

        // El total es igual al restante en el momento de instanciar
        m_totalTime = remainingTime;
    }

    public int getTotalTime()
    {
        return m_totalTime;
    }

    public void setTotalTime(int totalTime)
    {
        m_totalTime = totalTime;
    }
}

```

```

public Object getItem()
{
    return m_item;
}

public int getRemainingTime()
{
    return m_remainingTime;
}

public void decRemainingTime()
{
    m_remainingTime--;
}
}

```

Una vez se ha verificado la posibilidad de iniciar una producción, se creará un nuevo objeto queue que se añadirá a un array de queues que se encuentra en la clase que procesará esas colas, en el ejemplo anterior de tecnologías:

```
addTechnologyToQueue(new Queue(tech, tech.getProdTime()));
```

```

public void addTechnologyToQueue(Queue techQ)
{
    m_investigationQueue.add(techQ);
}

```

Como hemos visto, un objeto cola sólo almacena el ítem a producir, el tiempo inicial y un contador entero, por lo que el tiempo que representan estos valores son “turnos” (días en el contexto del juego) y cuando se realiza un update del juego (uno por turno/día de juego) se decrementará este contador, realizando la producción y destruyendo el objeto cola cuando llegue a cero como se puede apreciar en el siguiente fragmento:

```

// Método de update de country
public void update()
{
    // Update cola de tecnologías
    updateTechQueues();

    (...)
}

```

```

private void updateTechQueues()
{
    // Update a cola de edificios si los hay
    if (m_investigationQueue.size() > 0)
    {
        Queue techQueue = m_investigationQueue.get(0); // Primer
                                                         elemento

        // Decrementamos cola
        techQueue.decRemainingTime();

        // Barra de progreso
        int totalTime = techQueue.getTotalTime();
        int remainTime = techQueue.getRemainingTime();
        (...)

        // Si la cola esta terminada, ejecutar construccion y eliminar
        cola asociada.
        if (techQueue.getRemainingTime() < 0 || (Game.m_debugMode &&
                                                equals(Game.m_player)))
        {
            (...)
            // Desarrollamos tecnologia
            Technology tech = (Technology)techQueue.getItem();
            if (tech != null)
            {
                addTechnology(tech);
                finishedInvestigatingTechnology(tech);
            }

            // Eliminamos el primer elemento de la cola, desplazando
            toda la cola hacia la izquierda
            m_investigationQueue.remove(0);
        }
    }
}

```

Las colas y su progreso se muestran además en la interfaz del juego. En la siguiente imagen se pueden apreciar colas de construcción, de desarrollo tecnológico y de entrenamiento de tropas cerca de sus interfaces correspondientes.



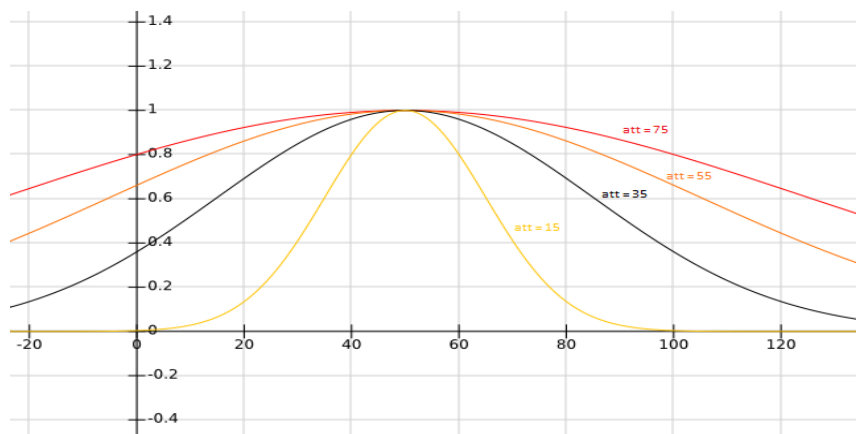
En el sistema del juego actual sólo se permite encolar un elemento simultáneamente ya que la propia comprobación de las producciones impide que el usuario pueda añadir nuevos elementos a la cola si la cola no está vacía, pero se puede apreciar que el sistema de colas está diseñado en principio para aceptar colas infinitas, tratándose de un diseño muy versátil.

3.9 DIPLOMACIA

En este apartado vamos a proceder a exponer cómo funciona el sistema diplomático de DomiNations para luego indagar en su implementación:

- Todo país posee un valor diplomático con el resto de países.
- La diplomacia se representa por un número entre 0 y 100.
- Dicho valor diplomático determina la relación entre los países según las siguientes equivalencias (que son paramétricas y pueden ser modificadas)
 - Hostil: 0-24
 - Neutral: 25-49
 - Amistoso: 49-74
 - Aliado: 75-100
- La relación diplomática es recíproca.
- Las distintas acciones de cada país tendrán un efecto sobre dicho valor, el impacto de cada acción sobre la diplomacia variará según la relación de los países.
- Los cambios diplomáticos estarán atenuados según una distribución gaussiana, de forma que tendrán menos efecto cuanto más cerca esté de los extremos la relación actual.
 - La función gaussiana actúa multiplicativamente sobre el valor base según la siguiente expresión:

- $$final_diplo_mod = diplo_mod \cdot e^{-\frac{current_diplo - neutral_diplo^2}{2 \cdot att^2}}$$
- La siguiente imagen muestra la curva del componente exponencial para una diplomacia neutral de 50 y distintos valores de atenuación:

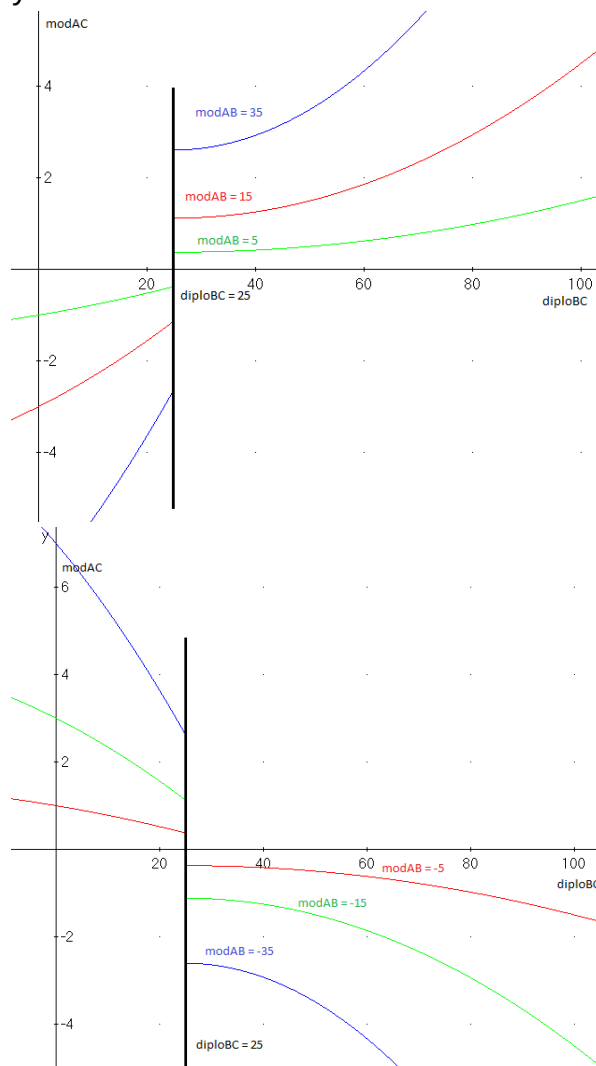


- El parámetro de atenuación en el juego actualmente es 35, lo cual significa que para países cuya relación actual se encuentre en los extremos (0, 100) una acción diplomática entre ambos que supusiera una ganancia o pérdida de 10 puntos, sería atenuada hasta un 35% de su valor, resultando ser 3.5 o -3.5.

- Los cambios diplomáticos se extenderán (spread) modificando la diplomacia no sólo entre los países afectados sino también con los demás. De forma que, si un país A declara la guerra a un país B, su diplomacia empeorará con todos los países aliados del país B y mejorará con los países enemigos del país B: la modificación de la diplomacia entre países A y B causará una modificación de la diplomacia entre los países A y C donde C son todos los países que se relacionan con B. La relación se expresa según la siguiente expresión:

- $diploMod_{AC} =$

$$\begin{cases} diploMod_{AB} \cdot (0.000133 \cdot diplo_{BC}^2 - 0.0066 \cdot diplo_{BC} + 0.33) \cdot 0.3 & \text{si } diplo_{BC} < warDiplomacy \\ -diploMod_{AB} \cdot (-0.0004000 \cdot diplo_{BC}^2 - 0.04000 \cdot diplo_{BC} + 2) \cdot 0.1 & \text{si } diplo_{BC} \geq warDiplomacy \end{cases}$$
- El resultado de la expresión anterior se multiplicará además por 2 si es negativo, de modo que la diplomacia en spread baja mucho más de lo que sube.
- La siguiente imagen muestra el valor de modificación de la diplomacia entre los países A y C en función de la diplomacia entre B y C para distintos valores de modificación de diplomacia entre A y B:



- La disposición de las curvas sugiere un comportamiento coherente: la relación diplomática que se extiende hacia otros países es proporcional a la modificación entre los dos países origen, estabilizándose en torno a la diplomacia de guerra y cambiando de signo si esta ocurre, de modo que una mejora diplomática con un país provocará una pérdida de diplomacia con sus países enemigos, y a la inversa si la modificación de diplomacia origen fuera negativa como se muestra en la gráfica derecha.

Una vez hemos expuesto el funcionamiento diplomático vamos a detallar la implementación. Todos los países poseen un HashMap que relaciona un par Country, Float; de esta forma tenemos la relación que cada país tiene con los demás, los valores iniciales se establecen en la inicialización como se expuso antes pero que vamos a analizar con un mayor detalle en esta parte:

```
private void initializeDiplomacy()
{
    // Base: Todos los países son neutrales entre si
    for (Country country : m_countryList.values())
        for (Country countryDest : m_countryList.values())
            setCountriesDiplomacy(country, countryDest,
                                  m_neutralDiplomacy);

    // Incluimos aqui alianzas reales
    String[] otan = new String[] {"Belgium", "Canada", "Denmark",
                                   "France", "Iceland", "Italy", "Luxembourg", "Netherlands",
                                   "Norway", "Portugal", "United Kingdom", "United States",
                                   "Greece", "Turkey", "Germany", "Spain", "Czech Republic",
                                   "Hungary", "Poland", "Bulgaria", "Estonia", "Latvia",
                                   "Lithuania", "Romania", "Slovakia", "Slovenia", "Albania",
                                   "Croatia"};
    String[] abca = new String[] {"United States", "United Kingdom",
                                   "Canada", "Australia"};

    String[] zpcas = new String[] {"Angola", "Argentina", "Benin",
                                    "Brazil", "Cameroon", "Cape Verde", "Congo",
                                    "Congo", "Ivory Coast", "Equatorial Guinea", "Gabon", "Gambia",
                                    "Ghana", "Guinea", "Guinea-Bissau",
                                    "Liberia", "Namibia", "Nigeria", "Senegal", "Sierra Leone",
                                    "South Africa", "Togo", "Uruguay"};
    String[] tiar = new String[] {"Argentina", "Bahamas", "Brazil",
                                   "Chile", "Colombia", "Costa Rica",
                                   "Dominican R", "El Salvador", "Guatemala", "Haiti", "Honduras",
                                   "Panama", "Paraguay", "Peru",
                                   "Trinidad Tobago", "United States", "Uruguay", "Bolivia",
                                   "Ecuador", "Nicaragua", "Mexico", "Venezuela"};
    String[] anzus = new String[] {"United States", "New Zealand",
                                   "Australia"};
    String[] tra = new String[] {"United States", "Taiwan"};
    String[] sk_us = new String[] {"United States", "South Korea"};
    String[] ph_us = new String[] {"United States", "Philippines"};
    String[] anpo = new String[] {"United States", "Japan"};
```

```

String[] matf = new String[] {"United States", "Morocco"};
String[] arab_league = new String[] {"Algeria", "Comoros",
    "Djibouti", "Egypt", "Iraq", "Jordan", "Kuwait",
    "Lebanon", "Libya", "Mauritania", "Morocco", "Oman", "Saudi
    Arabia", "Somalia", "Sudan", "Tunisia",
    "UA Emirates", "Yemen"};
String[] european_union = new String[] {"Austria", "Belgium",
    "Bulgaria", "Croatia", "Cyprus", "Czech
    Republic", "Denmark", "Estonia", "Finland", "France",
    "Germany", "Greece", "Hungary", "Ireland",
    "Italy", "Latvia", "Lithuania", "Luxembourg", "Netherlands",
    "Poland", "Portugal", "Romania",
    "Slovakia", "Slovenia", "Spain", "Sweden", "United Kingdom"};
String[] otsc = new String[] {"Armenia", "Belarus", "Kazakhstan",
    "Kyrgyzstan", "Russia", "Tajikistan"};
String[] ocs = new String[] {"China", "Kazakhstan", "Kyrgyzstan",
    "Russia", "Tajikistan", "Uzbekistan",
    "Pakistan", "India"};
String[] takm = new String[] {"Turkey", "Azerbaijan",
    "Kyrgyzstan", "Mongolia"};

// Creamos alianza
addAlliance(80, GetCountryListByNames(otan));
addAlliance(70, GetCountryListByNames(abca));
addAlliance(75, GetCountryListByNames(zpcas));
addAlliance(70, GetCountryListByNames(tiar));
addAlliance(65, GetCountryListByNames(anzus));
addAlliance(65, GetCountryListByNames(tra));
addAlliance(65, GetCountryListByNames(sk_us));
addAlliance(65, GetCountryListByNames(ph_us));
addAlliance(65, GetCountryListByNames(anpo));
addAlliance(62, GetCountryListByNames(matf));
addAlliance(80, GetCountryListByNames(arab_league));
addAlliance(74, GetCountryListByNames(european_union));
addAlliance(65, GetCountryListByNames(otsc));
addAlliance(70, GetCountryListByNames(ocs));
addAlliance(70, GetCountryListByNames(takm));

// Establecemos relaciones segun alianzas
for (Alliance alliance : m_alliances)
{
    ArrayList<Country> countries = alliance.getCountryNameList();
    int strength = alliance.getStrength();

    // Barremos países
    for (Country country : countries)
        for (Country countryDest : countries) // Barremos todos
            los países de la alianza para el país actual,
            garantizando el X*X
        {
            float diplo = country.getDiplomacyValueWithCountry
                (countryDest);

            if (diplo == -1)
                continue;
        }
    }
}

```

```

        if (diplo < strength) // Establecemos la diplomacia en
            el valor de fuerza de la alianza si esta es
            mayor al valor previo (la alianza mas fuerte
            prevalece)
            setCountriesDiplomacy(country, countryDest,
                                strength);
    }

    // Dispersion aleatoria en todos los países (entre -8 y +8 de
    aleatoriedad, como se reevalua el barrido dos
    veces al ser reciproco, la maxima dispersion es de -16 y +16
    // Base: Todos los países son neutrales entre si
    for (Country country : m_countryList.values())
        for (Country countryDest : m_countryList.values())
        {
            float diplo =
                country.getDiplomacyValueWithCountry(countryDest);
            if (diplo == -1)
                continue;
            float newValue = diplo + Utils.getRandomValue(-8, 8);
            setCountriesDiplomacy(country, countryDest, newValue);
        }
}

```

Podemos apreciar que el algoritmo de inicialización realiza las siguientes acciones:

1. Establece la diplomacia de todos los países en el valor neutral, 50 por defecto.
2. Define una serie de alianzas con un valor numérico y una lista de países que pertenecen a ella, después asignará a todos los países de cada alianza dicho valor numero al valor diplomático entre todos ellos. Si dos países se encuentran en varias alianzas el valor numérico final será el mayor entre las alianzas a las que pertenezca.
3. Realiza un nuevo barrido de las diplomacias de todos los países para sumarle un valor aleatorio que oscilará entre -8 y 8 para desuniformizar la inicialización y darle una cierta aleatoriedad a la partida.

Existen dos métodos para operar sobre la diplomacia de dos países:

- setCountriesDiplomacy(Country country, Country countryDest, float value)
 - Establecerá forzosamente el valor diplomático entre esos dos países, sólo se usa cuando se desea establecer dicho valor sin ningún efecto ni condición adicional, por ejemplo, en la inicialización.
- modifyDiplomacyValueWithCountry(Country targetCountry, float baseAddValue, boolean spread, boolean atenuation, boolean ignoreStrengths)
 - Modificará la relación entre el país desde donde se llama y el targetCountry en el valor baseAddValue proporcionado, procesándolo según todas las reglas definidas antes si los booleans están activos.
 - Este es el método principal que emplearemos para modificar la diplomacia, ya que al final realiza una llamada a setCountriesDiplomacy para establecer el valor final.

El método modifyDiplomacy implementa todas las reglas que hemos expuesto anteriormente:

```
public void modifyDiplomacyValueWithCountry(Country targetCountry,
float baseAddValue, boolean spread, boolean atenuation, boolean
ignoreStrengths)
{
    // Atenuamos el efecto de la diplomacia en los bordes (Se atenuan
    // segun una distribucion gaussiana que tiene su maximo en 50
    if (atenuation)
    {
        float diplo = getDiplomacyValueWithCountry(targetCountry);
        if (diplo != -1)
        {
            double gaussFactor =
                Formulas.getGaussAtenuationToDiplomacyGain(diplo);
            baseAddValue *= gaussFactor;
        }
    }

    // Aquí aplicamos las diferentes bonificaciones que hay al efecto
    // de la diplomacia.
    if (!ignoreStrengths)
    {
        // Variable output
        int diplomacyValueStrenght = 0;
```

```

// Add bonos de tecnologias
// Cultural Summit: "Strengthen your relations with all
    foreign countries by %v%."
Technology culturalSummit1 =
    getCountryDevelopedTechnologyByType(
        TechType.TECH_CULTURAL_SUMMIT_I);
if (culturalSummit1 != null)
    diplomacyValueStrenght += culturalSummit1.getValue();

Technology culturalSummit2 =
    getCountryDevelopedTechnologyByType(
        TechType.TECH_CULTURAL_SUMMIT_II);
if (culturalSummit2 != null)
    diplomacyValueStrenght += culturalSummit2.getValue();

Technology culturalSummit3 =
    getCountryDevelopedTechnologyByType(
        TechType.TECH_CULTURAL_SUMMIT_III);
if (culturalSummit3 != null)
    diplomacyValueStrenght += culturalSummit3.getValue();

// Diplomatic Reinforcement: "Strengthen your relations with
all foreign countries by %v%."
Technology diplomaticReinforcement1 =
    getCountryDevelopedTechnologyByType(
        TechType.TECH_DIPLOMATIC_REINFORCEMENT_I);
if (diplomaticReinforcement1 != null)
    diplomacyValueStrenght +=
        diplomaticReinforcement1.getValue();
Technology diplomaticReinforcement2 =
    getCountryDevelopedTechnologyByType(
        TechType.TECH_DIPLOMATIC_REINFORCEMENT_II);
if (diplomaticReinforcement2 != null)
    diplomacyValueStrenght +=
        diplomaticReinforcement2.getValue();

Technology diplomaticReinforcement3 =
    getCountryDevelopedTechnologyByType(
        TechType.TECH_DIPLOMATIC_REINFORCEMENT_III);
if (diplomaticReinforcement3 != null)
    diplomacyValueStrenght +=
        diplomaticReinforcement3.getValue();

// Si estamos perdiendo valor el fortalecimiento reducira el
    valor perdido.
if (baseAddValue < 0)
    baseAddValue -= Utils.ApplyPctToValue(baseAddValue,
        diplomacyValueStrenght);
else // Si estamos ganando valor, el fortalecimiento aumentara
    el valor ganado.
    baseAddValue += Utils.ApplyPctToValue(baseAddValue,
        diplomacyValueStrenght);
}

```

```

// Establecemos diplomacia
float diplo = getDiplomacyValueWithCountry(targetCountry);

(...)

// Establecemos diplomacia
Game.setCountriesDiplomacy(targetCountry, this, newValue);

(...)

// Si la bajada o subida no debe propagarse, abortamos aqui
if (!spread)
    return;

// Aplicamos el efecto de la modificacion de diplomacia sobre
// todos los demas paises
for (Country spreadedCountry : Game.getCountryList().values())
{
    // No aplicar spread a tu propio pais
    if (spreadedCountry.equals(this))
        continue;

    // No aplicar si es el pais target inicial
    if (spreadedCountry.equals(targetCountry))
        continue;

    float diploS =
    targetCountry.getDiplomacyValueWithCountry(spreadedCountry);
    (...)
    float diplomacySpreadValue =
        Formulas.getDiplomacySpreadValue(baseAddValue, diploS);

    // Nuevo valor se suma a la diplomacia previa con el pais de
    // spread
    float newSpreadValue =
        getDiplomacyValueWithCountry(spreadedCountry) +
        diplomacySpreadValue;

    (...)

    // Establecemos diplomacia
    Game.setCountriesDiplomacy(spreadedCountry, this,
        newSpreadValue);

    (...)
}
}

```

El método anterior llama para la aplicación de las fórmulas de las que se ha hablado a una clase externa llamada Formula, es en ella donde hemos ubicado todas las formas que influyen en el gameplay para poder ser ajustadas allí cómodamente.

```

public static double getGaussAtenuationToDiplomacyGain(float
countriesDiplomacy)
{
    // Valor maximo de atenuacion (en este caso x0.35)
    float offSetAtenuation = 35;
    double output = 1f * Math.exp(-(Math.pow(countriesDiplomacy -
        Game.m_neutralDiplomacy, 2f))/(2f *
        Math.pow(offSetAtenuation, 2f))));
    return output;
}

public static float getDiplomacySpreadValue(float baseAddValue, float
spreadCountryysDiplomacy)
{
    boolean simetricRelation = true;
    if (spreadCountryysDiplomacy < Game.m_warDiplomacy)
        simetricRelation = false;

    float output = 0;
    // Proporción parabolica final = inicial * 0.25 si ambos países
    son totalmente neutrales y inicial * 1 si
    estan en los extremos (proporción cuadrática entre medias)
    // Si la relación no es simétrica (el país B y el C son hostiles)
    el spread entre A y C debe ser inverso al
    habitual.
    // Si la relación no es simétrica, el efecto está atenuado a la
    mitad.
    if (simetricRelation)
        output = (float) (baseAddValue * (0.000133f *
            Math.pow(spreadCountryysDiplomacy, 2) - 0.0066f *
            spreadCountryysDiplomacy + 0.33f) * 0.30f);
    else // La relación no es simétrica, movimiento opuesto
        output = (float) (-baseAddValue * (-0.0004000f *
            Math.pow(spreadCountryysDiplomacy, 2) + 0.04000f *
            spreadCountryysDiplomacy + 2f) * 0.10f);

    // Si la diplomacia está bajando, el efecto es mayor (las bajadas
    son más grandes que las subidas y cuesta más recuperarse)
    if (output < 0)
        output *= 2f;

    return output;
}

```

En la siguiente tabla se muestran las acciones que producen un efecto sobre la diplomacia y cual es dicho efecto:

Evento	Calculo e implementación del efecto
Batalla con bajas	Si se produce una batalla en la que mueren tropas, se producirá un efecto diplomático negativo entre los propietarios de las tropas atacantes y los propietarios de las tropas defensoras que dependerá de los puntos totales de las tropas perdidas, según la siguiente expresión: $diploMod = -\frac{puntos_bajas}{1.000.000}$ Dicho cálculo es implementado por el siguiente método de la clase fórmulas: <pre>// -1 punto de damage por cada 1M de puntos de AP+DI+DA perdidos en total getDiplomacyImpactAfterBattle(float casualtiesPoints) { // Calculamos float output = -casualtiesPoints / 1000000f; // Devolvemos return output; }</pre>
Evento aleatorio	Existen eventos aleatorios implementados en el juego que provocan que en cada turno/día haya una probabilidad de que ocurra una mejora o empobrecimiento diplomático entre dos países aleatorios del juego, el valor de efecto del evento es también aleatorio y oscila entre 5 y 15 para ambos casos.
Acción diplomática	Una acción diplomática provocará una modificación de la diplomacia entre varios países de exactamente el valor que se defina en la acción, por lo que anula la mayoría de parámetros del método (atenuación, fuerza...) aunque los tiene en cuenta para calcular el coste de la acción.
Conquista de territorio	Una conquista provocará una bajada de 50 puntos en la diplomacia entre el invasor y el defensor, la cual está sujeta al efecto del spread y por tanto influirá radicalmente en el posicionamiento diplomático del atacante.

Tropas atravesand o territorios	Si unas tropas atraviesan un territorio, tendrá un impacto sobre la relación diplomática del dueño de las tropas y el del territorio que atraviesan, su efecto es diferente en función de la relación diplomática entre ambos países y respeta las siguientes expresiones: • Si son aliados: $diploMod = \frac{puntos_{tropas}}{10.000.000}$ • Si son amistosos: $diploMod = -2 \cdot \frac{puntos_{tropas}}{1.000.000}$ • Si son neutrales: $diploMod = -4 \cdot \frac{puntos_{tropas}}{1.000.000}$ • Si son hostiles: $diploMod = -4 \cdot \frac{puntos_{tropas}}{1.000.000}$ Podemos observar que se trata de un comportamiento coherente, ya que la presencia de tropas en un territorio aliado contribuye a su defensa, lo cual mejora las relaciones; en cualquier otro caso, podría ser tomado como una amenaza, por lo que tiene un efecto negativo inversamente proporcional a la diplomacia entre ambos países. El método que implementa estos efectos se encuentra también en la clase Formulas y se muestra a continuación: <pre> getDiplomacyImpactOfCrossingAnotherCountryWithTroops(float totalTroopPoints, Country troopsOwner, Country territoryOwner) { float output = 0; if (territoryOwner.isAlliedWith(troopsOwner)) { // Cada 10000k AP + DP sube 1 punto por dia output = totalTroopPoints / 10000000; } else if (territoryOwner.isFriendlyWith(troopsOwner)) { // Cada 1000k AP + DP baja 2 punto por dia output = -2 * totalTroopPoints / 1000000; } else if (territoryOwner.isNeutralWith(troopsOwner)) { // Cada 1000k AP + DP baja 4 puntos por dia output = -4 * totalTroopPoints / 1000000; } else if (territoryOwner.isHostileWith(troopsOwner)) { // Cada 1000k AP + DP baja 5 puntos por dia output = -8 * totalTroopPoints / 1000000; } // Devolvemos return output; } </pre>
---------------------------------------	--

Acciones diplomáticas

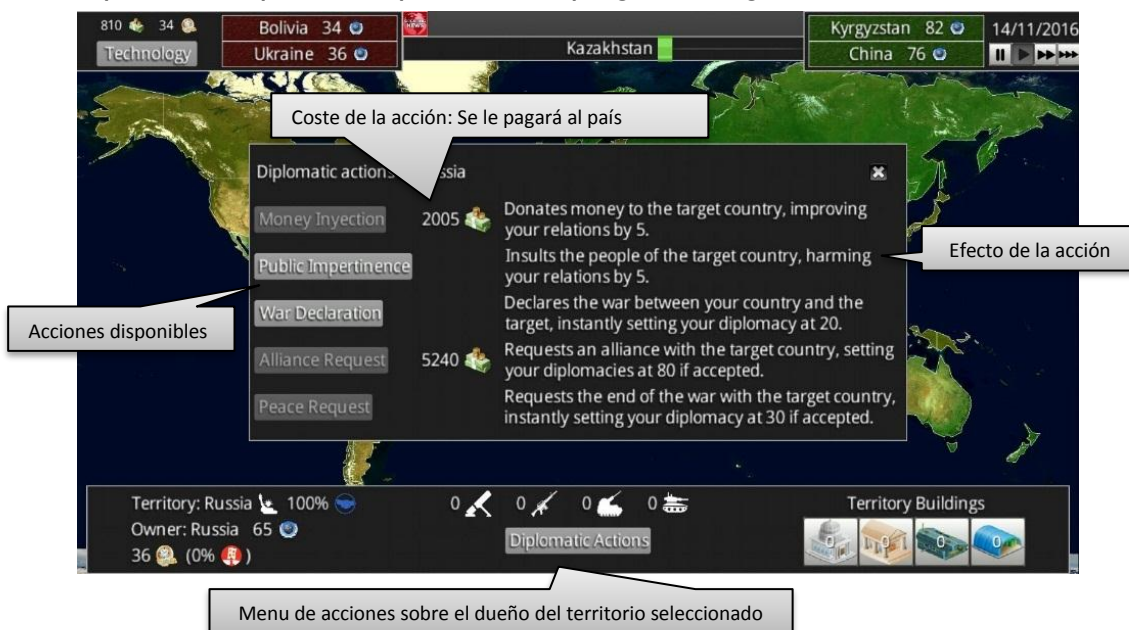
Las acciones diplomáticas son acciones voluntarias que pueden realizar los países para modificar su estado diplomático con otro país, la siguiente tabla muestra las opciones disponibles:

Tipo	Nombre	Descripción	Valor
TYPE_IMPROVE_DIPLOMACY	Money Injection	Donates money to the target country, improving your relations by %v.	5
TYPE_HARM_DIPLOMACY	Public Impertinence	Insults the people of the target country, harming your relations by %v.	5
TYPE_DECLARE_WAR	War Declaration	Declares the war between your country and the target, instantly setting your diplomacy at %v.	20
TYPE_COALITION_REQUEST	Alliance Request	Requests an alliance with the target country, setting your diplomacies at %v if accepted.	80
TYPE_PEACE_REQUEST	Peace Request	Requests the end of the war with the target country, instantly setting your diplomacy at %v if accepted.	30

Para implementar el sistema en primer lugar inicializamos las acciones que se muestran en la tabla anterior:

```
public void initializeDiplomaticActions()
{
    addDiplomaticAction(DiplomaticActionType.TYPE_IMPROVE_DIPLOMACY,
        "Money Injection", "Donates money to the target country,
        improving your relations by %v.", 5);
    addDiplomaticAction(DiplomaticActionType.TYPE_HARM_DIPLOMACY,
        "Public Impertinence", "Insults the people of the target
        country, harming your relations by %v.", 5);
    addDiplomaticAction(DiplomaticActionType.TYPE_DECLARE_WAR, "War
        Declaration", "Declares the war between your country and the
        target, instantly setting your diplomacy at %v.", 20);
    addDiplomaticAction(DiplomaticActionType.TYPE_COALITION_REQUEST,
        "Alliance Request", "Requests an alliance with the target
        country, setting your diplomacies at %v if accepted.", 80);
    addDiplomaticAction(DiplomaticActionType.TYPE_PEACE_REQUEST,
        "Peace Request", "Requests the end of the war with the target
        country, instantly setting your diplomacy at %v if accepted.",
        30);
}
```

Desde la interfaz de usuario (barra inferior) tenemos el botón de acción diplomática que al ser pulsado desplegará el siguiente menú al usuario:



Dicha interfaz se implementa del siguiente modo:

```
(...)  
m_tableDiplo.add(new Label("Diplomatic actions to " +  
m_selectedTerritory.getName(), defaultSkin)).left();  
m_tableDiplo.add(diploCrossButton).right().top().size(troopButtonWidth  
/ 1.5f, troopButtonHeight / 1.5f);  
m_tableDiplo.row();  
Table inTable = new Table();  
for (final DiplomaticAction action : m_diplomaticActions)  
{  
    // Add botón  
    TextButton button = new TextButton(action.getName(), defaultSkin);  
    button.addListener(new InputListener() {  
        @Override  
        public boolean touchDown(InputEvent event, float x, float y,  
            int pointer, int button)  
        {  
            // Sonido  
            Game.playSound(m_buttonClick_sound, 0.5f);  
  
            String response = action.tryExecuteAction(m_player,  
                                                        target, true);  
  
            if (response != null)  
                showDialog(response);  
  
            // Actualizamos  
            updateUI();  
  
            return true;  
        }  
    });  
    inTable.add(button).left().padRight(smallPad);  
  
    // Coste  
    int cost = action.getCost(m_player, target);  
    (...)  
  
    Label description = new Label(action.getDisplayDescription(),  
                                   defaultSkin);  
    (...)  
    inTable.add(description).width(descriptWidth);  
    inTable.row();  
  
    // Transparencia del botón si no es pulsable  
    if (!action.canBeExecuted(m_player, target).getResult())  
        button.setColor(1, 1, 1, 0.5f);  
    else  
        button.setColor(1, 1, 1, 1);  
}  
  
// Tabla interior  
m_tableDiplo.add(inTable).left().colspan(2).padBottom(smallPad);  
  
// Add Tabla final  
m_outTableDiplo.add(m_tableDiplo);  
stageUIDiplo.addActor(m_outTableDiplo);  
}
```

Observamos ya desde este punto que esta implementación está también soportada por el sistema de TryResponse que hemos comentado algunos apartados atrás. Al pulsar el botón de la acción ejecutamos tryExecuteAction, en la primera parte de dicho método apreciamos como hace una llamada a un método de comprobación de ejecución similar al de la construcción de edificios o tecnologías:

```
public String tryExecuteAction(Country executor, Country target,
boolean spread)
{
    TryResponse tryExecute = canBeExecuted(executor, target);

    if (!tryExecute.getResult())
        return tryExecute.getResponse();

    (...)
}

public TryResponse canBeExecuted(Country executor, Country target)
{
    switch (getType())
    {
        case TYPE_IMPROVE_DIPLOMACY:
            if (executor.isHostileWith(target))
                return new TryResponse("You are at war with " +
                                         target.getName() + ".", false);
            if (executor.getDiplomacyValueWithCountry(target) > 100 -
                                                         getValue())
                return new TryResponse("Your diplomacy with " +
                                         target.getName() + " is too high.", false);
            break;
        case TYPE_HARM_DIPLOMACY:
            if (executor.getDiplomacyValueWithCountry(target) <
                                                         getValue())
                return new TryResponse("Your diplomacy with " +
                                         target.getName() + " is too low.", false);
            break;
        case TYPE_DECLARE_WAR:
            if (executor.isHostileWith(target))
                return new TryResponse("You are at already at war with
                                         " + target.getName() + ".", false);
            break;
        case TYPE_COALITION_REQUEST:
            if (executor.isHostileWith(target))
                return new TryResponse("You are at war with " +
                                         target.getName() + ".", false);
            if (executor.isAlliedWith(target))
                return new TryResponse("You are at already allied with
                                         " + target.getName() + ".", false);
            break;
        case TYPE_PEACE_REQUEST:
            if (!executor.isHostileWith(target))
                return new TryResponse("You are not at war with " +
                                         target.getName() + ".", false);
            break;
    }
}
```

```

    if (!executor.hasEnoughMoney(getCost(executor, target)))
        return new TryResponse("You dont have enough money to execute
                                the action.", false);

    // Llegados a este punto, permitimos
    return new TryResponse("", true);
}

```

La función `getCost` obtiene el coste que el país destino exige por aceptar dicho acuerdo, ese coste se calcula en función de la diferencia de poder entre ambos países, si un país muy débil pide un acuerdo a un país muy fuerte, este le exigirá un coste muy grande, el sistema pretende simular una negociación de términos en función de la necesidad de ambos participantes, la función `getCost` llama a un método de la clase `Formulas` que detallamos a continuación:

```

public static int getCostToAcceptADiplomacyRequestOfCountry
(DiplomaticAction action, Country requester, Country target)
{
    double cost = 0;
    float diploWeight = 10000f;
    float economyWeight = 100f;
    float militaryWeight = 200f;

    // AI
    if (target.getAI() == null)
    {
        diploWeight = 5000f;
        economyWeight = 200f;
        militaryWeight = 100f;
    }
    else
    {
        switch (target.getAI().getEffectiveType())
        {
            case AI_IMPERIALIST:
                diploWeight = 5000f;
                economyWeight = 300f;
                militaryWeight = 200f;
                break;
            case AI_AGGRESSIVE:
                diploWeight = 2000f;
                economyWeight = 100f;
                militaryWeight = 500f;
                break;
            case AI_DEFENSIVE:
                diploWeight = 10000f;
                economyWeight = 300f;
                militaryWeight = 600f;
                break;
            case AI_PEACEFUL:
                diploWeight = 5000f;
                economyWeight = 200f;
                militaryWeight = 100f;
                break;
        }
    }
}

```

```

// Logica: el coste aumenta si la diplomacia es baja (inversamente
prop), el coste aumenta si la diferencia de producciones
es alta (directamente prop), el coste aumenta si la diferencia
de tropas es alta (directamente prop)
// Logica: Si se realizan acciones compactas (coalicion o guerra)
hay una diferencia de coste con respecto a lo que costaria
subirlo o bajarlo de 5 en 5.
// Logica: Todas las acciones diplomaticas tienen un coste
aumentado si se está en guerra con el target. (afecta al
impacto de la diplomacia)
// Logica: La atenuacion de la diplomacia en los extremos afecta
proporcionalmente al coste en este caso (a mayor
atenuacion, mayor coste necesario)
// Relacion diplomatica
float relations = target.getDiplomacyValueWithCountry(requester);
if (relations < 1)
    relations = 1;

// Si estan en guerra, la diplomacia pesará mas
int penalization = 1;
if (relations < 25)
    penalization *= 1.5f;

// Atenuacion sufrida por las acciones que suben la diplomacia de
golpe, reducen el coste con respecto a poco a poco en un 10%.
float atenuation = 0.9f;

// Diferencias de valores entre ambos países.
float requestProd = requester.getTotalMoneyProduction();
if (requestProd < 1)
    requestProd = 1;
float requestTroopPoints = requester.getTotalTroopPoints();
if (requestTroopPoints < 1)
    requestTroopPoints = 1;
float productionDiff = (float)target.getTotalMoneyProduction() /
                        (float)requestProd;
float troopPowerDiff = (float)target.getTotalTroopPoints() /
                        (float)requestTroopPoints;

// Calculo de coste base para subir un punto
float baseCost = (diploWeight / relations) * penalization *
    (float)(1 / getGaussAtenuationToDiplomacyGain(relations)) +
    economyWeight * productionDiff + militaryWeight *
    troopPowerDiff;

```

```

switch (action.getType())
{
    case TYPE_HARM_DIPLOMACY:
        cost = 0;
        break;
    case TYPE_DECLARE_WAR:
        cost = 0;
        break;
    case TYPE_COALITION_REQUEST:
    case TYPE_PEACE_REQUEST:
        // Subimos %v - valor_actual puntos de diplomacia
        cost = baseCost * (action.getValue() - relations) *
                                                    atenuation;
        break;
    case TYPE_IMPROVE_DIPLOMACY:
        // Subimos %v puntos de diplomacia
        cost = (int)baseCost * action.getValue();
        break;
}

(...)
return (int)cost;
}

```

Algunos apuntes sobre el algoritmo de cálculo de coste:

- La función de cálculo de coste devuelve un valor numérico que representa el coste en dinero por punto de modificación de diplomacia.
- El coste exigido por el país destino aumentará cuanto mayor sea el cociente de poder militar entre país destino y país origen.
- El coste exigido por el país destino aumentará cuanto mayor sea el cociente entre las producciones económicas del país destino y el país origen.
- El coste exigido por el país destino aumentará cuanto peor sea la relación entre el país origen y el país destino, aumentando un 50% adicional si se encuentran en guerra.
- El peso de cada uno de los factores anterior (diferencia diplomática, diferencia productiva, diferencia militar) está determinado por el tipo de IA del país destino, un país con IA defensiva le dará más peso a la diferencia militar que uno pacífico, por ejemplo.
- Si se trata de una acción de establecimiento diplomático en vez de modificación (firmar alianza, solicitar paz) el coste es un 10% inferior que el equivalente a realizar una sucesión de mejoras diplomáticas hasta alcanzar el valor destino.
- Las acciones de dañar diplomacia o declarar guerra tienen evidentemente coste cero.
- Las IAs también realizan ofertas diplomáticas, y están implementadas de modo que realicen ofertas siempre que el precio exigido por el país destino sea inferior al que exigirían ellos ante una oferta similar. (lógica de conveniencia) No obstante es una lógica pendiente de revisar y mejorar de forma iterativa.

Y ahora analizando el método completo podemos apreciar que una vez verificada la posibilidad de ejecución, si el objetivo es el jugador mostraremos un mensaje preguntando si el usuario desea aceptar el intercambio, en caso de ser una IA, lo aceptará inmediatamente porque entendemos que se ha cumplido el criterio de conveniencia al evaluar el coste:

```
public String tryExecuteAction(Country executor, Country target,
boolean spread)
{
    TryResponse tryExecute = canBeExecuted(executor, target);

    if (!tryExecute.getResult())
        return tryExecute.getResponse();

    // Llegados a este punto, la accion puede ejecutarse
    // Coste
    int cost = getCost(executor, target);

    // Si el target es un player, debemos preguntarle si quiere
    //                                     aceptar.
    // No hacen ofertas a coste 0
    if (target.equals(Game.m_player))
    {
        String text = "";
        switch(m_type)
        {
            case TYPE_IMPROVE_DIPLOMACY:
                if (cost > 0)
                {
                    text = executor.getName() + " offers you " + cost
                        + "$ to improve your relations by " +
                        getValue() + ", will you accept the offer?";
                    Game.confirmDiplomaticExchange(text, executor,
                        target, cost, this, spread);
                }
                break;
            case TYPE_COALITION_REQUEST:
                if (cost > 0)
                {
                    text = executor.getName() + " offers you " + cost
                        + "$ to sign a coalition between your
                        countries, will you accept the offer?";
                    Game.confirmDiplomaticExchange(text, executor,
                        target, cost, this, spread);
                }
                break;
            case TYPE_PEACE_REQUEST:
                if (cost > 0)
                {
                    text = executor.getName() + " offers you " + cost
                        + "$ to sign the peace between your
                        countries, will you accept the offer?";
                    Game.confirmDiplomaticExchange(text, executor,
                        target, cost, this, spread);
                }
                break;
        }
    }
}
```

```

        // En este caso no se pregunta nada
        case TYPE_HARM_DIPLOMACY:
        case TYPE_DECLARE_WAR:
            executeDiplomaticAction(executor, target, cost,
                                    spread);

            break;
    }
}
else // Si el target es una IA siempre aceptara la oferta, pues es
      provechosa.
    executeDiplomaticAction(executor, target, cost, spread);
return null;
}

```

El mensaje que se muestra al usuario al llamar a `confirmDiplomaticExchange` ejecutará también una llamada a `executeDiplomaticAction` que se llama en el método anterior cuando es una IA el objetivo y realiza lo siguiente:

```

public void executeDiplomaticAction(Country executor, Country target,
int cost, boolean spread)
{
    executor.wasteMoney(cost);
    target.gainMoney(cost);
    // Ejecutamos efecto
    float actualRelation =
        executor.getDiplomacyValueWithCountry(target);
    String alert = "";
    switch(m_type)
    {
        case TYPE_IMPROVE_DIPLOMACY:
            executor.modifyDiplomacyValueWithCountry(target,
                getValue(), spread, false, true);

            break;
        case TYPE_HARM_DIPLOMACY:
            executor.modifyDiplomacyValueWithCountry(target, -
                getValue(), spread, false, true);
            alert = executor + " has insulted you at public TV and
                your relations has been damaged.";

            break;
        case TYPE_DECLARE_WAR:
            executor.modifyDiplomacyValueWithCountry(target, -
                (actualRelation - getValue()), spread, false, true);
            alert = executor.getName() + " has declared war on you,
                your relations has been deeply damaged.";
            Game.updateBreakingNews(executor.getName() + " has
                declared war to " + target.getName() + ", the
                international community disavows.");

            break;
        case TYPE_COALITION_REQUEST:
            executor.modifyDiplomacyValueWithCountry(target,
                (actualRelation - getValue()), spread, false, true);
            Game.updateBreakingNews(executor.getName() + " has signed
                an alliance with " + target.getName() + "."");

            break;
    }
}

```

```

        case TYPE_PEACE_REQUEST:
            executor.modifyDiplomacyValueWithCountry(target,
                (actualRelation - getValue()), spread, false, true);
            break;
    }

    // Alertamos si es jugador y hay mensaje que mostrar
    if (target.equals(Game.m_player) && !alert.equals(""))
        Game.showDialog(alert);
}

```

Para finalizar ejecutamos un modifyDiplomacy con el valor exacto que queremos establecer ignorando atenuaciones gaussianas para fijar el valor esperado y con esto finaliza la ejecución de la acción.

3.10 TROPAS, BATALLAS Y CONQUISTAS

Las tropas se implementan mediante una serie de clases que dan soporte a distintos elementos relacionadas con ellas:

- Unit: Implementa los tipos de unidades que existen en el juego y sus valores.
- Troops: Implementa una agrupación de unidades, entendida como un número de soldados de una determinada unidad, que pertenecen a un país y que se han entrenado en un territorio concreto.
- TroopArray: Implementa una agrupación de tropas, se emplea como si se tratase de un objeto lista de Troops, con la adición de métodos cómodos de adición y eliminación que facilitan la gestión de las tropas.
- TroopSquad: Implementa una agrupación de tropas en el mapa, que puede ser desplazada por él para realizar acciones sobre territorios.
- TroopTask: Implementa tareas para las tropas, lo utilizan las IAs para asignar tareas priorizadas a determinadas agrupaciones de tropas y gestionar así en qué emplea a sus tropas.

La siguiente tabla muestra las unidades que existen en el juego:

Tipo	Name	Desc	Time	Cost	Vel	Att	Def Inf	Def Art	Icon
MILITIA	Militia	Angry citizens of the invaded country who will fight against the conquerors to push out them of their country.	0.005	0.6	2	5	10	5	militia Icon
INFANTRY _MORTAR	Mortar Unit	Disciplined defensive unit especially effective against artillery thanks to his weapon but their offensive capabilities are very low.	0.01	1.0	8	20	35	60	mortar Icon
INFANTRY _ASSAULT	Assault Unit	Fierce offensive unit which will decimate any defensive opposition when striking enemies. Low defensive capabilities.	0.01	1.4	6	60	30	30	assault Icon
ARTILLERY _AKATSIYA	2S3 Akatsiya	Fast mobile artillery with high defensive capabilities against infantry, it's fast but not so good as offensive troop.	0.03	2.2	12	50	100	40	akatsiya Icon
ARTILLERY _ABRAMS	M1 Abrams	Heavy offensive tank capable of pierce the strongest defence without difficulty. It is also usable as defending unit.	0.03	3.2	10	150	50	75	abrams Icon
AIRCRAFT _BOMBER	Aircraft	Fast and expensive aircraft capable of bombing distant territories without taking so much damage, causing heavy damage to the target.	1	100	40	900	500	300	aircraft Icon
SHIP _WARSHIP	Frigate	Warship with both capabilities against another ships. It supports your troops improving their capabilities inside its influence radius.	200	30000	18	9000	5000	3000	frigate Icon

En toda batalla distinguiremos tropas atacantes y tropas defensoras, siendo las tropas defensoras todas las tropas propiedad de un país cuyo territorio está siendo atacada y también las de países aliados que se encuentren en el territorio atacado; y tropas atacantes aquellas que sean propiedad de un país enemigo del territorio atacado y que se encuentren atacando el territorio. Dicho esto, hemos podido apreciar en la tabla anterior que las unidades tienen distintos valores numéricos que llamaremos valores de tropa y que tienen los siguientes significados:

- Tiempo de entrenamiento: Tiempo en turnos que tarda una unidad en ser entrenada, son tiempos muy bajos debido a que las tropas se entrenan en agrupaciones de millar. Por ejemplo, un millar de morteros tardaría 10 turnos.
- Coste de entrenamiento: Coste en dinero por unidad para ser entrenada, son costes muy bajos debido a que las tropas se entrenan en agrupaciones de millar. Por ejemplo, un millar de morteros costaría 1000 unidades monetarias.
- Velocidad: La velocidad a la que la tropa se desplaza por el mapa.
- Ataque: El valor que representa la fuerza que poseen las tropas atacantes.
- Defensa contra infantería: El valor que representa la fuerza defensiva de la unidad contra atacantes de tipo infantería.
- Defensa contra artillería: El valor que representa la fuerza defensiva de la unidad contra atacantes de tipo artillería.

En relación a los tipos de defensa, consideramos infantería las siguientes tropas:

- Milicia
- Mortero
- Asalto

Y artillería las siguientes:

- Akatsiya
- Abrams
- Bombardero
- Fragata

La última unidad (Frigate) no se encuentra implementada en el juego y su adición se ha pospuesto para fases posteriores de desarrollo.

A continuación, mostramos algunas de las declaraciones más importantes de las clases Troops y TroopArray, que sirven para entender su razón de ser:

```
public class Troops
{
    private int m_number;
    private Unit m_unit;
    private Country m_owner;
    private Territory m_territory;

    (...)

    public int getTotalAttackPower(float troopNumberPct)

    public int getTotalInfantryDefencePower(float troopNumberPct,
                                              Territory battleTerritory)

    public int getTotalArtilleryDefencePower(float troopNumberPct,
                                              Territory battleTerritory)

    public int getTotalSpeed()
}

public class TroopArray
{
    private ArrayList<Troops> m_troops;
    private TroopTask m_task;

    (...)

    public boolean add(Troops addTroops)
    {
        // Buscamos si ya tenemos de esta tropa
        for (Troops troops : m_troops)
            if (troops.getUnit().equals(addTroops.getUnit()))
            {
                // Si ya la tiene, add el numero a la previa
                troops.setNumber(troops.getNumber() +
                                addTroops.getNumber());

                // Informamos de que ya existia
                return true;
            }

        // Si estamos aqui es que no teniamos esa tropa, la añadimos
        m_troops.add(new Troops(addTroops.getNumber(),
                                addTroops.getUnit(), addTroops.getOwner(),
                                addTroops.getTerritory()));

        // Informamos de que no existia
        return false;
    }
}
```

```

public boolean subtract(Troops subTroops)
{
    // Buscamos si ya tenemos de esta tropa
    for (Troops troops : m_troops)
        if (troops.getUnit().equals(subTroops.getUnit()))
        {
            // Si ya la tiene, add el numero a la previa
            troops.setNumber(troops.getNumber() -
                             subTroops.getNumber());

            // Si hay 0, ya no quedan tropas, retiramos el objeto.
            if (troops.getNumber() <= 0)
                m_troops.remove(troops);

            // Informamos de que se ha restado
            return true;
        }

    // Si estamos aqui es que no tiene tropas de estas, reportar
    fracaso.
    return false;
}

public int getTroopNumberByUnit(Unit unit)

public boolean hasTroops()

public float getTotalAttackPower()

public float getTotalDefensePower(Territory terry)

public float getTotalDefenseArtilleryPower(Territory terry)

public float getTotalDefenseInfantryPower(Territory terry)

public int getSlowestTroopSpeed()

(...)
}

```

Observamos que la clase TroopArray incluye elementos add y subtract que permiten que añadamos objetos Troop al array garantizando que dichas tropas se van a sumar numéricamente a otras tropas del mismo tipo de unidad que ya estuvieran contenidas en el array, y equivalente en el subtract con además una verificación y limpieza de objetos Tropas vacíos.

Entrenamiento de tropas



La imagen muestra la interfaz de producción de tropas, el número que se muestra en cada botón representa las tropas que se entrenarán de golpe en un click, las IAs pueden decidir dicho número, pero el usuario no. Este diseño pretende evitar que el usuario tenga que usar el teclado para definir el número de tropas a entrenar, estableciendo un número base adecuado. Se pretende incluir en el futuro un input gráfico en la interfaz para modificar ese valor en factores x10.

La implementación del sistema de entrenamiento se realiza exactamente igual que el de edificios o tecnologías visto anteriormente:

```
// Boton Train
trainButton = new TextButton("Train", defaultSkin, "default");
trainButton.addListener(new InputListener() {
    @Override
    public boolean touchDown(InputEvent event, float x, float y, int
        pointer, int button)
    {
        // Sonido
        playSound(m_buttonClick_sound, 0.5f);

        // Intentamos construir
        String response =
        m_selectedTerritory.tryAddTroopToQueue(m_selectedTroops);
        if (response != null)
            showDialog(response);

        // Actualizar UI
        updateUI();

        return true;
    }
});
```

```

public String tryAddTroopToQueue(Troops troops)
{
    // Comprobamos si se puede trainear
    TryResponse tryTrain = troopsCanBeTrained(troops);

    // No se puede trainear
    if (!tryTrain.getResult())
        return tryTrain.getResponse();

    // Gastamos el dinero
    getOwner().wasteMoney(Math.round(troops.getUnit().getCost() *
                                      troops.getNumber()));

    // Tiempo de train
    int trainTime = Math.round(troops.getUnit().getProdTime(m_owner) *
                              troops.getNumber());

    // Iniciamos produccion
    addTroopsToQueue(new Queue(troops, trainTime));

    // Log debug IA
    if (!m_owner.equals(Game.m_player))
        Log.d("IA Debug", m_owner.getName() + " trained " +
              troops.getUnit().getName() + "x" +
              String.valueOf(troops.getNumber()));

    return null;
}

public TryResponse troopsCanBeTrained(Troops troops)
{
    if (troops == null)
        return new TryResponse("Troops does not exists.", false);
    if (this == null)
        return new TryResponse("Territory does not exists.", false);

    // No es mi territorio
    if (!m_owner.equals(this.getOwner()))
        return new TryResponse("This is not your territory.", false);

    // Comprobamos si ya tiene en produccion, abortar.
    if (this.hasAnyTroopInQueue())
        return new TryResponse("There is another troop in queue.",
                                false);

    // Si la tecnologia no lo permite, abortar.
    if (troops.getUnit().getType() == UnitType.UNIT_SHIP_WARSHIP)
        return new TryResponse("Ships are not available.", false);

    if (troops.getUnit().getType() == UnitType.UNIT_AIRCRAFT_BOMBER)
    {
        Technology dronesResearching1 =
            this.getOwner().getCountryDevelopedTechnologyByType(TechType.TE
                                                                    CH_DRONES_RESEARCHING_I);

        if (dronesResearching1 == null)
            return new TryResponse("You need to develop a technology
                                    to build aircrafts.", false);
    }
}

```

```

if (!this.getOwner().hasEnoughMoney(Math.round(
    troops.getUnit().getCost() * troops.getNumber())))
    return new TryResponse("You dont have enough money to train
                           that.", false);

// Llegados a este punto, permitimos
return new TryResponse("", true);
}

```

Si se puede entrenar, las tropas (unidad y número) se añadirán a la cola de entrenamiento, el método de update de Territory actualizará la cola y cuando termine la producción realizará una de estas tres acciones:

- Si la unidad que compone la tropa es Frigate, hará aparecer una TroopSquad en una posición de agua aleatoria cercana en torno al país.
- Si la unidad que compone la tropa es Aircraft, hará aparecer una TroopSquad en una posición del país aleatoria cercana al centro.
- En cualquier otro caso, añadirá esa tropa a las tropas acuarteladas en el territorio, estas tropas pueden ser arrastradas mediante la interfaz de deploy al mapa, conformando así un TroopSquad.

Escuadrones de tropas

Los escuadrones de tropas (TroopSquads) representan agrupaciones o batallones de tropas que se encuentran en una posición del mapa, que pueden ser desplazadas por él y que provocan efectos sobre los territorios en los que se encuentran en cada update. Se implementan en la siguiente clase:

```

public class TroopSquad extends Actor
{
    private TroopArray m_troopArray;
    private Country m_owner;
    private SquadType m_type;
    private Table m_infoTableoOut;
    (...)
    private Stage stagePath;
    private Vector2 m_destiny;
    private boolean m_movingInRectLine;
    (...)

    @Override
    public void draw(Batch batch, float alpha)

    public void updateTable()
    (...)

    public Territory getTerritory()
    (...)

    public boolean isAffectedByTerrain()
    (...)
}

```

```

    public void retractSquadToTerritory(Territory terry, boolean
                                         toDeployList)

    public void sendSquad(final Vector2 dest, final boolean rectLine)
    (...)
}

```

Observamos que la clase extiende Actor, permitiendo dibujarla y desplazarla cómodamente por el mapa además de guardar los valores extras referentes a las tropas que necesitamos. Los siguientes métodos son los más importantes de la clase:

```

@Override
public void draw(Batch batch, float alpha)
{
    Sprite icon = null;
    // Icono depende del tipo
    switch (m_type)
    {
        case SQUAD_LAND:
            if (Game.m_player.equals(m_owner))
                icon = Game.getSpriteByName("squadPinLandMe");
            else if (Game.m_player.isAlliedWith(m_owner))
                icon = Game.getSpriteByName("squadPinLandAlly");
            else if (Game.m_player.isNeutralWith(m_owner))
                icon = Game.getSpriteByName("squadPinLandNeutral");
            else if (Game.m_player.isHostileWith(m_owner))
                icon = Game.getSpriteByName("squadPinLandHostile");
            break;
        case SQUAD_MILITIA:
            if (Game.m_player.equals(m_owner))
                icon = Game.getSpriteByName("squadPinMilitiaMe");
            else if (Game.m_player.isAlliedWith(m_owner))
                icon = Game.getSpriteByName("squadPinMilitiaAlly");
            else if (Game.m_player.isNeutralWith(m_owner))
                icon = Game.getSpriteByName("squadPinMilitiaNeutral");
            else if (Game.m_player.isHostileWith(m_owner))
                icon = Game.getSpriteByName("squadPinMilitiaHostile");
            break;
        case SQUAD_AIR:
            if (Game.m_player.equals(m_owner))
                icon = Game.getSpriteByName("squadPinAirMe");
            else if (Game.m_player.isAlliedWith(m_owner))
                icon = Game.getSpriteByName("squadPinAirAlly");
            else if (Game.m_player.isNeutralWith(m_owner))
                icon = Game.getSpriteByName("squadPinAirNeutral");
            else if (Game.m_player.isHostileWith(m_owner))
                icon = Game.getSpriteByName("squadPinAirHostile");
            break;
    }
}

```

```

    case SQUAD_WATER: // TODO
        if (Game.m_player.equals(m_owner))
            icon = Game.getSpriteByName("squadPinLandMe");
        else if (Game.m_player.isAlliedWith(m_owner))
            icon = Game.getSpriteByName("squadPinLandAlly");
        else if (Game.m_player.isNeutralWith(m_owner))
            icon = Game.getSpriteByName("squadPinLandNeutral");
        else if (Game.m_player.isHostileWith(m_owner))
            icon = Game.getSpriteByName("squadPinLandHostile");
        break;
    }

    // Cubrimos excepcion
    if (icon == null)
        return;

    // Actualizamos bounds
    setBounds(getX(), getY(), m_baseWidth * Game.mainUICamera.zoom,
              m_baseHeight * Game.mainUICamera.zoom);

    // Dibujamos squad
    batch.draw(icon, getX() - getWidth() / 2, getY(), getWidth(),
              getHeight());

    // Actualizamos longitud de tabla a la nueva posicion si se ve
    if (m_infoTableoOut.isVisible())
    {
        // Actualizamos tabla
        updateTable();

        // Dibujamos
        m_infoTableoOut.setPosition(.....)
    }
}

```

El método draw hace override al draw del actor original para dibujar con un icono diferente en función del tipo de squad y de la relación diplomática con el jugador, aquí algunos ejemplos de los iconos:



El método además también implementa el dibujado de un actor adicional, una tabla que se ubica sobre la squad que muestra las tropas que contiene, la visibilidad de dicha tabla puede alternarse pulsándola. El método updateTabla actualiza su valor y genera la tabla adecuada.

El método `getTerritory` obtiene el territorio sobre el que está la squad, para ello realiza una conversión algebraica de coordenadas que convierte de la referencia de la stage y la cámara a la posición absoluta de nodos para saber sobre cual se encuentra exactamente la squad:

```
public Territory getTerritory()
{
    int terryId = Utils.getTerritoryIdByMapPoint(new Vector2(getX(),
                                                                getY()));
    Territory territory = Game.getTerritoryById(terryId);
    if (territory != null)
        return territory;
    return null;
}

public static int getTerritoryIdByMapPoint(Vector2 point)
{
    float coordX = point.x + Game.windowW / 2;
    float coordY = point.y + Game.windowH / 2;
    return Game.m_mapPoints[Math.round((coordX /
                                         Game.m_nodeSize))] [Math.round((coordY / Game.m_nodeSize))] +
                                         128;
}
```

El método `isAffectedByTerrain` determina si la squad está afectada por el terreno, es decir, si el tipo de terreno que atraviesan debería influir en su velocidad:

```
public boolean isAffectedByTerrain()
{
    boolean affectedByTerrain = false;
    switch (getType())
    {
        case SQUAD_MILITIA:
        case SQUAD_LAND:
            affectedByTerrain = true;
            break;
        case SQUAD_AIR:
            affectedByTerrain = false;
            break;
        case SQUAD_WATER:
            affectedByTerrain = false;
            break;
    }
    return affectedByTerrain;
}
```

Para generar una nueva squad, el usuario deberá arrastrar el icono de la squad azul que se encuentra en la interfaz deploy sobre el mapa, creando una nueva squad que contendrá las tropas que se muestran en esa fila. La implementación del sistema de Drag and Drop ya se ha comentado anteriormente y se aplica específicamente a este apartado, tanto para permitir la creación de squads como para el movimiento de las mismas en el mapa.



Arrastrar el icono azul de squad que se muestra en la imagen superior sobre el mapa y soltarlo en un punto desencadenará una llamada al método `deployTroops`, que limpiará el array que contiene a las tropas en modo deploy (dicha segunda fila de la imagen) y generará la nueva squad que contiene dichas tropas (en el caso de la imagen superior, 1000 artillerías akatsiya) con todos los eventos que el actor tendrá asociados:

```
private void deployTroops(TroopArray troops, Territory terry, float
posX, float posY, SquadType type)
{
    // Limpiamos las deploy del territorio (han sido extraidas todas)
    terry.setDeployedTroops(new TroopArray(new ArrayList<Troops>()));

    // Add nueva squad
    addSquad(troops, terry.getOwner(), posX, posY, type);
}

public static TroopSquad addSquad(TroopArray troops, Country owner,
float posX, float posY, SquadType type)
{
    final TroopSquad squad = new TroopSquad(troops, owner, posX, posY,
                                                type);

    squad.setTouchable(Touchable.enabled);

    // Click sobre la squad
    squad.addListener(new InputListener()
    {
        @Override
        public boolean touchDown(InputEvent event, float x, float y,
            int pointer, int button)
        {
            {
                // Toggle vision de la tabla
                if (squad.getTable().isVisible())
                    squad.hideTable();
                else
                    squad.showTable();
                return false;
            }
        }
    });
};
```

```

// Drag de la squad sobre el mapa (desplazamiento)
if (squad.getType() != SquadType.SQUAD_MILITIA)
{
    (...)
    public void dragStop(InputEvent event, float x, float y,
        int pointer, Payload payload, Target target)
    {
        // El player solo puede arrastrar sus tropas
        if (squad.getOwner() != m_player)
            return;

        (...)
        // Draw de squad sobre UI: Bottom table: Retract
        (Todas las comprobaciones simplificadas)
        squad.retractSquadToTerritory(m_selectedTerritory,
                                        true);

        (...)

        // El player mueve troop squad
        if (squad.getType() == SquadType.SQUAD_LAND)
            playSound(m_sendTroops_sound, 0.3f);
        else if (squad.getType() == SquadType.SQUAD_AIR)
            playSound(m_sendAirCraft_sound, 0.3f);

        // Desde la interfaz humana siempre en linea recta
        squad.sendSquad(new Vector2(destX, destY), true);
        return;
    }
}

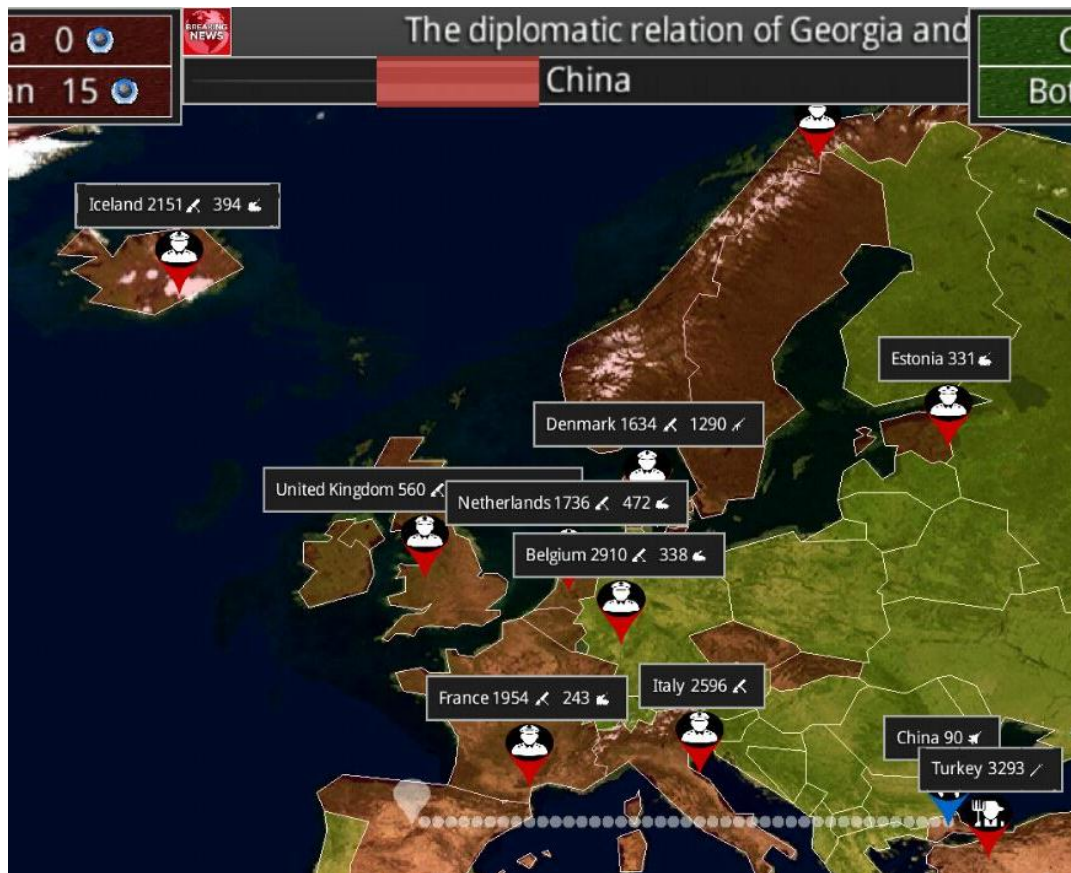
m_troopSquads.add(squad);
stageSquads.addActor(squad);
(...)
return squad;
}

```

El método `retractSquad` se emplea para devolver las tropas de la squad al territorio en el que se encuentre la squad, mientras que `SendSquad` envía la squad a un punto concreto del mapa, desencadenando toda la secuencia de cálculos de pathing que optimizan el camino a seguir y que detallaremos en el apartado de IA: Pathing.

Observamos que además las variables de la clase almacenan cosas como el destino final de la squad y si se mueve en línea recta (necesario para save/load) así como una stage que almacena un draw de puntos que sigue la squad en su movimiento, empleado para dibujar su trayectoria para el jugador.

La siguiente imagen muestra un ejemplo de squads en el mapa, una de ellas en movimiento.



Cuando los squads se mueven por el mapa, produce determinados efectos sobre los territorios:

- Efectos diplomáticos entre el dueño de las tropas que atraviesan un territorio y el dueño del territorio atravesado.
- Efectos sobre la lealtad del territorio que atraviesan.
- Combates y batallas con tropas enemigas que se encuentren en el territorio atravesado.

Los efectos diplomáticos los implementa el siguiente método, que se ejecuta en cada update para cada territorio:

```
private void updateDiplomacyByTroopsInTerritory()
{
    (...)
    float diplo =
        Formulas.getDiplomacyImpactOfCrossingAnotherCountryWithTroops(
            totalAttackPower + totalDefencePower, troopsOwner, getOwner());
    getOwner().modifyDiplomacyValueWithCountry(troopsOwner, diplo,
                                                true, true, false);
}
}
```

La fórmula se implementa en Formulas y vemos que el efecto diplomático depende de los puntos totales ofensivos y defensivos de las tropas que están atravesando el territorio, dicho efecto ya se ha descrito con detalle en el apartado dedicado a la diplomacia.

Los efectos sobre la lealtad del territorio se implementan en el método updateLoyalty:

```
private void updateLoyalty()
{
    // La lealtad no varia en las batallas
    if (m_territoryBattle.isInProgress())
        return;
    (...)
    float regeneratedLoyalty =
        Game.getTerritoryLoyaltyRegenerationSpeed(); // 0.3f de base.
    float addRegeneration = 0.0f;
    // Las tropas invasoras bajan la lealtad en relación asu valor de
    // ataque
    (...)
    Country countryPart = attackerGroup.getOwner();
    int attackPower = troop.getTotalAttackPower(100);

    float contribution =
        Formulas.getLoyaltyContributionByTroopsValue(attackPower);
    addRegeneration -= contribution;

    // Actualizamos contribucion del país para saber a quien
    // corresponde la conquista final
    (...)
    m_conquestContribution.put(countryPart, previousValue +
                                contribution);
    (...)
    // Las tropas defensoras suben la lealtad en sus valores de
    // defensa
    float defPower = troop.getTotalInfantryDefencePower(100, this) +
        troop.getTotalArtilleryDefencePower(100, this);
}
```

```

addRegeneration +=
    Formulas.getLoyaltyContributionByTroopsValue(defPower);

// Add bonos
int loyaltyStrenght = 0;

// Political Training: "Strengthen the loyalty on your territories
    by %v"
Technology politicalTraining1 =
    m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_POLIT
        ICAL_TRAINING_I);
if (politicalTraining1 != null)
    loyaltyStrenght += politicalTraining1.getValue();

Technology politicalTraining2 =
    m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_POLIT
        ICAL_TRAINING_II);
if (politicalTraining2 != null)
    loyaltyStrenght += politicalTraining2.getValue();
(...)

// Unbeatable Loyalty: "Strengthen the loyalty on your territories
    by %v"
Technology unbeatableLoyalty1 =
    m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_UNBEA
        TABLE_LOYALTY_I);
if (unbeatableLoyalty1 != null)
    loyaltyStrenght += unbeatableLoyalty1.getValue();
(...)

// Parliament: "Strengthen the loyalty of this territory by %v%"
Building parliamentBuilding =
    getBuildingByType(BuildType.BUILD_PARLIAMENT);
if (parliamentBuilding != null)
    loyaltyStrenght += parliamentBuilding.getValue();

// Si estamos perdiendo lealtad el fortalecimiento reducira la
    lealtad perdida.
if (addRegeneration < 0)
    addRegeneration -= Utils.ApplyPctToValue(addRegeneration,
        loyaltyStrenght);
else // Si estamos ganando lealtad el fortalecimiento aumentara la
    lealtad ganada.
    addRegeneration += Utils.ApplyPctToValue(addRegeneration,
        loyaltyStrenght);

float finalLoyaltyExchange = regeneratedLoyalty + addRegeneration;

// Si es menor que 0, se considera que el territorio esta bajo
    peligro de conquista
if (finalLoyaltyExchange < 0)
    updateUnderAttackTerritory();

// Calculamos lealtad nueva
m_loyalty += finalLoyaltyExchange;

(...)

```

```

// Conquistamos si se dan las condiciones
if (m_loyalty <= Game.m_conquestLoyalty)
{
    // Cual es el país que mas ha contribuido?
    HashMap.Entry<Country, Float> maxEntry = null;
    for (HashMap.Entry<Country, Float> entry :
        m_conquestContribution.entrySet())
        if (maxEntry == null ||
            entry.getValue().compareTo(maxEntry.getValue())
                > 0)
            maxEntry = entry;

    // El país que mas ha contribuido invade
    Country invader = maxEntry.getKey();
    conquestTerritory(invader);
}
}

```

Podemos realizar las siguientes apreciaciones del método anterior:

- Durante una batalla en el territorio su lealtad no se modifica.
- Las tropas atacantes (propiedad de un país hostil con el dueño del territorio en el que están) contribuyen negativamente a la lealtad según la formula *getLoyaltyContributionByTroopsValue*, y las defensivas contribuyen positivamente según la misma relación. Las contribuciones negativas de lealtad se acumulan para cada país, de forma que pueda cuantificarse en cada momento que país atacante contribuye a una mayor reducción de la lealtad.
- Los bonos de tecnologías y edificios se calculan por el mismo procedimiento de acumulación de incremento y suma, para prevenir el efecto de dispersión multiplicativa que comentábamos atrás.
- Si la lealtad de un territorio baja, se considera territorio bajo peligro de conquista y se llama al método *updateUnderAttackTerritory*.
- Si la lealtad llega a 0, el país será conquistado por el atacante que más haya contribuido a reducir su lealtad.
- Resulta evidente que, al ser el mismo territorio, la lealtad seguirá en 0 tras haber sido conquistado, pero las tropas que antes se consideraban ofensivas y que por tanto contribuían a reducir la lealtad, ahora son propiedad del mismo país que el territorio, ya que ha sido conquistado, por lo que ahora son tropas defensivas y contribuirán a aumentar la lealtad, esto hace que tras la conquista el territorio tenga una lealtad muy baja y sea muy vulnerable, lo cual encaja con la realidad de un territorio recién ocupado.

La fórmula que describe la contribución de lealtad comentada en los puntos anteriores es lineal con los puntos de las tropas implicadas:

```
public static float getLoyaltyContributionByTroopsValue(float
troopsValue)
{
    float output = troopsValue / 100000f;
    return output;
}
```

Y por último el método updateUnderAttackTerritory es el responsable de hacer aparecer milicias de vez en cuando.

```
public void updateUnderAttackTerritory()
{
    // Probabilidad base
    int baseMillitiaChance =
        Game.getTerritoryMillitiaSpawningChance(); // 10% de base.
    int addMillitiaBonus = 0;

    // Add technology bonus
    // Conscriptio: "Increases the chance of spawning militias by
    %v%."
    Technology conscription1 =
        m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_CONSC
        RIPTION_I);

    if (conscription1 != null)
        addMillitiaBonus += Utils.ApplyPctToValue(baseMillitiaChance,
            conscription1.getValue());

    Technology conscription2 =
        m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_CONSC
        RIPTION_II);

    if (conscription2 != null)
        addMillitiaBonus += Utils.ApplyPctToValue(baseMillitiaChance,
            conscription2.getValue());

    Technology conscription3 =
        m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_CONSC
        RIPTION_III);

    if (conscription3 != null)
        addMillitiaBonus += Utils.ApplyPctToValue(baseMillitiaChance,
            conscription3.getValue());

    // Spawnear milicia si es necesario
    if (Utils.roll_chance(baseMillitiaChance + addMillitiaBonus))
        spawnMillitia();
}
```

Podemos observar que las milicias tienen una probabilidad base de aparecer en cada update en el que el territorio pierda lealtad, esa probabilidad puede además aumentarse con distintos bonus que se calculan como siempre, mediante acumulación aditiva del efecto. Si finalmente se decide hacer aparecer milicia:

```
public void spawnMillitia()
{
    float militiaNumber =
        Formulas.getMilitiaNumberByProductionAndRemainingLoyalty(
            getOwner().getTotalMoneyProduction(), 100 -
            getLoyalty(), getOwner());

    // Nuevas tropas aparecen en el territorio
    ArrayList<Troops> militiaTroops = new ArrayList<Troops>();
    militiaTroops.add(new Troops((int) militiaNumber,
        Game.getUnitByType(UnitType.UNIT_MILITIA), getOwner(),
            this));

    // Add milicias
    Vector2 pos = getRandomPointFromCenterWithRandomRadius((int)
        Formulas.convertScaleToResolution(100));
    Game.addSquad(new TroopArray(militiaTroops), getOwner(), pos.x,
        pos.y, SquadType.SQUAD_MILITIA);
}
```

El método anterior hará aparecer una nueva TroopSquad en un punto aleatorio del territorio, el cálculo de dichos puntos aleatorios que también se emplea en la aparición de squads de barcos y aviones al ser entrenados será detallado en el apartado correspondiente de IA, que es donde más se usan estos métodos.

La cantidad de milicias que aparecen está definida por una expresión directamente proporcional a la producción total del país y el déficit de lealtad del territorio, afectada por algunas bonificaciones tecnológicas y levemente aleatorizada:

```
public static float getMilitiaNumberByProductionAndRemainingLoyalty
(float prod, float remainingLoyalty, Country deffCountry)
{
    // Mas milicia cuanto mas produccion tenga el país (100 tropas
    // milicia por unidad de produccion)
    // Mas milicia cuanto menos lealtad tenga el territorio (100
    // tropas milicia por unidad faltante de lealtad)
    float militiaNumber = (prod + remainingLoyalty) * 100;

    // Add valor aleatorio de -10% o +10%
    militiaNumber *= (float)Utils.getRandomValue(90, 110) / 100f;
```

```

// Efecto de las tecnologías
// International Support: "Your territories spawns %v% more
//   millitians when being conquered."
float bonusMillitia = 0f;
Technology intSupport =
    deffCountry.getCountryDevelopedTechnologyByType(TechType.TECH_I
                                                    NTERNATIONAL_SUPPORT);

if (intSupport != null)
    bonusMillitia += Utils.ApplyPctToValue(militiaNumber,
                                           intSupport.getValue());

return militiaNumber + bonusMillitia;
}

```

Sabemos que las milicias son un tipo de tropa con una capacidad bastante reducida, por lo que aparecen en números altos y están diseñadas para aumentar la capacidad defensiva de ciertos países pequeños frente a grandes invasores, pudiendo determinar la rentabilidad de una invasión.

De forma adicional, cuando el territorio llega a la lealtad de conquista (0 por defecto) se produce la conquista que además implica un aumento considerable del damage del territorio y un empobrecimiento considerable de las relaciones diplomáticas internacionales del conquistador como se puede ver en la implementación del método:

```
public void conquestTerritory(Country conqueror)
{
    // El conquistador es el owner actual.
    if (conqueror == this.getOwner())
        return;

    // Musica
    // Si lo gana
    if (conqueror.equals(Game.m_player))
        Game.playSound(Game.m_terryConquered_sound, 0.8f);
    else if (getOwner().equals(Game.m_player)) // Si lo pierde
        Game.playSound(Game.m_terryLost_sound, 0.8f);

    // Efecto sobre la diplomacia (conquistar baja 50 puntos)
    conqueror.modifyDiplomacyValueWithCountry(this.getOwner(), -50,
                                                true, true, false);

    // Le quitamos el territorio al owner actual
    getOwner().removeTerritory(this);

    // Se lo add al owner nuevo
    conqueror.addTerritory(this);

    // Establecemos lealtad a 0
    m_loyalty = 0;

    // Aumentamos damage del territorio por la conquista
    modifyDamage(Utils.getRandomValue(20, 50));

    // Forzamos update de la UI
    Game.m_forceUpdate = true;
}
```

Los efectos sobre las batallas en el territorio son implementados por el método `updateBattle` que forma parte del `update` de `territory` y con la ayuda de la clase `Battle`:

```
public class Battle
{
    private int m_battleDays;
    private boolean m_isInProgress;
    private Territory m_battleTerritory;
    private Image m_icon;
    private Sound m_sound;
    private boolean m_isPlayerImplicated;

    (...)

    public void executeBattle(ArrayList<TroopArray> attackers,
                             ArrayList<TroopArray> defenders, float combatPct)

    public void increaseBattleDays()

    public void startBattle()

    public void endBattle()

    public int getBattleDays()
    (...)
    public boolean isInProgress()
    (...)
    public boolean isPlayerImplicated()
    (...)
}
```

Podemos definir una batalla como una sucesión de días en los que se producen combates entre distintas tropas, que se inicia cuando ciertas tropas entran en contacto y se termina cuando todas las tropas de un bando han muerto o han huido. La clase `Territory` posee un objeto `Battle` que se gestiona en el método `updateBattle` o por ejemplo al comprobar si hay una batalla activa en `updateLoyalty` como hemos visto.

```

private void updateBattle()
{
    ArrayList<TroopArray> attackerSquads = getAttackerTroopGroups();
    ArrayList<TroopArray> defenderSquads = getDefenderTroopGroups();
    ArrayList<TroopSquad> participatingSquads =
        getParticipantSquadsTroopGroups();

    // Si hay tropas atacantes que bombardeen
    if (attackerSquads.size() != 0)
    {
        // DAMAGE TAKEN por el territorio si el atacante tiene aviones
        // bombardeando
        int totalBomberAttackPower = 0;
        boolean playerIsBombing = false;
        // Barremos tropas atacantes para ver si hay bombarderos y
        // obtener su AP
        for (TroopArray bomberGroup : attackerSquads)
        (...)

        // DAMAGE TAKEN POR TERRITORIO BOMBARDEADO
        float bomberDamage =
            Formulas.getDamageDoneToTerritoryByBombers(
                totalBomberAttackPower);

        if (bomberDamage > 0)
        {
            spawnExplosionsAnimations(bomberDamage, playerIsBombing);
            modifyDamage(bomberDamage);
        }
    }

    // Si no hay tropas atacantes o defensoras, no continuamos y
    // finalizar batalla si estaba en curso.
    if (defenderSquads.size() == 0 || attackerSquads.size() == 0)
    {
        // Si habia batalla en curso, se interrumpe
        if (m_territoryBattle.isInProgress())
            m_territoryBattle.endBattle();
    }
    else if (defenderSquads.size() != 0 && attackerSquads.size() != 0)
    // Hay tropas atacantes Y defensoras
    {
        // Iniciamos batalla si no estaba en marcha
        if (!m_territoryBattle.isInProgress())
            m_territoryBattle.startBattle();
        else // Incrementamos dia en caso contrario
            m_territoryBattle.increaseBattleDays();

        // Mostrar tablas
        for (TroopSquad squad : participatingSquads)
            squad.showTable();
    }
}

```

```

// La cantidad de tropas que luchan con el paso de los dias es
incremental
switch (m_territoryBattle.getBattleDays())
{
    case 0:
    case 1: // Los 2 primeros dias no luchan
    case 2:
        break;
    case 3: // Los siguientes 4 dias luchan al 15%
    case 4:
    case 5:
    case 6:
        m_territoryBattle.executeBattle(attackerSquads,
                                         defenderSquads, 15);
        break;
    case 7: // Los siguientes 3 al 35% de las restantes.
    case 8:
    case 9:
        m_territoryBattle.executeBattle(attackerSquads,
                                         defenderSquads, 35);
        break;
    case 10: // El ultimo dia luchan todas las que quedasen
    default:
        m_territoryBattle.executeBattle(attackerSquads,
                                         defenderSquads, 100);
        m_territoryBattle.endBattle();
        break;
}
(...)
}
}

```

Podemos sacar las siguientes conclusiones:

- Definimos tropas atacantes como aquellas cuyo dueño tenga una diplomacia hostil con el territorio que atraviesan.
- Definimos tropas defensoras como aquellas cuyo dueño tenga una diplomacia aliada con el territorio que atraviesan las tropas atacantes o que sean propiedad del dueño del territorio y se encuentren en él.
- Las tropas acuarteladas del territorio atacado también contribuyen como tropas defensivas.
- Si hay bombarderos entre las tropas atacantes, estos contribuirán provocando bombardeos y daños adicionales al territorio además de contribuir al poder de ataque como tropas atacantes.
- La batalla se inicia cuando se encuentran tropas atacantes y defensoras y se termina cuando deja de haber uno de los dos grupos.
- Los combates de la batalla se ejecutan en cada ciclo de update, enfrentando cada vez a un porcentaje mayor de tropas implicadas hasta que uno de los dos bandos pierde todas las tropas o se retira.

El daño que causan los bombardeos en el territorio depende del poder de ataque total de los bombarderos implicados y se calcula en el siguiente método:

```
public static float getDamageDoneToTerritoryByBombers(int
totalBomberAP)
{
    // 1% de dmg recibido por cada tick de batalla en la que haya 100k
    // de AP de aviones (un squad de 100)
    float output = totalBomberAP / 100000f;
    return output;
}
```

Los bombardeos producen la ejecución de gráficos y sonidos de explosiones en el territorio bombardeado, esto es gestionado por la siguiente función:

```
private void spawnExplosionsAnimations(float bomberDamage, boolean
playerIsBombing)
{
    // Numero de explosiones aleatorias
    int blastsPerDamage = (int)(10 * bomberDamage);
    int maxBlast = Utils.getRandomValue(0, 5 > blastsPerDamage ? 5 :
    blastsPerDamage);
    for (int i = 0; i < maxBlast; i++)
    {
        // Duracion, size y explosion a usar aleatoria
        int size =
        Utils.getRandomValue((int)Formulas.convertScaleToResolution(20)
        , (int)Formulas.convertScaleToResolution(40));
        int duration = Utils.getRandomValue(15, 25);
        int sprite = Utils.getRandomValue(1, 3);
        Animation anim = null;

        switch (sprite)
        {
            case 1:
                anim = new Animation(1f / duration,
                Game.m_explosion1);
                break;
            case 2:
                anim = new Animation(1f / duration,
                Game.m_explosion2);
                break;
            case 3:
                anim = new Animation(1f / duration,
                Game.m_explosion3);
                break;
        }
    }
}
```

```

if (anim != null)
{
    // Posicion de la explosion tambien aleatoria
    Vector2 pos = getRandomPoint();
    AnimatedActor actorAnim = new AnimatedActor(new
                                                AnimationDrawable(anim));
    actorAnim.setBounds(pos.x, pos.y, size, size);
    Game.stageAnimations.addActor(actorAnim);

    // Sonido solo si el jugador esta implicado
    if (getOwner().equals(Game.m_player) || playerIsBombing)
    {
        // Sonido aleatorio
        Sound sound = null;
        switch (Utils.getRandomValue(1, 4))
        {
            case 1:
                sound = Game.m_bomb_sound;
                break;
            case 2:
                sound = Game.m_bomb2_sound;
                break;
            case 3:
                sound = Game.m_bomb3_sound;
                break;
            case 4:
                sound = Game.m_bomb4_sound;
                break;
        }
        if (sound != null)
            Game.playSound(sound, 0.05f);
    }
}
}
}

```

Observamos que se ejecuta todo de forma pseudoaleatoria para dar una impresión caótica

- El gráfico elegido para la explosión es elegido aleatoriamente entre tres opciones.
- El numero de explosiones, su tamaño y duración son parámetros también elegidos aleatoriamente, no obstante, el número de explosiones tiene más probabilidad de ser un número mayor cuanto más daño producen los bombardeos.
- El sonido de las explosiones es también elegido aleatoriamente entre 4 opciones, pero sólo se escuchará cuando el jugador sea quien bombardea o el bombardeado sea uno de sus territorios.

La ejecución de batalla la implementa el método `executeBattle`, recibiendo una lista de tropas atacantes, de tropas defensoras y el porcentaje de dichas tropas que lucharán:

```
public void executeBattle(ArrayList<TroopArray> attackers,
ArrayList<TroopArray> defenders, float combatPct)
{
    (...)
    // Calculamos todos los valores ofensivos
    for (TroopArray attackerTroops : attackers)
    {
        (...)
        for (Troops troop : attackerTroops.getTroops())
        {
            // AP que aportan las tropas
            totalAttackPower += troop.getTotalAttackPower(combatPct);

            // Add ataque de cada tipo segun tipo de tropa
            switch (troop.getUnit().getType())
            {
                case UNIT_MILITIA:
                case UNIT_INFANTRY_MORTAR:
                case UNIT_INFANTRY_ASSAULT:
                    totalInfantryAttackPower += totalAttackPower;
                    break;
                case UNIT_ARTILLERY_AKATSIYA:
                case UNIT_ARTILLERY_ABRAMS:
                case UNIT_AIRCRAFT_BOMBER:
                case UNIT_SHIP_WARSHIP:
                    totalArtilleryAttackPower += totalAttackPower;
                    break;
            }
        }
        (...)
    }
}

// Calculamos todos los valores defensivos
for (TroopArray defenderTroops : defenders)
{
    (...)
    for (Troops troop : defenderTroops.getTroops())
    {
        // Defensas totales que aportan las tropas
        totalDefenseInfantryPower +=
            troop.getTotalInfantryDefencePower(combatPct,
                                                m_battleTerritory);
        totalDefenseArtilleryPower +=
            troop.getTotalArtilleryDefencePower(combatPct,
                                                m_battleTerritory);
        (...)
    }
}
```

```

// Calculamos Infantry/Artillery %
float infantryPct = (float)totalInfantryAttackPower /
                    (float)totalAttackPower;
float artilleryPct = (float)totalArtilleryAttackPower /
                    (float)totalAttackPower;

// Defensa efectiva
float totalEffectiveDefence = infantryPct *
    totalDefenseInfantryPower + artilleryPct *
    totalDefenseArtilleryPower;

// Establecemos perdidas en el ganador y en el perdedor
float winnerPts = 0f;
float loserPts = 0f;
if (totalAttackPower > totalEffectiveDefence) // Ganan los
    atacantes
{
    winnerPts = totalAttackPower;
    loserPts = totalEffectiveDefence;
    (...)

    // Porcentaje de tropas perdidas por el ganador
    float losePct = (float)Math.pow((loserPts / winnerPts), 1.5f);

    // El defensor pierde todas sus tropas
    for (TroopArray defenderTroops : defenders)
        for (Troops troop : defenderTroops.getTroops())
        {
            int casualties = Math.round(troop.getNumber() *
                                        (combatPct / 100f));
            troop.setNumber(troop.getNumber() - casualties);

            // Calcumos puntos de las bajas para el calculo del
            damage
            (...)
        }

    // El atacante pierde un %
    for (TroopArray attackerTroops : attackers)
        for (Troops troop : attackerTroops.getTroops())
        {
            int casualties = Math.round(troop.getNumber() *
                                        (combatPct / 100f) * losePct);
            troop.setNumber(troop.getNumber() - casualties);

            // Calcumos puntos de las bajas para el calculo del
            damage
            (...)
        }
}

else // Ganan los defensores.
{
    (...)
}

```

```

// DAMAGE TAKEN por el territorio donde se ha producido el combate
m_battleTerritory.modifyDamage(Formulas.
getDamageTakenByTerritoryAfterBattle(totalCasualtiesPoints));

// IMPACTO DIPLOMATICO
for (TroopArray attackerTroops : attackers)
    for (TroopArray defenderTroops : defenders)
        if (attackerTroops.hasTroops() &&
            defenderTroops.hasTroops())

            attackerTroops.getOwner().modifyDiplomacyValueWithCountry(
                defenderTroops.getOwner(),
                Formulas.getDiplomacyImpactAfterBattle(
                    totalCasualtiesPoints), true, true,
                    false);
}

```

Comentando un poco la lógica que sigue la ejecución de la batalla podemos decir lo siguiente:

1. Se obtiene el total de puntos de ataque de las tropas atacantes y se separa el ataque de tipo infantería y el ataque de tipo artillería según el tipo de tropa.
2. Se obtiene la defensa de cada tipo que poseen las tropas defensivas.
3. Se calcula el porcentaje que supone cada tipo de puntos ataque con respecto al poder de ataque total.
4. Se calcula una defensa efectiva multiplicando el porcentaje de tipo de ataque (calculado en el punto 3) por la defensa total frente a ese tipo de ataque (obtenido en el punto 2) y sumando el resultado de estos productos.
5. Si el poder de ataque es superior a la defensa efectiva (obtenida en el punto 4), gana el atacante, en caso contrario, gana el defensor.
6. Calculo de bajas:
 - El bando perdedor perderá todas las tropas que hayan combatido.
 - El bando ganador perderá un porcentaje de las tropas que hayan combatido, que se calculará como el cociente entre los puntos del perdedor y los puntos del ganador elevado a 1'5. Este exponente deslinealiza la relación, haciendo que una ventaja numérica grande del vencedor reduzca sus bajas de forma exponencial.
 - El cálculo de las bajas anteriores está siempre sujeto al porcentaje de tropas totales que luchan que se pasa como parámetro al método, de forma que, si por ejemplo luchan un 10% de las tropas, el cálculo se realizará teniéndolas en cuenta todas, pero en las bajas de ambos bandos se escalarán a un 10% del total, de forma que el bando perdedor perderá un 10% de sus tropas totales y el bando ganador un 10% de las bajas que fuese a tener originalmente.

7. Impacto adicional:

- El combate tiene un impacto en el damage del territorio donde se realiza que depende de las bajas que haya habido, entendiéndose que se habrá tratado de una batalla más encarnizada.
- El combate tiene un impacto en la diplomacia de todos los países implicados en la batalla, el impacto será mayor cuantas más bajas haya sufrido el país en cuestión.

Las fórmulas que relacionan las bajas sufridas con los impactos descritos en el punto 7 se implementan así:

// +1 punto de damage por cada 10k de puntos de AP+DI+DA de cada baja que haya habido en la batalla.

```
public static float getDamageTakenByTerritoryAfterBattle(float casualtiesPoints)
{
    // Calculamos
    float output = casualtiesPoints / 10000f;

    // Devolvemos
    return output;
}
```

// -1 punto de diplomacia por cada 1M de puntos de AP+DI+DA perdidos en total

```
public static float getDiplomacyImpactAfterBattle(float casualtiesPoints)
{
    // Calculamos
    float output = -casualtiesPoints / 1000000f;

    // Devolvemos
    return output;
}
```

La ejecución de la batalla se dispersa en varios días con intensidad creciente (cada vez lucha un porcentaje mayor de las tropas restantes) para permitir la incorporación de tropas durante la batalla, ya sean aliados del atacado o aliados del atacante que contribuyan al ataque. El sistema también permite que uno de los participantes se retire antes de que todas sus tropas hayan luchado y muerto.

En todos los métodos anteriores hemos visto la aplicación de la variable damages, este valor reduce la producción del territorio afectado:

```
public int getMoneyProduction()
{
    int baseProduction = m_moneyProduction;
    int addProduction = 0;

    (...)
    // Restar damages del territorio
    addProduction -= Utils.ApplyPctToValue(baseProduction, (int)
                                                m_damages);

    (...)
    return baseProduction + addProduction;
}
```

Y además los territorios poseen una regeneración pasiva de damage que puede ser bonificada con edificios y tecnologías, se implementa en un método que forma parte del update de territorio:

```
public void regenerateDamage()
{
    // La regeneracion de dmg no se produce en batallas
    if (m_territoryBattle.isInProgress())
        return;

    (...)

    float regeneratedDamage =
        Game.getTerritoryDamageRegenerationSpeed(); // 0.2f de base.
    float addRegeneration = 0.0f;

    // Add bonos de tecnologias
    if (m_owner != null)
    {
        // Robotic machinery: "Increase the damage regeneration on
        your territories by %v"
        Technology roboticMachinery1 =

        m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_ROBOT
                                                    IC_MACHINERY_I);

        if (roboticMachinery1 != null)
            addRegeneration +=
                Utils.ApplyPctToValue(regeneratedDamage,
                    roboticMachinery1.getValue());

        (...)
    }
}
```

```

// Tireless Workers: "Increases the damage regeneration of
your territories damaged over 50% by %v%."
if (m_damages > 50)
{
    Technology tirWork =
        m_owner.getCountryDevelopedTechnologyByType(
            TechType.TECH_TIRELESS_WORKERS);
    if (tirWork != null)
        addRegeneration += Utils.ApplyPctToValue(
            regeneratedDamage, tirWork.getValue());
}
(...)
}

// Reducimos damage
modifyDamage(-(regeneratedDamage + addRegeneration));
}

```

La regeneración de damage del territorio es considerablemente lenta, el objetivo del diseño es que un país conquistador tarde mucho en amortizar las pérdidas de su conquista y de ese modo mitigar la exponencial e inevitable acumulación de poder debido a conquistas sucesivas.

3.11 EVENTOS ALEATORIOS

Como ya vimos en la inicialización existen eventos aleatorios a lo largo del juego que producen una serie de efectos, actualmente sólo hay tres implementados, pero se espera ir añadiendo más, ya que la implementación es bastante dinámica.

Tipo	Probabilidad
TYPE_DIPLOMATIC_CONFUSION	2
TYPE_DIPLOMATIC_IMPROVEMENT	2
TYPE_NATURAL_DISASTER	1

Dichas probabilidades se ejecutan en cada día/turno en el método:

```

private void updateRandomEvents()
{
    for (RandomEvent event : m_randomEvents)
        if (Utils.roll_chance(event.getChance()))
            event.execute();
}

```

El método `roll_chance` realiza una “tirada de dados” y devuelve `true` cuando se da esa probabilidad sobre 100, por ejemplo, una llamada `roll_chance(50)` devolvería `true` la mitad de las veces que se llama.

En caso de darse la probabilidad se llamará al método `execute` del evento que ejecutará distintos métodos en función de qué evento se trate y cuyos códigos se muestran a continuación:

```
public void execute()
{
    switch (m_type)
    {
        case TYPE_DIPLOMATIC_CONFUSION:
            executeDiplomaticConfusionEvent();
            break;
        case TYPE_DIPLOMATIC_IMPROVEMENT:
            executeDiplomaticImprovementEvent();
            break;
        case TYPE_NATURAL_DISASTER:
            executeNaturalDisasterEvent();
            break;
    }
}

private void executeNaturalDisasterEvent()
{
    // Escogemos territorio aleatorio para el evento
    Country countryDest =
        (Country)Utils.getRandomElementFromHashMap(Game.m_countryList);
    Territory terry =
        Utils.getRandomElementFromArrayList(
            countryDest.getTerritories());

    if (terry == null)
        return;

    // Valor de damage por el desastre
    int value = Utils.getRandomValue(1, 10);
    terry.modifyDamage(-value);

    // Mensaje aleatorio
    switch (Utils.getRandomValue(1, 5))
    {
        case 1:
            Game.updateBreakingNews("A tornado in " + terry.getName()
                                    + " has damaged its economy.");
            break;
        case 2:
            Game.updateBreakingNews("A street riots in " +
                                    terry.getName() + " causes the territory economy to go
                                    down.");
            break;
        case 3:
            Game.updateBreakingNews("A financial crisis at " +
                                    terry.getName() + " market stock damages its
                                    economy.");
            break;
    }
}
```

```

        case 4:
            Game.updateBreakingNews("An earthquake destroyed some
                cities of " + terry.getName() + ".");
            break;
        case 5:
            Game.updateBreakingNews("The inundations continues at " +
                terry.getName() + " and its economy begins to
                    suffer.");
            break;
    }
}

private void executeDiplomaticImprovementEvent()
{
    Country countryO = null;
    Country countryDest = null;

    // Elegimos dos paises aleatorios
    int maxTrys = 100;
    int trys = 0;
    while (true)
    {
        countryO = (Country)Utils.getRandomElementFromHashMap(
            Game.m_countryList);
        countryDest = (Country)Utils.getRandomElementFromHashMap(
            Game.m_countryList);

        // Antifreeze
        trys++;

        // Antifreeze, no hay paises para hacer esto, fin.
        if (trys >= maxTrys)
            return;
        if (countryO == null || countryDest == null)
            continue;
        if (countryO.equals(countryDest))
            continue;
        if (!countryO.isAlive())
            continue;
        if (!countryDest.isAlive())
            continue;
        // Llegados a este punto, all es valido, abandonar bucle
        break;
    }

    // Valor de damage a la diplo
    int value = Utils.getRandomValue(5, 15);
    countryO.modifyDiplomacyValueWithCountry(countryDest, value, true,
true, false);
}

```

```

// Mensaje aleatorio
switch (Utils.getRandomValue(1, 5))
{
    case 1:
        Game.updateBreakingNews("The reunion of the presidents of
            " + countryO.getName() + " and " +
            countryDest.getName() + " has improved their
                relations.");
        break;
    case 2:
        Game.updateBreakingNews("The support of " +
            countryO.getName() + " to " + countryDest.getName() + "
                at the last summit has enhanced their
                    relations.");
        break;
    case 3:
        Game.updateBreakingNews("The good relations within
            populations of " + countryO.getName() + " and "
            + countryDest.getName() + " are improving the
                relations of their countries.");
        break;
    (...)
}
}

```

El evento de confusión diplomática se implementa del mismo modo que el anterior, pero disminuyendo la diplomacia en lugar de mejorarla y con mensajes de noticias adaptados al evento, no se ha copiado su código pues resultaría redundante.

3.12 CICLO DEL JUEGO

Una vez vistos todos los componentes del juego vamos a analizar el ciclo del juego que permite la ejecución de todo lo anterior. Como ya hemos introducido en la inicialización, el update del juego completo se ejecuta en un hilo independiente, de modo que no afectará al rendimiento del dibujado ni de las interacciones de interfaz, en el apartado de threading detallaremos más la implementación descrita, así como sus implicaciones. Los siguientes fragmentos de código muestran cómo se suceden las diferentes actualizaciones de eventos.

El método render acumula el tiempo que va transcurriendo y forzará una ejecución del thread de update cuando el tiempo transcurrido desde la última vez que se ejecutó supere el tiempo que corresponde a un día (que depende de la velocidad), en caso contrario incrementará el contador en el tiempo diferencial desde el ultimo render.

```
public void render()
{
    (...)

    // Update
    // No actualizar si esta parado
    if (m_secondsPerDay != -1)
    {
        if (m_tickCounter >= m_secondsPerDay)
        {
            // Restablecemos y actualizamos
            m_tickCounter = 0;
            m_updateThread.updateGame();
        }
        else
            m_tickCounter += Gdx.graphics.getDeltaTime();
    }
    break;
}
}
```

El thread updateThread se encargará de llamar al método de update general, actualizando de forma secuencial todos los elementos del juego:

```
// Heartbeat
public void updateGame()
{
    // No actualizamos hasta que no haya player
    if (m_player == null)
        return;

    // Guardado cada 10 dias
    if (m_saveTimer > m_saveFrequency)
    {
        m_saveTimer = 0;
        saveGame(false);
    }
    else if (m_autoSave)
        m_saveTimer++;

    // Updateamos fecha
    increaseDay();
    // Updateamos eventos aleatorios
    updateRandomEvents();
}
```

```

// Updateamos todos los paises
for (Country country : m_countryList.values())
{
    // Si el país aun tiene territorios, update.
    if (country.isAlive())
        country.update();
    else if (country.equals(m_player)) // Si no tiene territorios
        y es el player, gameover al player.
        gameOver();
}
// Check victoria
if (m_player.getTerritories().size() == m_territoryList.size())
    winGame();

// Update squads
updateSquads(true);
// Update AI
m_aiThread.update();
// Update UI
updateUI();
}

```

El método anterior incrementa el día de la fecha, guarda la partida cada cierto tiempo, actualiza los eventos aleatorios, las squads (borrando las que ya no deben existir), fuerza un update a todos los países y a sus inteligencias artificiales y por último la interfaz para reflejar todos los cambios que hayan ocurrido en el turno.

La actualización de squads se encarga de revisar aquellas squads que ya no deben existir como actores en el juego, aquellas que ya no contienen tropas, deben ser eliminadas, del mismo modo que las milicias que ya no sean necesarias en un territorio. También se encarga de forzar la actualización de tareas de tropas para las squads que pertenecen a IA, detallaremos con mayor cuidado este tema en el apartado de gestión de tropas y tareas de la inteligencia artificial.

