

Relazione relativa all'ingegnerizzazione di GeometryWars: un'applicazione ispirata al gioco Geometry Wars.

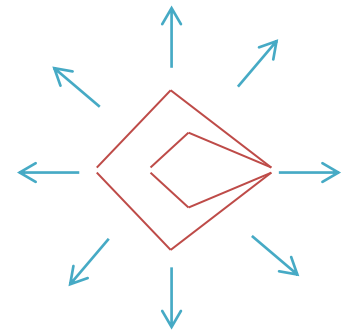
Componenti del gruppo: Federico Giusti 671418.

1 Analisi del problema

Si vuole ricreare una versione semplificata (adeguata a questo progetto) del gioco Geometry Wars. Il gioco consiste in una navicella, all'interno di un'area delimitata da un rettangolo, che deve riuscire a sopravvivere a dei nemici generati sempre più velocemente con l'avanzare dei livelli di gioco.

Nello specifico il giocatore (navicella) potrà:

- Muoversi in 8 direzioni differenti, in base alle varie combinazioni dei tasti W, A, S, e D, all'interno del campo di gioco.
- Direzionare lo sparo automatico attraverso il mouse, per colpire i nemici.
- Raccogliere power-ups (aumento fire-rate e scudo).



Uccidendo i nemici, il giocatore:

- Otterrà punti in base alla tipologia del nemico ucciso.
- Incrementerà l'avanzamento della barra del livello.

Mentre essere colpiti da un nemico determina la fine del gioco, mostrando il punteggio ottenuto.

I nemici sono di tre tipologie, differenti per velocità di movimento, forma e colore.

2 Progettazione architetturale

La fase di progettazione è stata la più ardua, in quanto si è dovuto trovare un modo per gestire i vari oggetti, con la possibilità di modificare il loro stato nel tempo, e contemporaneamente aggiornare il frame, che si occuperà della visualizzazione del gioco.

Per far questo si è deciso di basare il gioco su un pattern **Model-View-Controller**, in modo da gestire separatamente le varie classi del model, modificando gli oggetti istanziati nel loro stato (visibile o meno) e nella loro posizione, visualizzare i vari elementi e gestire il tutto attraverso il controller. In oltre, per poter richiamare il controller costantemente nel tempo, è stata creata la classe *GeometryWars* contenente il thread di base.

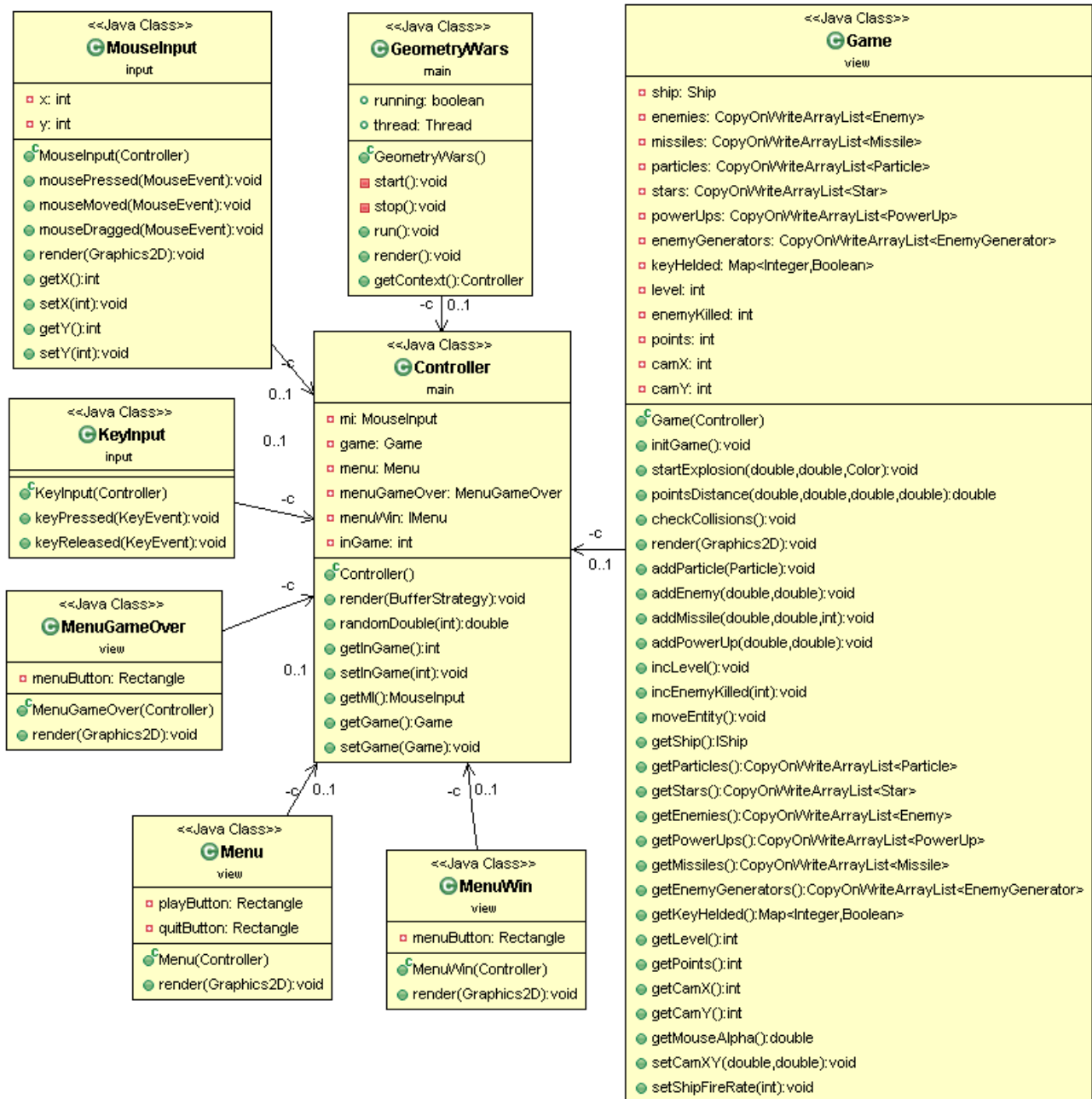
Altro problema riscontrato è quello di ottenere una renderizzazione performante dei vari oggetti. Per questo motivo si è deciso di far estendere alla classe *GeometryWars* la classe *java.awt.Canvas*, utilizzata sia per ottenere un'area rettangolare nella quale poter disegnare l'applicazione, sia per creare un multi-buffer attraverso il metodo *createBufferStrategy()*.

Inoltre alla classe *Canvas*, dal momento che a sua volta estende la classe *java.awt.Component*, verrà utilizzata per aggiungere all'applicazione dei listener per il controllo dell'input da mouse e tastiera, attraverso i metodi *addKeyListener()*, *addMouseListener()*, *addMouseMotionListener()*. Gli oggetti gestiti da questi metodi saranno passati sempre dal *Controller*.

Per poter disegnare i vari elementi (oggetti *Poly*, interfaccia di gioco, menu etc..) si è deciso di utilizzare semplicemente la classe *awt* di Java, senza ricorrere all'utilizzo di immagini esterne, per poter ridurre i tempi di sviluppo del progetto, in quanto si ha una maggiore dinamicità nell'inventare e creare nuovi elementi.

Ritornando al thread che gestisce il tutto, dovrà richiamare costantemente il controller, che si occuperà ogni 3ms di spostare, gestire le collisioni e renderizzare gli oggetti all'interno del campo di gioco. Inoltre ogni 2 secondi modificherà la posizione dai cui verranno generati nemici e power-ups (cioè quale *EnemyGenerator* utilizzare).

Di seguito il diagramma UML relativo alla struttura di base del progetto:

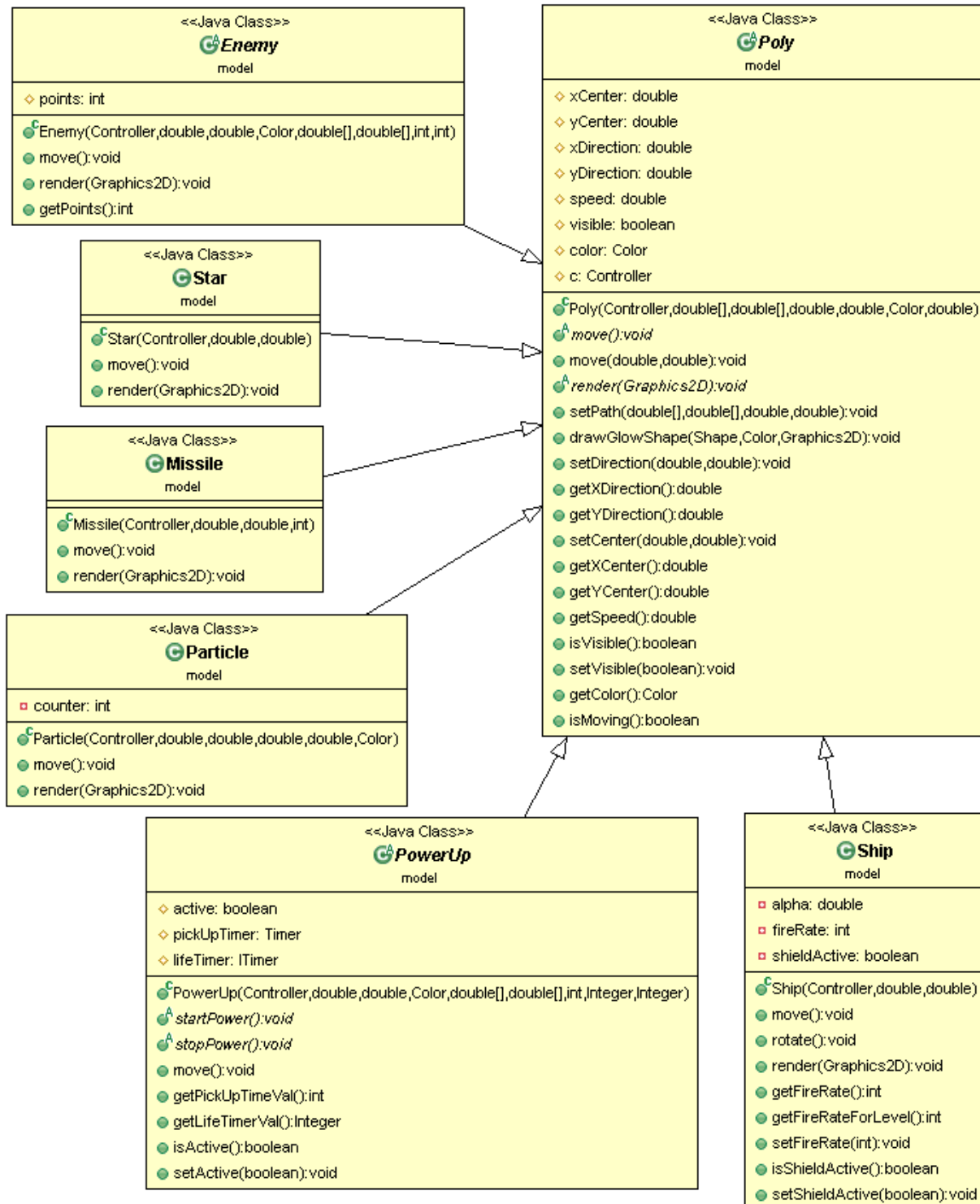


Per quanto riguarda il gioco vero e proprio, si è deciso di far gestire tutti gli oggetti che lo compongono dalla view *Game*. Le liste contenenti nemici, missili, power-up, particelle e stelle sono del tipo *CopyOnWriteArrayList* per ovviare a problemi di concorrenza derivanti dalla gestione dei thread.

Alla base di ogni oggetto che dovrà interagire con l'utente, che avrà possibilità di muoversi, di "nascere e morire" (visibile o meno), si trova la classe *Poly*, estendente la classe *Path2D.Double*. Questa classe permette di assegnare all'oggetto che la estende una forma geometrica, in questo caso con doppia precisione, con i relativi metodi per poterla gestire. Nello specifico sarà utilizzato il metodo *transform()* per poter spostare il poligono nell'area di gioco, riassegnando nuovi valori ai punti del poligono descrivente la forma dell'oggetto in questione.

La classe *Poly* verrà estesa da classi che specificheranno come si dovrà comportare quel specifico oggetto *Poly*, quale forma e colore dovrà avere ed eventuali “azioni” proprie dell’oggetto in questione (ad esempio applicare o togliere miglioramenti in seguito al raccoglimento di un power-up).

Di seguito il diagramma UML della classe *Poly* con le sue estensioni:



Il progetto inizialmente era stato pensato per poter gestire anche una vista “Opzioni” ma poi sostituito con un’interfaccia *Icommons*, implementata da quasi tutte le classi. Grazie a questa interfaccia è possibile dare al gioco innumerevoli aspetti e dinamiche di gioco differenti, dal momento che si può scegliere di quanti livelli è composto il gioco, quanti nemici devono essere generati e di che tipo, forma e colore dei vari oggetti disegnati, etc...

3 Organizzazione in Package

Il progetto è organizzato in 5 package differenti:

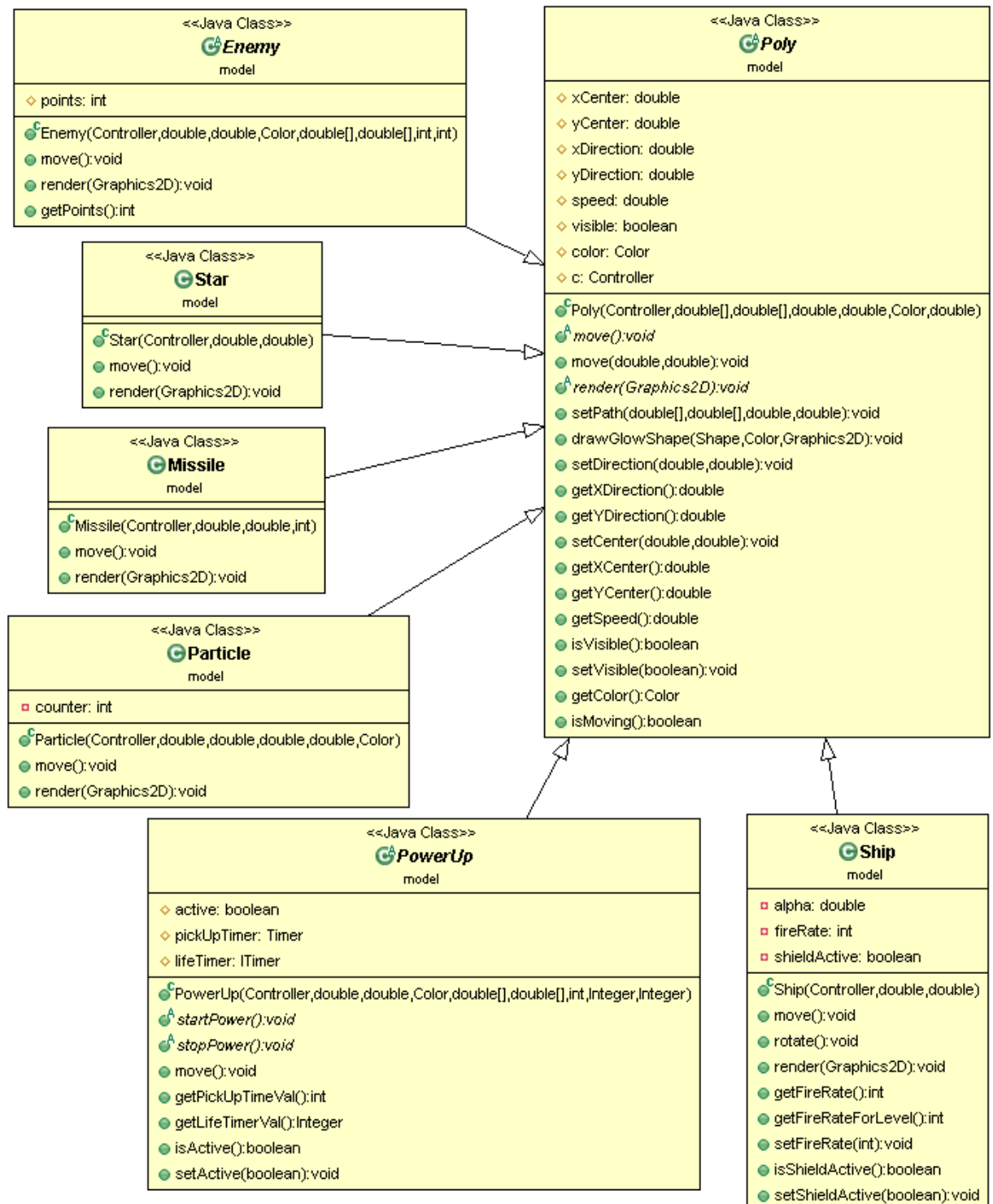
- **main:** Ci sono le classi principali che gestiscono l’intero gioco. La classe *Main* per avviare il programma, la classe *GeometryWars* contenente il thread principale e la classe *Controller*
- **input:** Contiene due classi per la gestione dell’input da tastiera e mouse.
- **model:** Contiene tutte le classi necessarie a “modellare” l’intero gioco, quali la classe *Ship* per la navicella (giocatore), i nemici, power-up, timer etc...
- **view:** Contiene tre tipologie diverse di menu (iniziale, vincintia, perdita) e la classe *Game*, quella che si occupa della visualizzazione del gioco.
- **interfaces:** contiene tutte le interfacce delle relative classi e la classe *Icommons* contenente tutte le variabili globali utilizzate nell’intero progetto.

4 Progettazione di dettaglio

Più in particolare si hanno diversi aspetti da considerare quali:

- **Quattro viste:** Sono dichiarati nella classe *Controller*, che gestirà la loro corretta visualizzazione attraverso la variabile `inGame`, assumendo questi valori: “0” se ci si trova in gioco, “1” se si deve visualizzare il menu iniziale, “2” per quello game over e “3” per il menu win. I tre menu, uno iniziale (*menu*) per poter iniziare la partita o uscire, e due per stabilire se si è perso (*menuGameOver*) o vinto, finendo i livelli prestabiliti (*menuWin*), mostrando in entrambi il punteggio ottenuto ed il livello raggiunto e dando la possibilità di ritornare al menu iniziale o uscire dal gioco. I tre menu implementano l’interfaccia *IMenu* contenente il metodo *render()*, che si occuperà di disegnare i vari componenti in base alla classe a cui appartiene. La quarta vista è la classe *Game*, che, quando attiva, gestirà il funzionamento e la renderizzazione del gioco vero e proprio.
- **La classe Game:** In questa classe sono memorizzati i vari elementi che interagiranno tra loro, quali la navicella del giocatore, i nemici, i missili, i potenziamenti, i generatori e le stelle di background, disegnate dinamicamente ogni volta dal metodo *render()*. Contiene anche le variabili per tener conto del livello raggiunto, dei punti ottenuti, dei tasti premuti da tastiera e della posizione della “telecamera” spiegata successivamente. Inoltre contiene i metodi per controllare se sono avvenute collisioni tra le varie estensioni diverse della classe *Poly*, per generare particelle al momento dell’esplosione di missili e nemici, ed in fine per gestire le variabili sopra citate.

- **La telecamera:** Per poter avere una vista dinamica che si muove assieme alla navicella mentre si sposta nell'area di gioco salvo la posizione della telecamera (il centro dello schermo inizialmente). Ogni volta che il giocatore sposta la navicella questi valori vengono incrementati o decrementati, attraverso il metodo *setCamXY()*, e, ogni volta che viene richiamato il metodo *render()* della view *Game*, vengono passati al metodo *translate()* della classe *Graphics2D*, spostando praticamente l'origine degli assi di disegno.
- **La classe Poly:** Ogni elemento in grado di muoversi, avente una posizione, una direzione, una velocità, una "forma" da disegnare ed un colore è esteso da questa classe, quali le classi *Ship*, *Missile*, *Enemy*, *PowerUp*, *Particle* e *Star*.



- **Il giocatore:** La classe *Ship* oltre ad avere le variabili derivanti dall'estensione della classe *Poly* dovrà gestire le variabili *alpha*, per la rotazione della navicella su se stessa attraverso il metodo *rotate()*, *fireRate*, per la velocità di generazione dei missili e *shieldActive*, per verificare se il power-up relativo allo scudo è attivo.
- **I nemici:** La classe *Enemy* è una classe astratta, dato che verrà estesa da altre tre classi, una per ogni tipo di nemico. Le classi *Enemy1*, *Enemy2* ed *Enemy3* avranno semplicemente costruttori differenti da passare alla classe *Enemy*, ma sarebbe stato possibile implementare per ognuno di essi un metodo *move()* differente, dandogli quindi modalità di movimento differenti. I diversi costruttori daranno allo specifico nemico una forma differente, un colore, una velocità e un punteggio per l'uccisione.
- **I power-up:** La classe *PowerUp*, come la classe *Enemy*, è una classe astratta che però si differenzia nell'estensione delle classi *PowerUp1* e *PowerUp2*, in quanto esse hanno al loro interno hanno anche il metodo *startPower()*, per applicare gli effetti del power-up, e *stopPower()* per rimuoverli.
- **La generazione di nemici e power-up:** La classe *EnemyGenerator* si occupa di tener posizione di un punto sul campo di gioco nel quale verranno generati nemici o power-up. Il progetto è stato impostato per averne quattro agli angoli del rettangolo di gioco, richiamati casualmente per ogni lasso di tempo stabilito.

5 Testing

Per la fase di testing ho utilizzato la *JConsole* per poter tenere sotto controllo l'ulizzo del CPU e delle risorse ed impostando `EDIT_MODE = true` in *ICommons*, mostrando durante i test di gioco anche il centro degli oggetti *Poly*, il loro raggio di collisione, la direzione della navicella, il centro del campo di gioco ed in fine dati relativi al numero di oggetti generati e telecamera.

6 Note finali

Il progetto è stato sviluppato nei tempi stabiliti senza riscontrare particolari problemi ed è stato pensato per rientrare nelle 100 ore ma potrebbe essere ampliato e migliorato, con più tempo a disposizione.

Ampliato:

- Si potrebbe creare più tipologie di nemici e per ogni nemico si potrebbe dare modalità differenti di movimento cambiando solo il metodo *move()*.
- Si potrebbe creare più tipologie di power-up implementando le nuove classi simili a power-up1 e 2 cambiando semplicemente i metodi *start-power* e *stop-power*.
- La classe *ICommons* è una versione "grezza" di una possibile view di configurazione e le variabili statiche finale descritte in questa classe potrebbero essere salvate in un file di testo.
- I punteggi ottenuti alla fine del gioco potrebbero essere salvati in un'ulteriore view con la classifica generale delle partite giocate.

Migliorato:

- Il codice non è particolarmente “pulito”.
- Si sarebbe potuto utilizzare un pattern SINGLETON sia per il controller, che per la classe ICommons (ampliandola anche con metodi per gestirla più velocemente e dinamicamente), dando loro una visibilità statica globale e rimuovendo la necessità di doverli passare come oggetti nei riferimenti delle classi.
- Inoltre si sarebbero potute creare più classi per poter gestire meglio ed in maniera più pulita telecamera, l’interfaccia di gioco, i punteggi etc...