UNIVERSIDAD TECNICA
FEDERICO SANTA MARIA

Proyecto final elo329: utilizacion de LWJGL para java 3D.

Integrantes:

- Carlos Ibañez
- Gabriel Juri
- Andrés Medina
- Lukas Perez

Descripcion del Problema.

- Videojuegos estan cada vez mas presentes en nuestras vidas. Ej: celular, PC, consolas, etc.
- Al analizar los videojuegos como elementos de software, vemos que estan implementados en base a complejos algoritmos. Ej: deteccion de colisiones, IA de algun tipo, etc.
- Ademas, la gran mayoria de estos poseen complejas interfaces 3D las cuales permiten darles mayor realismo.
- Todos estos elementos motivaron nuestra curiosidad y nos llevan a plantearnos una serie de consultas:

Descripcion del problema. (continuacion)

- ¿En que consiste una interfaz grafica 3D para un videojuego?
- ¿ Podria construirse una interfaz de este tipo en java ?
- ¿Como podriamos llevar una aplicacion con interfaz 2D a una aplicacion 3D?
- ¿ Podriamos con los conocimientos adquiridos en D.P.O.O realizar una aplicacion 3D en java?
- La idea de este proyecto, fue responder estas preguntas por medio del desarrollo de una aplicacion 3D en java utilizando una libreria de creacion de juegos llamada Light Weight Java Game Library (LWJGL).

Un poco sobre LWJGL

- Libreria open source que permite el desarrollo de juegos y aplicaciones 3D en java.
- Permite utilizar librerias de graficos y audio para juegos como OpenGL y OpenAL.
- Posee un API tal que permite facilmente tener acceso a configuracion de controles, gamepad, volantes de juego y otro tipo de controladores para juegos

Aplicacion a desarrollar

- Aplicacion que simula una batalla espacial.
- Cada nave debe intentar destruir a las demas para asi poder sobrevivir.
- La aplicación debe implementar algún algoritmo que detecte las colisiones entre las naves de forma eficiente y destruya ambas naves en caso de colisión.
- Cada nave dispondra de un arma la cual solo se activara si es que se detecta otra nave dentro de un campo de vision limitado.

- La cantidad inicial de naves en el campo de batalla va a ser 0 hasta que el usuario las agregue por medio de un click en una de las ventanas de la aplicacion.

- Esta aplicacion posee la particularidad que puede visualizarse en 2D, usando lo que ya conocemos de java y en 3D usando la libreria LWJGL.

- La aplicacion posee 2 ventanas:
  - La primera es una vista panoramica del campo de batalla (2D), en la que ademas de visualizar la batalla, se permita al usuario agregar naves por medio de un click.
  - La segunda es una vista 3D en la que se permite al usuario formar parte del campo de batalla en su propia nave con total libertad de movimiento, pero solo como un mero observador.

Casos de Uso

- Con el fin de aclarar más la interaccion del usuario con el programa, se utilizo la herramienta de casos de uso:

Caso1:

Nombre: Crear naves en la batalla espacial.

Descripcion: El sistema permite al usuario, agregar naves en un espacio vacio del campo de batalla por medio de un click sobre este.

Secuencia Normal:

1) Programa es abierto.

2) Usuario hace click en la ventana 2D (en un espacio vacio del campo de batalla).

3) El listener del mouse escucha el click.

4) Se crea un objeto nave en el espacio, el cual es visualizable.

5) El objeto comienza a moverse en busca de naves enemigas

Secuencia Alternativa: 2A1) Usuario hace click en la ventana 2D (en un lugar ocupado por otra nave).

3A1) La aplicacion detecta esto y elimina ambas naves.

Caso2:

Nombre: Movimiento en campo de batalla 3D.

Descripcion: El sistema permite al usuario, moverse libremente en el campo de batalla 3D utilizando los controles del teclado. Los controles permitidos son los de desplazamiento y rotación y están dados por las teclas A,W,S,D, FLECHA ARRIBA, FLECHA ABAJO, FLECHA IZQUIERDA Y FLECHA DERECHA.

Secuencia Normal: 1) Programa es abierto.

2) Usuario selecciona la ventana 3D.

3) Usuario presiona alguna de las teclas de desplazamiento o rotación.

4) El sistema detecta la tecla presionada por el usuario.

5) El sistema realiza los cambios en los sistemas coordenados según lo requiera la acción hecha por el usuario.

Secuencia Alternativa: 3A1) Usuario presiona una tecla que distinta a las definidas previamente como controles permitidos.
4A1) El sistema no detecta la tecla presionada por el usuario por lo que no realiza ninguna acción.

Diseño de la solución: aplicación de batalla espacial

- Detección de colisiones -> algoritmo de detección por área
  - Campo de batalla se divide en pequeños elementos de área
  - Si es que 2 elementos coinciden en un mismo cuadrante

↓

Peligro de colisión: ambos elementos son destruidas

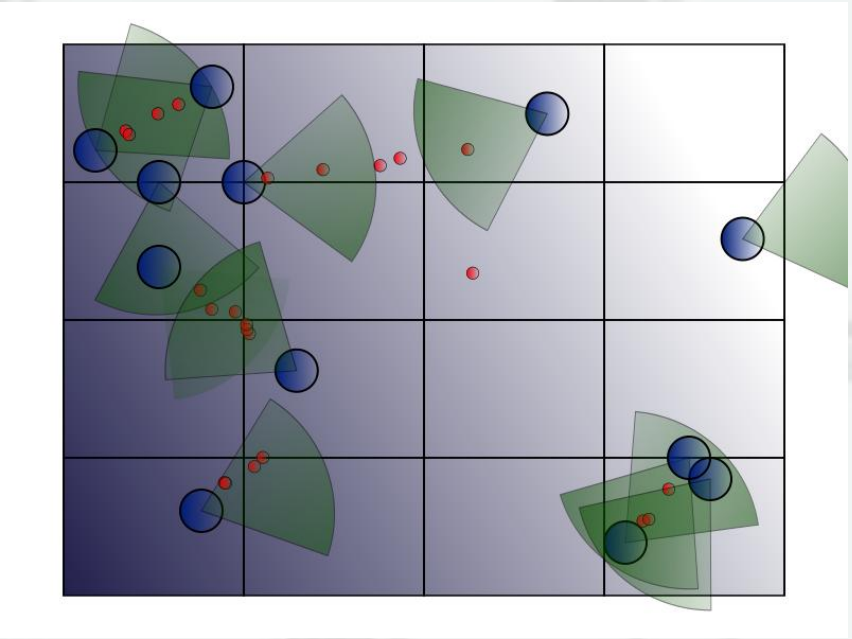


Diagrama de clases:

- **World**: Clase que representa al mundo donde interactúan los elementos del programa como naves balas y sensores
- **CollisionDetector:** Clase que presta utilidades de detección de colisiones.
- **Bullet**: Clase bala que se mueven en línea recta. Tiene representación grafica, es derivada de GElement.
- **Gun:** Es la clase que crea objetos Bullet en un objeto World.
- **Sensor:** Es una clase intermediaria entre el objeto World y el objeto SElement, permite a SElement obtener informacion sobre World.
- **SElement:** Representa al objeto nave en un nivel mayor, esta es derivada de GElement..

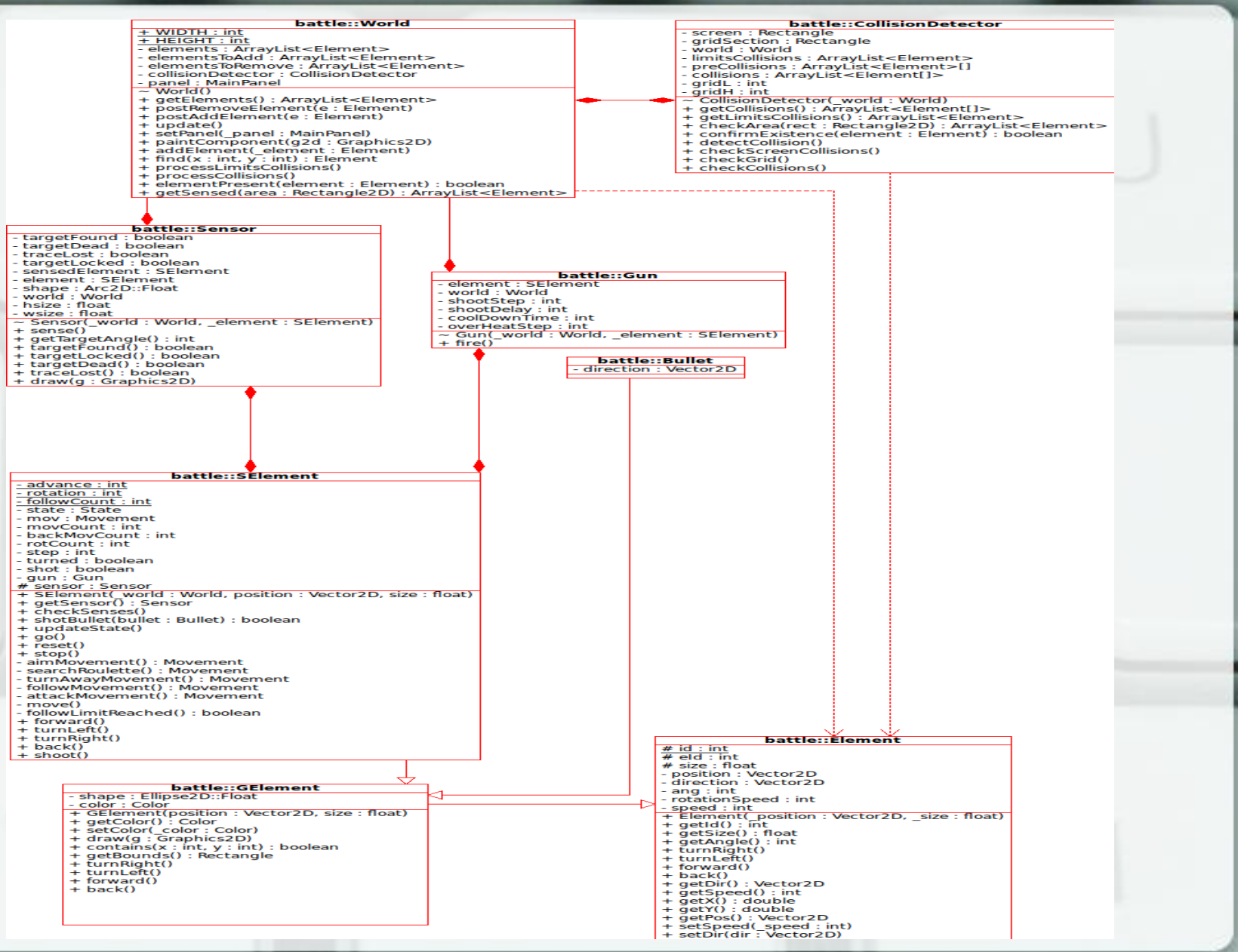

battle::World
+ WIDTH : int
+ HEIGHT : int
- elements : ArrayList<Element>
- elementsToAdd : ArrayList<Element>
- elementsToRemove : ArrayList<Element>
- collisionDetector : CollisionDetector
- panel : MainPanel
~ World()
+ getElements() : ArrayList<Element>
+ postRemoveElement(e : Element)
+ postAddElement(e : Element)
+ update()
+ setPanel(_panel : MainPanel)
+ paintComponent(g2d : Graphics2D)
+ addElement(_element : Element)
+ find(x : int, y : int) : Element
+ processLimitsCollisions()
+ processCollisions()
+ elementPresent(element : Element) : boolean
+ getSensed(area : Rectangle2D) : ArrayList<Element>
battle::CollisionDetector
- screen : Rectangle
- gridSection : Rectangle
- world : World
- limitsCollisions : ArrayList<Element>
- preCollisions : ArrayList<Element>[]
- collisions : ArrayList<Element[]>
- gridL : int
- gridH : int
~ CollisionDetector(_world : World)
+ getCollisions() : ArrayList<Element[]>
+ getLimitsCollisions() : ArrayList<Element>
+ checkArea(rect : Rectangle2D) : ArrayList<Element>
+ confirmExistence(element : Element) : boolean
+ detectCollision()
+ checkScreenCollisions()
+ checkGrid()
+ checkCollisions()
battle::Sensor
- targetFound : boolean
- targetDead : boolean
- traceLost : boolean
- targetLocked : boolean
- sensedElement : SElement
- element : SElement
- shape : Arc2D::Float
- world : World
- hsize : float
- wsize : float
~ Sensor(_world : World, _element : SElement)
+ sense()
+ getTargetAngle() : int
+ targetFound() : boolean
+ targetLocked() : boolean
+ targetDead() : boolean
+ traceLost() : boolean
+ draw(g : Graphics2D)
battle::Gun
- element : SElement
- world : World
- shootStep : int
- shootDelay : int
- coolDownTime : int
- overHeatStep : int
~ Gun(_world : World, _element : SElement)
+ fire()
battle::Bullet
- direction : Vector2D
battle::SElement
- advance : int
- rotation : int
- followCount : int
- state : State
- mov : Movement
- movCount : int
- backMovCount : int
- rotCount : int
- step : int
- turned : boolean
- shot : boolean
- gun : Gun
sensor : Sensor
+ SElement(_world : World, position : Vector2D, size : float)
+ getSensor() : Sensor
+ checkSenses()
+ shotBullet(bullet : Bullet) : boolean
+ updateState()
+ go()
+ reset()
+ stop()
- aimMovement() : Movement
- searchRoulette() : Movement
- turnAwayMovement() : Movement
- followMovement() : Movement
- attackMovement() : Movement
- move()
- followLimitReached() : boolean
+ forward()
+ turnLeft()
+ turnRight()
+ back()
+ shoot()
battle::GElement
- shape : Ellipse2D::Float
- color : Color
+ GElement(position : Vector2D, size : float)
+ getColor() : Color
+ setColor(_color : Color)
+ draw(g : Graphics2D)
+ contains(x : int, y : int) : boolean
+ getBounds() : Rectangle
+ turnRight()
+ turnLeft()
+ forward()
+ back()
battle::Element
id : int
eid : int
size : float
- position : Vector2D
- direction : Vector2D
- ang : int
- rotationSpeed : int
- speed : int
+ Element(_position : Vector2D, _size : float)
+ getId() : int
+ getSize() : float
+ getAngle() : int
+ turnRight()
+ turnLeft()
+ forward()
+ back()
+ getDir() : Vector2D
+ getSpeed() : int
+ getX() : double
+ getY() : double
+ getPos() : Vector2D
+ setSpeed(_speed : int)
+ setDir(dir : Vector2D)

Diagrama de estados de la nave:

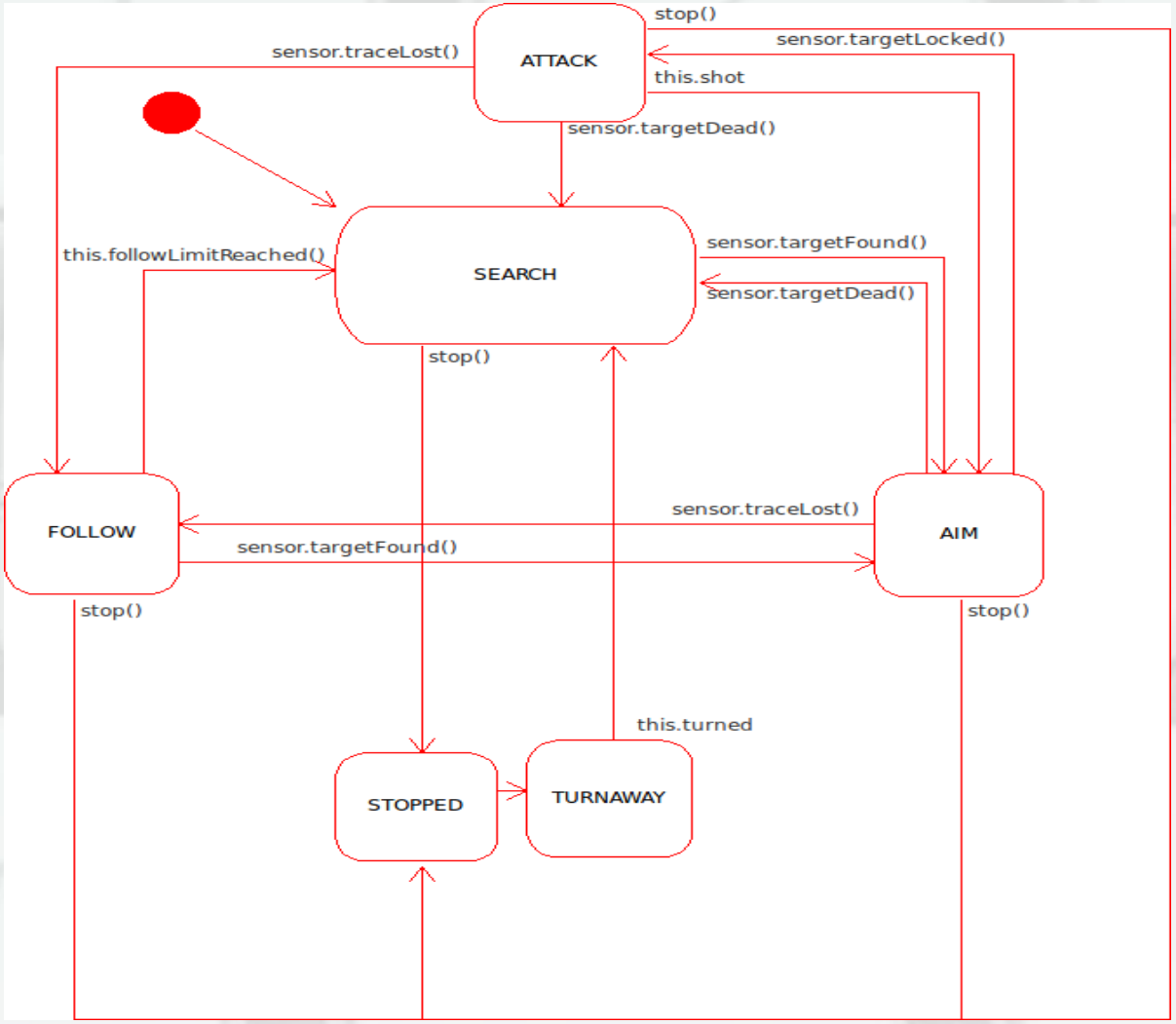
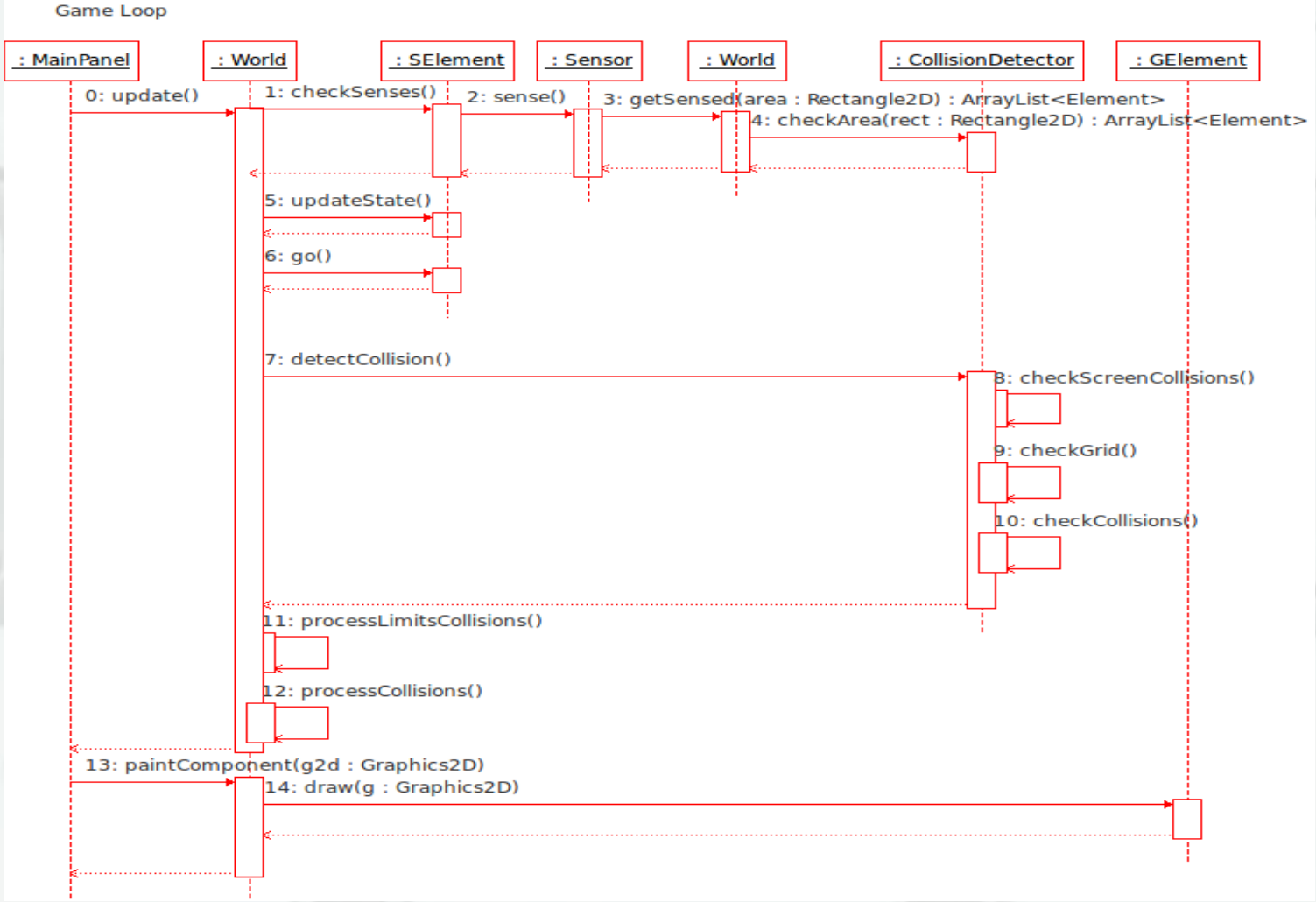
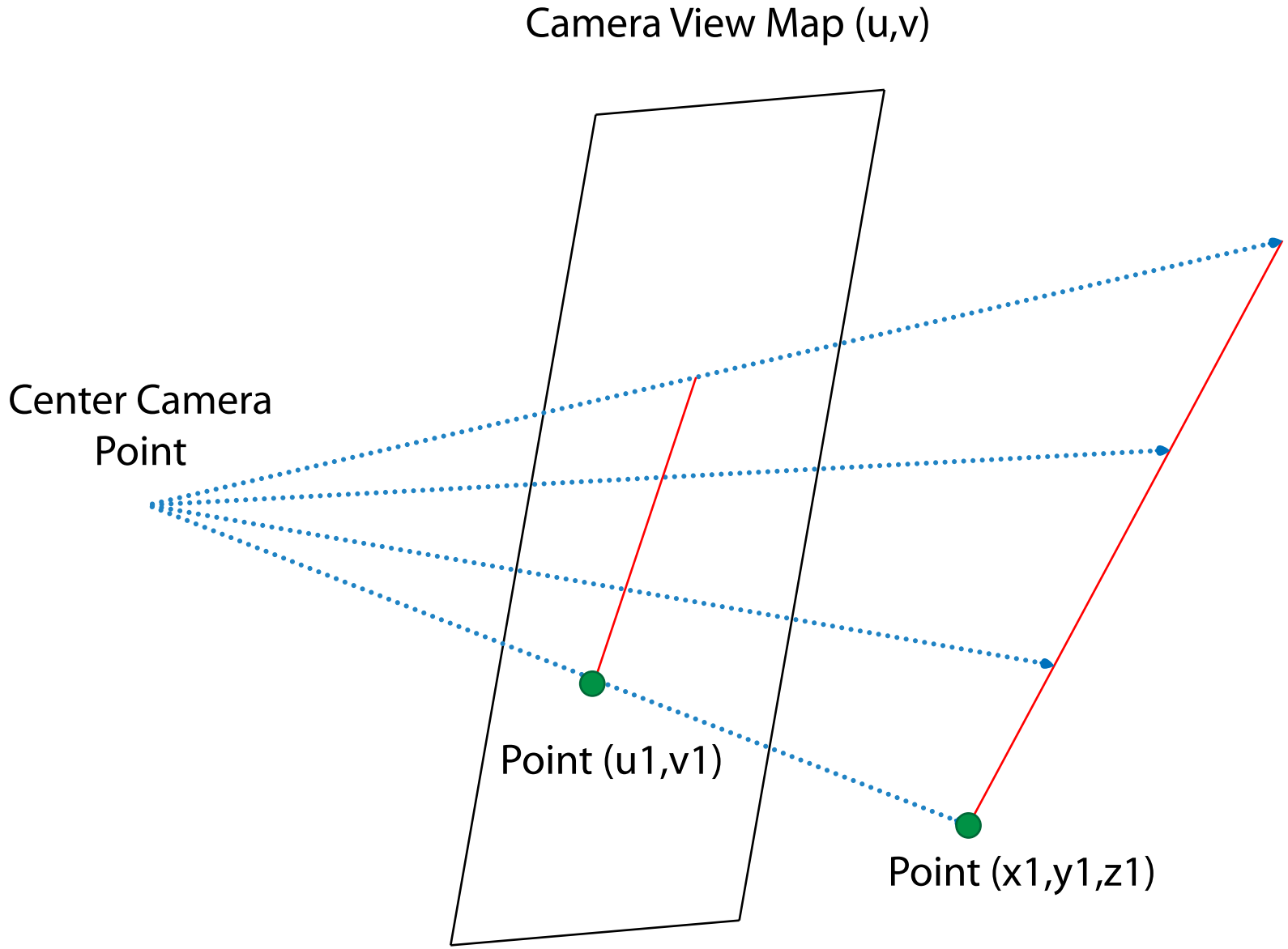


Diagrama de Secuencia:



¿Como Funciona una Camara?
Proyectivas



¿Que se espera?
Visión y Movimiento 3D

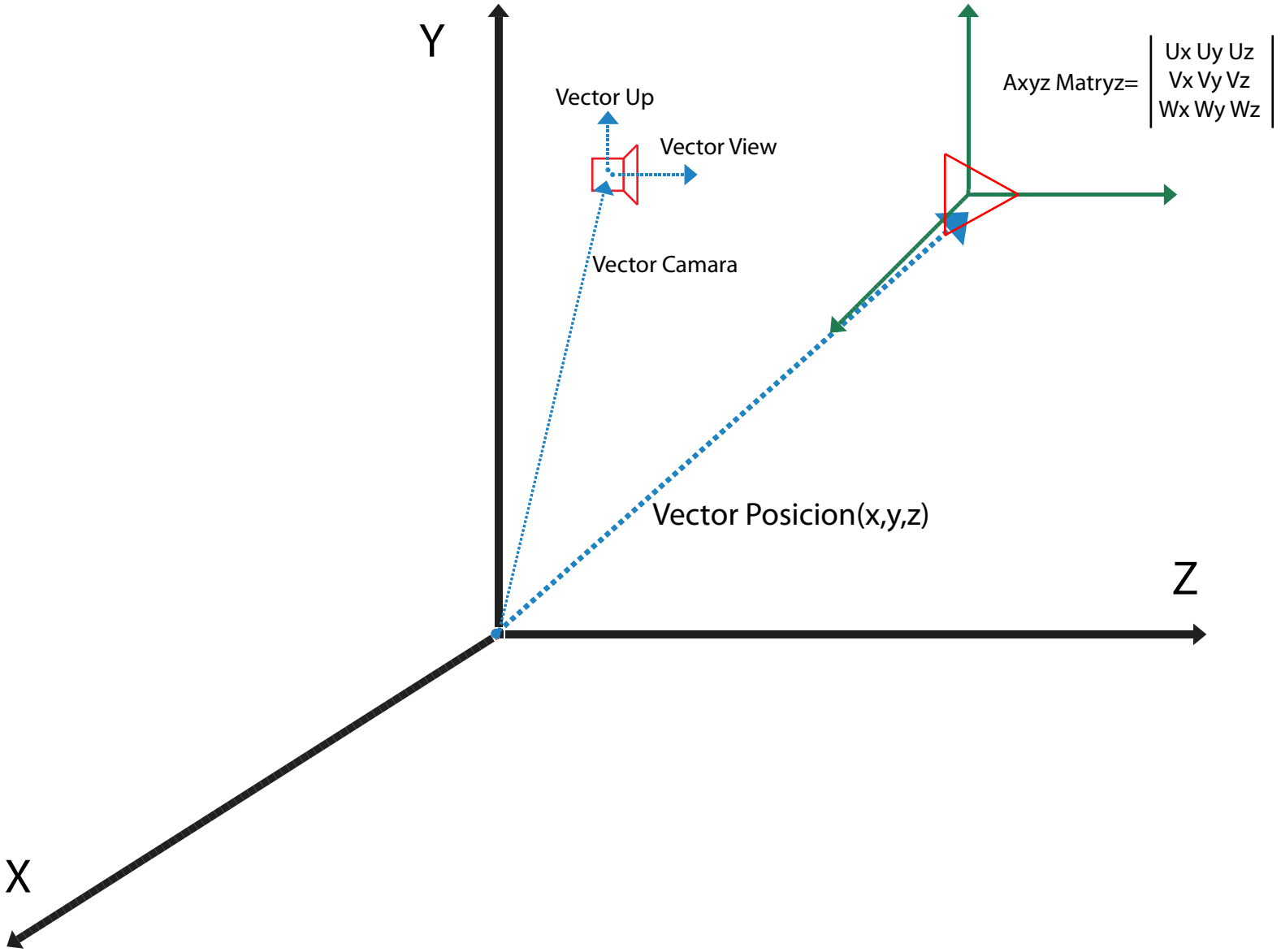


Diagrama de Clases
Paquete GlGame Primera Iteración

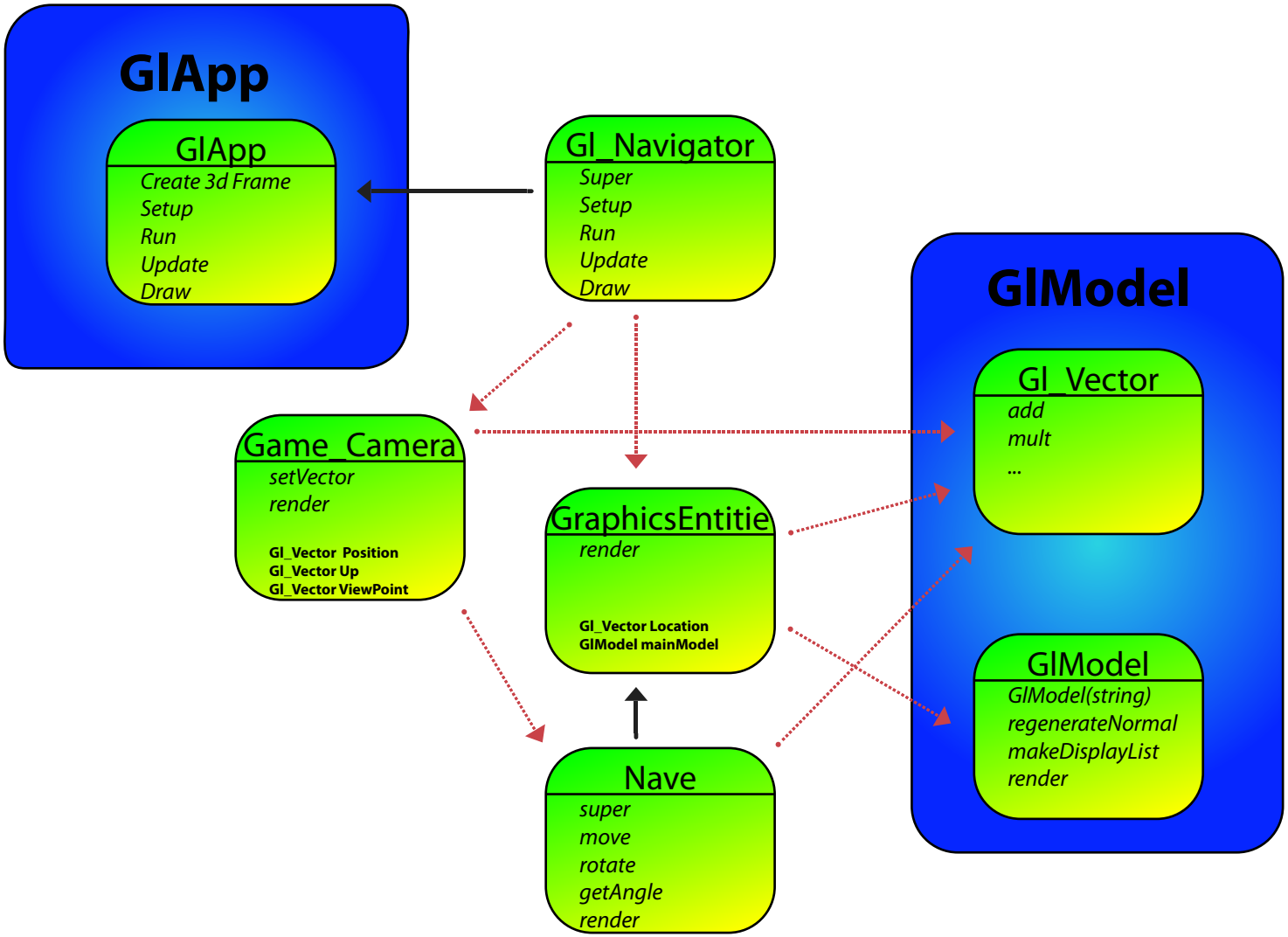


Diagrama de Clases
Paquete GlGame

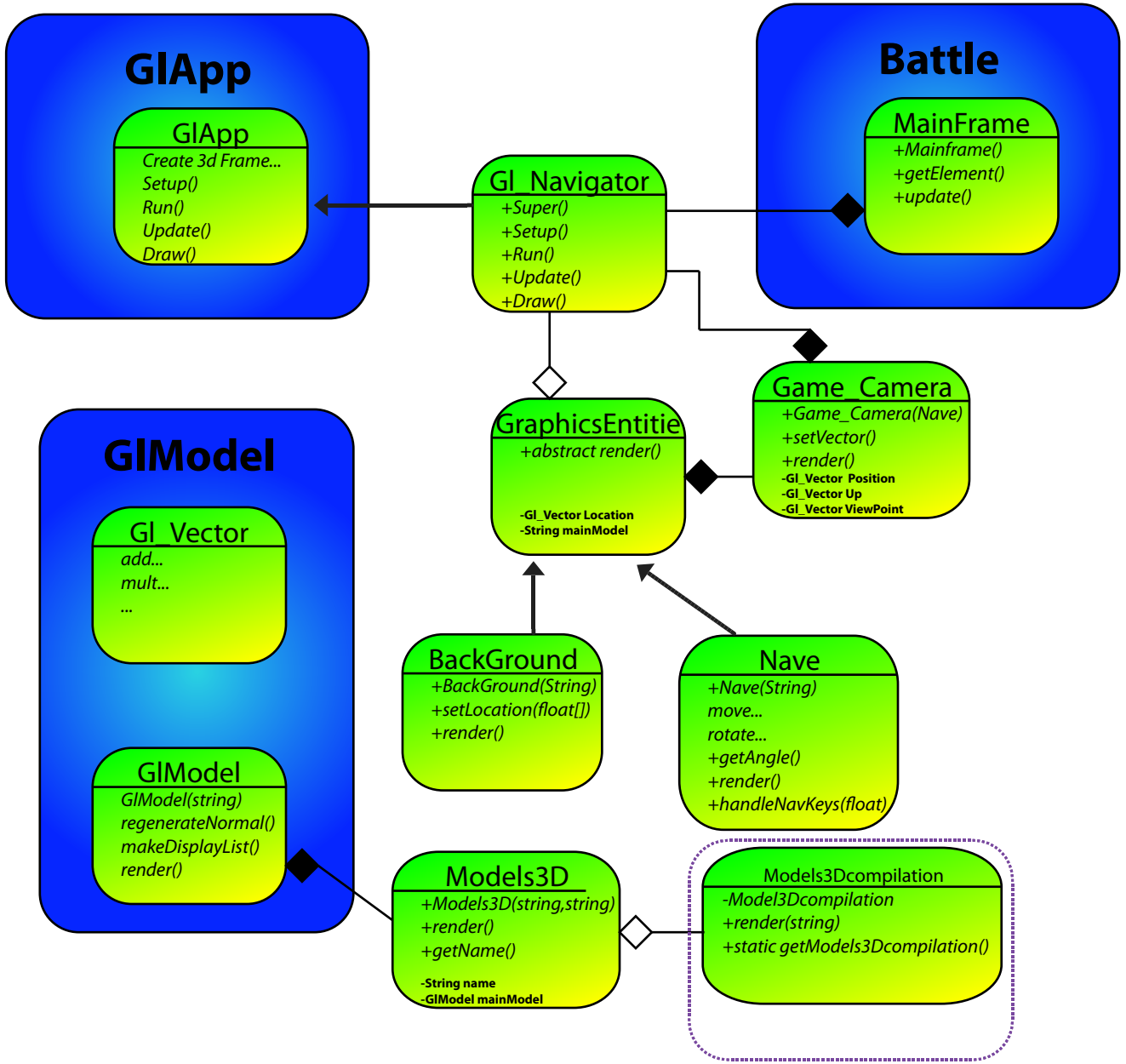
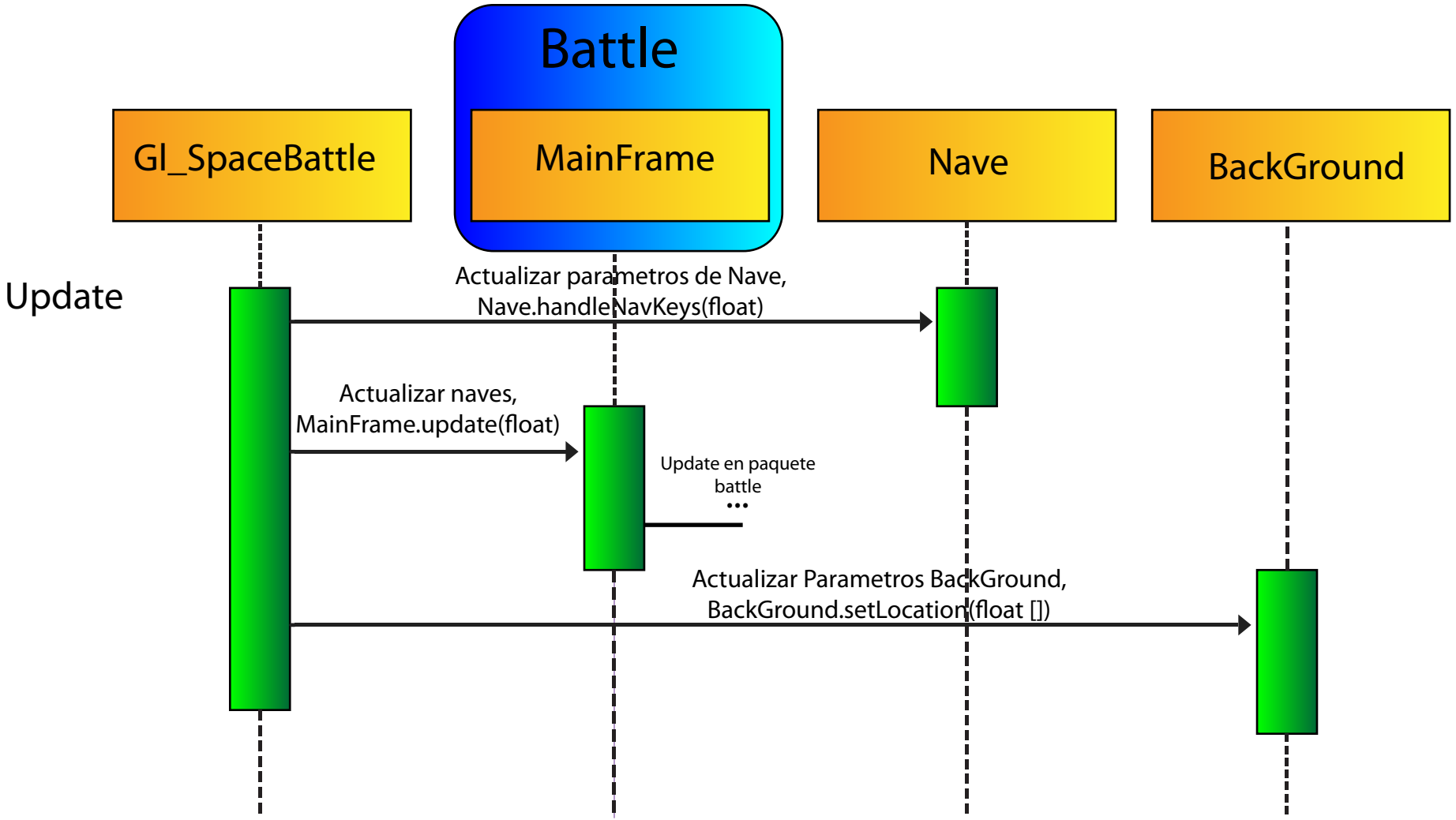
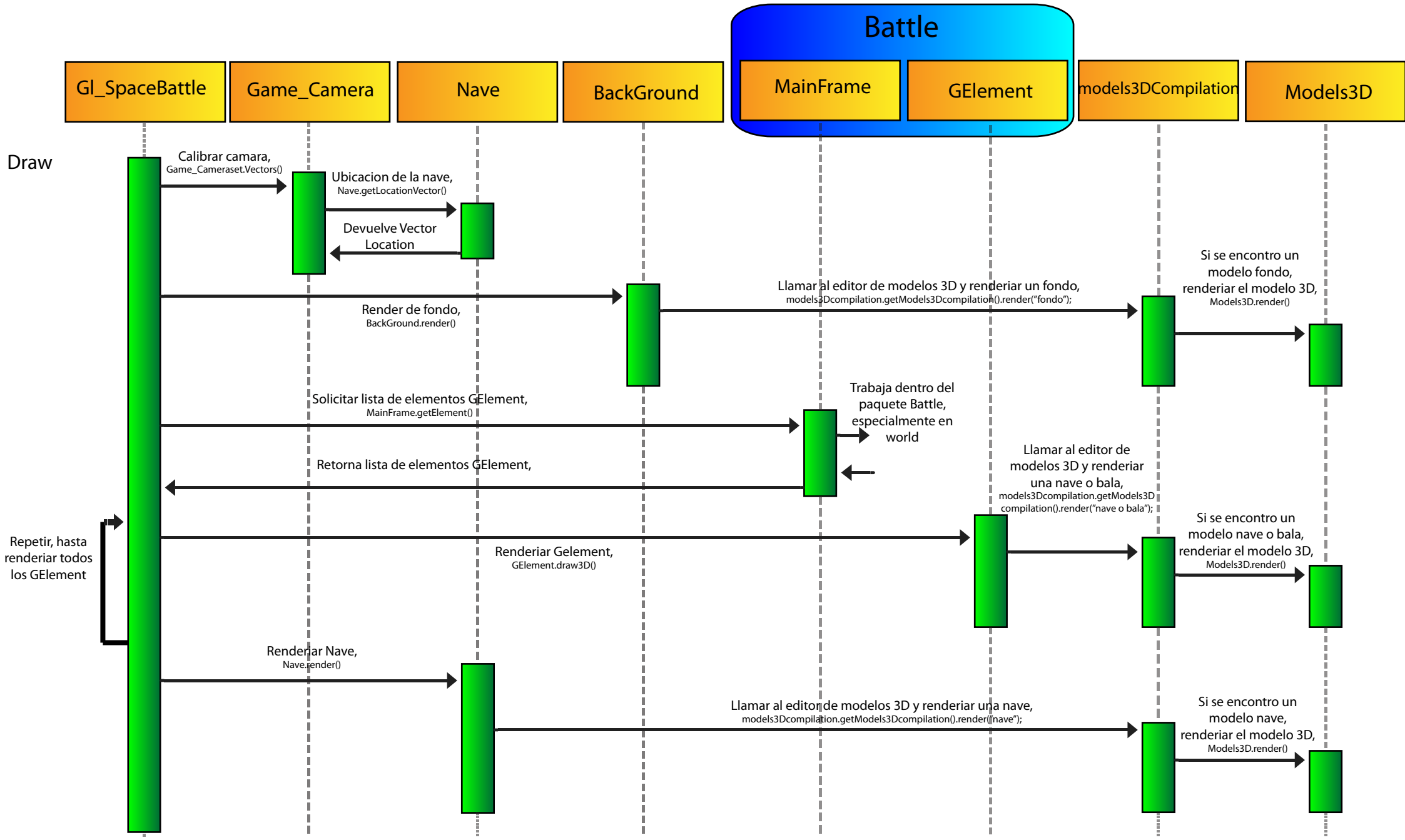


Diagrama de secuencia
Update y draw




Battle
Gl_SpaceBattle
Game_Camera
Nave
BackGround
MainFrame
GElement
models3DCompilation
Models3D
Draw
Calibrar camara,
Game_Cameraset.Vectors()
Ubicacion de la nave,
Nave.getLocationVector()
Devuelve Vector
Location
Render de fondo,
BackGround.render()
Llamar al editor de modelos 3D y renderiar un fondo,
models3Dcompilation.getModels3Dcompilation().render("fondo");
Si se encontro un
modelo fondo,
renderiar el modelo 3D,
Models3D.render()
Solicitar lista de elementos GElement,
MainFrame.getElement()
Trabaja dentro del
paquete Battle,
especialmente en
world
Retorna lista de elementos GElement,
Llamar al editor de
modelos 3D y renderiar
una nave o bala,
models3Dcompilation.getModels3D
compilation().render("nave o bala");
Si se encontro un
modelo nave o bala,
renderiar el modelo 3D,
Models3D.render()
Repetir, hasta
renderiar todos
los GElement
Renderiar Gelement,
GElement.draw3D()
Renderiar Nave,
Nave.render()
Llamar al editor de modelos 3D y renderiar una nave,
models3Dcompilation.getModels3Dcompilation().render("nave");
Si se encontro un
modelo nave,
renderiar el modelo 3D,
Models3D.render()

Lo que se espera.
¿Se logrará?