```

public static void updateSquads(boolean updateTasks)
{
    // Squads para limpiar
    ArrayList<TroopSquad> removeSquads = new ArrayList<>();
    for (int i = 0; i < m_troopSquads.size(); i++)
    {
        TroopSquad squad = m_troopSquads.get(i);

        // Update task
        if (updateTasks)
            if (!squad.getOwner().equals(m_player)) // solo a las de
                                                        las IAs
                squad.updateTask();

        // Comprobamos squads vacios
        if (!squad.hasTroops())
            removeSquads.add(squad);
    }
}

```

```

// Comprobamos milicias de territorios con lealtad al 100%
if (squad.getType() == SquadType.SQUAD_MILITIA)
{
    Territory squadTerr = squad.getTerritory();
    if (squadTerr != null)
        if (squadTerr.getLoyalty() >= 100)
            removeSquads.add(squad);
}

// Barremos array y eliminamos las vacias
for (TroopSquad remSquad : removeSquads)
{
    // Actualizamos capacidades antes de eliminar la squad
    remSquad.getOwner().updateTotalAttackPower();
    remSquad.getOwner().updateTotalDeffPower();

    // Eliminamos squad
    remSquad.removeSquad();
}
}

```

En el caso de la inteligencia artificial, observamos que se fuerza un update a su thread asociado, que llama a un método que actualiza las IAs de todos los países:

```

(...)
for (Country country : m_countryList)
    country.updateAI();
(...)

public void updateAI()
{
    if (equals(Game.m_player))
        return;
    if (!Game.ai_active)
        return;

    // Update IA
    m_ai.updateAI();
}

```

```

public void updateAI ()
{
    switch (m_type)
    {
        case AI_IMPERIALIST:
            updateImperialistAI ();
            break;
        case AI_AGGRESSIVE:
            updateAgressiveAI ();
            break;
        case AI_DEFENSIVE:
            updateDefensiveAI ();
            break;
        case AI_PEACEFUL:
            updatePeacefulAI ();
            break;
        (...)
    }

    // Limpiamos squads
    clearUntaskedSquads ();
}

```

El método de update de IA se ramifica en función del tipo de IA, profundizaremos más adelante en este aspecto en su apartado correspondiente, ahora analizaremos el update que se realiza sobre cada país desde updateGame():

```

public void update ()
{
    // Update cola de tecnologías
    updateTechQueues ();

    // Barremos territorios
    for (int i = 0; i < m_territoryList.size(); i++)
    {
        Territory terr = m_territoryList.get(i);

        if (terr == null)
            continue;

        m_money += terr.getMoneyProduction();

        // Actualizar territorio
        terr.update ();
    }

    // Update diplomatic position
    updateDiplomaticPosition ();
}

```

En el método anterior observamos que cada país actualiza su dinero en función de la producción de sus territorios, actualiza también las colas tecnológicas y actualiza la posición diplomática del país, que consiste en establecer un valor que cuantifique cuanto de bien posicionado está un país diplomáticamente hablando, es una variable que será empleada por las IAs y que es mucho más eficiente calcular en este momento y almacenarla para cuando se necesite que en cada cálculo de IA.

```
public int getMoneyProduction()
{
    int baseProduction = m_moneyProduction;
    int addProduction = 0;
    (...)

    // Add produccion del banco
    Building productionBuilding =
        getBuildingByType(BuildType.BUILD_BANK);
    if (productionBuilding != null)
        addProduction += Utils.ApplyPctToValue(baseProduction,
            productionBuilding.getValue());

    // Add bonos de tecnologias
    if (m_owner != null)
    {
        // Digital Exchange : "Increases the production of all of your
            territories by %v"
        Technology digitalExchange1 =

        m_owner.getCountryDevelopedTechnologyByType(TechType.TECH_DIGIT
            AL_EXCHANGE_I);

        if (digitalExchange1 != null)
            addProduction += Utils.ApplyPctToValue(baseProduction,
                digitalExchange1.getValue());

        (...)

        // Local Taxes: Increases the production of each territory you
            own by %v% for each building it has at
            its maximum level.
        Technology localTaxes =
            m_owner.getCountryDevelopedTechnologyByType(
                TechType.TECH_LOCAL_TAXES);

        if (localTaxes != null)
            for (int i = 0; i < m_buildings.size(); i++)
                if (m_buildings.get(i).isBuildingAtMaxLevel())
                    addProduction += Utils.ApplyPctToValue(
                        baseProduction, localTaxes.getValue());

    }

    // Restar damages del territorio
    addProduction -= Utils.ApplyPctToValue(baseProduction, (int)
        m_damages);

    (...)

    return baseProduction + addProduction;
}
```

Posteriormente a la actualización económica llamará a update() de cada territorio:

```
public void update()  
{  
    updateBuildingQueues();  
    updateTrainingQueues();  
    updateDiplomacyByTroopsInTerritory();  
    updateBattle();  
    regenerateDamage();  
    updateLoyalty();  
    updateTerritoryTroopsTasks();  
    updateTroopArrays();  
    updateValue();  
}
```

Observamos que es ya en este método donde actualizamos todas las variables importantes de cada territorio y no anidamos más las actualizaciones. El siguiente esquema muestra todas las operaciones que realiza un ciclo de actualización:

- UpdateGame(), método de actualización principal:
 - Autoguardado del juego.
 - Incrementar día.
 - Comprobar si ocurren eventos aleatorios y ejecutarlos en caso de ocurrir.
 - Desencadenar victoria si el jugador posee todos los territorios.
 - Actualizar estado de squads.
 - Actualizar interfaz de usuario
 - Desencadenar derrota si el jugador no está vivo (no posee territorios ni tropas)
 - Actualizar las inteligencias artificiales de países vivos no jugador.
 - Actualizar todos los países que estén vivos.
 - Actualizar colas tecnológicas.
 - Actualizar valor de posición diplomática.
 - Calcular el nuevo dinero del país en función de la producción de sus territorios.
 - Actualizar todos sus territorios
 - Actualizar colas de construcción de edificios.
 - Actualizar colas de entrenamiento de tropas.
 - Actualizar cambios en diplomacia provocados por tropas que se encuentran sobre este territorio.
 - Actualizar o desencadenar batallas en este territorio provocadas por tropas enemigas pasando sobre él.
 - Aplicar regeneración pasiva de daños del territorio.
 - Aplicar modificaciones de lealtad del territorio, tanto la regeneración pasiva como la regeneración potenciada por tropas defensivas como la pérdida de lealtad por tropas enemigas sobre el territorio.
 - Actualizar tareas de las tropas del territorio, empleado por las IAs que asignan distintas tareas a las tropas, este método es responsable de eliminar las tareas ya terminadas.
 - Actualizar listas de tropas del territorio, por ejemplo, si las tropas han sido eliminadas deberá eliminar el objeto lista.
 - Actualizar valor del territorio, similar al valor diplomático, es un valor que será leído por las IAs y que resulta más óptimo calcularlo en cada ciclo en lugar de bajo demanda.

3.13 GUARDADO Y CARGA DE PARTIDAS

El juego tiene implementado un sistema de guarda y carga que funciona del siguiente modo:

- El juego se guarda automáticamente cuando el usuario abandona el juego.
- El juego se guarda automáticamente si se producen ejemplos no disruptivos que interrumpan la ejecución del juego.
- El juego se guarda automáticamente cada cierto periodo de tiempo, dicho tiempo está definido por el valor de la variable `m_saveFrequency` y actualmente es de 5 turnos o días.
- La decisión de cargar una partida previa se realiza al ejecutar el juego, si no se elige cargar una partida previa, esta será reemplazada, ya que sólo puede existir una partida guardada simultáneamente.
- El guardado se realiza escribiendo en un fichero externo de extensión `.dat` que está codificado en base64.
- El fichero externo de guardado está contenido en las carpetas internas de recursos de la aplicación.
- El guardado se realiza en un hilo independiente, de modo que el evento I/O no suponga un cuello de botella durante la ejecución del juego.
- El guardado se implementa mediante la escritura del valor de las variables relevantes del juego en formato String concatenadas con separadores en el fichero `.dat`
- La carga se implementa mediante la lectura y decodificación del fichero externo y el parse de la información en base a los separadores, por lo que es importante que el fichero no sea modificado externamente o se podrían producir errores de lectura.
- El sistema es muy sensible a cambios, si cambian los diseños del juego o se añaden componentes deberán incluirse en el sistema de guardado.
- Si el sistema de carga genera alguna excepción durante la lectura del fichero, se considerará que el archivo de guardado está corrupto, se borrará y se generará uno nuevo.

A continuación, los métodos de guardado y carga, que escriben y leen sobre un fichero externo respectivamente. La coherencia del formato que escribe el método de guardado y la forma en la que lee el fichero el método de carga es esencial para que el sistema funcione correctamente:

```
public void saveGame(final boolean closeAfterEnd)
{
    // Solo guardara si estamos jugando
    if (m_gameState != GameState.STATE_GAME)
        return;

    // Si no hay jugador aun, tampoco guardar.
    if (m_player == null)
        return;

    Thread saveThread = new Thread(new Runnable()
    {
        public void run()
        {
            // Generacion de strings
            StringBuilder saveString = new StringBuilder();

            // Info general
            SaveType saveTypeGame = SaveType.SAVE_GAME;
            saveString.append(saveTypeGame.toString());
            saveString.append(";");

            // Debug
            saveString.append(String.valueOf(m_debugButton));
            saveString.append(";");
            saveString.append(String.valueOf(m_debugMode));
            saveString.append(";");
            saveString.append(String.valueOf(m_debugMMapsMode));
            saveString.append(";");
            saveString.append(String.valueOf(ai_active));
            saveString.append(";");

            // Temporizacion
            saveString.append(String.valueOf(m_tickCounter));
            saveString.append(";");
            saveString.append(m_gameSpeed.toString());
            saveString.append(";");
            saveString.append(m_secondsPerDay);
            saveString.append(";");
            saveString.append(m_secondsPerDayLow);
            saveString.append(";");
            saveString.append(m_secondsPerDayMed);
            saveString.append(";");
            saveString.append(m_secondsPerDayHigh);
            saveString.append(";");

            // Player
            saveString.append(m_player.getId());
            saveString.append(";");
        }
    });
}
```

```

// Opciones de partida
saveString.append(m_baseLoyalty);
saveString.append(";");
saveString.append(m_conquestLoyalty);
saveString.append(";");
saveString.append(m_neutralDiplomacy);
saveString.append(";");
saveString.append(m_baseInternationalSanctions);
saveString.append(";");
saveString.append(m_warDiplomacy);
saveString.append(";");
saveString.append(m_allyDiplomacy);
saveString.append(";");
saveString.append(m_neutralDiplomacy);
saveString.append(";");

(...)
// Countries
for (Country country : m_countryList.values())
{
    saveTypeGame = SaveType.SAVE_COUNTRY;
    saveString.append(saveTypeGame.toString());
    saveString.append(";");

    // Id
    int id = country.getId();
    saveString.append(id);
    saveString.append(";");

    // Dinero
    int money = country.getMoney();
    saveString.append(money);
    saveString.append(";");

    // Investigaciones
    ArrayList<Technology> investigationTree =
        country.getInvestigationTree();
    for (Technology tech : investigationTree)
    {
        // Tipo
        TechType techType = tech.getType();
        saveString.append(techType.toString());
        saveString.append(",");
    }
    saveString.append(";");

    // Territorios
    ArrayList<Territory> territoryList = country.getTerritories();
    for (Territory terry : territoryList)
    {
        saveString.append(terry.getId());
        saveString.append(",");
    }
    saveString.append(";");
}

```

```

// Diplomacia
HashMap<Country, Float> diplomacyValueMap =
    country.getDiplomacyValueMap();
for (Map.Entry<Country, Float> entry :
    diplomacyValueMap.entrySet())
{
    // Country
    saveString.append(entry.getKey().getId());
    saveString.append("&");

    // Value
    saveString.append(String.valueOf(entry.getValue()));
    saveString.append(",");
}

saveString.append(";");

// Cola investigacion
ArrayList<Queue> investigationQueue =
    country.getInvestigationQueue();
for (Queue queue : investigationQueue)
{
    // Tech
    Object item = queue.getItem();
    Technology tech = (Technology)item;

    saveString.append(tech.getType().toString());
    saveString.append("&");

    // Remain time
    int remainingTime = queue.getRemainingTime();
    saveString.append(remainingTime);
    saveString.append("&");

    // Total time
    int totalTime = queue.getTotalTime();
    saveString.append(totalTime);
    saveString.append(",");
}
saveString.append(";");

// Sanciones
int internationalSanctions =
    country.getInternationalSanctions();
saveString.append(internationalSanctions);
saveString.append(";");

// AI
AI ai = country.getAI();
saveString.append(ai.getType().toString());
saveString.append(";");

// Nueva linea
saveString.append("/");
}
(...)

```

```

// Creamos
FileHandle saveGame = Gdx.files.local("SavedGames/saveGame.dat");
if (!saveGame.exists())
{
    try
    {
        // Creamos fichero vacio
        saveGame.file().createNewFile();
        saveGame.writeString(Base64Coder.encodeString(""), false);
    }
    catch (IOException e)
    {
        e.printStackTrace();
    }
}

// Escribimos el texto (cifrado)
saveGame.writeString(Base64Coder.
    encodeString(saveString.toString()), false);

// Cerramos si es la orden
if (closeAfterEnd)
    Gdx.app.exit();
}
});
saveThread.start();
saveThread.setName("Saving Thread");
}

```

Observamos una comprobación de cerrado tras el guardado, esto es importante ya que en la opción “Guardar y salir” del propio juego no podemos simplemente llamar al método `saveGame` y cerrarlo, debido a que se ejecuta en otro hilo y corremos el riesgo de cerrarlo antes de que se haya terminado de guardar, por ese motivo parametrizamos el close de la aplicación en este método.

El método de lectura de partida guardada tiene una complementariedad obvia con el de guardado:

```
public void loadGame()
{
    try
    {
        // Leemos fichero
        FileHandle file = Gdx.files.local("SavedGames/saveGame.dat");
        String text = file.readString();

        // Desciframos
        text = Base64Coder.decodeString(text);

        // Limpiamos variables que se van a recargar completamente
        m_troopSquads.clear();
        stageSquads.clear();
        stageSquadTables.clear();

        String[] splitedLine = text.split("/");
        for (int i = 0; i < splitedLine.length; i++)
        {
            String[] splitedEntry = splitedLine[i].split(";");
            switch (SaveType.valueOf(splitedEntry[0]))
            {
                case SAVE_GAME:
                    m_debugButton = Boolean.valueOf(splitedEntry[1]);
                    m_debugMode = Boolean.valueOf(splitedEntry[2]);
                    m_debugMMapsMode = Boolean.valueOf(splitedEntry[3]);
                    ai_active = Boolean.valueOf(splitedEntry[4]);
                    m_tickCounter = Float.valueOf(splitedEntry[5]);
                    setSpeed(GameSpeed.valueOf(splitedEntry[6]));
                    m_secondsPerDay = Integer.valueOf(splitedEntry[7]);
                    m_secondsPerDayLow = Integer.valueOf(splitedEntry[8]);
                    m_secondsPerDayMed = Integer.valueOf(splitedEntry[9]);
                    m_secondsPerDayHigh = Integer.valueOf(splitedEntry[10]);
                    m_player =
                        getCountryById(Integer.valueOf(splitedEntry[11]));
                    m_baseLoyalty = Integer.valueOf(splitedEntry[12]);
                    m_conquestLoyalty = Integer.valueOf(splitedEntry[13]);
                    m_neutralDiplomacy = Integer.valueOf(splitedEntry[14]);
                    m_baseInternationalSanctions =
                        Integer.valueOf(splitedEntry[15]);
                    m_warDiplomacy = Integer.valueOf(splitedEntry[16]);
                    m_allyDiplomacy = Integer.valueOf(splitedEntry[17]);
                    m_neutralDiplomacy = Integer.valueOf(splitedEntry[18]);
                    m_baseTerritoryDamageRegenerationSpeed =
                        Float.valueOf(splitedEntry[19]);
                    m_baseTerritoryLoyaltyRegenerationSpeed =
                        Float.valueOf(splitedEntry[20]);
                    m_territoryMillitiaSpawningChance =
                        Integer.valueOf(splitedEntry[21]);
                    try
                    {
                        DateFormat formatter = new
                            SimpleDateFormat("dd[]MM[]yyyy");
                        m_gameDate = formatter.parse(splitedEntry[22]);
                    }
                    catch (ParseException e)
                    {
                        e.printStackTrace();
                    }
                    break;
            }
        }
    }
    (...)
}
```

```

case SAVE_COUNTRY:
    int id = Integer.valueOf(splitedEntry[1]);
    int money = Integer.valueOf(splitedEntry[2]);

    ArrayList<Technology> investigationTree = new
                                                ArrayList<>();
    String[] techSplit = splitedEntry[3].split(",");
    for (int j = 0; j < techSplit.length; j++)
    {
        if (techSplit[j] == "")
            continue;

        investigationTree.add(getTechnologyByType(
            TechType.valueOf(techSplit[j])));
    }

    ArrayList<Territory> territories = new ArrayList<>();
    String[] terrySplit = splitedEntry[4].split(",");
    for (int j = 0; j < terrySplit.length; j++)
    {
        if (terrySplit[j] == "")
            continue;
        territories.add(getTerritoryById(
            Integer.valueOf(terrySplit[j])));
    }

    String[] diplomacySplit = splitedEntry[5].split(",");
    for (int j = 0; j < diplomacySplit.length; j++)
    {
        // Valores
        String[] diploEntry = diplomacySplit[j].split("&");

        // Cobertura
        if (diploEntry[0] == "")
            continue;
        if (diploEntry[1] == "")
            continue;

        Country country =
            getCountryById(Integer.valueOf(diploEntry[0]));
        float value = Float.valueOf(diploEntry[1]);

        Country countryDest = getCountryById(id);
        Game.setCountriesDiplomacy(country, countryDest,
            value);
    }

    ArrayList<Queue> investigationQueue = new ArrayList<>();
    String[] investigationSplit = splitedEntry[6].split(",");
    for (int j = 0; j < investigationSplit.length; j++)
    {
        // Valores
        String[] queueItems =
            investigationSplit[j].split("&");

        // Cobertura
        if (queueItems[0] == "")
            continue;
        if (queueItems[1] == "")
            continue;
        if (queueItems[2] == "")
            continue;
    }

```

```

        Technology tech =
            getTechnologyByType(TechType.valueOf(queueItems[0]));
        int remainTime = Integer.valueOf(queueItems[1]);
        int totalTime = Integer.valueOf(queueItems[2]);

        if (tech == null)
            continue;

        // Cola nueva
        Queue queue = new Queue(tech, remainTime);
        queue.setTotalTime(totalTime);
        investigationQueue.add(queue);
    }

    int sanctions = Integer.valueOf(splitedEntry[7]);
    AIType aiType = AIType.valueOf(splitedEntry[8]);

    // Establecemos valores
    Country countryDest = getCountryById(id);

    // Establecemos valores
    countryDest.setMoney(money);
    countryDest.setInvestigationTree(investigationTree);
    countryDest.setTerritories(territories);
    countryDest.setInvestigationQueue(investigationQueue);
    countryDest.setInternationalSanctions(sanctions);
    countryDest.setAI(new AI(aiType, countryDest));
    break;
    (...)
}
catch (Exception e) // Si estamos aqui es que ha ocurrido algun error,
                    // consideramos el fichero corrupto y reiniciamos.
{
    // Dialogo
    showCorruptedFileDialog();

    // Registramos error
    Log.d("ERROR", "The save file you loaded is corrupted, the game will
        be closed now.");
}
}

```

3.14 MODO DEBUG

La aplicación incorpora además unas variables debug que si se activan en el código producirán cambios notables en el juego que facilitan la comprobación de diferentes cosas, estas variables son se establecen en el método de inicialización del juego:

```

public void initializeGameVariables()
{
    // Variables del juego
    m_debugButton = false;
    m_debugMode = false;
    m_debugMMapsMode = false;
    (...)
}

```

La variable `debugButton` activa dos botones que aparecen a la izquierda del juego y que sirven para implementarles efectos concretos y ver como se ejecutan. Los siguientes fragmentos ilustran muy bien lo comentado:

```
private void updateUI()
{
    (...)
    if (m_debugButton)
    {
        Table debugTable = new Table();
        debugTable.setFillParent(true);
        debugTable.center().left();
        debugTable.add(debugButton);
        debugTable.row();
        debugTable.add(debugButton2);
        stageUI.addActor(debugTable);
    }
    (...)
}
```

Donde estos botones definen listeners de click que llaman a métodos que usaremos para verificar el funcionamiento de acciones:

```
debugButton.addListener(new InputListener() {
    @Override
    public boolean touchDown(InputEvent event, float x, float y, int
        pointer, int button)
    {
        (...)
        debugAction();
        return true;
    }
});
```

```
debugButton2.addListener(new InputListener() {
    @Override
    public boolean touchDown(InputEvent event, float x, float y, int
        pointer, int button)
    {
        (...)
        debugAction2();
        return true;
    }
});
```

Como se puede apreciar en el método `debugAction2()`, éste se emplea para verificar de forma activa y bajo input del usuario que la milicia aparece correctamente:

```
private void debugAction2()
{
    getCountryByName("Mongolia").getBaseTerritory().spawnMillitia();
}
```

La variable `debugMode` por otro lado activa ciertas ventajas o “cheats” sobre el jugador, así como una aceleración de la velocidad de la partida, permitiendo al programador hacer pruebas de todo tipo mediante las siguientes ventajas:

- Velocidad del juego establecida a 1 día por segundo. (el doble que a la velocidad normal más alta)
- Colas de producción de todo tipo reducidas a un turno.
- Producción de todos los territorios del jugador establecida a 1000€ por turno.

Un ejemplo de fragmento de código que implementa estas ventajas:

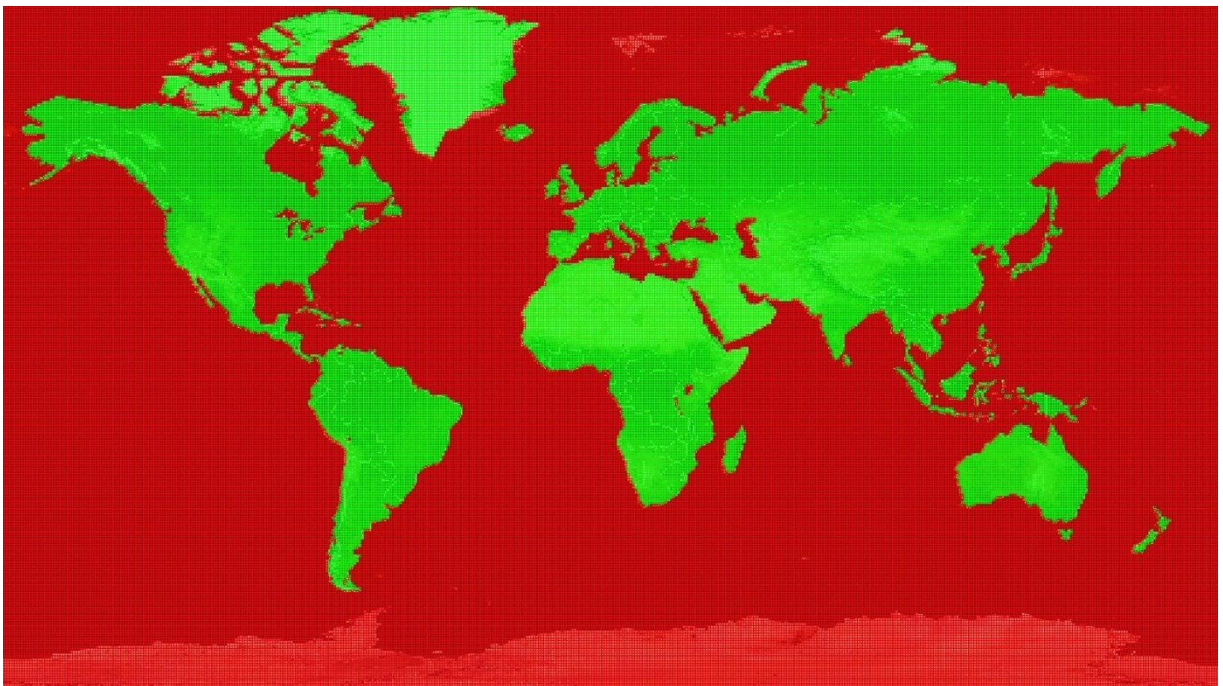
```
public int getMoneyProduction()
{
    int baseProduction = m_moneyProduction;
    int addProduction = 0;

    // Modo debug
    if (Game.m_debugMode && getOwner().equals(Game.m_player))
        baseProduction = 1000;
    (...)
}
```

La variable `debugMMapsMode` tiene una utilidad diferente a las anteriores y lo que hace es activar la ejecución de un fragmento de código ubicado en `render` que dibuja los nodos del mapa sobre el mismo como círculos rojos o verdes en función de si el nodo se encuentra sobre un territorio o sobre el mar. Esto tuvo una gran utilidad durante el desarrollo del sistema de nodos tanto para verificar su funcionamiento como para decidir qué resolución de los nodos se adaptaba de forma suficiente a las características del mapa.

```
// Debug de nodos de colision
if (m_debugMMapsMode)
{
    ShapeRenderer shapeRenderer = new ShapeRenderer();
    shapeRenderer.begin(ShapeRenderer.ShapeType.Line);
    for (int i = 0; i < m_mapPoints.length; i++)
        for (int j = 0; j < m_mapPoints[i].length; j++)
        {
            int id = m_mapPoints[i][j] + 128;
            if (id == 0)
                shapeRenderer.setColor(Color.RED);
            else
                shapeRenderer.setColor(Color.GREEN);
            shapeRenderer.circle(i * m_nodeSize, j * m_nodeSize,
                                m_nodeSize / 2);
        }
    shapeRenderer.end();
}
```

La siguiente imagen muestra la apariencia del mapa cuando esta variable está activada y en ella podemos apreciar cómo los nodos son capaces de establecer perfectamente las fronteras entre los territorios y el mar:



La imagen no se muestra de forma demasiado definida debido a problemas de aliasing en el software de captura de pantalla con los patrones repetitivos que forman los diminutos círculos que representan cada nodo.

Las variables debug deben estar obviamente desactivadas en el release de la aplicación.

4.- APLICACIÓN DOMINATIONS: INTELIGENCIA ARTIFICIAL

4.1 PERSONALIDADES DE IA

En este apartado vamos a comentar detalladamente todas las implementaciones que dan soporte a la inteligencia artificial de los países no controlados por el jugador, dotándoles de capacidad de decisión y planificación.

Hemos definido distintos tipos de IA que dan lugar a comportamientos o personalidades diferentes:

Tipo	Descripción
AI_IMPERIALIST	Países imperialistas, su meta va a ser expandir el territorio a base de conquistas.
AI_AGGRESSIVE	Países agresivos, cuya finalidad es en principio hacer daño y provocar conflictos. Tienen tendencia a atacar.
AI_DEFENSIVE	Países defensivos, su finalidad es defenderse a sí mismos y apoyar a los países aliados.
AI_PEACEFUL	Países pacíficos, no se dedican a producir tropas, intentan mantener buenas relaciones con todo el mundo y sobreviven a base de alianzas militares (Canadá)
AI_VARIABLE	Comportamiento variable, la personalidad del país cambia aleatoriamente entre una de las anteriores cada cierto tiempo, intenta simular el comportamiento de países políticamente inestables.
AI_RANDOM	A los países que se declaran con esta IA se les asignará otra IA aleatoria entre alguna de las anteriores.
AI_CUSTOM	Países con un comportamiento único, particularizado para ese país en concreto (no se ha utilizado)

La mayoría de decisiones de IA se han diseñado en base a métodos probabilísticos, dichos métodos se han usado también a lo largo del resto del juego y ya han sido nombrados en alguna ocasión, los siguientes métodos implementan las lógicas probabilísticas que se usarán:

```
// Devuelve un número aleatorio equiprobable entre el mínimo y el
máximo.
public static int getRandomValue(int min, int max)
{
    Random rand = new Random();
    int randomNum = rand.nextInt((max - min) + 1) + min;
    return randomNum;
}

// Realiza una "tirada de dados" que tiene un porcentaje (chance) de
dar true.
// Ejemplo: si hacemos 100 llamadas a roll_chance(10), diez de ellas
debería ser true y las demás false.
public static boolean roll_chance(int chance)
{
    int ranValue = getRandomValue(1, 100);
    if (ranValue <= chance)
        return true;
    return false;
}

// Le pasamos una lista de probabilidades y devuelve la opción que ha
resultado de la "tirada"
// Ejemplo: si hacemos 100 llamadas con los chances 10, 40, 50: 10
llamadas devolverán un 1, 40 un 2 y 50 un 3
// Este método es el más empleado en IA, ya que podemos tomar
decisiones en base a probabilidades diferentes.
// En el ejemplo anterior los pesos estaban sobre 100, pero no tiene
por que ser, una llamada con 1, 4 y 5 daría exactamente el mismo
resultado que la comentada.
public static int rollOptionsByChance(int[] chances)
{
    int output = 0;

    // Referencia de valor maximo, normalmente 100%
    int maxReff = 0;
    for (int i = 0; i < chances.length; i++)
    {
        // Cualquier chance menor que 0 es 0
        if (chances[i] < 0)
            chances[i] = 0;

        // Valor maximo
        maxReff += chances[i];
    }
}
```

```

    int ranValue = getRandomValue(0, maxReff);

    // Comprobamos que opcion se ha dado
    int botValue = 0;
    int topValue = 0;
    for (int i = 0; i < chances.length; i++)
    {
        // Valor minimo
        if (i != 0)
            botValue += chances[i - 1];

        // Valor maximo
        topValue += chances[i];

        // Comprobacion
        if (ranValue > botValue && ranValue <= topValue)
            output = i;
    }
    return output;
}

// Devuelve un elemento aleatorio de un HashMap
public static <T> Object getRandomElementFromHashMap(HashMap map)

// Devuelve un elemento aleatorio de un ArrayList
public static <T> T getRandomElementFromArrayList(ArrayList<T> list)

```

Otros métodos probabilísticos que también empleamos en IA son aquellos que nos permiten obtener puntos aleatorios en los territorios que respeten ciertas reglas. En la inicialización del territorio se guarda un array con los nodos que lo forman por lo que estos métodos se basarán en obtener puntos de esa lista.

```

// Devuelve un punto aleatorio del territorio
public Vector2 getRandomPoint()
{
    (...)

    // Devolvemos un punto aleatorio del array de puntos
    ArrayList<Integer> ranPos =
        m_positions.get(Utils.getRandomValue(0,
                                                m_positions.size() - 1));
    return Utils.getMapPositionByNodePosition(new
        Vector2(ranPos.get(0), ranPos.get(1)));
}

// Devuelve un punto aleatorio del territorio comprendido en el
rectángulo que circunscribe al circulo centrado en el centro del
territorio y de radio dado.
public Vector2 getRandomPointFromCenterWithRandomRadius(int radius)
{
    return getRandomPointFromCenterWithRandomXY(radius, radius);
}

```

```

// Devuelve un punto aleatorio del territorio comprendido en el
rectángulo centrado en el centro del territorio y de lado horizontal 2
* maxX y 2 * maxY de lado vertical
public Vector2 getRandomPointFromCenterWithRandomXY(int maxX, int
    maxY)
{
    float newX;
    float newY;
    int maxTrys = 100;
    int trys = 0;
    int terryId = 0;

    // Si el punto anterior no esta contenido en el territorio,
    seguiremos probando puntos hasta que alguno si
    este dentro o venza el antifreeze.
    do
    {
        // Probamos otro punt
        newX = m_center.x + Utils.getRandomValue(-maxX, maxX);
        newY = m_center.y + Utils.getRandomValue(-maxY, maxY);

        // ID
        terryId = Utils.getTerritoryIdByMapPoint(
            new Vector2(newX, newY));

        // Antifreeze
        trys++;
    }
    while (terryId != m_id && trys < maxTrys); // Antifreeze

    // Si ha vencido el antifreeze entonces usamos el centro y fin
    if (trys >= maxTrys)
    {
        newX = m_center.x;
        newY = m_center.y;
    }

    // Add radio aleatorio
    return new Vector2(newX, newY);
}

```

En este método se comprueban puntos hasta encontrar uno que se encuentre en el rectángulo que envuelve al territorio, para evitar un loop infinito se ha implementado un contador, en caso de vencer se abandonará la comprobación y se le asignará el centro como punto, tratándose éste de un punto bien conocido.

La implementación de decisiones por eventos probabilísticos resulta beneficiosa al aleatorizar las decisiones, acercando así a las IAs al razonamiento humano que tienen un alto componente de impredecibilidad.

Otro parámetro relevante en la toma de decisiones es el parámetro dangerouness, que mide la peligrosidad en la que se encuentra un país concreto, dicho valor le ayudará a tomar decisiones, se calcula como el cociente la fuerza ofensiva efectiva que podría atacar al país entre la fuerza defensiva efectiva que podría defenderlo en ese caso. A continuación, se muestra su implementación y en los comentarios pueden verse ejemplos de cálculo:

```
public static float getDangerouness(Territory myTerry)
{
    // El valor de peligrosidad: Representa el factor entre el valor
    // de ataque de tropas atacantes que podrian
    // atacar el territorio y el valor de defensa de tropas defensoras
    // que podrian defenderlo y afectado por la distancia.
    // Ejemplo: Si tenemos 10k de AP a 2 de distancia y 20k de DP a 4
    // de distancia el factor seria (10k AP / 2) / (20k DP / 4) = 1,
    // la peligrosidad seria 1.
    // Ejemplo 2: Tenemos 10k de AP a 10 de distancia y 20k de DP a 10
    // de distancia, el factor será (10k / 10) / (20k / 10) = 0.5, es
    // decir, un escenario poco peligroso.
    // Ejemplo 3: Tenemos 10k de AP a 1 de distancia y 20k de AP a 10
    // de distancia, el factor será (10k / 1) / (20k / 10) = 5, es
    // decir, un escenario bastante peligroso, pues el AP efecto es 5
    // veces el DP efectivo.
    (...)

    // Comprobamos distancia a países y la relacion con ellos
    Vector2 myPosition = myTerry.getTerritoryCenter();
    for (Country country : Game.m_countryList.values())
        for (Territory terry : country.getTerritories())
        {
            float dist = Utils.calculateEuclideanDistance(
                terry.getTerritoryCenter(), myPosition);
            (...)

            // Comprobamos tropas de territoryo
            TroopArray terriTroops = terry.getTerritoryTroops();
            if (terriTroops.hasTroops())
            {
                // Calculo de tropas
                if (myTerry.getOwner().isAlliedWith(country) ||
                    myTerry.getOwner().equals(country))
                    DPFactor +=
                        (terriTroops.getTotalDefensePower(myTerry) / 2)
                        / dist;
                else if (myTerry.getOwner().isHostileWith(country))
                    APFactor += terriTroops.getTotalAttackPower() /
                        dist;
            }
        }
}
```

```

        // Comprobamos tropas deployed
        TroopArray deploTroops = terry.getDeployedTroops();
        if (deploTroops.hasTroops())
        {
            // Calculo de tropas
            if (myTerry.getOwner().isAlliedWith(country) ||
                myTerry.getOwner().equals(country))
                DPFactor +=
                    (deploTroops.getTotalDefensePower(myTerry) / 2)
                    / dist;
            else if (myTerry.getOwner().isHostileWith(country))
                APFactor += deploTroops.getTotalAttackPower() /
                    dist;
        }
    }

    // Comprobamos distancia a squads y sus diplomacias
    for (TroopSquad squad : Game.m_troopSquads)
    {
        float dist = Utils.calculateEuclideanDistance(
            squad.getPosition(), myPosition);

        (...)

        // Calculo de tropas
        if (myTerry.getOwner().isAlliedWith(squad.getOwner()) ||
            myTerry.getOwner().equals(squad.getOwner()))
            DPFactor += (squad.getTotalDefensePower(myTerry) / 2) /
                dist;
        else if (myTerry.getOwner().isHostileWith(squad.getOwner()))
            APFactor += squad.getTotalAttackPower() / dist;
    }

    (...)
    dangerousness = APFactor / DPFactor;
    return dangerousness;
}

```

Si el factor de peligrosidad es inferior a 1, el país se encuentra en una posición defensiva buena, si es superior, en una posición peligrosa.

Procedemos ahora a detallar la implementación de la propia IA, en primer lugar, la actualización se produce desde el método de Game updateGame como ya hemos visto antes, que fuerza la ejecución de un thread independiente que es quien ejecuta la actualización de la IA llamando al método updateAI para cada país:

```

public void updateGame()
{
    (...)
    // Update AI
    m_aiThread.update();
    (...)
}

```

```

@Override
public void run()
{
    (...)
    for (Country country : m_countryList)
        country.updateAI();
}

public void updateAI()
{
    if (equals (Game.m_player))
        return;
    if (!Game.ai_active)
        return;

    // Update IA
    m_ai.updateAI();
}

public void updateAI()
{
    switch (m_type)
    {
        case AI_IMPERIALIST:
            updateImperialistAI();
            break;
        case AI_AGGRESSIVE:
            updateAgressiveAI();
            break;
        case AI_DEFENSIVE:
            updateDefensiveAI();
            break;
        case AI_PEACEFUL:
            updatePeacefulAI();
            break;
        case AI_VARIABLE:
            updateVariableAI();
            break;
        case AI_CUSTOM:
            updateCustomAI();
            break;
        case AI_RANDOM:
        default:
            // No deberiamos llegar aqui nunca
            m_type = getRandomAI();
            break;
    }
}

```

Observamos que para IAs de tipo random obtenemos una IA aleatoria y la establecemos como tipo básico tras el primer update:

```
private AIType getRandomAI()
{
    AIType type = null;
    int option = Utils.rollOptionsByChance(new int[]{20, 20, 20, 20,
                                                    20});

    switch (option)
    {
        case 0:
            type = AIType.AI_IMPERIALIST;
            break;
        case 1:
            type = AIType.AI_AGGRESSIVE;
            break;
        case 2:
            type = AIType.AI_DEFENSIVE;
            break;
        case 3:
            type = AIType.AI_PEACEFUL;
            break;
        case 4:
            type = AIType.AI_VARIABLE;
            break;
    }
    return type;
}
```

La IA de tipo variable también se implementa de forma diferente al resto ya que la IA debe guardar el tipo variable y además el tipo temporal asignado:

```
private void updateVariableAI()
{
    // Si el timer vence, asignamos nuevo tipo y reiniciamos
    if (m_variableTick >= m_variableTimeout)
    {
        m_variableTick = 0;

        // Los timeouts de cambio van de 30 a 300 dias (mucho tiempo)
        m_variableTimeout = Utils.getRandomValue(30, 300);

        // Asignamos tipo aleatorio
        int option = Utils.rollOptionsByChance(new int[]{25, 25, 25,
                                                         25});

        switch (option)
        {
            case 0:
                m_variableTemporalType = AIType.AI_IMPERIALIST;
                break;
            case 1:
                m_variableTemporalType = AIType.AI_AGGRESSIVE;
                break;
            case 2:
                m_variableTemporalType = AIType.AI_DEFENSIVE;
                break;
            case 3:
                m_variableTemporalType = AIType.AI_PEACEFUL;
                break;
        }

        // Log debug IA
        Log.d("IA Debug", m_owner.getName() + " has faked its type to "
              + m_variableTemporalType.toString());
    }
    else
    {
        // Incrementamos contador
        m_variableTick++;

        // Llamamos a método de su tipo fake
        switch (m_variableTemporalType)
        {
            case AI_IMPERIALIST:
                updateImperialistAI();
                break;
            case AI_AGGRESSIVE:
                updateAgressiveAI();
                break;
            case AI_DEFENSIVE:
                updateDefensiveAI();
                break;
            case AI_PEACEFUL:
                updatePeacefulAI();
                break;
        }
    }
}
```

El funcionamiento es sencillo: cuando vence un contador le asigna una IA aleatoria entre las estándar, y el resto del tiempo ejecuta los updates correspondientes a su IA temporal.

La existencia de la IA variable ha de ser tenida en cuenta también en el getter de tipo, habiendo un método que devuelve el tipo real o efectivo:

```
public AIType getEffectiveType()
{
    // Si es variable no devolvemos el tipo,
    if (m_type == AIType.AI_VARIABLE)
        return m_variableTemporalType;
    return m_type;
}
```

El resto de IAs llaman a un método específico para actualizar su IA como se puede apreciar en la función de update, aunque son métodos diferentes todos siguen una estructura muy similar, vamos a observar dos ejemplos, la IA agresiva y la IA defensiva para comentar las similitudes y diferencias:

```
private void updateAgressiveAI()
{
    // 10% por turno de gastar dinero
    if (Utils.roll_chance(10))
    {
        // Opciones de gasto economico
        int option = Utils.rollOptionsByChance(new int[]{20, 20, 50, 10});

        switch (option) {
            case 0: // 20%
                decideBuildingToBuild(15, 15, 5); // umbrales de
                                                    damage y lealtad en los cuales construir el
                                                    edificio de reparacion asociado sera
                                                    prioritario
                break;
            case 1: // 20%
                decideTechToDevelop(60, 25, 15); // Peso de
                                                    investigacion de tipo militar, de tipo
                                                    diplomatico y de tipo economico.
                break;
            case 2: // 50%
                decideTroopsToTrain(80, 20); // Peso de tropas
                                                    ofensivas, defensivas
                break;
            case 3: // 10%
                decideToImproveDiplomacy();
                break;
        }

        // Decide a que territorio deberia enviar tropas
        decideTerritoryToDefend(5, 5);
    }
}
```

```

// Decide a que territorio deberia enviar tropas ofensivas
decideTerritoryToAttack(10);

// Limpiamos squads
clearUntaskedSquads();
}

private void updateDefensiveAI()
{
    // 10% por turno de gastar dinero
    if (Utils.roll_chance(10))
    {
        // Opciones de gasto economico
        int option = Utils.rollOptionsByChance(new int[]{25, 25, 35,
                                                         15});

        switch (option)
        {
            case 0: // 25%
                decideBuildingToBuild(15, 15, 0.7f); // umbrales de
                                                         damage y lealtad en los cuales construir
                                                         el edificio de reparacion asociado sera
                                                         prioritario.

                break;
            case 1: // 25%
                decideTechToDevelop(60, 25, 15); // Peso de
                                                         investigacion de tipo militar, de tipo
                                                         diplomatico y de tipo economico.

                break;
            case 2: // 35%
                decideTroopsToTrain(20, 80); // Peso de tropas
                                                         ofensivas, defensivas

                break;
            case 3: // 15%
                decideToImproveDiplomacy();

                break;
        }
    }

    // Decide a que territorio deberia enviar tropas
    decideTerritoryToDefend(25, 1.5f);

    // Decide a que territorio deberia enviar tropas ofensivas
    decideTerritoryToAttack(5);

    // Limpiamos squads
    clearUntaskedSquads();
}

```

Podemos hacer las siguientes apreciaciones de los métodos anteriores que además serán generalizables a los otros métodos de update IA:

- Inicialmente se realiza una decisión relacionada con la **gestión económica**: la IA decide si construir edificios, desarrollar tecnología, entrenar tropas o mejorar las relaciones diplomáticas.
- Posteriormente se realiza una decisión relacionada con la **gestión defensiva**: la IA decide que territorios debe defender si es que debe defender alguno.
- Seguido a lo anterior realizará una decisión similar relacionada con la **gestión ofensiva**: la IA decide que territorios debe atacar si es que debe atacar alguno.
- Por último, la IA realiza una limpieza de tropas sin tarea, lo cual englobaremos dentro de la **gestión de tropas y tareas**.
- Los parámetros que recibe cada llamada de las que aparecen tiene un impacto sobre el peso de las decisiones y es en lo que se diferencian fundamentalmente los distintos updates de cada IA, por ejemplo, observamos que en la IA defensiva tiene mucho más peso la producción de tropas defensivas y en la IA ofensiva a la inversa.

4.2 GESTIÓN ECONÓMICA

Conjunto de decisiones que se llaman desde (particularizado al update defensivo) desde este fragmento de código:

```
private void updateDefensiveAI()
{
    // 10% por turno de gastar dinero
    if (Utils.roll_chance(10))
    {
        // Opciones de gasto economico
        int option = Utils.rollOptionsByChance(new int[]{25, 25, 35,
                                                         15});

        switch (option)
        {
            case 0: // 25%
                decideBuildingToBuild(15, 15, 0.7f); // umbrales de
                                                         damage y lealtad en los cuales construir
                                                         el edificio de reparacion asociado sera
                                                         prioritario

                break;
            case 1: // 25%
                decideTechToDevelop(60, 25, 15); // Peso de
                                                         investigacion de tipo militar, de tipo
                                                         diplomatico y de tipo economico.

                break;
            case 2: // 35%
                decideTroopsToTrain(20, 80); // Peso de tropas
                                                         ofensivas, defensivas

                break;
            case 3: // 15%
                decideToImproveDiplomacy();

                break;
        }
    }
    (...)
}
```

Podemos observar en primer lugar que solo en 1 de cada 10 updates de IA el país comprobará la posibilidad de realizar un gasto y que, en caso de hacerlo, se decidirá entre 4 posibilidades con probabilidades diferentes definidas en el array que le pasamos al método `rollOptionsByChance`, estos porcentajes dependerán del tipo de IA, dándole distinto peso a las posibilidades.

Para la primera opción posible: Construir edificio

```
private void decideBuildingToBuild(float damageThreshold, float
loyaltyThreshold, float dangerousnessThreshold)
{
    for (Territory terry : m_owner.getTerritories())
    {
        BuildType build = null;
        // Si la situacion es peligrosa
        if (Formulas.getDangerouness(terry) > dangerousnessThreshold)
        // Umbral de peligrosidad
        {
            // 70% de construir fortaleza y 15% de construir taller y
            // 15% de construir parlamento
            int option = Utils.rollOptionsByChance(new int[]{70, 15,
                                                                15});

            switch (option)
            {
                case 0: // 70%
                    build = BuildType.BUILD_FORTRESS;
                    break;
                case 1: // 15%
                    build = BuildType.BUILD_WORKSHOP;
                    break;
                case 2: // 15%
                    build = BuildType.BUILD_PARLIAMENT;
                    break;
            }
        }
        // La situacion no es peligrosa
        // Si el damage del territorio supera cierto umbral
        else if (terry.getDamage() > damageThreshold)
            build = BuildType.BUILD_WORKSHOP;
        else if (terry.getDamage() > loyaltyThreshold)
            build = BuildType.BUILD_PARLIAMENT;
    }
}
```

```

else // La situacion ni es peligrosa no tiene problemas de
    damage o lealtad
{
    // 70% de construir banco y 15% de construir taller y 15%
    de construir parlamento
    int option = Utils.rollOptionsByChance(new int[] {70, 15,
                                                    15});

    switch (option)
    {
        case 0: // 70%
            build = BuildType.BUILD_BANK;
            break;
        case 1: // 15%
            build = BuildType.BUILD_WORKSHOP;
            break;
        case 2: // 15%
            build = BuildType.BUILD_PARLIAMENT;
            break;
    }
}

if (build != null)
{
    // Leemos edificio actual del territorio
    Building building = null;
    Building prevBuilding = terry.getBuildingByType(build);
    if (prevBuilding == null)
        building = Game.getBuildingByTypeAndLevel(build, 1);
    else
        building = Game.getBuildingByTypeAndLevel(build,
                                                    prevBuilding.getLevel() + 1);

    if (building != null)
        if (canCountryAffordThisWaste(building.getCost()));
        {
            // Intentamos construir
            String result =
                terry.tryAddBuildingToQueue(build);

            // Si lo logramos, abortamos iteraciones, solo
            debe construir una vez por roll.
            if (result == null)
                break;
        }
}
}
}

```

El algoritmo de construcción de edificios sigue la siguiente lógica en vista de la implementación del método:

- Se le pasan por parámetro umbrales de damage, de lealtad y de peligrosidad.
- Realizará las siguientes comprobaciones para todos los territorios que el país posea:
 - Si la peligrosidad supera el umbral construirá edificios defensivos con probabilidades definidas en el array de opciones.
 - Si no supera el umbral de peligrosidad, comprobará si el damage del territorio supera el umbral de damage, de ser así construirá el taller, que es un edificio que acelera la recuperación del damage.
 - Si no supera el umbral de peligrosidad ni el de damage, comprobará si la lealtad del territorio está por debajo del umbral, de ser así construirá el parlamento, que es un edificio que fortalece la lealtad del territorio.
 - Si no se cumple ninguna de las condiciones anteriores construirá banco, taller o parlamento, con probabilidades definidas en el array de opciones y que le dan una considerable prioridad al banco.

Una vez decidido el edificio a construir realizará las dos siguientes comprobaciones, si no se cumple alguna de ellas el edificio no se construirá:

- ¿El edificio puede ser construido? Para ello lo comprobará con la función `tryAddBuildingToQueue` que es la misma que verifica que el jugador puede construir el edificio al intentarlo desde la interfaz.
- ¿Puede el país afrontar este gasto? Mediante el método `canCountryAffordThisWaste` se verifican una serie de reglas sobre el gasto que supone esta producción para la economía del país, se puede observar su implementación en el siguiente fragmento.

```
private boolean canCountryAffordThisWaste(int cost)
{
    return cost <= getMaxAffordableWaste();
}
```

```

private int getMaxAffordableWaste()
{
    // El gasto en un único item no podra superar nunca la produccion
    // sumada de productionDays (numero de dias)
    // Se establece ademas una tasa de ahorro, no se podrá gastar en
    // un único item mas de la cantidad que dejase
    // al país sin savingPct * dinero total, si tiene 1000 y una tasa
    // de ahorro de 0.3, un gasto no podra exceder los 700$)
    int productionDays = 10;
    float savingPct = 0.3f;
    switch (getEffectiveType())
    {
        case AI_IMPERIALIST:
            productionDays = 5;
            savingPct = 0.2f;
            break;
        case AI_AGGRESSIVE:
            productionDays = 3;
            savingPct = 0.1f;
            break;
        case AI_DEFENSIVE:
            productionDays = 8;
            savingPct = 0.3f;
            break;
        case AI_PEACEFUL:
            productionDays = 10;
            savingPct = 0.4f;
            break;
    }

    // Caps
    int productionCap = m_owner.getTotalMoneyProduction() *
        productionDays;
    int actualMoneyCap = (int) (m_owner.getMoney() * (1 - savingPct));

    // Producimos segun el cap inferior
    return Math.min(productionCap, actualMoneyCap);
}

```

El máximo gasto que puede ejecutar el país no puede exceder su producción total de cierto número de días ni tampoco un porcentaje del total de su dinero actual definido mediante una tasa de ahorro, ambos valores comentados dependen del tipo de IA. Añadir que además este método se utiliza para verificar la capacidad de gasto en cualquier otra decisión relacionada con la gestión económica.

Para la segunda opción posible: Desarrollar tecnología

```
private void decideTechToDevelop(int militaryWeight, int
diplomacyWeight, int economyWeight)
{
    ArrayList<Technology> militaryTech = new ArrayList<>();
    ArrayList<Technology> economyTech = new ArrayList<>();
    ArrayList<Technology> diplomacyTech = new ArrayList<>();
    ArrayList<Technology> selectedTechSchool = new ArrayList<>();
    for (Technology tech : Game.m_technologyList)
    {
        switch (tech.getTechSchool())
        {
            case TECH_SCHOOL_MILITARY:
                militaryTech.add(tech);
                break;
            case TECH_SCHOOL_ECONOMY:
                economyTech.add(tech);
                break;
            case TECH_SCHOOL_DIPLOMACY:
                diplomacyTech.add(tech);
                break;
        }
    }

    int option = Utils.rollOptionsByChance(new int[]{militaryWeight,
                                                    diplomacyWeight, economyWeight});
    switch (option)
    {
        case 0:
            selectedTechSchool = militaryTech;
            break;
        case 1:
            selectedTechSchool = diplomacyTech;
            break;
        case 2:
            selectedTechSchool = economyTech;
            break;
    }
    // Elegimos (de momento aleatorio, TODO: Logica más desarrollada
    Technology tech =
        Utils.getRandomElementFromArrayList(selectedTechSchool);
    // Intentamos desarrollar
    if (canCountryAffordThisWaste(tech.getCost()))
        m_owner.tryAddTechnologyToQueue(tech);
}
```

La decisión de este tipo es considerablemente más simple que en el caso anterior, primero separamos todas las tecnologías por rama (militar, económica, diplomática), en segundo lugar, elegimos el tipo a desarrollar, dicha decisión será probabilística y dependerá de los pesos que se le pasen a la función como parámetro, recordamos que en el caso defensivo:

```
case 1: // 25%
    decideTechToDevelop(60, 25, 15); // Peso de investigacion de tipo
    militar, de tipo diplomatico y de tipo economico.
```

Para finalizar se elegirá una tecnología aleatoria del tipo elegido y se intentará desarrollar si eso es posible y el gasto entra dentro de las condiciones de ahorro que se han expuesto ya para el ejemplo anterior. Este algoritmo de decisión es considerablemente primitivo y se desarrollará más en el futuro.

Para la tercera opción posible: Entrenar tropas

```
private void decideTroopsToTrain(int offensiveWeight, int
defensiveWeight)
{
    for (Territory terry : m_owner.getTerritories())
    {
        UnitType type = null;
        // Si la situacion es peligrosa, la probabilidad de sacar
        // tropas defensivas será aumentada en un
        // dangerounessImpact por cada punto de peligrosidad.
        int option = Utils.rollOptionsByChance(new
            int[]{offensiveWeight, (int)(defensiveWeight * (1 +
                Formulas.getDangerouness(terry)))});
        switch (option)
        {
            case 0: // Ofensiva
                int offType = Utils.rollOptionsByChance(new int[]{60,
                                                                    30, 10});
                switch (offType)
                {
                    case 0: // 70% assault
                        type = UnitType.UNIT_INFANTRY_ASSAULT;
                        break;
                    case 1: // 30% abrams
                        type = UnitType.UNIT_ARTILLERY_ABRAMS;
                        break;
                    case 2: // 10% fighters
                        type = UnitType.UNIT_AIRCRAFT_BOMBER;
                        break;
                }
                break;
            case 1: // Defensiva
                int deffType = Utils.rollOptionsByChance(new int[]{70,
                                                                    30});
                switch (deffType)
                {
                    case 0: // 70% mortar
                        type = UnitType.UNIT_INFANTRY_MORTAR;
                        break;
                    case 1: // 30% akatsiya
                        type = UnitType.UNIT_ARTILLERY_AKATSIYA;
                        break;
                }
                break;
        }
    }
}
```

```

if (type != null)
{
    float randomNumber = (float)Utils.getRandomValue(70, 90) /
                                                                    100f;

    int maxTrainableUnits =
        (int) (((float) getMaxAffordableWaste() * randomNumber) /
              (float) Game.getUnitByType(type).getCost());

    // Si la cantidad es demasiado baja, ignorar
    if (maxTrainableUnits > getMinUnitsTrainableByType(type))
    {
        Troops troop = new Troops(maxTrainableUnits,
                                   Game.getUnitByType(type), m_owner, terry);

        // Intentamos encolar
        String result = terry.tryAddTroopToQueue(troop);

        // Si lo logramos, abortamos iteraciones, solo debe
        // construir una vez por roll.
        if (result == null)
            break;
    }
}
}
}
}

```

Tras analizar el código podemos concluir que la decisión de entrenamiento de tropas sigue las siguientes reglas:

- Todas las decisiones a la que nos referimos a continuación se realizan en cada territorio.
- Se elige en primer lugar si entrenar tropa ofensiva o defensiva, la probabilidad de cada opción viene dada como parámetro que, como en los casos anteriores, dependerá del tipo de IA.
- El peso que determina la probabilidad de producir tropas defensivas está además amplificado por la variable peligrosidad del territorio en cuestión.
- La tropa a producir es probabilística también, pero sus valores son estáticos.
- El número de tropas a producir está determinado por un porcentaje que oscila aleatoriamente entre un 70% y un 90% del máximo gasto asumible por el país en cuestión.
- Si se cumplen todas las condiciones necesarias, se producirá la tropa, además de dichas condiciones habituales, deberá verificar que el número de tropas a producir supera un umbral definido por el método *getMinUnitsTrainableByType*.

```

private int getMinUnitsTrainableByType(UnitType type)
{
    int output = 0;
    switch (type)
    {
        case UNIT_INFANTRY_MORTAR:
            output = 500;
            break;
        case UNIT_INFANTRY_ASSAULT:
            output = 500;
            break;
        case UNIT_ARTILLERY_AKATSIYA:
            output = 100;
            break;
        case UNIT_ARTILLERY_ABRAMS:
            output = 100;
            break;
        case UNIT_AIRCRAFT_BOMBER:
            output = 50;
            break;
    }
    return output;
}

```

Para la cuarta opción posible: Mejorar diplomacia

```

private void decideToImproveDiplomacy()
{
    // Ordenamos los países por hostilidad, el objetivo es que se
    // intente potenciar a los mas hostiles
    ArrayList<Country> hostile =
        Utils.orderAscCountriesHashMapKeysByValue(
            m_owner.getDiplomacyValueMap());
    Collections.reverse(hostile);
    for (int i = 0; i < hostile.size(); i++)
    {
        // Hay un 30% de probabilidad de que la iteracion de un salto
        // aleatorio de entre 0 y 10 unidades, para
        // darle aleatoriedad a la par que se intenta potenciar a
        // los países mas hostiles
        if (Utils.roll_chance(30))
            i += Utils.getRandomValue(0, 10);

        // Hemos llegado al final
        if (i >= hostile.size())
            break;

        Country target = hostile.get(i);
        DiplomaticActionType diploType = null;

        // Ignorarse a si mismo
        if (m_owner.equals(target))
            continue;
    }
}

```

```

// La accion cambia segun estado
// Si es hostil y las ias son defensivas o pacificas, intentar
    siempre firmar la paz
if (m_owner.isHostileWith(target) && (getEffectiveType() ==
    AIType.AI_DEFENSIVE || getEffectiveType() ==
    AIType.AI_PEACEFUL))
    diploType = DiplomaticActionType.TYPE_PEACE_REQUEST;
else if (!m_owner.isAlliedWith(target)) // Si no es aliado,
    ofrecera aumento y a veces coalicion.
{
    int option = Utils.rollOptionsByChance(new int[]{70, 30});
    switch (option)
    {
        case 0:
            diploType =
                DiplomaticActionType.TYPE_IMPROVE_DIPLOMACY;
            break;
        case 1:
            diploType =
                DiplomaticActionType.TYPE_COALITION_REQUEST;
            break;
    }
}

if (diploType != null)
{
    // Coste que le van a exigir
    int costImprove =
        Formulas.getCostToAcceptADiplomacyRequestOfCountry(
            Game.getDiplomaticActionByType(diploType),

m_owner, target);

    // TODO: mejorar la inteligencia de este punto, cuando
    haya ataques y demas, las IAs deberian de
    tener en cuenta esto para su planificacion militar.
    // Si el coste es inferior al deseado,
    if (canCountryAffordThisWaste(costImprove))
        Game.getDiplomaticActionByType(diploType)
            .tryExecuteAction(m_owner, target, true);
}
}
}

```

El código anterior muestra el siguiente comportamiento:

- Se elabora una lista de los países más hostiles del país que toma la decisión.
- Comienza a comprobar los países con los que mejorar la diplomacia en orden.
- Cada comprobación tiene un 30% de realizar un salto en la lista de un número aleatorio de 0 y 10 unidades, de esta forma evitamos que siempre compruebe los mismos, aunque manteniendo cierta prioridad cuanto peor sea la diplomacia entre ambos.
- Si el país está en guerra y la IA es defensiva o pacífica, intentará firmar la paz.
- Si el país no está en guerra ni es aliado, intentará mejorar la diplomacia o firmar una coalición (70% y 30% de probabilidad respectivamente)
- Una vez elegida la acción diplomática comprobará cuánto dinero deberá ofrecer por dicha oferta para que sea aceptada, las IAs siempre ofrecerán la cantidad exacta que la IA del otro país vaya a exigir, dicho valor se obtiene con el método que `getCostToAcceptADiplomacyRequestOfCountry` que es el mismo que se emplea para calcular cuánto dinero debe ofrecer el jugador a otros países para que éstos acepten las propuestas.
- Si el coste es aceptable para los parámetros de ahorro del país en cuestión y se cumplen las condiciones habituales verificadas por el método `tryExecuteAction`, se realizará el acuerdo.

4.3 GESTIÓN DEFENSIVA

Dentro de la gestión defensiva englobamos todas aquellas decisiones del país encaminadas a la defensa de los territorios propios y los de sus aliados. Dicho sistema está implementado en base a la cuantificación del valor de los territorios candidatos a defender y selección del mejor de ellos.

Dichas gestiones se inician en el método `decideTerritoryToDefend` que se llama desde todos los `updateIA` y el algoritmo funciona del siguiente modo:

- La decisión de envío de tropas defensivas se realiza en orden de prioridad, de forma que, si se han enviado todas las tropas para defender territorios de mayor prioridad, puede ocurrir que queden territorios de menor prioridad sin defender. Esta priorización se realiza secuencialmente en el orden en el que se ejecutan las comprobaciones en el método, que también encaja con su posición en este esquema.

- En primer lugar comprobamos los territorios propios, si alguno de ellos está siendo conquistado se ejecutará una tarea defensiva sobre dicho territorio inmediatamente si no está ya siendo defendido, la prioridad de defensa es además por valor de territorio, de modo que si varios están siendo atacados se defenderá al territorio más valioso, dicho valor del territorio no se calcula en ese instante sino que es una variable de territorio que se actualiza cada vez que se modifica alguna de las variables que intervienen en su cálculo para aliviar carga computacional. Dicho valor se calcula con el siguiente método de la clase Formulas, que calcula el valor en base a la producción del territorio y los edificios que posee, cada nivel de edificio equivale a 10 puntos de producción en el cálculo del valor:

```
public static int getTerritoryValue(Territory terry)
{
    int buildingLevelWeight = 10;
    float value = 0;
    value += terry.getBaseMoneyProduction();
    for (Building build : terry.getBuildings())
        value += build.getLevel() * buildingLevelWeight;
    return (int)value;
}
```

- En segundo lugar, realizamos una comprobación idéntica al punto anterior, pero sobre los territorios de los países aliados, de forma que, si quedan tropas libres de la defensa de los territorios propios bajo peligro de conquista, se podrán emplear para defender territorios aliados que estén siendo atacados, del mismo modo que en el caso anterior, se priorizarán los territorios más valiosos.
- En tercer lugar, habrá una probabilidad que dependerá de un parámetro del método que, en caso de darse, provocará que el país mande tropas preventivas de refuerzo a territorios propios que se encuentren en un estado de peligrosidad alta. Se calculará la peligrosidad del territorio y se enviarán tropas preventivas si dicho valor supera un umbral.
- En cuarto lugar, si aun han sobrado tropas defensivas de todos los pasos anteriores, al igual que en el punto previo, se enviarán tropas preventivas a territorios aliados con una peligrosidad mayor al umbral.
- Las decisiones de defensa preventiva están además limitadas por el número de tareas de ese tipo, tanto a nivel general en todo el juego como en tareas de este tipo realizadas por este país, dichas limitaciones existen para evitar un spam gráfico de squads en movimiento en el juego, forzando a los países a sólo defender los objetivos más valiosos, la limitación general está definida por la variable maxAITroopSquads (10) mientras que la limitación de tareas de este tipo realizables por este país se define por la fórmula getMaxTaskOfThisTypeByAI, método que detallaremos en el apartado de gestión de tropas y tareas.

El algoritmo comentado se implementa mediante el siguiente método:

```
private void decideTerritoryToDefend(int preventiveMovementChance,
float dangerousnessThreshold)
{
    // PRIORIDAD 1: Territorios propios bajo conquista
    // Territorios del país ordenados segun valor (prioridad a los mas
    // valiosos)
    ArrayList<Territory> ownerTerritories =
        Utils.getTerritoriesOrderedByValueDesc(
            m_owner.getTerritories());
    for (Territory terry : ownerTerritories)
        if (terry.isBeingConquered())
        {
            TroopTask task = new TroopTask(terry,
                TaskType.TASK_DEFENDING_CONQUEST, -1);
            if (thereAreIdenticalTasksInProgress(task))
                continue;
            executeDefensiveTask(task, 0);
        }

    // PRIORIDAD 2: Apoyo a aliados siendo conquistados
    for (int i = 0; i < m_owner.getAllies().size(); i++)
    {
        Country ally = m_owner.getAllies().get(i);
        ArrayList<Territory> allyTerritories =
            Utils.getTerritoriesOrderedByValueDesc(
                ally.getTerritories());
        for (Territory terry: allyTerritories)
            if (terry.isBeingConquered())
            {
                TroopTask task = new TroopTask(terry,
                    TaskType.TASK_DEFENDING_CONQUEST_ALLY, -1);
                if (thereAreIdenticalTasksInProgress(task))
                    continue;
                executeDefensiveTask(task, 0);
            }
    }

    // Los apoyos preventivos no se dan siempre, probabilidad de ser
    // evaluados
    boolean preventiveActionsActive = true;
    if (Utils.roll_chance(preventiveMovementChance) &&
        preventiveActionsActive)
    {
        // No atacar si hay mas de 10 squads en el juego de IAs
        // atacando
        int maxAITroopSquads = 10;
        if (getAITotalTroopSquadsDefendingPreventive() >
            maxAITroopSquads)
            return;

        // PRIORIDAD 3: Envio de tropas preventivas a los territorios
        // mas peligrosos propios
        int maxTasksOfThisType = Formulas.getMaxTasksOfThisTypeByAI(
            TaskType.TASK_DEFENDING_PREVENTIVE,
            getEffectiveType());
    }
}
```

```

if (getNumberOfTasksByTypeToDifferentTargets (
    TaskType.TASK_DEFENDING_PREVENTIVE) <
        maxTasksOfThisType)
{
    TreeMap<Float, Territory> terriesDanger = new
        TreeMap<>(Collections.reverseOrder());
    for (Territory terry :
        Utils.getTerritoriesOrderedByValueDesc(
            m_owner.getTerritories()))
    {
        TroopTask task = new TroopTask(terry,
            TaskType.TASK_DEFENDING_PREVENTIVE, 0);
        if (thereAreIdenticalTasksInProgress(task))
            continue;

        float dangerouness = Formulas.getDangerouness(terry);

        // Solo si la peligrosidad supera el umbral
        if (dangerouness > dangerounessThreshold)
            terriesDanger.put(dangerouness, terry);
    }
    for (Map.Entry<Float, Territory> entry :
        terriesDanger.entrySet())
    {
        float danger = entry.getKey();
        TroopTask task = new TroopTask(entry.getValue(),
            TaskType.TASK_DEFENDING_PREVENTIVE,
            Formulas.getDurationOfDefencesByDanger
                (danger));
        executeDefensiveTask(task, danger);
    }
}

```

```

// PRIORIDAD 4: Envio de tropas preventivas a los territorios
// mas peligrosos de los países aliados
maxTasksOfThisType = Formulas.getMaxTasksOfThisTypeByAI
    (TaskType.TASK_DEFENDING_PREVENTIVE_ALLY,
        getEffectiveType());
if (getNumberOfTasksByTypeToDifferentTargets
    (TaskType.TASK_DEFENDING_PREVENTIVE_ALLY) <
        maxTasksOfThisType)
{
    TreeMap<Float, Territory> terriesAlliesDanger = new
        TreeMap<>(Collections.reverseOrder());
    for (int i = 0; i < m_owner.getAllies().size(); i++)
    {
        Country ally = m_owner.getAllies().get(i);
        for (Territory terry :
            Utils.getTerritoriesOrderedByValueDesc
                (ally.getTerritories()))
        {
            TroopTask task = new TroopTask(terry,
                TaskType.TASK_DEFENDING_PREVENTIVE_ALLY, 0);
            if (thereAreIdenticalTasksInProgress(task))
                continue;

```

```

        float dangerouness =
            Formulas.getDangerouness(terry);

        // Solo si la peligrosidad supera el umbral
        if (dangerouness > dangerounessThreshold)
            terriesAlliesDanger.put(dangerouness, terry);
    }
}
for (Map.Entry<Float, Territory> entry :
    terriesAlliesDanger.entrySet())
{
    float danger = entry.getKey();
    TroopTask task = new TroopTask(entry.getValue(),
        TaskType.TASK_DEFENDING_PREVENTIVE_ALLY,
        Formulas.getDurationOfDefencesByDanger
            (danger));
    executeDefensiveTask(task, danger);
}
}
}
}

```

El funcionamiento de las Tasks, así como la implementación del método `executeDefensiveTask` será detallado en el apartado dedicado a la gestión de tropas y tareas. Observamos además que, en la implementación de las defensas preventivas, en primer lugar, elaboramos una lista de territorios que exceden el umbral de peligrosidad y posteriormente lo barremos decrecientemente en función de dicho valor, defendiendo primero los territorios más peligrosos de forma similar a la defensa de conquista, que priorizaba en los territorios más valiosos.

4.4 GESTIÓN OFENSIVA

La gestión ofensiva se realiza de forma casi idéntica a la defensiva, pero con distinto sistema de priorización de territorio a atacar. El algoritmo ofensivo es el siguiente:

- Un parámetro que recibe el método definirá la probabilidad de ejecutar una decisión de ataque, esto reduce la carga computacional al evitar que los países valoren dichas decisiones en cada ciclo.
- Al igual que en el caso defensivo, los máximos squads que puede haber en el juego atacando está fijado a 10, de forma que las IAs no podrán tener más de dichos squads en movimiento para realizar tareas ofensivas, evitando así el spam de ataques. También existe una limitación de las máximas tareas de este tipo que el país puede tener activas, definidas por el mismo método que en el caso defensivo: `getMaxTaskOfThisTypeByAI`.
- Si el poder de ataque total del país no supera un umbral, no realizará ataques, esto previene que el país envíe pequeñas cantidades de tropas que no serán capaces de impactar en la lealtad, dicho umbral está definido por el siguiente método y depende del tipo de IA:

```
public static int getAttackPowerThreshold(AIType type)
{
    // Referencia, AP de 1000 asaltos
    int reffValue =
        Game.getUnitByType(UnitType.UNIT_INFANTRY_ASSAULT)
            .getAttack() * 1000;

    int APThreshold = 0;
    switch (type)
    {
        case AI_IMPERIALIST:
            APThreshold = (int)(reffValue * 0.8f);
            break;
        case AI_AGGRESSIVE:
            APThreshold = (int)(reffValue * 0.4f);
            break;
        case AI_DEFENSIVE:
            APThreshold = (int)(reffValue * 1.2f);
            break;
        case AI_PEACEFUL:
            APThreshold = (int)(reffValue * 2.0f);
            break;
    }
    return APThreshold;
}
```

- Ahora detallaremos el algoritmo de decisión de objetivo:
 - El país nunca atacará territorios propios ni aliados.
 - El país no atacará territorios de países con los que no esté actualmente en guerra si la distancia a ese territorio supera un umbral de distancia que actualmente está establecido en 100 puntos, distancia que corresponde aproximadamente a la distancia entre Madrid y París.
 - Para los territorios que superen estas condiciones básicas calculará un peso y lo comparará con un umbral, si el peso no supera el umbral será ignorado.
 - De los territorios cuyo peso supere el umbral, se atacarán en orden de prioridad, priorizando en aquellos que tengan un peso mayor.

El cálculo de pesos es el corazón del algoritmo de decisión de objetivo ofensivo, el correcto calibrado de la expresión que determina el peso de cada territorio tendrá un gran impacto en el juego. A continuación, detallamos los factores que intervienen en ella y su impacto sobre la misma. Si el valor de los factores excede los intervalos definidos se establecerán forzosamente en el límite más cercano a su valor.

- Variables directas:
 - Valor del territorio: El valor del territorio objetivo. A mayor valor, mayor peso.
 - Distancia: La distancia entre el atacante y el objetivo, la distancia estará comprendida entre 50 y 500. A mayor distancia, menor peso.
 - Diplomacia: La relación diplomática entre el atacante y el objetivo, comprendida entre 1 y 100. A mayor diplomacia, menor peso.
 - Posición diplomática del objetivo: Variable que representa como se posiciona el país objetivo diplomáticamente, cuantos más aliados tenga y mayor capacidad militar tengan sus aliados, mejor posicionado estará, y peor estará cuantos más enemigos tenga y con mayor capacidad militar. Su valor estará comprendido entre 0.1 y 5. A mayor posición, menor peso.
 - Ejemplo: Tiene un enemigo con 20k de tropas iguales, tiene dos aliados con 10k de tropas iguales, posición = $1 * 20k / (2 * 10k) = 1$.
 - Relación militar: Relación numérica que relaciona la capacidad militar del atacante y el objetivo, cuanto mayor sea la diferencia de tropas entre el atacante y el territorio objetivo, mayor será este valor. Comprendido entre 0.1 y 5. A mayor relación militar, mayor peso.

- Otros factores:

- Si el territorio es propiedad del jugador, el peso del territorio para ser atacado aumentará en un 50%, se prevé escalar este factor con la dificultad elegida por el jugador, que no está implementada.
- Si el territorio objetivo pertenece a un país con el que el atacante se encuentra en guerra, el peso del territorio para ser atacado aumentará un 300%.

Cálculo de pesos:

La siguiente tabla sintetiza todo lo anterior y muestra el impacto que tiene cada variable en el cálculo, así como su peso.

		Pesos de cada parámetro			
	Relación proporcional	IA Imperialista	IA Agresiva	IA Defensiva	IA Pacífica
Distancia	Inversa	3	2.5	3	3
Diplomacia	Inversa	1	0.9	1	1.5
Posición Diplomática	Inversa	1	0.9	1	1
Relación Militar	Directa	1	1.1	2	1
Valor del territorio	Directa	1	1	1	1
¿Es jugador?	Si es jugador, el peso aumenta un 50%.				
¿Están en guerra?	Si están en guerra, el peso aumenta un 300%.				

El peso final se calcula con la siguiente expresión que depende de todos los factores anteriores y sus pesos:

$$\begin{aligned}
 \text{peso} = & \text{valor}_{\text{territorio}} \cdot \frac{\text{diplomacia}_{\text{neutral}}(50)^{\text{peso}_{\text{diplomacia}}}}{\text{diplomacia}} \\
 & \cdot \text{relacion}_{\text{militar}}^{\text{peso}_{\text{rel_militar}}} \cdot \frac{1}{\text{posicion}_{\text{diplo}}}^{\text{peso}_{\text{pos_diplo}}} \cdot \frac{1}{\text{dist}}^{\text{peso}_{\text{dist}}} \\
 & \cdot \text{factor}_{\text{jugador}} \cdot \text{factor}_{\text{guerra}} \cdot 1000
 \end{aligned}$$

El siguiente método implementa el cálculo de pesos tal y como se ha descrito:

```
public static float getTerritoryWeightForCountry(float dist, float
territoryValue, float diplo, float militaryRelation, float
targetDiplomaticPosition, boolean isPlayer, AIType type)
{
    // Limites a valores
    dist = Math.max(dist, 50);
    dist = Math.min(dist, 500);
    diplo = Math.max(diplo, 1);
    diplo = Math.min(diplo, 100);
    targetDiplomaticPosition = Math.max(targetDiplomaticPosition,
                                         0.1f);
    targetDiplomaticPosition = Math.min(targetDiplomaticPosition, 5);
    militaryRelation = Math.max(militaryRelation, 0.1f);
    militaryRelation = Math.min(militaryRelation, 5);

    // Constante multiplicativa, para escalar el valor y no tener
    // tanto decimal
    float constant = 1000f;

    // Factor de player, incrementa el peso de ser atacado en un 100%
    float playerFactor = isPlayer ? 1.5f : 1f;

    // Factor de hostilidad (si es hostil, peso aumenta un 300%)
    float hostileFactor = diplo < Game.m_warDiplomacy ? 3.0f : 1f;

    // Pesos
    float distWeight = 3.0f;
    float diploWeight = 1.0f;
    float militaryWeight = 1.0f;
    float diplomaticPosWeight = 1.0f;
    switch(type)
    {
        case AI_IMPERIALIST:
            distWeight = 3.0f;
            diploWeight = 1.0f;
            militaryWeight = 1.0f;
            diplomaticPosWeight = 1.0f;
            break;
        case AI_AGGRESSIVE:
            distWeight = 2.5f;
            diploWeight = 0.9f;
            militaryWeight = 1.1f;
            diplomaticPosWeight = 0.9f;
            break;
        case AI_DEFENSIVE:
            distWeight = 3.0f;
            diploWeight = 1.0f;
            militaryWeight = 2.0f;
            diplomaticPosWeight = 1.0f;
            break;
        case AI_PEACEFUL:
            distWeight = 3.0f;
            diploWeight = 1.5f;
            militaryWeight = 1.0f;
            diplomaticPosWeight = 1.0f;
            break;
    }
}
```

```

// Expresion
float output = (terryValue) *
    (float)Math.pow(Game.m_neutralDiplomacy / diplo, diploWeight) *
    (float)Math.pow(militaryRelation, militaryWeight) *
    (float)Math.pow(1 / targetDiplomaticPosition,
        diplomaticPosWeight) * (float)Math.pow(1 / dist, distWeight) *
    hostileFactor * playerFactor * constant;
return output;
}

```

El cálculo del umbral de peso (peso mínimo para valorar la opción) se calcula mediante el siguiente método. Se puede observar que se calcula mediante un escalado de un peso de referencia obtenido de la función anterior:

```

public static float getTerritoryWeightThreshold(AIType type)
{
    // Valor de referencia: Territorio a 100 de distancia, con 50 de
    // produccion, diplomacia de 30, al que
    // duplico en tropas, con el mismo numero de aliados que de
    // enemigos y que no es jugador
    float reffValue = getTerritoryWeightForCountry(100, 50, 30, 2, 1,
                                                    false, type);

    float weightThreshold = 0;
    switch(type)
    {
        case AI_IMPERIALIST:
            weightThreshold = reffValue * 1f;
            break;
        case AI_AGGRESSIVE:
            weightThreshold = reffValue * 0.5f;
            break;
        case AI_DEFENSIVE:
            weightThreshold = reffValue * 1.2f;
            break;
        case AI_PEACEFUL:
            weightThreshold = reffValue * 3;
            break;
    }
    return weightThreshold;
}

```

El siguiente método muestra la implementación del algoritmo de decisión de objetivo ofensivo que se ha descrito anteriormente con todos los parámetros que se han expuesto:

```
private void decideTerritoryToAttack(int attackExecutionChance)
{
    // Probabilidad de atacar segun IA
    if (!Utils.roll_chance(attackExecutionChance))
        return;

    // No atacar si hay mas de 10 squads en el juego de IAs atacando
    int maxAITroopSquads = 10;
    if (getAITotalTroopSquadsAttacking() > maxAITroopSquads)
        return;

    // Maximas tareas de este tipo que esta IA puede tener activa
    int maxTasksOfThisType = Formulas.getMaxTasksOfThisTypeByAI
        (TaskType.TASK_ATTACKING_COUNTRY,
         getEffectiveType());
    if (getNumberOfTasksByTypeToDifferentTargets
        (TaskType.TASK_ATTACKING_COUNTRY) >= maxTasksOfThisType)
        return;

    // Umbral de actuacion, si el peso no esta por encima de este
    // valor, no actuaremos
    float weightThreshold =
        Formulas.getTerritoryWeightThreshold(getEffectiveType());

    // Maxima distancia para atacar a objetivos no hostiles
    float maxDistToStartWar = 100;

    // Umbral de capacidad militar, no atacara a nadie si tiene menos
    // de cierta cantidad de AP
    int APThreshold =
        Formulas.getAttackPowerThreshold(getEffectiveType());

    // Capacidad militar de este país
    float countryTotalAP = m_owner.getTotalAttackPower();
    // Si la capacidad esta por debajo del umbral, no atacará.
    if (countryTotalAP <= APThreshold)
        return;

    // Lista de territorios
    TreeMap<Float, Territory> territoriesToAttackList = new
        TreeMap<>(Collections.reverseOrder());
    for (Country country : Game.m_countryList.values())
    {
        // Nunca atacar territorios propios
        if (country.equals(m_owner))
            continue;

        // Nunca atacar territorios de países aliados
        if (country.isAlliedWith(m_owner))
            continue;
    }
}
```

```

// Diplomacia conmigo
float diplo = m_owner.getDiplomacyValueWithCountry(country);

// Diferencia de capacidad militar
...cubrimos división por 0.
float militaryRelation = countryTotalAP /
    Math.max(country.getTotalDefensivePower(), 1);

// Posicionamiento diplomatico del objetivo: Relacion de
    numero de aliados * tropas / numero de enemigos
* tropas
// Ejemplo: Tiene un enemigo con 20k de tropas iguales, tiene
    dos aliados con 10k de tropas iguales, posicion = 1 *
    20k / (2*10k) = 1.
// Representa la relacion entre posibles tropas que apoyen al
    país objetivo con respecto a posibles tropas que
    ataquen al país objetivo.
float targetDiplomaticPosition =
    country.getDiplomaticPosition();

for (Territory terry : country.getTerritories())
{
    // Si el país esta siendo atacado por un jugador que no
    soy yo, ni es un aliado mio, no atacar
    if (terry.isBeingAttackedByThirdParty())
        continue;

    // Distancia entre el base mio y este otro territorio
    float dist = Utils.calculateEuclideanDistance
        (m_owner.getBaseTerritory().getTerritoryCenter(),
        terry.getTerritoryCenter());

    // Si no es hostil (deberiamos empezar guerra)
    if (!terry.getOwner().isHostileWith(m_owner))
        if (dist > maxDistToStartWar) // ...Y supera el umbral
            de empezar guerra, abortar
            continue;

    // Calculamos peso
    float weight = Formulas.getTerritoryWeightForCountry(dist,
        terry.getValue(), diplo, militaryRelation,
        targetDiplomaticPosition, terry.getOwner().equals(
        Game.m_player), getEffectiveType());

    // No add a la lista si el peso no supera el umbral
    if (weight < weightThreshold)
        continue;

    // Add a la lista
    territoriesToAttackList.put(weight, terry);
}
}

```

```

// Ejecutamos tareas si es posible
for (Territory terry : territoriesToAttackList.values())
{
    TroopTask task = new TroopTask(terry,
                                   TaskType.TASK_ATTACKING_COUNTRY, -1);
    if (thereAreIdenticalTasksInProgress(task))
        continue;
    executeOffenssiveTask(task);
}
}

```

La lógica acumula en una lista todos aquellos territorios atacables: aquellos que cumplen todas las condiciones impuestas, incluida que su peso exceda el umbral. Posteriormente intentará atacar a los países de la lista en orden de peso decreciente. La ejecución de la tarea ofensiva se detallará más adelante.

4.5 GESTIÓN DE TROPAS Y TAREAS

En este apartado vamos a describir como gestionan las IAs sus tropas, como las reparten para realizar distintas tareas y como eligen que tropas emplear para cada tarea. Para ello necesitaremos definir la clase TroopTask:

```

public class TroopTask
{
    private Territory m_targetTerritory;
    private TaskType m_type;
    private int m_taskDuration;
    private int m_counter;
    (...)
}

```

Observando las variables podemos comprender rápidamente el funcionamiento de este objeto, una TroopTask se asignará a un grupo de tropas que tendrán dicha tarea y que se definirá por un tipo, una duración máxima y un territorio sobre el que recae la tarea. Los métodos de la clase se han obviado en el fragmento ya que son simples getters y setters.

Detallando de forma más precisa el funcionamiento de las TroopTasks:

- Una TroopTask se define para un determinado grupo de tropas, a las que se les asigna un tipo de tarea, en un territorio concreto y con una duración máxima en ciclos o días de juego.
- Una tarea terminará cuando venza su tiempo máximo u ocurran eventos específicos que pongan fin a su ejecución.
- Existen tareas que poseen un tiempo infinito (valor -1) y que por tanto sólo terminarán bajo evento específico.
- Las tareas poseen niveles de prioridad que se definen para cada IA, una nueva tarea puede reemplazar a una tarea anterior si el nivel de prioridad de la nueva tarea es superior o igual al de la anterior.

La siguiente tabla muestra un resumen de las tareas que existen y todos sus parámetros relevantes:

				Prioridad (IA)			
	Inicio	Tiempo	Evento final	Imp	Agr	Def	Pac
TASK_NONE	No hay tarea.	-1	Se establece cualquier otra tarea.	0	0	0	0
TASK_BATTLING	Cuando se realiza una ejecución de batalla en la que dichas tropas intervienen.	2	Ninguno.	7	9	6	1
TASK_DEFENDING_CONQUEST	Cuando se marcan las tropas para defender un territorio propio bajo conquista.	-1	Cuando el territorio a defender tiene la lealtad al 100% de nuevo.	10	8	10	10
TASK_DEFENDING_CONQUEST_ALLY	Cuando se marcan las tropas para defender un territorio aliado bajo conquista.	-1	Cuando el territorio a defender tiene la lealtad al 100% de nuevo.	9	7	9	9
TASK_DEFENDING_PREVENTIVE	Cuando se marcan las tropas para defender preventivamente un territorio propio.	Danger*	Ninguno.	6	2	8	8
TASK_DEFENDING_PREVENTIVE_ALLY	Cuando se marcan las tropas para defender preventivamente un territorio aliado.	Danger*	Ninguno.	5	1	7	7
TASK_ATTACKING_COUNTRY	Cuando se marcan las tropas para atacar un territorio enemigo hasta conquistarlo	-1	Cuando el territorio a conquistar ya ha sido conquistador por el atacante.	8	10	2	2

TASK_RETURNING_BASE	Cuando se marcan las tropas para regresar a un territorio propio.	-1	Cuando las tropas llegan al territorio propio y se acuartelan en él.	1	1	1	1
TASK_PREVENT_DESPAWN	Cuando se marcan tropas para que permanezcan sin acuartelar en un territorio durante unos turnos.	5	Ninguno	0	0	0	0

*La duración de la defensa depende de la peligrosidad del territorio a defender. El método que devuelve la duración de las tareas de defensa preventiva y que depende de la peligrosidad es el siguiente:

```
public static int getDurationOfDefencesByDanger(float danger)
{
    int output = (int) (danger * 1);
    return output;
}
```

El siguiente ejemplo muestra la asignación de una tarea a las tropas de un TroopSquad:

```
TroopTask task = new TroopTask(baseTerry,
TaskType.TASK_RETURNING_BASE, -1);
squad.setTask(task);
```

Donde el constructor de la TroopTask recibe los siguientes parámetros:

```
TroopTask(Territory, TaskType, int duration)
```

La finalización de ejecución de tareas se ejecuta mediante el método updateTask, método que se llama desde el update del juego tanto para los arrays de tropas acuarteladas como para los squads:

```
// Update del territorio
public void update()
{
    (...)
    updateTerritoryTroopsTasks();
    updateTroopArrays();
    updateValue();
}
private void updateTerritoryTroopsTasks()
{
    m_territoryTroops.updateTask();
    m_deployedTroops.updateTask();
}
```

```

// Update game
public void updateGame()
{
    (...)
    // Update squads
    updateSquads(true);
    (...)
}

public static void updateSquads(boolean updateTasks)
{
    (...)
    for (int i = 0; i < m_troopSquads.size(); i++)
    {
        TroopSquad squad = m_troopSquads.get(i);

        // Update task
        if (updateTasks)
            if (!squad.getOwner().equals(m_player)) // solo a las de
                                                    las IAs
                squad.updateTask();
    }
    (...)
}

```

El propio método `updateTask` verifica tanto el vencimiento del temporizador asignado a la Task como el cumplimiento de eventos que fuercen la finalización de la tarea, por tanto, dichas verificaciones se realizan en cada ciclo de juego:

```

public void updateTask()
{
    // No hay task
    if (m_task.getType() == TaskType.TASK_NONE)
        return;

    // Comprobacion de final de tarea por tiempo
    if (m_task.hasEnded())
        m_task.clear();
    else
        m_task.increaseCounter();

    // Finalizacion artificial de tasks (debido a eventos)
    switch (m_task.getType())
    {
        case TASK_DEFENDING_CONQUEST_ALLY:
        case TASK_DEFENDING_CONQUEST:
            if (m_task.getTargetTerritory() != null)
                if (m_task.getTargetTerritory().getLoyalty() >= 100)
                    // Si el territorio ya esta a salvo, esperamos 3 turnos
                    // mas antes de retirar tropas
                    m_task.setTaskDuration(3);
                else // El territorio vuelve a no estar a salvo,
                    // duracion ilimitada
                    m_task.setTaskDuration(-1);
    }
}

```

```

        if (m_task.getTargetTerritory() != null) // Si el
            territorio ya no es de mi aliado ni mio.
            if (!m_task.getTargetTerritory().getOwner()
                .isAlliedWith(getOwner()) &&
                    !m_task.getTargetTerritory().
                        getOwner().equals(getOwner()))
                m_task.clear();
            break;
        case TASK_ATTACKING_COUNTRY:
            if (m_task.getTargetTerritory() != null)
                if (m_task.getTargetTerritory()
                    .getOwner().equals(getOwner())) // Si el
                                                            territorio ya es mio
                    m_task.clear();
            break;
        case TASK_NONE:
        case TASK_BATTLING:
        case TASK_RETURNING_BASE:
        case TASK_DEFENDING_PREVENTIVE_ALLY:
        case TASK_PREVENT_DESPAWN:
        case TASK_DEFENDING_PREVENTIVE:
            break;
    }
}

```

El método hasEnded que se llama para verificar el vencimiento del contador deberá tener en cuenta también las tareas de duración infinitas:

```

public boolean hasEnded()
{
    // Duracion infinita, nunca termina por tiempo
    if (m_taskDuration == -1)
        return false;
    else
        return m_counter >= m_taskDuration;
}

```

Observamos que la tarea RETURNING_BASE no se finaliza bajo evento en el método anterior, la verificación de finalización del evento para este caso se realiza desde la propia IA.

La implementación de la Task de batallas es necesaria para evitar que unas tropas que se encuentran luchando abandonen la batalla a medias por una tarea que podría tener una prioridad inferior, la implementación se realiza del siguiente modo: En cada ejecución de batalla se establece de nuevo la tarea de batalla con un contador a 2 segundos, de forma que dicho contador no vencerá nunca mientras la batalla se ejecute en cada ciclo, y una vez haya terminado, tras dos ciclos, las tropas volverán a estar libre, se hace de este modo para evitar tener que acumular todas las squads que participaban en una batalla para limpiarles la tarea al finalizar la misma, dejando que el propio sistema de temporización se encargue. El siguiente fragmento de código muestra a lo que nos referimos:

```
public void executeBattle(ArrayList<TroopArray> attackers,
ArrayList<TroopArray> defenders, float combatPct)
{
    (...)
    for (TroopArray attackerTroops : attackers)
    {
        // Tarea
        // Small hack, dura 2 turnos, pero en cada batalla se
        // refresca, por lo que la task durara un turno mas que la
        // batalla
        TroopTask task = new TroopTask(m_battleTerritory,
                                         TaskType.TASK_BATTLING, 2);
        if (attackerTroops.canChangeTask(task))
            attackerTroops.setTask(task);

        (...)
    }

    for (TroopArray defenderTroops : defenders)
    {
        // Tarea
        // Small hack, dura 3 turnos, pero en cada batalla se
        // refresca, por lo que la task durara un turno mas que la
        // batalla
        TroopTask task = new TroopTask(m_battleTerritory,
                                         TaskType.TASK_BATTLING, 2);
        if (defenderTroops.canChangeTask(task))
            defenderTroops.setTask(task);

        (...)
    }
}
```

Por otra parte, la definición de las prioridades de cada tarea se ejecuta en la siguiente función:

```
// Cuanto mas alto mas prioritario
public int getTaskPriority(TroopTask task)
{
    int output = 0;
    switch(m_ai.getEffectiveType())
    {
        case AI_IMPERIALIST:
            switch (task.getType())
            {
                case TASK_PREVENT_DESPAWN:
                case TASK_NONE:
                    output = 0;
                    break;
                case TASK_RETURNING_BASE:
                    output = 1;
                    break;
                case TASK_BATTLING:
                    output = 7;
                    break;
                (...)
            }
            break;
        case AI_AGGRESSIVE:
            switch (task.getType())
            {
                case TASK_PREVENT_DESPAWN:
                case TASK_NONE:
                    output = 0;
                    break;
                case TASK_RETURNING_BASE:
                    output = 1;
                    break;
                (...)
            }
            break;
        (...)
    }
    return output;
}
```

Ejecución de tareas ofensivas y defensivas

Una vez que hemos expuesto detalladamente el funcionamiento del sistema de tareas vamos a pasar a explicar cómo lo utilizan los algoritmos de envío de tropas defensivas y ofensivas. En los apartados previos vimos cómo se tomaba la decisión de elegir un territorio a atacar o defender en función del cálculo de pesos y prioridades, pero una vez decidido se llamaba a los métodos *executeOffensiveTask* y *executeDeffensiveTask* cuyo funcionamiento no podíamos describir hasta ahora y que se encargan de buscar a las tropas adecuadas para realizar estas tareas y las envían.

```

private void executeDefensiveTask(TroopTask task, float danger)
{
    (...)
    // Obtenemos todos los squads que deberiamos enviar
    ArrayList<TroopSquad> squadsToSend = getSquadsToSendToTask(task,
                                                                territoryToDefend, 0, danger);
    for (TroopSquad squad : squadsToSend)
    {
        // Esta squad ya tiene esta tarea, ignoramos
        if (squad.hasTask(task))
            continue;

        // Add la squad si no existia y la enviamos
        Vector2 target = territoryToDefend.getRandomPoint();
        if (Game.hasSquad(squad))
            squad.sendSquad(target, false);
        else
            Game.addSquad(squad).sendSquad(target, false);

        // Declaramos la guerra a los atacantes
        for (TroopArray attackers :
            territoryToDefend.getAttackerTroopGroups())
        {
            Country attacker = attackers.getOwner();
            if (!m_owner.isHostileWith(attacker))
            {
                DiplomaticAction war = Game.getDiplomaticActionByType(
                    DiplomaticActionType.TYPE_DECLARE_WAR);
                war.tryExecuteAction(m_owner, attacker, false);
            }
        }

        // Tarea, defendiendo territorio
        squad.setTask(task.clone());

        // Log debug IA
        Log.d("IA Debug", m_owner.getName() + " has sent a squad to
            defend " + territoryToDefend.getName());
    }
}

```

```

private void executeOffensiveTask(TroopTask task)
{
    (...)
    ArrayList<TroopSquad> squadsToSend = getSquadsToSendToTask(task,
        territoryToAttack, Formulas.getAttackPowerThreshold
                                (getEffectiveType()), 0);

    for (TroopSquad squad : squadsToSend)
    {
        // Esta squad ya tiene esta tarea, ignoramos
        if (squad.hasTask(task))
            continue;

        // Add la squad si no existia y la enviamos
        Vector2 target = territoryToAttack.getRandomPoint();
        if (Game.hasSquad(squad))
            squad.sendSquad(target, false);
        else
            Game.addSquad(squad).sendSquad(target, false);

        // Declaramos guerra si no eran ya enemigos
        if (!m_owner.isHostileWith(territoryToAttack.getOwner()))
        {
            DiplomaticAction war = Game.getDiplomaticActionByType(
                DiplomaticActionType.TYPE_DECLARE_WAR);
            war.tryExecuteAction(m_owner, territoryToAttack.getOwner(),
                                true);
        }

        // Tarea, atacando territorio
        squad.setTask(task.clone());

        // Log debug IA
        Log.d("IA Debug", m_owner.getName() + " has sent a squad to
            attack " + territoryToAttack.getName());
    }
}

```

En ambos métodos observamos una lógica idéntica:

- Obtenemos los grupos de tropas a enviar con el método `getSquadsToSendToTask`.
- Si la squad existe como tal en el mapa, la enviamos al territorio objetivo, si no existe, es que el grupo de tropas está acuartelado en algún territorio, creamos una squad en el territorio y la enviamos al objetivo.
- Si estamos defendiendo un territorio, declararemos la guerra a sus atacantes, si por el contrario estamos atacando un territorio, declararemos la guerra al objetivo.
- Establecemos la tarea de la squad que hemos enviado.

En ambos casos hemos obtenido una lista de squads a enviar mediante una llamada a `getSquadsToSendToTask`, dicho método ejecuta decide qué tropas conviene enviar a dicha tarea y cuantas, representando una parte muy importante de la capacidad decisora de la inteligencia artificial. Su algoritmo funciona del siguiente modo:

1.- Decisión de número de tropas a enviar: Se establece un número máximo y mínimo de tropas (cuantificado como poder de ataque en tareas ofensivas y poder de defensa en tareas defensivas) que se debe enviar, si tras la decisión, el valor asociado con las tropas elegidas no supera el valor mínimo, no se enviarán. Tampoco se enviarán nunca más tropas del valor máximo.

Defensa de conquistas propias	Porcentaje del poder de ataque de las tropas invasoras.	
Tipo de IA	Mínimo	Máximo
Imperialista	50%	200%
Agresiva	50%	100%
Defensiva	10%	300%
Pacífica	80%	120%
Defensa de conquistas aliadas	Porcentaje del poder de ataque de las tropas invasoras.	
Tipo de IA	Mínimo	Máximo
Imperialista	30%	150%
Agresiva	30%	70%
Defensiva	7%	250%
Pacífica	50%	80%
Defensa preventiva propia	Porcentaje del AP equivalente a la peligrosidad ($AP_{eq} = 1000 \cdot \text{peligrosidad}$)*	
Tipo de IA	Mínimo	Máximo
Imperialista	15%	70%
Agresiva	15%	35%
Defensiva	3.5%	125%
Pacífica	25%	40%
Defensa preventiva aliada	Porcentaje del AP equivalente a la peligrosidad ($AP_{eq} = 1000 \cdot \text{peligrosidad}$)*	
Tipo de IA	Mínimo	Máximo
Imperialista	10%	50%
Agresiva	10%	25%
Defensiva	2.5%	100%
Pacífica	15%	30%

Ataque a objetivo	Porcentaje de la defensa total de las tropas defensoras.	
Tipo de IA	Mínimo	Máximo
Imperialista	150%	300%
Agresiva	80%	200%
Defensiva	200%	500%
Pacífica	300%	600%

*El método que implementa el cálculo de AP equivalente a la peligrosidad del territorio es el siguiente:

```
public static int getEquivalentAPFromDanger(float danger)
{
    int output = (int) (danger * 1000);
    return output;
}
```

2.- Asignación de pesos a las tropas: Se realiza un barrido de todas las tropas que posee el país, tanto en squads como acuarteladas en sus territorios y se calcula un peso para cada una, cuanto mayor sea el peso, mayor utilidad tienen dichas tropas para ser enviadas a realizar esa tarea en concreto. El peso se calcula con la siguiente función:

$$weight = taskDiff + \frac{1000}{dist} + reqPct \cdot 20$$

La variable *taskDiff* representa la diferencia de prioridad entre la tarea que estén haciendo las tropas ahora y la que se quiere establecer, de forma que cuanto mayor diferencia de prioridades haya mayor peso tendrán las tropas para ser enviadas. Si la diferencia fuera negativa, es decir, que la tarea actual tuviera más prioridad que la nueva, hay una excepción en el código que implementa estos cálculos que impediría que dichas tropas se valorasen como opción.

La variable *dist* representa la distancia a la que están las tropas del objetivo y evidentemente es inversamente proporcional, está limitada inferiormente por 50, de modo que cualquier valor inferior a este será redondeado a 50. Por otra parte, la variable *reqPct* cuantifica el valor que tiene esta tropa para la tarea encomendada, calculando que porcentaje de sus valores es ofensivo y cual defensivo, al método se le pasará el adecuado en cada caso y esto actuará con proporcionalidad directa sobre el cálculo del peso.

El método implementa el cálculo anterior se encuentra en la clase Formulas y es el siguiente:

```
public static float getTroopsWeight(int taskDiff, float dist, float reqPct)
{
    if (dist < 50)
        dist = 50;

    float output = taskDiff + 1000 / dist + reqPct * 20;
    return output;
}
```

3.- Acumulación y verificación final: Una vez que ya tenemos una lista de tropas ordenada decrecientemente por pesos, comenzaremos a barrerla acumulando tropas, si dicha acumulación alcanza el valor máximo de tropas a enviar, se abandonará la iteración y se enviarán estas tropas. Si tras haber barrido la lista entera no hemos acumulado un valor de tropas superior al mínimo, se abortará la operación y no se enviará ninguna tropa.

El método que implementa todo lo expuesto es el siguiente:

```
private ArrayList<TroopSquad> getSquadsToSendToTask(TroopTask task,
Territory targetTerritory, int threshold, float danger)
{
    // 1.- Decisión de número de tropas a enviar: Simplificamos por
    // equivalencia con la tabla
    int targetTroopPoints = 0;
    float maxPctToSend = 0;
    float minPctToSend = 0;
    boolean off = true;
    switch(task.getType())
    {
        case TASK_DEFENDING_CONQUEST:
            off = false;
            targetTroopPoints = targetTerritory.getTotalAttackersAP();
            switch(getEffectiveType())
            {
                case AI_IMPERIALIST:
                    maxPctToSend = 2.0f;
                    minPctToSend = 0.5f;
                    break;
                case AI_AGGRESSIVE:
                    maxPctToSend = 1.0f;
                    minPctToSend = 0.5f;
                    break;
                (...)
            }
            break;
    }
}
```

```

case TASK_DEFENDING_CONQUEST_ALLY:
    off = false;
    targetTroopPoints = targetTerritory.getTotalAttackersAP();
    switch(getEffectiveType())
    {
        case AI_IMPERIALIST:
            maxPctToSend = 1.5f;
            minPctToSend = 0.3f;
            break;
        (...)
    }
    break;
case TASK_DEFENDING_PREVENTIVE:
    off = false;
    targetTroopPoints =
        Formulas.getEquivalentAPFromDanger(danger);
    switch(getEffectiveType())
    {
        case AI_IMPERIALIST:
            maxPctToSend = 0.7f;
            minPctToSend = 0.15f;
            break;
        (...)
    }
    break;
case TASK_DEFENDING_PREVENTIVE_ALLY:
    off = false;
    targetTroopPoints =
        Formulas.getEquivalentAPFromDanger(danger);
    switch(getEffectiveType())
    {
        case AI_IMPERIALIST:
            maxPctToSend = 0.5f;
            minPctToSend = 0.10f;
            break;
        (...)
    }
    break;
case TASK_ATTACKING_COUNTRY:
    off = true;
    targetTroopPoints = targetTerritory.getTotalDefendersDP();
    switch(getEffectiveType())
    {
        case AI_IMPERIALIST:
            maxPctToSend = 3.0f;
            minPctToSend = 1.5f;
            break;
        (...)
    }
    break;
    // Estas tareas no envian tropas a ninguna parte
case TASK_NONE:
case TASK_RETURNING_BASE:
case TASK_PREVENT_DESPAWN:
case TASK_BATTLING:
    return null;
}

```

