PROGETTO

TESSUTO DIGITALE METROPOLITANO

POR FESR 2014-2020, Azione 1.2.2

Progetto Complesso area ICT della S3

Deliverable 6.1 – TECNOLOGIE PER LA VISUALIZZAZIONE SCALABILE (VERSIONE PRELIMINARE)	
Data di consegna prevista: 6/2020	**Data di consegna effettiva:** 6/2020
Natura: Rapporto	**Versione:** 1.0
Livello di Disseminazione: (PU)	
Sommario	
TDM è un progetto collaborativo tra CRS4 e Università di Cagliari che combina attività di ricerca, sviluppo, sperimentazione e formazione nel campo dell'informatica urbana. Il lavoro nel campo della visualizzazione mira a facilitare la comprensione di dati massivi e complessi attraverso metodi visuali. Per far questo, ci concentriamo su due linee di attività ben distinte. Da un lato, ci occupiamo di creare e validare metodi e applicazioni funzionanti su piattaforma web per la visualizzazione interattiva di open data, rendendoli disponibili sul portale di progetto. Dall'altro avanziamo lo stato dell'arte nella visualizzazione di grandi volumi di dati attraverso lo studio e sviluppo di nuovi metodi matematici ed informatici. In questo deliverable presentiamo la versione preliminare delle tecnologie di visualizzazione sviluppate nel progetto. Il deliverable è stato completato durante il periodo di restrizioni dovute alle misure di contrasto alla pandemia COVID-19 e, per causa di forza maggiore, non riporta quindi gli aspetti relativi all'integrazione completa con la sensoristica di progetto. Questi aspetti saranno discussi nel prossimo deliverable (D6.2).	

REGIONE AUTÒNOMA DE SARDIGNA
REGIONE AUTONOMA DELLA SARDEGNA
POR FESR SARDEGNA 2014-2020
SARDEGNA RICERCHE
UNIONE EUROPEA
Fondo europeo di sviluppo regionale
REPUBBLICA ITALIANA

Redazione			
	Nome	Partner	Data
Autore	Fabio Bettio, Alberto Jaspe, Enrico Gobbetti, Fabio Marton, Antonio Zorcolo	CRS4	05/06/2020
Autore	Gianmarco Cherchi, Riccardo Scateni	UNICA	05/06/2020
Approvato da	E. Gobbetti	CRS4	05/06/2020

Storia e contributi			
Vers.	Data	Commento	Autori
V0.1	12/03/2020	Primo draft con organizzazione contenuti	E. Gobbetti (CRS4)
V0.2	20/03/2020	Primi contributi su visualizzazione volumetrica scalabile	F. Marton, E. Gobbetti, A. Zorcolo (CRS4)
V0.3	20/04/2020	Visualizzazione open data	F. Bettio (CRS4)
V0.4	20/03/2020	Finalizzazione visualizzazione volumetrica scalabile	E. Gobbetti (CRS4)
V0.5	28/05/2020	Monitoraggio indoor e elettrico	G. Cherchi e R. Scateni (UNICA)
V0.6	29/05/2020	Revisione contributi di visualizzazione open data e descrizione librerie multilayer	A. Jaspe, E. Gobbetti (CRS4)
V1.0	01/06/2020	Versione finale	E. Gobbetti (CRS4)

Indice

1 Introduzione **6**
1.1 Obiettivi di ricerca nel campo della visualizzazione . . . 6
1.1.1 Metodi visuali per la presentazione di open data da integrare su portale di progetto . . . 6
1.1.2 Tecnologie abilitanti per la visualizzazione scalabile . . . 7
1.2 Ambito di questo deliverable . . . 8
1.3 Struttura del report . . . 8
1.4 Riferimenti bibliografici . . . 8

2 Metodi visuali per la presentazione di open data da integrare su portale di progetto **10**
2.1 Visualizzazione di eventi meteo . . . 10
2.1.1 Implementazione . . . 11
2.1.2 Interfaccia generale . . . 13
2.2 Monitoraggio Indoor . . . 14
2.2.1 Livello di benessere . . . 15
2.2.2 Implementazione . . . 15
2.3 Monitoraggio Consumi elettrici . . . 16
2.4 Visualizzazione multi-scala di dati annotati . . . 18
2.5 Discussione . . . 20

3 Tecnologie scalabili per la visualizzazione di dati di simulazione **21**
3.1 Metodi di compressione per l'esplorazione in tempo reale di volumi scalari rettilinei variabili nel tempo . . . 21
3.2 Lavori correlati . . . 25
3.3 Panoramica . . . 26
3.4 Layout dei dati e rappresentazione dei dati compressi . . . 27
3.4.1 Compressione near-lossless per frame d'alta qualità . . . 28
3.4.2 Compressione low-bitrate per la presentazione dinamica dei dati . . . 29
3.5 Rendering adattivo da dati compressi . . . 32
3.5.1 Renderer configurabile con accelerazione GPU . . . 32
3.5.2 Operazioni con accelerazione GPU . . . 34
3.6 Distribuzione dei dati e design di applicazioni di rendering . . . 38
3.6.1 Fat client e snapshot renderer . . . 38
3.6.2 Thin client e visualizzatore remoto interattivo . . . 40
3.7 Implementazione e risultati . . . 41
3.7.1 Prestazioni della compressione . . . 42
3.7.2 Prestazioni di rendering . . . 46
3.8 Simulazione urbana . . . 50

3.9 Pianificazione della distribuzione open source del codec 50
3.9.1 Dipendenze e librerie di base 51
3.9.2 Struttura della libreria 51
3.9.3 Formato file 52
3.9.4 Applicazione dimostrativa 52
3.10 Discussione e lavori futuri 53

4 Conclusioni **54**

1 Introduzione

TDM è un progetto collaborativo tra CRS4 e Università di Cagliari, che mira ad offrire nuove soluzioni intelligenti per aumentare l'attrattività cittadina, la gestione delle risorse e la sicurezza e qualità di vita dei cittadini, attraverso lo studio e sviluppo di tecnologie abilitanti e di soluzioni verticali innovative per la protezione dai rischi ambientali, l'efficienza energetica e la fruizione dei beni culturali.

Uno degli obiettivi principali del progetto TDM è di realizzare un'architettura scalabile per l'acquisizione, l'integrazione e l'analisi di dati provenienti da sorgenti eterogenee, in grado di gestire i dati generati da un'area metropolitana estesa. L'implementazione prevista nel quadro del progetto riguarda casi di studio nell'area metropolitana della città di Cagliari, ma l'obiettivo è quello di generare soluzioni scalabili generali, che possano servire da best practice per future implementazioni di servizi in aree geografiche ampie e/o densamente popolate. Il progetto deve quindi combinare, sviluppare ed estendere soluzioni tecnologiche in vari campi, dalla sensoristica ai big data e dalla simulazione alla visualizzazione. Questo deliverable è relativo agli aspetti di visualizzazione.

1.1 Obiettivi di ricerca nel campo della visualizzazione

Il lavoro di ricerca e sviluppo nel campo della visualizzazione mira soprattutto a facilitare la comprensione di dati complessi attraverso metodi visuali. Per far questo, il lavoro si concentra su due linee di attività ben distinte:

- la creazione e validazione di metodi funzionanti su piattaforma web per la visualizzazione interattiva di open data, sia statici sia dinamici. Gli strumenti sviluppati sono stati validati in termini di usabilità e resi disponibili per l'integrazione nel portale di progetto realizzato in OR2.
- l'obiettivo tecnico-scientifico di avanzare lo stato dell'arte nella compressione di dati per la visualizzazione scalabile, finalizzata alla ispezione interattiva di grandi modelli, con particolare riferimento ai modelli volumetrici generati da simulazioni, sia in soluzioni locali che distribuite.

1.1.1 Metodi visuali per la presentazione di open data da integrare su portale di progetto

Uno degli aspetti maggiormente interessanti in questo progetto è la produzione e l'aggregazione in scala massiva di dati provenienti dall'area metropolitana di Cagliari. Questi dati saranno messi a disposizione in forma aperta, seguendo il principio dell'open-data. Rimane il rilevante problema che la disponibilità di un dato e la sua possibile fruizione sono due azioni non necessariamente equivalenti. La disponibilità, infatti, è solo una precondizione necessaria, ma non sufficiente, per avere dati fruibili al grande pubblico, in forma aggregata e sintetica.

Una delle modalità più dirette e coinvolgenti per la rappresentazione dei dati è, abbastanza naturalmente, la loro rappresentazione visuale. Sotto il nome di Information Visualization vanno tutte le attività che prevedono la progettazione di rappresentazioni visuali dei dati e di tecniche di interazione, che consentono all'essere umano di migliorare la comprensione del significato di dati che non hanno in sé una natura spaziale. Per fare un esempio, è molto più semplice rappresentare le informazioni relative alla piovosità all'interno di una regione geografica che rappresentare, in maniera grafica, come si possano correlare tra loro i dati sul consumo elettrico dei nuclei familiari con i loro dati anagrafici. Nel primo caso la rappresentazione spaziale è implicita nei dati (ogni punto del territorio ha un dato di piovosità, misurato o interpolato), mentre nel secondo caso devono essere studiate delle modalità di rappresentazione dei dati che ne consentano la migliore interpretazione. L'interazione dell'utente, in entrambi i casi, è fondamentale per dare la flessibilità adeguata di rappresentazione.

Il contributo in questo ambito prevede quindi di elaborare innovative modalità di rappresentazione dei dati raccolti all'interno degli altri OR del progetto, utilizzando direttamente la piattaforma e la sensoristica sviluppata. I risultati sono integrati nel portale di progetto.

1.1.2 Tecnologie abilitanti per la visualizzazione scalabile

In numerosi settori applicativi collegati al soggetto generale dell'urban computing, tra cui beni culturali, architettura, ingegneria, sicurezza e ambiente, sono richieste analisi di scene e modelli tridimensionali complessi, generati sia da simulazioni numeriche, sia da rilievi dettagliati di scene o oggetti d'interesse. Tipici esempi sono lo studio della genesi di eventi meteorologici estremi, che creano di routine svariati terabyte di dati volumetrici per singola simulazione, la generazione rapida e l'esplorazione di mappe 3D di siti ed edifici, ad esempio per applicazioni di sicurezza e gestione di emergenza, e lo studio, promozione e valorizzazione di beni culturali attraverso repliche virtuali.

Fino a poco tempo fa, l'uso più efficace e diffuso di tecnologie multimediali per esplorare dati complessi, sia in applicazioni professionali sia consumer, è stato quello di utilizzare modalità di presentazione visiva per lo più passive, come video o animazioni generate dal computer, in quanto questo approccio permette di gestire la complessità del dato al momento della generazione offline del materiale visivo, operazione che può essere svolta senza gli stretti vincoli di tempo e qualità dettati dalla percezione umana, utilizzando tutto il tempo necessario a produrre immagini da modelli complessi, senza doverli semplificare oltre misura.

Questi metodi passivi hanno, però, moltissime limitazioni, e l'interesse si sta ora spostando verso modalità più flessibili, come i sistemi di navigazione virtuale, che consentono agli utenti di guidare direttamente e dinamicamente l'esplorazione verso i dettagli di loro interesse. Da un lato, è, infatti, ormai noto che l'analisi visuale interattiva d'informazioni spaziali e dati immersi in tre dimensioni ha un ruolo principe nel comprendere la struttura e le implicazioni di dati complessi, in un mondo in cui la scienza, la tecnologia e l'ingegneria sono sempre maggiormente caratterizzate dal bisogno di estrarre informazioni da grandi quantità di dati (scientific and information visualization). L'utilizzo di soluzioni interattive per l'esplorazione di dati complessi e massivi, specialmente se associate agli emergenti display a grande scala e alta risoluzione (LHD: 4K e oltre) o a visori mobili, richiedono la generazione di centinaia di milioni di pixels al secondo, con ritardi di pochi centesimi di secondo, a partire da scene che sono spesso di molteplici gigabyte e richiedono, per la loro visualizzazione

efficace, la simulazione in tempo reale di effetti di shading. Questo richiede la risoluzione di problemi aperti di ricerca, al fine di creare tecnologie scalabili appropriate.

Il contributo in questo ambito è lo sviluppo di nuovi metodi basati sulla compressione di dati che possano essere implementati all'interno di sistemi interattivi capaci di gestire grandi quantità di dati. Queste tecnologie abilitanti sono applicate a dati volumetrici tempo-varianti, in particolare griglie di valori scalari, la cui visualizzazione interattiva è una delle maggiori sfide nel campo della visualizzazione scientifica.

1.2 Ambito di questo deliverable

In questo deliverable presentiamo la descrizione dettagliata dei risultati ottenuti nel progetto per la visualizzazione dei dati, nella loro versione preliminare.

Il deliverable è stato completato durante il periodo in cui erano attive le misure per il contenimento della pandemia COVID-19, che hanno impedito tutti i lavori sul campo e le collaborazioni pianificate con scuole, cittadini ed enti pubblici. I risultati presentati, pertanto, sono relativi alla fase di sviluppo del sistema e non all'integrazione con i dati della sensoristica e con il portale di progetto, in particolare per quanto riguarda le applicazioni descritte nel capitolo 2.

1.3 Struttura del report

Il report è organizzato come segue:

- al Capitolo 2 presentiamo le tecnologie e le applicazioni sviluppate per la visualizzazione di dati aperti di progetto, con particolare riferimento a casi d'uso in campo ambientale ed elettrico.
- al Capitolo 3 descriviamo i risultati ottenuti nello sviluppo di tecnologie abilitanti per la visualizzzione volumetrica tempo-variante scalabile.

Il rapporto si conclude con un riassunto dei principali obiettivi raggiunti (Capitolo 4).

1.4 Riferimenti bibliografici

I risultati dell'attività di ricerca nel campo della visualizzazione hanno portato a diverse pubblicazioni scientifiche, che forniscono maggiori dettagli sui metodi sviluppati e risultati ottenuti.

In particolare, l'architettura generale del progetto TDM è stata presentata al convegno GARR 2019 [1]. I metodi di acquisizione 3D di ambienti indoor, che saranno utilizzati anche per la generazione dei piani necessari alle applicazioni di monitoraggio descritte al Capitolo 2, sono stati presentati a Pacific Graphics 2019 e pubblicati su rivista [2] e sono stati oggetto di un survey dello stato dell'arte, pubblicato su rivista [3] e presentato a EUROGRAPHIS 2020.

L'applicazione data-driven per l'analisi dei consumi elettrici è stata oggetto di un poster e demo presentati a STAG2019 [4]. L'applicazione al campo dei beni culturali delle tecniche di visualizzazione multilayer descritte al Capitolo 2, è stata descritta in un lavoro presentato a GCH 2019 [5], che ha vinto il best paper award.

Il metodo di visualizzazione volumetrica scalabile presentato al Capitolo 3, è descritto in un lavoro presentato ad Eurovis 2019 e pubblicato su rivista [6] ed è stato oggetto di un follow-up presentato a STAG 2019 [7]. Il primo lavoro è stato shortlisted nei 5 higlights della conferenza, mentre il secondo ha vinto il best paper award. Le tecniche scalabili per la visualizzazione sono state inoltre oggetto di tutorials presentati a SIGGRAPH Asia 2017 [8] e EUROGRAPHIS 2018 [9] e di una keynote lecture a Eurographics 2019 [10].

2 Metodi visuali per la presentazione di open data da integrare su portale di progetto

L'esplorazione in forma visuale dei dati facilita l'utente nella comprensione del loro significato. Se le decisioni devono essere prese partendo da grandi volumi di dati spazio-temporali, questo genere di rappresentazione è particolarmente importante sia se l'esplorazione deve essere fatta dal comune cittadino, sia se decisioni devono essere prese da esperti.

In questo capitolo, presentiamo le tecnologie ed applicazioni sviluppate dal progetto in questo ambito, con particolare riferimento all'utilizzo di dati aperti. Vedremo, in particolare, le tecnologie sviluppate per supportare le attività nei settori applicativi del progetto (applicazioni meteo-ambientali e applicazioni energetiche) e tecnologie di uso generale, che hanno trovato applicazioni anche in altri ambiti (ad es., nei beni culturali).

2.1 Visualizzazione di eventi meteo

Il visualizzatore integrato di eventi meteorologici è un esempio operativo di utilizzo dell'intera pipeline TDM, che, attraverso le varie fasi a partire dalla produzione del dato, alla sua elaborazione e all'inserimento nella piattaforma open-source, consente la visualizzazione e il confronto di dati georeferenziati meteo-ambientali derivanti da simulazioni di previsione meteorologica o da misure da sensore (radar) relative a precipitazioni atmosferiche nella zona di Cagliari. La pipeline, per quanto riguarda gli aspetti di pubblicazione di open data, è descritta nel deliverable D3.2, che descrive come i dati originari di simulazioni e misure sono automaticamente elaborati dal sistema TDM e resi disponibili sulla rete, per i fini della visualizzazione, in formati grafici georeferenziati di tipo standard.

Data l'attuale disponibilità di efficienti risorse grafiche nei moderni dispositivi mobili e personal computer, la scelta del framework di visualizzazione per la pipeline TDM si è naturalmente orientata verso il rendering client-side implementato in Javascript. Questa scelta, in linea con le tendenze attuali, consente di sfruttare moltissime librerie grafiche già realizzate ad-hoc per questi scopi e sviluppare soluzioni specifiche per i dati del sistema TDM.

La pipeline integra i dati da simulatore o da sensoristica passando attraverso l'interfaccia fornita da TDM CKAN descritta nelle sezione precedenti. Tutte le variabili misurate o simulate sono georeferenziate e tempo-varianti.

Le componenti del sistema di visualizzazione sono state progettate per le due tipologie di dati: campi scalari e campi vettoriali 2D. L'approccio scelto, basato su piattaforme altamente customizzabili, consente comunque di estendere in futuro le visualizzazioni ad altri tipi di dati. Il progetto delle componenti di visualizzazione tiene quindi conto del tipo di grandezze (scalari o vettori), della loro localizzazione geografica e della loro variabilità nel tempo.

I dati georeferenziati sono rappresentati su mappe geografiche disponibili in formato aperto e multiscala sulla rete (tiled web map). Il sistema di visualizzazione è quindi a strati (layer): ogni layer

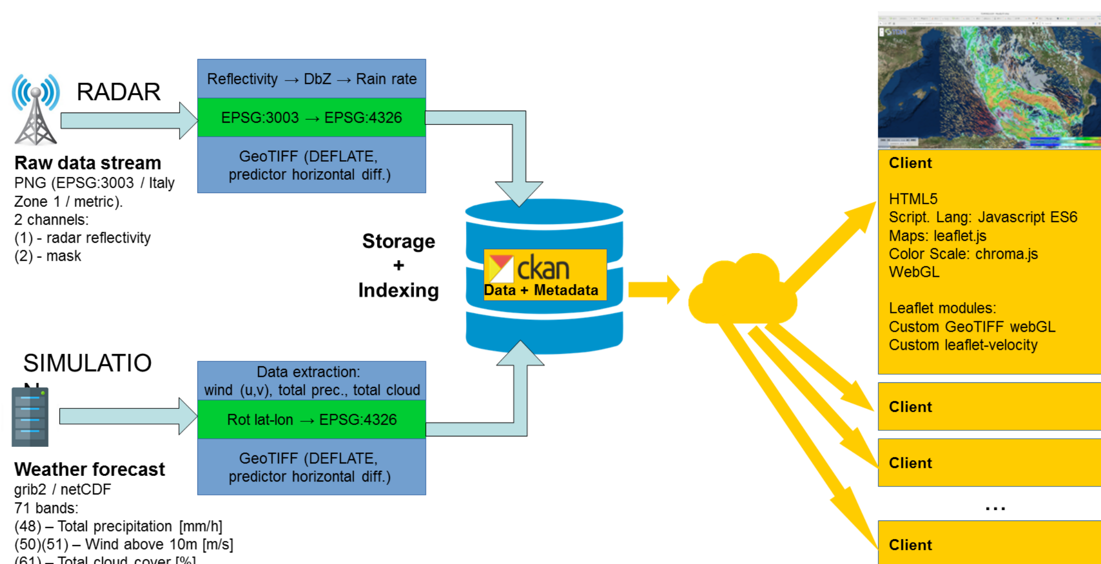

Figura 2.1: *Struttura del sottosistema di visulizzazione eventi meteo*

rappresenta un tipo di dato da visualizzare, e tutti sono sovrapposti (con un certo ordine di disegno e livello di trasparenza) alla mappa geografica di base (es. OpenStreetMap). Il risultato è quindi una cartina geografica su cui sono rappresentate, con le tecniche opportune di sciviz, le varie grandezze da visualizzare.

I campi scalari sono rappresentati tramite colormap, ad ogni dato scalare posizionato sulle sue coordinate geografiche, compreso tra un valore massimo e minimo, è associato quindi un colore preso da una scala di colori continua. Tale mapping genera quindi delle strutture visuali utilizzate per migliorare la percezione dei dati e la loro interpretazione efficace. Sono uno dei modi più comuni per vedere i dati, e tra i più studiati in letteratura.

I campi vettoriali sono rappresentati tramite animated particle tracing: delle particelle, posizionate in modo casuale ed uniforme sul campo vettoriale, sono "trasportate" dal campo stesso lungo quindi le linee di flusso del campo. Ogni particella traccia una scia evanescente di lunghezza proporzionale al modulo del campo vettoriale. Anche il colore di ogni particella rappresenta il modulo del campo vettoriale (l'associazione tra colore e valore è definita da una colormap). Il risultato grafico è molto efficace e consente di valutare velocemente la presenza di perturbazioni del campo del campo vettoriale (come turbolenze o vortici) o valori di attenzione delle grandezze visualizzate.

2.1.1 Implementazione

L'implementazione del visualizzatore client-side, basata su Javascript, si appoggia su alcune librerie open-source standard:

- **Leaflet**: Libreria Javascript per la gestione di mappe cartografiche interattive mobile-friendly. URL: https://leafletjs.com/Release: 1.3.1
- **geotiff.js**: Libreria Javascript per la lettura di numerosi tipi di file (Geo)TIFF e gestione di metadati geospaziali e raw data array. URL: https://geotiffjs.github.io/geotiff.js/ Release: 1.0.0-beta.3

- **twgl.js**: Libreria Javascript di utility per WebGL orientata alla programmazione a basso livello. URL: https://twgljs.org/. Release: 1.9.0
- **chroma.js**: Libreria Javascript per la gestione efficiente di colormap e conversioni tra spazi colore. URL: https://github.com/gka/chroma.js/. Release: 2.0.2
- **L.CanvasLayer.js**. Libreria Javascript di utility per la gestione di layer generici Leaflet in overlay (rel. $>$ 1.0.0). https://github.com/Sumbera/gLayers.Leaflet. Release: 1.0.4
- **leaflet-velocity.js**: Leaflet plugin per l'implementazione di un canvas per il disegno di campi vettoriali. https://github.com/danwild/leaflet-velocity. Release: 1.2.4

Le librerie Javascript sopra elencate sono state impiegate per lo sviluppo di moduli specifici di visualizzazione. In particolare sono stati realizzati:

- **tdm-leaflet-velocity**: Una versione modificata ed estesa della libreria leaflet-velocity.js in grado in particolare di trattare direttamente dati di campi vettoriali in formato GeoTIFF
- **L.TDMCanvasLayer.js**: Una variante di L.CanvasLayer.js integrata nella struttura del visualizzatore per la costruzione di layer leaflet GeoTIFF
- **tdm-basic-geotiff-layer.js**: Un'estensione di L.TDMCanvasLayer.js per la visualizzazione di campi scalari encodati in immagini GeoTIFF. Il layer supporta la definizione di valori soglia, coordinate geografiche, colormap, e vari livelli di opacità. La visualizzazione avviene tramite WebGL, utilizzando texturing e shaders.
- **Mutex.js**: Versione di un mutex per garantire la mutua esclusione di chiamate parallele. E' utilizzato nel codice per sfruttare al meglio il caricamento asincrono dei dati.

Sulla base di queste librerie è stato realizzato il visualizzatore dimostrativo di TDM. I dati meteo-ambientali attualmente gestiti da questo visualizzatore sono:

- **Dati da sensore (radar)**. Intensità di precipitazione [mm/h] (scalare)
- **Dati da simulazione**. Intensità di precipitazione totale [mm/h] (scalare); Copertura nuvolosa totale [%] (scalare); Temperatura a 2m [C]: Velocità del vento a 10m [m/s] (vettore 2D)

Alla figura 2.2 sono raffigurati dall'alto al basso, da sinistra a destra: il layer di base, il layer intensità di precipitazione derivata da dati radar, la copertura nuvolosa totale da previsione, l'intensità di precipitazione totale da previsione, la velocità del vento da previsione e la sovrapposizione di tutti questi layer in semitrasparenza.

Il visualizzatore, pienamente integrato nell'infrastruttura open data di TDM, è inizializzato fornendo il descrittore di evento oppure direttamente le due chiamate REST per l'accesso ai descrittori dei dati derivanti da simulazioni di previsioni meteorologiche e/o radar.

All'inizializzazione il visualizzatore analizza le informazioni ricevute e crea tutto quanto è necessario per la visualizzazione. In particolare:

- La descrizione dei singoli timestep con associati metadati (data, ora, tipologia di dato e URL per l'accesso diretto ai dati)
- La mappa cartografica di base (per default elaborazione ESRI da dati NCEP NOAA)
- L'interfaccia grafica per il controllo di Pan/Zoom, l'attivazione e disattivazione dei singoli layer, l'accesso temporale ai singoli timestep, la legenda relativa a tutti i layer attivi.

Per il timestep corrente, tutti i layer sono caricati in asincrono e visualizzati appena disponibili, in un definito ordine di priorità, in overlay sulla mappa di base.

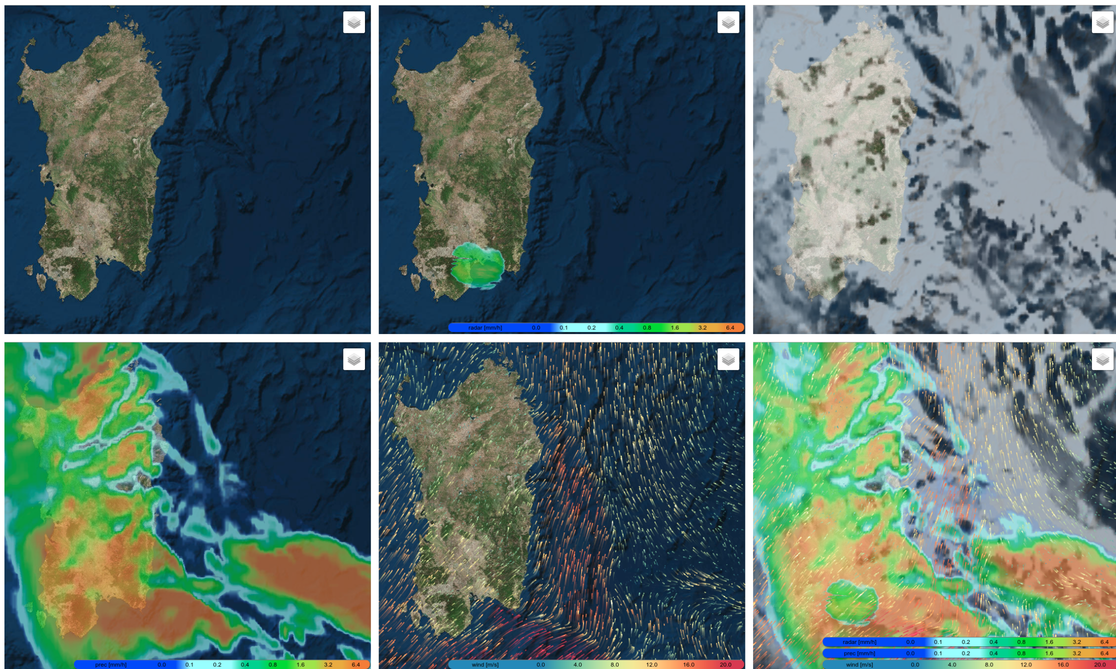

Figura 2.2: *Visualizzazione di alcuni layer meteo disponibili nel visualizzatore TDM*

2.1.2 Interfaccia generale

Il visualizzatore di dati meteo-ambientali di TDM è un'applicazione web client-side che si presenta con l'interfaccia grafica raffigurata alla figura 2.1

La finestra del visualizzatore presenta la mappa cartografica di base su cui sono disegnati i layer in overlay relativi a dati scalari (rappresentazione colormap) e vettoriali (particle tracing).

Il visualizzatore, progettato come responsive website, consente all'utente di interagire utilizzando il mouse o il touch screen di un mobile device. La mappa cartografica e i layer di dati possono essere traslati alla posizione desiderata tenendo premuto il tasto sinistro del mouse o muovendo un dito sul touch screen. Mappa e dati possono essere zoomati tramite la rotellina del mouse o il movimento di indice e pollice (pinch-to-zoom) tipico dei dispositivi mobili multitouch.

In alto a destra è posizionato il *selettore dei layer* diviso in due parti: i layer di base e i layer dei dati. E' possibile configurare il visualizzatore con più layer di base (mutualmente esclusivi) e con numerosi layer overlay (ognuno singolarmente attivabile dal selettore). Nell'esempio in figura è presente un solo layer di base (Satellite) e 4 layer dati (*Total Cloud, Total Prec, 10m Wind, Radar*) di cui 1 (*Total Prec*) non attivato.

La *legenda delle colormap* è posizionata in basso a destra. Per ogni layer dati attivo vengono indicati la grandezza rappresentata, l'unità di misura e la scala di colori graduata impiegata per il mapping dei dati. Sotto la legenda delle colormap sono indicate alcune informazioni sulle fonti dei dati.

In basso a sinistra sono presenti altri due elementi grafici: il *selettore del timestep* e informazioni puntuali su direzione e velocità del vento nella posizione del cursore (mouse).

Il *selettore del timestep* consente di selezionare data e ora dei dati da visualizzare (scelti tra quelli di un evento definito dal descrittore della chiamata REST di inizializzazione del sistema).

Nel campo centrale del selettore del timestep è indicata la data e ora corrente a cui si riferiscono i dati visualizzati. Sono presenti quattro pulsanti (da sinistra a destra) per selezionare il primo step temporale della sequenza (evento), selezionare lo step precedente (rispetto all'attuale), selezionare lo step seguente (rispetto all'attuale) e selezionare l'ultimo step temporale. Per facilitare la selezione di un determinato step temporale è possibile scegliere direttamente un elemento da una drop-down list cliccabile nella parte in basso del selettore del timestep.

In alto a sinistra è presente un selettore di scala.

Le componenti del visualizzatore sono state sviluppate in modo da integrarsi completamente e semplicemente all'interno del framework Leaflet. Nell repository demo alla URL `https://github.com/tdm-project/tdm-tools` sono illustrati tutti gli esempi che permettono all'utente di interfacciarsi col sistema per la visualizzazione degli eventi meteo.

2.2 Monitoraggio Indoor

L'obiettivo di questa linea di attività è quello di mostrare, in maniera semplice e intuitiva, la correlazione tra le condizioni meteorologiche, il consumo interno di elettricità e il livello di benessere interno di un edificio.

Questo genere di applicazione è di primaria importanza sia per la gestione di edifici privati, per rendere consapevole il cittadino della correlazione tra consumi, impatto ambientale e benessere, sia soprattutto per la gestione di grandi edifici pubblici (ad esempio scuole o uffici).

Nella nostra prima implementazione, ci siamo focalizzati sullo sviluppo delle componenti di base, dimostrandole per il caso di una singola abitazione, mentre nel resto del progetto ci focalizzeremo sull'utilizzo di queste tecnologie per il monitoraggio di edifici pubblici.

La schermata principale è composta da diverse sezioni (vedi figura 2.4). La prima mostra una piantina dell'ambiente monitorato con le stanze disponibili evidenziate con colori diversi. Passando il mouse sopra una stanza nella piantina, è possibile visualizzare un pop-up che mostra l'identificatore della stanza e il livello di benessere registrato.

Sul lato destro della pagina vengono mostrate diverse informazioni. L'utente può visualizzare l'elenco degli ambienti monitorati disponibili e selezionarli per conoscerne i dettagli. È inoltre possibile selezionare l'intero edificio per avere un report con i dati aggregati.

Una delle parti fondamentali presenti nella demo riguarda il consumo interno dell'edificio. Un grafico mostra il consumo elettrico giornaliero espresso in wattora. Nella stessa sezione è visibile un grafico a barre che mostra l'utilizzo di elettricità di ogni stanza disponibile.

Una porzione fondamentale della schermata è la sezione posizionata al di sotto della piantina. In questa parte è possibile visualizzare il livello di benessere con un indicatore apposito. È inoltre possibile visualizzare il confronto tra i valori di temperatura e umidità registrati all'interno all'ambiente selezionato e gli stessi valori registrati all'esterno all'edificio. L'obiettivo di questa parte dell'interfaccia è dare all'utente una stima del consumo elettrico necessario per raggiungere un preciso valore di benessere dell'ambiente. Tutte le misure riportate nella schermata sono disponibili a partire dal primo giorno di monitoraggio fino alla data corrente.

2.2.1 Livello di benessere

Il livello di benessere utilizzato nella demo è un valore compreso tra 0 e 1 che fornisce un feedback sul benessere di ogni stanza monitorata. È calcolato utilizzando i valori di temperatura e umidità registrati, combinandoli insieme e confrontandoli con le tabelle messe a disposizione dal Ministero della Salute Italiano (vedi tabella 2.1). Questa tabelle riportano una serie di parametri da rispettare, soprattutto in edifici pubblici, di temperatura e umidità, nei diversi periodi dell'anno.

Stagione	Temperatura	Umidità
Invernale	$19 - 22°C$	$40 - 50\%$
Estiva	$24 - 26°C$	$50 - 60\%$

Tabella 2.1: *Valori di riferimento di temperatura e umidità, per ambienti interni, forniti dal Ministero della Salute.*

Considerando il valore medio degli intervalli riportati nella tabella in questione, si calcola la differenza tra il valore registrato dai sensori e i valori di riferimento. Il valore ottenuto viene normalizzato tra 0 e 1 e suddiviso in tre intervalli: da 0 a 0,6 siamo in presenza di un livello di benessere basso, moderato da 0,6 a 0,95 e alto (ideale) da 0,95 a 1. Tale valore è poi visualizzato in maniera intuitiva tramite l'utilizzo di icone (vedi figura 2.3).

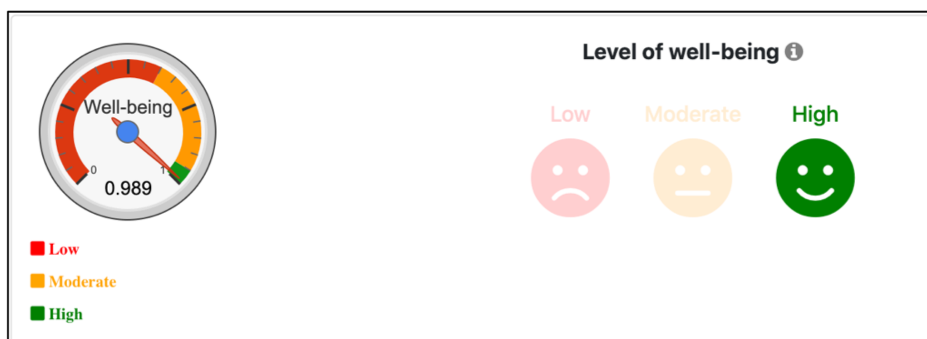

Figura 2.3: ***Porzione dell'interfaccia che mostra, tramite combinazione di widget e icone, il livello di benessere di una stanza/edificio.***

2.2.2 Implementazione

La parte server dell'applicazione è scritta in Java, compresa la communicazione con il database che al momento risiede in locale con dati simulati. Il DOM è manipolato dinamicamente tramite jQuery. La colorazione della piantina dell'ambiente monitorato è implementata tramite la libreria Javascrip p5.js, mentre il layout della pagina è implementato utilizzando elementi Bootstrap.

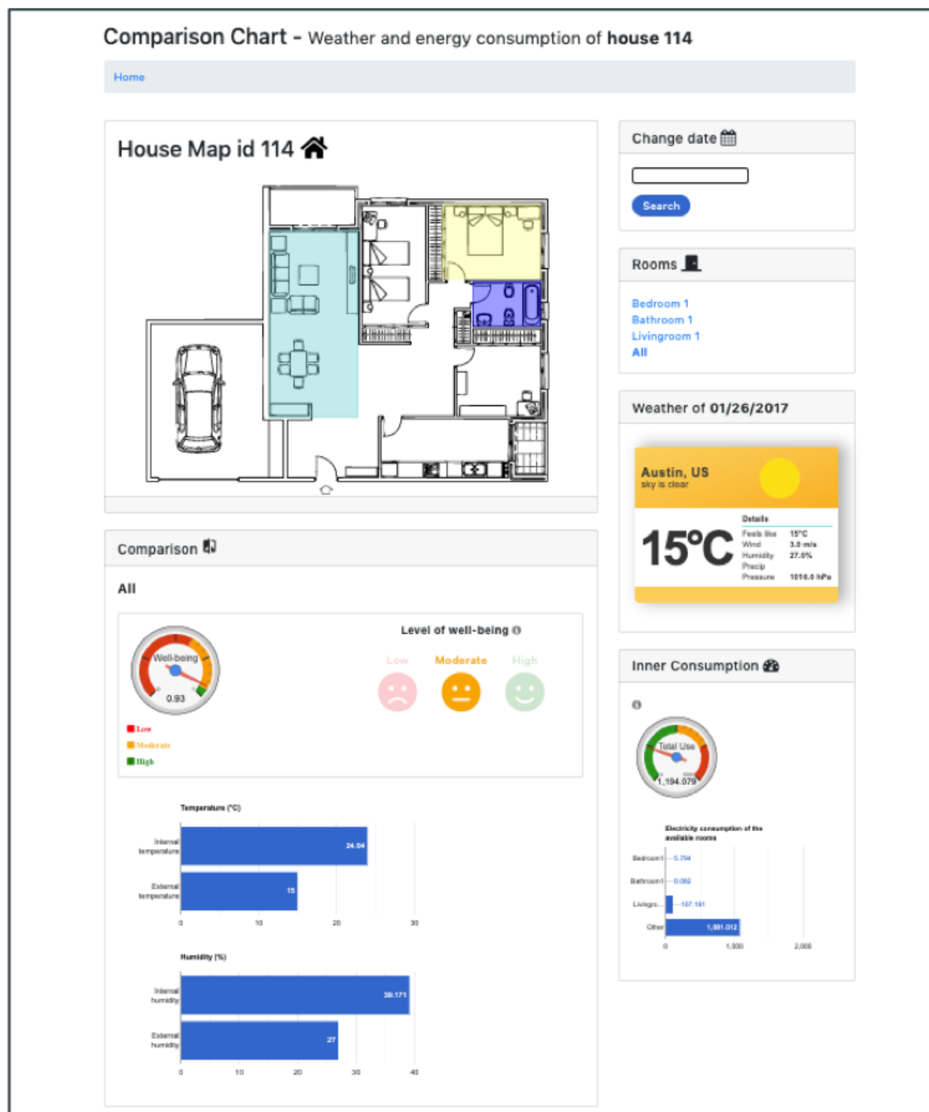

Figura 2.4: *Schermata principale della demo in cui sono visibili la piantina dell'edificio monitorato e una serie di indicatori che mostrano l'analisi dei dati raccolti.*

2.3 Monitoraggio Consumi elettrici

L'obiettivo di questa linea di attività è quello di permettere il monitoraggio dei consumi elettrici a partire da misure provenienti dai sensori di progetto.

L'obiettivo di questa demo consiste nella modifica/sviluppo di diversi plug-in per la visualizzazione dei consumi elettrici in maniera semplice e intuitiva per l'utente non esperto (vedi figura 2.5). Come tool per la realizzazione dei plug-in si utilizza Grafana, leader nella visualizzazione moderna di dati di questo tipo.

Il primo plug-in sviluppato è una versione modificate del *SingleStat* di Grafana che ci consente di confrontare il consumo personale attuale con il consumo personale medio. Questo plug-in ci consente inoltre di confrontare la propria media dei consumi con la media degli altri utenti presenti nel sistema (opportunamente aggregata e anonimizzata). Nella versione originale del plug-in è possibile confrontare un singolo valore recuperato tramite query nel

database con una serie di valori (intervalli) impostati staticamente. Nella modifica proposta tutti i valori utilizzati per i vari confronti sono recuperati in maniera dinamica tramite query sul database.

Un secondo plug-in sviluppato è una versione modificata del *MultiStat* di Grafana. Attraverso questo plug-in possiamo visualizzare il consumo effettivo di ciascun dispositivo elettrico presenta nell'ambiente monitorato, e confrontarlo con il suo consumo medio registrato nel tempo. Possiamo inoltre mostrare il consumo medio degli elettrodomestici monitorati rispetto al consumo medio della stessa categoria di elettrodomestici in abitazioni di altri utenti (i cui dati sono stati, ovviamente, opportunamente anonimizzati). Nella versione originale del plug-in, anche in questo caso, era possibile effettuare una singola query sul database per recuperare i valori da utilizzare per la creazione dell'intero grafico a barre. Nella modifica proposta, ogni barra del grafico (rappresentante un elettrodomestico differente) è recuperata con una query appositamente configurata. Anche i valori di riferimento (le medie dei consumi per i confronti) possono ora essere recuperati (o calcolati) dinamicamente mediante query sul database.

Un ultimo plug-in sviluppato è quello che abbiamo chiamato WorldMap. La versione originale di questo plug-in consiste in uno strumento di output, in cui è possibile visualizzare i dati recuperati dal database e localizzarli su una mappa. Con le modifiche apportate, la mappa presente nel plug-in consente di selezionare un punto della città e aggiornare tutti i dati presenti nella dashboard in base alla zona selezionata. Di fatto il plug-in viene ora trasformato in uno strumento di input per permettere l'interazione con l'utente. Nello specifico, grazie all'utilizzo di Leaflet (libreria precedentemente descritta), la mappa viene divisa in settori "cliccabili" che corrispondono alle zone della città monitorate. Il valore relativo all'area selezionata viene poi utilizzato come input per una query nel database.

Considerando che le modifiche apportate ai vari plug-in possono essere ampiamente utili anche in altri contesti, tutti gli strumenti modificati ad hoc sono stati resi pubblici per l'utilizzo da parte della comunità scientifica:

- SingleStat: `https://github.com/cg3hci/GrafanaSinglestatTDM`
- MultiStat: `https://github.com/cg3hci/GrafanaMultistatTDM`
- WorldMap: `https://github.com/cg3hci/GrafanaWorldmapTDM`

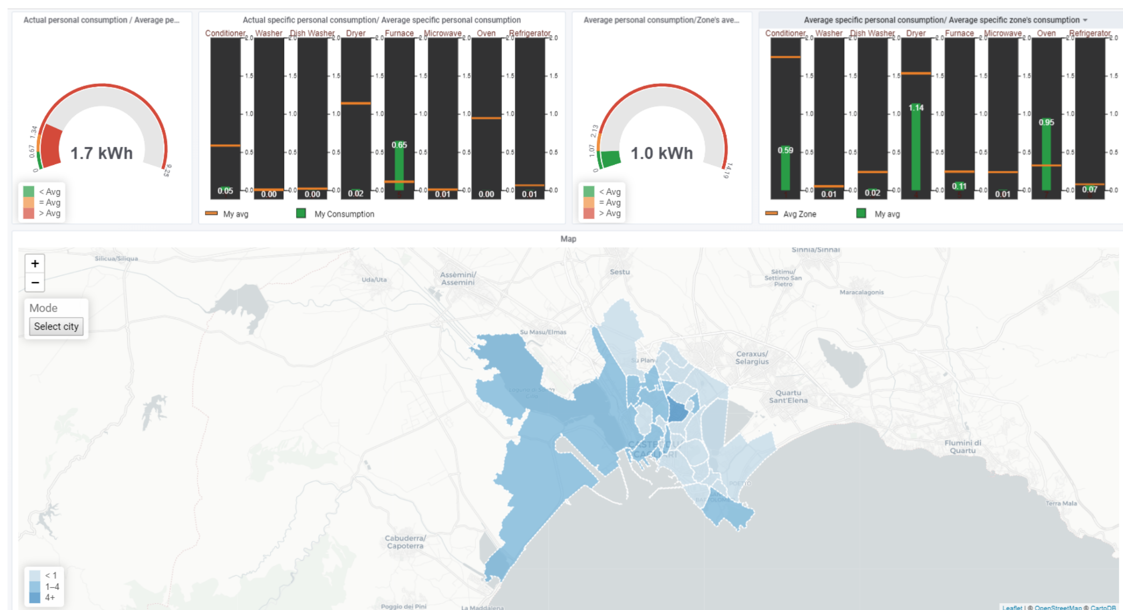

Figura 2.5: ***Schermata principale della demo implementata con Grafana in cui sono visibili i vari plugin opportunamente modificati per la visualizzazione dei consumi, secondo quanto descritto.***

2.4 Visualizzazione multi-scala di dati annotati

I contenuti multimediali digitali e i mezzi di presentazione stanno rapidamente aumentando la loro sofisticazione e sono ora in grado di descrivere in dettaglio le rappresentazioni del mondo fisico. Le esperienze di esplorazione 3D permettono alle persone di apprezzare, comprendere e interagire con oggetti intrinsecamente virtuali.

La comunicazione di informazioni sugli oggetti 3D richiede la capacità di mescolare presentazioni altamente fotorealistiche o illustrative dell'oggetto stesso con dati aggiuntivi che forniscono ulteriori approfondimenti su questi oggetti, tipicamente rappresentati sotto forma di *annotazione*. Una delle maggiori sfide per la visualizzazione è come presentare questi modelli in maniera efficace ed intuitiva.

Nel quadro del progetto TDM, abbiamo sviluppato un sistema web-based che permette l'esplorazione in tempo reale di modelli multi-layer con annotazioni. I dettagli tecnici sono forniti nel lavoro che abbiamo presentato a GCH 2019 [5], vincendo il best paper award.

Il concetto generale del sistema è illustrato alla figura 2.6.

Le fasi di preparazione dei dati off-line assemblano diverse rappresentazioni di dati in un set di dati multistrato basato su immagini. Ogni stato descrive la forma (ad esempio una mappa di elevazione o di normali) e l'apparenza dell'oggetto (mappe di colore o di materiali). Le annotazioni sono sovraimposte come immagini semitrasparenti. Dei metadati aggiuntivi sono utilizzati per assemblare diverse immagini in un unico dataset coerente.

Il dataset viene reso disponibile in rete utilizzando un server web standard. Un client JavaScript/WebGL2 carica i dati e supporta in modo interattivo la selezione degli strati da visualizzare, la reilluminazione interattiva, la visualizzazione delle annotazioni e l'esplorazione di più livelli sovrapposti usando la metafora delle lenti di visualizzazione.

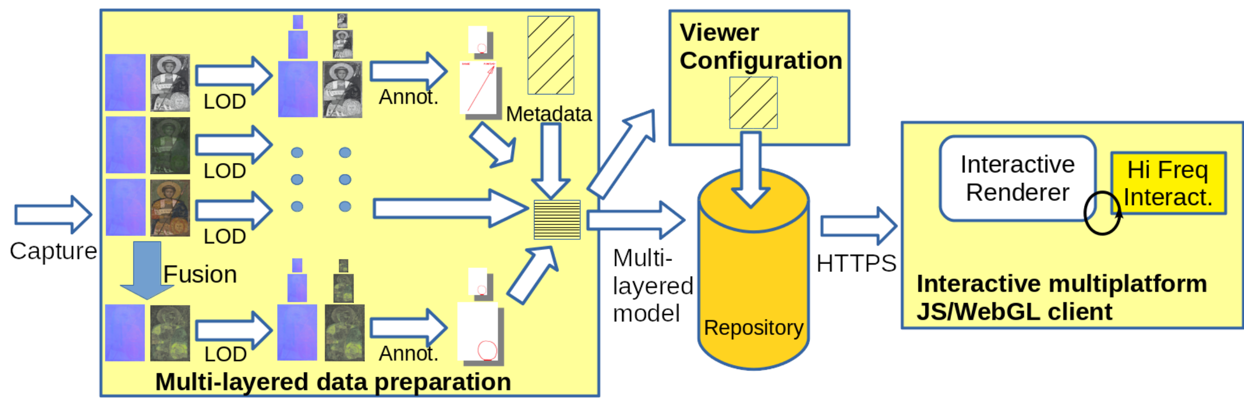

Figura 2.6: ***Architettura per la visualizzazione multilayer di dati annotati.*** .

Il sistema è stato reso disponibile in open sourca all'indirizzo: https://vicserver.crs4.it/marlie/about.html.

2.5 Discussione

Il lavoro discusso in questo capitolo risponde ad uno degli obiettivi di progetto: la creazione e validazione di metodi funzionanti su piattaforma web per la visualizzazione interattiva di dati, sia statici sia dinamici. Abbiamo visto come siano stati sviluppate sia tecnologie di uso generale che applicazioni verticali. Nel primo caso, ci si è interessati alla visualizzazione di campi scalari e vettoriali applicate a dati meteo e alla visualizzazione multi-scala di dati annotati. Nel secondo caso, ci si è interessati in particolare allo sviluppo di sistemi interattivi per il monitoraggio indoor e di consumi elettrici.

La versione discussa in questo report è quella preliminare, precedente alla diffusione sul territorio della sensoristica e all'integrazione delle applicazioni nel portale di progetto. Nei prossimi deliverable di OR6 saranno discusse ulteriori sviluppi delle tecnologie e delle applicazioni ed il loro utilizzo pratico nel caso di studio della città di Cagliari.

3 Tecnologie scalabili per la visualizzazione di dati di simulazione

Il precedente capitolo si è concentrato sulla presentazione di tecnologie e applicazioni sviluppate per la visualizzazione di dati aperti di progetto e sul loro deployment su piattaforma web per un utilizzo da parte del cittadino. In questo capitolo, invece, presentiamo i risultati ottenuti rispetto all'obiettivo tecnico-scientifico di avanzare lo stato dell'arte nella visualizzazione scalabile di grandi modelli, con particolare riferimento ai modelli volumetrici generati da simulazioni. Il risultato principale è un codec innovativo, che sarà rilasciato in open source alla fine del progetto, per la codifica compatta di grandi stream di volumi in un formato facile da decodificare in real-time su GPU.

3.1 Metodi di compressione per l'esplorazione in tempo reale di volumi scalari rettilinei variabili nel tempo

Molte applicazioni di tipo scientifico, medico e ingegneristico generano volumi scalari rettilinei, variabili nel tempo con migliaia di timesteps, ciascuno di miliardi di voxel [11, 12, 13]. Uno dei settori di interesse per il progetto TDM è la simulazione fluidodinamica a scala urbana [14], in cui le simulazioni computazionali possono essere utilizzate per studiare il microclima urbano a diverse scale spaziali, che vanno dalla mesoscala meteorologica alla microscala meteorologica fino alla scala dell'edificio e quella dell'ambiente interno [14] (vedi Fig. 3.1).

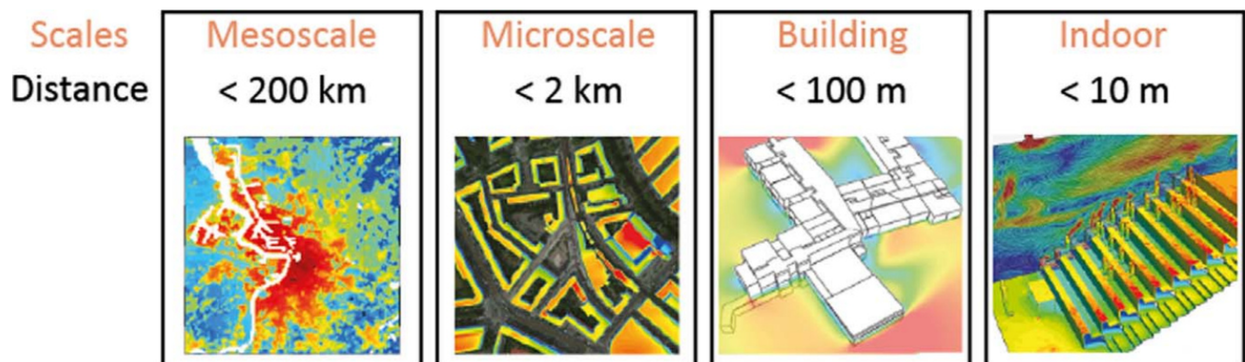

Figura 3.1: ***Rappresentazione schematica delle scale spaziali nella modellazione del clima, con dimensioni orizzontali tipiche.*** *Immagine ripresa dal recente survey di Toparlar et al. [14]*

Mentre sono già stati compiuti numerosi sforzi per generare rappresentazioni astratte ridotte di intere animazioni [15, 16, 17], per comprendere molti fenomeni in campo scientifico e tecnologico l'esplorazione visiva interattiva di interi dataset è parimenti cruciale [18, 19].

La richiesta di interattività nell'indagare i dataset nel tempo, spazio e parametri di rendering, impone alle attuali applicazioni di visualizzazione, eseguite tipicamente su PC desktop o altri dispositivi

personali, di rispettare rigorosi vincoli di memoria e di tempo. Soddisfare tali vincoli all'interno di un'applicazione real-time è estremamente difficile, date le dimensioni dei dati esplorati e la complessità del rendering volumetrico. Questo ha portato i ricercatori ad introdurre svariate tecniche dedicate di visualizzazione adattiva approssimata e di strutture che sacrificano la qualità per velocità e risparmio di memoria(vedi Sez. 3.2). Tali metodi devono supportare, in pratica, un'ampia varietà di scenari di esplorazione dei dati, con vincoli di fedeltà temporale e spaziale molto diversi. Questi variano dall'esplorazione delle animazioni con l'adattamento delle impostazioni della fotocamera e delle impostazioni di rendering, finalizzate alla miglior percezione di effetti dinamici, allo spostamento rapido non lineare nel tempo, per identificare le fasi temporali interessanti, fino all'analisi di specifici timestep al fine di percepire nei dettagli una data figura. Inoltre, mentre le simulazioni sono calcolate su potenti macchine parallele e memorizzate su server, per esplorarne visivamente i risultati sono comunemente utilizzati PC grafici, workstation o cellulari [20]. Abbiamo quindi bisogno di trasferire in modo efficiente enormi quantità di dati dai server remoti allo storage e all'hardware grafico locale, nonché di gestire i problemi di scalabilità del rendering.

Negli ultimi anni sono state sviluppate molte architetture di rendering volumetrico diretto in tempo reale (DVR) basate su GPU. Esse impiegano metodi out-of-core, rappresentazioni in multirisoluzione, compressione e streaming dei dati, per consentire la visualizzazione interattiva di enormi dataset volumetrici. Mentre sono molto efficaci nell'esplorazione di dataset statici [21, 22, 23], le tecniche attuali non supportano pienamente l'esplorazione in tempo reale, con pieno controllo spaziale e temporale, di dati dinamici (vedere Sez. 3.2). In particolare, per far fronte alle limitazioni della larghezza di banda e alla limitata velocità di decodifica, molte delle tecniche attuali in real-time sono costrette a diluire gli aggiornamenti di un workingset di rendering su più frame. Questo approccio, tuttavia, pur riducendo la banda passante richiesta, introduce il risultato indesiderato di mescolare effetti dinamici reali con effetti spuri, creati dagli aggiornamenti incrementali.

In questo capitolo, introduciamo un nuovo approccio flessibile che sfrutta rappresentazioni multiple compresse all'interno di un renderer configurabile, con accelerazione GPU nel dominio di compressione, per supportare una varietà di mezzi di esplorazione per grandi volumi di campi scalari rettilinei, sia statici che dinamici (Sez. 3.3). Il lavoro è stato pubblicato su rivista e presentato a Eurovis 2019 [6], dove è stato selezionato come uno dei dei lavori di maggior spicco della conferenza (selezione delle prime cinque relazioni). Uno sviluppo successivo di questo lavoro, che sarà incluso nella versione finale del deliverable TDM previsto alla fine del progetto, ha vinto il best paper award a STAG 2019 [7].

Nel nostro approccio, un campo scalare rettilineo tempo-variante viene convertito off-line in una struttura dati out-of-core GPU-friendly, consistente in una sequenza temporale di piramidi multirisoluzione, una per frame, con una doppia rappresentazione near-lossless e lossy. La rappresentazione near-lossless è impiegata per supportare il rendering statico ad alta qualità, mentre la rappresentazione con perdita è appositamente concepita per la decodifica on-the-fly ad alta velocità. Estendendo i precedenti approcci di fixed-rate sparse-approximation [24], un nuovo algoritmo di allocazione dei bit approssima un tipico problema MCKP (Multiple Choice Knapsack Problem), per convertire in pagine di dimensioni fisse combinazioni lineari sparse a lunghezza variabile di blocchi prototipo, memorizzati in un dizionario completo. Ogni livello di risoluzione è organizzato in una griglia 3D gerarchicamente tassellata, che organizza i dati in pagine di bricks non sovrapposti di blocchi di voxel, supportando l'accesso spazio-temporale diretto e lo streaming alla GPU in formati compressi (Sez. 3.4). Un framework di rendering flessibile mescola e adatta i kernel della GPU per supportare senza soluzione di continuità vari casi d'uso dell'esplorazione, traendo

vantaggio dalla nostra organizzazione dei dati per eseguire efficientemente operazioni parallele in object-space e image-space (Sez. 3.5). Animazione e navigazione temporale esenti da artefatti temporali sono ottenute grazie a un algoritmo di raycasting adattivo cache-less su GPU, che invia alla GPU un workingset della rappresentazione lossy convenientemente identificato, usando una rapida decodifica transitoria per il rendering nel dominio di compressione. L'esplorazione dinamica di singoli frame è ottenuta incorporando un processo di streaming ray-guided, che supporta il feedback di visibilità e l'affinamento incrementale della rappresentazione lossy, così come gli snapshot in alta qualità generati da quella near-lossless, quando la camera smette di muoversi. Questo design supporta sia dei fat client, che eseguono l'algoritmo di rendering completo, sia dei thin client, che ricevono i dati delle immagini in streaming da un server di rendering (Sez. 3.6). Come risultato, il nostro approccio può supportare diversi casi d'uso di esplorazione, su fat e thin client. Questi includono l'animazione e la navigazione temporale immune da artefatti temporali di dataset con migliaia di frame di svariati miliardi di voxel, l'esplorazione dinamica di singoli frame e gli snapshot in alta qualità da dati near-lossless (Sez. 3.7).

La parte del lavoro presentata nel contesto TDM è la base matematica e informatica ed il design di riferimento del sistema, descritti in questo deliverable. Alla fine del progetto sarà inoltre rilasciato in open source una versione di riferimento del codec sviluppato, che dimostra il metodo codificando e decodificando singoli volumi su CPU. Vengono anche riportati benchmark di una versione interna dell'architettura proposta, che integra il codec che sarà rilasciato nel quadro di TDM.

Figura 3.2: ***Fat client e thin client.*** *Il nostro framework flessibile supporta l'esplorazione spazio-temporale completamente interattiva di simulazioni con migliaia di frame di miliardi di voxel. In alto: fat client che esegue il rendering locale su un touch screen 1920x1080 pilotato da un singolo PC grafico con NVIDIA GTX 1080Ti. In basso: thin client su tablet Android Samsung Galaxy Note Pro SM-P905.*

3.2 Lavori correlati

La visualizzazione di enormi dataset richiede soluzioni scalabili in termini di rappresentazione dei dati, partizionamento dati/lavoro e riduzione lavoro/dati. Trattiamo qui brevemente i metodi che sono più strettamente correlati ai nostri. Per una copertura più ampia, rimandiamo il lettore a studi consolidati su approcci di modellazione e visualizzazione per dati volumetrici tempo-varianti [18], compression-domain DVR [22], GPU-based large-scale DVR [23], e mobile DVR [20].

Gli approcci sensibili all'output richiedono il caricamento adattivo di dati compressi memorizzati in strutture out-of-core, che vanno, per dataset statici, da un set a singola risoluzione di bricks compressi [21] a strutture multirisoluzione come octrees [25, 26, 24, 27] o griglie gerarchiche di bricks [28, 29]. In questo contesto, per ottenere i massimi benefici, sono necessari un metodo di preprocessing scalabile, lo streaming compresso e la decompressione on-demand, veloce e spatially independent, sulla GPU [30]. I più semplici schemi fixed-rate supportati dall'hardware [31, 32, 33] consentono un accesso diretto generale ma presentano una flessibilità limitata in termini di compressione realizzabile e formati di dati supportati. Sono stati proposti anche diversi metodi basati sulla quantizzazione vettoriale, unita a texture mapping adattivo [34, 35, 36], tecniche di compressione LP (Laplacian Pyramid) [37], o codificatori basati su wavelet [30, 38], ma la qualità e i rapporti di compressione sono limitati dalla dimensione del dizionario della fase di quantizzazione vettoriale, che forza il risultato a contenere una quantità limitata di blocchi differenti [39], rendendo queste tecniche applicabili principalmente ai dati a bassa gamma dinamica (ad esempio, 8-16 bit interi). Un'alternativa alla quantizzazione vettoriale è l'approssimazione del tensore [40, 41], che si basa su una riduzione di rango di una base preferenziale specifica del dato. La decompressione, tuttavia, non può essere eseguita a velocità interattiva per interi timestep. Un'altra alternativa è il recente compressore in virgola mobile ZFP [42], che funziona meglio per una compressione ad alta qualità, ed è quindi utilizzato, nel nostro contesto, per il rendering ad alta qualità dei frame statici.

La compressione volumetrica 3D è spesso estesa a 4D per sfruttare la correlazione tra i timestep [43, 44, 45, 46, 47, 48]. Il principale limite di questi approcci full-4D è l'incremento di complessità nel movimento non convenzionale attraverso il tempo (accesso diretto, indietro, avanti veloce, ...), e la necessità di avere in memoria un piccolo insieme di frame per il rendering di un singolo timestep. Altre tecniche sfruttano la coerenza temporale, codificando ogni voxel rispetto ai key-frames di riferimento [49, 50, 51, 52, 53, 54, 15, 19, 16]. L'accesso diretto a diversi timesteps è più semplice, ma questi metodi soffrono di limitazioni simili rispetto ai precedenti: poiché i key-frame sono necessari per decodificare singoli timesteps, la dimensione dei dati che può essere trattata è ancora più severamente ristretta dalle limitazioni di memoria e di larghezza di banda. Una combinazione delle tecniche precedenti si può trovare in molti sistemi (ad esempio, [30, 55, 56, 57]).

La codifica individuale di ogni timestep usando metodi di compressione 3D (ad esempio, [24, 21, 58]) garantisce un accesso completamente casuale ed è meno limitante rispetto alla compressione video, in quanto dinamica. I volumi 3D forniscono già una dimensione in più, rispetto alle immagini 2D, da sfruttare per la riduzione dei dati. Per ottenere una buona compressione, tuttavia, sono necessari codificatori molto avanzati ai bassissimi bitrate richiesti dalla presentazione dinamica dei volumi. Treib et al. [21] utilizzano rappresentazioni a brick per timestep basate su wavelet, combinate con run-length encoding e entropy encoding. Pulido et al. [58] impiegano un approccio simile, supportando la visualizzazione scale-specific con l'invio dei soli coefficienti richiesti per

la specifica scala. Pur potendo ottenere eccellenti rapporti segnale-rumore e supportando l'esplorazione efficace di singoli timestep, non si ottengono prestazioni in tempo reale quando si avanza nel tempo, poiché la trasformazione inversa è relativamente costosa.

COVRA [24] impiega anche una rappresentazione sparsa, basata su un dizionario appreso. Mentre COVRA si concentra sull'esplorazione di dataset statici, il nostro framework differisce significativamente a livello di sistema. Il nostro layout con pagine di dimensione fissa migliora rispetto alla rappresentazione ad octree di COVRA, supportando l'accesso ai brick e l'identificazione della loro posizione spaziale attraverso il calcolo dell'indice, piuttosto che l'attraversamento della struttura (Sez. 3.4), che rende possibile la scrittura efficiente, attraverso kernel GPU di rendering che implementano in parallelo le operazioni object-space e image-space (vedi Sez. 3.5) e attraverso il loro mix-and-match, per supportare una varietà di use-case di esplorazione. Analogamente a COVRA, costruiamo il dizionario applicando l'algoritmo K-SVD [59] ad un piccolo sottoinsieme casuale ponderato di blocchi di volume. Il vantaggio di questo approccio sparse-coding è che la ricostruzione è ottenuta con una semplice combinazione lineare di blocchi, che può essere calcolata molto velocemente sulle GPU attuali. In termini di rappresentazione compressa, miglioriamo rispetto allo schema fixed-rate del COVRA, grazie a un nuovo encoder variabile-rate vincolato e a un migliore schema di allocazione dei bit, che aumenta le prestazioni di rate-distortion, pur producendo pagine di dimensioni fisse, per ottimizzare l'I/O e la gestione della memoria. Inoltre, il nostro schema produce bricks non sovrapposti, con un significativo guadagno di compressione rispetto ai brick con apron di COVRA e approcci similari, che mantengono un extra bordo di voxel attorno ad ogni brick, duplicando i valori oltre i suoi confini. Infine, COVRA si basa su aggiornamenti incrementali non conservativi e visualizza piccoli dataset dinamici decodificandoli completamente e precaricandoli in cache, mentre il nostro framework è progettato per supportare aggiornamenti dinamici completi da sequenze temporali illimitate, memorizzate out-of-core.

3.3 Panoramica

La nostra architettura flessibile si propone di supportare l'implementazione di sistemi di esplorazione interattivi, che consentano cambiamenti spazio-temporali completamente dinamici, così come un'accurata ispezione dei frame statici. È costituita da componenti di rendering ed elaborazione generale che possono essere combinati per implementare le desiderate caratteristiche di osservazione. La struttura generale di una configurazione tipica è rappresentata nella Fig. 3.3.

I dati di ingresso del processo sono una serie di T volumi scalari rettilinei di dimensione $N_x \times N_y \times N_z$. La nostra rappresentazione out-of-core, basata su livelli di griglie 3D tassellate gerarchicamente per-frame, supporta l'accesso diretto spazio-temporale e lo streaming alla GPU in formati compressi. Ogni frame, codificato indipendentemente da tutti gli altri, ha una doppia rappresentazione basata su due strutture parallele multirisoluzione di dati compressi: una struttura lossy altamente compressa e una near-lossless (Sez. 3.4). A run-time, la struttura altamente compressa viene utilizzata per eseguire il rendering durante l'interazione o l'animazione, in modo da consentire un adeguato sfruttamento della limitata larghezza di banda, disponibile per i dati variabili temporalmente. La rappresentazione near-lossless viene utilizzata, invece, per il rendering ad alta qualità nei frame statici. Entrambi i sistemi sono eseguiti da un framework di rendering flessibile, che mescola e combina i kernel per la GPU per supportare senza soluzione di continuità i diversi casi d'uso d'esplorazione (Sez. 3.5). Inoltre, mentre la rappresentazione high-bandwidth

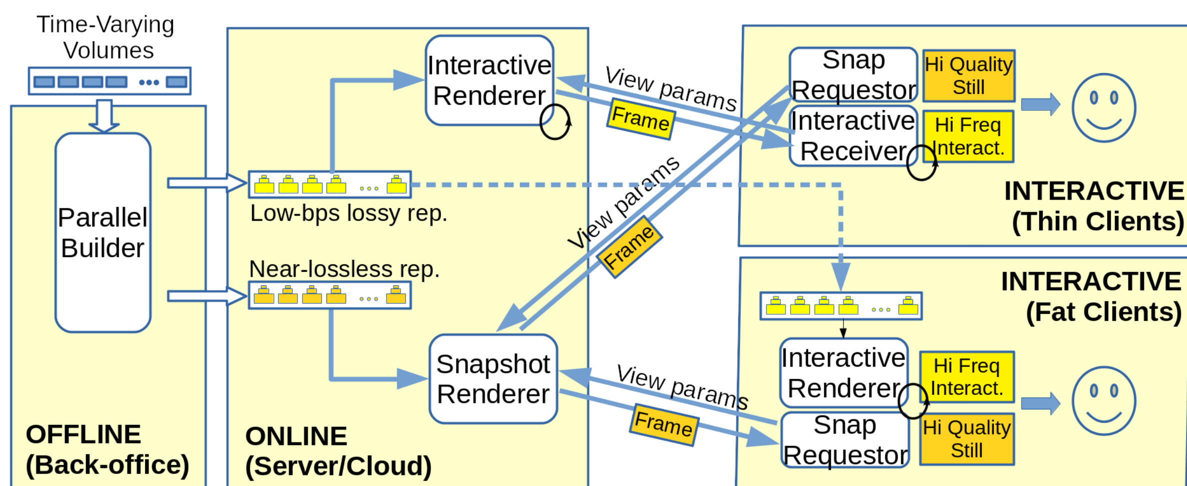

Figura 3.3: *Panoramica dell'architettura.* *Una rappresentazione near-lossless e una rappresentazione low-bitrate sono costruite a partire dai volumi scalari rettilinei in ingresso. La rappresentazione near-lossless è sfruttata per calcolare, lato server, frame statici di alta qualità, mentre quella low-bitrate viene utilizzata per la generazione ad alta frequenza di immagini. I thin client ricevono le immagini da un renderer lato server, mentre i fat client eseguono il rendering locale su una copia scaricata localmente di dati low-bitrate. frame*

è tipicamente mantenuta su un server remoto, la rappresentazione low-bandwidth può essere trasmessa a un fat-client per il rendering locale ad alta velocità o, opzionalmente, utilizzata sul server per fornire flussi di immagini interattive per thin-client, come quelli implementati, ad esempio, su dispositivi mobili (Sez. 3.6).

3.4 Layout dei dati e rappresentazione dei dati compressi

Per supportare LOD spaziali, ogni frame è rappresentato da una piramide di griglie 3D, ognuna delle quali è una rappresentazione a risoluzione progressivamente inferiore dello stesso volume rettilineo. Ogni livello inferiore rappresenta il volume utilizzando (approssimativamente, vedi sotto) la metà dei voxel in ogni dimensione.

La Fig. 3.4 illustra il layout di un singolo livello di risoluzione, a sua volta basato su una indicizzazione ricorsiva HTA (Hierarchically Tiled Array) [60] della griglia 3D di voxel, con pagine composte da P^3 brick compressi, composti da B^3 voxel. Le pagine rappresentano l'unità principale per il caricamento dei dati sulla GPU, mentre i brick sono usati come unità per tutte le operazioni della GPU. Per semplificare l'indicizzazione, l'estensione di ogni livello di risoluzione è regolata in modo da essere un multiplo di $P \cdot B$ per ogni asse. Poiché tutte le pagine contengono lo stesso numero di brick, e i brick contengono lo stesso numero di voxel, le pagine, i brick e i voxel possono essere indirizzati usando una funzione di layout che esegue semplici calcoli di indice. Questa tassellatura a due livelli migliora la località del riferimento. Tutte queste caratteristiche sono sfruttate nei nostri kernel di rendering (si veda la Sez. 3.5). Usando questo layout, il calcolo dell'indice successivo e di quello precedente richiede solo di conoscere il numero di pagine per livello e l'offset del brick di partenza di ogni livello. Queste informazioni sono precalcolate al momento della costruzione. Le conversioni avanti e indietro possono essere implementate direttamente con semplici somme, moltiplicazioni e operazioni div e mod.

Per ogni timestep, registriamo l'intervallo $v_{min}..v_{max}$ dei valori contenuti. Tre strutture HTA parallele con lo stesso layout vengono utilizzate per rappresentare un frame. La prima è un array di istogrammi, che contiene per ogni brick l'istogramma binario quantizzato dei valori contenuti nel brick stesso e di tutte le sue versioni a più alta risoluzione. Il secondo è la rappresentazione low-bitrate (vedi Sez. 3.4.2), mentre il terzo è la rappresentazione near-lossless (vedi Sez. 3.4.1).

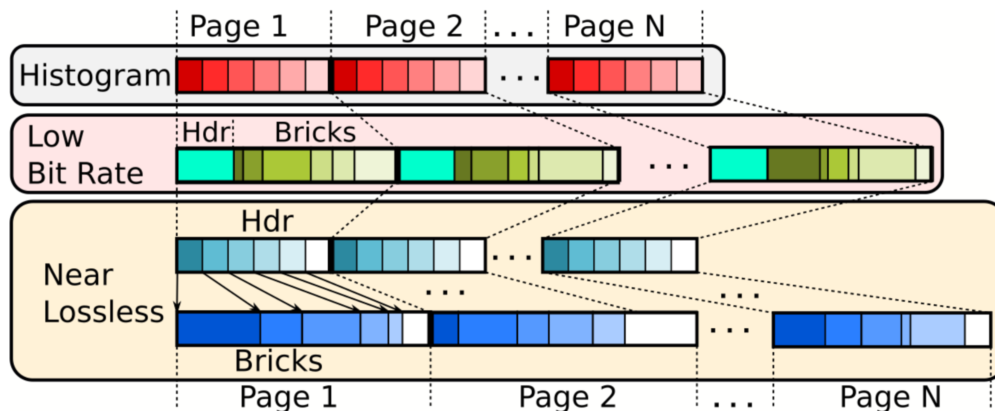

Figura 3.4: ***Layout dei dati.*** *Per rappresentare un frame vengono utilizzate tre strutture HTA parallele di dimensioni fisse, con stesso layout. Per supportare una codifica di alta qualità, la rappresentazione near-lossless utilizza la struttura di dimensioni fisse come indice della rappresentazione a variable-rate.*

Il processing inizia calcolando il numero di livelli necessari per coprire il volume. Successivamente è eseguito, possibilmente in parallelo per ogni timestep, il processo di compressione. Prima tutti i livelli della piramide multi-risoluzione in formato near-lossless (si veda Sez. 3.4.1) sono calcolati bottom-up e memorizzati su disco nel layout HTA. Poi si procede al calcolo del rappresentazione low-bitrate, che viene eseguita in due passi (vedi Sez. 3.4.2), prima con l'apprendimento dei parametri della rappresentazione, poi codificando tutte le pagine con i parametri appresi. Per velocizzare il processo, le singole pagine vengono codificate in parallelo. La parallelizzazione tra i frame viene effettuata utilizzando processi differenti, che possono essere eventualmente eseguiti su macchine diverse. La parallelizzazione all'interno del singolo frame avviene, invece, nell'ambito dello stesso processo, in quanto la condivisione della rappresentazione dell'output e la sincronizzare della scrittura sui file di frame è facilitata. Alla fine della costruzione, si memorizza su disco una piccolo header contenente le informazioni per il calcolo in avanti e all'indietro dell'indice (cioè numero di livelli, numero di pagine per livello e offset del brick iniziale del livello).

3.4.1 Compressione near-lossless per frame d'alta qualità

La rappresentazione near-lossless deve fornire una codifica di alta qualità dei suoi contenuti, che viene decodificata temporaneamente durante il rendering dei frame statici. Dal momento che non devono essere rispettati rigidi vincoli di frame-rate, e quindi di larghezza di banda, per il rendering dei frame statici adottiamo una rappresentazione generale, in cui ogni brick è codificato a bitrate variabile, per soddisfare i vincoli di qualità. Per garantire l'indicizzazione diretta dei brick, l'HTA a dimensione costante della rappresentazione near-lossless contiene solo un puntatore a 32 bit per i dati codificati dei brick. I dati dei brick sono memorizzati in un differente array out-of-core, nello stesso ordine dell'HTA. Diversi encoder di alta qualità possono essere utilizzati per la rappresentazione near-lossless di dati dei brick. Per questa configurazione distribuibile, utilizziamo

il codec ZFP [42] (versione 0.5.4 [61]). Utilizziamo la modalità di precisione fissa (che di solito produce i migliori tassi di compressione) e variamo la tolleranza di errore assoluta (`-a`).

3.4.2 Compressione low-bitrate per la presentazione dinamica dei dati

La rappresentazione low-bitrate è utilizzata per operazioni bandwidth-critical, che si verificano quando si trasferisce il dataset da un server remoto ad un fat client locale, o quando, durante le operazioni dinamiche, ad ogni frame si trasferiscono i dati dalla memoria di archiviazione alla memoria grafica. È quindi essenziale garantire la massima compressione, considerando, ad esempio, che un singolo frame da 1 Gvox (o workingset) richiederebbe 64 MB a 0,5 bps (bit per sample), corrispondenti a 64GB per lo streaming di animazioni a 1K-frame. Inoltre, il supporto di un'animazione a 10 fps richiede un codec in grado di decomprimere e visualizzare almeno 10 Gvox/s. A un bitrate così basso, le tecniche in grado di fornire le migliori approssimazioni, pur supportando la decompressione veloce su GPU, sono basate su rappresentazioni apprese [22, 41]. In questo lavoro, risolviamo un problema di ottimizzazione, inserendo nelle pagine a dimensione costante una approssimazione sparsa variable-rate dei blocchi di volume, che minimizza l'errore complessivo. Le pagine a dimensione costante consentono un'indicizzazione implicita, una facile gestione dell'I/O e della memoria, cosi' come un'occupazione di memoria costante e tempo quasi costante, per il trasferimento e la decodifica di porzioni locali del volume. Allo stesso tempo, la nostra approssimazione variable-rate porta a un basso errore, pur supportando una rapida decodifica parallela.

La nostra rappresentazione approssima ogni blocco b_i della gerarchia dei volumi attraverso una combinazione lineare sparsa di blocchi prototipo, memorizzati in un dizionario completo $\boldsymbol{D} \in \mathbb{R}^{m \times n}$, appreso dalla sequenza volumetrica in ingresso. Per questo, mappiamo ogni blocco b_i di dimensione $m = M^3$ su un vettore colonna $\boldsymbol{y}_i \in \mathbb{R}^m$. Il dizionario $\boldsymbol{D}$ è strutturato in modo da avere ogni blocco prototipo mappato su un vettore colonna $\boldsymbol{d}_k \in \mathbb{R}^m$ di lunghezza unitaria. Si assume che la prima colonna $\boldsymbol{d}_1$ sia la constante $\boldsymbol{d}_1 = \frac{\mathbf{1}}{\sqrt{m}}$ non utile al training del dizionario, mentre le altre sono mantenute a media nulla. Dato $\boldsymbol{D}$, la nostra rappresentazione compressa per un blocco $\boldsymbol{y}_i$, consiste quindi in un insieme di indici k_i e associati coefficienti $\boldsymbol{\gamma}_i$ non nulli, tale che $\boldsymbol{y}_i \approx \sum_{k=1}^{K_i} \gamma_{ik} \boldsymbol{d}_{k_{ik}}$, dove K_i è il numero di coefficienti non nulli nella rappresentazione del blocco i. Al fine di inserire rappresentazioni a lunghezza variabile dei blocchi in una dimensione di pagina fissa, si esegue prima l'apprendimento del dizionario e poi, col dizionario acquisito, si esegue la codifica in modo da soddisfare i vincoli di dimensione.

3.4.2.1 Apprendimento del dizionario

In primo luogo, data la dimensione desiderata della pagina, calcoliamo la media K del numero di coefficienti non nulli, necessaria per raggiungere la taglia desiderata. Il dizionario ottimale $\boldsymbol{D}$ viene poi calcolato ottimizzando congiuntamente le colonne $2..n$ del dizionario e la rappresentazione sparsa secondo la funzione obiettivo

$$\min_{\boldsymbol{D},\gamma_i} \sum_i w_i \left\|\boldsymbol{y}_i - \boldsymbol{D}\gamma_i\right\|_2^2 \text{ subject to } \forall i, \left\|\boldsymbol{\gamma}_i\right\|_0 \leq K \tag{3.1}$$

dove $\|\gamma_i\|_0$ conta i valori non nulli di $\boldsymbol{\gamma}_i$ e w_i é un peso associato ad ogni campione di training. Per il training utilizziamo una variante ponderata di K-SVD [59], che esegue l'apprendimento del

dizionario solo su un piccolo sottoinsieme casuale ponderato dei campioni originali (un core-set), invece che su tutti i campioni di input, rendendo possibile l'apprendimento per dati massivi [24]. Poiché i blocchi costanti possono essere codificati banalmente per effetto di $\boldsymbol{d}_1$, stimiamo per ogni blocco in ingresso il potenziale errore residuo $e_i = \|\boldsymbol{y}_i - (\boldsymbol{y}_i \cdot \boldsymbol{d}_1)\boldsymbol{d}_1\|_2^2$. Al fine di costruire un core-set di dimensioni C, preleviamo per il training campioni con probabilità proporzionale a e_i, utilizzando un metodo streaming one-pass basato sul weighted reservoir sampling [62], assegnando un peso proporzionale al reciproco della probabilità di prelievo, per tenere conto del campionamento non uniforme in ingresso. In contrasto con COVRA [24], estraiamo il core-set a costo trascurabile, durante la costruzione bottom-up della piramide sottocampionata e la rappresentazione near-lossless.

Algorithm 1: Elastic OMP. *Algoritmo per l'allocazione ottimale di codici sparsi in pagine di dimensioni fisse*

Data: Dictionary $\boldsymbol{D}$, input blocks $\boldsymbol{y}_i$, page size P, brick size B, block size M, average non-zero count K
Result: For all i, encoding I_i, γ_i such that $\boldsymbol{y}_i \approx \sum_k \gamma_{ik} \boldsymbol{d}_{(I_i)_k}$ and $\sum_i count(I_i) = (PB/M)^3 K$

```
   //Initialize
1  K_P = (PB/M)^3 K  //Number of non-zeros in page
2  G = D^T D  //Precomputed Gram matrix
   //Compute solution for sparsity 0
3  foreach block i in 1..(PB)^3 do
4      p_i = D^T y_i ; L_i = [1]
       //Compute null solution
5      I_i = {} ; γ_i = {} ; α_i = p_i
       //Compute next best grow direction
6      k_i^+ = {argmax_k ||α_ik||}
7      γ_i^+ = {p_{i k_i^+}} ; α_i^+ = p_i − G_{k_i^+} γ_i^+ ; δ_i^+ = γ_i^+^T (p_i − α_i^+)
8      if δ_i^+ > 0 then push pair(δ_i^+, (k_i^+, γ_i^+, α_i^+)) into priority queue Q
9  end
   //Greedy grow until page filled
10 while Σ_i count(I_i) < K_p and Q is not empty do
       //Select block with largest decrease in error
11     pop pair(δ_i^+, (k_i^+, γ_i^+, α_i^+)) from priority queue Q
       //Move to next best solution for the block
12     I_i = {I_i ∪ k_i^+} ; γ_i = γ_i^+ ; α_i = α_i^+
       //Compute next best grow direction
13     k_i^+ = {argmax_k ||α_ik||}
14     w = solve for w { L_i w = G_{I_i} }  //Update Choleski decomposition
15     L_i = [ L_i   0 ; w^T   sqrt(1 − w^T w) ]
16     γ_i^+ = solve for c { L_i L_i^T c = α_i } ; α_i^+ = p_i − G_{k_i^+} γ_i^+ ; δ_i^+ = γ_i^+^T (p_i − α_i^+)
17     if δ_i^+ > 0 then push pair(δ_i^+, (k_i^+, γ_i^+, α_i^+)) into priority queue Q
18 end
```

3.4.2.2 Sparse-coding elastico

Dato il dizionario ottimale $\boldsymbol{D}$ per un dato frame, procediamo a calcolare l'approssimazione ottimale che si adatta alle nostre pagine a formato fisso. Per fare ciò, codifichiamo le pagine nell'ordine dettato dal nostro layout HTA, durante un unico passaggio in streaming del volume di ingresso. Il nostro obiettivo è quello di trovare, per tutti i blocchi i in una data pagina composta da P^3 brick

compressi, composti da $(B/M)^3$ blocchi, la migliore codifica $I_i, \boldsymbol{\gamma}_i$ tale che $\boldsymbol{y}_i \approx \sum_k \gamma_{ik} \boldsymbol{d}_{I_{ik}}$, e $\sum_i count(I_i) = (PB/M)^3 K$. Il problema generale di scelta delle dimensioni ottimali per la rappresentazione dei blocchi è un'istanza di MCKP (Multiple Choice Knapsack Problem), notoriamente NP-hard. Esistono molte soluzioni euristiche efficienti [63], ma la loro applicazione richiederebbe la precomputazione di tutti i possibili errori associati ad ogni possibile valutazione non nulla del blocco i. Approfittiamo qui dell'esistenza di soluzioni incrementali efficienti per codifiche sparse, per proporre un'approssimazione "greedy" al problema dell'allocazione.

Il nostro approccio si basa su una generalizzazione dell'algoritmo greedy batch-OMP [64], che è stato introdotto per la codifica di un gran numero di segnali sullo stesso dizionario. Ad ogni passo, l'algoritmo batch-OMP seleziona la colonna del dizionario con la più alta correlazione con l'attuale residuo, proietta ortogonalmente il segnale in ingresso sull'insieme delle colonne selezionate nel dizionario e, se la convergenza non è stata raggiunta, ricalcola il residuo e ripete il processo. Elevate prestazioni si ottengono sostituendo, nel passo di ortogonalizzazione dell'Orthogonal Matching Pursuit standard, la pseudo-inversa con un aggiornamento progressivo di Choleski. Sfruttiamo questo approccio di aggiornamento progressivo per calcolare in modo greedy l'adattamento ottimale della rappresentazione sparsa ad una pagina (vedi Algoritmo 1). Prima, calcoliamo la matrice di Gram $\boldsymbol{G} = \boldsymbol{D}^\intercal \boldsymbol{D}$ impiegata nella ricerca di correlazione e nelle fasi di ortogonalizzazione. Poi, per ciascuno dei blocchi della pagina, calcoliamo la soluzione iniziale, utilizzando coefficienti nulli, e il residuo associato. Calcoliamo anche la miglior soluzione successiva eseguendo un passo del metodo Batch-OMP e la inseriamo in una coda di priorità, ordinata in base alla massima riduzione dell'errore residuo. Continuiamo quindi iterativamente rimuovendo il miglior candidato dalla coda, aumentando il suo conteggio di coefficienti non nulli della rappresentazione, applicando la soluzione precalcolata e eseguendo ancora un passo del metodo Batch-OMP, per calcolare la soluzione successiva da inserire nella coda. Il metodo si ferma quando il vincolo di dimensione complessiva è soddisfatto.

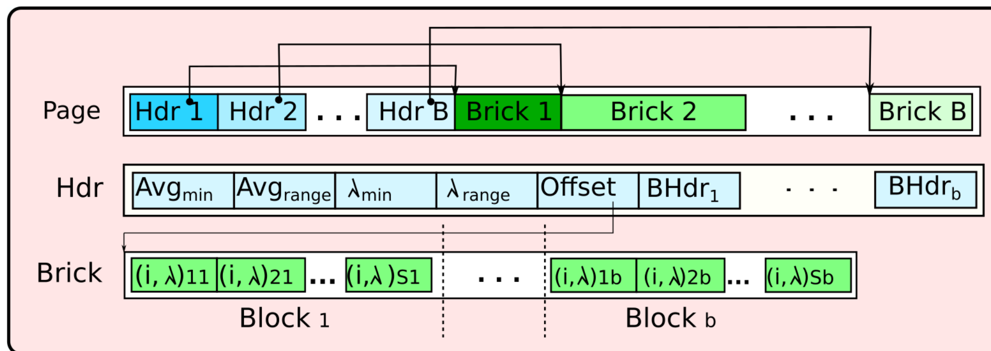

Figura 3.5: *Layout pagina di dati lossy.* *Una pagina di dimensione costante è composta da B intestazioni di brick e da B blocchi di dati dei brick. Le intestazioni (seconda fila) sono a dimensione costante e puntano all'associato blocco di dati a dimensione variabile.*

Quando l'algoritmo greedy termina, memorizziamo la soluzione nel formato di pagina compatto a dimensione costante, rappresentato nella Fig. 3.5. Ogni blocco è approssimato utilizzando un valore medio più un numero variabile di coppie (indice, valore), per la rappresentazione sparsa. I primi byte della rappresentazione della pagina sono dedicati a B intestazioni di brick a dimensione costante, che puntano al contenuto di brick a dimensione variabile. Il layout dei brick utilizza parole allineate a 32 bit ed è progettato per la massima prestazione in decodifica, piuttosto che per la massima compressione. In particolare, non applichiamo alcuna trasformazione di bit o entropy

coding. L'intestazione del brick contiene 4 float a 32 bit per la dequantizzazione (intervallo di media e intervallo di coefficienti), un offset a 32 bit per la rappresentazione del blocco e b descrittori di blocchi a 32 bit. Ogni descrittore contiene il contatore non-zero a 8 bit del blocco, una versione quantizzata a 12 bit della media del blocco e due versioni quantizzate a 6 bit dell'intervallo dei coefficienti del blocco. L'intervallo è espresso rispetto all'intervallo dei coefficienti dell'intero blocco. Il resto della pagina è dedicato alla rappresentazione sparse-coded, che concatena, per ogni brick, le (k, γ) coppie, utilizzando 16 bit a coppia, con $bitcount(k) = \log_2(n)$ e $bitcount(\gamma) = 16 - bitcount(k)$. L'uso di tali allineamenti permette al kernel di decodifica (si veda la Sez. 3.5.2) di leggere la rappresentazione dei dati con accessi allineati a 32 bit, così da non generare conflitti di accesso alla memoria della GPU, legati alla sua struttura in banchi.

L'approccio si dimostra molto appropriato per la codifica low-bitrate nel nostro contesto, in quanto fornisce per i dati in virgola mobile una qualità allo stato dell'arte a meno di 0,5 bps, garantendo al contempo una rapida decodifica ad accesso diretto (si veda Sez. 3.5). In particolare, le prestazioni di decodifica su GPU sono sufficientemente elevate ($\gg$10 GVox/s su una NVIDIA GeForce GTX 1080Ti) da rendere possibile l'uso di questa codifica per l'esplorazione temporale di ingenti dataset dinamici. Si veda la Sez. 3.7 per maggiori dettagli.

3.5 Rendering adattivo da dati compressi

La nostra architettura di rendering su GPU sfrutta il fatto che i singoli frame possono essere completamente memorizzati nella memoria grafica in formato HTA, utilizzando la rappresentazione low-bitrate, compatibile con le GPU. Questo ci permette di implementare il processo di rendering ad alta larghezza di banda, replicando nella memoria grafica, attraverso un processo di paging, la rappresentazione out-of-core e di implementare decompressione transitoria e rendering come operazioni di GPU, operanti su strutture residenti.

3.5.1 Renderer configurabile con accelerazione GPU

Il processo di rendering di un singolo timestep è descritto nell'Algoritmo 2, ed è reso efficiente sfruttando una serie di strutture dati visibili ai kernel CUDA (fedi Fig. 3.6).

Per supportare il rendering di più timestep alla volta (vedi Sez. 3.6), all'avvio preallochiamo spazio per il numero massimo di timestep che possono essere visualizzati insieme. Ad ogni frame, dato il timestep selezionato da visualizzare, vengono caricati gli istogrammi e il dizionario, se non sono già presenti (linea 1). Le informazioni necessarie per il calcolo degli indici in avanti e a ritroso (Sez. 3.4) vengono invece caricate una volta per ogni dataset, in quanto sono costanti per tutti i frame. Poi, in parallelo per ogni brick candidato, dati i parametri di visualizzazione V, la funzione di trasferimento corrente τ, e, se l'animazione non è attualmente attiva, il feedback di visibilità del frame precedente, determiniamo l'insieme dei brick che formano la rappresentazione a risoluzione variabile della porzione attualmente potenzialmente visibile del dataset (line 2). Allo stesso tempo, gli indici dei brick selezionati vengono inseriti in una griglia spaziale di indici, che è una griglia regolare con la stessa estensione spaziale del volume e celle con la dimensione dei brick nel livello gerarchico più dettagliato. Si ordinano quindi gli indici dei brick potenzialmente visibili in un numero di layer ortogonali all'asse più prossimo alla direzione di vista (line 4). Il numero di layer dipende dalla situazione, poiché vengono utilizzate per effettuare il rendering di frame troppo

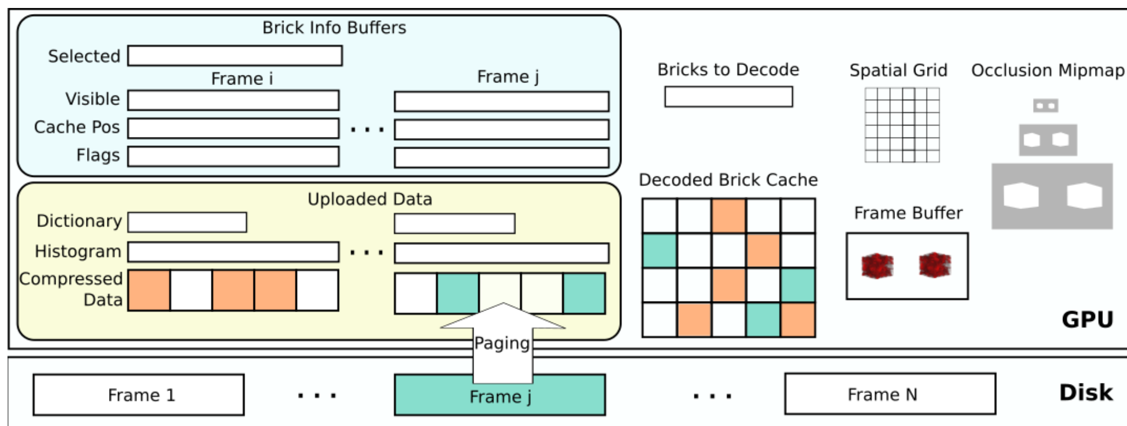

Figura 3.6: ***Strutture dati GPU.*** *Supportiamo il rendering di diversi timestaps in un unico frame. In GPU, eseguiamo il rendering accedendo ai brick memorizzati nella struttura HTA. La rappresentazione low-bandwidth viene mappata nella memoria grafica attraverso un processo di paginazione. Una cache di brick decodificati e altre strutture transitorie supportano lo streaming e il rendering ray-guided.*

Algorithm 2: Processo di rendering *Struttura dell'algoritmo di rendering su GPU a partire da dati out-of-core.*

Data: timestep t, view parameters V, transfer function τ, screen-space tolerance ϵ
Result: updated framebuffer, occlusion info for ray-guided streaming, and GPU caches

```
1  init(t)
2  identify_renderable_sets(V,τ, ε)
3  cleanup_cache()
4  sort_renderable_set_by_slab(V)
5  foreach slab in front to back order do
6  |   cull_occluded_bricks()
7  |   identify_bricks_to_decode()
8  |   upload_missing_pages()
9  |   associate_bricks_to_cache_positions()
10 |   decode_bricks()
11 |   fill_brick_aprons()
12 |   raycast(V,τ)
13 |   update_occlusion_mipmap()
14 end
15 pullup_visibility()
```

grandi per essere decodificati completamente in una sola volta sulla GPU, oltre che per evitare inutile decodifica per effetto dell'occlusion culling. Se stiamo esplorando un singolo frame e l'intero workingset del frame corrente può stare in cache, il numero di slabs è impostato a 1 per eseguire il rendering a passaggio singolo, altrimenti è impostato di default su un numero piccolo prestabilito (2 per i dati statici che beneficiano del caching multi-frame, 8 per i dati dinamici che aggiornano la cache ad ogni frame), ma le slabs che contengono più del numero di brick che possono essere decodificati in una sola volta e memorizzati sulla GPU sono ulteriormente suddivise. Il processo di rendering procede quindi su base slab-by-slab (linee 5-14), attraversandole front-to-back. In primo luogo, si riduce ulteriormente il workingset rimuovendo dal set potenzialmente visibile quei brick che sono occlusi dai dati visualizzati in un layer precedente (linea 6). Questo processo di eliminazione conservativa, attivo anche per i dataset dinamici, è supportato da un occlusion buffer mipmappato, che mantiene l'opacità raggiunta per ogni pixel. I brick sono considerati occlusi se la loro proiezione è coperta da pixel a piena opacità. I brick che sopravvivono sono contrassegnati

come necessari per il rendering della slab corrente (linea 7). Poiché i brick out-of-core non si sovrappongono, il workingset viene espanso con i brick che contengono i 2 voxel extra necessari per l'interpolazione trilineare e il calcolo del gradiente.

Una cache di brick decodificati supporta il compito di rendering: se abbiamo a che fare con dati statici, i brick decodificati vengono messi in cache per il riutilizzo su più fotogrammi, altrimenti la cache viene pulita ad ogni nuovo frame. Se stiamo effettuando il rendering a partire dalla rappresentazione low-bitrate, includiamo dapprima tutte le pagine che contengono brick necessari, ancora non inclusi nella cache dei brick decodificati (linea 8). Per la rappresentazione near-lossless questo non è necessario, poiché non carichiamo e manteniamo la rappresentazione compressa nella memoria grafica in modo incrementale. In entrambi i casi, la cache dei brick decodificati viene aggiornata. I brick nella cache sono calcolati per essere autosufficienti durante il rendering, e devono quindi avere un apron di 2 voxels, cioè un bordo, intorno ad ogni brick, che duplica i valori oltre i confini del brick stesso, per consentire di sfruttare le operazioni di texturing per il filtraggio trilineare e il calcolo dei gradienti. L'aggiornamento della cache decodificata viene quindi eseguito in due passi, aggiornando prima i voxel interni dalla decodifica dei brick di una rappresentazione compressa (linea 9-10), successivamente riempiendo i voxel dell'apron con l copia dei dati dai brick vicini già decodificati (linea 11). Evitando di memorizzare apron di 32^3 brick si risparmia il 42% dello spazio di memorizzazione e della larghezza di banda per il trasferimento dei dati.

Con tutti i dati richiesti in cache, il rendering viene poi eseguito tramite raycasting (Linea 12), seguendo i raggi per-pixel limitati all'attuale estensione del layer. Mentre tutte le operazioni precedenti sono state parallelizzate brick per brick, questo passaggio utilizza un thread GPU per pixel. L'attraversamento dei raggi sfrutta l'indice spaziale, calcolato dinamicamente, per l'identificazione del brick e per il salto dello spazio vuoto, e supporta la terminazione anticipata dei raggi, fermandosi al raggiungimento della massima opacità e aggiornando la mipmap di occlusione (aggiornata nella linea 13 e utilizzata nella linea 6). Inoltre, per supportare lo streaming ray-guided, lo stato del feedback di visibilità di ogni brick attraversato è impostato su true.

Dopo il rendering di tutti i layer, il frame-buffer contiene l'immagine finale composta per il volume e lo stato di visibilità di tutti i brick visualizzati è aggiornato e può essere usato per guidare il passo di perfezionamento del frame successivo (linea 2). Poiché è calcolato solo per i brick che sono foglie della rappresentazione corrente, alla fine del frame lo stato di visibilità viene propagato a livelli di griglia meno dettagliati, considerando visibile un brick se almeno uno dei suoi brick di livello più fine è stato attraversato (15).

3.5.2 Operazioni con accelerazione GPU

Lo schema di rendering può essere completamente implementato usando una serie di operazioni ben definite della GPU. Le operazioni effettuate in fase di inizializzazione sono le seguenti.

- **init()** verifica prima di tutto se il timestep corrente è nel set attivo e, in caso contrario, carica nella memoria grafica il suo istogramma e il dizionario per la decodifica low-bitrate. Se l'insieme attivo è pieno, e tutti i timestep attivi sono per il frame corrente, quello usato più di recente viene sostituito, altrimenti, viene sostituito quello più vecchio. Impostiamo quindi la viewport al colore di sfondo, la mipmap dell'occlusione a opacità nulla, e lo stato visitato dei brick a false.

- **identify_renderable_sets()** viene eseguito come un singolo kernel di GPU, con un thread per ogni brick. Poiché la nostra mappatura ci permette di ricavare dalla posizione del brick il suo livello e la posizione 3D, tutti i brick possono essere gestiti in parallelo. I brick sono così contrassegnati come parte del set potenzialmente visibile, se interni al view-frustum e comprendenti qualche voxel a opacità non nulla, secondo l'istogramma precalcolato e i loro voxel proiettati alla dimensione corretta, secondo l'attuale tolleranza dello screen-space. Quando l'occlusione inter-frame è abilitata, scartiamo il brick se sia lo stesso che il suo genitore non erano visibili sul frame precedente, secondo il buffer di visibilità. Se il brick è selezionato, il kernel lo segna come foglia sul buffer dei flag e ne scrive l'id e l'indice di slab nel buffer dei brick selezionati (si veda la Fig. 3.6). Per saltare efficientemente lo spazio vuoto, tutte le celle dell'indice spaziale coperte dal brick sono inizializzate con l'identificazione del brick $(level, x, y, z)$ (4 x uint8), dove $level$ ha il bit più alto impostato ad uno se il brick è vuoto. Al termine del kernel, quindi, l'indice spaziale indicizza correttamente la rappresentazione multirisoluzione utilizzata per il frame corrente, in modo che, durante la traversata, sia possibile inserire un brick, accedere ai suoi dati attraverso la funzione di layout, e avanzare al brick successivo, utilizzando l'estensione dettata dal livello.

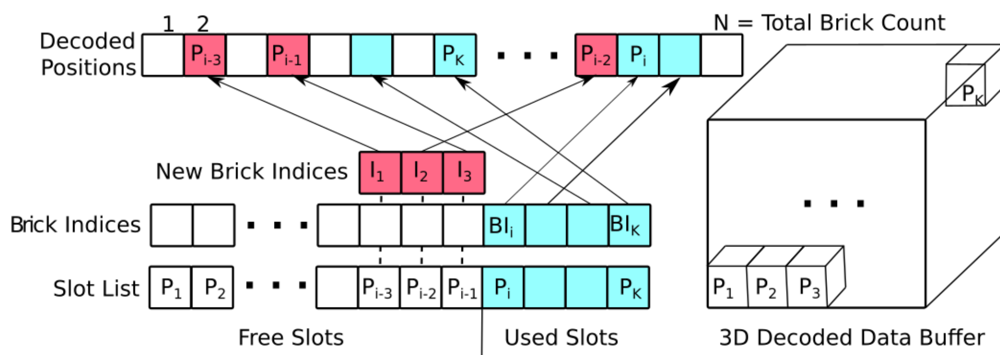

Figura 3.7: ***Cache buffers.*** *La lista degli slot contiene tutti gli slot del buffer 3D. L'indice del brick corrispondente viene memorizzato per ogni slot utilizzato (in ciano). Gli indici dei nuovi brick da decodificare sono in magenta. Gli slot corrispondenti vengono copiati dalla lista degli slot nel buffer delle posizioni decodificate.*

- **cleanup_cache()** viene eseguita come kernel di GPU, con un thread per ciascuno degli slot d'identificazione d'uso della cache, seguito da una fase di compattazione della GPU. La cache, si veda la Fig. 3.7, è organizzata in due buffer paralleli, con il primo contenente tutte le posizioni dei brick residenti e il secondo contenente l'associato id composto *(brick, time step slot)*. L'associazione dello slot di timestep all'id permette di gestire più timestep contemporaneamente. I primi $available_slot_counter$ slot sono liberi, i restanti sono quelli utilizzati. Questo kernel imposta a liberi tutti gli slot che puntano a brick non facenti parte dell'attuale workingset. Una fase di compattazione della GPU posiziona quindi gli slot liberi prima di quelli usati e aggiorna $available_slot_counter$.
- **sort_renderable_set_by_slab()** esegue l'ordinamento via GPU del buffer dei brick selezionati in base all'id della slab. Dopo questa operazione, tutti i brick selezionati appartenenti ad un layer possono essere facilmente acceduti linearmente attraverso questo buffer.

Dopo queste operazioni iniziali, iteriamo la seguente sequenza di passi per ogni layer.

- **cull_occluded_bricks()** viene eseguito come kernel GPU, con un thread per ogni brick selezionato nel layer corrente. Sfruttando la mipmap di occlusione, si controlla se il brick è

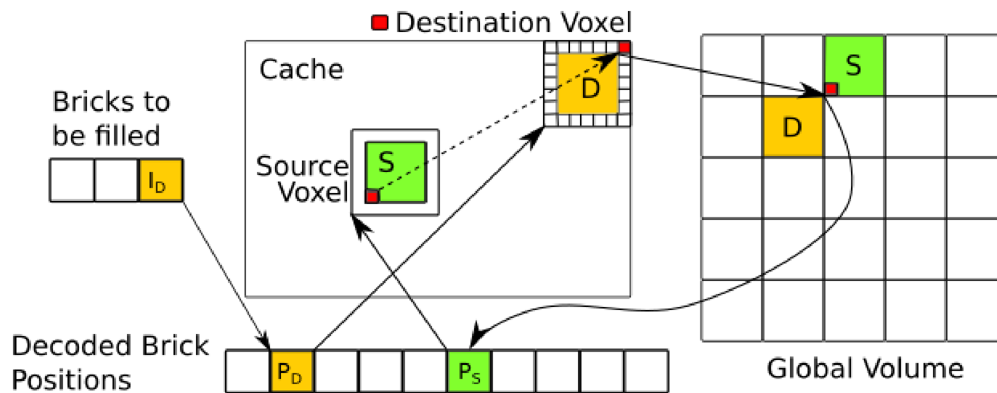

Figura 3.8: ***Riempimento apron.*** *I voxel di apron del brick di destinazione D (giallo) vengono riempiti seguendo questi passaggi: (1) trovare lo slot cache di I_D; (2) dallo slot cache e dalla posizione del voxel, calcolare il $(level, x, y, z)$ globale; (3) convertire $(level, x, y, z)$ in un offset nel layout HTA per identificare il brick sorgente (verde); (4) trovare la posizione corrispondente nella cache, prelevare voxel e copiarlo nella posizione di destinazione. Per semplicità è qui rappresentato un solo strato di apron, invece di due.*

occluso, identificando a quale livello della mipmap il box proietta su un singolo pixel e poi verificando se tutti i 2x2 pixel che coprono la proiezione del box sono contrassegnati in modo conservativo come completamente occlusi. Il buffer di flag dei brick viene aggiornato di conseguenza.

- **identify_bricks_to_decode()** viene eseguito come kernel GPU, con un thread per ogni brick selezionato nel layer corrente. È responsabile di identificare quali sono i brick che devono essere decodificati, ovvero i brick attualmente non occlusi nel workingset, che non sono già in cache. In questa fase si identificano anche gli *auxiliary bricks* necessari per il calcolo dei voxel dell'apron. Quindi, per ogni brick attivo, controlliamo tutti i suoi 26 vicini e contrassegniamo come *auxiliary* quelli non contrassegnati come attivi e non presenti in cache.
- **upload_missing_pages()** è responsabile della replica in memoria grafica delle pagine della rappresentazione low-bitrate contenenti i brick attivi. Questa operazione può essere eseguita automaticamente utilizzando CUDA Unified Memory Access (UMA) [65], mappando il file out-of-core nella memoria grafica. Abbiamo tuttavia scoperto che si possono ottenere prestazioni significativamente più elevate implementando la paginazione in un esplicito gestore di memoria, aggiornando un piccolo buffer di flag (un byte/pagina) durante l'identificazione del brick, scaricandolo sulla CPU e spostando su GPU con `cudaMemcpyAsync()` le pagine mancanti.
- **associate_bricks_to_cache_positions()** viene eseguito come kernel GPU, con un thread per ogni brick da decodificare. Il thread associa al brick la posizione disponibile nella lista degli slot, in corrispondenza del proprio id. Le posizioni della cache sono copiate nel buffer delle posizioni decodificate, mentre gli indici dei brick sono scritti nel buffer di indici della cache. Infine, $available_slot_counter$ viene aggiornato.
- **decode_bricks()** calcola la rappresentazione del brick decompresso, comportandosi in modo diverso nel rendering dei frame statici dalla rappresentazione near-lossless o in quello degli aggiornamenti ad alta frequenza dalla rappresentazione low-bandwidth. In tutti i casi, vengono calcolati solo i voxel interni dei brick, dato che i voxel dell'apron saranno riempiti separatamente in seguito. La decodifica di ogni brick è quindi totalmente locale. La

versione decompressa di ogni brick è caricata nella cache dalla struttura near-lossless, sfruttando il codec ZFP [61]. Con un kernel di GPU per ogni brick, la decompressione dalla rappresentazione low-bitrate inizia col calcolo dell'offset nella rappresentazione compressa di ogni blocco variable-rate e della versione dequantizzata della media e dell'intervallo dei coefficienti del blocco. La decompressione viene poi avviata usando un thread per voxel decodificato, con una dimensione di griglia pari alla dimensione del blocco. I thread di decodifica caricano in modo cooperativo la rappresentazione compressa del blocco nella memoria condivisa. Ogni thread carica 32 bit e decodifica nella memoria condivisa della GPU due coppie indice-coefficienti, onde evitare conflitti di accesso legati alla sua struttura in banchi. Dopo la sincronizzazione dei thread, ogni thread calcola separatamente la combinazione lineare dei suoi elementi associati, necessaria per calcolare il voxel associato.

- **fill_brick_aprons()** calcola l'apron 2-voxel di ogni brick appena decodificato che non è contrassegnato come ausiliario. L'operazione viene eseguita da tre kernel GPU: uno per i layer top+bottom, uno per quelli left+right, uno per quelli back+front di ogni brick. Abbiamo sperimentalmente determinato che la decodifica di 4 voxel per thread è un buon compromesso tra le operazioni di calcolo e quelle di lettura/scrittura. I thread sono associati ai voxel di confine dei brick di destinazione. Per ognuno di questi voxel, l'originale brick sorgente e il voxel associato vengono identificati sfruttando il buffer di posizione decodificata, che mappa le posizioni globali sui brick decodificati, e la nota corrispondenza tra le coordinate locali del brick sorgente e di quello di destinazione (Fig. 3.8). Questa procedura viene applicata una volta per layer, per i brick appena decodificati privi di apron.
- **raycast()** esegue il rendering e l'accumulo della porzione di raggi nel layer corrente, accedendo alle informazioni decodificate attraverso la griglia spaziale. L'operazione viene eseguita utilizzando un raggio per thread, che attraversa la griglia regolare con approccio DDA. Utilizzando la griglia spaziale si ottiene un approccio più veloce rispetto al KD-restart [66], o ai puntatori ai vicini [67], perché ci permette di saltare tutte le ulteriori discendenze gerarchiche, che sono generalmente utilizzate per attraversare altre simili strutture gerarchiche di volume. Ogni volta che si accede a una cella della griglia, il tipo, la posizione e la dimensione del brick indicizzato sono calcolati a partire dal $(level, x, y, z)$ della cella memorizzato. I brick vuoti sono accumulati in una sola volta, mentre i brick non vuoti sono acceduti recuperando la loro posizione in cache dal buffer di posizione dei brick decodificati, prima di eseguire l'accumulo di tutti i voxel. Poi l'accumulo continua spostandosi verso la cella di uscita, determinata dalla direzione del raggio e dalla dimensione del brick. Il processo si ripete fino a quando il raggio non si ferma, o per completa opacità o perché raggiunta la fine del layer. Quando un brick produce un contributo non nullo, un flag di visibilità viene impostato a uno nel buffer di visibilità. La scrittura può essere fatta simultaneamente senza la necessità di funzioni atomiche, perché tutte le scritture dei thread concorrenti avranno lo stesso valore e non ci sarà conflitto. Inoltre, raggiunta l'occlusione completa, viene aggiornato il valore di occlusione del pixel.
- **update_occlusion_mipmap()** costruisce la mipmap dei valori di occlusione partendo dalla mappa d'occlusione di livello più accurato, in modo da permettere un efficiente calcolo dell'occlusione intra-frame. Un kernel con un thread per ogni pixel della versione più grossolana della mappa viene eseguito per ogni livello di mipmap. La intra-frame occlusion-culling è completamente conservativa, ed è usata per ridurre il numero di brick decodificati.
- **pullup_visibility()** aggiorna i valori di visibilità dei nodi interni partendo dai valori dei nodi

visitati a livello di foglia, al fine di consentire un efficiente calcolo dell'occlusione inter-frame, per lo streaming ray-guided. Questa operazione viene eseguita bottom-up nella CPU, marcando come invisibili tutti i nodi al di sotto della rappresentazione attualmente visualizzata, e come visibile ogni nodo per il quale è visibile almeno un figlio. L'occlusion culling inter-frame viene utilizzata per ridurre il caricamento e la decodifica dei dati attraverso lo streaming ray-guided. Poiché questa operazione non conservativa può portare ad effetti dinamici indesiderati, dovuti al caricamento incrementale durante le animazioni, viene attivata solo per il rendering di frame statici.

3.6 Distribuzione dei dati e design di applicazioni di rendering

Il nostro processo di rendering flessibile consente di generare una varietà di sistemi che sfruttano sia l'alto tasso di compressione che l'alta qualità dei dati. La configurazione predefinita, rappresentata in Fig. 3.3, supporta sia i thin client, che ricevono tutte le immagini renderizzate dai renderer lato server, sia i fat client, che eseguono il rendering locale ad alta frequenza su una copia low-bitrate scaricata localmente e utilizzano un processo separato (locale o su un server di rendering remoto) per generare snapshot di alta qualità da dati near-lossless.

Nella nostra implementazione di riferimento, utilizzata internamente per testare il sistema, un'interfaccia utente sposta liberamente la telecamera, modifica la funzione di trasferimento e determina quali intervalli di tempo vengono mostrati. I controlli disponibili includono l'uso di un jog shuttle per saltare ad un timestep selezionato o per muoversi interattivamente avanti e indietro nel tempo, oltre a consentire la riproduzione con velocità e direzione nel tempo definite dall'utente. Inoltre, tutti i client supportano il rendering di immagini associate ad un singolo timestep della simulazione, così come il rendering di più timestep in viewport parallele, utilizzando la stessa funzione di trasferimento, gli stessi parametri di visualizzazione e le stesse impostazioni di animazione. Quest'ultima opzione è resa possibile dal fatto che manteniamo in cache su GPU le descrizioni di più frame (Sez. 3.5.1). Questa opzione rende semplice il rendering di evoluzioni parallele in viewport separate (si veda la Fig. 3.9 e il video di accompagnamento alla pubblicazione [6]) o la visualizzazione di più campi scalari nella stessa immagine attraverso, ad esempio, lenti virtuali (si veda la Fig. 3.10), ma sono facilmente implementabili anche altre opzioni più avanzate per la presentazione di frame multipli [68, 69]. A differenza dell'approccio di Pulido et al. [58], che ugualmente utilizzava frame multipli visualizzati contemporaneamente, per migliorare la comprensione temporale delle simulazioni, noi non ci limitiamo a frame multipli statici ma, mantenendo costante la differenza di tempo tra i vari frame e condividendo la stessa telecamera interattiva, possiamo eseguirne l'esplorazione spazio-temporale muovendoci anche nel tempo .

3.6.1 Fat client e snapshot renderer

Il fat client (Fig. 3.2 sinistra) inizia le sue operazioni scaricando progressivamente la rappresentazione low-bitrate del dataset esplorato. Il comportamento di default del download progressivo è quello di scaricare prima un timestep ogni otto e, per ogni timestep, i dati a metà risoluzione, portando ad una riduzione di 1/64 dei dati trasferiti. Dopo l'arrivo di questi dati iniziali, l'interazione può iniziare e i dati continuano ad essere scaricati in background, affinandosi prima lungo la direzione temporale, poi nello spazio. L'implementazione di questo metodo progressivo è semplice,

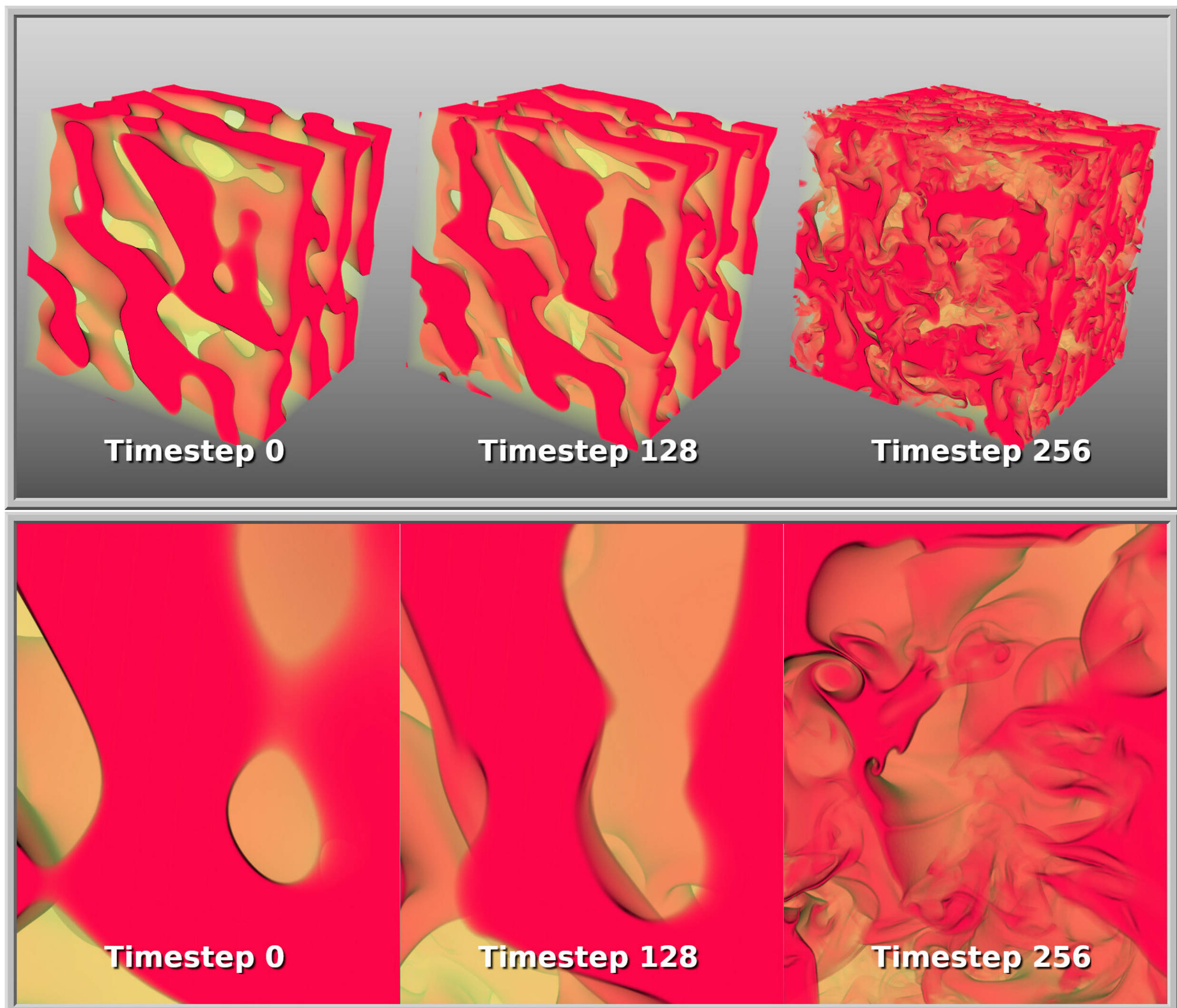

Figura 3.9: ***Rendering di viste temporali multiple.*** *Tre timesteps del dataset HBDT ispezionati in modo sincrono all'interno dello stesso frame.*

poiché i passi di tempo sono memorizzati separatamente e ogni passo di tempo è memorizzato in ordine di risoluzione crescente. Per un dataset 1024×1024^3, abbiamo quindi bisogno di ricevere solo circa 1 GB di dati prima di iniziare ad interagire e 64 GB per ricevere il dataset completo. Su una tipica rete a 1 Gbit/s questo si traduce in una latenza iniziale di pochi secondi, dopo i quali l'utente può iniziare ad eseguire le prime operazioni di osservazione (ad esempio, scorrere l'evoluzione, selezionare le funzioni di trasferimento), in attesa dei dati completi. Durante l'esplorazione interattiva, le operazioni di rendering sono controllate regolando il comportamento del renderer in base allo stato di interazione. Per implementare questo comportamento utilizziamo una semplice macchina a stati. Quando ci si muove con continuità tra i timestep (riproduzione, jog & shuttle), il renderer è configurato in modo da favorire la continuità temporale rispetto alla continuità spaziale. In questo caso l'occlusione inter-frame è disabilitata. Al contrario, quando l'animazione è ferma e l'utente è in movimento, l'occlusione inter-frame è abilitata. Non appena l'utente smette di muoversi o di modificare altri parametri di visualizzazione (inclusa la funzione di trasferimento) per più di un frame, viene inviata una richiesta asincrona al renderer di snapshot (tipicamente residente su

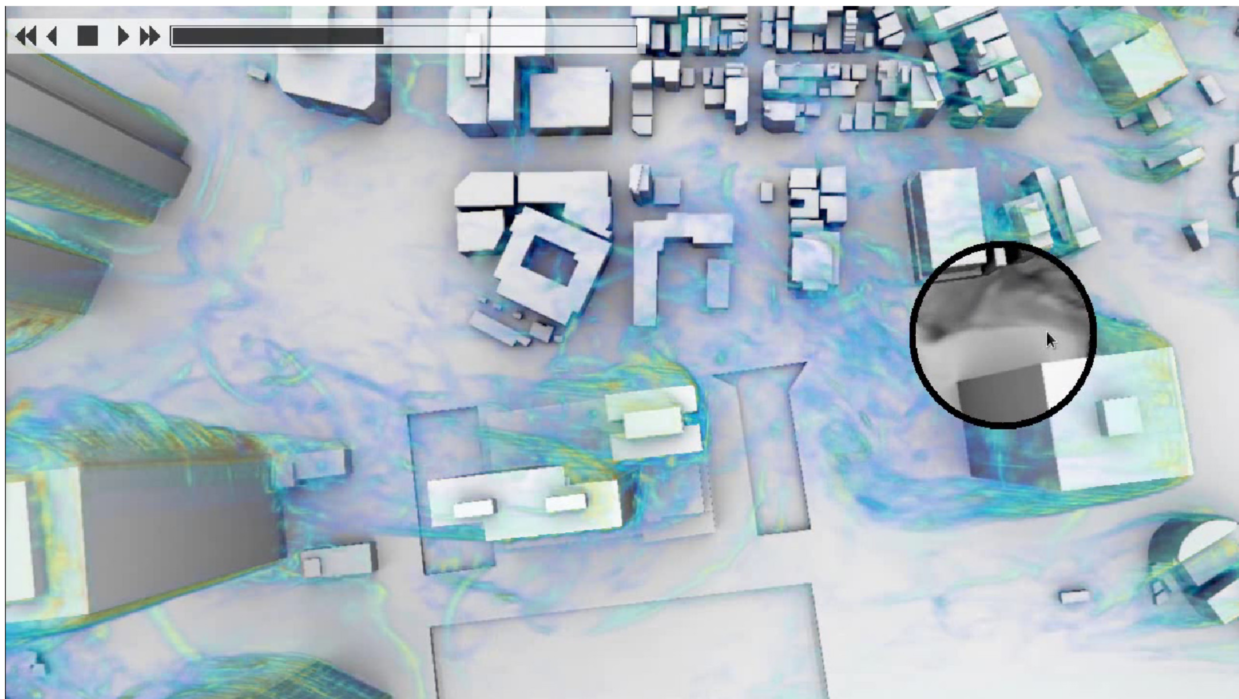

Figura 3.10: ***Rendering di dataset multipli.*** *Due campi scalari diversi di una simulazione CFD urbana sono visualizzati in parallelo utilizzando una lente virtuale. All'esterno della lente, il campo visualizzato è la vorticità del campo di vento, mentre all'interno della lente, che può essere spostata interattivamente, viene visualizzato uno scalare passivo trasportato dal vento.*

un server), che inizia immediatamente a produrre un frame di alta qualità utilizzando l'algoritmo di rendering con i dati near-lossless. Il processo di rendering viene eseguito layer per layer ed è interrompibile. Quando si sposta, il client può annullare immediatamente la richiesta se l'immagine di alta qualità non è ancora arrivata. Per la riduzione della larghezza di banda end-to-end, utilizziamo un encoder CUDA JPEG, che comprime il framebuffer sulla GPU, prima di scaricarlo per la trasmissione in rete [70]. Alla massima qualità, possiamo codificare e trasmettere immagini 4K con meno di 200ms di latenza su una NVIDIA GTX 1080Ti, valore sufficiente per snapshot statici.

3.6.2 Thin client e visualizzatore remoto interattivo

Il design del thin client è pensato per lavorare su piattaforme non in grado di effettuare il rendering volumetrico ad alta frequenza (Fig. 3.2 destra). Il tipico caso d'uso è il rendering su mobile. Per dimostrare la fattibilità dell'approccio, abbiamo implementato un client Android con la stessa interfaccia utente e lo stesso comportamento del fat client. L'unica importante differenza è che i frame ad alta frequenza sono delegati a un renderer remoto, inviati in rete come immagini e visualizzati localmente. Il renderer remoto inizia come una copia semplificata del fat client, senza interfaccia utente e aspettando la connessione del client. Successivamente, utilizzando i parametri di visualizzazione e della funzione di trasferimento comunicati continuamente dal client, prosegue generando immagini, codificandole e inviandole al client per la visualizzazione. Il frame grabber codificatore utilizzato in questo lavoro è lo stesso codec CUDA utilizzato per le immagini di alta qualità, configurato per un alto tasso di compressione (impostazioni utilizzate per questi

benchmark: qualità 75%, nessun sottocampionamento cromatico). Una soluzione alternativa e più promettente, sarebbe quella di utilizzare il decoder hardware H.265 delle schede NVIDIA. Al momento non lo stiamo utilizzando poiché, nella nostra prima implementazione, abbiamo avuto una maggiore latenza sui client Android. Abbiamo intenzione di indagare su questi problemi in futuro. Con il nostro attuale codec, un'immagine full HD è codificata su ˜0.5MB/frame in meno di 8ms, che è largamente compatibile con le velocità interattive su una tipica impostazione di rete. A questa velocità di compressione, 20 fotogrammi/s si traducono in uno stream di 80 Mbit/s, che rientra pienamente nei limiti tipici del wireless, e saranno facilmente utilizzabili anche in ambienti a banda larga, data l'attuale evoluzione verso le reti a 5G, con una velocità media prevista di 490 Mbit/s per le bande a 3,5GHz a livello consumer.

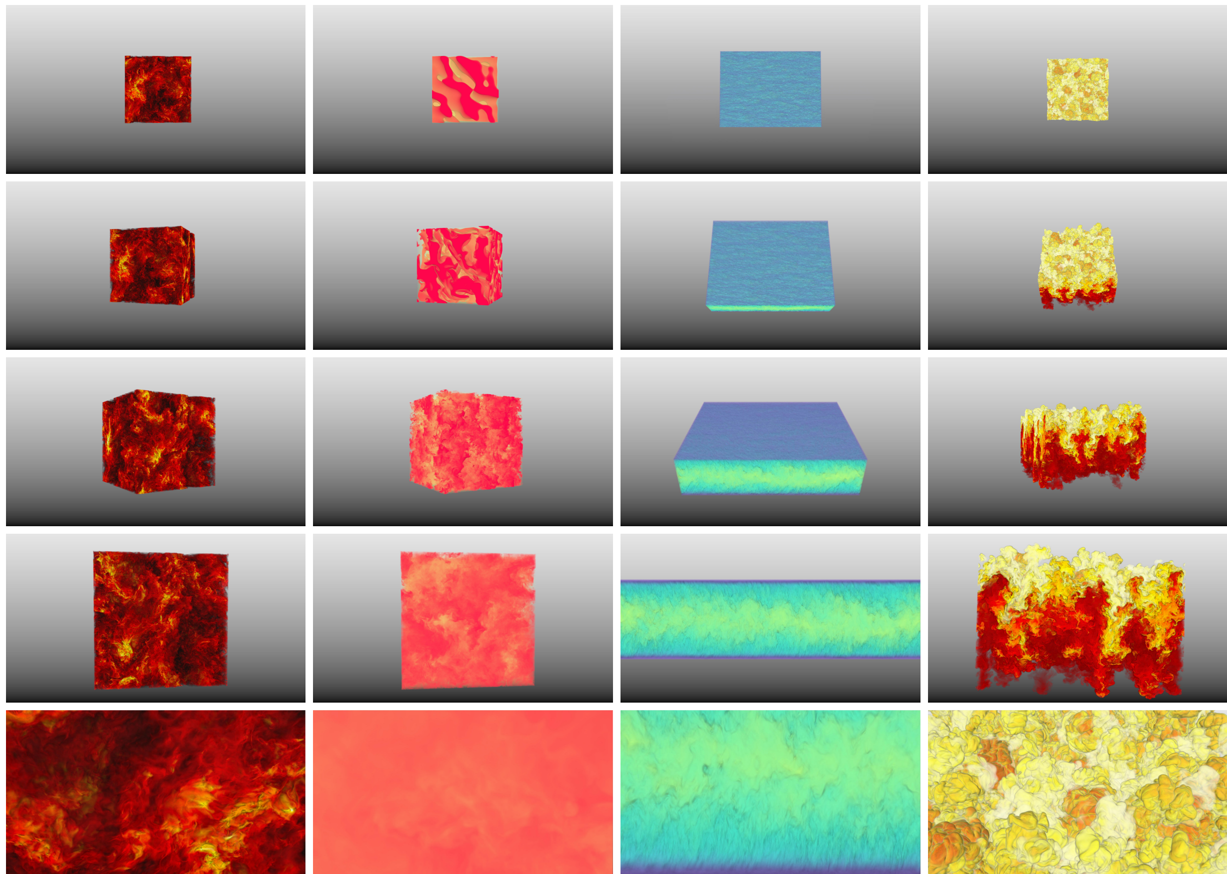

Figura 3.11: ***Sequenze di benchmark.*** *Frame rappresentativi di sequenze di esplorazione interattiva spazio-temporale di enormi dataset dinamici e statici utilizzate per il nostro benchmarking. Da destra verso il sinistra: ISO* (1024 *timestep* 1024^3*), HBDT* (1010 *timestep* 1024^3*), CHAN* (4000 *timestep* $2048 \times 512 \times 1536$*), RT, RT* ($1$ *timestep* 3072^3*)*

3.7 Implementazione e risultati

Un'implementazione della nostra architettura è stata realizzata su Linux, con C++, OpenGL e NVIDIA CUDA 10.0, per l'elaborazione, il rendering lato server e i fat client, e su Android per la prova di fattibilità del thin client. L'abbiamo testata con una varietà di modelli statici e dinamici ad alta risoluzione. In questo capitolo, discutiamo i risultati ottenuti con tre rappresentativi dataset

tempovarianti del database JHU Turbulence [13]: il campo di ampiezza della velocità di una simulazione di turbolenza isotropa forzata (*ISO*, 1024 timestep 1024^3, float, 4TB), il campo di densità di una simulazione di turbolenza convettiva omogenea (*HBDT*, 1010 timestep 1024^3, float, 4TB), e la componente x del campo di velocità di una simulazione di flusso di un canale (*CHAN*, 4000 timestep $2048 \times 512 \times 1536$, float, 24TB). Questi dataset di dominio pubblico sono stati selezionati in quanto rappresentano uno standard de-facto per il benchmarking della visualizzazione tempovariante e permettono il confronto con tecniche alternative allo stato dell'arte. I nostri risultati di compressione sono ottenuti con il codice che sarà rilasciato in open source alla fine del progetto (vedi Sezione 3.9).

Inoltre, per testare le prestazioni di rendering anche su un massiccio dataset statico, presentiamo i risultati di rendering per un singolo frame del campo di densità di una simulazione di instabilità di Rayleigh-Taylor [71] (*RT*, 1 timestep 3072^3, ushort, 54 GB). La Fig. 3.11 presenta diversi frame di sequenze di osservazione di questi dataset. Questi rendering sono ottenuti con la versione interna del codice di visualizzazione.

3.7.1 Prestazioni della compressione

Abbiamo valutato la nostra strategia di compressione low-bitrate eseguendo una batteria di test sui dataset tempo-varianti selezionati, cambiando il desiderato non-zero K a $6, 9, 12, 15$ per ottenere tassi di compressione tra ˜0.25 bps e ˜0.50 bps. In tutti i test abbiamo usato pagine di $P^3 = 4^3$ brick di $B^3 = 4^3$ blocchi di $M^3 = 8^3$ voxel. Abbiamo costruito dizionari di 1024 blocchi prototipo, in 50 K-SVD iterazioni su coresets di 16 Mvoxels. Nonostante il nostro sistema supporti diverse possibili impostazioni, la configurazione di cui sopra è regolata per le codifiche a basso contenuto di bitrate. In particolare, l'utilizzo di impostazioni power-of-two si allinea bene con le nostre griglie multirisoluzione, che dimezzano la risoluzione ad ogni livello più semplificato. In questo contesto, la dimensione del blocco selezionata è la più piccola che può supportare il tasso di compressione richiesto, mentre 1024 prototipi, sufficienti a generare un dizionario completo per la dimensione del blocco selezionato, possono essere indicizzati con 10 bit, lasciando 6 bit per la codifica dei coefficienti. Il numero di iterazioni e la dimensione del coreset sono stati selezionati sulla base di precedenti esperienze con encoder simili [24].

Al fine di fornire un contesto per la valutazione della componente di compressione del nostro lavoro, forniamo anche un confronto con metodi di compressione alternativi, che supportano la decodifica in tempo reale (cioè, oltre 1 GVox/frame a 10 frame/s). In particolare, valutiamo una codifica fixed-rate simile al COVRA [24], la ASTC [33] (versione 10/2017 [72]) e la quantizzazione vettoriale gerarchica (HVQ) [37] (versione utilizzata per la valutazione della performance del sistema COVRA [24]). Includiamo anche, come riferimento, un confronto con metodi che non soddisfano i vincoli di velocità di decodifica, cioè il codec wavelet Cuda Compress (CC) [21] (versione 2013 [73]), lo ZFP [61] (versione 0.5.4 [61]) e lo SZ [74] (versione 2.1.0 [75]). Per ZFP, abbiamo usato la modalità a precisione fissa, che di solito fornisce i migliori rapporti segnale-rumore, e abbiamo variato la tolleranza di errore assoluta (`-a`), per ottenere i bitrate desiderati, per CC abbiamo prefissato il bitrate richiesto, per SZ abbiamo variato l'errore relativo mentre per ASTC abbiamo selezionato la massima qualità e il bitrate più basso ottenibile. Utilizzando impostazioni simili alla valutazione di COVRA [24], HVQ è stato eseguito con un dizionario di 1024 elementi per livello e 12 bit per la quantizzazione della media di blocco.

3.7.1.1 Velocità di compressione

L'elaborazione è stata eseguita su un singolo PC Arch Linux con 256 MB di RAM e due CPU Intel Xeon E5-2650 v4 a 24 core a 2,20 GHz, con dati di ingresso e di uscita memorizzati su una Synology RS3617Xs+, con un RAID 5 composto da dischi SEAGATE ST10kNE004-1ZF101 (file system BTRFS), collegato tramite un link a 10 Gbit/s. La compressione è stata eseguita utilizzando quattro processi paralleli, codificando quattro frame alla volta. L'analisi delle diverse fasi di costruzione mostra che la fase di filtraggio bottom-up (eventualmente compresa la fase di estrazione del coreset), comune a tutti i metodi, è ragionevolmente veloce (20 s/frame per *ISO* e *HBDT*, 32 s/frame per *CHAN*), ed è limitata dal tempo di trasferimento dati dal file server al nodo di elaborazione. I vari codec testati hanno tempi di codifica molto diversi. Il più veloce è il CC con accelerazione hardware, che è in grado di codificare i dati ad una velocità molto elevata (circa 4 s/frame per *ISO* e *HBDT* e 6 s/frame per *CHAN*), seguito da ZFP (15-29 s/frame per *ISO* e *HBDT*, 23-44 per *CHAN*, tempi più bassi per velocità di compressione più elevate). I codec più lenti sono ASTC (oltre 12 min/frame per *ISO*, 15 min/frame per *HBDT*, 17 min/frame per *CHAN*) e SZ (13 min/frame per *ISO* e *HBDT*, 21 min/frame per *CHAN*). I tempi di compressione per i metodi di sparse-coding si collocano a metà strada tra questi estremi. Il tempo di apprendimento del dizionario è indipendente dalla dimensione del dataset e dipende solo leggermente dalla sparsità (42-74s per $K = 6..15$). Il tempo di codifica è lineare con la dimensione del dataset e cresce in modo sub-lineare con la sparsità voluta. La codifica elastica è, in media, da 1,5x a 2x più lenta della codifica a dimensione fissa (24s-33 s/frame per la codifica a dimensione fissa contro 42s-74s per l'ottimizzazione elastica di *ISO* e *HBDT*, 35s-49s vs. 56s-74s per *CHAN*, tempi inferiori per tassi di compressione più elevati). Anche se non consideriamo la velocità di codifica un dato essenziale per questa applicazione asimmetrica, dove abbiamo bisogno di codificare i dati una volta sola per simulazione, prevediamo di migliorare la velocità di compressione sui dati tempo-varianti riutilizzando i dizionari per i frame vicini ed eseguendo solo 1-2 operazioni di affinamento su questi dizionari già addestrati, poiché ci aspettiamo che i dizionari dei frame vicini siano simili. Dato il tempo dedicato all'apprendimento dei dizionari nella configurazione utilizzata, ci aspettiamo per questa strategia di riutilizzo una velocità di almeno 2x.

	Real-Time								**Non-Real-Time**							
	Elastic		**Fixed**		**ASTC**		**HVQ**		**Unc.**		**CC**		**ZFP**		**SZ**	
bps	D	H	D	H	D	H	D	H	D	H	D	H	D	H	D	H
~0.26	33.33	29.41	41.67	35.71	–	–	–	–	–	–	4.18	4.11	0.23	0.14	0.01	0.01
~0.35	30.30	26.32	34.48	29.41	–	–	–	–	–	–	4.05	3.97	0.22	0.14	0.01	0.01
~0.45	27.03	23.26	31.25	26.32	–	–	–	–	–	–	4.01	3.92	0.20	0.13	0.01	0.01
~0.54	24.39	20.83	28.57	23.81	–	125.00	34.48	28.57	–	–	3.92	3.82	0.18	0.12	0.01	0.01
32	–	–	–	–	–	–	–	–	–	0.40	–	–	–	–	–	–

Tabella 3.1: ***Prestazioni di decodifica.*** *Velocità di caricamento e decodifica in GVox/s per i vari codec. I codec in tempo reale devono supportare oltre 1 GVox/frame a 10 frames/s. La colonna D riporta la velocità di decodifica pura per i dati nella memoria del dispositivo, per i codec GPU, e nella RAM, per quelli CPU, mentre la colonna H riporta la velocità di spostamento dei dati dall'host al dispositivo e della loro decompressione.*

3.7.1.2 Velocità di decompressione

Il supporto di almeno 10 frame/s di animazione richiede un codec in grado di caricare i dati sulla GPU, decomprimerli e visualizzarli a una velocità di almeno 10 Gvox/s. La tabella 3.1 riassume

le prestazioni di caricamento e decompressione, ottenute su un singolo PC con CPU 9-7900X 3,30 GHz e una GeForce GTX 1080Ti, per i vari codec. Dalla tabella si evince chiaramente che, nonostante l'aumento della larghezza di banda, il formato non compresso non è ancora utilizzabile (meno di 2,5 GVox/s dalla RAM alla memoria texture). Il codec più performante in tempo reale è ASTC, grazie al supporto hardware diretto. ASTC, tuttavia, ha dei limiti in termini di compressione e qualità realizzabili (vedi sotto). Gli approcci sparse-coding e HVQ soddisfano chiaramente anche il vincolo della velocità di decodifica. È interessante notare che la sparse-coding elastica non sta introducendo un overhead di decodifica significativo, rispetto alla codifica a taglia fissa, poiché il kernel di decodifica è attentamente progettato per la velocità, piuttosto che per la massima compressione ottenibile. I conflitti di memoria sono evitati attraverso operazioni di lettura/scrittura allineate e, poiché tutti i voxel di un blocco sono decodificati in parallelo e hanno lo stesso valore non-zero, la divergenza è minima. Il codec non real-time più veloce è di gran lunga il codec wavelet CC accelerato da CUDA, che, comunque, resta al di sotto della soglia di decodifica real-time richiesta, poiché, per raggiungere la sua eccellente qualità di ricostruzione a bassi bitrate, include diverse operazioni costose, come la decodifica RLE inversa e la decodifica Huffman, oltre alla trasformazione wavelet inversa. Inoltre, per coerenza con il documento originale [21], riportiamo qui per CC la velocità ottenuta per una granularità di 256^3 voxel/brick, che produce la massima velocità raggiungibile, mentre gli altri codec in tempo reale sono stati configurati ad una granularità molto più fine di 32^3, al fine di supportare un culling più efficiente. I codec della CPU, ZFP e ancor più SZ, non possono attualmente essere utilizzati in un contesto di banda larga, sia a causa della loro limitata velocità, sia perché i dati devono viaggiare in formato decompresso dalla RAM alla GPU. ZFP ha introdotto molto recentemente un decoder con accelerazione GPU (versione 0.5.4 [61]), che però supporta solo la codifica a taglia fissa, con qualità ridotta rispetto alla versione a tolleranza fissa.

	Real-Time								**Non-Real-Time**					
	Elastic		**Fixed**		**ASTC**		**HVQ**		**CC**		**ZFP**		**SZ**	
	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR	bps	PSNR
ISO	0.26	43.84	0.24	42.88	–	–	–	–	0.26	47.70	0.29	29.89	0.26	35.54
	0.35	45.80	0.34	44.86	–	–	–	–	0.35	49.71	0.34	33.07	0.34	35.73
	0.45	47.18	0.43	46.26	–	–	–	–	0.45	51.51	0.41	36.60	0.46	35.99
	0.54	48.22	0.52	47.34	0.59	45.85	0.50	41.47	0.54	52.89	0.53	40.52	0.55	36.28
HBDT	0.26	37.90	0.24	35.43	–	–	–	–	0.24	41.90	0.30	3.94	0.29	45.41
	0.35	39.61	0.34	37.40	–	–	–	–	0.32	44.64	0.36	13.54	0.33	45.56
	0.45	40.95	0.43	38.92	–	–	–	–	–	–	0.44	22.27	0.45	46.20
	0.54	41.66	0.52	39.91	0.59	38.02	0.50	23.28	–	–	0.58	30.44	0.55	46.88
CHAN	0.26	46.47	0.24	45.09	–	–	–	–	0.26	49.88	0.27	23.82	0.25	36.87
	0.35	48.43	0.35	47.10	–	–	–	–	0.35	51.98	0.34	33.21	0.33	37.07
	0.45	49.82	0.43	48.55	–	–	–	–	0.44	53.94	0.49	40.77	0.43	37.33
	0.54	50.80	0.52	49.64	0.59	23.17	0.50	41.70	0.52	55.47	0.63	44.81	0.51	37.65

Tabella 3.2: ***Tasso di compressione e distorsione.*** *I metodi considerati real-time sono quelli in grado di sostenere una velocità di decompressione superiore a 1 GVox/frame a 10 frames/s: sparse-coding elastica e sparse-coding fissa con sparsità variabile (K=6,9,12,15), ASTC ad altissima qualità e massima compressione, HVQ con dizionario fisso di 1K. Per riferimento, includiamo anche i risultati ottenuti con metodi non in tempo reale: CC, ZFP con precisione variabile (-a) e SZ con errore relativo variabile (RE). Le celle vuote corrispondono al tasso di compressione non ottenibile con il metodo dato.*

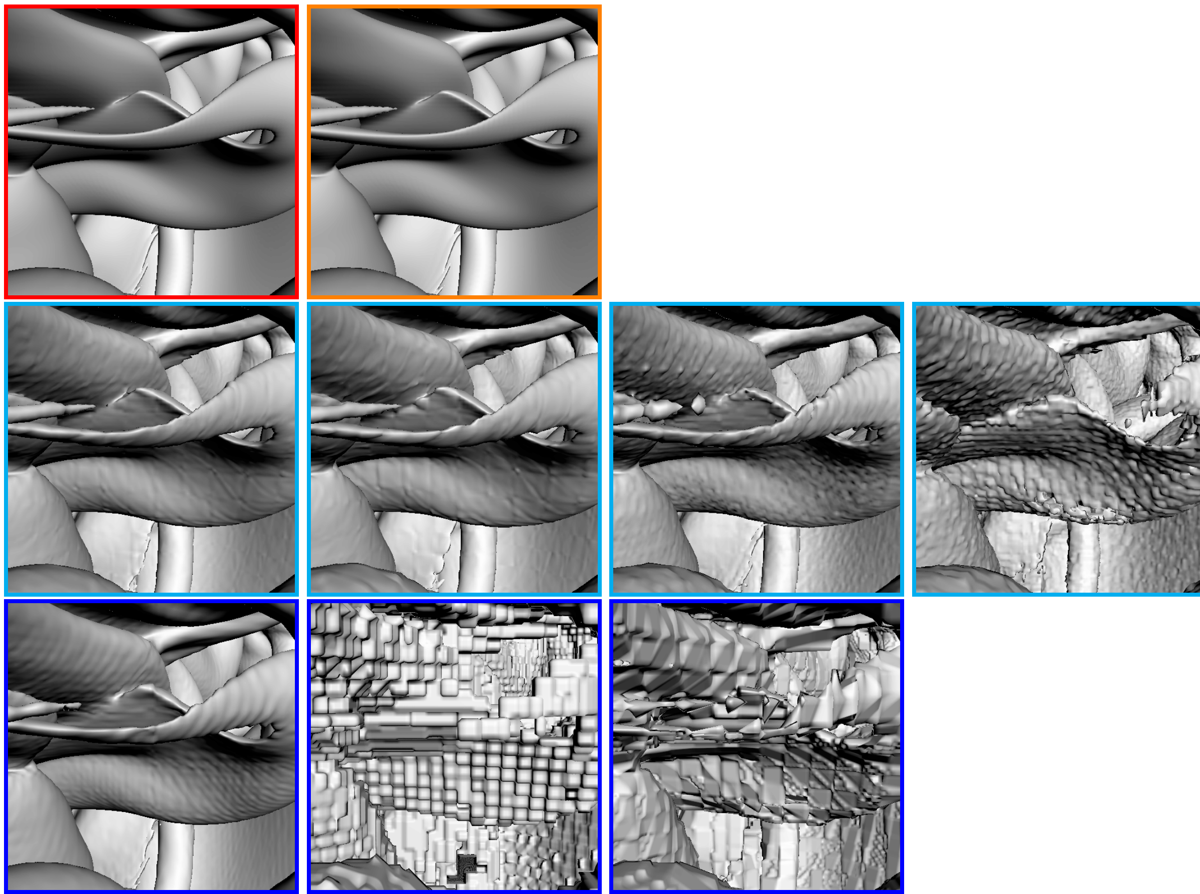

Figura 3.12: ***Qualità del rendering d'isosuperficie HBDT.*** *Da sinistra a destra e dall'alto verso il basso: non compresso, ZFP near-lossless (7,4 bps, volume PSNR=92,06, SSIM=0,999), sparse-coding elastica (0,45 bps, SSIM=0,856), sparse-coding fissa (0.43 bps, SSIM=0.809), ASTC (0.59 bps, SSIM=0.697), HVQ (0.50 bps, SSIM=0.351), CC (0,45 bps, 0,833), ZFP (0,41 bps, SSIM=0,206), SZ (0,46 bps, SSIM=0,347). I codecs che supportano la decodifica in tempo reale sono contrassegnati in ciano, mentre gli altri sono contrassegnati in blu.*

3.7.1.3 Tasso di compressione e distorsione

La tabella 3.2 riassume le prestazioni di compressione low-bitrate sullo stsso timestep selezionato di ogni dataset tempo-variante (timestep 256, per tutti i dataset). Altri timestep forniscono risultati coerenti. Le caselle vuote equivalgono al tasso di compressione non ottenibile con il metodo dato. Per consentire il confronto con altri metodi a risoluzione singola, riportiamo i risultati ottenuti solo per la griglia a risoluzione completa, a livello di foglia. La velocità di compressione è misurata in bits-per-sample (bps), mentre la qualità è misurata con il rapporto tra segnale di picco e rumore (PSNR), definito come $10 \log_{10} \frac{(\max_i x_i - \min_i x_i)^2}{\frac{1}{N}\sum_i (x_i - y_i)^2}$, dove x_i è l'originale valore del voxel, y_i quello approssimato, N il numero totale di voxel. La Fig. 3.12 mostra l'effetto della compressione sulla qualità dell'immagine, per un dettaglio di rendering di una isosuperfice del dataset HBDT e include anche l'indice di similarità strutturale (SSIM) [76] di ogni immagine, calcolata con dati compressi rispetto a riscontri oggettivi. Sulla base della valutazione delle prestazioni di decodifica, i metodi considerati in real-time sono la sparse-coding elastica e la sparse-coding fissa con sparsità variabile (K=6,9,12,15), ASTC ad altissima qualità e massima compressione e HVQ con dizionario fisso 1K. La scalabilità del nostro metodo è dimostrata dal fatto che, scegliendo opportunamente la sparsità

di destinazione, tanto la codifica fissa quanto quella elastica possono abbracciare una buona gamma sia di velocità di compressione che di qualità. La codifica elastica dimostra di essere in grado di migliorare significativamente la qualità per tutti i dataset, nell'intervallo di compressione testato, in quanto aumenta il PSNR da +1dB a +2dB rispetto alla versione fissa e di diversi dB rispetto ai metodi concorrenti in tempo reale. La migliore qualità di compressione del nostro metodo rispetto ad altri codec in tempo reale si traduce anche in un significativo miglioramento della qualità dell'immagine a bitrate comparabili (vedi Fig. 3.12). Per riferimento, abbiamo incluso anche i risultati ottenuti con metodi non in tempo reale: CC, ZFP con accuratezza variabile (-a) e SZ con errore relativo variabile (RE). CC risulta essere il codec più performante a questi bassi bitrate, con una performance leggermente inferiore in termini di SSIM e superiore in termini di PSNR, rispetto alla sparse-coding elastica, grazie ai metodi di trasformazione e di entropy-coding implementati, non inclusi nelle tecniche sparse-coding, a supporto della decodifica in real-time. ZFP e SZ, anche se non funzionano bene a un bitrate così basso, sono più scalabili di CC, che, per progettazione, non è applicabile in modalità near-lossless, quando molti coefficienti wavelet sono diversi da zero e la loro quantizzazione porta a tabelle Huffman grandi e con molti simboli diversi. Le due caselle vuote della tabella corrispondono ai casi in cui il CC non può raggiungere il bit rate desiderato, per l'overflow della tabella Huffman. Per questo motivo, attualmente utilizziamo ZFP per frame near-lossless (vedi Diffenderfer et al. [77] per uno studio del comportamento di ZFP a bitrate da moderato ad alto). La Fig. 3.12 include anche l'immagine generata da ZFP a 7.4bps (PSNR volumetrico=92.06).

3.7.2 Prestazioni di rendering

La prestazione del nostro sistema di rendering, implementato come fat client, è stata valutata su un singolo PC Arch Linux con 128 GB di RAM, una CPU 24-core i9-7900X 3,30 GHz, una GeForce GTX 1080Ti e un SSD locale Samsung 9160 Pro 1TB per lo storage. Il server è la stessa macchina utilizzata per l'elaborazione, connessa su LAN locale 10 Gbit/s. La prestazione della nostra implementazione proof-of-concept del thin client è stata valutata, invece, su un tablet Samsung Galaxy Note Pro SM-P905 Android 5.0, con un Chipset Qualcomm Snapdragon 800 (Quad-core 2,3 GHz Krait 400 CPU e GPU Adreno 330) connesso a 144 Mbit/s, su una LAN wireless moderatamente caricata, allo stesso server utilizzato per il fat client. Il renderer remoto e il renderer di singoli fotogrammi ad alta qualità sono stati eseguiti sullo stesso nodo con tutte le risorse condivise. I risultati quantitativi, qui presentati in dettaglio, sono stati collezionati su percorsi interattivi preregistrati, progettati per essere rappresentativi dei tipici compiti di ispezione volumetrica e con attività di forte stress per il sistema, incluse rotazioni e rapidi cambiamenti dalla visione d'insieme a primi piani estremi e viceversa (vedi anche Fig. 3.11). Ogni percorso registrato è stato riprodotto con diverse impostazioni su una finestra di dimensioni di 1920×1080 pixel. Per stressare il sistema, la risoluzione del renderer è stata sempre impostata alla massima precisione (1 voxel/pixel). Abbiamo misurato frame rate effettivi, cioè non limitati ai tempi netti di rendering, ma comprensivi dei tempi tra frame e frame. Le prestazioni qualitative del nostro sistema sono illustrate anche in un video di accompagnamento alla pubblicazione [6], che mostra registrazioni dal vivo delle sequenze analizzate, così come di sequenze interattive similari, con tutti i dataset.

3.7.2.1 Latenza di avvio

Per questi test, abbiamo utilizzato l'impostazione 0.45 bps elastico, che porta a creare rappresentazioni a basso contenuto di bitrate di 65GB (ISO), 64GB (HBDT), 376GB (CHAN) e 1,8GB (RT). Al momento dell'avvio, il sistema esegue il download progressivo dalla memoria remota, salvando la rappresentazione su SSD locale per un ulteriore accesso. Dati sufficienti per l'esplorazione iniziale vengono ricevuti dal server dopo 2,5s per ISO e HBDT, 3,0s per CHAN, e meno di un secondo per la RT a frame singolo. In questo modo è garantita un'interazione quasi immediata. I client riceveranno l'intera quantità di dati in meno di due minuti per ISO e HBDT, circa due minuti e mezzo per CHAN, e meno di tre secondi per la RT. Questo rende possibile mantenere i dati su un server e scaricarli su richiesta su un veloce SSD locale solo all'inizio di una sessione di osservazione. L'utilizzo di dati non compressi richiederebbe invece ore.

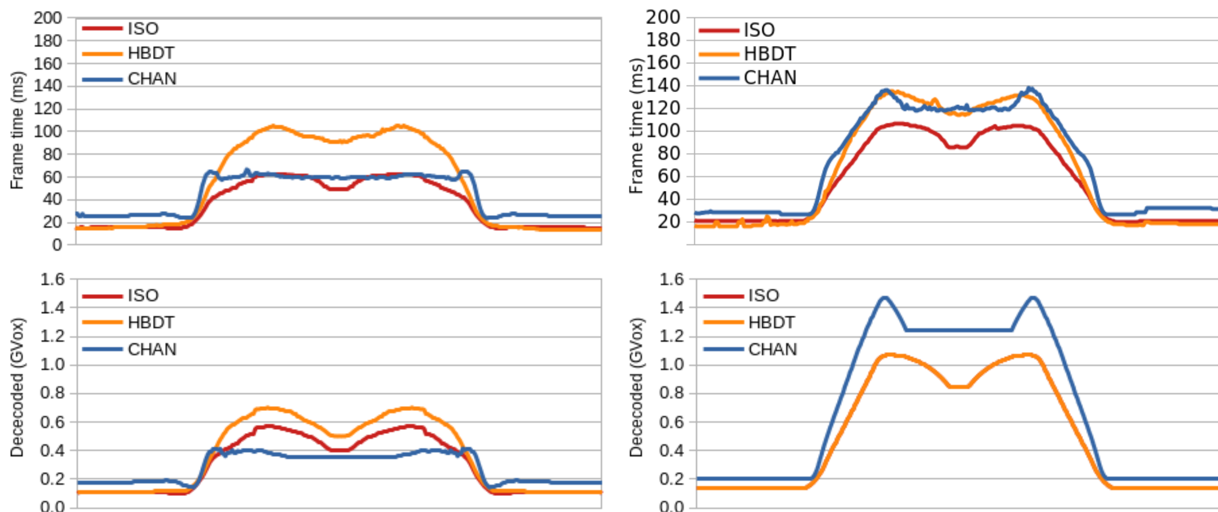

Figura 3.13: ***Esplorazione di un dataset dinamico.*** *In alto a sinistra: tempo di frame con occlusion culling intra-frame. In alto a destra: tempo di frame senza occlusion culling intra-frame. In basso a sinistra: voxel decodificati per frame con occlusion culling intra-frame. In basso a destra: voxel decodificati per frame senza occlusion culling intra-frame.*

3.7.2.2 Esplorazione dinamica di un dataset su un fat client

Quando ci si muove con continuità tra i timestep (riproduzione, jog&shuttle), il renderer è configurato per privilegiare la continuità temporale rispetto alla continuità spaziale. In questo caso, l'occlusione intra-frame conservativa è abilitata, mentre è disabilitata quella inter-frame non conservativa, per evitare effetti dinamici spuri. La Fig. 3.13 riporta i tempi di rendering e la quantità di voxel decodificati nei percorsi spazio-temporali del video di accompagnamento alla pubblicazione [6], per i tre dataset dinamici. I frame rappresentativi sono in Fig. 3.11. In questi percorsi spazio-temporali, la simulazione viene riprodotta prima avanzando e poi retrocedendo nel tempo, mentre la telecamera è in movimento. Come riportato, il frame rate è interattivo sia con che senza occlusion culling, dato che il tempo medio di rendering con l'occlusion culling abilitato è di 36 ms/frame per ISO, 54 ms/frame per HBDT e 44 ms/frame per CHAN, mentre il tempo medio di frame senza occlusion culling è di 56 ms/frame per ISO, 66 ms/frame per HBDT e 73 ms/frame per CHAN. Il picco del tempo di rendering con l'occlusion culling abilitato lo si ha per l'HBDT

altamente trasparente (105 ms/frame), mentre con l'occlusion culling disabilitato lo si ha per il più grande, ma più opaco, CHAN (138 ms/frame). L'occlusione intra-frame è efficace nel miglioramento delle prestazioni, almeno per funzioni di trasferimento moderatamente opache, riducendo il numero di brick decodificati per effetto della terminazione anticipata dei raggi. Il numero totale di brick decodificati è ridotto del 70% per l'ISO, oltre il 148% per la più opaca CHAN, e il 41% per il piu' trasparente HBDT. Questo porta a velocizzare il rendering del 55% per l'ISO, dell'88% per CHAN e del 22% per HBDT. La velocità di decodifica è sempre mantenuta a circa 19 GVox/s per tutti i dataset, mentre il riempimento dell'apron è stato misurato intorno ai 18 GVox/s. Poiché la memorizzazione di brick non sovrapposti riduce l'ingombro dello storage, così come le larghezze di banda server-client e client-GPU del 42%, l'utilizzo dell'approccio di riempimento dell'apron si dimostra efficace. Per il benchmarking completo delle capacità di accesso diretto, durante la riproduzione della stessa animazione, abbiamo anche misurato la prestazione ottenuta con time-step presi casualmente, che è risultata compresa fra 18ms-122ms (media 42ms) per tutti i frame e tutti i dataset. Questo è appena poco al di sotto della massima velocità di rendering raggiunta, con una diminuzione media delle prestazioni dell'11% (dell'81% nel peggiore dei casi), dovuta principalmente alla ridotta efficienza nell'accesso al file system.

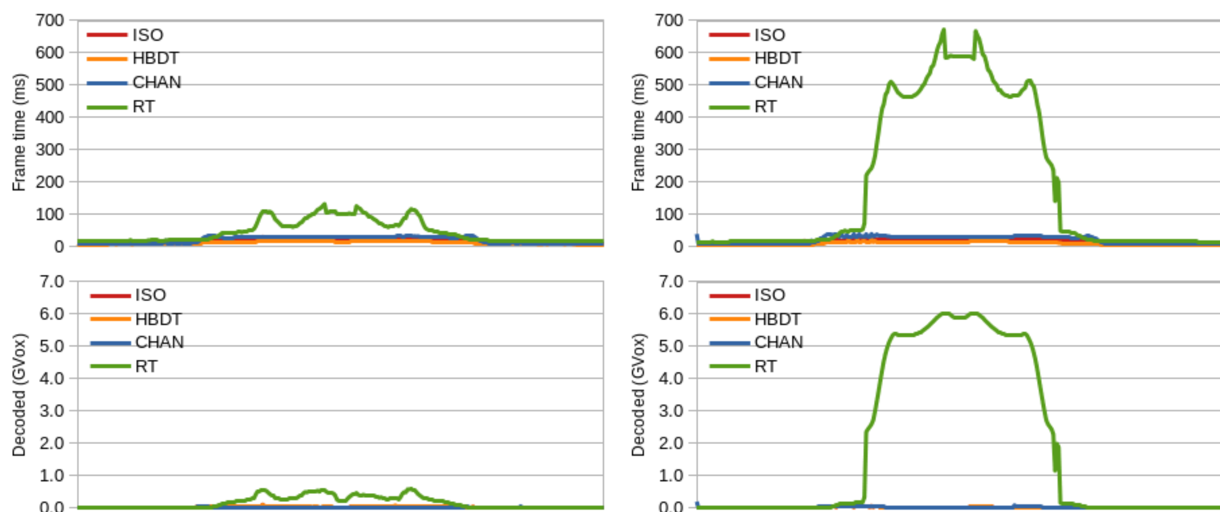

Figura 3.14: ***Esplorazione dataset statico.*** *In alto a sinistra: tempo di frame con occlusion culling. In alto a destra: tempo di frame senza occlusion culling. In basso a sinistra: voxel decodificati per frame con occlusion culling. In basso a destra: voxel decodificati per frame senza occlusion culling.*

3.7.2.3 Esplorazione statica del timestep su fat client

Quando l'animazione è ferma e l'utente è in movimento, viene abilitata l'occlusione inter-frame, al fine di aumentare le prestazioni dell'esplorazione spaziale, anche se gli aggiornamenti incrementali possono introdurre effetti dinamici. In queste condizioni, la cache si dimostra efficace, e il numero di voxel/frame decodificati crolla a causa dello streaming ray-guided. La Fig. 3.14 riassume i valori delle prestazioni misurate durante l'esplorazione spaziale del frame utilizzato per la valutazione della compressione nella Sez. 3.7.1. Quando la dimensione della cache di brick decompressi è abbastanza grande da contenere l'intero workingset, solo pochi brick/frame vengono decompressi per cache fault, e la prestazione è simile ai precedenti raycaster GPU single-pass [28, 24]. Per i

tre dataset dinamici, utilizzando gli stessi percorsi testati per il benchmark dinamico, il frame rate medio è di 54 frames/s per ISO, 78 frames/s per HBDT e 45 frames/s per CHAN. Il frame rate non scende mai al di sotto delle prestazioni interattive, poiché la frequenza di aggiornamento più lenta misurata è di 27,1 frames/s per CHAN. Il numero medio di voxel decodificati per frame non supera mai i 106 Mvoxel, poiché gran parte del working set richiesto per una data immagine è già presente in cache prima del rendering. Per questi dataset, inoltre, l'occlusion culling è meno efficace che nel caso dinamico, dato che lo sforzo di caricamento e decodifica per frame è già basso, per effetto della cache. Di conseguenza, il frame rate con e senza occlusion culling è quasi lo stesso. Invece, quando la dimensione del dataset è maggiore della dimensione della cache di brick decompressi, come per la RT, la situazione cambia. In questo caso si passa all'approccio multipass basato su layer e si abilita l'occlusion culling intra-frame, in aggiunta a quella inter-frame. Alla tolleranza 1, raggiungiamo per questo dataset un frame rate minimo di 7.5 frames/s (media per il percorso 22 frames/s), con un picco di 599 Mvoxels/frame decodificati. Al contrario, senza occlusion culling, le prestazioni scendono ad un frame rate minimo di 1.5 frames/s, con un picco di voxel caricati e decodificati di oltre 6 GVoxels/frame.

3.7.2.4 Immagine fissa di alta qualità su fat client

Quando l'utente smette di modificare la vista per più di un frame, il client richiede al renderer remoto di produrre uno snapshot di alta qualità, applicando il nostro raycaster out-of-core ai dati near-lossless. Utilizzando una tolleranza di 1 voxel/pixel, la latenza misurata per ricevere l'immagine prodotta dai dati near-lossless è di 0,5-1,5s per i dataset dinamici e di 0,8-7s (la tempistica dipende dalle viste) per il dataset RT statico, molto più grande. Quasi tutto il tempo è dovuto al caricamento e alla decodifica dei dati dallo storage (recupero dei dati dal file server piuttosto che da SSD locali), decodifica ZFP e caricamento dei brick. Anche se questi aspetti possono essere ottimizzati (ad esempio, utilizzando un decodificatore CUDA ZFP o un file server più veloce), la latenza è già accettabile per un'applicazione interattiva, almeno per i nostri dati tempo-varianti.

3.7.2.5 Prestazioni del thin client

Poiché il rendering per un thin client viene eseguito in remoto, la differenza di prestazioni è dovuta principalmente all'aumento della latenza e alla riduzione del frame rate collegato alla comunicazione. Abbiamo valutato la nostra implementazione di proof-of-concept eseguendo nuovamente i test dinamici sul client Android. Abbiamo scoperto che, sulla nostra attuale implementazione, la velocità del renderer remoto rimane, entro una piccola variazione percentuale, esattamente la stessa di quella del fat client (Fig. 3.13), poiché l'unico overhead è costituito dalla codifica su GPU delle immagini generate e dalla trasmissione socket. Sulla nostra implementazione, il fattore limitante, a 1080p, è il client Android proof-of-concept. Un tempo costante di ˜30 ms/frame viene speso nel funzionamento della rete e nel refresh OpenGL, mentre l'operazione più impegnativa è la decodifica software delle immagini ricevute, che richiede ˜98 ms/frame. Di conseguenza la velocità di refresh è limitata a ˜8 frame/s su tutti i dataset. L'abbassamento della risoluzione a 720p, riduce il tempo di decodifica a ˜46 ms/frame innalzando il massimo frame rate del client a ˜13 frame/s. Ci aspettiamo di rimuovere questa limitazione utilizzando un decoder ottimizzato su Android, ma questo problema di implementazione è ortogonale al nostro lavoro.

3.8 Simulazione urbana

Il nostro attuale lavoro mira a utilizzare l'approccio descritto per l'esplorazione dei dati di simulazione, specialmente nell'area della simulazione CFD su larga scala. La Fig. 3.15 e il video di accompagnamento alla pubblicazione [6] illustrano i nostri risultati preliminari, che mostrano la possibilità di esplorare in tempo reale il campo di vorticosità intorno agli edifici a scala urbana. L'uso dei dati qui utilizzati è gentilmente concesso dall'Architectural Institute of Japan.

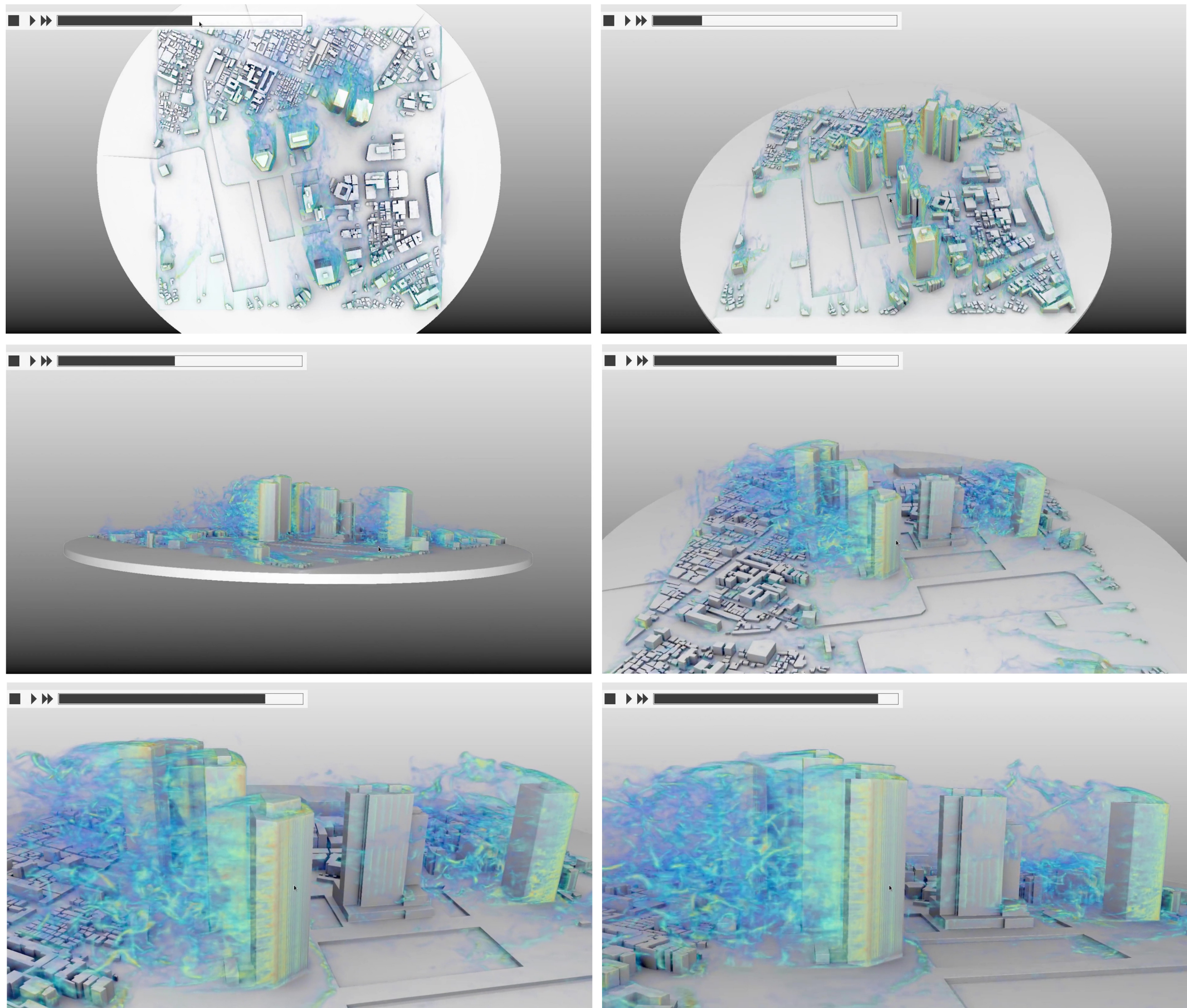

Figura 3.15: Esempio di applicazione ad una simulazione CFD urbana. *Frames da uns sequenza interattiva di esplorazione spazio-temporale del campo di vorticosità intorno agli edifici a scala urbana (Shinjuku, Giappone).*

3.9 Pianificazione della distribuzione open source del codec

Il software che sarà rilasciato in open source nel quadro del progetto TDM è il codice di compressione/decompressione di volumi. Il codice è composto da una software library che contiene i metodi di compressione implementati e da un'applicazione per dimostrare l'utilizzo di questi metodi. Il rilascio è previsto congiuntamente al prossimo deliverable di progetto di OR6. In questa

sezione descriviamo le caratteristiche del sistema che sarà rilasciato, che è in fase di testing interno ed è stato utilizzato per tutti i risultati presentati in questo capitolo

3.9.1 Dipendenze e librerie di base

Il codice che sarà rilasciato è previsto compilabile in c++ su piattaforma linux utilizzando gcc con OpenMP. Le dipendenze esterne sono:

- *sl*: libreria di template C++ di supporto all'accesso ai file e al calcolo matematico e geometrico, sviluppata da CRS4. Disponibile su `http://vic.crs4.it/vic/download/`. Licenza: Free for non commercial use.
- *Eigen*: libraria di template C++ per algebra lineare. Disponibile su `http://eigen.tuxfamily.org/index.php?title=Main_Page`. Licenza: Mozilla Public License 2.0. Free/Copyleft.
- *Qt*: libreria e tool per lo sviluppo multi-piattaforma di applicazioni grafiche. Disponibile su `https://www.qt.io/`. Licenza: GPL.

3.9.2 Struttura della libreria

La libreria che sarà distribuita verrà suddivisa in due semplici sottosistemi:

- *vol*: classi per l'accesso ai volumi scalari in formato non compresso. In particolare, le implementazioni principali sono nei seguenti file:
  - *float_xarray.hpp*: classe per l'accesso out-of-core in lettura e scrittura di array di numeri reali. Le codifiche supportate sono i float 32 bit e i numeri reali normalizzati nell'intervallo 0..1 e quantizzati uniformemente a 8 o 16bit.
  - *raw_volume.hpp*: classe per l'accesso out-of-core in lettura e scrittura di volumi di scalari, secondo il formato definito alla Sezione 3.9.3. La classe permette sia l'accesso ai singoli voxel che l'accesso per blocchi, che vengono restituiti linearizzati in un singolo vettore.
- *sparse_coding*: classi per la compressione/decompressione attraverso metodi di codifica sparsa. Queste classi non utilizzano il concetto di volume, ma immaginano che il dato da comprimere sia uno stream di vettori. Le applicazioni che utilizzano i volumi linearizzano blocchi di voxel in singoli vettori, utilizzando ad esempio l'interfaccia fornita in *raw_volume.hpp*.
  - *signal_stream.hpp*: classi che forniscono un'interfaccia astratta ad uno stream di vettori.
  - *streaming_coreset_builder.hpp*: classi per l'estrazione di un coreset da uno o più stream di vettori. Le varie sottoclassi implementano diverse strategie.
  - *raw_volume.hpp*: classe per l'accesso out-of-core in lettura e scrittura di volumi di scalari, secondo il formato definito alla Sezione 3.9.3.
  - *matching_pursuit.hpp*: funzioni per la codifica sparsa dato un dizionario. Le funzioni si applicano sia a singoli vettori che a gruppi di vettori, nel secondo caso utilizzando ottimizzazioni batch. In particolare, viene fornita una versione ottimizzata di orthogo-

nal matching pursuit [64]. Le implementazioni sono rientranti per supportare l'utilizzo paralelo.

- *incremental_sparse_coder*.hpp*: classi per l'applicazione incrementali di algoritmi di sparse coding. Le classi permettono il calcolo rapido di soluzioni a vari livelli di sparsità e sono ottimizzate per il loro utilizzo incrementale, partendo dalla soluzione di base a zero coefficienti e incrementando un coefficiente alla volta.
- *dictionary_coding_budget_optimizer*.hpp*: classi per l'implementazione dell'elastic sparse coding, per la codifica sparsa con numero variabile di coefficienti di un set di vettori, all'interno di pagine a taglia fissa. La classe *dictionary_coding_budget_optimizer_greedy_grow* implementa in particolare l'algoritmo di allocazione ottimale descritto nell'Algoritmo 1.
- *dictionary_coder_*.hpp*: classi per l'apprendimento di dizionari ottimali per la codifica sparsa di stream di vettori, tramite l'utilizzo di coreset. In partiolare la classe *dictionary_coder_ksvd* implementa la variante ponderata dell'algoritmo K-SVD descritto alla Sezione 3.4.2.1.

3.9.3 Formato file

Il formato file adottato per la rappresentazione di volumi associa un file header (estensione `.hdr`), che descrive il formato della griglia, ad un file di dati (estensione `.raw`), che enumera i dati della griglia rettilinea nell'ordine z, y, x. Il file header ha il seguente formato:

```
raw_volume_header
sample_counts <nx> <ny> <nz>
sample_spacing <sx> <sy> <sz>
bps <bps>
```

dove *sample_counts* indica il numero di voxel che compongono il volume in ognuna delle direzioni spaziali, *sample_spacing* indica la dimensione di un voxel in ognuna delle direzione spaziali e *bps* vale 32 per un coding in FP32 (IEEE 754 32-bit base-2 floating-point), 16 per un coding su due byte a quantizzazione uniforme e 8 per un coding su singolo byte a quantizzazione uniforme.

3.9.4 Applicazione dimostrativa

L'applicazione dimostrativa che sarà rilasciata ha lo scopo di illustrare l'utilizzo della libreria per encoding e decoding di volumi scalari. L'applicazione è lanciata da linea di comando con la seguente sintassi:

```
Usage:
vic_test_volume_compression [options] input-file
General options:
  -o output file            (output file for decoded volume)
  -d delta  file            (output file for difference volume)
  -b block_size             (default 6)
  -c coreset method         (covra|reservoir|uniform) - default reservoir
  -C Mvalues                (max coreset size - default 512^3)
  -m training method        (hvq|ksvd - default ksvd)
  -D dict size              (default 1024)
```

```
  -K sparsity               (default 6)
  -T tolerance              (k variable if non null otherwise fixed) - default 0
  -I training epochs        (default 100)
  -z 0/1                    (1: encode average separately; 0: no -- default 1)
  -e encoding method        (fixed|dpmck|greedygrow|greedygrow_max - default fixed)
  -B brick_block_count1d    number of blocks per brick width (default 4)
  -P page_brick_count1d     number of bricks per page width (default 4)
Quantization
  -Q nbits                  (0: no quant.; 1 = 24 bits for index-value pairs;
                             >1: use spec. bits)
```

I parametri, per default, sono configurati come descritto nelle sezioni predecenti. L'esecuzione di

```
# vic_test_volume_compression -o decoded.raw volume.raw
```

esegue il codice di compressione, applicandolo al volume *volume.raw*, stampando le statistiche di compressione e salvando la versione decodata sul file *decoded.raw*. Il codice sorgente illustra direttamente l'uso delle librerie distribuite.

3.10 Discussione e lavori futuri

Abbiamo presentato un nuovo approccio flessibile di supporto alla esplorazione di volumi scalari rettilinei tempo-varianti. Introducendo un nuovo codec ad alte prestazioni e basso bitrate, che fa progredire lo stato dell'arte in termini di qualità ottenibile a bassissimi bitrate in regolazioni real-time, e combinandolo con un codec near-lossless all'interno di una nuova architettura software, siamo in grado di supportare un'architettura spaziale completa e l'esplorazione spazio-temporale in una varietà di configurazioni, dall'analisi locale su postazioni di lavoro grafiche all'esplorazione remota su thin clienti mobili.

Le principali limitazioni, condivise con altri approcci basati sulla compressione, sono il non trascurabile tempo di codifica e i limiti (inevitabili) della qualità ottenibile durante l'animazione, dettati dalla necessità di adattare le ingenti dimensioni dei frame alla larghezza di banda disponibile. I risultati mostrano, tuttavia, che la nostra soluzione è di immediato interesse pratico, poiché è possibile ottenere risultati di qualità eccellente su dataset variabili nel tempo con miliardi di voxel per frame e migliaia di timestep, e, combinando a livello di sistema soluzioni per l'esplorazione spaziale e per la generazione automatica di frame statici a massima qualità, molti casi d'uso possono essere gestiti.

Questo lavoro è stato fatto con la prima versione del codec TDM. I risultati ottenuti sono decisamente all'avanguardia. Attualmente stiamo sperimentando alternative e promettenti soluzioni per dataset che contengono molti spazi vuoti. Il nuovo approccio, presentato in versione preliminare a STAG 2019, sfrutta una nuova strategia di apprendimento, che permette di ottenere una migliore distribuzione spaziale degli errori all'interno dei frame, mantenendo i vincoli di bitrate per frame. Questo nuovo lavoro sarà finalizzato durante il prossimo anno del progetto ed incluso nella release finale.

4 Conclusioni

Uno degli obiettivi principali del progetto TDM è di realizzare un'architettura scalabile per l'acquisizione, l'integrazione e l'analisi di dati provenienti da sorgenti eterogenee, in grado di gestire i dati generati da un'area metropolitana estesa. L'implementazione prevista nel quadro del progetto riguarda casi di studio nell'area metropolitana della città di Cagliari, ma l'obiettivo è quello di generare soluzioni scalabili generali, che possano servire da best practice per future implementazioni di servizi in aree geografiche ampie e/o densamente popolate. Il progetto deve quindi combinare, sviluppare ed estendere soluzioni tecnologiche in vari campi, dalla sensoristica ai big data e dalla simulazione alla visualizzazione. Questo deliverable è relativo agli aspetti di visualizzazione.

Nel campo della visualizzazione, soggetto di questo deliverable, il lavoro ha due obiettivi principali: da un lato creare e validare strumenti funzionanti su piattaforma web per la visualizzazione interattiva di open data, dall'altro avanzare lo stato dell'arte nelle tecnologie abilitanti per la visualizzazione scalabile di grandi quantità di dati.

Come discusso nei capitoli precedenti, tutti i risultati attesi di progetto sono stati raggiunti nei tempi e con le caratteristiche attese, con la limitazione di quanto non possibile per cause di forza maggiore dovute alle restrizioni COVID-19. In particolare, il lavoro di ricerca ha portato allo sviluppo di nuove tecniche allo stato dell'arte per la distribuzione e la visualizzazione di dati volumetrici tempo-varianti, di tecniche per la visualizzazione di dati multilayer, e di tecnologie ed applicazioni pratiche per la visualizzazione di dati meteo, dati da monitoraggio indoor e dati da monitoraggio di consumi elettrici. Dal punto di vista della ricerca, il lavoro su questo obiettivo realizzativo ha prodotto diverse pubblicazioni internazionali [1, 2, 3, 4, 5, 6, 7], oltre a essere stato oggetto di tutorials presentati a SIGGRAPH Asia 2017 [8] e EUROGRAPHIS 2018 [9] e di una keynote lecture a Eurographics 2019 [10]. Dal punto di vista applicativo, le tecnologie di visualizzazione meteo sono già operative ed integrate nel sistema, mentre l'integrazione dei prototipi di visualizzazione di open data ambientali ed elettrici saranno presentate nel prossimo deliverable di OR6.

Trattandosi di un sistema in fase di continuo sviluppo e integrazione, le informazioni qui presentate potrebbero essere soggette a variazioni in futuro. Successivi aggiornamenti alla documentazione saranno messi a disposizione attraverso il portale del progetto `http://www.tdm-project.it/` e il repository GitHub `https://github.com/tdm-project/`.

Bibliografia

[1] F. Bettio, G. Busonera, M. Cogoni, R. Deidda, M. Del Rio, M. Gaggero, E. Gobbetti, S. Leo, S. Manca, M. Marrocu, L. Massidda, F. Marton, M. Piras, L. Pireddu, G. Pusceddu, A. Seoni, and G. Zanetti, "TDM: un sistema aperto per l'acquisizione di dati, l'analisi e la simulazione su scala metropolitana," in *Proc. GARR 2019 - Selected Papers*, 2019, pp. 44–49.

[2] G. Pintore, F. Ganovelli, A. Jaspe Villanueva, and E. Gobbetti, "Automatic modeling of cluttered multi-room floor plans from panoramic images," *Computers Graphics Forum*, vol. 38, no. 7, pp. 347–358, 2019.

[3] G. Pintore, F. Ganovelli, L. Fuentes-Perez, R. Pajarola, and E. Gobbetti, "State-of-the-art in automatic 3d reconstruction of structured indoor environments," *Computer Graphics Forum*, vol. 39, no. 2, 2020.

[4] G. Merlin, D. Ortu, G. Cherchi, and R. Scateni, "Design and implementation of a visualization tool for the in- depth analysis of the domestic electricity consumption." in *Proc. STAG*, 2019.

[5] A. Jaspe Villanueva, R. Pintus, A. Giachetti, and E. Gobbetti, "Web-based multi-layered exploration of annotated image-based shape and material models," in *The 16th Eurographics Workshop on Graphics and Cultural Heritage*, November 2019, pp. 33–42.

[6] F. Marton, M. Agus, and E. Gobbetti, "A framework for gpu-accelerated exploration of massive time-varying rectilinear scalar volumes," *Computer Graphics Forum*, vol. 38, no. 3, 2019.

[7] J. Díaz, F. Marton, and E. Gobbetti, "MTV-Player: Interactive spatio-temporal exploration of compressed large-scale time-varying rectilinar scalar volumes," in *Proc. STAG*, 2019, pp. 1–10.

[8] M. Agus, E. Gobbetti, F. Marton, G. Pintore, and P.-P. Vázquez, "Mobile graphics," in *SIGGRAPH Asia 2017 Courses*, November 2017.

[9] U. Assarsson, M. Billeter, D. Dolonius, E. Eisemann, A. Jaspe Villanueva, L. Scandolo, and E. Sintorn, "Voxel dags and multiresolution hierarchies: From large-scale scenes to precomputed shadows," in *Proc. EUROGRAPHICS Tutorials*, T. Ritschel and A. Telea, Eds., April 2018.

[10] E. Gobbetti, "Creation and exploration of reality-based models," *Computers Graphics Forum*, vol. 38, no. 2, p. xvii, 2019.

[11] Y. Li, E. Perlman, M. Wan, Y. Yang, C. Meneveau, R. Burns, S. Chen, A. Szalay, and G. Eyink, "A public turbulence database cluster and applications to study Lagrangian evolution of velocity increments in turbulence," *Journal of Turbulence*, vol. 9, 2008.

[12] R. Irion, "The terascale supernova initiative: Modeling the first instance of a star's death," *SciDAC Review*, vol. 2, no. 1, pp. 26–37, 2006.

[13] JHU, "Johns Hopkins Turbulence Databases," http://turbulence.pha.jhu.edu/datasets.aspx, 2016, [accessed: 2018-10-31].

[14] Y. Toparlar, B. Blocken, B. Maiheu, and G. Van Heijst, “A review on the cfd analysis of urban microclimate,” *Renewable and Sustainable Energy Reviews*, vol. 80, pp. 1613–1640, 2017.

[15] C. Wang, H. Yu, and K.-L. Ma, “Importance-driven time-varying data visualization,” *IEEE TVCG*, vol. 14, no. 6, pp. 1547–1554, 2008.

[16] Y. Jang, D. S. Ebert, and K. P. Gaither, “Time-varying data visualization using functional representations,” *IEEE TVCG*, vol. 18, no. 3, pp. 421–433, 2012.

[17] S. Frey and T. Ertl, “Flow-based temporal selection for interactive volume visualization,” *Computer Graphics Forum*, vol. 36, no. 8, pp. 153–165, 2017.

[18] K. Weiss and L. Floriani, “Modeling and visualization approaches for time-varying volumetric data,” in *Proc. Advances in Visual Computing*, 2008, pp. 1000–1010.

[19] B. She, P. Boulanger, and M. Noga, “Real-time rendering of temporal volumetric data on a GPU,” in *Proc. IEEE InfoVis*, 2011, pp. 622–631.

[20] J. M. Noguera and J. R. Jiménez, “Mobile volume rendering: past, present and future,” *IEEE transactions on visualization and computer graphics*, vol. 22, no. 2, pp. 1164–1178, 2016.

[21] M. Treib, K. Burger, F. Reichl, C. Meneveau, A. Szalay, and R. Westermann, “Turbulence visualization at the terascale on desktop PCs,” *IEEE TVCG*, vol. 18, no. 12, pp. 2169–2177, 2012.

[22] M. Balsa Rodriguez, E. Gobbetti, J. Iglesias Guitián, M. Makhinya, F. Marton, R. Pajarola, and S. Suter, “State-of-the-art in compressed GPU-based direct volume rendering,” *Computer Graphics Forum*, vol. 33, no. 6, pp. 77–100, 2014.

[23] J. Beyer, M. Hadwiger, and H. Pfister, “State-of-the-art in GPU-based large-scale volume visualization,” *Computer Graphics Forum*, vol. 34, no. 8, pp. 13–37, 2015.

[24] E. Gobbetti, J. Iglesias Guitián, and F. Marton, “COVRA: A compression-domain output-sensitive volume rendering architecture based on a sparse representation of voxel blocks,” *Computer Graphics Forum*, vol. 31, no. 3/4, pp. 1315–1324, 2012.

[25] C. Crassin, F. Neyret, S. Lefebvre, and E. Eisemann, “GigaVoxels: Ray-guided streaming for efficient and detailed voxel rendering,” in *Proc. I3D*, 2009, pp. 15–22.

[26] K. Engel, “CERA-TVR: A framework for interactive high-quality teravoxel volume visualization on standard PCs,” in *Proc. IEEE LDAV*, 2011, pp. 123–124.

[27] F. Reichl, M. Treib, and R. Westermann, “Visualization of big SPH simulations via compressed octree grids,” in *Proc. IEEE Big Data*, 2013, pp. 71–78.

[28] M. Hadwiger, J. Beyer, W.-K. Jeong, and H. Pfister, “Interactive volume exploration of petascale microscopy data streams using a visualization-driven virtual memory approach,” *IEEE TVCG*, vol. 18, no. 12, pp. 2285–2294, 2012.

[29] T. Fogal, A. Schiewe, and J. Kruger, “An analysis of scalable GPU-based ray-guided volume rendering,” in *Proc. IEEE LDAV*, Oct 2013, pp. 43–51.

[30] N. Fout and K.-L. Ma, “Transform coding for hardware-accelerated volume rendering,” *IEEE TVCG*, vol. 13, no. 6, pp. 1600–1607, 2007.

[31] H. Yela, I. Navazo, and P. Vazquez, “S3Dc: A 3Dc-based volume compression algorithm,” *Computer Graphics Forum*, pp. 95–104, 2008.

[32] J. A. Iglesias Guitián, E. Gobbetti, and F. Marton, "View-dependent exploration of massive volumetric models on large scale light field displays," *The Visual Computer*, vol. 26, no. 6–8, pp. 1037–1047, 2010.

[33] J. Nystad, A. Lassen, A. Pomianowski, S. Ellis, and T. Olson, "Adaptive scalable texture compression," in *Proc. HPG*, 2012, pp. 105–114.

[34] M. Kraus and T. Ertl, "Adaptive texture maps," in *Proc. Graphics Hardware*, 2002, pp. 7–15.

[35] S. Guthe and M. Goesele, "Variable length coding for GPU-based direct volume rendering," in *Proc. VMV*, 2016, pp. 77–84.

[36] S. