UNIVERSITA' DEGLI STUDI ROMA TRE

Facoltà di Ingegneria

Corso di Laurea Magistrale in Ingegneria Informatica

Modellazione e navigazione di un ambiente 3D semi-realistico su web

Relatore : Prof. Paoluzzi Alberto

Candidato : Kwiatkowski Marcin

Matricola : 404033

Co-relatore : Marino Enrico

Spini Federico

Anno Accademico 2013/2014

Introduzione

Il progetto, in collaborazione con la società Sogei S.p.A., ha riguardando la modellazione di un ambiente 3D semi-realistico (stanze definite da mura, pavimentazione e soffitto, illuminate da luci in modo da avere zone più illuminate e meno, contenenti diversi oggetti) e la successiva poi navigazione (automatica secondo un percorso e non) del medesimo.

La fase iniziale é stata dedicata interamente al capire in cosa consisteva WebGL ([2] [3] [4] [5]) e allo studio della sua libreria grafica Three.js ([1] [2] [6] [7] [8] [9]).

Nella seconda fase l' attenzione é stata posta sulla modellazione di oggetti complessi e alla loro consecutiva importazione nella scena creata con Three.js; in particolare questo obbiettivo é stato raggiunto studiando e usando il programma Blender ([10] [11]).

Dopo aver acquisito le conoscenze necessarie, é stata fatta l' analisi della realtà d' interesse (riguardante la creazione delle varie stanze): sono stati scritti i casi d' uso ([12]) e in base a questi, sono stati creati (usando UML – Unified Modelling Language ([12])) il modello di dominio, i diagrammi di sequenza di sistema relativi (SSD – System Sequence Diagrams ([12])) e i contratti. Questo tipo di metodologia si é rivelato utile e prezioso per la comprensione delle funzionalità e ne ha beneficiato poi la fase di progettazione e conseguentemente la scrittura del codice, ovvero la realizzazione del modello 3D e la sua navigazione.

Infine si é passati alla fase di progettazione in cui sono stati creati i diagrammi di interazione e il diagramma delle classi (DCD – Design Class Diagram ([12])) che hanno permesso uno studio più approfondito della realtà d' interesse; in alcuni casi, seguendo riflessioni e linee guida sulla progettazzione (come i pattern GRASP - General Responsibility Assignment Software Patterns ([12])) si é ritenuto opportuno rivedere, aggiornare e modificare alcuni elaborati che invece durante la fase di analisi erano state ritenute valide (sono stati quindi modificati diagrammi e contratti creati nella fase di

analisi).

La pianificazione di un'attenta analisi dei requisiti e di una buona progettazione ha reso possibile la semplificazione della fase di sviluppo del software. Gli elaborati scritti in precedenza si sono rivelati determinanti per aver permesso in molti casi, una semplice traduzione dei diagrammi e dei modelli in classi software e metodi. Al termine della parte di sviluppo é seguita una breve fase di testing.

La realizzazione del progetto é stata eseguita seguendo i principi OOA/D (Object-Oriented Analysis/Design ([12])) e applicati tramite il processo iterativo UP agile (Unified Process; ([12])). Durante lo studio del caso l' approccio seguito é stato il domain model, in cui la maggior parte degli oggetti incapsula sia dati che operazioni, ripartendosi le responsabilità del sistema.

Indice

Elenco immagini

31 {07} ssd2

32 {08} ssd3

39 {09} modelloDominioFinale

41 {10} dc1

41 {11} dc1_1

42 {12} dc1_2

43 {13} dc1_3

43 {14} dc1_4

44 {15} dc1_5

44 {16} dc1_6

46 {17} dc2_1

46 {18} dc2_2

47 {19} dc2_3

47 {20} dc2_4

48 {21} dc3

49 {22} dcd

Capitolo 1 : Introduzione a Three.js

1.1 : Introduzione

Con il progredire delle tecnologie, le innovazioni appaiono anche su WWW (World Wide Web): in particolare quelle riguardanti la grafica 3D. Fino a qualche anno fa, la visualizzazione e manipolazione di oggetti 3D (sia nei giochi che in programmi specifici di grafica) era resa possibile solo tramite apposito software per pc, sviluppato nella maggior parte dei casi, con linguaggio C; nel 2006 tuttavia, arriva l' innovazione che porterà la grafica anche sul web: Vladimir Vukićević mostra un primo prototipo di canvas 3D funzionante sul browser Mozilla; tra il 2009 e il 2011, a seguito degli esperimenti 3D di Vukićević, nasce la versione 1.0 di WebGL (Web Graphics Library) ([2] [3] [4]). Ma che cos' é? WebGL é una API (Application Programming Interface) di JavaScript di basso livello che permette l' esecuzione di grafiche 2D e 3D, su browser compatibili, senza che venga richiesto all' utente l' uso/installazione di plug-ins.
I programmi WebGL, usano gli elementi canvas HTML (che permettono di avere piena integrazione con tutte le interfacce DOM – Document Object Model; questo significa che WebGL può essere usato con tutti i linguaggi DOM-compatibili, come Java o in alcuni casi anche Objective C) e sono composti da un *control code* che viene scritto in JavaScript e da un *shader code* (usa lo shading language di OpenGL, GLSL) che invece viene eseguito dalla GPU (Graphics Processing Unit) del pc dell' utente (questo fatto ovviamente introduce alcuni spunti di riflessione, come ad esempio la vulnerabilità ad un attacco da parte di un hacker direttamente sulla GPU; resta tuttavia solo come cosa su cui riflettere, ma non sarà argomento trattato in questa tesi).

Programmare direttamente in WebGL é un processo molto complesso e non privo di errori. Per questo motivo si é scelto di usare un framework: ne esistono molti, alcuni più adatti alla creazione dei giochi (come GLGE), altri sono più adatti alla visualizzazione dinamica di dati, globi, mappe (come Cesium), altri ancora per la rappresentazione grafica di qualità di diagrammi matematici (come MathBox), etc. Per

il progetto é stato scelto Three.js che non solo é una libreria molto generale e non specializzata come quelle sopra citate, ma conta su una vasta comunità e popolarità (su GitHub – https://github.com/mrdoob/three.js/): questo dato é molto importante in quanto ci dice che Three.js non solo viene più apprezzato degli altri framework, ma anche che é in continuo sviluppo e la presenza di bug ha un impatto minore in quanto molte più persone si adoperano a migliorare il codice esistente. Guardando da un altro punto di vista, tale scelta può anche essere giustificata nel seguente modo: una vasta comunità non solo porta più esempi su cui fare pratica, e conseguentemente un apprendimento più veloce di come fare le cose, ma anche, in presenza di bug, un più facile confronto con altri che hanno avuto lo stesso problema ma hanno risolto in maniera diversa.

1.2 : Creazione della scena

Prima di iniziare ad inserire i modelli e le animazioni, bisogna capire come creare la scena in cui questi verranno visualizzati. Iniziamo con un documento html in cui inseriamo nel *<head>* le dimensioni della canvas, ovvero dell' ambiente in cui verranno visualizzati/animati i vari oggetti:

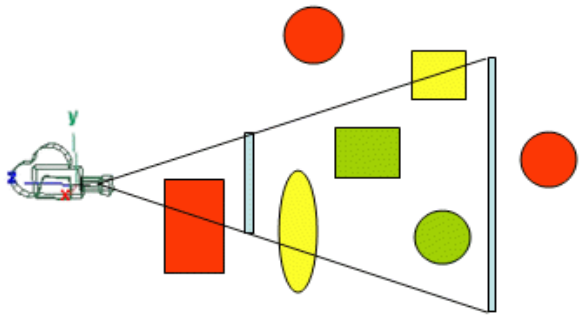
```
<style>
    canvas {
        width: 100%; height: 100%
        }
</style>
. . .
```

e proseguiamo con l' inserire le librerie di Three.js e quelle di cui avremo bisogno nel *<body>*:

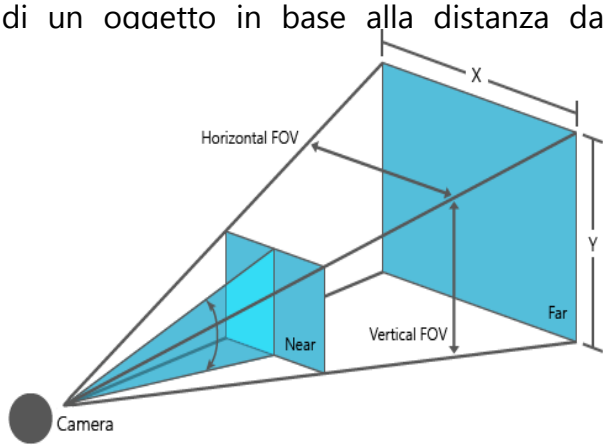
```
<script src="jsLib/Three.js"></script>
. . .
```

Una volta scritto il set-up iniziale, si suddivide per convenzione il progetto in 3 parti: inizializzazione (rappresentata dalla funzione **init()**, in cui verrà inizializzato il progetto stesso), una di aggiornamento (rappresentata da **animate()**, in cui verrà scritto il codice di animazione; se si sviluppa un progetto interamente statico, questa funzione viene omessa) e una di disegno (rappresentata da **render()**, in cui i modelli creati vengono disegnati nella canvas).

Per creare il mondo tridimensionale che si vuole manipolare e mostrare su schermo, si necessità di tre oggetti fondamentali: la scena, un renderer e la camera. La scena é l' area a cui verranno aggiunti i vari oggetti del mondo che si vuole creare. Il secondo oggetto da creare é il renderer (assegnato ad un *HTML DOM element* scelto), con il quale verrà rappresentata la scena. Three.js offre 2 tipi di renderer: CanvasRenderer e WebGLRenderer; entrambi sono incorporati nella web page usando l' HTML tag <*canvas*>. Nel progetto verrà utlizzato il secondo in quanto potendo usufruire della GPU offre prestazioni nettamente superiori. Il terzo oggetto é invece la camera; vi sono due tipi di camera: OrtographicCamera (genera una visione assonometrica a partire da una proiezione ortogonale – in sostanza un oggetto che viene duplicato, ma a maggior distanza dalla camera rispetto all' originale, viene comunque visto con le stesse dimensioni) e PerspectiveCamera (come suggerisce il nome, questo tipo camera utilizza una proiezione prospettica, ovvero lo stesso oggetto posizionato più lontano dalla camera viene visto più piccolo); nel progetto verrà utilizzato il secondo tipo di camera, in quanto é quello che maggiormente si avvicina alla percezione umana delle dimensioni di un oggetto in base alla distanza dall' osservatore.



{01} – cameraObjectVision



{02} - cameraVision

Per creare un oggetto camera che mostri il mondo tridimensionale desiderato, bisogna tenere conto dei parametri della camera e come essi influiscono sulla visualizzazione. Il primo ad essere passato é il *field of view*: l' angolo (normalmente impostato tra 45° e 70°) che determina il campo in cui gli oggetti vengono visualizzati ({02}). Il secondo parametro é l' *aspect ratio* ({02}), ovvero il rapporto tra la larghezza e l' altezza della scena che verrà ripresa nella camera (visto che il progetto verrà visuallizzato su tutto lo schermo, verranno prese le dimensioni della canvas). I sucessivi due parametri passati sono invece il *near* e il *far* ([01] [02]), che rappresentano rispettivamente la distanza minima e massima dalla camera, in cui il rederer disegna gli oggetti: tutti quelli fuori da quest' area non vengono disegnati (come mostrato in figura {01} dalle forme di colore rosso); al contrario, le forme verdi e gialle (che parzialmente escono dall' area) verranno disegnate; da notare come il cerchio verde, nonostante venga disegnato dal renderer, non sarà visibile in quanto occultato dal cerchio giallo.

Qui sotto é disponibile il codice da inserire nella funzione init() per inizializzare la scena del mondo tridimensionale che si vuole rappresentare e il codice della funzione render():

```
function init() {
    scene = new THREE.Scene();
    var WIDTH = window.innerWidth, HEIGHT = window.innerHeight;
    renderer = new THREE.WebGLRenderer();
    renderer.setSize( WIDTH, HEIGHT );
    document.body.appendChild( renderer.domElement );
    camera = new THREE.PerspectiveCamera( 70, WIDTH / HEIGHT, 0.1, 3000 );
    camera.position.set( 0,0,0 );
    scene.add( camera );
    render();
    }


function render () {       renderer.render( scene, camera );       }
```

1.3 : Le luci

Un altro oggetto importante da considerare per poter visualizzare gli oggetti 3D presenti nella scena, sono le luci. In Three.js abbiamo diverse tipologie di luce, a seconda del grado di realismo che vogliamo dare alla scena (elencate qui sotto dal grado più basso a quello più alto):

- AmbientLight (coloreDellaLuce) – risulta la meno realistica tra tutte in quanto illumina tutti gli oggetti allo stesso modo
- DirectionalLight (coloreDellaLuce, intensità) - illumina da una specifica direzione (non posizione: il metodo position.set serve ad indicare appunto la direzione); i raggi che illuminano gli oggetti della scena si comportano come se provenissero da un punto molto lontano e quindi sono tutti paralleli; la migliore analogia per questa luce é quella con il sole: i raggi sono così lontani che illuminano con la stessa angolazione
- PointLight (coloreDellaLuce, intensità, distanza) - illumina gli oggetti con una luce a mano a mano più debole con l' aumentare della distanza dal punto di emanazione: quando la distanza dalla sorgente é uguale al parametro 'distanza' passato alla luce, l' intensità diventa 0; una buona analogia é una lampadina, che illumina in tutte le direzioni e all' aumentare della distanza la luce diminuisce
- SpotLight (coloreDellaLuce, intensità, distanza) - come la PointLight ma gli oggetti possono anche essere dotati di ombra (ques' ultima può essere personalizzata modificando i vari parametri 'shadow' della luce); si comporta come un faretto.

1.4 : Gli oggetti 3D

Una volta creata la scena, si possono finalmente inserire gli oggetti 3d (i Mesh) da visuallizare; questi sono composti da una geometria e da un materiale. La geometria di un Mesh definisce la sua forma e a sua volta é composta da vertici e facce. Vi sono vari

modi per ottenere la geometria di un oggetto: il più semplice di tutti é usare le forme base previste da Three.js ([6] [14]), come la PlaneGeometry, BoxGeometry, SphereGeometry, TubeGeometry, TorusGeometry, etc.; se tuttavia non si vuole essere limitati solo alle forme basi, Three offre la possibilità di creare forme più complesse come ad esempio usando le operazioni Booleane (addizione, sottrazione, unione, intersezione) ([13]) tra i vari mesh a disposizione; per creare forme più complesse si può inoltre usare l' estrusione ([14]) a partire da delle spline o anche da geometrie in 2D; infine, é possibile creare delle geometrie dichiarando un oggetto come nuova geometria ed eseguendo dei push di vertici e facce su di essa:

```
var objGeometry = new THREE.Geometry();
var v1 = new THREE.Vector3( -10, 10, 0 );
var v2 = new THREE.Vector3( -10, -10, 0 );
var v3 = new THREE.Vector3( 10, -10, 0 );
objGeometry.vertices.push( v1, v2, v3 );
geometry.faces.push( new THREE.Face3( 0, 1, 2 ) );
```

Per quanto con questo metodo si possa creare qualsiasi forma, é preferibile (in termini di tempo impiegato e visibilità del codice) importare modelli già esistenti o crearne di nuovi da importare usando programmi appositi, come 3ds Max, Bleder ([10] [11]), Maja, etc. La vasta scelta di software per la modellazione 3D e i numerosi tipi di importazione dei modelli (colladaLoader, jsonLoader, objLoader, etc.) in Three.js rendono molto semplice la creazione di geometrie complesse.

Ottenuta la forma dell' oggetto che si vuole disegnare, bisogna assegnarli anche un materiale. Gli elementi canvas usati da WebGL sono composti da un *control code* ed uno *shader code*, ma per ora sono stati riportati solo esempi scritti in JavaScript del primo: questo perchè Three.js fa lo shader coding da solo, senza bisogno di alcun codice aggiuntivo da parte del programmatore, grazie ad un libreria di codici GLSL predefiniti adatti a molteplici usi; gli shaders vengono rappresentati quindi attraverso il concetto di materiale; esso é un oggetto che definisce le proprietà di superficie di un mesh 3D. Esistono vari tipi di materiali in Three.js:

- MeshBasicMaterial ([6] [14]) – come suggerisce il nome, é un materiale base senza ombre o ombreggiatura (la luce non contribuisce in nessun modo al rendering dell' oggetto) e quindi adatto al disegno di oggetti con geometria piana o con wireframe. Una sfera con questo materiale verrà vista dall' utente come un cerchio
- MeshDephtMaterial ([6]) – materiale per il disegno in profondità: a seconda della vicinanza o lontanza dell' oggetto dalla camera esso verrà disegnato bianco o nero (o gradazioni tra i due colori se esso non coincide nè con il parametro *near* nè con il *far* della camera)
- MeshFaceMaterial ([6]) – usato per ottenere un oggetto che non sia tutto dello stesso materiale, ma che possa avere materiali diversi su ogni sua faccia
- MeshLambertMaterial ([6] [14]) – materiale più realistico del basic in quanto crea un effetto di ombreggiatura sulle facce coperte dalla sorgente di luce; tuttavia l' influenza della luce sull' oggetto dipende solo dalla posizione dei due, non tenendo conto dell' angolo dell' osservatore (della camera): dipende solamente dall' angolo α tra la direzione della sorgente di luce e la normale alla superficie e dalla distanza della sorgente luminosa. La superficie dell' oggetto riflette la luce nello stesso modo in tutte le direzioni; dunque la essa appare ugualmente luminosa per tutti gli angoli di vista.
- MeshPhongMaterial ([6] [14]) – usato spesso per dare alle superfici un effetto lucente, quasi riflettente. L' area i cui spigoli sono rivolti lontano dalla sorgente di luce verrà disegnata a mano a mano più scura. La differenza sostanziale tra il modello di Phong e quello di Lambert é che quest' ultimo non é in grado di considerare le riflessioni speculari. Tra tutti i materiali, questo é il più realistico, per quanto sia un modello empirico non basato sulla fisica ma su osservazioni del comportamento della luce quando é riflessa da certi tipi superfici

Infine si aggiungono ulteriori dettagli all' oggetto, come il colore o una texture (il texture mapping é un metodo che aggiunge dettagli ad un mesh 3D applicando un' immagine a una o più facce, senza doverne modificare la geometria), la trasparenza,

la riflessione, le bump map (é una bitmap che disloca i vettori della superficie per creare apparentemente una superficie rugosa/porosa: i valori dei pixel non vengono trattati come colori ma come vettori che influenzano quelli della superficie. Tali dettagli tuttavia non fanno parte della geometria e vengono aggiunti solo in fase di rendering. Ai valori più alti, bianchi, corrispondono sporgenze, mentre a quelli neri depressioni. Con questa tecnica non aumenta il numero di poligoni da renderizzare), e molte altre.

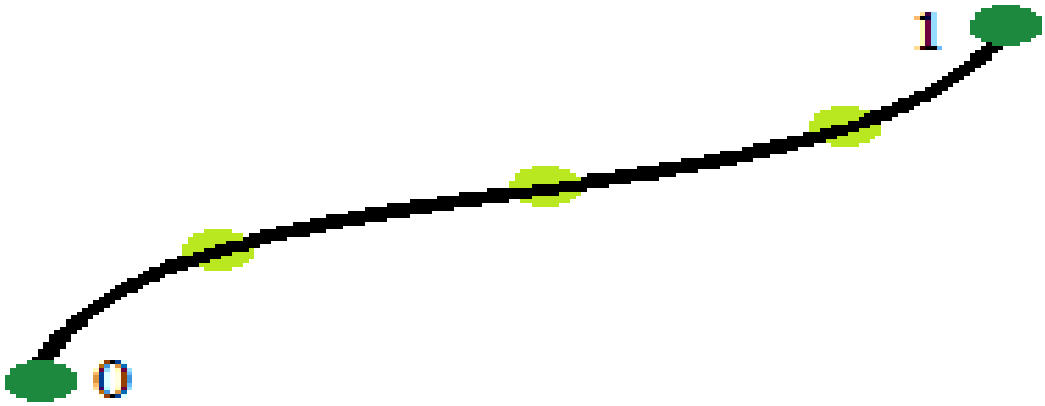
1.5 : Animazione

L' ultima cosa di cui tener conto per avere il massimo grado di realismo nella scena 3D creata, é l' animazione ([14]), che significa poter effettuare dei cambiamenti all' immagine disegnata sullo schermo nel tempo, dando vita ad una scena 3D altrimenti statica; per quanto ci siano vari modi di animare la scena e di concepire il problema, alla fine si tratta di far muovere i pixel: in questo modo si possono far muovere / rotare / scalare gli oggetti, cambiare la loro geometria o il loro materiale, etc. Ecco alcuni dei metodi che si possono usare:

- *requestAnimationFrame()* – per creare un ciclo di esecuzione. Inizialmente le web applications usavano dei timer (come *setTimeOut()* oppure *setInterval()*) per animare i propri contenuti, ma con l' aumentare della complessità delle animazioni e del numero di interazioni con esse, questi metodi cominciarono a presentare dei problemi: essendo le funzioni chiamate a specifici intervalli di tempo, potenzialmente molte chiamate al rendering venivano sprecate in quanto chiamate anche quando la web page non era visibile / attiva; inoltre le applicazioni JavaScript non hanno modo di sincronizzarsi con le altre forme di animazione (come quelle generate da SVG o CSS) e conseguentemente devono scegliere arbitrariamente il valore dell' intervallo: se la scelta cade su 1/24 di un secondo (24fps), allora l' utente sarà privato di una buona risoluzione su un display a 60Hz; se invece viene scelto 1/60 di un secondo (60fps) e un display

lento nell' aggiornare l' immagine si avrà uno spreco di CPU (disegnerà molte immagini che non verranno mai visualizzate). Con r*equestAnimationFrame()* non viene chiesto al browser di ridisegnare dopo un determinato intervallo di tempo ma quando esso é pronto a ripresentare la pagina (ovvero quando avvengono dei cambiamenti, come la pressione di un tasto da tastiera) e non si ha sprechi di CPU quando l' utente si trova su un' altra pagina (e quindi questa non é attiva)

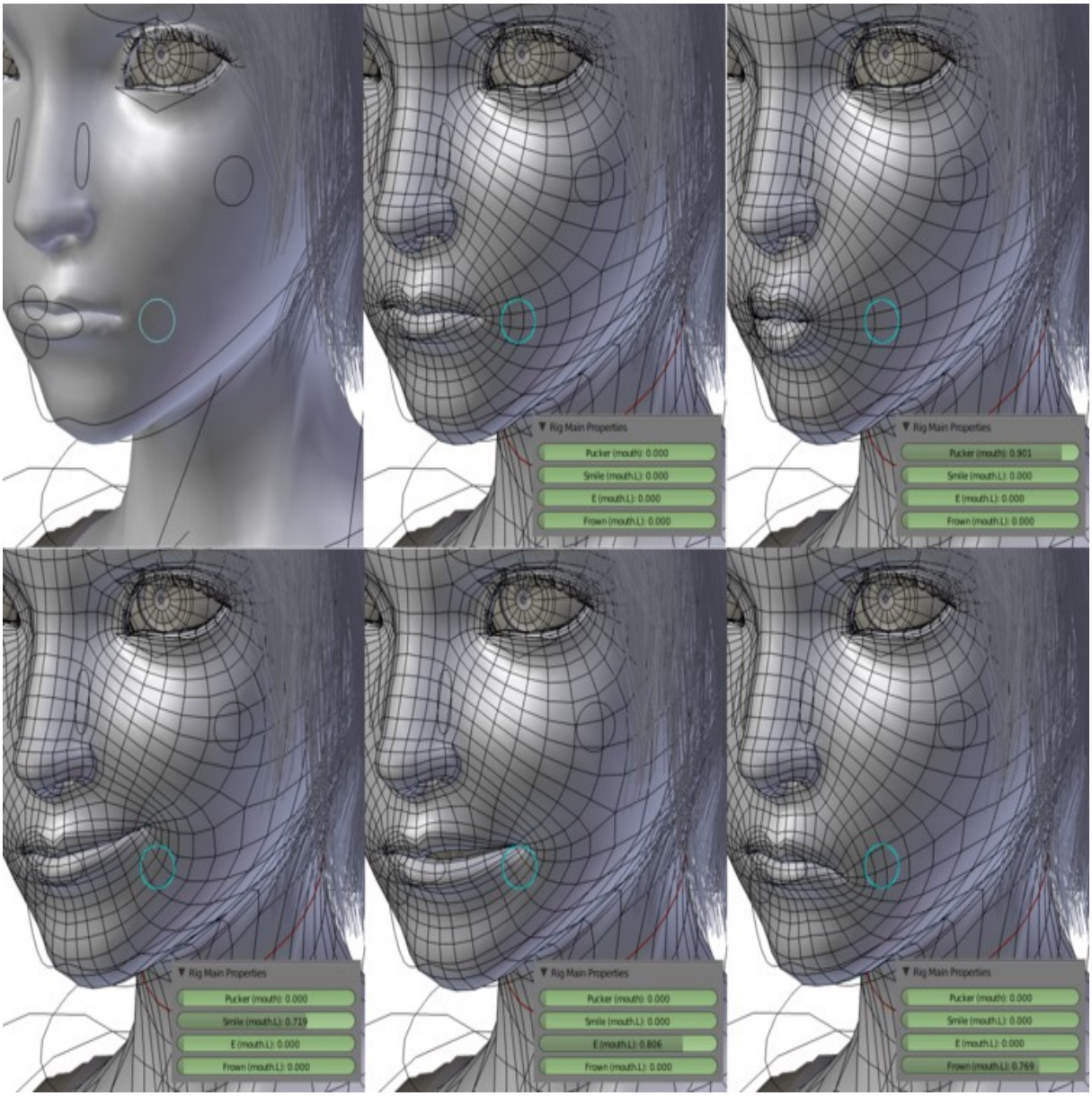
- *tweening* – transizione in modo melifluo da un valore ad un altro; tecnica usata principalmente per far muovere oggetti da una posizione ad un' altra su di un path (i valori intermedi tra il primo e l' ultimo vengono generati dal processo stesso – questa tecnica di calcolo matematico é chiamata interpolazione)



{03} – tweening

- *key frames* – per animazioni più complesse tuttavia, il tweening non é adatto; non vengono più specificate le due chiavi di inizio e fine (con i relativi valori associati, ad esempio di posizione [x, y, z]), ma una lista di chiavi (ad esempio temporali), potenzialmente con diverse durate tra i singoli valori, a cui vengono associati i relativi valori; in seguito quindi viene applicato il metodo di tweening tra i valori delle diverse coppie di chiavi; anche in questo caso si può usare l' interpolazione lineare come si può usare delle spline. La differenza quindi rispetto al tweening é che questo metodo permette non solo di avere più di due chiavi, ma sopratutto di poter manipolare la durata (se ad esempio usiamo chiavi temporali) tra una chiave ed un' altra. Il termine *key frames* é fuorviante in quanto si riferisce sia alle animazioni basate sui frame che su quelle basate sul tempo

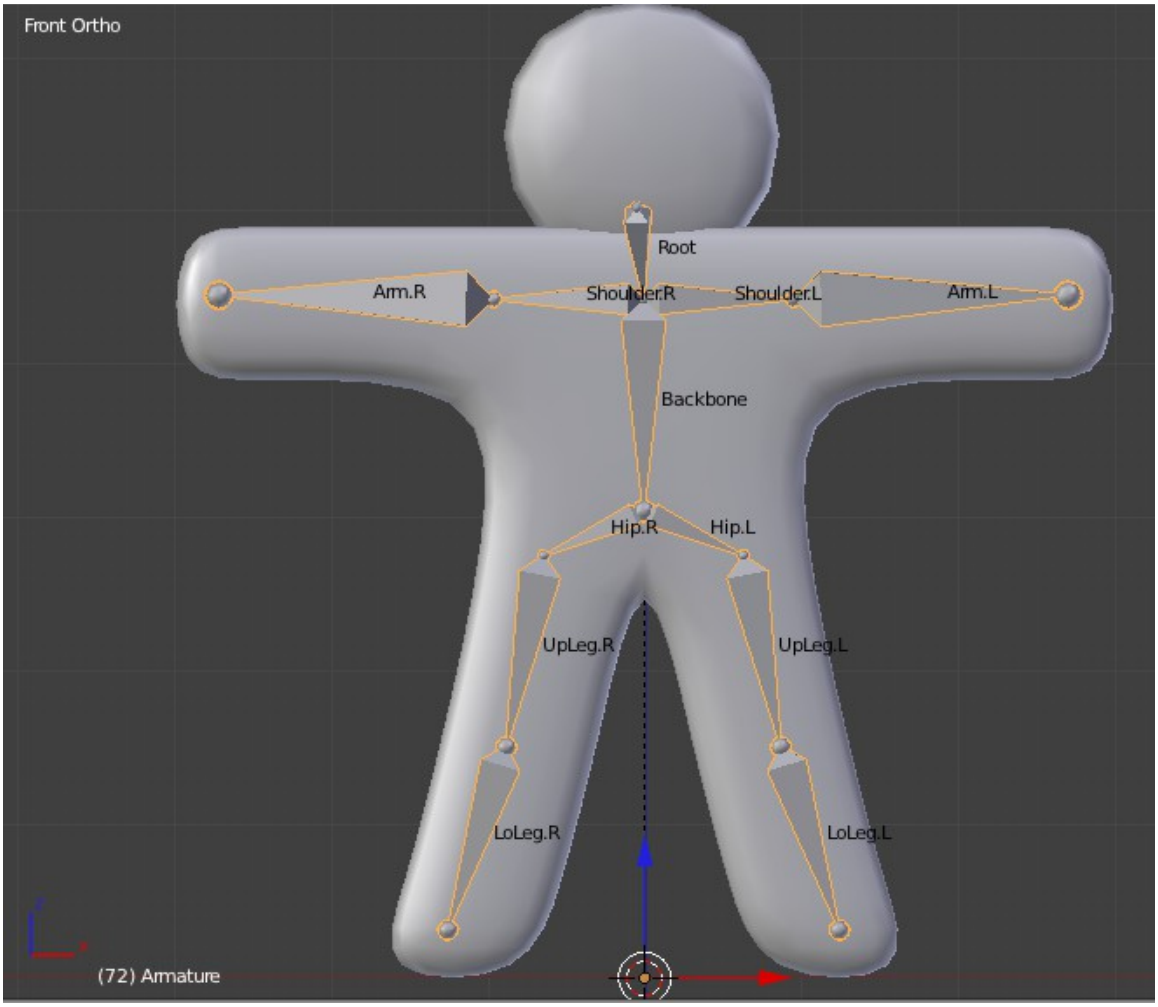
- *morph targets* – usato per deformare la geometria di un oggetto con la combinazione delle forme di un set predefinito; usato spesso per semplici animazioni facciali. Vengono usate le interpolazioni basate su vertici per cambiare i vertici di un mesh: normalmente un sotto-insieme di vertici di un mesh viene memorizzato come un set di morphed targets che verrà poi usato con il tweening



{04} – MorphTargets_face

- *skinning* – deforma la geometria agendo sui vertici di essa sulla base dello "*skeleton*" (in Blender viene definito come "*armature*") inserito; tecnica usata per l' animazione di forme coplesse. Uno skeleton é composto da *bones* (ovvero ossa), organizzate hierarchicamente e quindi la trasformazione di un *bone*

influenza tutti i suoi figli; ad ogni *bone* viene associato un set di vertici del mesh insieme ad *blend/vertex weight* (specifica quanto quel determinato osso influenza i suoi vertici) per ogni vertice associatovi; inoltre ad ogni vertice può essere associato più di un *bone*.



{05} – BlendersArmature

Capitolo 2 : Software usato

Durante lo sviluppo sono stati usati diversi software riguardanti diversi aspetti del progetto stesso, come ad esempio la scrittura del codice, il testing o l' aspetto grafico. In questo capitolo si farà una piccola panoramica solo su quelli più usati.

2.1 : Blender

Blender ([10] [11]) é un programma di grafica 3D che riunisce al suo interno gli strumenti per realizzare sia immagini che complete animazioni in computer grafica. La filosofia Open Source che caratterizza Blender é forse il suo più grande punto di forza: oltre a essere disponibili i suoi codici sorgente, viene rilasciato gratuitamente e senza nessuna particolare restrizione d'utilizzo. Per quanto riguarda gli export dei modelli da Blender a Three.js, quest' ultimo offre uno script (https://github.com/mrdoob/three.js/tree/master/utils/exporters/blender/) per Blender in modo da poter esportare in formato json, tuttavia tutta la geometria esportata in un unico file .js viene unita formando un unico oggetto (Three: github: problema #524 (fino alla versione in uso nel progetto, r57)): si perde quindi l' informazione riguardante i vari gruppi e sotto-gruppi. Altre soluzioni di import invece sono ColladaLoader (in formato .dae) e ObjLoader (in formato .obj). Qui sotto vengono elencate alcune osservazioni ([15]) riguardanti vari esperimenti fatti nell' esportare modelli da Blender ad una scena in Three.js (per il progetto é stato scelto di usare ColladaLoader):

- bisogna esportare un oggetto che abbia vertici, per cui se si vuole esportare un testo, bisogna prima trasformarlo in un mesh, altrimenti non verrà renderizzato
- creare l' oggetto con *Blender render* e non con *Cycles*, altrimenti il .dae non sarà visualizzato secondo le aspettazioni
- la scelta di materiali é ancora limitata a quelli base (Basic, Lambert, Phong); altri tipi non verranno disegnati

- se la scena in Blender é composta da più oggetti differenti (ovvero provenienti da altri file) bisogna fare il “join” tra loro, altrimenti le loro posizioni e rotazioni non verranno rispettati nella scena di Three.js.: il processo di esportazione terrà in considerazione la posizione e rotazione dell' oggetto originale, non delle modifiche fattegli nella scena a cui si é fatto il link / append e che poi é stata esportata.

Effetuato il processo di esportazione, basta inserire poche righe di codice per poter visualizzare il modello anche in Three.js; inanzitutto bisogna aggiungere la libreria di export usata (in questo caso ColladaLoader.js):

```
<script src="jsLib/ColladaLoader.js"></script>
```

e aggiungere quindi la scena (estraendo gli oggetti di interesse – senza le luci, camera, etc.) esportata da Blender alla scena creata in Three.js:

```
var loader = new THREE.ColladaLoader();
loader.options.convertUpAxis = true;
loader.load( 'model.dae', function ( collada ) {
    // with this you can get the objects of the scene;
    var obj1 = collada.scene.children[0];
    // you can name the object so you can use it even out of the function
    obj1.name = "daeObj1";
    // you can set here some material properties as trasparency
    obj1.material.needsUpdate = true;
    obj1.material.transparent = true;
    obj1.material.opacity = 0.5;
    // and set some position and rotation
    obj1.position.set( 0, -5, -0.6 ); //x,z,y
    obj1.rotation.set( 0, 45, 0 );
    // and add the obj to the threeJs scene
    scene.add( obj1 );
    });
```

2.2 : FileZilla

FileZilla Client é un software libero multi-piattaforma che permette il trasferimento di file in rete attraverso il protocollo FTP (tra i vari protocolli supportati, vi sono anche l' SFTP, e l' FTP su SSL/TLS). Lavorando in rete e testando il codice non solo in locale ma anche su browser, questo software gratuito si rivela essenziale.

2.3 : GitHub

Durante lo sviluppo é stato indispensabile usare un repository (come BitBucket o GitHub) che permetta di avere tutti i log di ogni modifica e quindi di poter tornare indietro ad una versione stabile precedente. GitHub é un servizio web di hosting per lo sviluppo di progetti software (e non solo) che usa il sistema di controllo di versione git; offre diversi piani per repository privati sia a pagamento, sia gratuiti, molto utilizzati per lo sviluppo di progetti open-source; fornisce funzionalità simili a un social network come feeds, follower e grafici per vedere come gli sviluppatori lavorano sulle varie versioni dei repository.

2.4 : GutHub pages

Per visualizzare il lavoro svolto si é preferito usare un servizio di web hosting anziché lavorare in locale; in questo modo il progetto diventa indipendente dal computer usato ed é possibile visualizzare l' esito del lavoro svolto in qualsiasi momento e luogo (purchè si disponga di una connessione internet); utilizzando già Github come repository, si é deciso di creare un progetto privato (in modo che persone non autorizzate non potessero accedervi) e usare le GitHub pages per la visualizzazione.

Capitolo 3 : Sviluppo e osservazioni

Il progetto é nato dall' esigenza di Sogei s.p.a. di dover mostrare il Centro Elaborazione Dati (CED) senza incorrere in nessun tipo di rischio di fuori-uscita di informazioni (data la natura riservata e segreta dei dati contenuti all' interno) o di manomissione dei server: Il CED garantisce l' operatività dei servizi erogati direttamente ai cittadini e ai professionisti (ad esempio l' invio telematico delle dichiarazioni, le visure ipocatastali, i pagamenti tramite F24, ...), tutti servizi fruibili 365 giorni l' anno per 24 ore al giorno. Per far fronte a queste problematiche si é deciso di ricostruire un modello virtuale del CED in modo che le future visite al complesso vengano effettuate tramite la rete e non fisicamente.

La parte di analisi della realtà d' interesse é stata decisiva per l’ evoluzione del progetto, portando in evidenza quali aspetti dovevano essere sviluppati subito e quali in futuro; per questo motivo durante la parte di progettazione e di sviluppo del software, si é rivolta particolare attenzione alla comprensibilità e alla riusabilità del codice, facilitando così le prossime iterazioni. Per raggiungere questo scopo si é rivelato indispensabile applicare le linee guida offerte dai pattern GRASP e GOF ([12]) e dalla RDD ([12]), in aggiunta all’approccio UP ([12]).

In questo capitolo verranno mostrati in evidenza alcuni passi chiave del progetto con relativo codice significativo e osservazioni.

