SEVEN CLICKERS

You Wish We Click

7clickersgroup@gmail.com

Specifica Tecnica

Versione	1.0.0
Stato	Approvato
Uso	Esterno
Approvazione$_g$	Mirko Stella
Redazione	Mirko Stella
	Rino Sincic
Verifica$_g$	Giacomo Mason
	Gabriele Mantoan
Distribuzione	*Seven Clickers* Prof. Vardanega Tullio Prof. Cardin Riccardo

Descrizione

Specifica Tecnica dell'architettura di progetto

Registro delle modifiche

Vers.	Data	Autore	Ruolo	Descrizione
1.0.0	21-05-23	Mirko Stella	Responsabile	Approvazione documento
0.0.9	19-05-23	Rino Sincic Giacomo Mason Gabriele Mantoan	Programmatore Verificatori	Stesura sezione Architettura di deployment
0.0.8	15-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione StoreDiagram
0.0.7	13-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione InterfaceFeaturesDiagram
0.0.6	12-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione SidebarFeaturesDiagram
0.0.5	11-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione PlayerFeaturesDiagram
0.0.4	08-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione CartFeaturesDiagram
0.0.3	07-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione Design pattern architetturale determinato dalle tecnologie adottate
0.0.2	05-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione Elenco dei componenti
0.0.1	04-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Stesura sezione Introduzione
0.0.0	03-05-23	Mirko Stella Giacomo Mason Gabriele Mantoan	Responsabile Verificatori	Creazione struttura documento

Indice

Elenco delle figure

Elenco delle tabelle

1 Introduzione

1.1 Glossario

In questo documento sono state segnate con il pedice "g" tutte le parole che, secondo noi, necessitano di una spiegazione ulteriore per evitare eventuali ambiguità o incomprensioni.

La spiegazione di questi termini la si può trovare nel documento di *Glossario*.

1.2 Scopo del documento

Il documento *Specifica Tecnica* ha lo scopo di descrivere i componenti utilizzati e le scelte progettuali fatte per la realizzazione del prodotto.

Dopo aver fornito un elenco descrittivo dei componenti verranno spiegati nel dettaglio, utilizzando l'ausilio degli schemi UML, i seguenti punti di interesse:

- Design pattern architetturale determinato dalle tecnologie adottate;
- Architettura logica (dipendenze e interazioni tra componenti);
- Architettura di deployment.

1.3 Suggerimenti per la comprensione del documento

Per comprendere al meglio l'architettura utilizzata è importante comprendere in primo luogo il design pattern architetturale derterminato dalle tecnologie adottate in quanto gran parte delle scelte fatte si basano su di esso.

Si suggerisce quindi di leggere la sezione 2.2 riguardante il design pattern architetturale determinato dalle tecnologie adottate prima di proseguire con le successive.

1.4 Riferimenti

1.4.1 Riferimenti normativi

- Capitolato d'appalto C6;
- *Norme di Progetto v*;
- *Analisi dei Requisiti c.*

1.4.2 Riferimenti informativi

- Materiale didattico IS: Diagrammi delle classi.

2 Descrizione dell'architettura

2.1 Elenco dei componenti

- **Slices**

  Le Slice contengono una porzione dello stato globale dell'applicazione che è gestita da un reducer specifico.

  - **CartSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti il carrello;
  - **ProductsSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti i prodotti presenti all'interno dell'ambiente 3D;

- **RayCasterSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti il raycaster;
- **DecorationSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti il caricamento delle decorazioni all'interno dell'ambiente 3D;
- **SidebarSlice**: componente che permette la gestione dello stato che contiene i dati riguardanti la side-bar.

- ## Initial states

Gli stati iniziali risiedono all'interno di ciascuna slice e la loro unione contiene i dati che formano lo stato globale dell'applicazione gestito dallo store. Tutti gli altri dati sono contenuti e gestiti dai componenti React che li utilizzano.

- **CartInitialState**: componente che contiene i dati relativi agli oggetti presenti all'interno del carrello;
- **ProductsInitialState**: componente che contiene i dati relativi ai prodotti presenti all'interno dell'ambiente 3D;
- **RayCasterInitialState**: componente che contiene i dati utili ad interagire con un oggetto presente all'interno dell' ambiente 3D;
- **DecorationInitialState**: componente che contiene la lista di tutte le decorazioni presenti all'interno dell'ambiente 3D;
- **SidebarInitialState**: componente che contiene i dati utili a reperire informazioni sulla sidebar;

- ## Actions

Le azioni sono emesse dai componenti e sono inviate ai reducers per aggiornare lo stato corrispondente. Le azioni sono caratterizzate da un type e da un payload che viene utilizzato dalle slice per modificare lo stato.
Per convenzione il nome delle azioni segue il formato: [nome Slice dalla quale viene catturata].[nome azione].

- **sidebar.toggleSidebarIsOpen**: azione emessa quando si apre o si chiude la sidebar relativa ad un prodotto presente all'interno della scena 3D.
  Payload: nessuno;
- **products.setSelectedColor**: azione emessa quando si seleziona nella sidebar un nuovo colore per un prodotto.
  Payload: PayloadSetSelectedColor {id:number, selectedColor: String};
  id: ID del prodotto a cui cambiare il colore.
  selectedColor: colore selezionato;
- **rayCaster.setLastProductPointed**: azione emessa per aggiornare l'ID che rappresenta l'ultimo oggetto puntato dal raycaster.
  Payload: {id:number}
  id: ID dell'ultimo prodotto puntato dal raycaster;
- **rayCaster.toggleRayCasterEnabled**: azione emessa per abilitare o disabilitare il raycaster.
  Payload: nessuno;
- **cart.addItems**: azione emessa per aggiungere uno o più prodotti al carrello.
  Payload: PayloadAddItems {id: number, price: number, quantity: number,selectedColor: String}
  id: ID del prodotto da aggiungere al carrello.
  price: prezzo del prodotto da aggiungere al carrello.
  quantity: quantità di prodotti da aggiungere al carrello.
  selectedColor: colore selezionato del prodotto da aggiungere al carrello.

  - **cart.removeItem**: azione emessa per rimuovere un prodotto dal carrello.
    Payload: {id:number}
    id: ID del prodotto da rimuovere al carrello.
  - **cart.removeAll**: azione emessa per rimuovere tutti i prodotti presenti nel carrello.
    Payload: nessuno;

- **Model components**

  Classi offerte dalle librerie utilizzate:

  - **Camera**: classe offerta dalla libreria three.js. Rappresenta l'utente all'interno dell'ambiente 3D. Modificando gli attributi di questa classe l'utente compie movimenti spaziali e può esplorare l'ambiente che lo circonda da prospettive diverse;
  - **Octree**: classe offerta dalla libreria three.js. L'Octree è una struttura dati ad albero utilizzata per la rappresentazione e la gestione di dati spaziali tridimensionali. In particolare, viene utilizzata per dividere lo spazio tridimensionale in regioni più piccole, suddividendo ciascuna regione in otto sotto-regioni. Utile per determinare con quale prodotto si sta interagendo o per rilevare le collisioni;
  - **Vector3**: classe offerta dalla libreria three.js. Rappresenta un vettore tridimensionale, ovvero una grandezza fisica caratterizzata da una direzione e da una lunghezza. Il vettore tridimensionale viene comunemente utilizzato per definire la posizione, la rotazione, la scala e la direzione degli oggetti all'interno di una scena 3D;
  - **Capsule**: classe offerta dalla libreria three.js. La classe Capsule offre la possibilità di impostare una forma di collisione personalizzata per l'oggetto rappresentato dalla capsula;
  - **Store**: Lo store in Redux è l'oggetto che tiene traccia dello stato dell'applicazione. Contiene il reducer che specifica come le azioni influenzano lo stato e offre metodi per accedere allo stato corrente, inviare azioni e registrare funzioni di callback per essere avvisati quando lo stato cambia;
  - **RootReducer**: componente Redux che combina tutti i reducers dell'applicazione in uno stato globale. Questa funzione viene passata allo store per gestire lo stato complessivo dell'applicazione.

  Classi definite dall'utente:

  - **CartItem**: classe che rappresenta un item all'interno del carrello. Quando viene aggiunto un prodotto al carrello viene creato un item che ne raccoglie le informazioni utili alla sua visualizzazione all'interno del carrello e all'acquisto;
  - **Product**: classe che rappresenta un prodotto acquistabile dall'utente;
  - **ModelGLTF**: classe che rappresenta il modello 3D di un prodotto acquistabile da caricare all'interno della scena;
  - **PayloadAddItems**: classe che rappresenta il payload da passare ad una AddItems action;
  - **Decoration**: classe che rappresenta una decorazione all'interno della scena;
  - **DecorationModelGLTF**: classe che rappresenta il modello 3D di una decorazione da caricare all'interno della scena;
  - **PayloadSetSelectedColor**: classe che rappresenta il payload da passare ad una SetSelectedColor action;
  - **Color**: classe che rappresenta un colore che può essere associato ad un ModelGLTF;
  - **Coordinate**: classe utilizzata per indicare un punto all'interno dell'ambiente 3D;
  - **Player**: classe con il compito di tenere traccia ed aggiornare le informazioni riguardanti lo stato dell'utente all'interno dell'ambiente 3D. Le informazioni principali di cui si occupa sono la posizione spaziale, la posizione della camera e la velocità con cui l'utente si muove.

- **UI React components**

  I seguenti componenti hanno il compito di visualizzare i dati dello stato che costituiscono l'interfaccia utente.

  - **UI**: contiene i componenti grafici che vanno a costituire l'interfaccia utente;
  - **Crosshair**: componente react-three-fiber che viene utilizzato per aiutare l'utente a puntare un determinato punto nello spazio 3D. Viene indicato con un punto o una croce al centro dello schermo ed usato come puntatore dall'utente;
  - **Cart**: rappresenta il carrello;
  - **CartItem**: rappresenta un prodotto all'interno del carrello. Contiene le informazioni necessarie alla visualizzazione e all'acquisto di un prodotto;
  - **ProductUI**: contiene i componenti dell'interfaccia utente per la visualizzazione del ProductInteractionPrompt e della sidebar;
  - **Sidebar**: contiene i componenti necessari a visualizzare i dettagli del prodotto con cui si sta iteragendo e i componenti con cui è possibile modificare le caratteristiche del prodotto e aggiungerne la quantità specificata al carrello;
  - **ColorSelector**: contiene i componenti necessari per compiere un'azione di selezione colore;
  - **SelectColorItem**: componente con cui è possibile compiere un'azione di selezione colore;
  - **ProductDetails**: componente per la visualizzazione dei dettagli di un prodotto;

- **3D React components** I seguenti componenti hanno il compito di visualizzare i dati dello stato contenuti all'interno del canvas sul quale viene renderizzata la scena 3D.

  - **Provider**: Il Provider nel contesto Redux è un componente che consente di rendere lo store accessibile a tutti i componenti dell'applicazione. Viene 'avvolto' intorno al componente principale dell'applicazione App e accetta lo store come prop;
  - **App**: componente radice dell'applicazione che contiene tutti gli altri componenti. È 'avvolto' dal Provider di Redux per fornire accesso allo store a tutti i componenti figli;
  - **Canvas**: componente react-three-fiber che comprende gli elementi grafici che vanno a costituire la scena 3D. Canvas si occupa di creare un nuovo canvas HTML e di associarlo a una nuova istanza di THREE.WebGLRenderer, che viene utilizzata per renderizzare la scena. Fornisce una camera e una scena con una serie di props opzionali utili alla configurazione dell'ambiente 3D;
  - **Scene**: componente react-three-fiber che rappresente l'ambiente 3D. Si tratta di uno spazio virtuale 3D in cui gli oggetti vengono posizionati, orientati e illuminati;
  - **PointerLock**: componente react-three-fiber che consente di "bloccare" il puntatore del mouse all'interno di un elemento specifico della pagina web. Permette quindi di controllare la camera e di disattivarne i controlli quando necessario (ad esempio quando si apre un menu di interazione con un prodotto);
  - **Environment**: componente react-three-fiber utilizzato per creare un'ambientazione configurata secondo impostazioni standard settabili;
  - **Map**: contiene i componenti che costituiscono la scena.
  - **FlashLight**: rappresenta una luce direzionale utilizzata dall'utente per illuminare l'ambiente 3D;
  - **Player**: rappresenta l'utente all'interno dell'ambiente 3D;
  - **Lights**: rappresenta le luci all'interno della scena;
  - **Model**: rappresenta un prodotto all'interno dell'ambiente 3D; I modelli vengono caricati all'interno della scena specificandone il path del modello .gltf, la posizione in cui si vogliono collocare e le caratteristiche del prodotto legato al modello;

  - **Decorations**: modello .gltf che si differisce dal modello di un prodotto per il fatto che non è modificabile e con il quale non si può interagire;
  - **RayCaster**: il raycaster genera un raggio virtuale che parte dalla posizione del puntatore del mouse e attraversa lo schermo fino a raggiungere un oggetto nella scena, se presente. È possibile quindi utilizzare questa informazione per eseguire azioni in risposta all'interazione dell'utente.

2.2 Design pattern architetturale determinato dalle tecnologie adottate

2.2.1 Redux-Toolkit

I componenti che costituiscono l'architettura utilizzata seguono il pattern offerto dalla libreria Redux-Toolkit.

Redux-Toolkit è pensato per integrarsi con React e il principale vantaggio che offre è quello di poter gestire i dati condivisi tra i componenti React in modo centralizzato semplificando la gestione dello stato globale dell'applicazione.

I componenti che formano l'architettura di Redux-Toolkit sono:

- **Store**: componente che contiene lo stato globale dell'applicazione.

  All'avvio dell'applicazione viene configurato utilizzando RootReducer e i componenti che utilizzano lo stato globale fanno il subscribe allo *store* in modo da venire renderizzati ogni volta che un dato di interesse cambia valore. Questo modo di operare può essere visto come un pattern *Observer* in cui lo *store* è il *Subject* e gli *Observers* sono i componenti React che hanno fatto il subscribe allo *store*;

- **RootReducer**: componente utilizzato per configurare lo store combinando le slice;

- **Slice**: componente che contiene un proprio stato che rappresenta una porzione dello stato globale dell'applicazione, i *reducer* che operano su tale stato e i *selector* per consentire ai suoi client il reperimento dei dati.

  Per definire una *slice* è buona norma raggruppare i dati in modo che siano legati da un sottoinsieme di funzionalità offerte dal sistema che lavorano su dati comuni;

- **Reducer**: componente che riceve come parametri uno stato iniziale (*InitialState*) e una *action* (composta da un type e un payload) e restituisce lo stato dopo aver operato sui dati.