```

// 2.- Asignación de pesos a las tropas
// Primero tropas de squads
TreeMap<Float, MovableTroops> movableTroopsMap = new
    TreeMap<>(Collections.reverseOrder());
for (int i = 0; i < m_owner.getTroopSquads().size(); i++)
{
    TroopSquad squad = m_owner.getTroopSquads().get(i);
    if (squad == null)
        continue;

    // Esta squad ya tiene esta tarea
    if (squad.hasTask(task))
        continue;

    // Las milicias no se tendran en cuenta, no se pueden mover
    if (squad.getType() == SquadType.SQUAD_MILITIA)
        continue;

    // Si la squad ya esta en el territorio no hay que hacer nada
    // con ella
    if (squad.getTerritory() != null)
        if (squad.getTerritory().equals(targetTerritory))
            continue;

    int taskDiff = squad.getTaskPriorityDiff(task);
    if (taskDiff > 0)
    {
        float dist = Utils.calculateEuclideanDistance(
            targetTerritory.getTerritoryCenter(),
            squad.getPosition());

        float reqPct = 0;
        if (off) // Necesitamos tropas ofensivas
            reqPct = squad.getOffensivePct();
        else // Necesitamos tropas defensivas
            reqPct = squad.getDefensivePct(targetTerritory);

        // Peso
        float weight = Formulas.getTroopsWeight(taskDiff, dist,
            reqPct);

        // Almacenamos en el mapa de tropas usables con su peso
        movableTroopsMap.put(weight, new
            MovableTroops(MovableType.TYPE_SQUAD, squad,
                null, null, null));
    }
}

```

```

// Despues tropas acuarteladas en territorios
for (Territory myTerry: m_owner.getTerritories())
{
    // Si las tropas ya estan en el territorio atacado, no tenemos
    // que hacer nada con ellas
    if (myTerry.equals(targetTerritory))
        continue;

    TroopArray terryTroops = myTerry.getTerritoryTroops();
    if (terryTroops.hasTroops())
    {
        int taskDiff = terryTroops.getTaskPriorityDiff(task);
        if (taskDiff > 0)
            for (Troops troop : terryTroops.getTroops())
            {
                if (troop.getNumber() == 0)
                    continue;

                float dist = Utils.calculateEuclideanDistance(
                    targetTerritory.getTerritoryCenter(),
                    myTerry.getRandomPointFromCenter(
                        WithRandomRadius(0)));

                float reqPct = 0;
                if (off)
                    reqPct = troop.getOffensivePct();
                else
                    reqPct =
                        troop.getDefensivePct(targetTerritory);

                // Peso
                float weight = Formulas.getTroopsWeight(taskDiff,
                                                            dist, reqPct);

                // Almacenamos en el mapa de tropas usables con tu
                // peso
                movableTroopsMap.put(weight, new
                    MovableTroops(MovableType.TYPE_TROOP,
                                    null, troop, myTerry, terryTroops));
            }
    }

    (...)
    // Hariamos lo mismo que en el caso anterior para las deployed
    // troops.

```

```

// 3.- Acumulación y verificación final
// Ya tenemos una lista de tropas ordenadas por peso, calculamos
// ahora los squads que necesitamos generar
int pointsSum = 0;
final ArrayList<TroopSquad> squadsToSend = new ArrayList<>();
HashMap<Territory, ArrayList<Troops>> troopsToSend = new
                                                                    HashMap<>();

for (Map.Entry<Float, MovableTroops> entry :
      movableTroopsMap.entrySet())
{
    // Si la defensa/ataque excede el máximo
    if (pointsSum >= (int)(targetTroopPoints * maxPctToSend))
        break;

    // Movable
    MovableTroops movable = entry.getValue();
    switch (movable.getType())
    {
        case TYPE_SQUAD:
            // Obtenemos la squad
            TroopSquad squad = movable.getSquad();

            // Defensa que aporta
            pointsSum += squad.getTotalAttackPower();

            // Squad a la lista de enviados
            squadsToSend.add(squad);
            break;
        case TYPE_TROOP:
            // Obtenemos la tropa
            Troops troop = movable.getTroops();

            // Defensa que aportan
            pointsSum += troop.getTotalAttackPower(100);

            // Tiene que agrupar estas tropas segun territorio
            Territory originTerry = movable.getOriginTerritory();

            // Add tropas a lista de tropas a squadear + enviar
            if (troopsToSend.containsKey(originTerry)) // Si ya lo
                                                         tenia
                troopsToSend.get(originTerry).add(troop.clone());
            else // Si no lo tenia, creamos nueva entrada
            {
                ArrayList<Troops> troopArray = new ArrayList<>();
                troopArray.add(troop.clone());
                troopsToSend.put(originTerry, troopArray);
            }

            // Eliminamos tropa del array origen
            movable.getOriginalArray().subtract(troop);
            break;
    }
}

```

```

// Una vez hemos terminado el barrido tenemos el array de tropas
// locales listo para construir las squads asociadas y lanzarlas
// Generamos squads y los enviamos
for (Map.Entry<Territory, ArrayList<Troops>> entry :
    troopsToSend.entrySet())
{
    // Construimos squad y la add al juego
    TroopArray group = new TroopArray(entry.getValue());
    Vector2 pos = entry.getKey().getRandomPoint();

    // Add nuevo squad al array de pendientes
    squadsToSend.add(new TroopSquad(group, m_owner, pos.x, pos.y,
        SquadType.SQUAD_LAND));
}

// Si no hemos superado los umbrales, vaciamos el array, no hay
// nada para enviar
if (pointsSum < threshold || pointsSum < (int)(targetTroopPoints *
    minPctToSend))
    squadsToSend.clear();
return squadsToSend;
}

```

Consideraciones adicionales a tener en cuenta en la implementación:

- Como acumulamos tropas tanto de squads ya existentes como de listas acuarteladas en territorios, se ha creado una clase adicional llamada movableTroops que almacena el peso de la tropa, el array de tropas que necesitamos referenciar y el tipo al que pertenece para aplicarle una lógica diferenciada en el momento de acumularlas. Esta clase resulta en cierto modo trivial y solo tiene fines organizativos para esta parte del código fuente, sin implicaciones adicionales en el diseño de la aplicación. Las ventajas de esta estructuración de datos son fácilmente apreciables en el código.
- El método siempre genera objetos squad, incluso para tropas acuarteladas, siendo el método superior quien debe verificar que dichos squads existen en el mapa.

Gestión de acumulación de tareas

En los métodos de decisión de territorios a atacar o defender pudimos ver como se realiza una comprobación del número de tareas de este tipo que tiene vigentes el país para evitar que realice demasiadas tareas del mismo tipo. Dicha comprobación no podía ser explicada en ese momento ya que no se había introducido el concepto de tareas. Existe un método que comprueba el numero de tareas de un tipo que cada país posee, basta con comparar el output del método con un valor base bien diseñado para evitar que aparezca un spam de tareas similares en el mapa.

El número máximo de tareas del mismo tipo depende de la tarea y el tipo de IA y se muestran en la siguiente tabla, las tareas que no aparezcan en la misma no se encuentran limitadas.

Tipo de IA	Ataque	Defensa preventiva propia	Defensa preventiva aliada
Imperialista	3	2	2
Agresiva	2	1	1
Defensiva	1	3	3
Pacífica	1	1	1

El siguiente método de la clase Formulas implementa la tabla:

```
public static int getMaxTasksOfThisTypeByAI(TaskType task, AIType type)
{
    int maxTasks = 0;
    switch (type)
    {
        case AI_IMPERIALIST:
            switch (task)
            {
                case TASK_ATTACKING_COUNTRY:
                    maxTasks = 3;
                    break;
                case TASK_DEFENDING_PREVENTIVE:
                    maxTasks = 2;
                    break;
                (...)
            }
            break;
    }
}
```

```

        case AI_AGGRESSIVE:
            switch (task)
            {
                case TASK_ATTACKING_COUNTRY:
                    maxTasks = 2;
                    break;
                (...)
            }
            break;
        case AI_DEFENSIVE:
            (...)
        case AI_PEACEFUL:
            (...)
    }
    return maxTasks;
}

```

Gestión de squads

En los métodos de updateAI hemos podido observar que se realiza una llamada al método clearUntaskedSquads como se observa en el código de updateDeffensiveAI

```

private void updateDefensiveAI()
{
    (...)
    // Decide a que territorio deberia enviar tropas ofensivas
    decideTerritoryToAttack(5);

    // Limpiamos squads
    clearUntaskedSquads();
}

```

Dicha llamada se realiza desde los updates de todos los tipos de IA, la función de ese método es retirar del mapa todos los squads que el país tenga desocupados, evitando así que las tropas en squads se acumulen en el juego:

```
private void clearUntaskedSquads()
{
    for (int i = 0; i < m_owner.getTroopSquads().size(); i++)
    {
        TroopSquad squad = m_owner.getTroopSquads().get(i);
        if (squad != null)
        {
            // las milicias las ignoramos
            if (squad.getType() == SquadType.SQUAD_MILITIA)
                continue;

            if (squad.getTask().getType() == TaskType.TASK_NONE)
                // Squad ociosa
            {
                // Si no tiene territorio
                if (squad.getTerritory() == null)
                    sendToBaseTerritory(squad);
                else // Si esta en algun territorio
                {
                    // Ese territorio es mio
                    if (squad.getTerritory().getOwner()
                        .equals(squad.getOwner()))

                        squad.retractSquadToTerritory
                            (squad.getTerritory(), false);
                    else // No es mio
                        sendToBaseTerritory(squad);
                }
            }

            // Si es squad de regreso
            if (squad.getTask().getType() ==
                TaskType.TASK_RETURNING_BASE) // Squad ociosa
            {
                // Si no tiene territorio
                if (squad.getTerritory() != null)
                    if (squad.getTerritory().equals(squad.getTask()
                        .getTargetTerritory()))
                    {
                        TroopTask task = new TroopTask(null,
                            TaskType.TASK_NONE, -1);

                        squad.setTask(task);

                        squad.retractSquadToTerritory
                            (squad.getTerritory(), false);
                    }
            }
        }
    }
}
```

El método realiza las siguientes operaciones sobre todas las squads desplazables que tenga el país:

- Si la squad tiene tarea *TASK_NONE*, entenderá que la squad está desocupada. En este escenario acuartelará las tropas de la squad en el territorio en el que estén si se encuentran sobre algún territorio del que el país sea propietario. En caso contrario, mandará las tropas al territorio en el que se crearon mediante una llamada a *sendToBaseTerritory* que además les asignará una nueva tarea: *RETURNING_BASE*.
- Si la squad tiene tarea *RETURNING_BASE*, establecida en otro momento por la lógica del punto anterior, comprobará que la tropa ya se encuentre en el territorio objetivo de dicha tarea, y en el momento en el que esto ocurra, cancelará dicha tarea y acuartelará definitivamente las tropas en el territorio.

4.6 IMPLEMENTACIÓN DEL PATHFINDING

En este apartado vamos a exponer que lógica sigue el cálculo de rutas de las tropas en el mapa, para después indagar en su implementación. El pathing en dominations se apoya sobre el sistema de nodos, que como ya hemos detallado en apartados anteriores, define una serie de puntos que ocupan posiciones en el mapa y que están interconectados entre sí, cada conexión de cada nodo tiene un peso diferente, y el algoritmo de pathing deberá hallar una ruta que minimice el coste del camino final entre el nodo en el que se encuentran las tropas hasta el nodo al que se las desea mover, y deberá hacerlo de la forma más rápida y precisa posible. Para lograr estos objetivos hemos empleado el algoritmo A* indexado, que recibe una lista de nodos, las conexiones entre ellos y sus costes, y devuelve un grafo que determina el camino que minimiza el coste.

Algoritmo A*

El algoritmo A* es algoritmo heurístico de búsqueda que encuentra un camino a través de un espacio discreto definido por puntos o nodos. El algoritmo se basa en la selección iterativa del nodo más cercano a la posición actual que se encuentra más cerca del destino. El algoritmo sólo garantizará el hallazgo del camino más corto si la función heurística cumple dos objetivos:

- Consistencia: La heurística empleada debe ser consistente, esto es, respetar el principio de desigualdad de Minkowski, que establece que la suma de los lados de un triángulo siempre será mayor que la del lado restante.
- Admisibilidad: La heurística nunca sobreestimaré la distancia al objetivo.

El cumplimiento de estas dos condiciones garantiza que, si un camino existe, será encontrado y además será el más corto posible.

El algoritmo recibe como input: una función objetivo, una función de expansión y una función heurística, y devuelve una lista ordenada de nodos que representan el camino.

La función heurística se emplea para estimar en todo momento la distancia desde cualquier nodo al objetivo, con vistas a poder estimar de forma recurrente a qué distancia total estamos del objetivo. En el nuestro caso la heurística empleada es la Euclidiana, que calcula las distancias en base a una generalización del teorema de Pitágoras; esta heurística además cumple las condiciones de consistencia y admisibilidad.

El algoritmo usa además una lista abierta que representa la lista de nodos no expandidos, y una lista cerrada, que contiene los nodos ya expandidos. Esta diferenciación es necesaria para prevenir bucles infinitos que pueden resultar de iterar indefinidamente los mismos caminos.

El algoritmo añade el nodo inicial a la lista de abiertos y lo expande, abriendo cada nodo con los que se encuentra conectado y calculando un valor $F = \text{coste} + \text{distancia heurística(euclidiana)}$ y cerrando el nodo inicial, el algoritmo realizará este proceso de forma recursiva con los nodos abiertos de forma que comprobará las alternativas posibles y descartará los caminos subóptimos. El siguiente fragmento de pseudocódigo ejemplifica lo expuesto:

Algorithm 1: A^*

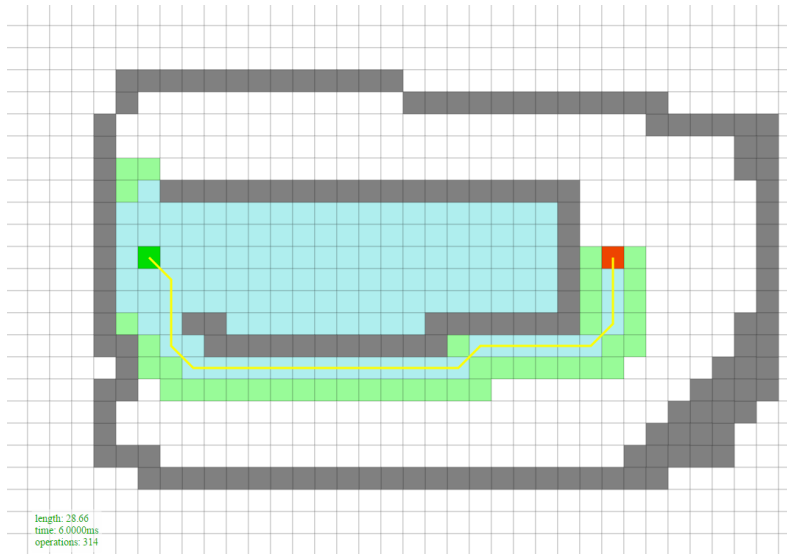
Input: $start, goal(n), h(n), expand(n)$
Output: $path$

```

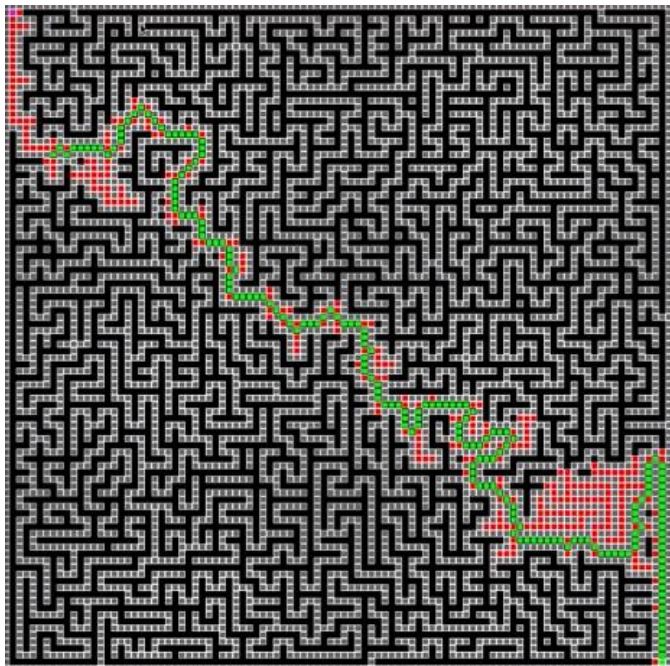
1 if  $goal(start) = true$  then return  $makePath(start)$ 
2
3  $open \leftarrow start$ 
4  $closed \leftarrow \emptyset$ 
5 while  $open \neq \emptyset$  do
6    $sort(open)$ 
7    $n \leftarrow open.pop()$ 
8    $kids \leftarrow expand(n)$ 
9   forall the  $kid \in kids$  do
10     $kid.f \leftarrow (n.g + 1) + h(kid)$ 
11    if  $goal(kid) = true$  then return  $makePath(kid)$ 
12    if  $kid \cap closed = \emptyset$  then  $open \leftarrow kid$ 
13   $closed \leftarrow n$ 
14 return  $\emptyset$ 

```

Las dos siguientes imágenes muestran ejemplos del funcionamiento del algoritmo:



- Código de colores:
 - El nodo rojo representa el destino.
 - El nodo verde oscuro representa el inicio.
 - Los nodos grises representan paredes o nodos sin conectar, no pueden atravesarse.
 - Los nodos azules son nodos expandidos.
 - Los nodos verdes representan nodos aun abiertos.
 - Los nodos blancos representan nodos ni abiertos ni expandidos.



En esta imagen podemos apreciar en verde los nodos abiertos y en rojo los expandidos, que en principio han sido descartados.

Complejidad del algoritmo

En el peor de los casos el algoritmo puede degenerar en un algoritmo de búsqueda en anchura con una complejidad de b^d donde b es el factor de ramificado y d es la profundidad de meta. La complejidad del algoritmo es sensible por tanto a factores como:

- Ordenación de los datos y frecuencia de empleo.
- Estructuras de datos empleadas para soportar la ejecución.
- Complejidad de la función heurística.
- Si se realizan detecciones duplicadas.
- Si aceptamos o no soluciones subóptimas.

El algoritmo no puede, por su diseño, devolver soluciones parciales, por lo que si los inputs que emplea el algoritmo son modificados durante el proceso de búsqueda no se garantiza una solución que cumpla las condiciones de optimalidad.

El Algoritmo A* indexado se trata de una modificación del algoritmo A* primitivo que se basa en el indexado de los nodos, ahorrando de ese modo tener que recorrer las dos listas para encontrar nodos, permitiendo eliminar así la lista de nodos cerrados y acelerar considerablemente la ejecución, asumiendo por otra parte un incremento en el consumo de memoria de la aplicación al tener que almacenar el indexado. La lista de nodos abiertos debe mantenerse ya que es necesaria para obtener los elementos con el mínimo coste.

Implementación del pathing en la aplicación

En el apartado destinado al sistema de nodos ya explicamos con detalle el diseño del sistema de nodos, no obstante, vamos a hacer un pequeño repaso.

Durante la inicialización se generan cerca de 150.000 nodos uniformemente distribuidos por el mapa, después cada nodo establece conexiones con sus 9 nodos vecinos como muestra la siguiente imagen:

-1, 1	0, 1	1, 1
-1, 0	0, 0	1, 0
-1, -1	0, -1	1, -1

El algoritmo de pathfinding tendrá en cuenta estas conexiones y sus costes que son definidas dinámicamente por una función de costes que se muestra a continuación:

```
@Override
public float getCost()
{
    // Posicion inicial
    int basePosX = m_fromNode.getPositionX();
    int basePosY = m_fromNode.getPositionY();
    int originCountryId = m_mapPoints[basePosX][basePosY] + 128;

    // Posicion final
    int destPosX = m_toNode.getPositionX();
    int destPosY = m_toNode.getPositionY();
    int destCountryId = m_mapPoints[destPosX][destPosY] + 128;

    // Coste
    return getCostForConnectionBetweenCountries(originCountryId,
                                                destCountryId);
}
```

Dicha función obtiene la ID del territorio que se encuentra bajo los nodos origen y destino en la obtención del coste de la conexión entre ellos, para ello lee el valor almacenado en el array de puntos del mapa y en base a esa ID obtiene el coste con el siguiente método:

```
public int getCostForConnectionBetweenCountries(int originTerryId, int
destTerryId)
{
    // Si es linea recta, todos los caminos tienen el mismo peso
    // incuestionablemente
    if (m_rectLine)
        return 1;

    // Variable de salida
    int cost = 0;

    // SALTO DE FRONTERAS
    float atenuation = 1;
    if (!m_squad.isAffectedByTerrain())
        atenuation = 0.1f;
    // Si saltamos de tierra a agua o de agua a tierra
    if ((originTerryId != 0 && destTerryId == 0) || (originTerryId ==
0 && destTerryId != 0))
        cost = (int)(50 * atenuation);
    // Vamos por agua
    else if (originTerryId == 0 && destTerryId == 0)
        cost = (int)(30 * atenuation);
    // Si saltamos una frontera cualquiera
    else if (originTerryId != destTerryId)
        cost = (int)(20 * atenuation);
}
```

```

// MOVIMIENTO POR PAISES Y DIPLOMACIA
// Si nos movemos por el pais del ejecutor
else if (m_squad.getOwner().equals(
    Game.getTerritoryById(destTerryId).getOwner()))
    cost = 10;
else
{
    // Leemos paises
    Territory terryOrigin = Game.getTerritoryById(originTerryId);
    Territory terryDest = Game.getTerritoryById(destTerryId);
    if (terryOrigin != null && terryDest != null)
    {
        if (terryOrigin.getOwner().isAlliedWith(
            terryDest.getOwner()))
            cost = 10;
        else if (terryOrigin.getOwner().isFriendlyWith(
            terryDest.getOwner()))
            cost = 50;
        else if (terryOrigin.getOwner().isNeutralWith(
            terryDest.getOwner()))
            cost = 60;
        else if (terryOrigin.getOwner().isHostileWith(
            terryDest.getOwner()))
            cost = 40;
    }
}

// Valor minimo
if (cost < 1)
    cost = 1;
return cost;
}

```

Observamos que el algoritmo de cálculo de costes sigue la siguiente lógica:

- El método tiene un parámetro para realizar movimientos en línea recta, para ello establece el mismo coste en todas las conexiones, esto provocará que la heurística euclidiana encontrará el camino óptimo en la línea recta.
- Costes dependientes del terreno: todos los costes siguientes se reducen a un 10% del valor si las tropas no son afectadas por el terreno (aviones, por ejemplo)
 - Si la conexión entre los dos nodos cambia de medio (tierra/agua) tendrá un coste de 50.
 - Si las tropas se desplazan por mar, tendrá un coste de 30.
 - Si la conexión entre los dos nodos cambia de país (atraviesa frontera) tendrá un coste de 20.

- Si no atraviesa un salto de fronteras, de tipo de terreno ni va por agua, entraremos a valorar el coste de atravesar determinados países en función de la diplomacia.
 - Si el territorio es propio o de algún aliado, el coste de atravesarlo será 10.
 - Si el dueño del territorio es amistoso, el coste de atravesarlo será 50.
 - Si el dueño del territorio es neutral, el coste de atravesarlo será 60.
 - Si el dueño del territorio está en guerra, el coste de atravesarlo será 40.

El coste de atravesar el mar es mayor debido a que las tropas afectadas por el terreno reducirán su velocidad durante la travesía. El coste aumentado al atravesar países amistosos y neutrales se debe a que mover tropas sobre ellos tiene un impacto negativo sobre la diplomacia de ambos países, cosa que no ocurre en situación de alianza, por eso el coste en ese caso es tan bajo como el de atravesar un territorio propio. Una situación similar ocurre al atravesar territorios hostiles, el coste diplomático es ya más o menos irrelevante, lo que se traduce en un coste inferior que cuando atraviesas países amistosos o neutrales.

El correcto diseño de los valores de estos costes tiene un impacto relevante en el desarrollo del juego, por lo que deberán revisarse y ajustarse de forma periódica.

Movimientos de tropas

Los cálculos del pathfinding se ejecutan en otro thread ya que resultan pesados y provocarían pequeños freezes que afectarían radicalmente a la experiencia de juego, es por esto que cuando un país desea realizar un movimiento de tropas. Es por esto que el hilo dedicado al pathing se ejecuta sobre una cola FIFO a la que se añadirá la nueva petición y que será servida lo antes posible, todo esto será expuesto con mayor detalle en el apartado de threading.

Cuando el thread de pathing ejecuta una tarea pendiente de pathfinding llamará al siguiente método, que es el directamente encargado de realizar el envío de tropas, a continuación, procederemos a comentar por separado cada parte del método:

```
private void executePathing(final TroopSquad squad, Vector2 dest,
boolean rectLine)
{
    (...)
    // Distancia a partir de la cual se procesara en linea recta y
    fin
    float distThreshold = (Game.windowW / Game.m_nodeSize) *
                                0.15f;

    // Posiciones de los nodos mas cercanos
    int startX = Math.round((squad.getX() + Game.windowW / 2) /
                                Game.m_nodeSize);
    int startY = Math.round((squad.getY() + Game.windowH / 2) /
                                Game.m_nodeSize);
    int endX = Math.round((dest.x + Game.windowW / 2) /
                                Game.m_nodeSize);
    int endY = Math.round((dest.y + Game.windowH / 2) /
                                Game.m_nodeSize);
    float totalDist = Utils.calculateEuclideanDistance(new
        Vector2(startX, startY), new Vector2(endX, endY));
    if (totalDist > distThreshold)
        rectLine = true;

    // Limpiamos ultima trayectoria que hubiera y su display
    (...)
    Game.m_path.clear();

    SequenceAction seqAction = new SequenceAction();

    // Leemos nodos
    DynamicGraph.Node startNode = Game.m_graph.getNode(startX,
                                startY);
    DynamicGraph.Node endNode = Game.m_graph.getNode(endX, endY);

    // Calculamos path optimo
    Game.m_graph.searchPath(startNode, endNode,
        Heuristics.euclidean(), Game.m_path, squad, rectLine);

    (...)
}
```

En el anterior fragmento se realizan las siguientes acciones:

- Se traducen las posiciones de la scene2D al formato nodo para encontrar el nodo origen y nodo destino para el input del algoritmo A* indexed.
- El algoritmo A* indexed recibirá el parámetro rectLine que define su el movimiento debe ser en línea recta, esto ocurrirá tanto si el parámetro del método es true (generalmente cuando el movimiento lo realiza el jugador) como si la distancia del movimiento es demasiado grande, momento en el cual el cálculo sería demasiado pesado incluso ejecutándose en un hilo aparte. Dicha distancia máxima es relativa al tamaño de pantalla.
- Se limpiará la stage que guarda los puntos de la trayectoria de la squad que se dibujan en el mapa
- Se ejecuta el algoritmo A* indexed con la llamada al método `m_graph.searchPath`, que devolverá una lista de nodos en el objeto `m_path` según la lógica expuesta en el apartado anterior.

En esta segunda parte se crearán las actions sobre el actor que representa en el juego a la TroopSquad:

(...)

```
// Creamos acciones asociadas
for (int i = 1; i < Game.m_path.getCount(); i++)
{
    DynamicGraph.Node originNode = Game.m_path.get(i - 1);
    DynamicGraph.Node destNode = Game.m_path.get(i);
    Vector2 originPos = new Vector2(originNode.getPositionX(),
                                    originNode.getPositionY());
    Vector2 destPos = new Vector2(destNode.getPositionX(),
                                  destNode.getPositionY());
    float speed = squad.getSlowestTroopSpeed();

    // Velocidad segun medio
    if (squad.isAffectedByTerrain())
    {
        // Si saltamos de tierra a agua o de agua a tierra
        if ((originNode.getCountryId() != 0 &&
            destNode.getCountryId() == 0) ||
            (originNode.getCountryId() == 0 &&
            destNode.getCountryId() != 0))
            speed /= 5;
        // Si vamos por agua
        else if (originNode.getCountryId() == 0 &&
            destNode.getCountryId() == 0)
            speed /= 3;
        // Si saltamos una frontera
        else if (originNode.getCountryId() !=
            destNode.getCountryId())
            speed /= 2;
    }
}
```

```

float space = Utils.calculateEuclideanDistance(originPos,
                                                destPos) * Game.m_nodeSize;
float time = space / speed;
Action action = Actions.moveTo(destPos.x * Game.m_nodeSize
                               - Game.windowW / 2, destPos.y *
                               Game.m_nodeSize - Game.windowH / 2, time);
seqAction.addAction(action);

// Accion de detectar fin de movimiento
Action completeAction = new Action(){
    (...)

    if (squad.getStagePath().getActors().size > 0)
    {
        // Eliminamos punto del path
        squad.getStagePath().getActors().get(0).remove();

        // Si ya no quedan puntos, hemos llegado.
        if (squad.getStagePath().getActors().size == 0)
        {
            squad.setDestiny(null);
            squad.setMovingInRectLine(false);
        }
    }
    (...)
    seqAction.addAction(completeAction);
}
    (...)

```

En el anterior fragmento se realizan las siguientes acciones:

- Se barren todos los nodos del camino, almacenados en el `m_path`, de ella se extrae el nodo origen y destino.
- Se establece la velocidad del movimiento entre cada pareja de nodos en función del tipo de terreno que se atraviesa y la velocidad de la tropa más lenta de la `squad`.
- Se crea una acción de movimiento entre esos dos nodos cuyo tiempo de ejecución se calcula en función de la distancia entre ellos y la velocidad calculada en el punto anterior.
- Las acciones de movimiento se encapsulan en una acción de secuencia, de forma que se garantice su ejecución en orden.
- Añadimos entre cada acción de movimiento una acción personalizada que se encargará de borrar el dibujo del punto que se acaba de atravesar, y de esta forma ir limpiando la trayectoria en el mapa conforme se recorra.

```

(...)

// Punto del path
if (squad.getOwner().equals(Game.m_player))
{
    Image point = new Image(Game.getSpriteByName("pointImage"));

    // Es el ultimo
    if (i == Game.m_path.getCount() - 1)
    {
        point = new Image(Game.getSpriteByName("squadPinMove"));
        point.setSize(squad.getWidth() / 2, squad.getHeight() / 2);
    }
    else
    {
        point.setSize(Game.m_nodeSize, Game.m_nodeSize);
        point.setPosition(destPos.x * Game.m_nodeSize - Game.windowW / 2 -
            point.getWidth() / 2, destPos.y * Game.m_nodeSize - Game.windowH /
                2);

        point.setColor(1, 1, 1, 0.5f);
        final Image finalPoint = point;
        Gdx.app.postRunnable(new Runnable() {
            @Override
            public void run()
            {
                {
                    squad.getStagePath().addActor(finalPoint);
                }
            }
        });

        }
        squad.clearActions();
        squad.addAction(seqAction);

        // Nuevo destino
        squad.setDestiny(dest);
        squad.setMovingInRectLine(rectLine);
    }
}

```

En este último fragmento del método se crea el gráfico que se dibujará en ese punto de la trayectoria, con una imagen diferente para el punto final y cierta transparencia estética. Por último, se limpian las acciones previas que pudiera tener la squad y se le añade la acción secuencial de movimientos que acabamos de crear. Además de esto fijaremos su destino y un boolean que represente si dicho movimiento es o no en línea recta. La razón de guardar estas dos variables es almacenar el estado de movimiento por si ocurre una guarda/carga durante el mismo, permitiendo que se reanude a posteriori.

Una vez realizadas todas estas gestiones, desde el método render se procesarán las squads y sus acciones para realizar directamente el movimiento como muestran los siguientes fragmentos pertenecientes al método render:

```
// Draw Path
for (int i = 0; i < m_troopSquads.size(); i++)
{
    TroopSquad squad = m_troopSquads.get(i);
    if (squad == null)
        continue;
    if (squad.getStagePath() == null)
        continue;

    squad.getStagePath().act(Gdx.graphics.getDeltaTime());
    squad.getStagePath().draw();
}
(...)
// Solo mover si no esta parado
if (m_gameSpeed != GameSpeed.SPEED_STOP)
    stageSquads.act(Gdx.graphics.getDeltaTime() *
        ((float)m_secondsPerDayHigh / (float)m_secondsPerDay));
    // Pequeño hack para mover las tropas a la velocidad adecuada
    // segun speed.
stageSquads.draw();
```

En primer lugar, barremos las squads para dibujar sus trayectorias si es que las tienen, cosa que en principio solo ocurre con las squads de jugadores en movimiento. Posteriormente, realizará una llamada al método act sobre las squads, lo que provocará que ejecuten sus acciones: el parámetro que se le pasa al método act representa el tiempo entre cada llamada y tiene un efecto directo sobre la velocidad a la que se desplazan las squads, dicha propiedad se ha explotado para implementar el efecto que las distintas velocidades del juego tienen sobre la velocidad final de las tropas.

4.7 FUTURO DE LA IA EN DOMINATIONS

Como hemos podido observar en todo el diseño, los algoritmos de inteligencia artificial articulan las decisiones en base al cálculo de cuantificaciones y pesos de las distintas decisiones posibles y la elección de aquellas que maximizan dicho peso. El sistema se comportará de forma deseada en base a esta premisa siempre que el cálculo del peso esté diseñado de forma que encaje con el óptimo del juego, esto requerirá un minucioso y continuo análisis para ajustar las contribuciones de los distintos factores de cada ecuación. Otras decisiones como las económicas requieren sin embargo un aumento de la complejidad de los algoritmos de decisión que se traduzca en elecciones más precisas e inteligentes por parte de las IAs.

Se espera implementar en el futuro un sistema de aprendizaje por refuerzo que permita a las distintas IAs el ajuste dinámico de los pesos que influyen en cada variable de las ecuaciones que determinan las decisiones, de modo que, tras jugar muchas partidas, las IAs estén cada vez mejor adaptadas al juego y realicen jugadas más inteligentes que supongan un reto hasta para los jugadores más veteranos.

5.- RENDIMIENTO

5.1 MULTITHREADING

Las diferentes tareas de la aplicación están divididas en hilos por los siguientes motivos:

- Al contar el sistema en el que se ejecuta la aplicación con un procesador multinúcleo, el diseño permite la ejecución de tareas en paralelo.
- Aliviamos carga del thread de renderizado, aumentando los FPS de la aplicación dando lugar a una experiencia de juego más satisfactoria.
- Una excepción provocada en uno de los hilos causará la interrupción de su ejecución, pero el resto del juego seguirá funcionando normalmente, dando opción, por ejemplo, a guardar partida si la interrupción del hilo forzase una interrupción completa de la aplicación.
- Ventajas organizativas, los distintos hilos ejecutan tareas similares.

La principal desventaja del sistema multihilo es que incrementa la probabilidad de sufrir excepciones debido a problemas de sincronización, impactando negativamente en la estabilidad de la aplicación.

La siguiente tabla muestra los hilos con los que contamos y su función:

Nombre	Función
GL Thread	Renderizado de pantalla, lectura de inputs de usuario, gestión de UI.
Update Thread	Ciclos de juego: update de países, territorios, lealtades, conquistas, batallas...
IA Thread	Decisiones de inteligencia artificial para cada país.
Saving Thread	Ejecución de I/O de guardado de partida en fichero externo.
Pathing Thread	Cálculos de pathfinding para los movimientos de tropas.

Update Thread

Este hilo está dedicado a la ejecución del `updateGame` que ocurre en cada ciclo o día de juego. Se inicializa durante el arranque de la aplicación en el método `initializeThreads`

```
public void initializeThreads()
{
    (...)
    // Thread de updates
    m_updateThread = new UpdateThread(this);
    m_updateThread.start();
    m_updateThread.setName("Update Thread");
    (...)
}
```

Donde la clase `UpdateThread` es la siguiente:

```
public class UpdateThread extends Thread
{
    private Game m_game;

    public UpdateThread(Game game)
    {
        m_game = game;
    }

    @Override
    public void run()
    {
        while (true)
        {
            synchronized (this)
            {
                try
                {
                    wait();
                }
                catch (InterruptedException e)
                {
                    continue;
                }
            }
            if (m_game != null)
                m_game.updateGame();
        }
    }

    public void updateGame()
    {
        // Arrancamos
        synchronized (this)
        {
            notify();
        }
    }
}
```

El funcionamiento del sistema es fácilmente inferible en base al código: El hilo recibe el objeto Game como parámetro, ya que necesitará llamar al método updateGame que no es un método estático. Por otra parte, el thread ejecutará infinitamente el código contenido en el while(true) que hay en el método run().

Tras cada ejecución de un ciclo del while la iteración se detendrá en virtud del wait que hay en su inicio, esperando a reanudarse cuando se realice un evento notify. Resulta fácil inferir que wait y notify se implementan en base a semáforos y que han de llamarse en un contexto sincronizado.

El evento notify se dispara cuando se llama al método updateGame del objeto thread que se almacena dentro de Game como m_updateThread, esto sucede cuando llega el momento de realizar un update del juego:

```
public void render()
{
    (...)
    // Update
    // No actualizar si esta parado
    if (m_secondsPerDay != -1)
    {
        if (m_tickCounter >= m_secondsPerDay)
        {
            // Restablecemos y actualizamos
            m_tickCounter = 0;
            m_updateThread.updateGame();
        }
        else
            m_tickCounter += Gdx.graphics.getDeltaTime();
    }
    break;
}
```

Esta llamada provocará que se realice una iteración del while y cuando termine la actualización se alcanzará de nuevo el wait, pausando el hilo hasta la llegada de un nuevo ciclo de actualización.

IA Thread

El hilo de actualización de IA funciona de forma idéntica al de actualización del juego expuesto anteriormente, primero lo inicializamos:

```
public void initializeThreads()
{
    (...)
    // Thread de update IA
    m_aiThread = new AIThread(m_countryList);
    m_aiThread.start();
    m_aiThread.setName("IA Thread");
}
```

Donde el objeto IAThread es equivalente al UpdateThread, con la única salvedad de que recibe la lista de países del juego, que necesitará para actualizar la IA de cada uno de forma recursiva como se muestra en el código:

```
public class AIThread extends Thread
{
    private AI m_ai;
    private ArrayList<Country> m_countryList;

    public AIThread(HashMap<Integer, Country> countryList)
    {
        m_countryList = new ArrayList<Country>(countryList.values());
    }

    @Override
    public void run()
    {
        while (true)
        {
            synchronized (this)
            {
                try
                {
                    wait();
                }
                catch (InterruptedException e)
                {
                    continue;
                }
            }
            for (Country country : m_countryList)
                country.updateAI();
        }
    }
}
```

```

public void update ()
{
    // Arrancamos
    synchronized (this)
    {
        notify();
    }
}

```

Al igual que en el caso anterior, el thread se reanuda ante la ocurrencia de un evento notify que en este caso se activa desde el método updateGame que se llama desde el thread de update:

```

public void updateGame ()
{
    (...)
    // Update AI
    m_aiThread.update();
    (...)
}

```

Pathing Thread

El thread de pathing tiene la función de realizar los cálculos de pathfinding del algoritmo A* indexado y se implementa mediante una cola de tareas con disciplina FIFO como veremos. La inicialización se realiza del modo habitual:

```

public void initializeThreads()
{
    // Thread de movimiento de tropas
    m_pathingThread = new PathingThread();
    m_pathingThread.start();
    m_pathingThread.setName("Pathing Thread");
    (...)
}

```

```

public class PathingThread extends Thread
{
    private ArrayList<PathingTask> m_pathingTasks;

    @Override
    public void run()
    {
        // Inicializamos variable
        m_pathingTasks = new ArrayList<PathingTask>();

        while (true)
        {
            try
            {
                if (m_pathingTasks.size() > 0)
                {
                    // Ejecutamos tarea 0
                    PathingTask task = m_pathingTasks.get(0);
                    executePathing(task.getSquad(), task.getDest(),
                                   task.isRectLine());

                    // Hemos terminado la ejecucion, eliminamos primer
                                                                elemento
                    m_pathingTasks.remove(0);
                }
            }
            catch (Exception e)
            {
                Log.d("Exception", "Pathing thread exception");
            }
        }

        private void executePathing(final TroopSquad squad, Vector2 dest,
                                     boolean rectLine)
        {
            (...)
            // Obtenemos el path optimo
            Game.m_graph.searchPath(startNode, endNode,
                                     Heuristics.euclidean(), Game.m_path, squad, rectLine);
            (...)
        }

        public void add(TroopSquad squad, Vector2 dest, boolean rectLine)
        {
            m_pathingTasks.add(new PathingTask(squad, dest, rectLine));
        }
    }
}

```

Este hilo funciona como vemos de forma muy similar a los anteriores, la única diferencia es que en lugar de reanudar la ejecución del hilo cuando ocurre un evento externo, añadimos elementos a una lista de tareas y el hilo las va sirviendo en orden de llegada.

La adición de elementos a la cola del hilo se realiza desde el juego cada vez que se ejecuta un movimiento de tropas:

```
public void sendSquad(final Vector2 dest, final boolean rectLine)
{
    Game.m_pathingThread.add(this, dest, rectLine);
}
```

El método anterior se llama siempre que se desea enviar una squad a una posición del mapa, dicho movimiento se convertirá una tarea para el hilo de pathfinding que la servirá tan pronto como sea posible.

Saving Thread

El thread de guardado se implementa de forma mucho más simple que los anteriores, ya que ni siquiera requiere de una clase independiente, implementándose directamente sobre el código de guardado del método saveGame:

```
public void saveGame(final boolean closeAfterEnd)
{
    (...)
    Thread saveThread = new Thread(new Runnable()
    {
        public void run()
        {
            // Generacion de strings
            StringBuilder saveString = new StringBuilder();

            // Info general
            SaveType saveTypeGame = SaveType.SAVE_GAME;
            saveString.append(saveTypeGame.toString());
            saveString.append(";");

            (...)
            // Escribimos el texto (cifrado)
            saveGame.writeString(Base64Coder.
                                encodeString(saveString.toString()), false);
            (...)
        }
    });
    saveThread.start();
    saveThread.setName("Saving Thread");
}
```

Podemos apreciar que el código de guardado inicia un thread nuevo que hace la función de hilo de guardado en cada ejecución de guardado. Esta implementación no está normalizada con el resto de diseños de hilos ya que se hizo muy previamente al resto, se espera normalizar su diseño en el futuro.

GL Thread

El hilo de renderizado ejecuta básicamente todas las tareas que no se definan en un hilo independiente, es pues, el hilo por defecto en el que se ejecuta la aplicación y por tanto no requiere más explicaciones: toda tarea que no sea ejecutada por alguno de los hilos anteriores será procesada por este thread.

Problemas de sincronización

La mayor parte de los problemas de sincronización entre hilos se producen en el renderizado, es decir, cuando se intenta renderizar un actor que está siendo escrito desde otro hilo en ese instante, para ayudar a corregir este problema, el framework GDX implementa una forma de posponer la ejecución de un fragmento de código hasta la llamada al siguiente render, de este modo se garantiza la sincronización, ya que los fragmentos comprendidos entre esa estructura nunca se ejecutarán en paralelo a un render. Ese código es el siguiente:

```
Gdx.app.postRunnable(new Runnable()
{
    @Override
    public void run()
    {
        // Insertar codigo sincronizado aquí
    }
});
```

Un ejemplo de lo descrito es lo siguiente:

```
private void executePathing(final TroopSquad squad, Vector2 dest,
boolean rectLine)
{
    (...)
    // Limpiamos ultima trayectoria que hubiera y su display
    Gdx.app.postRunnable(new Runnable()
    {
        @Override
        public void run() {
            squad.getStagePath().clear();
        }
    });
    (...)
}
```

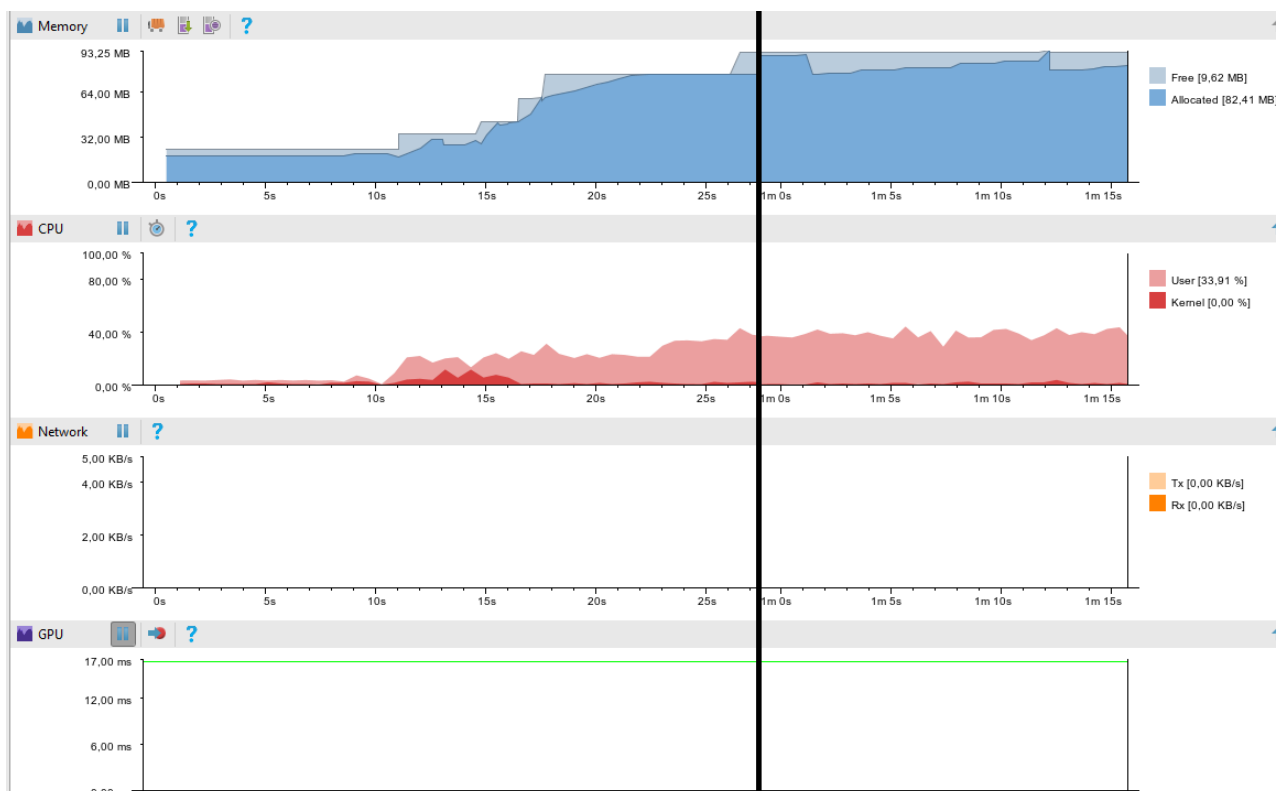
El fragmento anterior corresponde con el método de ejecución de cálculo de pathfinding, en esta parte del método, se limpia la stage que almacena los puntos que se dibujan en la trayectoria de movimiento anterior que pudieran tener las tropas sobre las que ahora se calcula una nueva ruta. Si en este momento el thread OpenGL estuviera tratando de renderizar esta stage, se encontraría ante un problema de sincronización de tipo lectores-escribtores al tratar de leer un objeto que está siendo borrado simultáneamente desde otro hilo.

El problema queda completamente resuelto al cubrir el fragmento con el código expuesto anteriormente, ya que se garantiza de ese modo que la ejecución del mismo se intercale entre dos llamadas a render. Todos los problemas de sincronización que surgen esporádicamente se resuelven rodeando el fragmento que presenta problemas de sincronización y que puede causar eventuales excepciones con dicha estructura.

5.2 PERFILADO: MEMORIA, CPU, RED, GPU

Los siguientes gráficos muestran el consumo de distintos recursos computacionales en la fase de ejecución capturados con el Android Studio en el siguiente dispositivo:

Smartphone empleado:	Samsung Galaxy s5
CPU:	Qualcomm MSM8974AC Snapdragon 801 2.5 GHz
RAM:	2 GB
GPU:	Adreno 330



Respecto a la memoria podemos observar un crecimiento de la memoria desde el segundo 13 aproximadamente hasta el 22, corresponde con toda la inicialización del juego, los parámetros, la carga de imágenes, la generación de nodos en el juego y sus conexiones, y en definitiva todas las llamadas que se realizan en el método initializeGame. Por otra parte, la memoria final consumida es muy aceptable para un Smartphone con 2 GB de memoria RAM, suponiendo menos de un 5% de la memoria total. La mayor parte del consumo de memoria se debe, como es de esperar, al cacheado de los ficheros de imagen y sonido del juego, seguidos por el almacenamiento de las variables float. El almacenamiento de todo el sistema de nodos también tiene un impacto importante en la memoria, ya que se almacenan casi 150.000 nodos, todos con información de 9 conexiones, un byte que representa la ID del territorio que hay bajo ellos y determinados valores de posición.

Respecto al consumo de CPU podemos apreciar un consumo considerablemente alto de la aplicación, llegando a consumir casi un 35% del total de la CPU disponible, en este consumo va incluida también toda la carga computacional del renderizado gráfico. Los siguientes perfilados muestran el consumo de los distintos métodos en cada uno de los hilos que se ejecutan en el juego.

Nota: Existe otro thread dedicado al cálculo de pathing que no se muestra en las capturas de profiling y tampoco lo considero necesario, ya que sus ejecuciones son discretas y rápidas en relación a los demás.

Respecto al consumo de red no hay mucho que decir ya que no se han implementado funciones que la empleen por el momento y así se refleja.

Respecto a la gráfica de GPU se muestra el periodo de actualización de pantalla o tiempo entre renders OpenGL, 17ms de periodo equivale a una frecuencia de 60Hz de actualización, que es el valor estándar en el mundo de los videojuegos, en vista de la estabilidad de dicho valor podemos decir que se trata de una frecuencia de renderizado excelente.

Respecto a la estabilidad de la aplicación podemos asegurar que la aplicación es estable, suceden algunos crashes aleatorios muy de vez en cuando debidos a problemas de sincronización entre threads, no se espera que estos crashes supongan un problema a largo plazo ya que ocurren de forma muy esporádica, se corrigen inmediatamente y el sistema de autoguardado mitiga su efecto.

6.- CAPITALIZACIÓN Y MARKETING

Capitalización

La capitalización de la aplicación se realizará mediante micro transacciones: una evolución de los servicios freemium que permite al usuario comprar ventajas en el juego y desbloquear distintas funcionalidades desde el propio juego.

Este sistema aún no está implementado, pero se diseñará en torno a unas monedas especiales que el usuario podrá comprar con dinero y que podrá intercambiar por las ventajas comentadas.

Las ventajas que el usuario podrá comprar en el juego funcionarán de modo similar a las tecnologías y se aplicarán a todas las partidas que juegue. Algunos ejemplos son:

- +2%/+4%/+6%/+8%/+10% a la producción de dinero diaria.
- +2%/+4%/+6%/+8%/+10% a los valores de tus tropas.
- -5%/-10%/-15% duración de colas.

Podemos observar que las ventajas tienen distintos niveles de mejora, su coste se diseñará de forma exponencial para ralentizar el progreso del jugador a altos niveles, dando lugar a una experiencia más satisfactoria.

Las monedas también se podrán utilizar para desbloquear nuevos modos de juego que se vayan implementando en Dominations, algunos ejemplos podrían ser los siguientes:

- Partida al triple de velocidad.
- Partida donde la lealtad máxima es 30 (conquistas muy rápidas)
- Partida donde se invierte el PIB de los países y los pobres son los ricos.
- Partidas que simulan escenarios concretos: Eventos scriptados de guerras entre distintos países.
- Partida en la que todos los países son inicialmente hostiles entre sí.

La adición de dichos modos de forma progresiva mantendrá vivo el juego y explotará al máximo el modelo geopolítico subyacente para dar lugar a situaciones divertidas y desafiantes.

La aplicación se publicará en la Play Store y aprovechará las implementaciones que proporciona google para monetizar las aplicaciones, en nuestro caso: In-App Billing API.

La API comentada funciona del siguiente modo:

- El usuario elige en la aplicación el producto que quiere comprar.
- La App manda una solicitud al servicio de Google Play, quien gestiona todo el proceso de compra y devuelve a la aplicación una confirmación en caso de éxito.
- La aplicación añade las monedas compradas a la cuenta de usuario del jugador.

Hemos apreciado la necesidad de la existencia de una cuenta de usuario para este sistema, en apartados posteriores veremos que dicha cuenta será empleada también para hacer posible otras implementaciones, como la acumulación de estadísticas de jugador o el juego multijugador.

Dicha cuenta de usuario podrá implementarse en base a la cuenta de Google con la que el usuario ha de identificarse para descargar la aplicación desde la tienda Play Store, para lo cual Google proporciona una API muy cómoda de usar, o podrá ser implementada de forma autónoma y gestionada desde un servidor maestro que será quien realice también las gestiones necesarias para dar soporte al modo multijugador.

La siguiente URL muestra el funcionamiento de la API In-App Billing así como ejemplos de la implementación. Esta parte aun no se ha implementado en Dominations:

https://developer.android.com/google/play/billing/billing_integrate.html

Marketing y publicidad

La aplicación es sólo compatible con Android en su lanzamiento inicial, por ello nos centraremos en la tienda Android: Google Play. El marketing a la aplicación se articulará sobre los siguientes principios:

- Visibilidad y posicionamiento: La visibilidad de la aplicación es muy relevante para que llegue a ser conocida y permitarnos alcanzar una masa crítica y provocar efecto red, que sería muy beneficioso para la captación y retención de clientes. Un buen posicionamiento SEO es clave para la visibilidad de la misma. Para esto invertiremos en páginas de publicidad que nos permitan llegar a más usuarios del público objetivo.
- Imagen y atractivo: Una imagen atractiva de la aplicación, ratings positivos en la tienda google hará más atractiva la aplicación para potenciales clientes.

Para la obtención y mantenimiento de **imagen** se realizarán las siguientes gestiones:

- Elección adecuada del nombre de la aplicación, iconos e imagen de la misma. Esto ya está en principio logrado en el diseño actual.
- Diseño del juego cuidado y con mejoras continuas que permitan la retención del cliente.
- Recompensa de monedas en el juego por votar la aplicación en Google Play.
- Diseño de una web responsive muy simple que posea información básica de la aplicación y links a la tienda de google. Garantizar el adecuado posicionamiento SEO de esta website es muy relevante para que cumpla su cometido. La elección del dominio es también relevante, por lo que hemos elegido el dominio www.dominationsapp.com, que se encuentra libre actualmente.
- Elaboración de un video (tráiler) del juego que pueda ser empleado en todas las plataformas anteriores a modo de muestra, deberá ser representativo del juego y atractivo, con una buena postproducción.
- Creación de páginas en redes sociales asignadas a la aplicación: Facebook, twitter y Snapchat.
- Atención de usuario adecuada y personalizada, respondiendo educadamente a los comentarios críticos que puedan surgir en plataformas públicas como la propia tienda google.

Para la garantizar la **visibilidad** de la aplicación emplearemos las siguientes técnicas para publicitar la aplicación y darla a conocer:

- Servicio App2Top ofrece las siguientes ventajas a un coste relativamente bajo:
 - Servicios gratuitos analíticos del estado de la aplicación, progresión de número de instalaciones, de reviews...
 - Servicio de instalaciones pagadas para mejorar el posicionamiento de la aplicación a un coste de 10 céntimos la descarga.
 - Servicio de voto de 5 estrellas a 20 céntimos por voto.
 - Servicio de review muy positivo de la aplicación a 25 céntimos por review.
- Publicidad oficial en Facebook: El servicio de publicidad en Facebook permite una publicidad dirigida a un perfil muy específico de personas, lo cual tiene un impacto muy positivo en el coste del producto. El objetivo es diseñar un anuncio oficial de Facebook que cumpla las siguientes expectativas:
 - Que se muestre a personas con edades comprendidas entre 14 y 25 años, que constituyen la base de clientes.
 - Que se muestre durante las horas de máxima actividad de estas personas en esta red social, presumiblemente fines de semana por las tardes.
- Publicidad oficial en Snapchat: El servicio de publicidad en Snapchat permite mostrar un anuncio personalizado sobre la aplicación, a un coste de aproximadamente 2 céntimos por visualización, resulta relevante como alternativa publicitaria ya que se trata de un servicio de mensajería utilizado por nuestro público objetivo.
- Publicidad en las redes sociales de la aplicación: Facebook, Snapchat, twitter, concursos, viralización de contenidos a cambio de recompensas...
- Publicidad en sites famosos de valoración de aplicaciones: Conseguir que nuestra aplicación sea comentada en alguna página de aplicaciones como AppAdvice.com o 148apps.com podría tener un considerable impacto en la visibilidad de nuestro producto sin incurrir en ningún coste adicional, logrando la publicitación de forma gratuita en virtud de la calidad del producto.
- Presentación de la aplicación a concursos de aplicaciones: La victoria proporcionaría una gran visibilidad a la aplicación, pero sólo el hecho de concursar ya tendría una contribución positiva.
- Publicidad en blogs o canales de divulgación influyentes.
- Integración de redes sociales en la aplicación: Permitir a los jugadores compartir sus puntuaciones, victorias, etc.

Habremos observado que todas las estrategias de marketing comentadas responden a un objetivo: Lograr la máxima divulgación posible mediante unos recursos económicos absolutamente limitados. El auge de internet y las tecnologías de la información, redes sociales y demás, ha abierto la puerta a la aparición de un nuevo tipo de marketing a un coste muy bajo que permite a aplicaciones como esta darse a conocer si se trabaja duro en ello y se ofrece un producto verdaderamente atractivo para los clientes.

Esperamos desarrollar en un futuro cercano un plan específico basado en lo expuesto para publicitar la aplicación y de ese modo obtener unos costes aproximados que nos permitan valorar la inversión, optimizando así el proceso de publicitación y así posicionar el producto lo más alto posible en el mercado de los juegos móviles.

Copyright y marcas

Para el lanzamiento será imprescindible realizar un análisis cuidadoso de todo el material que interviene en la aplicación en relación a la propiedad intelectual.

La mayoría de material audiovisual que se emplea se ha elegido con cuidado, sin copyright ni derechos de autor asociado, no obstante, podría ser que algunos iconos o elementos sí tuvieran asociada una propiedad intelectual subyacente que no se reflejase en los orígenes de descarga de los mismos, por este motivo deberá realizarse una investigación exhaustiva sobre cada elemento de la aplicación y su estado de protección intelectual.

Todos aquellos elementos sensibles a causar problemas legales de naturaleza IP deberán ser reemplazados por otros que no presenten este tipo de conflictos, ya sean de propiedad pública o de propia creación.

7.- CONCLUSIONES Y PLANES DE FUTURO

Mejoras en IA: Aprendizaje

Hemos observado durante el desarrollo de este apartado que la IA no es más que un conjunto de algoritmos de decisión más o menos complejos, pretendemos mejorar esos algoritmos en el futuro, además de haciéndolos más complejos, dotándolos de una cierta capacidad de aprendizaje.

Dicha implementación se planteará según la lógica del aprendizaje por refuerzo, permitiendo a las IAs acumular datos de las partidas jugadas y sus resultados para modificar dinámicamente los pesos de los parámetros de las ecuaciones que definen las fronteras de decisión. El dinamismo de estos pesos y su convergencia tras la ejecución de miles de partidas permitirá que cada país se comporte de forma diferente a cualquier otro, partiendo de su tipo de personalidad, y tendiendo a un comportamiento óptimo para la maximización de los objetivos que se definan.

Se pretende acumular los datos de las partidas de todos los usuarios para el aprendizaje de las IAs y, a largo plazo, centralizar las decisiones IA en un servidor remoto que garantice un comportamiento síncrono en las partidas de todos los usuarios. La sincronización propuesta es indispensable, cualquier otro escenario daría lugar a comportamientos IA diferentes en función de las partidas de cada usuario, impidiendo el diseño de estrategias de juego consistentes.

Soporte multiplataforma

Como ya hemos comentado en otros apartados, la librería GDX que da soporte a esta aplicación presenta compatibilidad con las siguientes plataformas:

Cross-Platform

A single API to target:

- Windows
- Linux
- Mac OS X
- Android (2.2+)
- BlackBerry
- iOS
- Java Applet (requires JVM to be installed)
- Javascript/WebGL (Chrome, Safari, Opera, Firefox, IE via **Google Chrome Frame**)

Con el potencial que demuestra la compatibilidad del framework parece evidente la oportunidad que se presenta de expandir las plataformas sobre las que se ejecuta el juego y alcanzar una mayor cuota de mercado.

La prioridad es la compatibilidad con iOS, logrando así abarcar todo el mercado de aplicaciones móviles, que es en principio el contexto sobre el que se ha diseñado la aplicación. En segundo lugar, se pretende adaptar la aplicación a Windows, que requerirá además el rediseño de algunas partes del juego en referencia a los inputs de usuario, dando soporte al zoom con rueda de ratón, por ejemplo.

La adaptación a otras plataformas requerirá además rediseñar características internas del juego. Por ejemplo, en PC la aplicación sí podría aceptar la introducción de inputs de teclado, permitiendo al usuario definir el número de tropas que quiere producir o arrastrar a la interfaz. Otro ejemplo de adaptaciones lógicas se puede encontrar en la posibilidad de añadir más información en la interfaz de usuario debido a la existencia de mayores pantallas en otras plataformas.

Soporte multijugador

Se pretende incluir en un futuro cercano el soporte multijugador que permita a distintas personas jugar la misma partida, dando lugar a partidas más interesantes y desafiantes. A continuación, mostramos una lista de ideas sobre dicha implementación:

- Una persona podrá “apuntarse” a distintos modos de juego multijugador que se diseñarán.
- El sistema emparejará a todos los jugadores que estén en espera de cada modo de juego.
- Dichos modos de juego determinarán, entre otras cosas, el máximo número de jugadores por partida.
- Cuando un jugador abandone una partida en curso se asignará una IA al país que controlaba.
- Se implementará un sistema de puntuación y ranking que permita a los usuarios competir entre ellos.
- Se implementará un modo competitivo que emparejará jugadores de un nivel similar y que les permitirá escalar en los rankings públicos.
- El sistema se implementará sobre una arquitectura cliente-servidor en la que toda la lógica de la partida estará centralizada en el servidor remoto, propiedad de los creadores de la aplicación.

Los cambios arquitecturales necesarios para adaptar la aplicación actual al soporte multijugador incluyen los siguientes:

- Implementación de un sistema de sockets que permita la comunicación entre cliente y servidor.
- Implementación de una estructura lógica en la aplicación que la permita comportarse como maestro o como esclavo.
 - Una aplicación en modo esclavo no ejecutará comprobaciones lógicas cíclicas como la ejecución de `updateGame`, en su lugar recibirá la información del servidor y mostrará los eventos gráficamente en el cliente cuando reciba la orden de hacerlo.
 - La aplicación en modo maestro se comportará como hemos descrito en esta aplicación, sólo dando soporte `singleplayer`.
 - El servidor centralizado se diseñará modificando la aplicación en modo maestro, realizará el procesamiento lógico descrito y mandará el resultado del procesamiento a los clientes.
 - Por poner un ejemplo: La producción económica, actualmente el cliente incrementa el dinero de cada país localmente en el ciclo de `update`, en el nuevo diseño, el servidor realizará ese incremento y mandará un paquete a los clientes informando del nuevo dinero de cada país, el cliente no realizará nunca esa comprobación, en su lugar se mantendrá en espera del paquete, y actualizará el dinero cuando lo reciba. Siendo el renderizado y la gestión de inputs de usuario las únicas tareas cuya gestión seguirá dependiendo del cliente.
- Implementación de opcodes que definan el tipo de paquetes que existirán y sus formatos, normalizando así la comunicación de cliente y servidor.
- Modificación de ciertos métodos para permitir el procesamiento multiusuario, algunos fragmentos de lógica podrían suponer que sólo existe un jugador.

La implementación del multijugador en la aplicación resultará indispensable en un mercado cada vez más interconectado, por lo que ésta deberá realizarse a corto plazo si la aplicación quiere lograr una penetración sostenida en el mercado de los videojuegos móviles.

Soporte de idiomas

Se pretende implementar en el futuro traducciones para los idiomas más hablados del mundo, dicha implementación se realizará del siguiente modo:

- Todos los strings que se muestran en el juego serán reemplazados por una referencia a una llamada a un método de una nueva clase: Translate.
- Dicho método devolverá el string correspondiente al idioma en el que nos encontremos, lo leerá de una base de datos en el que los strings estarán tabulados en una relación ID/Idioma.
- Dicha base de datos estará implementada en un fichero externo .dat como el resto de información externa del juego.
- La traducción de cada cadena se realizará usando el inglés como lenguaje base a cada uno de los otros idiomas soportados de forma manual. Se plantea la posibilidad de crear una pequeña aplicación que permita modificar de forma cómoda el fichero .dat para mayor comodidad de los traductores.

Cuentas de usuario: progreso y micro transacciones

Las adiciones comentadas en los apartados anteriores van a requerir la implementación de un sistema que permita identificar al usuario a través de distintas plataformas e identificarlos dentro del entorno multijugador.

Este sistema constará de un registro de usuario simplificado que sólo requerirá un nombre de usuario, una contraseña, y un correo para garantizar la robustez del sistema.

Los usuarios almacenarán estadísticas de juego, posiciones en el ranking multijugador y unas monedas que le permitirán comprar ventajas en el juego, dichas monedas se obtendrán ganando partidas, y también se podrán obtener de pago. Esto supondrá la principal fuente de obtención de ingresos de la aplicación. La siguiente lista muestra una serie de ideas en torno al sistema de monedas:

- Cuando una partida termine, se mostrarán estadísticas al usuario que mostrarán su rendimiento durante la misma mediante un sistema de medallas. Las monedas que reciba dependerán de dicho rendimiento y por supuesto de si ha ganado o perdido.
- Las monedas se podrán comprar a un precio considerablemente barato en relación al tiempo que se tarda en obtener dichas monedas jugando, provocando la necesidad de comprar.

- Las monedas se podrán gastar en comprar nuevos modos de juego, desbloquear dificultades o países y comprar bonificaciones globales para el jugador que se aplicarán durante cualquier partida que juegue, como aumentar un 10% la producción total.
- El modo de juego competitivo restringirá o limitará las ventajas de este tipo para que la habilidad del jugador siga sea determinante sobre otros parámetros.

La cuenta de usuario será común a todas las plataformas en las que se ejecute el juego, permitiendo al usuario usar sus ventajas en cualquier plataforma en la que ejecute el juego. También es imprescindible para identificar al usuario en el mundo online, permitiéndole tener una lista de amigos con los que interactuar y jugar a Dominations.

La implementación técnica del sistema de pagos es algo que todavía no ha sido investigado con profundidad como para exponerse en esta memoria, pero se realizará un análisis cuidadoso en el momento de incluir todos estos elementos en la aplicación.

Conclusiones finales

Dominations es una aplicación que combina elementos que han permitido el ascenso de muchas aplicaciones móviles del mundo del videojuego. El desafío intelectual que supone combinado con un estilo de juego adictivo y gratuito tiene un inmenso potencial para alcanzar una masa crítica de usuarios que constituyan una comunidad suficiente para provocar un efecto red.

Por otra parte, el sistema de micro transacciones permitirá una capitalización recurrente que dependerá de un correcto diseño de la relación ventaja/coste y del interés que los usuarios tengan en progresar en el juego, así como lo adictivo que resulte, es por ello que las prioridades de diseño del juego deberán ser las siguientes:

1. Diseño adictivo y desafiante.
2. Diseño accesible y usable.
3. Sistema de progresión permanente del usuario.
4. Juego competitivo multijugador atractivo.
5. Política de precios que el usuario vea útil pero no necesario consumir para poder ascender tanto en el sistema de progresión como en las jerarquías online.

Un correcto diseño de estos cinco factores determinará el éxito o fracaso de la aplicación. Con una arquitectura tan versátil como la expuesta en esta memoria resultará sencillo iterar sobre los diseños hasta alcanzar el punto óptimo que satisfaga las restricciones anteriores, permitiéndonos establecer un entorno de juego que acabará siendo atractivo para sus inversores, sus programadores y en definitiva para todos los usuarios.