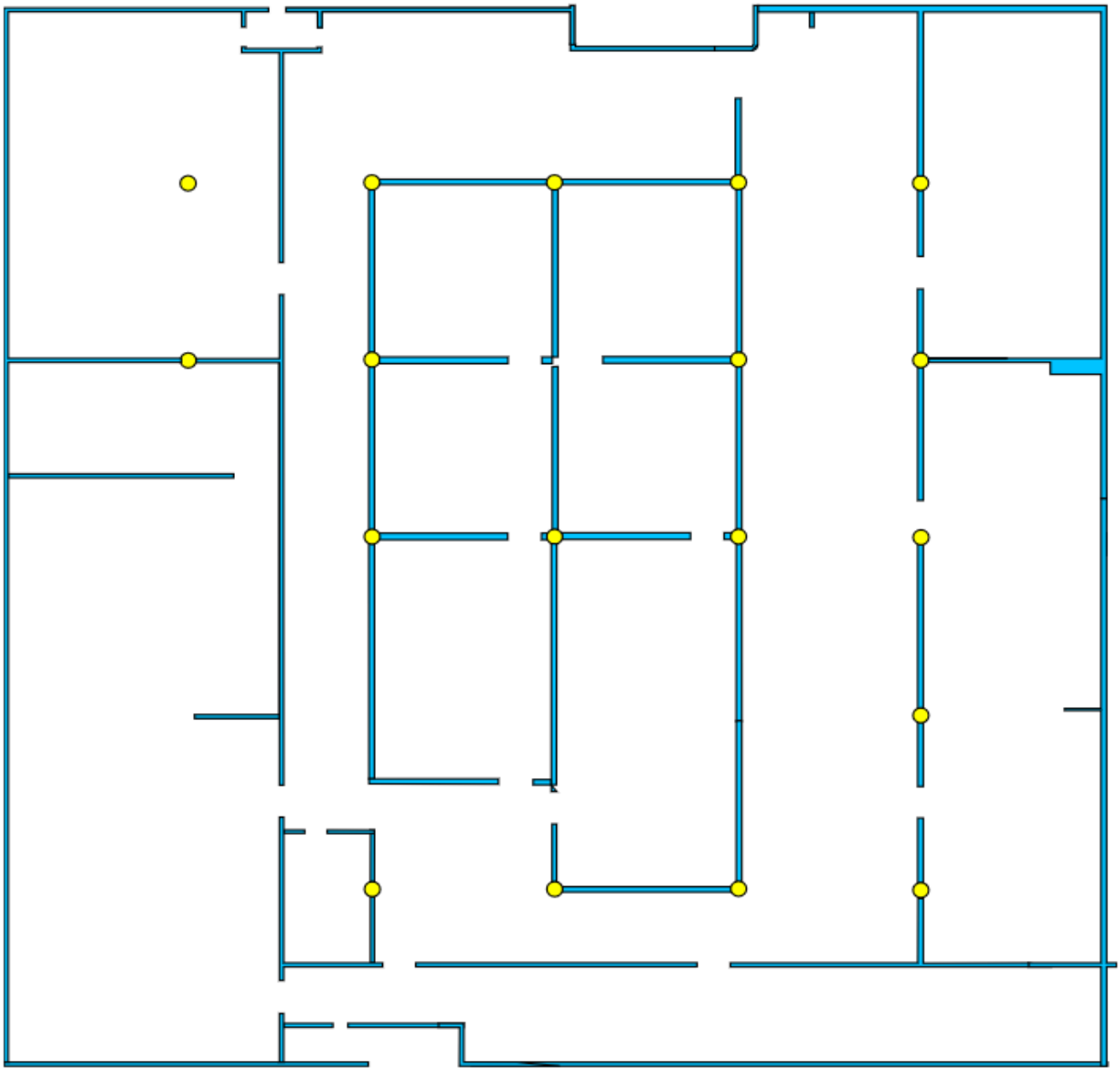
3.01 : La base del progetto

Dopo aver costruito la scena, si é passati alla creazione del modello da navigare. Con le informazioni ricavate dal file .svg ricevuto in merito alle mura dell' ambiente (vedi immagine {06}), si é proceduto a crearne il modello usando il software Blender e successivamente si é importato tale modello nel progetto di ThreeJs come un file .dae usando ColladaLoader.js per caricarlo. Successivamente sono stati aggiunti anche il

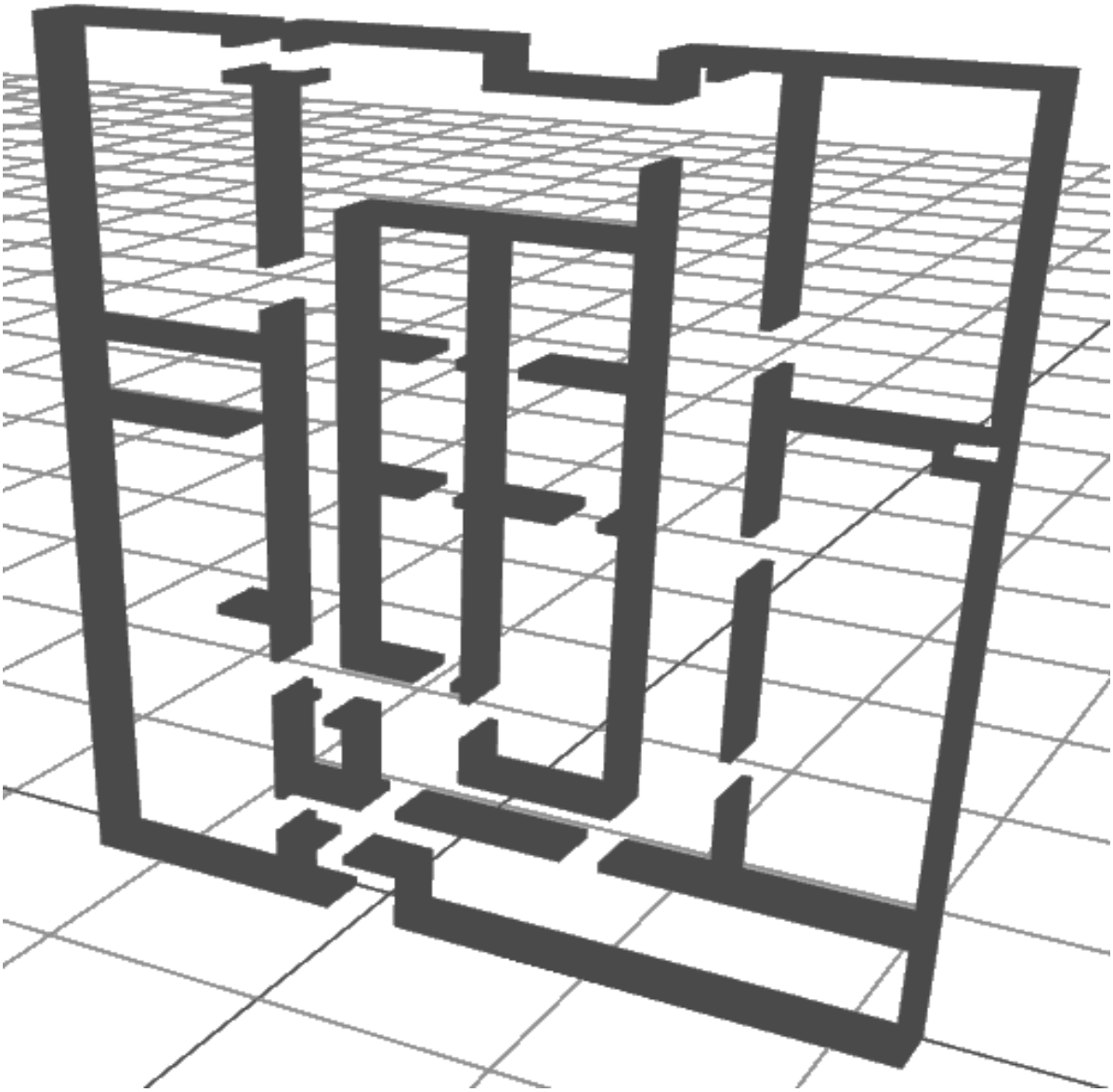
pavimento e il soffitto (le cui dimensioni sono state calcolate in base alla lunghezza e larghezza totali del mura create); per quest' ultimi, trattandosi di figure estremamente semplici, non é stato usato un programma di modellazione bensì sono stati creati direttamente con ThreeJs con poche righe di codice (il pavimento viene creato allo stesso modo, per cui viene mostrato solo il codice riguardante il soffitto):

```
var ceilingGeometry = new THREE.PlaneGeometry( 45, 44.5 );
var ceiling = new THREE.Mesh( ceilingGeometry, ceilingMaterial );
```

In questa prima fase, é la geometria a ricoprire un ruolo centrale, mentre non viene data importanza al materiale che per ora rimane di tipo *basic* e di colore grigio: questo sarà poi soggetto a cambiamenti quando verranno applicate le *texture*.



{06} – walls



{07} – walls3D

Per poter visualizzare il modello correttamente si ha bisogno anche di luci: non facenti parte tuttavia del cuore del progetto, la loro realizzazione viene posticipata ad una fase successiva, mentre in questa prima parte viene inserita solo una luce di tipo *ambient* con colore grigio chiaro ([16]), che serve per poter visualizzare il tutto e non avere una schermata nera.

```
var ambientLight = new THREE.AmbientLight( 0xB3B3B3 );
```

3.02 : Il movimento base

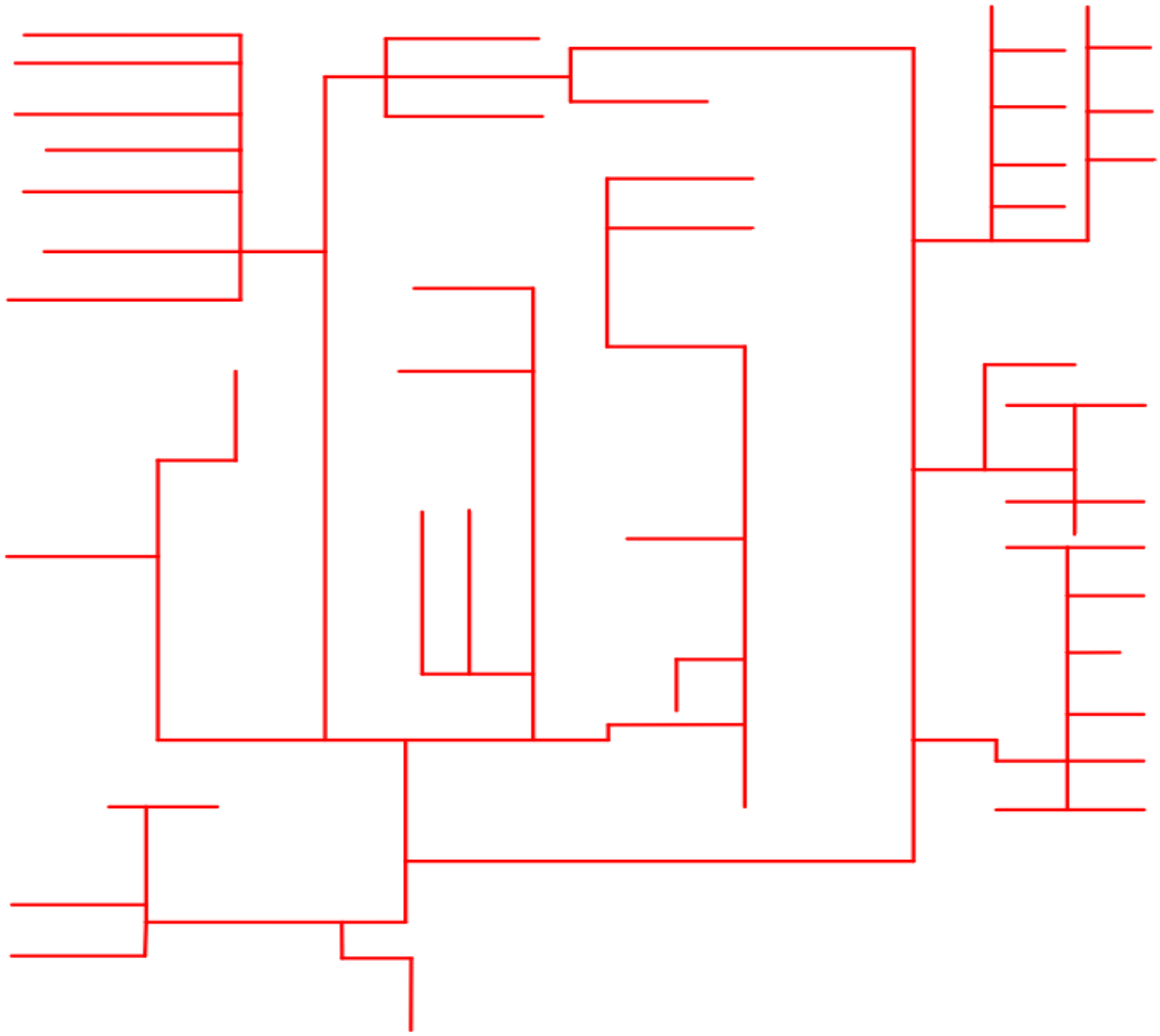
Un' altra parte rilevante del progetto concerne il movimento / la navigazione del modello creato. Sono stati implementati due tipi di movimento: uno automatico, che consiste nel muoversi lungo un percorso selezionato precedentemente dall' utente (in base ad una lista di percorsi disponibili) mentre il secondo si basa sull' imput da tastiera ricevuto dall' utente ad ogni istante.

Per il movimento manuale, inizialmente è stato pensato di sfruttare le proprietà del boundigBox che possiede ogni geometria, per controllare la presenza di collisioni, ma questo metodo si é rilevato poco adatto alla situazione in quanto implicava estrarre dal modello delle mura ogni singolare parete a cui poter applicare il boundigBox: usarlo sull' intera struttura non aveva senso in quanto il visitatore si sarebbe sempre trovato tra il punto massimo e minimo del boundigBox delle mura. Si é passato quindi ad un approccio differente: sono stati determinati degli spazi in cui é possibile muoversi (le mura hanno svolto la funzione di delimitare tali spazi fungendo da perimetro); l' utente quindi, premendo uno dei tasti assegnati al movimento (W A S D) può spostare, sul piano XZ un piccolissimo oggetto trasparente chiamato cameraEye (esso viene creato nello stesso momento in cui viene creata la camera): in questo modo viene traslata anche la camera, puntata a inquadrare la cameraEye, ottenendo l' effetto di movimento desiderato; prima di eseguire l' operazione tuttavia, viene fatto un controllo se il movimento nella specifica direzione non crea collisione con una parete (questo avviene semplicemente cotrollando se la cameraEye si trova in una zona dove non può stare), nel qual caso essa viene bloccata. Similmente avviene per la rotazione, rispetto al piano Y, usando i tasti Q E.

```
if ( keyboard.pressed("W") && ( canAccessArea(moveDistance) ||
                unlockWMovement ) ) {
        cameraEye.translateZ( -moveDistance );
. . .
// rotate left/right/up/down
```

```
if ( keyboard.pressed("Q") )
        cameraEye.rotation.y += 0.01;
if ( keyboard.pressed("E") )
        cameraEye.rotation.y -= 0.01;

var relativeCameraOffset = new THREE.Vector3(0,0,1);
var cameraOffset = relativeCameraOffset.applyMatrix4( cameraEye.matrixWorld );
camera.position.x = cameraOffset.x;
camera.position.y = cameraOffset.y;
 camera.position.z = cameraOffset.z;
camera.lookAt( cameraEye.position );
```



{08} – paths

Il movimento automatico, a differenza di quello manuale riceve l' imput da parte dell' utente solo due volte: un punto di inizio e uno di fine. In base a questi due punti il sistema calcola il percorso più corto, usando l' algoritmo di Djikstra, e fa partire la navigazione facendo visitare all' utente in modo fluido il modello disponibile sul quel determinato percorso (i punti sono rappresentati dalle intersezioni di linee rosse mentre i percorsi disponibili sono tutti quelli che si possono creare con almeno due linee rosse che abbiano un punto in comune – vedi immagine {08}).

Ma come funziona nello specifico? Ottenuta una sequenza di punti (formanti un percorso) dal grafo passato all' algortimo di Djikstra, viene creata una curva Spline aggiungendo ulteriori punti in modo da avere curve morbide e non ad angolo retto sul percorso:

```
var smothing = 400;
var splinePath = new THREE.SplineCurve3( getPointsFromGraph() );
controls.points = splinePath.getPoints( smothing );
```

Successivamente, per creare l' animazione, ovvero il movimento automatico sui punti così ottenuti, viene usata la libreria PathControls.js che imposta la *camera* da muovere sui *controls*, dove é salvato il percorso da percorrere:

```
pathControlsRun();
scene.add( pathControls.animationParent );
```

Inoltre vengono settati alcuni parametri, come la possibilità di avere una navigazione ciclica / infinita o di fermarsi quando si arriva al punto di fine:

```
//true-> infinite animation - false -> stops at the end
pathControls.animation.play( false, 0 );
```

o il tempo in cui percorre tale percorso:

```
pathControls.duration = 25;
pathControls.useConstantSpeed = true;
```

o ancora la possibilità di poter muovere il mouse e quindi la visuale della camera, non obbligando così l' utente a guardare sempre in fronte a se:

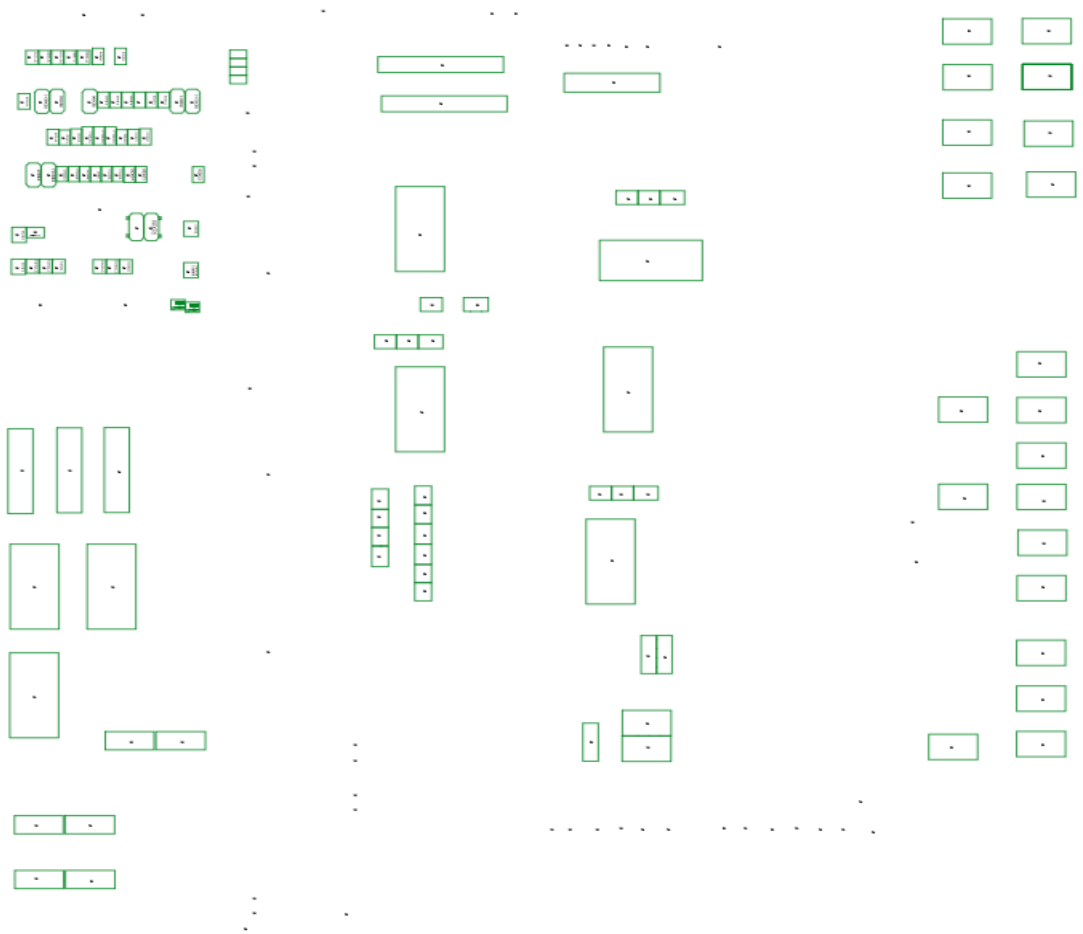
```
pathControls.lookSpeed = 0.02; // speed on mouse move
```

```
pathControls.lookVertical = true;
pathControls.lookHorizontal = true;
pathControls.verticalAngleMap = {srcRange: [ 0, 2 * Math.PI ], dstRange: [ 1.5, 3 ]};
pathControls.horizontalAngleMap = {srcRange: [ 0, 2*Math.PI ], dstRange: [ 0, 3 ]};
```

Ovviamente trattandosi di un' animazione, non basta solo inizializzarla, ma questa deve essere aggiornata in continuo, per cui dobbiamo inserire nella funzione *animate()* l' aggiornamento dei *pathControls*:

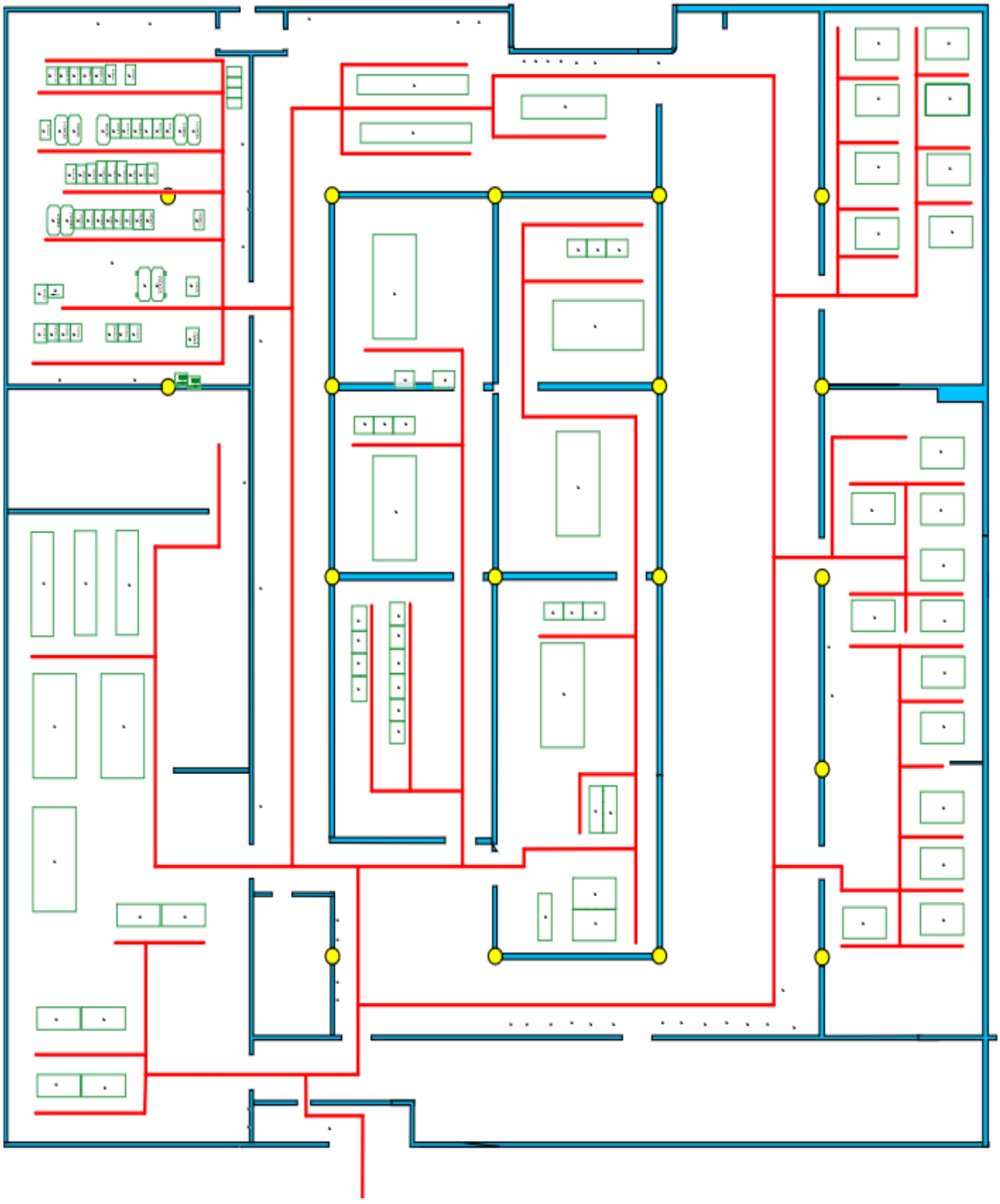
```
var delta = clock.getDelta();
pathControls.update( delta );
```

3.03 : I server da visitare



{09} – servers

Il cuore del progetto é costituito dai modelli 3D dei server che rappresentano quelli reali del CED. Anche per questi oggetti si ha a disposizione un file .svg che fornisce informazioni utili come la dimensione o la posizione di essi (vedi immagine {09}). Non essendo molto esplicativa l' immagine dei servers da sola, bisogna comporla con le altre due, ottenendo la seguente {10}:

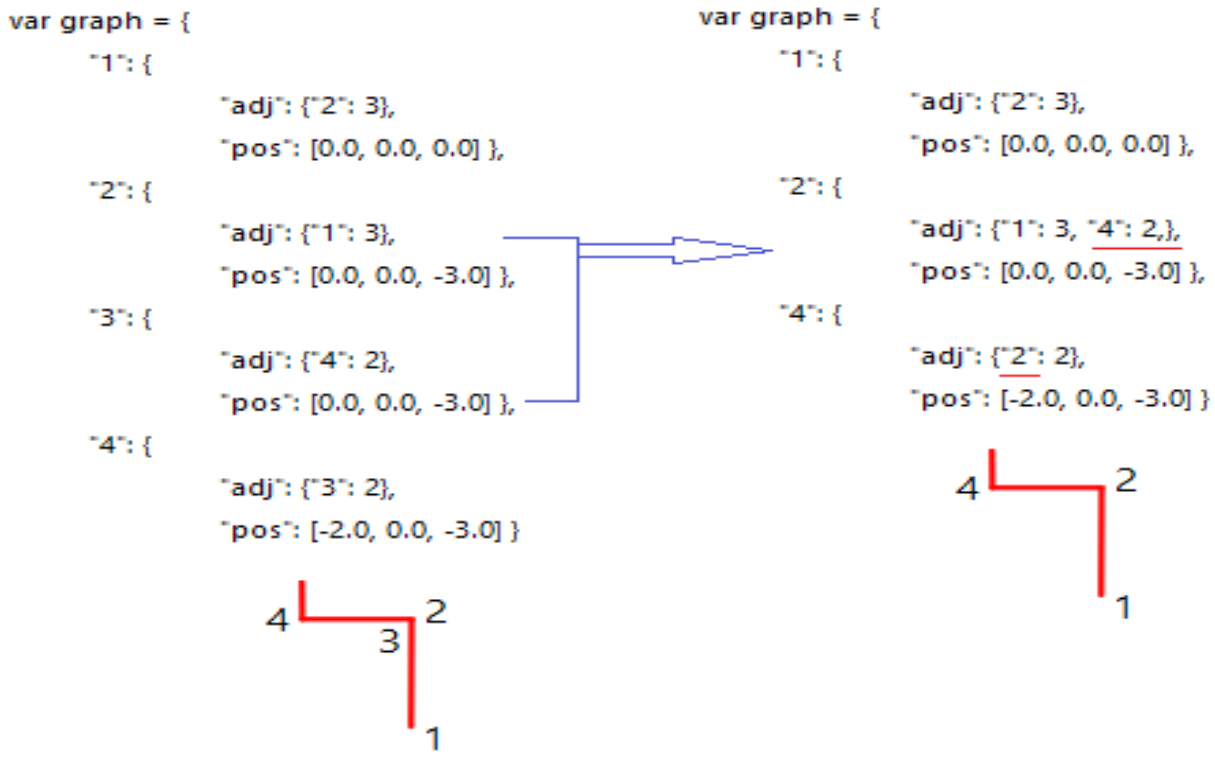


{10} – paths_servers_walls

Per rappresentare quindi questi oggetti é stato creato un file di configurazione contenente informazioni come le dimensioni, posizione e il tipo (in questo caso, tutti i server sono rappresentati da cuboidi).

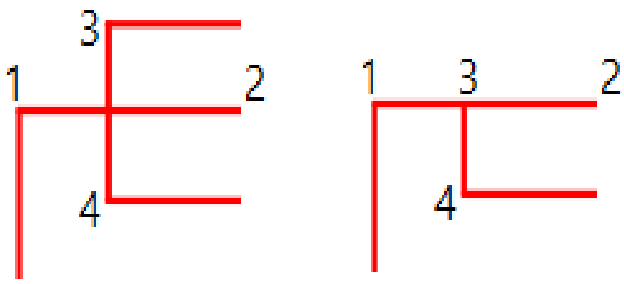
3.04 : Creazione automatica di path usando l' .svg

Una volta completato il modello base, l' attenzione é stata spostata su alcuni dettagli per migliorare il gradimento della visita del modello. In primo luogo si é deciso di sfruttare il file .svg (riguardante i path) a disposizione e rendere obsoleta la tecnica manuale di creazione dei punti: il file viene trasformato in un formato json (usando un convertitore gratuito a questo indirizzo: http://www.utilities-online.info/xmltojson/#.U7-ZCvRdVAU), quindi vengono letti i punti di partenza e di fine delle linee (che erano state disegnate nel file .svg); su questi inoltre bisogna effettuare un' operazione di traslamento e scalarli in modo che la loro posizione sia concordante con quella delle mura e degli oggetti; in questo modo tuttavia si ha a



{11} – delDuplicatedPoints

disposizione solo una sequenza di coppie di punti: bisogna quindi elaborarli e trasformarli in un grafo di percorsi disponibili. La prima operazione da fare quindi, é analizzare tutti i punti e assegnare ai punti di cui esistono duplicati, i punti che erano collegati ai duplicati stessi; infine rimuovere i punti duplicati. Una migliore comprensione di quello che é stato fatto si può avere guardando la figura ({11}), in cui vengono mostrati i punti prima e dopo con il relativo grafo. Come si può vedere il punto 2 e il punto 3 hanno la stessa posizione, quindi il secondo é un duplicato del primo: il punto 4, collegato al punto 3, viene quindi assegnato come vicino addizionale al punto 2 (e viceversa); infine viene rimosso il punto duplicato come viene rimossa l' informazione del suo collegamento al punto 4 dal medesimo. Con questi semplici passi é possibile trasformare una sequenza di coppie di punti in un grafo connesso esattamente identico all' immagine visualizzata aprendo il file .svg. Per quanto questa tecnica sia semplice, tuttavia, non é completa: esistono infatti dei casi in cui pur essendo due o più linee collegate tra loro visualmente, il loro punto in comune, di intersezione non é presente nella lista: l' informazione che si ha dal .svg infatti riguarda solo i punti di inizio e quello di fine di una linea, ma non dice nulla in merito ai punti intermedi; prendendo come esempio i segmenti nella figura accanto ({12}) si può notare due casi in cui la tecnica illustrata precedentemente non sia efficace: nel primo caso i due segmenti si intersecano in un punto che non appartiene né al punto d' inizio né a quello di fine di entrambi i segmenti per cui non é un punto presente nella lista delle coppie di punti; nel secondo caso invece il punto d' intersezione é un punto d' inizio o fine di uno solo dei due segmenti e quindi il metodo non trova dei duplicati di tale punto e pertanto non può assegnare al punto 3 i suoi vicini 1 e 2: questo comporta che il path non é connesso, in quanto dal punto 3 non si può andare al punto 1 o 2, anche se visualmente sembrano connessi. Per risolvere questo problema, si é dovuto introdurre una seconda funzione di analisi, che tratti tutti i segmenti come rette; si é



{12} – adjustIntersections

proceduto quindi a trovare i punti di intersezione, se presenti (in caso di rette parallele tra loro, avremmo nell' ultima equazione mostrata m2/n2 avrebbe lo stesso valore di m1/n1 e quindi la loro sottrazione darebbe come risultato 0 e conseguentemente avremmo una divisione per 0 non gestibile dal sistema), tra tutti i segmenti e a verificare, usando le equazioni delle rette, se tali punti appartenevano anche ai segmenti (delimitati dai punti di inizio e fine, ovvero da z e x massimi (z2 e x2) e z e x minimi (z1 e x1)):

```
nz = mx + q;
m = (nz2-nz1)/(x2-x1);
q = nz-mx;
(m1x + q1)/n1 = (m2x + q2)/n2;
x = (q1/n1 – q2/n2) / (m2/n2 – m1/n1)
```

inoltre é stato considerato anche il caso in cui i segmenti fossero paralleli a uno degli assi del piano cartesiano; in questo caso si avrebbe delle rette così definite: z = q/n e quindi m = 0 e/o x = q/m e quindi con n = 0, che comporterebbe ad avere, nelle ultime due equazioni, delle divisione per 0 che il sistema non riuscirebbe a gestire: i casi quindi sono stati separati quindi usando degli *if* e gestiti con delle equazione delle rette adattate alla situazione.

3.05 : Multi path

Avendo a disposizione funzioni per ottenere automaticamente dei path a partire da dei file .svg, é nata l' esigenza di avere dei path dipendenti dalle autorizzazioni dell' utenza che visitava il CED; ci si é quindi posti al problema che non tutti i gruppi di persone potevano visitare le stesse cose: un visitatore qualunque, o un impiegato normale non potevano avere accesso a tutte le aree del complesso come ad esempio l' amministratore del CED; inoltre anche i visitatori sarebbero stati diversi tra loro e conseguentemente avrebbero dovuto diverse zone, così come i vari gruppi di tecnici

avrebbero diversi compiti di manutenzione tra loro e quindi diverse aree da gestire. Per far fronte a questa esigenza si é deciso di avere più di un file .svg da cui ottenere grafi contenenti dei percorsi: si é quindi creato un file di configurazione contenente i vari gruppi di utenze a cui sono stati associati diversi grafi di percorsi. In questo modo l' utente all' inizio decide quale tipo di utenza impersonificare e quindi può scegliere su quale grafo di percorsi navigare in base alle autorizzazioni di navigazione del gruppo scelto. Effettuate entrambe le scelte é quindi possibile il percorso su cui si vuole effettuare la visita.

3.06 : Interazione : picking su oggetti e uso di QRCode

Oltre alla realizzazione del modello del CED e alla navigazione al suo interno, doveva essere realizzata ancora una cosa importante: l' interazione con il mondo virtuale creato: non si tratta infatti di una mostra di quadri in cui il visitatore si può muovere in delle stanze e ammirare cosa vi sia all' interno, bensì di sale contenenti server che di per se non sono interessanti da guardare; la parte interessante nel visitare i server del CED è conoscere il contenuto, non ammirare cuboidi metallici e cavi su cavi. Per far fronte a questa esigenza si é introdotta l' interazione con gli oggetti presenti nella scena 3D; in particolare si é usata la tecnica del picking: per usare questo metodo servono due dati importanti: la posizione della camera e la posizione del mouse nel momento del click/interazione; le due posizioni vengono unite quindi da una segmento (con inizio dalla posizione della camera ed estensione fino al parametro far della camera): questo raggio, attraversando la scena, colpisce/attraversa degli oggetti che vengono inseriti in una lista:

```
mouse.x = ( event.clientX / windowInnerWidth ) * 2 - 1;
mouse.y = - ( event.clientY / windowInnerHeight ) * 2 + 1;
var vector = new THREE.Vector3( mouse.x, mouse.y, 1 );
projector.unprojectVector( vector, camera );
```

```
var ray = new THREE.Raycaster( camera.position,
        vector.sub( camera.position ).normalize() );
var intersects = ray.intersectObjects( scene.children );
```

Tuttavia, nel modello virtuale creato, vi sono molti oggetti che non devono avere delle proprietà di interazione (facendo click con il mouse non dovrebbe scattare nessun evento) come ad esempio le pareti delle mura, o il soffitto/pavimento, etc.; per questo motivo é stato pensato di creare, durante l'inizializzazione del sistema, una lista contenente solo gli oggetti che hanno proprietà di interazione in modo da poterli confrontare con gli oggetti trovati con il *raycasting* e quindi richiedere l' interazione solo a quelli che ne hanno la proprietà. Nel caso dei server é stato inoltre stabilito che solo una faccia dell'oggetto fosse cliccabile (il fronte) e conseguentemente é stato inserito un ulteriore controllo: in questo modo non solo viene controllato che l' oggetto cliccato sia un oggetto che possa avere proprietà di interazione, ma anche che sia una sua determinata faccia che se cliccata fa scattare l' evento; per rendere questo possibile viene editato il file di configurazione degli oggetti aggiungendo ad ognuno due attributi: qual' é la faccia che fa scattare l' interazione e il link da aprire in caso questa avvenga.

Per quanto riguarda i link da aprire (e anche per differenziare i vari server tra loro) si é deciso di usare dei QRCode che vengono assegnati alle facce cliccabili degli oggetti. Il QRCode (Quick Response Code) è un codice a barre bidimensionale, ossia a matrice, composto da moduli neri disposti all' interno di uno schema di forma quadrata; viene impiegato per memorizzare informazioni: in un solo crittogramma sono contenuti 7.089 caratteri numerici o 4.296 alfanumerici.

{13} – QRCode

3.07 : Stats

Dopo aver concluso le parti fondamentali del progetto si é passato a realizzare alcuni dettagli per migliorarne la visualizzazione. Come prima cosa sono state aggiunte le informazioni riguardanti gli fps (Frame Per Second) usati dal sistema. Per realizzare ciò é stata importata la libreria Stats.js e si é creato un contenitore all' interno del documento che potesse contenere il grafo con le informazioni degli fps..

3.08 : Texture, perchè non solo la geometria conta

Tra i vari miglioramenti apportati per una visualizzazione più piacevole e realistica maggiore, vi é quello di usare delle texture. Questo é reso possibile grazie al fatto che quando si crea un oggetto ad esso viene attribuito un materiale, e a quest' ultimo é possibile non solo attribuire la proprietà di un colore, ma anche quella di mappare sulla sua superficie un' immagine; é possibile attribuire ad un mesh anche più di un materiale usando una lista di materiali, ognuno attribuito ad una faccia dell' oggetto:

```
var cuboidMaterials = new THREE.MeshFaceMaterial( cuboidMaterialArray );
var cuboid = new THREE.Mesh( cuboidGeometry, cuboidMaterials );
```

Una cosa importante da tenere a mente é che le immagini usate con WebGL devono avere una dimensione in pixel che é una potenza di 2 sia in larghezza che in altezza, altrimenti verranno visualizzate come immagini nere.

Rimanendo nell' ottica di migliorare la visualizzazione del modello rendendolo più realistico, sono state introdotte più luci (quella iniziale di tipo *ambient* é stata comunque lasciata in modo che il modello fosse illuminato ovunque, anche se poco – si é infatti cambiato il colore passando ad un gradiente più scuro); é stato quindi creato un file di configurazione contenente tutte le informazioni riguardante le luci usate: posizione, intensità, colore, etc.; inizialmente si era pensato ad usare delle luci di tipo *spot* in modo che gli oggetti presenti sulla scena potessero avere la proprietà di ombra (e conseguentemente ricevere l' ombra di altri) ma per tenere il progetto leggero e

veloce nell' avviarsi si optato per l' uso di luci di tipo *point*, che comunque avendo un comportamento simile a quello di una lampadina rendono il modello piuttosto realistico.

{14} – projectScreen

Apendice 1 : Analisi della realtà d' interesse

1.1 : Casi d' uso

I casi d' uso ([12]) sono storie scritte che descrivono il funzionamento del sistema che deve essere realizzato; utili nell' identificare e registrare i requisiti. Un caso d' uso é una collezione di scenari (sequenza specifica di azioni e interazioni; descrive una particolare storia) correlati, di successo o fallimento, che descrivono un attore (qualcosa o qualcuno che dotato di comportamento che interagisce con il sistema, compreso il sistema stesso; l' *attore primario* raggiunge degli obbiettivi usando il sistema; quello *di supporto* offre dei servizi al sistema mentre quello *fuori scena* ha interesse/influenza sul comportamento del sistema) che usa un sistema per raggiungere un obiettivo. Alternativamente un caso d' uso può essere definito come un insieme di istanze di casi d' uso, in cui ciascuna istanza é una sequenza di azioni che un sistema esegue per produrre un risultato osservabile e di valore per uno specifico attore. Qui di seguito verranno riportati i casi d' uso più importanti del progetto.

UC1 : inizializzazione della scena 3D

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : utente

Parti interessate e interessi :

utente: vuole poter navigare in una scena 3D

Pre-condizioni : nessuna

Post-condizioni : il sistema inizializza i componenti base della scena 3D

Scenario principale di successo :

1. l' utente si collega alla pagina desiderata
2. il sistema viene avviato

3. il sistema inizializza i componenti base della scena 3D da navigare (scena, renderer, camera, luci, etc.)
4. il sistema mostra all' utente la pagina di configurazione per poter navigare la scena

UC2 : creazione di tutti i percorsi

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : sistema stesso

Parti interessate e interessi :

utente: vuole poter scegliere un set di percorsi (definiti in base al gruppo utente scelto) su cui poter navigare

Pre-condizioni :
- il sistema é stato avviato
- il sistema é a conoscenza di file di configurazione contenenti le informazioni sui vari percorsi disponibili
- il sistema é a conoscenza di un file di configurazione contenente le dipendenze tra i percorsi e i vari gruppi di utenze

Post-condizioni : il sistema crea i tutti i percorsi disponibili

Scenario principale di successo :

1. il sistema legge i file di configurazione e in base alle informazioni ricavate crea tutti i percorsi

UC3 : creazione del soffitto

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : sistema stesso

Pre-condizioni : il sistema é stato avviato

Post-condizioni : viene aggiunto il soffitto al modello 3D

Scenario principale di successo :

1. il sistema crea il soffitto

UC4 : creazione del pavimento

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : sistema stesso

Pre-condizioni : il sistema é stato avviato

Post-condizioni : viene aggiunto il pavimento al modello 3D

Scenario principale di successo :

1. il sistema crea il pavimento

UC5 : creazione degli oggetti

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : sistema stesso

Pre-condizioni :
- il sistema é stato avviato
- il sistema é a conoscenza di file di configurazione contenenti le informazioni sui vari oggetti da inserire nel modello

Post-condizioni : vengono aggiunti gli oggetti al modello 3D

Scenario principale di successo :

1. il sistema legge i file di configurazione e in base alle informazioni ricavate crea i vari oggetti

UC6 : creazione delle pareti

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : sistema stesso

Pre-condizioni :
- il sistema é stato avviato
- il sistema é a conoscenza di file di configurazione contenenti le informazioni sulle pareti

Post-condizioni : vengono aggiunte le pareti al modello 3D

Scenario principale di successo :

1. il sistema legge i file di configurazione e in base alle informazioni ricavate crea le pareti

UC7 : visualizza i gruppi di percorsi disponibili

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : utente

Pre-condizioni :
- il sistema é stato avviato
- il sistema ha a disposizione i vari gruppi di percorsi
- il sistema é a conoscenza di un file di configurazione contenente le dipendenze tra i gruppi di percorsi e le utenze

Post-condizioni : all' utente viene sbloccata la possibilità di effettuare la scelta sul gruppo di percorsi che vuole navigare

Scenario principale di successo :

1. l' utente sceglie di quale gruppo di utenze vuole vedere tutti i percorsi disponibili
2. il sistema rende possibile all' utente la scelta del gruppo di percorsi su cui fare la navigazione

UC8 : visualizza i percorsi disponibili

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : utente

Pre-condizioni :
- il sistema é stato avviato
- il sistema ha a disposizione i vari gruppi di percorsi
- il sistema é a conoscenza di un file di configurazione contenente le dipendenze tra i gruppi di percorsi e le utenze
- l' utente ha effettuato la scelta su una utenza

Post-condizioni : all' utente viene sbloccata la possibilità di poter scegliere un percorso su cui navigare

Scenario principale di successo :

1. l' utente sceglie un gruppo di percorsi
2. il sistema rende possibile all' utente la possibilità di andare al modello 3D e fare la navigazione
3. il sistema rende possibile all' utente la scelta del percorso (inizio e fine) su cui fare la navigazione

Estensioni :

*a. in qualsiasi momento l' utente decide di cambiare l' utenza
1. il sistema aggiorna i gruppi di percorsi disponibili per quella utenza
2. se l' utente aveva in selezione un gruppo di percorsi non disponibile per quella utenza, il gruppo viene settato al primo disponibile della nuova utenza

UC9 : naviga la scena 3D

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : utente

Pre-condizioni :
- il sistema é stato avviato
- il sistema ha a disposizione i vari gruppi di percorsi
- il sistema é a conoscenza di un file di configurazione contenente le dipendenze tra i gruppi di percorsi e le utenze

- l' utente ha effettuato la scelta su una utenza
- l' utente ha effettuato la scelta su un set di percorsi

Post-condizioni : l' utente naviga la scena 3D fino al punto di fine scelto

Scenario principale di successo :

1. l' utente sceglie un percorso (inizio e fine) su cui navigare
2. l' utente sceglie di andare alla navigazione
3. il sistema mostra all' utente il modello 3D
4. il sistema mostra all' utente la possibilità di cambiare percorso
5. il sistema mostra all' utente la possibilità di cambiare il tipo di navigazione
6. il sistema calcola il percorso più breve tra il punto di inizio e di fine
7. il sistema avvia la navigazione automatica lungo il percorso scelto
8. l' utente visualizza il modello 3D lungo il percorso scelto

Estensioni :

1a. l' utente decide di cambiare l' utenza
    1. il sistema aggiorna i gruppi di percorsi disponibili per quella utenza
    2. se l' utente aveva in selezione un gruppo di percorsi non disponibile per quella utenza, il gruppo viene settato al primo disponibile della nuova utenza
    3. il sistema aggiorna i punti di inizio e fine disponibili per l' attuale gruppo di percorsi selezionato
    4. se l' utente aveva in selezione un punto di inizio e/o fine che non é disponibile per il gruppo di percorsi attuale, il sistema lo reimposta a quello di default

2a. l' utente decide di cambiare il gruppo di percorsi disponibile per l' attuale utenza selezionata
    1. il sistema aggiorna i punti di inizio e fine disponibili per l' attuale gruppo di percorsi selezionato
    2. se l' utente aveva in selezione un punto di inizio e/o fine che non é disponibile per il gruppo di percorsi attuale, il sistema lo reimposta a quello di default

*a. in qualsiasi momento della navigazione l' utente decide di cambiare il punto di inizio e/o il punto di fine
    1. si ritorna al punto 6 dello scenario principale

*b. in qualsiasi momento della navigazione l' utente decide di cambiare il tipo di visualizzazione (da navigazione automatica a quella manuale)
    1. il sistema registra la posizione attuale al momento del cambio
    2. il sistema blocca il movimento automatico
    3. il sistema sblocca all' utente la possibilità di muoversi secondo le

proprie scelte

4. l' utente decide da solo dove muoversi e cosa visitare

*a. in qualsiasi momento l' utente decide di tornare alla navigazione automatica

1. il sistema registra gli attuali punti di inizio e fine selezionati
2. il sistema blocca il movimento manuale
3. si ritorna al punto 6 dello scenario principale

UC10 : interagisci con gli oggetti

Portata : applicazione web per la modellazione e navigazione di un modello 3D

Attore primario : utente

Pre-condizioni :
- il sistema é stato avviato
- il sistema ha a disposizione i vari gruppi di percorsi
- l' utente ha effettuato la scelta su un set di percorsi
- il sistema é a conoscenza di file di configurazione contenenti le informazioni sui vari oggetti da inserire nel modello
- l' utente sta navigando la scena 3D

Post-condizioni : l' utente interagisce con l' oggetto scelto

Scenario principale di successo :

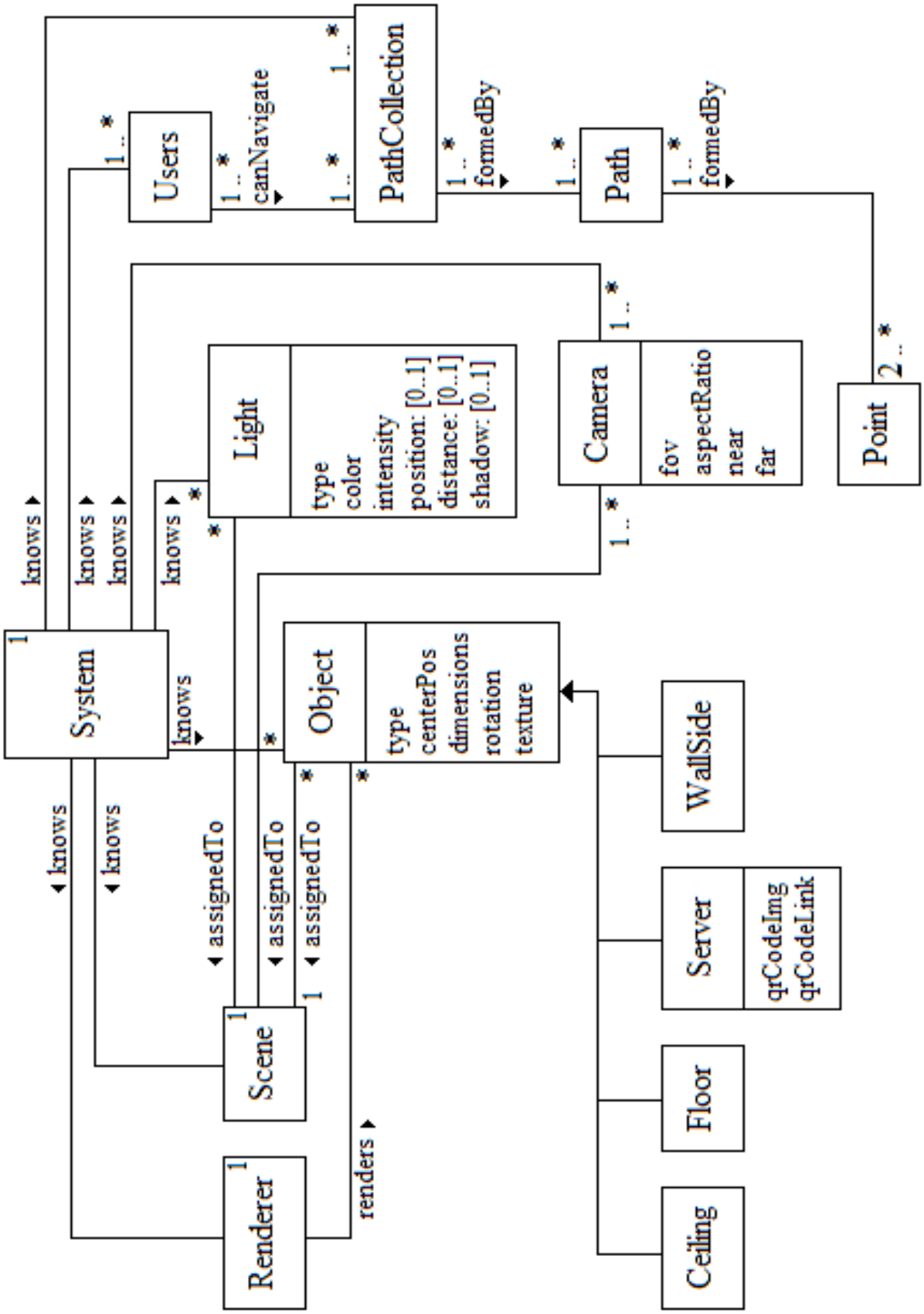
1. l' utente sceglie un oggetto con cui interagire
2. il sistema attiva l' interazione secondo quanto definito nella configurazione dell' oggetto

Estensioni :

2a. non ci sono interazioni per il determinato oggetto

1. il sistema si comporta come se l' utente non avesse scelto di interagire con il determinato oggetto

1.2 : Modello di dominio



{05} – modelloDominio

Nell' analisi orientata agli oggetti, la modellazione del dominio ([12]) ha lo scopo di descrivere le informazioni della reatltà d' interesse secondo una rappresentazione concettuale ed ad oggetti (la realtà d' interesse viene chiamata *dominio del problema* – i diagrammi vengono chiamati *modelli*, mentre i formalismi per esprimere tali modelli vengono chiamati *linguaggi*, ad esempio linguaggio UML ([12])). Il modello di dominio é una rappresentazione visuale di classi concettuali o di oggetti del mondo reale, nonché delle relazioni tra di essi. Il modello di dominio viene principalmente creato per comprendere il sistema da realizzare e il suo vocabolario. Nel seguente modello non vengono rappresentate le associazioni "corrente".

1.3 : Diagrammi di sequenza di sistema

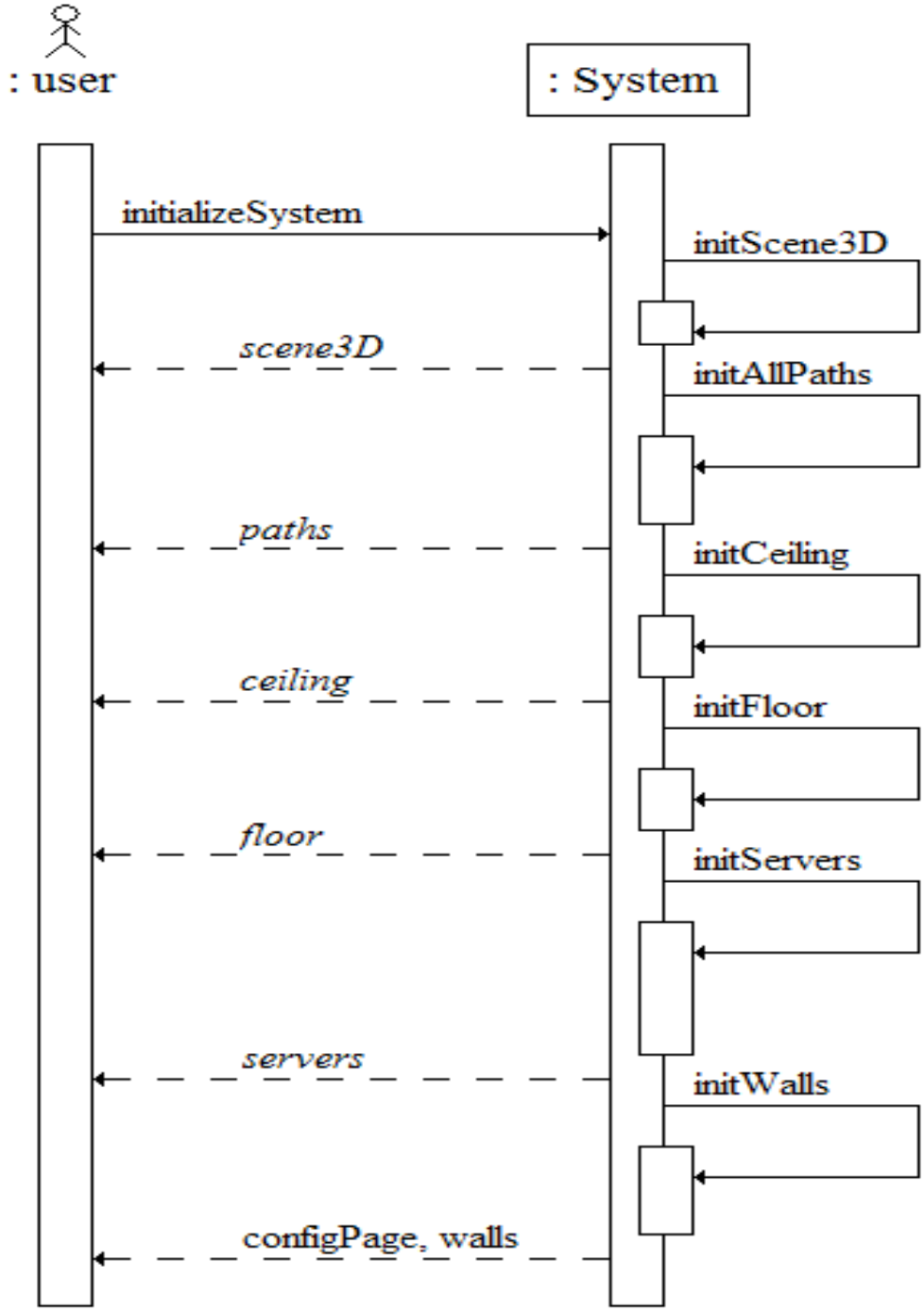
Serve a mostrare gli eventi (non le operazioni) di input e output del sistema in discussione: descrive le sue funzioni, gli attori che interagiscono con esso e le interazioni esterne con altri sistemi. Gli attori, interagendo con il sistema generano degli eventi di sistema (rappresentano le richieste fatte al sistema, solitamente da un' attore) per richiedere l' esecuzione di operazioni di sistema (trasformazione / interrogazione che il sistema può essere chiamato ad eseguire). I diagrammi in UML descrivono queste iterazioni (descrivono un solo caso d' uso alla volta – utili per scenari principali di successo o quelli alternativi più complessi). Gli eventi vengono descritti come intenti, non come azioni.

Come spesso succede proseguendo con l' analisi della realtà d' interesse, é nata la necessità di dover modificare il materiale scritto in una fase precedente; in questo caso verranno modificati i casi d' uso: la modifica riguarda l' unione di UC1, UC2, UC3, UC4, UC5 e UC6 in un unico caso d' uso e l' unione di UC7, UC8 e UC9 in un altro unico caso d' uso. I diagrammi di sequenza di sistema (SSD) faranno pertanto riferimento ai nuovi casi d' uso.

Di seguito verranno mostrati i tre SSD creati con una piccola descrizione.

SSD1 : inizializzazione del sistema

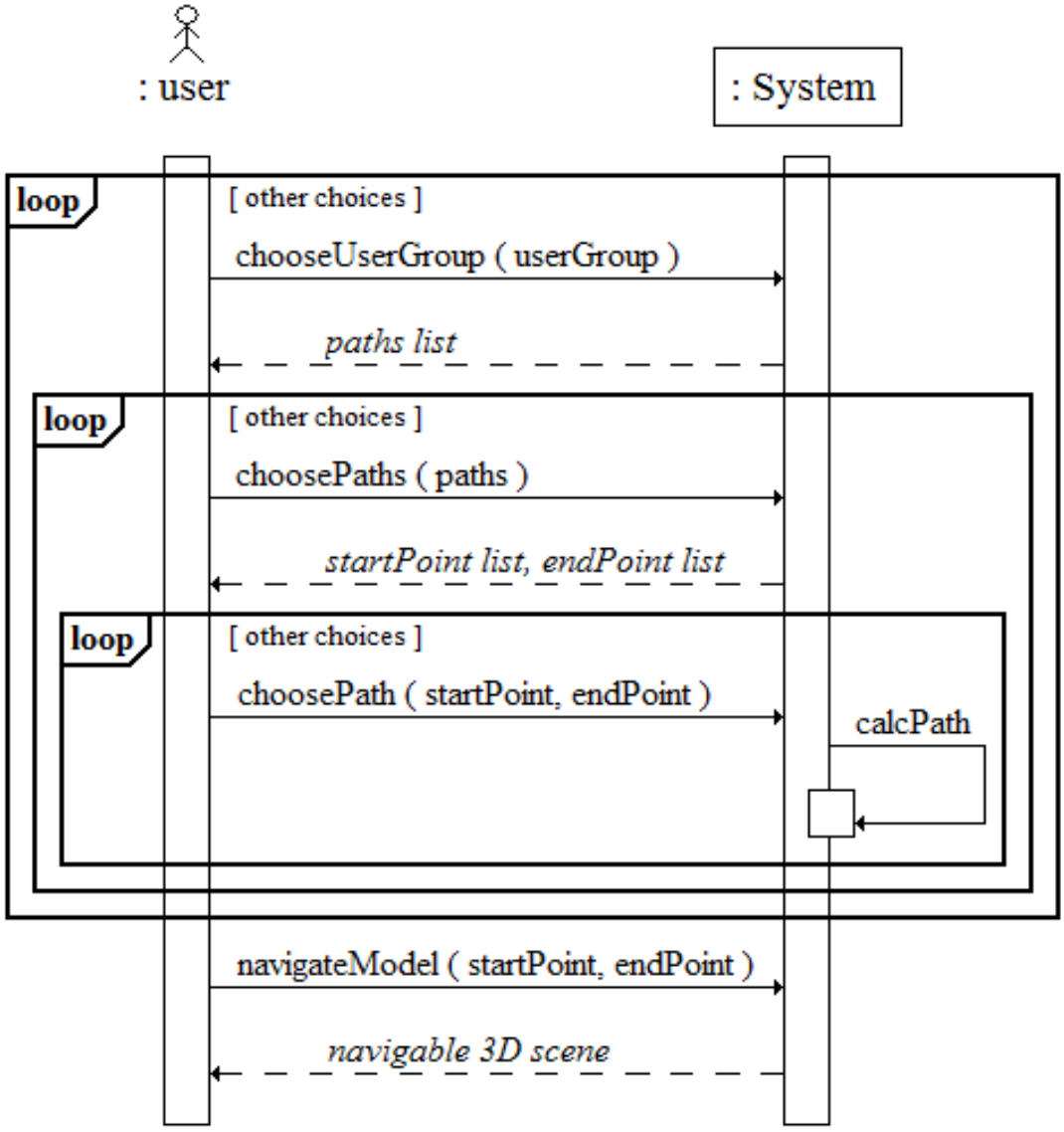
L' utente fa partire il sistema aprendo la pagina; quest' ultimo inizializza la scena e il modello (pareti, mura, oggetti, soffitto e pavimento) che a mano a mano viene disegnato, anche se rimane nascosta all' utente. Al completamento di tutto il processo di creazione, il sistema mostra all' utente la pagina di configurazione in modo che possa decidere il percorso su cui navigare.



{06} – ssd1

SSD2 : configurazione e navigazione

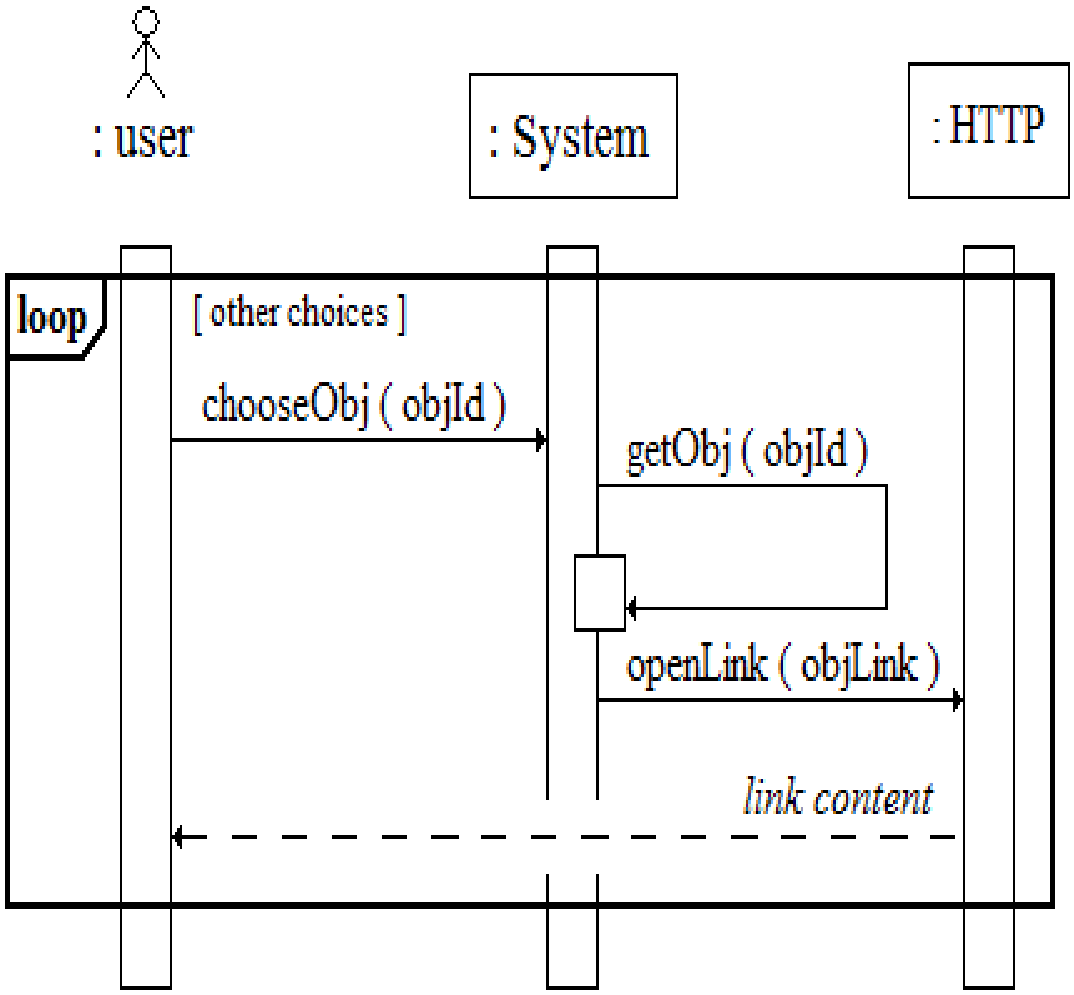
L' utente sceglie dalla schermata di configurazione iniziale una utenza; il sistema risponde mostrando una lista di percorsi; a questo punto quindi l' utente può scegliere uno dei tanti gruppi di percorsi disponibili in base al gruppo di utenze scelto precedentemente; il sistema mostra i punti di inizio e fine in modo che l' utente possa scegliere un percorso su cui navigare. La scelta del percorso e le due scelte precedenti (l' utenza e il gruppo di percorsi) vanno in loop, ovvero l' utente può cambiare le proprie scelte quante volte vuole. Infine si ha la possibilità di navigare il modello; il sistema calcola il percorso più breve tra il punto di inizio e fine selezionati e mostra il modello 3D iniziando la navigazione.



{07} – ssd2

SSD3 : interazione con gli oggetti

Durante la navigazione della scena, l' utente può decidere di interagire con gli oggetti che vede; sceglie l' oggetto d' interesse e il sistema, dopo un controllo se quest' ultimo ha interazioni disponibili, manda la richiesta a HTTP e all' utente quindi viene aperto il link relativo all' oggetto scelto. L' operazione può essere ripetute tante volte quante volte vuole l' utente (ed é permessa l' iterazione anche con oggetti già scelti precedentemente).



{08} – ssd3

1.4 : Contratti

I contratti delle operazioni ([12]) consentono di descrivere il comportamento del sistema, in termini dell' effetto prodotto dall' esecuzione delle operazioni di sistema, richieste tramite gli eventi. Esistono 3 soli cambiamenti di stato possibili: creazione / cancellazione di un' istanza; cambiamento del valore di un attributo; associazione formata o spezzata. Per operazioni molto complesse o poco chiare conviene sempre scrivere dei contratti.

Durante la creazione degli SSD, alcuni casi d' uso sono stati modificati e pertanto i seguenti contratti faranno riferimento ai nuovi casi d' uso (gli stessi usati per i diagrammi di sequenza). Inoltre nel modello di dominio é stato ritenuto opportuno cambiare il nome dell' associazione tra "System" e "PathCollection": da "knows" é diventata "generate"; infine come ulteriore modifica al modello, é stata aggiunta la classe concettuale "Config" collegata a "System" mediante "knows" e l' associazione "navigateOn" tra "Camera" e "Path".

CO1.1 : initializeSystem
Operazione : init()

Riferimenti : UC1

Pre-condizioni : nessuna

Post-condizioni :
- é stata creata l' istanza sy singleton di System

CO1.2 : initializeScene3D
Operazione : initScene()

Riferimenti : UC1

Pre-condizioni :

- esiste l' istanza sy di System

Post-condizioni :

- é stata creata l' istanza s singleton di Scene
- é stata creata l' istanza r singleton di Renderer
- é stata creata almeno un' istanza c di Camera
- s é stato associato a sy mediante “corrente”
- r é stato associato a sy mediante “corrente”
- c é stato associato a sy mediante “corrente”
- ogni istanza l di Light esistente é stata associata a s mediante “assignedTo”
- ogni istanza c di Camera é stata associata a s mediante “assignedTo”

CO1.3 : initAllPaths

Operazione : initAllPaths()

Riferimenti : UC1

Pre-condizioni :

- esiste l' istanza sy di System
- esiste almeno un' istanza c di Camera

Post-condizioni :

- sono state create tutte le istanze di PathCollection
- sono state create tutte le istanze di Path
- sono state create tutte le istanze di Point
- ogni istanza di PathCollection é stata associata ad almeno un' istanza u di Users mediante “canNavigate”
- ogni istanza di Path é stata associata ad almeno un' istanza di PathCollection mediante “formedBy”
- almeno due istanze di Point sono state associate ad almeno un' istanza di Path mediante “formedBy”
- ogni istanza di Path é stata associata ad ogni istanza di Camera mediante “navigateOn”

CO1.4 : initCeiling

Operazione : initCeiling()

Riferimenti : UC1

Pre-condizioni :

- esiste l' istanza sy di System
- esiste l' istanza corrente r di Renderer
- esiste l' istanza corrente s di Scene

Post-condizioni :

- sono state creare tutte le istanze di Ceiling
- ogni istanza di Ceiling é stata associata a sy di System mediante “knows”
- ogni istanza di Ceiling é stata associata a r di Renderer mediante “renders”
- ogni istanza di Ceiling é stata associata a s di Scene mediante “assignedTo”

CO1.5 : initFloor

Operazione : initFloor()

Riferimenti : UC1

Pre-condizioni :

- esiste l' istanza sy di System
- esiste l' istanza corrente r di Renderer
- esiste l' istanza corrente s di Scene

Post-condizioni :

- sono state creare tutte le istanze di Floor
- ogni istanza di Floor é stata associata a sy di System mediante “knows”
- ogni istanza di Floor é stata associata a r di Renderer mediante “renders”
- ogni istanza di Floor é stata associata a s di Scene mediante “assignedTo”

CO1.6 : initServers

Operazione : initServers()

Riferimenti : UC1

Pre-condizioni :

- esiste l' istanza sy di System
- esiste l' istanza corrente r di Renderer
- esiste l' istanza corrente s di Scene

Post-condizioni :

- sono state creare tutte le istanze di Server
- ogni istanza di Server é stata associata a sy di System mediante “knows”
- ogni istanza di Server é stata associata a r di Renderer mediante “renders”
- ogni istanza di Server é stata associata a s di Scene mediante “assignedTo”

CO1.7 : initWalls

Operazione : initWalls()

Riferimenti : UC1

Pre-condizioni :

- esiste l' istanza sy di System
- esiste l' istanza corrente r di Renderer
- esiste l' istanza corrente s di Scene

Post-condizioni :

- sono state creare tutte le istanze di Wall
- ogni istanza di Wall é stata associata a sy di System mediante “knows”
- ogni istanza di Wall é stata associata a r di Renderer mediante “renders”
- ogni istanza di Wall é stata associata a s di Scene mediante “assignedTo”

CO2.1 : chooseUserGroup

Operazione : chooseUserGroup (userGroup)

Riferimenti : UC2

Pre-condizioni :

- esiste l' istanza sy di System

Post-condizioni :

- l' istanza u di Users sulla base di 'userGroup' viene associata con sy di System mediante “corrente”

CO2.2 : choosePaths

Operazione : choosePaths (paths)

Riferimenti : UC2

Pre-condizioni :

- esiste l' istanza sy di System
- esiste l' istanza corrente u di Users

Post-condizioni :

- l' istanza pc di PathCollection sulla base di 'paths' viene associata con sy di System mediante “corrente”

CO2.3 : choosePath

Operazione : choosePath (startPoint, endPoint)

Riferimenti : UC2

Pre-condizioni :

- esiste l' istanza sy di System
- esiste l' istanza corrente u di Users
- esiste l' istanza corrente pc di PathCollection

Post-condizioni :

- l' istanza p di Path sulla base di 'startPoint' e 'endPoint' viene associata con sy di System mediante “corrente”

CO2.4 : navigateModel

Operazione : navigateModel (startPoint, endPoint)

Riferimenti : UC2

Pre-condizioni :

- esiste l' istanza sy di System
- esiste l' istanza corrente u di Users
- esiste l' istanza corrente pc di PathCollection
- esiste l' istanza corrente p di Path
- esiste almeno un' istanza c di Camera

Post-condizioni :

- ogni istanza c di Camera viene associata con p di Path mediante “navigateOn”

CO3.1 : chooseObj

Operazione : chooseObj (objId)

Riferimenti : UC3

Pre-condizioni :

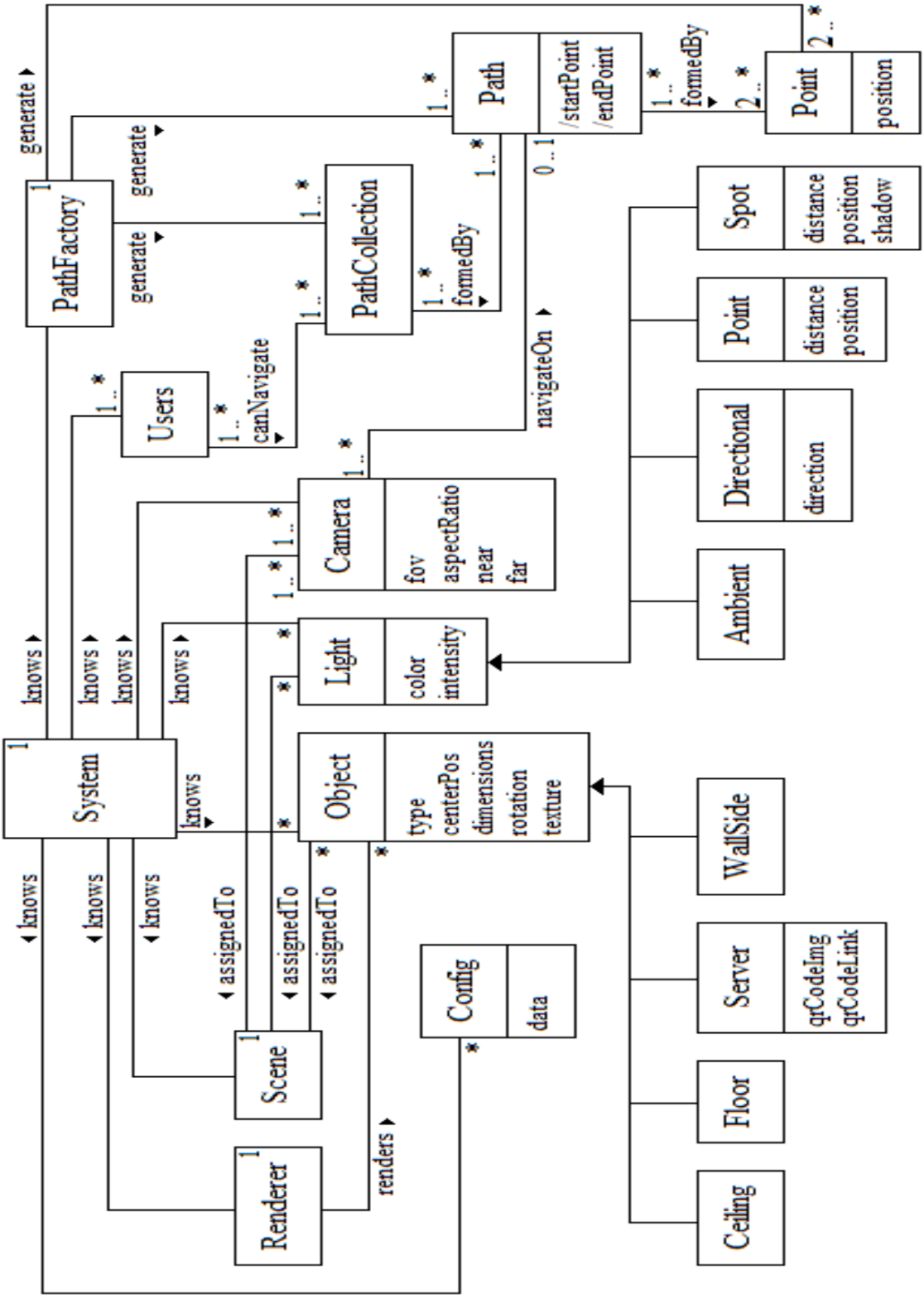
- esiste l' istanza sy di System
- esiste l' istanza corrente p di Path
- esiste almeno un' istanza c di Camera
- esiste almeno un' associazione “corrente” tra c di Camera e p di Path corrente

Post-condizioni :

- l' istanza o di Object viene associata a sy di System mediante “corrente” sulla base di 'objId'

1.5 : Modello di dominio finale

In seguito a varie modifiche durante la fase di analisi (e anche in fase di progettazione), per completezza viene mostrato il modello di dominio finale.



{09} – modelloDominioFinale

Apendice 2 : Progettazione ad oggetti

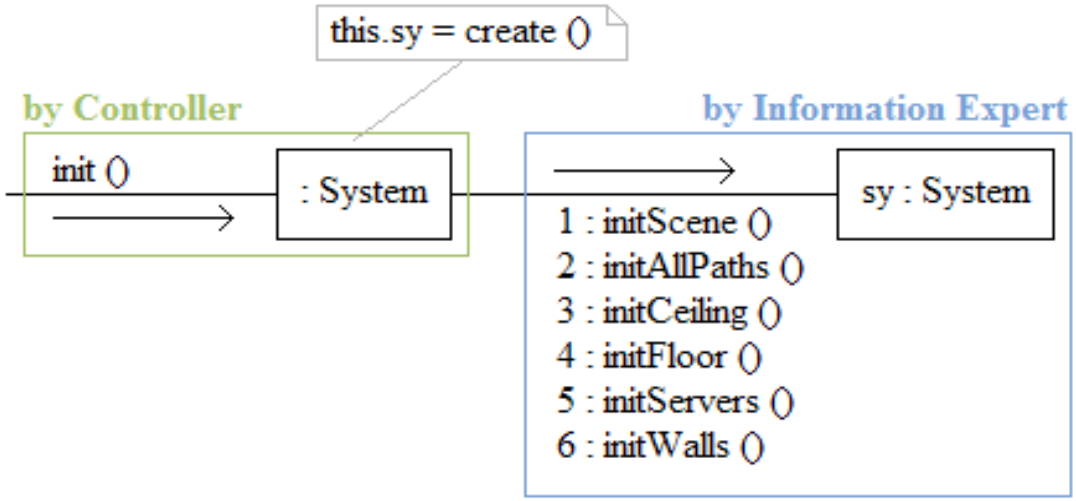
2.1 : Diagrammi di comunicazione e DCD parziali

Seguendo il paradigma della programmazione a oggetti e tenendo presente le linee guida fornite dai pattern GRASP (General Responsability Assignment Software Patterns) e GOF (Gang Of Four) ([12]) (descrivono principi fondamentali per la progettazione di oggetti e l' assegnazione di responsabilità ad oggetti software ispirati al modello di dominio), e dalla RDD (Responsability Driven Design) ([12]) é stata fatta la progettazione; su ogni diagramma di comunicazione sono indicati i pattern GRASP presi in considerazione per giustificare le scelte prese. Durante questa fase, la prima scelta da fare é stata l' individuazione dell' oggetto che si interconnette allo strato UI e che ha quindi il compito di ricevere e coordinare un' operazione di sistema. Seguendo il pattern Controller la responsabilità é stata assegnata a System, ovvero ad un oggetto che rappresenta il sistema complessivo (un *facade controller*). In seguito, per non assegnare troppe responsabilità al facade controller, seguendo i pattern di Information Expert, Low Copling e High Cohesion é stato deciso di assegnare la responsabilità di creare gli Oggetti al Renderer, mentre per gli stessi principi la creazione dei vari path (e ciò che gli riguarda) viene assegnata ad una classe artificiale (o di convenienza) che non rappresenta nessun concetto del dominio, ovvero una *Pure Fabrication* o una *Factory*.

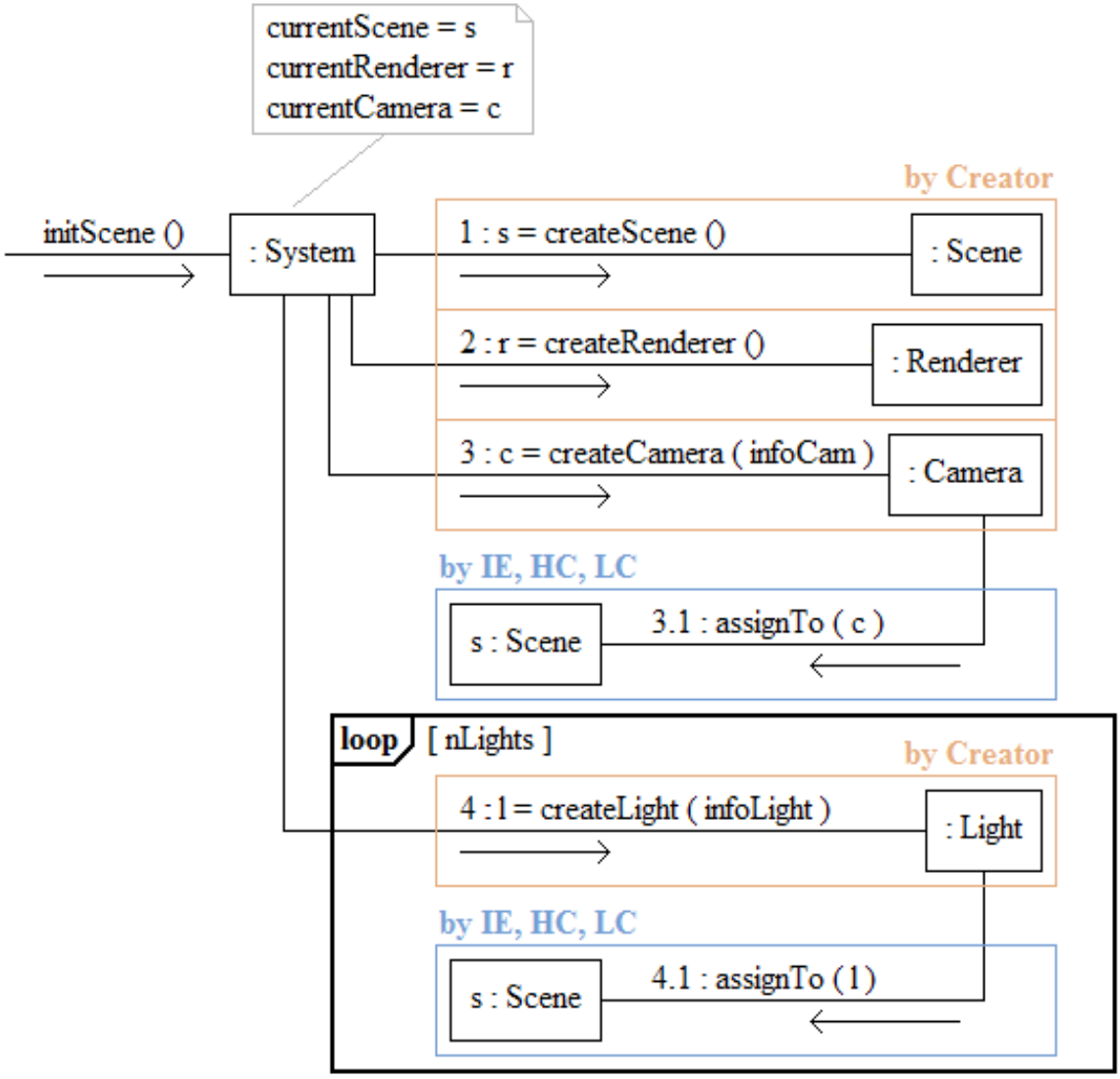
Raggruppando le somiglianze dei vari Oggetti, si é deciso di usare il polimorfismo per avere una classe che potesse essere la generalizzazione di questi; in questo modo si ha la possbilità di mettere in comune molti dei metodi delle varie sottoclassi; inoltre l' eventuale aggiunta o modifica di ulteriori Oggetti risulta più facile, perchè quest' ultimi sono meno accopiati con il resto del progetto (uso del pattern Low Copling).

In fase di progettazione viene assunto come prerequisito che l' applicazione é stata appena messa in funzione, ovvero l' utente si trova a dover aspettare che vengano create tutte le istanze di quasi tutte le classi d' interesse.

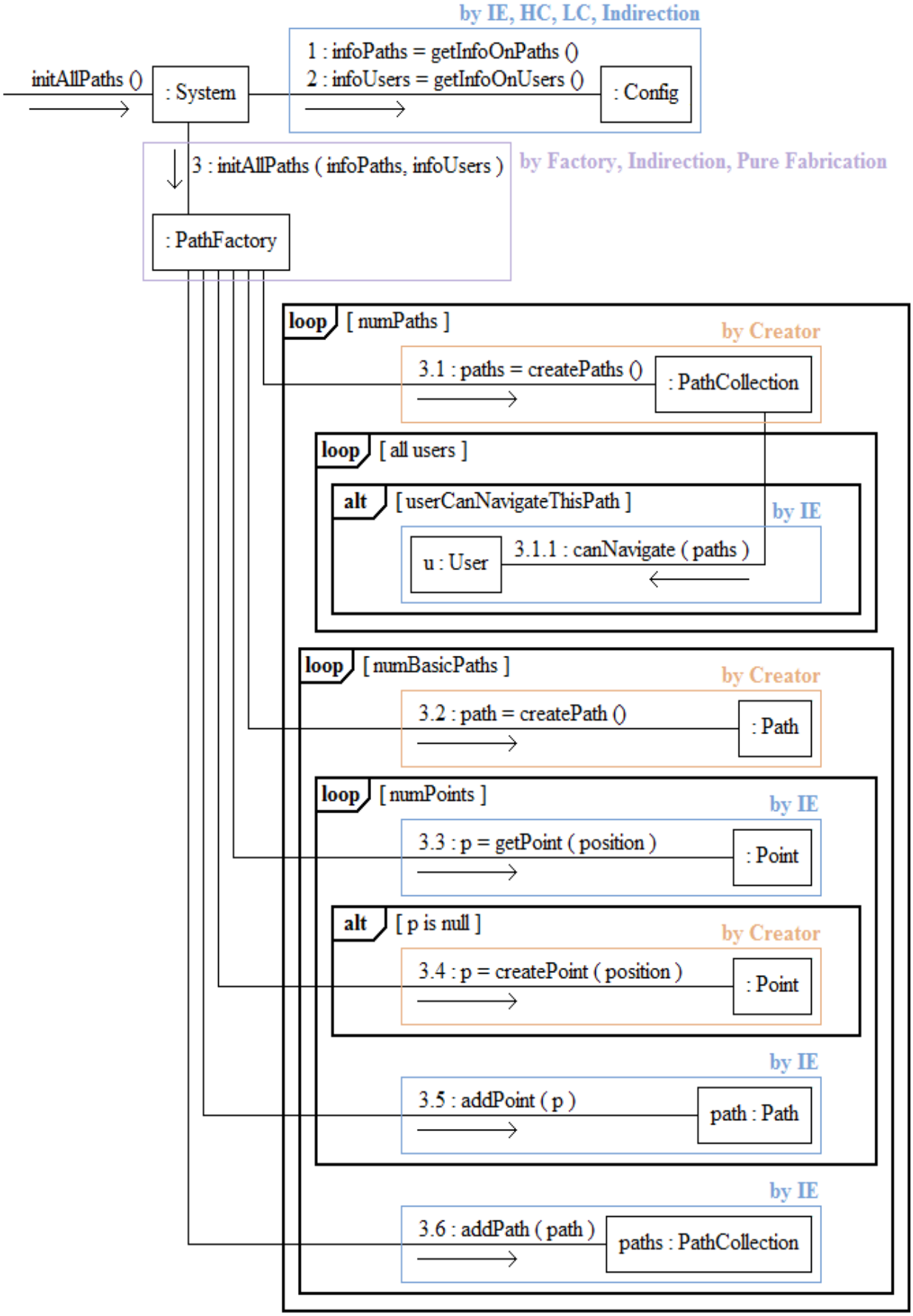
2.1.1 : avvio sistema



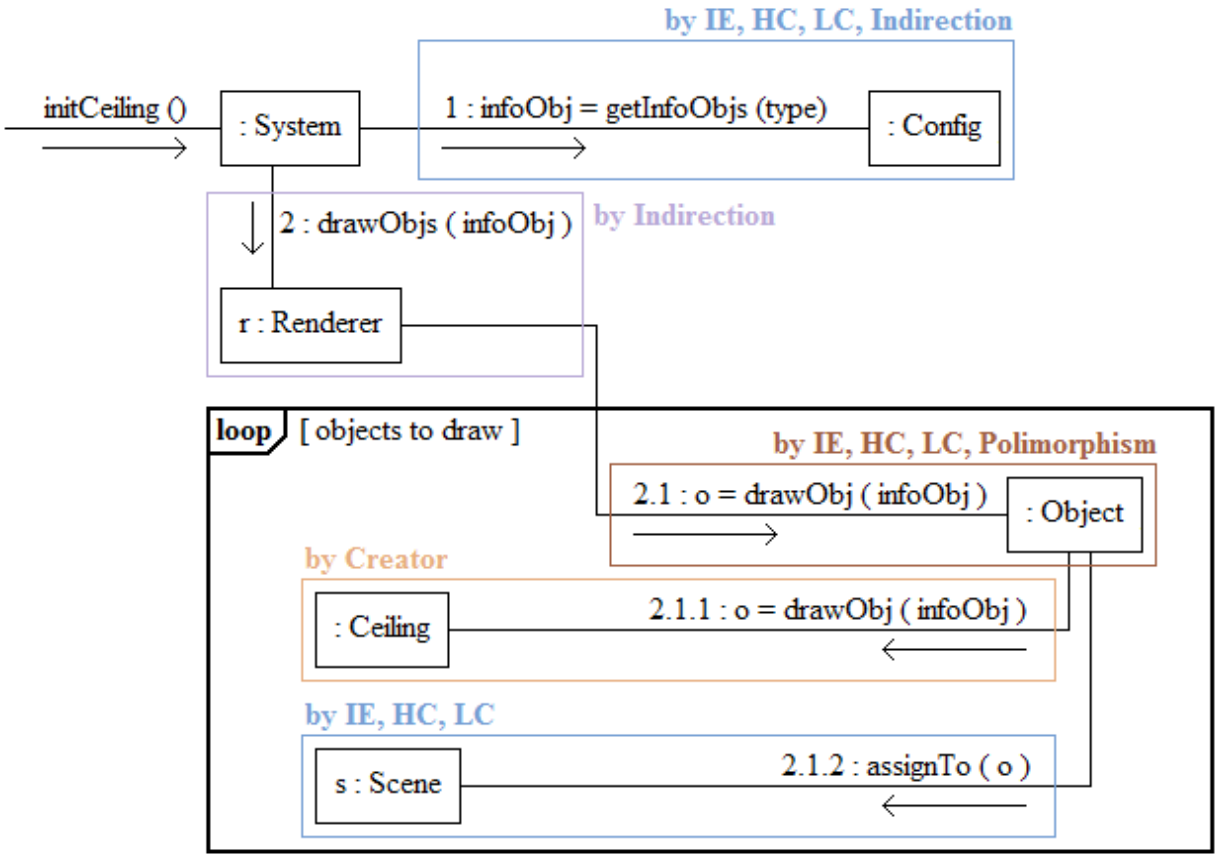
{10} – dc1



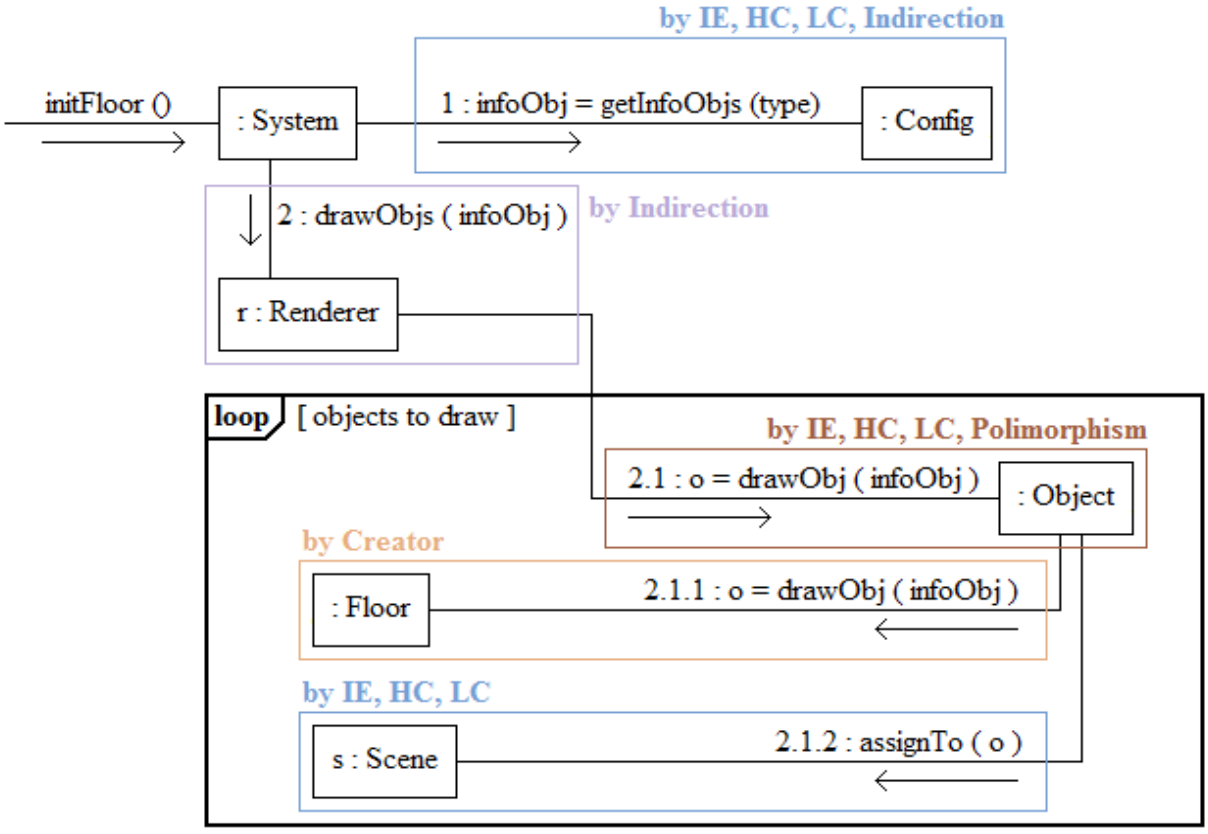
{11} – dc1_1


by IE, HC, LC, Indirection
initAllPaths ()
: System
1 : infoPaths = getInfoOnPaths ()
2 : infoUsers = getInfoOnUsers ()
: Config
3 : initAllPaths (infoPaths, infoUsers)
by Factory, Indirection, Pure Fabrication
: PathFactory
loop
[numPaths]
by Creator
3.1 : paths = createPaths ()
: PathCollection
loop
[all users]
alt
[userCanNavigateThisPath]
by IE
u : User
3.1.1 : canNavigate (paths)
loop
[numBasicPaths]
by Creator
3.2 : path = createPath ()
: Path
loop
[numPoints]
by IE
3.3 : p = getPoint (position)
: Point
alt
[p is null]
by Creator
3.4 : p = createPoint (position)
: Point
by IE
3.5 : addPoint (p)
path : Path
by IE
3.6 : addPath (path)
paths : PathCollection

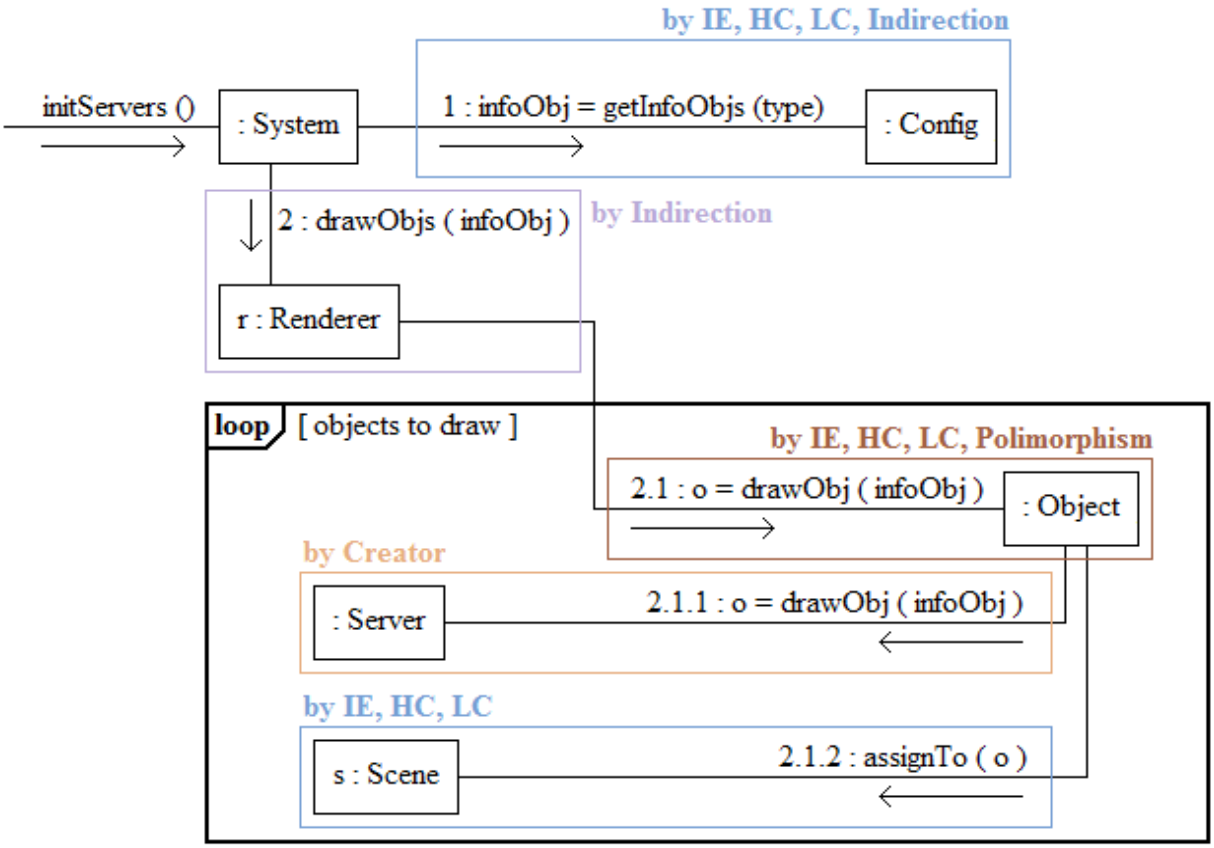
{12} – dc1_2



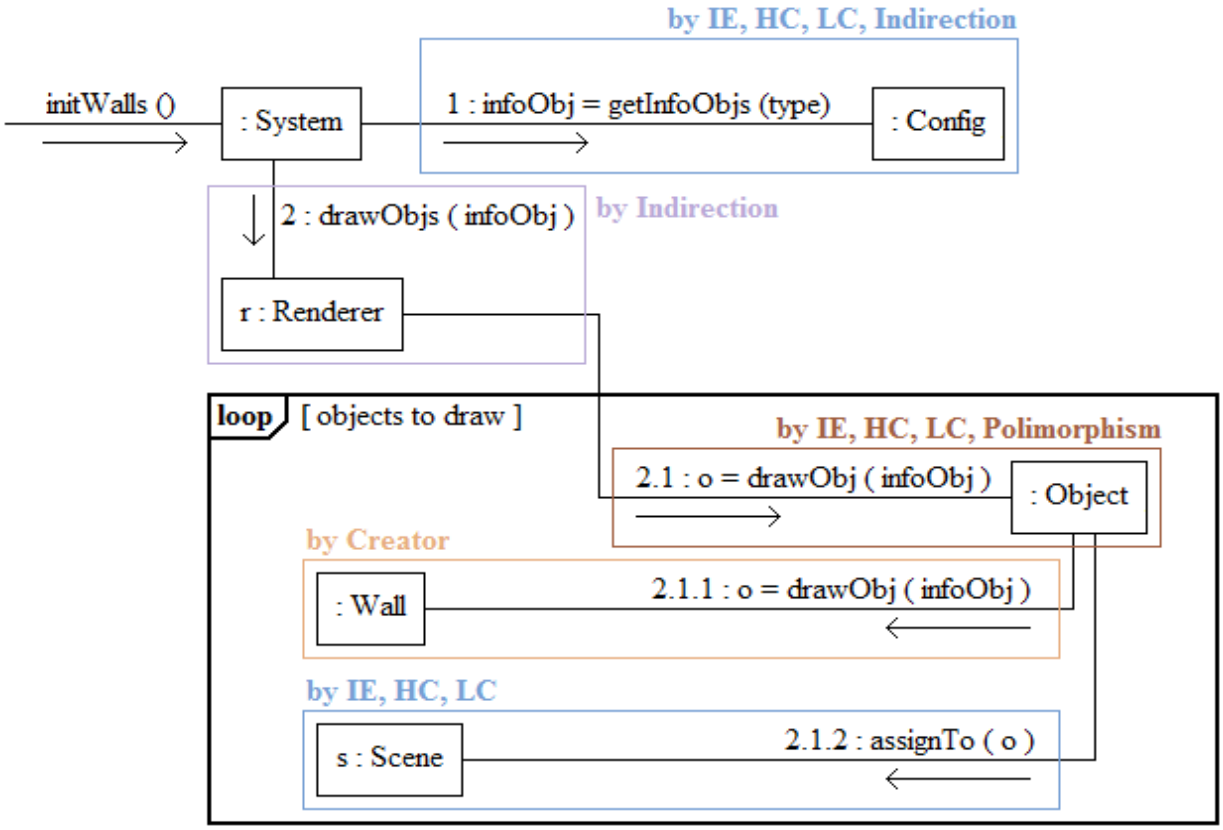
{13} – dc1_3



{14} – dc1_4



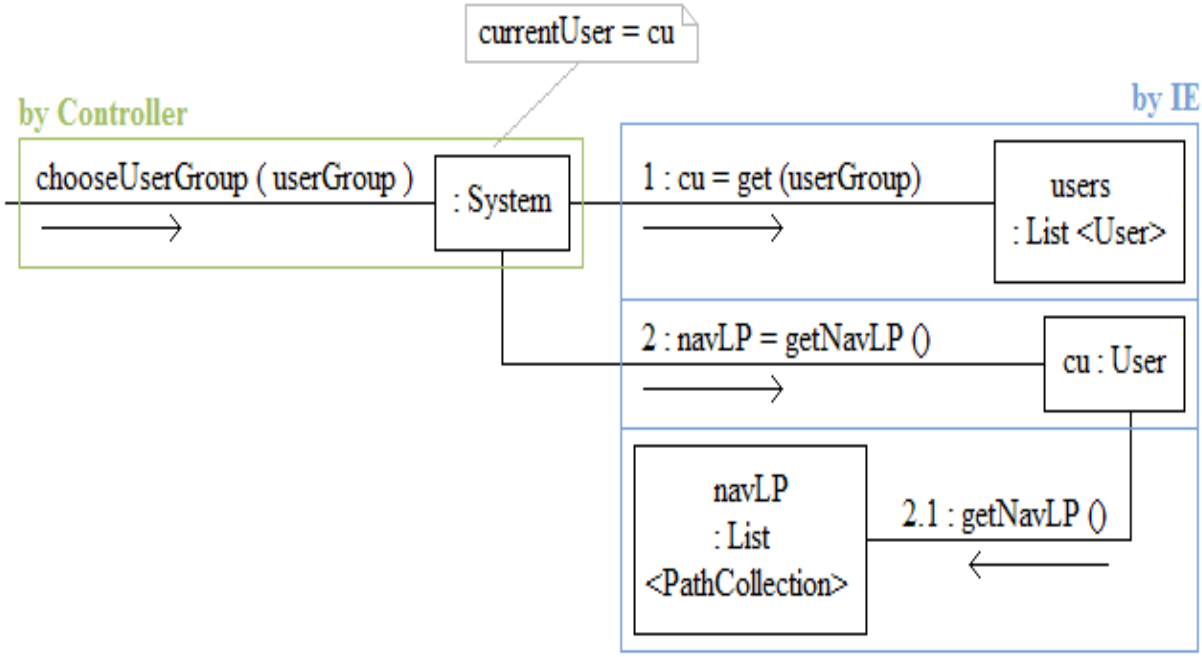
{15} – dc1_5



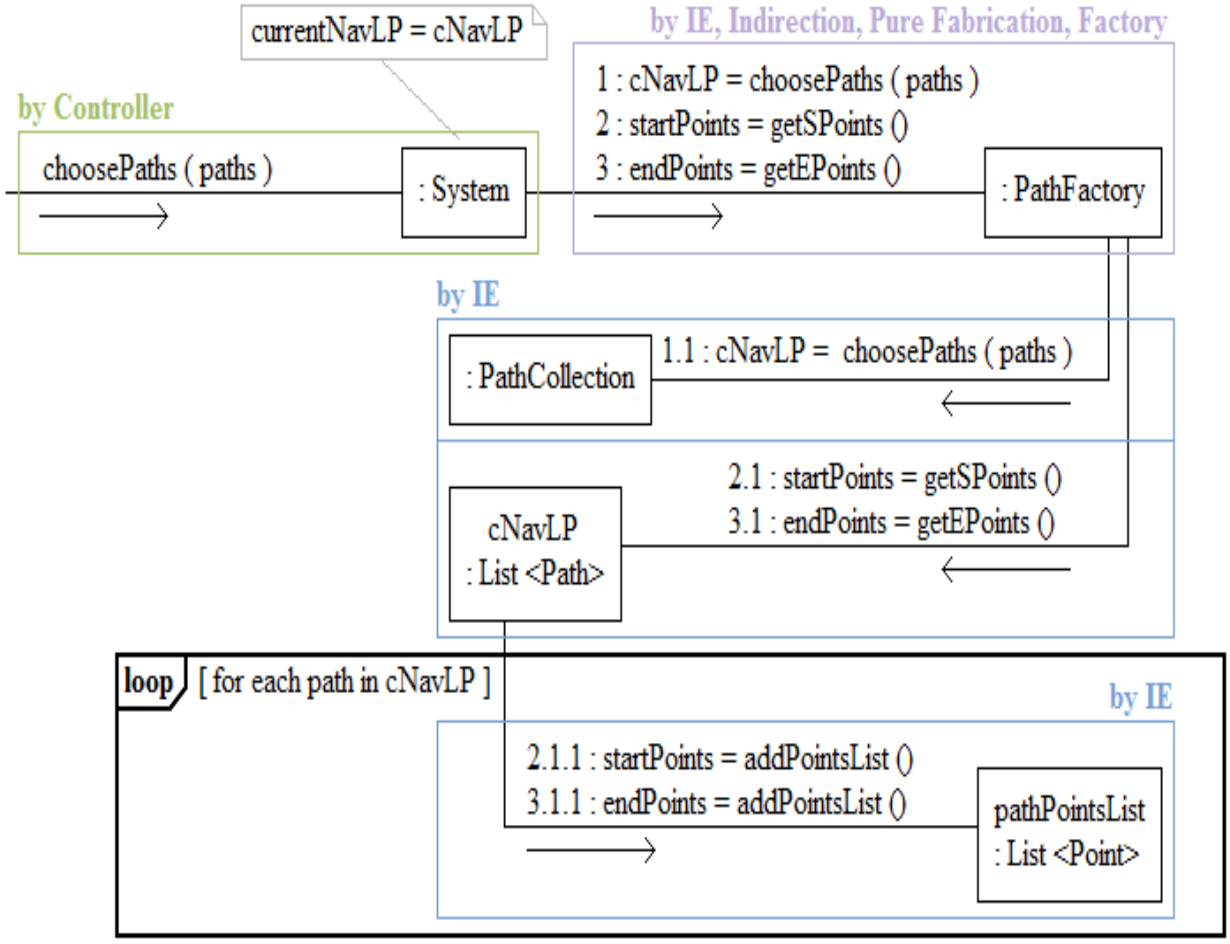
{16} – dc1_6

Avviato il sistema, esso chiede a se stesso di creare tutto ciò che serve per poter navigare la scena 3D (tra cui la scena 3D stessa): vengono creati quindi i muri, il pavimento e soffitto, gli oggetti, le luci e anche tutti i percorsi navigabili; i primi ad essere creati sono la scena, il renderer, la camera e le luci (delle ultime due, per ogni istanza creata, la classe corrispondente richiede alla scena corrente di essere aggiunta) (come mostrato nel diagramma {11}). Successivamente vengono creati tutti i percorsi navigabili (diagramma {12}); seguendo le indicazioni dei pattern *Factory*, *Indirection* e *Pure Fabrication* é stato deciso di creare una classe artificiale *PathFactory* che contenesse tutta la conoscenza riguardante la loro creazione in modo da alleggerire il carico di responsabilità su *System*; inoltre, sempre seguendo le linee guida dei pattern *Information Expert* e *Indirection* e con l' intenzione di dare meno responsabilità al *System*, si é deciso di assegnare la conoscenza necessaria per la creazione di tutti gli oggetti (ovvero la configurazione iniziale) come anche per la correlazione tra i vari user e i percorsi loro disponibili alla classe *Config*. Durante la creazione dei punti facenti parte di un percorso, si é deciso di inserire un controllo per verificare se il punto esiste già nella posizione destinata, in modo da evitare doppioni delle istanze di *Point*. Per togliere ulteriori responsabilità a *System* é stato deciso di assegnare la responsabilità di creare gli oggetti come le pareti ({16}), il soffitto ({13}), il pavimento ({14}) e i vari server ({15}) al *Renderer* (tenendo in considerazione che in Three.js é proprio il renderer che disegna tali oggetti e quindi in un certo senso é come se li creasse); una volta creata l' istanza di un oggetto esso richiede alla scena corrente di essere aggiunto in modo da poter essere visibile all' utente che naviga quest' ultima.

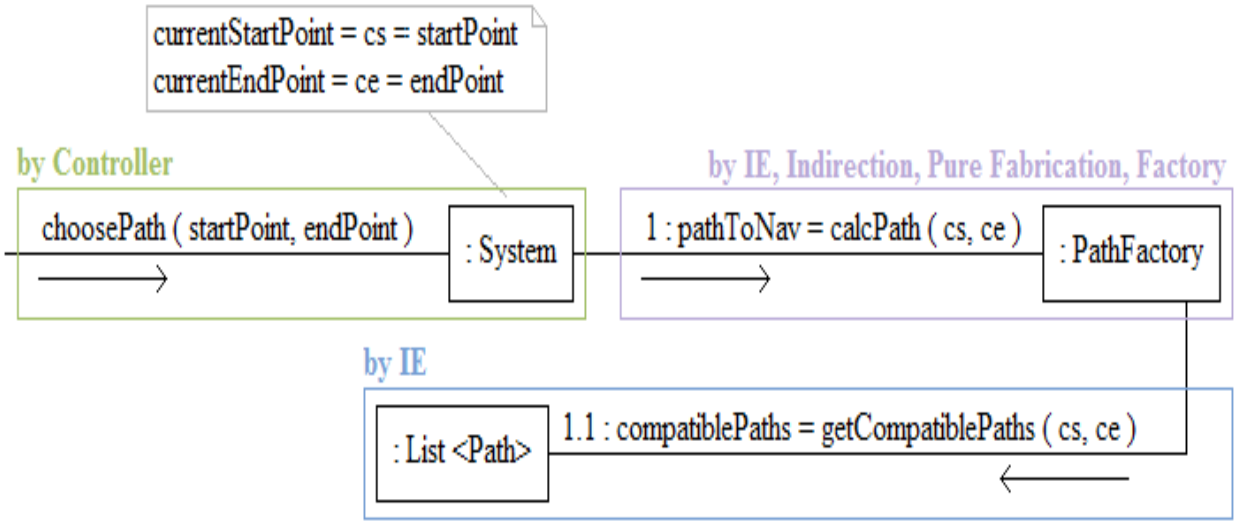
2.1.2 : navigazione modello



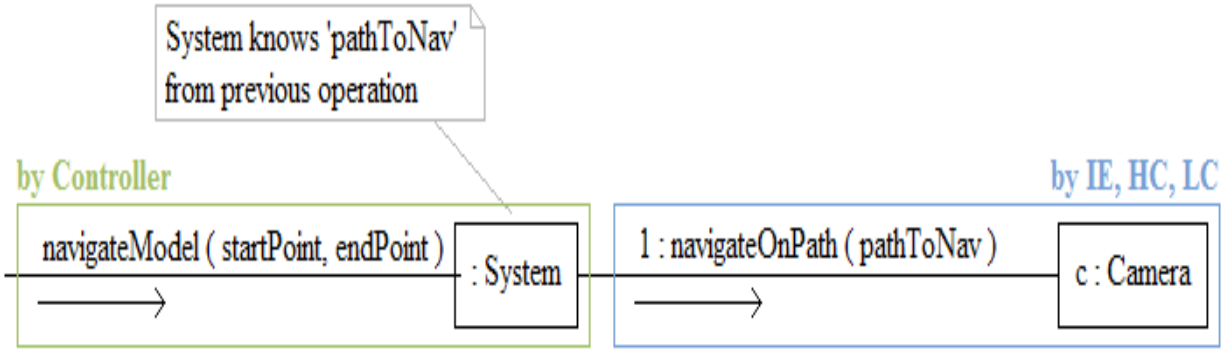
{17} – dc2_1



{18} – dc2_2



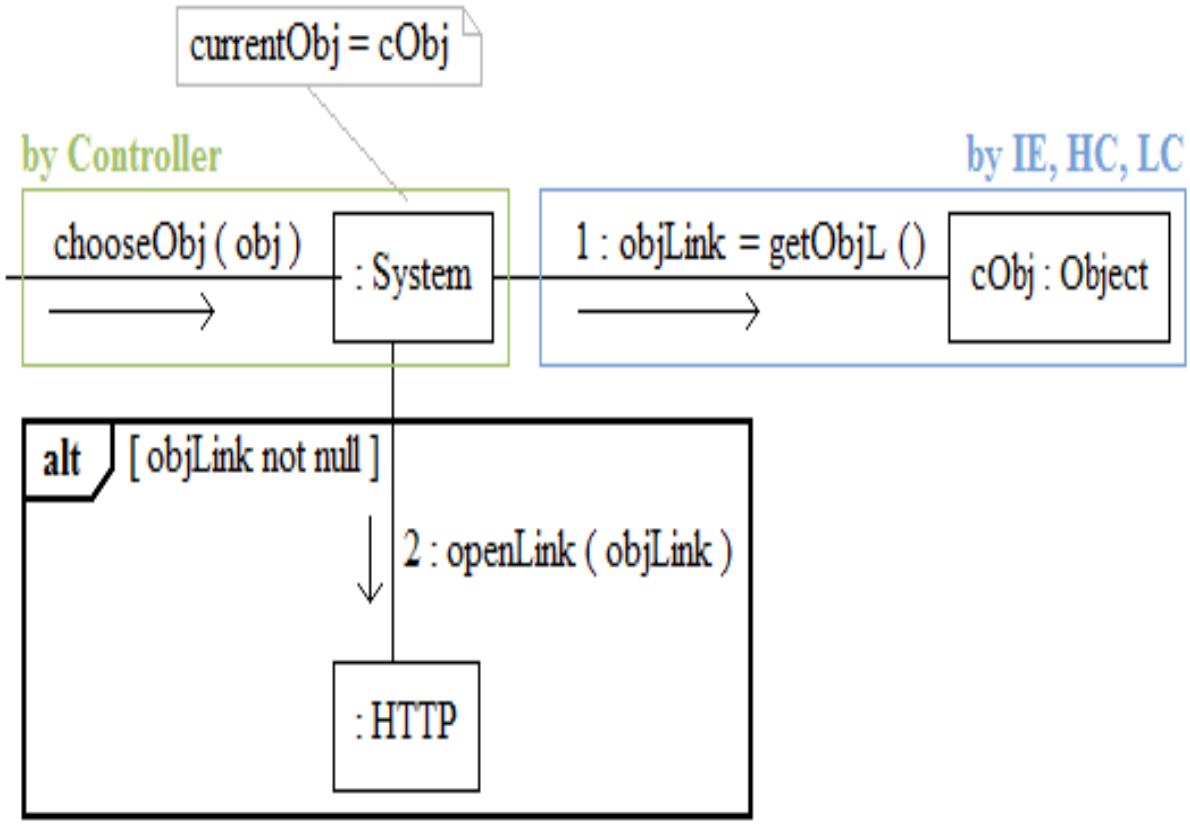
{19} – dc2_3



{20} – dc2_4

Una volta avviato il sistema e avvenuta la creazione della scena 3D, il sistema mostra all' utente la schermata di configurazione per la navigazione del modello; questi quindi può scegliere l' utenza (a seconda della scelta potrà navigare percorsi diversi); effettuata la scelta il sistema chiede a *User* quali gruppi di path sono percorribili dal quel utente e lo mostra all' utente ({17}). Successivamente quindi l' utente può effettuare una scelta su quali gruppi di percorsi percorrere ({18}) e infine scegliere il singolo percorso ({19}). Scelto il percorso (formato da un punto di partenza e uno di arrivo) il sistema richiede a *PathFactory* di calcolare il percorso più breve tra i due punti (verrà usato l' algoritmo di Djikstra). L' utente può cambiare in qualsiasi momento le sue scelte, riavviando le varie operazioni. Dopo aver effettuato tutte le scelte é possibile finalmente navigare e visitare il modello ({20}): la camera viene associata al percorso calcolato dalla *PathFactory* e comincia a ripercorrere i punti che formano il percorso scelto.

2.1.3 : interazione con i server

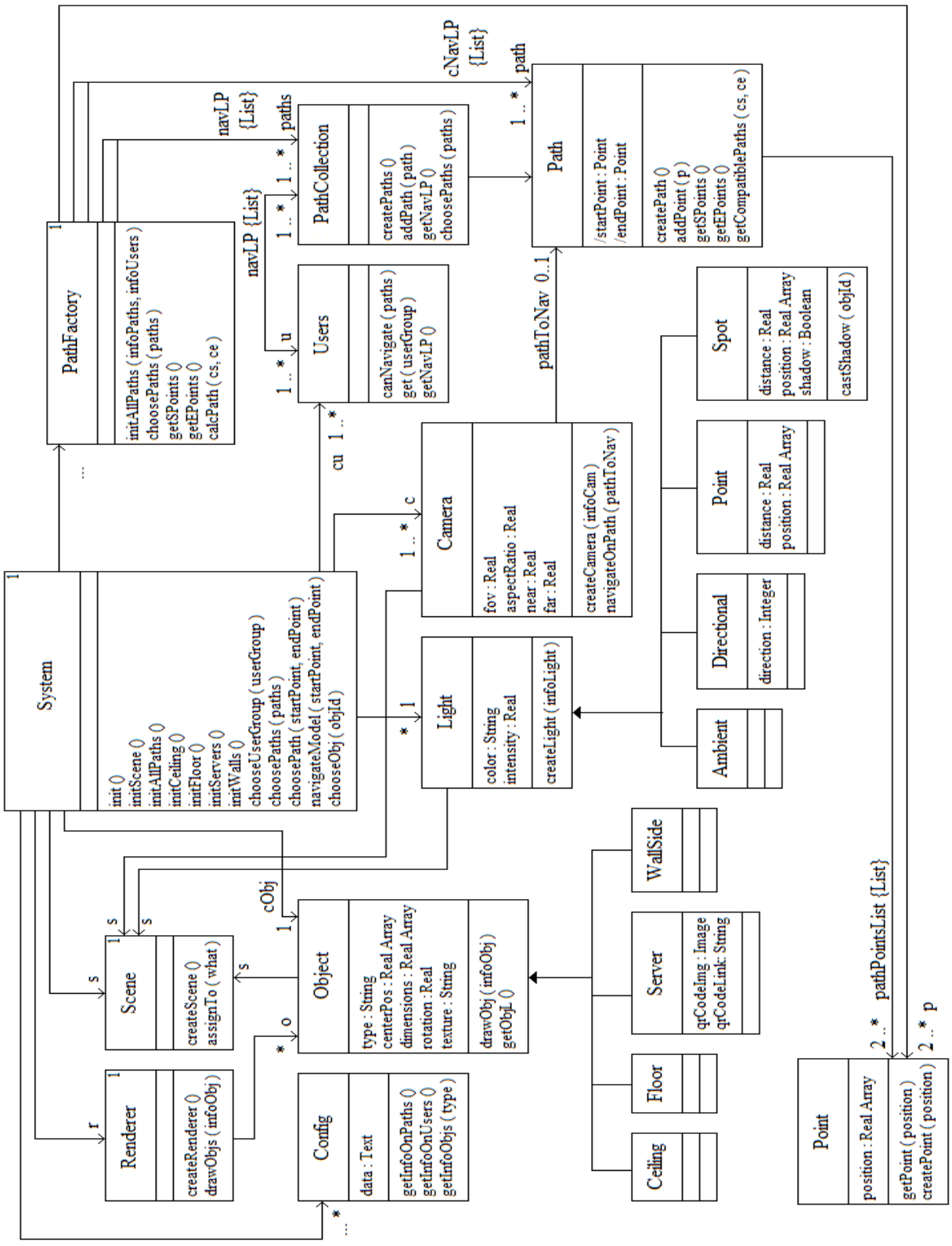


{21} – dc3

Durante la navigazione é possibile interagire con i server che si vede. L' utente seleziona il server di cui vuole avere le informazioni; il sistema richiede quindi a *Object* le informazioni riguardanti il link da aprire dell' oggetto selezionato (che richiede poi ad HTTP di aprire).

2.2 : Diagramma delle classi

Contemporaneamente ai diagrammi d' interazione é stato disegnato il DCD (diagramma delle classi) ([12]).



{22} – dcd

Capitolo 7 : Bibliografia

[1] Learning Three.js: The JavaScript 3D Library for WebGL
Jos Dirksen, 2013

[2] WebGL
http://www.khronos.org/news/press/khronos-releases-final-webgl-1.0-specification

[3] Sito web su WebGL
http://www.html5today.it/

[4] Sito web su WebGL
http://learningwebgl.com/blog/

[5] Esempi di grafica 3D riguardanti webGl
http://www.chromeexperiments.com/webgl

[6] Three.js
http://threejs.org/

[7] Esempi di grafica con Three.js
http://stemkoski.github.io/Three.js/

[8] Librerie riguardanti Three.js
https://github.com/mrdoob/three.js/

[9] Estensioni per Three.js
http://www.threejsgames.com/extensions/

[10] Guida su come usare Blender
http://www.html.it/guide/blender-3d-guida/

[11] Comunità Italiana di Blender
http://www.blender.it/index.php/articles.html

[12] Applicare UML e I Pattern
Caig Larman. 2005, lar05

[13] Geometria solida con Three.js

http://www.chandlerprall.com/2011/12/constructive-solid-geometry-with-three-js/

[14] Programming 3D Applications with HTML5 and WebGL
Tony Parisi, 02/2014, O'Reilly Media

[15] Esportazione di modelli da Blender a Three.js usando ColladaLoader
http://stackoverflow.com/questions/23224940/

[16] Codici hex dei colori
http://www.w3schools.com/tags/ref_colorpicker

[17]