  React-Toolkit gestisce le chiamate ai *reducer* in seguito ai dispatch delle *action* che avvengono specificando solamente l'oggetto che rappresenta il payload;

- **Actions**: oggetto composto da un type e da un payload di cui viene effettuato il dispatch quando opportuno.

  Il payload è un oggetto che contiene i dati da passare al *reducer* che catturerà l'*action*;

- **InitialState**: componente che contiene i dati di una *slice* su cui essa opera.

  Importante precisare che Redux-Toolkit utilizzando la libreria immer gestisce anche l'immutabilità dei dati in modo che i reducer restituiscano delle copie dello stato in modo che esso non possa venire modificato dall'esterno e utilizzato in modo improrio.

  L'unico modo per modificare i dati dello stato globale è quindi con il dispatch di un'*action*;

- **Selector**: funzione che prende lo stato corrente di una *slice* come argomento e ritorna un sottoinsieme specifico del suo stato. In altre parole, un *selector* consente di 'selezionare' una parte specifica dello stato in modo da poterla utilizzare in modo isolato all'interno di un componente React.

2.2.2 React-three-fiber

Questa libreria fornisce un 'punto d'incontro' tra React (libreria javascript per la creazione di interfacce utente) e Three.js (libreria usata per la modellazione dell'ambiente 3D) semplificando la creazione dei componenti da inserire all'interno dell'ambiente 3D.

React-three-fiber rende la scrittura del codice dichiarativa creando dei componenti React 'preconfezionati' che rappresentano i componenti 3D. Questi componenti sono personalizzabili modificandone le caratteristiche tramite le props di React.

Un esempio è il componente *Canvas* che fornisce con un'unica dichiarazione la *scene* e la *camera* con una configurazione standard adatta alla maggior parte dei casi di utilizzo. Per inserire i componenti all'interno dell'ambiente è sufficiente dichiararli come figli del componente *scene*.

2.3 Architettura logica

Per facilitare la lettura dei diagrammi delle classi è stato scelto di organizzarli per feature in modo che ogni diagramma rappresenti i componenti che permettono l'implementazione di funzionalità specifiche. Sono presenti dei diagrammi che non seguono questa convenzione che sono utili per avere una visione generale sulle dipendenze di alcuni componenti.

I diagrammi prodotti che rappresentano funzionalità specifiche sono:

- **CartFeaturesDiagram**: include i componenti che svolgono le funzioni riguardanti il carrello;
- **PlayerFeaturesDiagram**: include i componenti che svolgono le funzioni riguardanti le interazioni dell'utente con l'ambiente 3D;
- **SidebarFeaturesDiagram**: include i componenti che svolgono le funzioni riguardanti la side-bar;
- **InterfaceFeaturesDiagram**: include i componenti necessari per il corretto aggiornamento dell'interfaccia utente.

I diagrammi prodotti che forniscono una visione generale delle dipendenze tra componenti sono:

- **StoreDiagram**: include lo *store* e le slice che compongono lo stato globale dell'applicazione;
- **SkyLevelReactComponentsHierarchy**: contiene i macrocomponenti React che costituiscono l'applicazione;
- **UIReactComponentsHierarchy**: contiene i componenti che costituiscono l'interfaccia utente;
- **CanvasReactComponentsHierarchy**: contiene i componenti che costituiscono gli elementi dell'ambiente 3D.

2.3.1 CartFeaturesDiagram

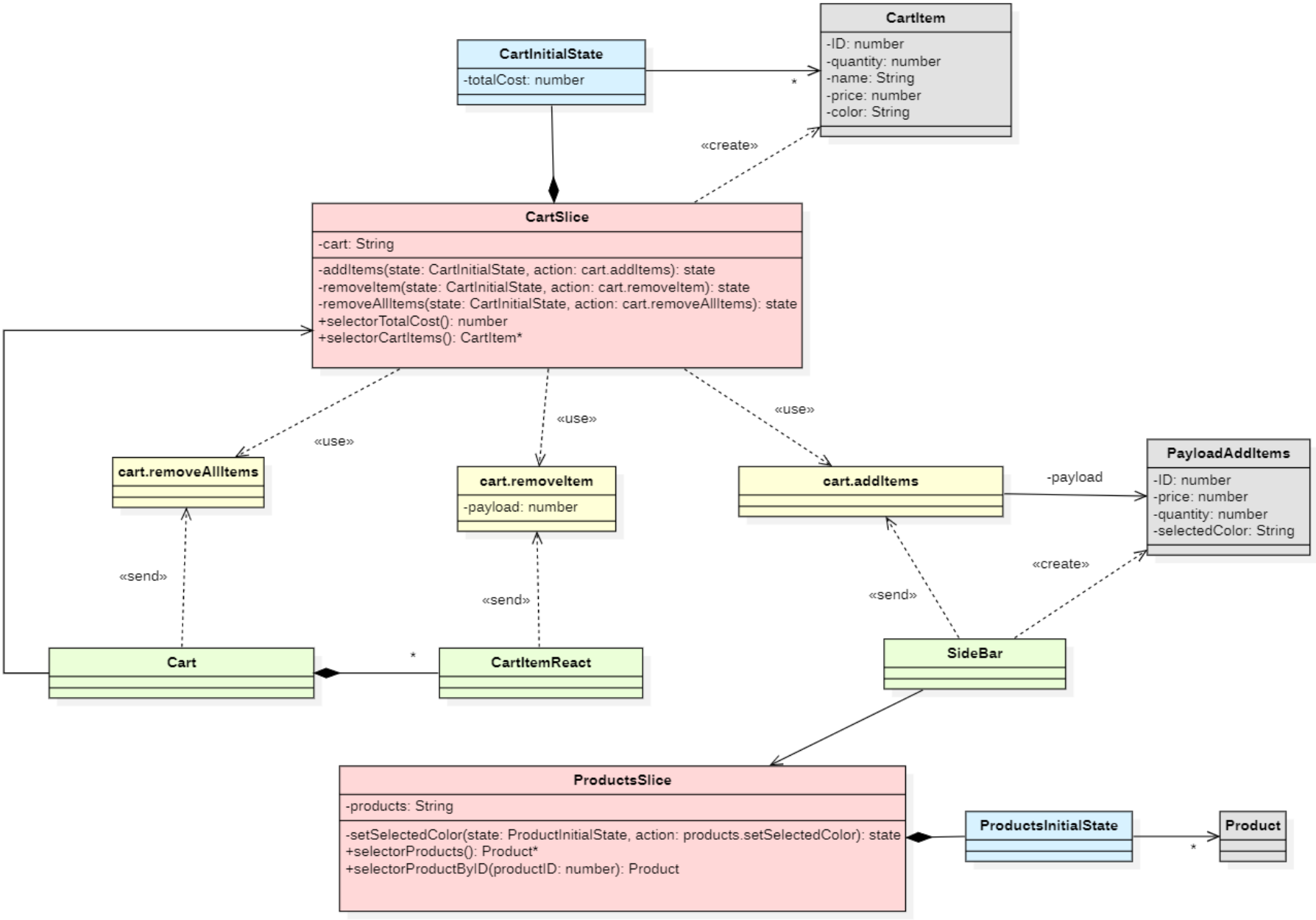

Figura 1: UML delle classi CartFeaturesDiagram.
Legenda: [*Slices*: rosso] - [*Actions*: giallo] - [*Model classes*: grigio] - [*Initial states*: azzurro] - [*UI React components*: verde]

Descrizione del diagramma: CartFeaturesDiagram include i componenti che svolgono le funzioni riguardanti il carrello.

- **CartSlice**

  Dipendenze:

  - *CartInitialState* (composizione): si occupa della costruzione e distruzione dell'istanza di CartInitialState che non viene condivisa con altri componenti in quanto CartSlice si occupa in modo esclusivo della sua gestione.
  - *CartItem* (dipendenza semplice <create>): si occupa della costruzione dei CartItem da inserire all'interno della lista items di CartInitialState;
  - *cart.addItems* (Dipendenza semplice <use>): cattura un'istanza di cart.addItems e il reducer ne utilizza il payload per aggiungere un item ad items in CartInitialState. Se le caratteristiche specificate nel payload corrispondono a quelle di un item presente ne incrementa la quantità del valore specificato altrimenti viene aggiunto un nuovo item con le caratteristiche e quantità specificate;
  - *cart.removeItem* (Dipendenza semplice <use>): cattura un'istanza di cart.removeItem e il reducer ne utilizza il payload per decrementare la quantità dell'item specificato. Se la quantità dopo il decremento risulta essere uguale a 0 l'item viene rimosso da items in CartInitialState;
  - *cart.removeAllItems* (Dipendenza semplice <use>): cattura un'istanza di cart.removeAllItems e chiama il reducer che svuota la lista items presente in CartInitialState ed imposta il totalCost a 0.

  Interazioni:

  - *CartInitialState*: viene modificato in base alle actions catturate dal reducer della slice.

  Actions catturate:

  - *cart.addItems*: utilizzata dal reducer per chiamare *addItems*;
  - *cart.removeItem*: utilizzata dal reducer per chiamare *removeItem*;
  - *cart.removeAllItems*: utilizzata dal reducer per chiamare *removeAllItems*.

- **CartInitialState**

  Dipendenze:

  - *CartItem* (Associazione): contiene la lista degli item che rappresentano i prodotti presenti nel carrello.

- **Cart**

  Dipendenze:

  - *CartItem* (Composizione): crea le istanze di CartItem durante il rendering. Cart gestisce interamente il ciclo di vita delle istanze di un CatrItem che quindi per composizione non può essere condiviso con altri componenti.
  - *CartSlice* (Associazione): possiede come attributo implicito (dovuto a Redux) un'istanza di CartSlice.
  - *cart.removeAllItems* (Dipendenza semplice <send>): crea ed emette un'istanza dell'azione cart.removeAllItems.

  Interazioni:

- *CartSlice*: chiama i selettori *selectorCartItems* e *selectorTotalCost* per reperire gli items e il costo totale dei prodotti allo scopo di compiere il rendering dei suoi componenti figli.

Action emesse:

- *cart.removeAllItems*: emessa quando l'utente rimuove tutti i prodotti dal carrello.

- **CartItemReact**

  Dipendenze:

  - *cart.removeItem* (Dipendenza semplice <send>): crea ed emette un'istanza dell'azione cart.removeItem.

  Action emesse:

  - *cart.removeItem*: emessa quando l'utente rimuove il prodotto corrispondente all'item dal carrello.

- **Sidebar**

  Dipendenze:

  - *ProductsSlice* (Associazione): possiede come attributo implicito (dovuto a Redux) un'istanza di ProductsSlice.
  - *PayloadAddItems* (Dipendenza semplice <create>): crea il payload da fornire alla action cart.addItems.
  - *cart.addItems* (Dipendenza semplice <send>): crea ed emette un'istanza dell'azione cart.addItems.

  Interazioni:

  - *ProductsSlice*: chiama il selettore *selectorProductByID* per reperire le informazioni del prodotto allo scopo di compiere il rendering dei suoi componenti figli.

2.3.2 PlayerFeaturesDiagram

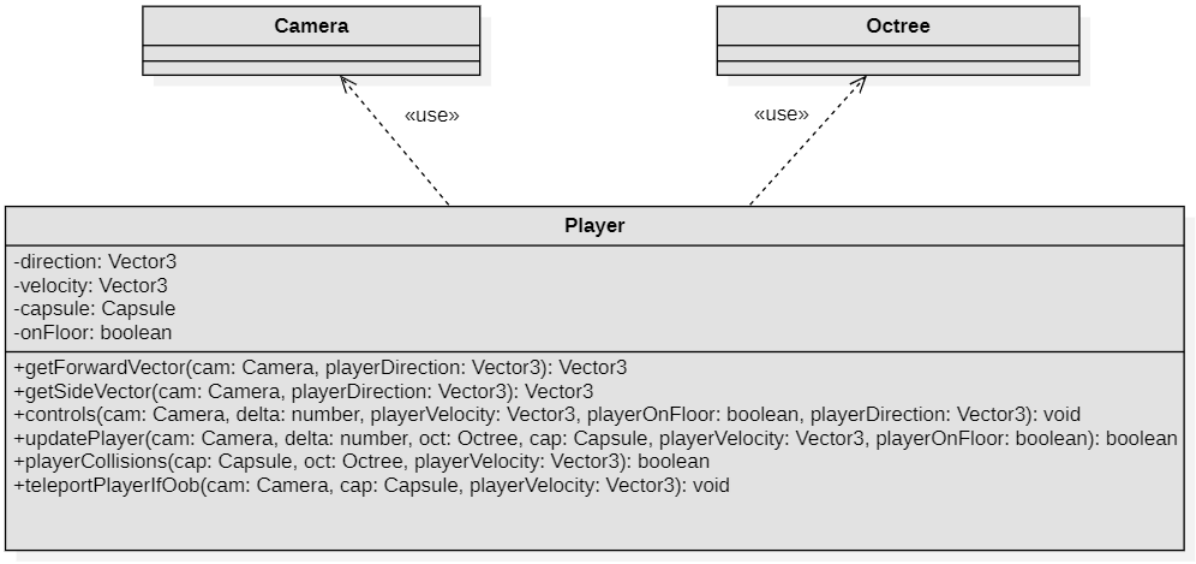

Figura 2: UML delle classi PlayerFeaturesDiagram.
Legenda: [*Model classes*: grigio]

Descrizione del diagramma: PlayerFeaturesDiagram include i componenti che svolgono le funzioni riguardanti le interazioni dell'utente con l'ambiente 3D.

- **Player**

  Dipendenze:

  - *Camera* (Dipendenza semplice <use>): utilizza la camera del canvas in cui viene renderizzata l'applicazione per aggiornare la scena 3D;
  - *Octree* (Dipendenza semplice <use>): usa l'Octree della scena per individuare le collisioni dell'utente con gli oggetti all'interno dell'ambiente 3D.

  Interazioni:

  - *Camera*: sposta la camera a seconda dei comandi dell'utente;
  - *Octree*: quando rileva una collisione la velocità del player viene impostata a 0.

2.3.3 SidebarFeaturesDiagram

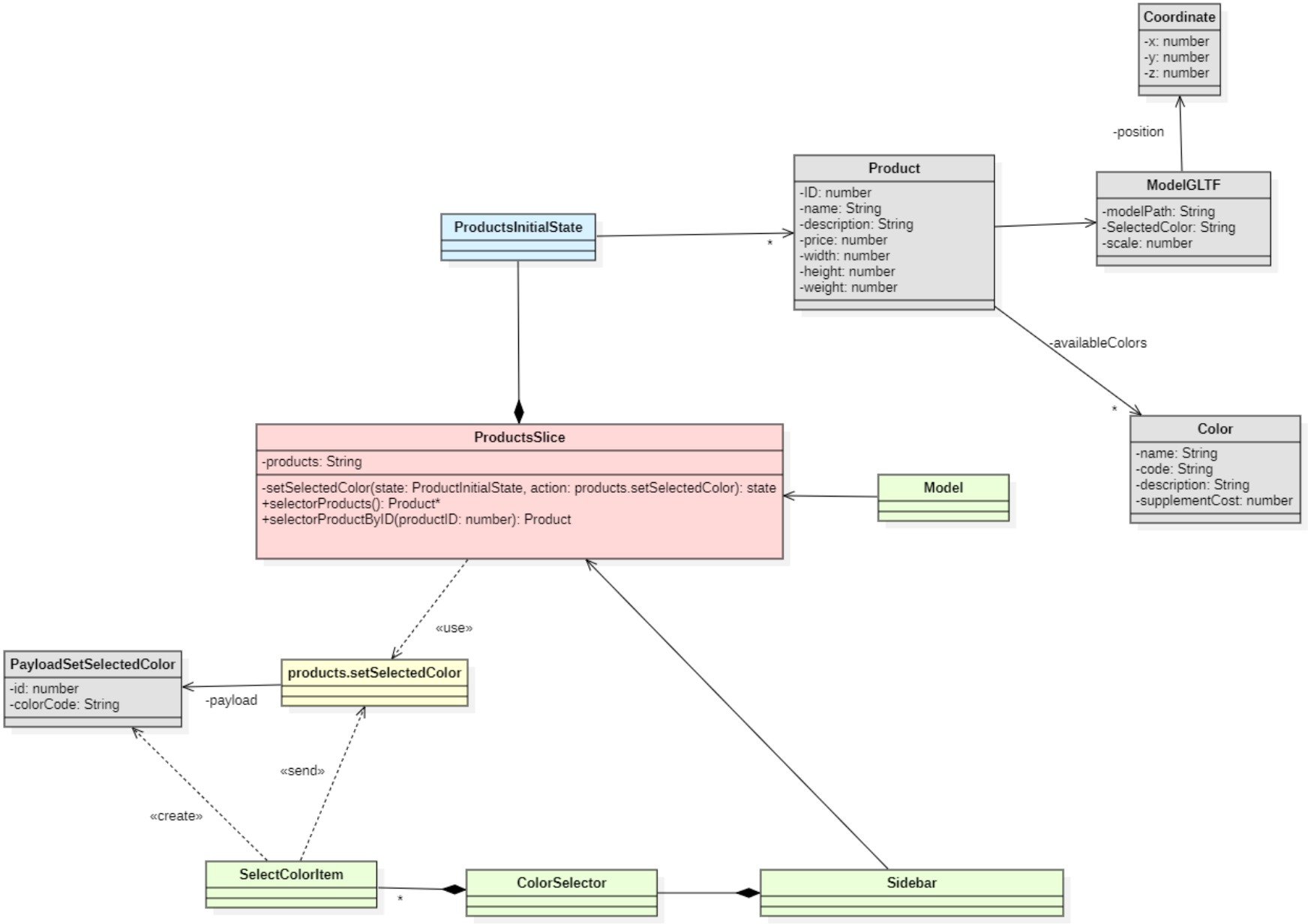

Figura 3: UML delle classi SidebarFeaturesDiagram.
Legenda: [*Slices*: rosso] - [*Actions*: giallo] - [*Model classes*: grigio] - [*Initial states*: azzurro] - [*UI React components*: verde]

Descrizione del diagramma: SidebarFeaturesDiagram include i componenti che svolgono le funzioni riguardanti la side-bar.

- **ProductsSlice**

  Dipendenze:

  - *ProductsInitialState* (Composizione): si occupa della costruzione e distruzione dell'istanza di ProductsInitialState che non viene condivisa con altri componenti in quanto ProductsSlice si occupa in modo esclusivo della sua gestione;
  - *products.setSelectedColor* (Dipendenza semplice <use>): cattura un'istanza di products.setSelectedColor e il reducer ne utilizza il payload per modificare l'attributo selectedColor del prodotto che ha ID specificato.

  Actions catturate:

  - *products.setSelectedColor*: utilizzata dal reducer per chiamare *setSelectedColor*.

- **ProductsInitialState**

  Dipendenze:

  - *Product* (Associazione): ha come attributo la lista di tutti i prodotti disponibili.

- **Sidebar**

  Dipendenze:

  - *ProductsSlice* (Associazione): possiede come attributo implicito (dovuto a Redux) un'istanza di ProductsSlice.

  Interazioni:

  - *ProductsSlice*: chiama selectorProductByID per ricavare le informazioni necessarie a fare il rendering dei suoi componenti figli.

- **ColorSelector**

  Dipendenze:

  - *SelectColorItem* (Composizione): contiene gli item che rappresentano i colori selezionabili del prodotto.

- **SelectColorItem**

  Dipendenze:

  - *PayloadSetSelectedColor* (Dipendenza semplice <create>): crea il payload necessario all'azione products.setSelectedColor.
  - *products.setSelectedColor* (Dipendenza semplice <send>): emette products.setSelectedColor.

  Action emesse:

  - *products.setSelectedColor*: emessa quando viene selezionato un colore diverso per il prodotto.

- **Product**

  Dipendenze:

  - *ModelGLTF* (Associazione): corrisponde al modello 3D che viene visualizzato all'interno dell'ambiente 3D;
  - *Color* (Associazione): l'attributo availableColors indica i colori disponibili per il prodotto.

2.3.4 InterfaceFeaturesDiagram

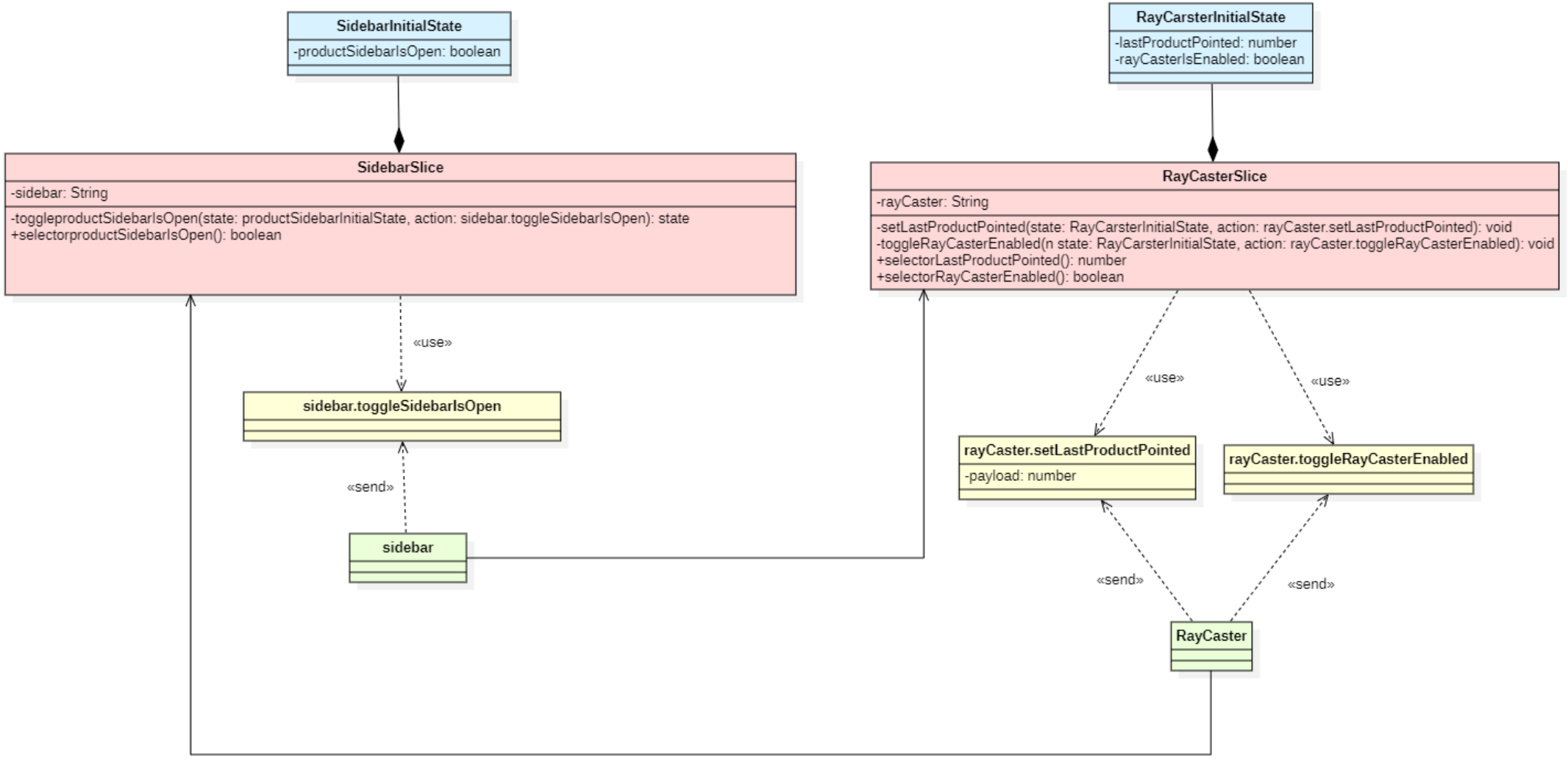

Figura 4: UML delle classi InterfaceFeaturesDiagram.
Legenda: [*Slices*: rosso] - [*Actions*: giallo] - [*Model classes*: grigio] - [*Initial states*: azzurro] - [*UI React components*: verde]

Descrizione del diagramma: InterfaceFeaturesDiagram include i componenti necessari per il corretto aggiornamento dell'interfaccia utente.

- **SidebarSlice**

  Dipendenze:

  - *SidebarInitialState* (Composizione): si occupa della costruzione e distruzione dell'istanza di SidebarInitialState che non viene condivisa con altri componenti in quanto SidebarSlice si occupa in modo esclusivo della sua gestione.
  - *sidebar.toggleSidebarIsOpen* (Dipendenza semplice <use>): cattura un'istanza di sidebar.toggleSidebarIsOpen per commutare il valore dell'attributo productSidebarIsOpen in SidebarInitialState.

  Actions catturate:

  - *sidebar.toggleSidebarIsOpen*: utilizzata dal reducer per chiamare *toggleSidebarIsOpen*.

- **SideBar**

  Dipendenze:

  - *RayCasterSlice* (Associazione): utilizza selectorLastProductPointed per ricavare l'ID dell'ultimo prodotto puntato dal rayCaster. Se il raycaster non sta puntando a nessun prodotto il selettore ritorna null.
  - *sidebar.toggleSidebarIsOpen* (Dipendenza semplice <send>): emette un'azione sidebar.toggleSidebarIsOpen.

  Action emesse:

  - *sidebar.toggleSidebarIsOpen*: emessa quando l'utente apre o chiude la sidebar

- **RayCasterSlice**

  Dipendenze:

  - *RayCarsterInitialState* (Composizione): si occupa della costruzione e distruzione dell'istanza di RayCarsterInitialState che non viene condivisa con altri componenti in quanto RayCasterSlice si occupa in modo esclusivo della sua gestione.
  - *rayCaster.setLastProductPointed* (Dipendenza semplice <use>): cattura un'istanza di rayCaster.setLastProductPointed per aggiornare il valore dell'attributo lastProductPointed con l'ID dell'ultimo oggetto puntato dal raycaster.
  - *rayCaster.toggleRayCasterEnabled* (Dipendenza semplice <use>): cattura un'istanza di sidebar.toggleSidebarIsOpen per commutare il valore dell'attributo rayCasterIsEnabled in SidebarInitialState.

  Action catturate:

  - *rayCaster.setLastProductPointed*: utilizzata dal reducer per chiamare *setLastProductPointed*.
  - *rayCaster.toggleRayCasterEnabled*: utilizzata dal reducer per chiamare *toggleRayCasterEnabled*.

- **RayCaster**

  Dipendenze:

  - *SidebarSlice* (Associazione): utilizza selectorproductSidebarIsOpen.
  - *rayCaster.setLastProductPointed* (Dipendenza semplice <send>): emette un'azione rayCaster.setLastProductPointed.

– *rayCaster.toggleRayCasterEnabled* (Dipendenza semplice <send>): emette un'azione rayCaster.toggleRayCasterEnabled.

Interazioni:

– *SidebarSlice*: chiama selectorproductSidebarIsOpen per verificare se la sidebar è aperta o chiusa in modo da abilitare o disabilitare il raycaster a seconda del caso.

Action emesse:

– *rayCaster.setLastProductPointed*: emessa quando il raycaster punta un prodotto all'interno dell'ambiente 3D.

– *rayCaster.toggleRayCasterEnabled*: emessa quando viene aperta la sidebar per disabilitare i raycaster.

2.3.5 StoreDiagram

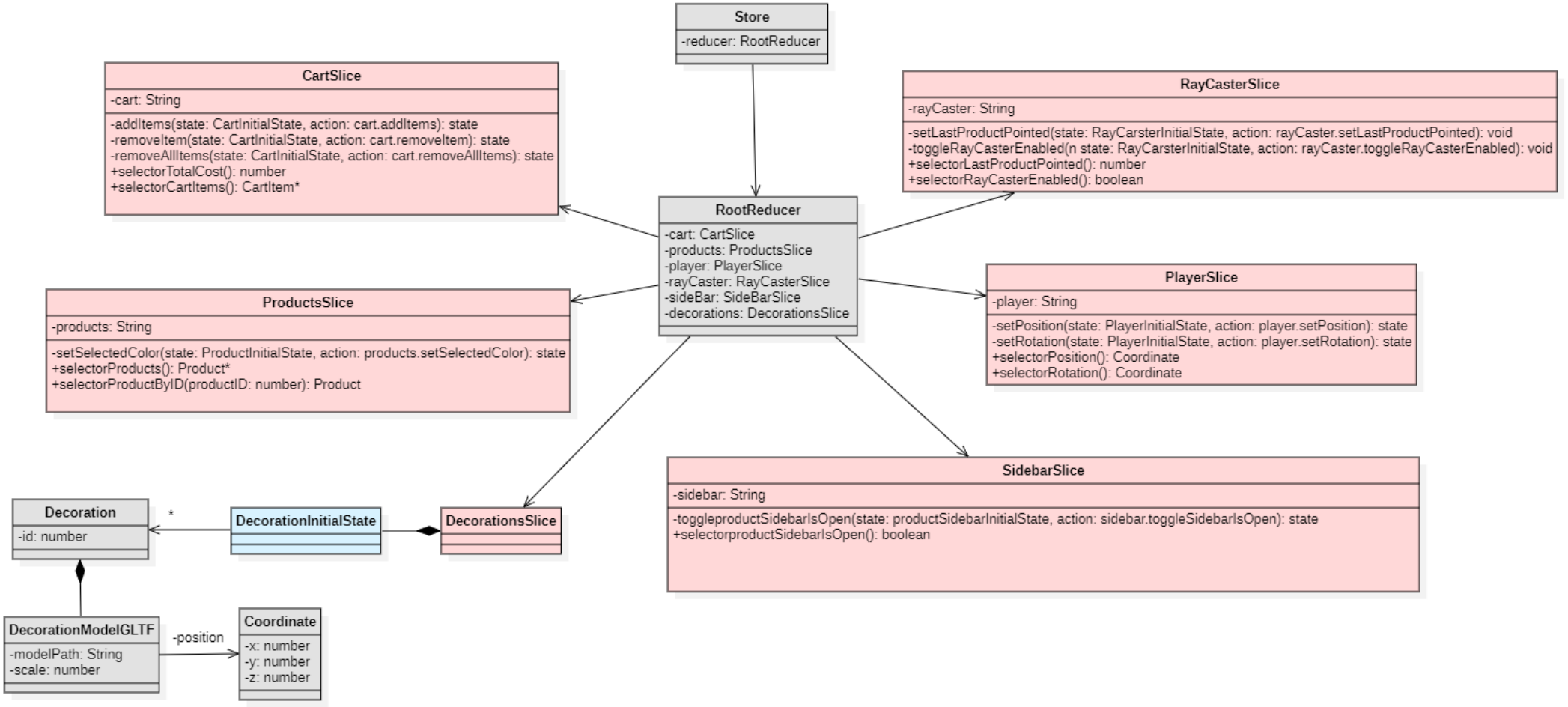

Figura 5: UML delle classi StoreDiagram.
Legenda: [*Slices*: rosso] - [*Model classes*: grigio] - [*Initial states*: azzurro]

Descrizione del diagramma: Include lo *store* e le slice che compongono lo stato globale dell'applicazione.

- **Store**

  Dipendenze:

  - *RootReducer* (Associazione): ha un attributo RootReducer che permette di combinare piú slice con cui lo store interagisce per modificare lo stato globale dell'applicazione.

- **RootReducer**

  Dipendenze:

  - *CartSlice* (Associazione): ha un attributo CartSlice per offrire allo store l'accesso alla slice.
  - *ProductsSlice* (Associazione): ha un attributo ProductsSlice per offrire allo store l'accesso alla slice.
  - *DecorationsSlice* (Associazione): ha un attributo DecorationsSlice per offrire allo store l'accesso alla slice.
  - *SidebarSlice* (Associazione): ha un attributo SidebarSlice per offrire allo store l'accesso alla slice.
  - *PlayerSlice* (Associazione): ha un attributo PlayerSlice per offrire allo store l'accesso alla slice.
  - *RayCasterSlice* (Associazione): ha un attributo RayCasterSlice per offrire allo store l'accesso alla slice.

- **DecorationsSlice**

  Dipendenze:

  - *DecorationInitialState*: si occupa della costruzione e distruzione dell'istanza di DecorationInitialState che non viene condivisa con altri componenti in quanto DecorationsSlice si occupa in modo esclusivo della sua gestione.

- **DecorationInitialState**

  Dipendenze:

  - *Decoration*: contiene la lista delle decorazioni presenti all'interno dell'ambiente 3D.

- **Decoration**

  Dipendenze:

  - *DecorationModelGLTF* (Composizione): costruisce l'istanza del modello che rappresenta la decorazione.

2.3.6 SkyLevelReactComponentsHierarchy

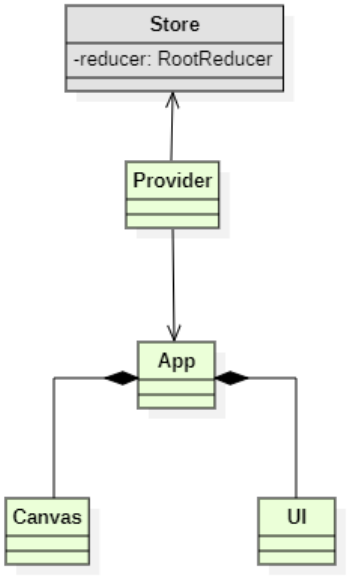

Figura 6: UML delle classi SkyLevelReactComponentsHierarchy.
Legenda: [*Model classes*: grigio] - [*UI React components*: verde]

Descrizione del diagramma: Include i macrocomponenti React che costituiscono l'applicazione.

- **Provider**

  Dipendenze:

  - *Store* (Associazione): tutti i componenti figli fanno riferimento allo store passato come props al componente Provider.
  - *App* (Associazione): contiene i componenti che costituiscono l'applicazione che comunicano con lo store.

- **App**

  Dipendenze:

  - *IU* (Composizione): crea il componente figlio che contiene l'interfaccia utente.
  - *Canvas* (Coomposizione): crea il componente figlio che contiene l'ambiente 3D.

2.3.7 UIReactComponentsHierarchy

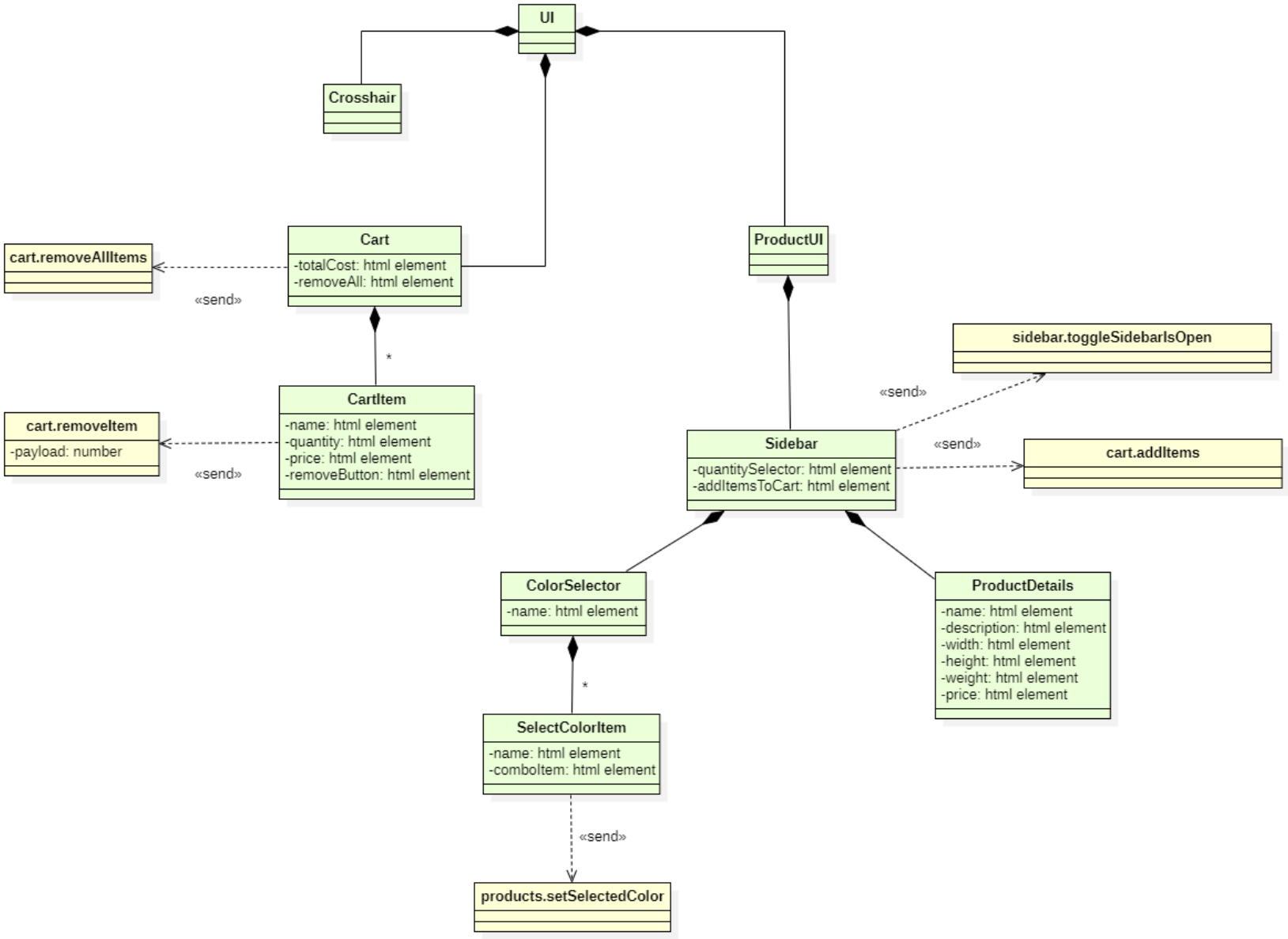

Figura 7: UML delle classi UIReactComponentsHierarchy. **Legenda**: [*Actions*: giallo] - [*UI React components*: verde]

Descrizione del diagramma: Contiene i componenti che costituiscono l'interfaccia utente. Lo schema illustra come è organizzato l'albero delle discendenze tra componenti dell'interfaccia utente.

2.3.8 CanvasReactComponentsHierarchy

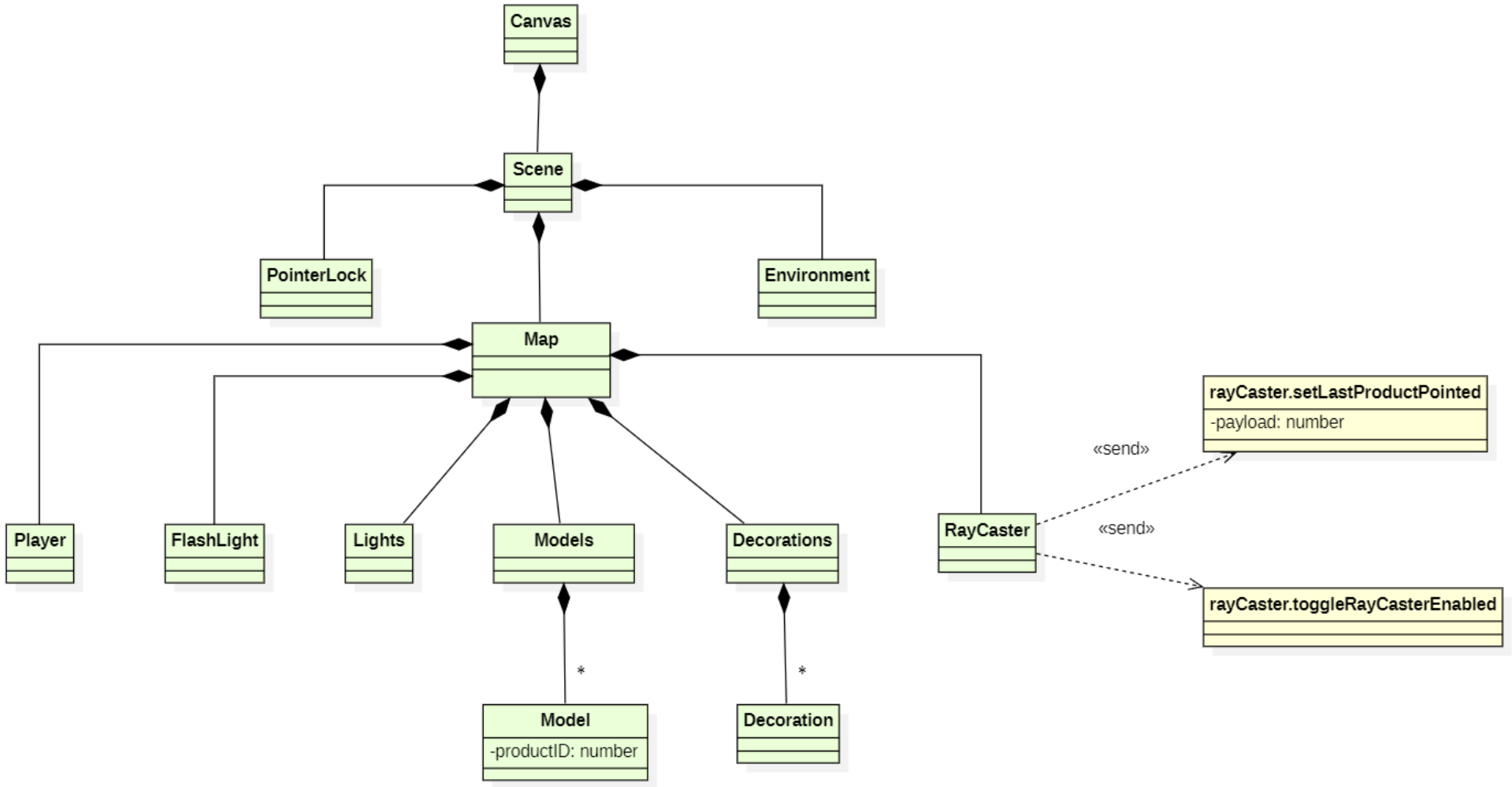

Figura 8: UML delle classi CanvasReactComponentsHierarchy.
Legenda: [*Actions*: giallo] - [*UI React components*: verde]

Descrizione del diagramma: Contiene i componenti che costituiscono l'ambiente 3D. Lo schema illustra come è organizzato l'albero delle discendenze tra componenti del canvas.

2.4 Architettura di deployment

L'obiettivo è di creare un pacchetto di file statici che possa essere eseguito in un browser web senza necessità di un server di sviluppo. Questo processo avviene in due fasi principali: la *fase di build* e la *fase di produzione*.

2.4.1 Build

Nella fase di build, usiamo Vite come strumento di build. Vite è responsabile della trasformazione codice sorgente JavaScript in un formato che può essere facilmente eseguito dai browser. Durante il processo di build Vite esegue vari task tra cui la trasformazione codice JSX in JavaScript, il bundling dei nostri file JavaScript in un unico file e l'ottimizzazione di questo file per la produzione. Vite compila e ottimizza i file del progetto, inclusi tutti i file di codice sorgente e i file di risorse statiche come CSS, immagini e modelli, in un pacchetto di file ottimizzati per la produzione. Questo pacchetto viene poi scritto in una directory di output, chiamata dist. Per avviare il processo di build, usiamo il comando npm run build. Questo comando esegue un task definito nel nostro file package.json che avvia Vite.

2.4.2 Produzione

Una volta che il processo di build è completo, l'applicazione è pronta per la fase di produzione. In questa fase, il pacchetto di file statici generato da Vite può essere caricato su un server web. Questi file sono poi pronti per essere serviti da un server web a un client per l'esecuzione nel browser.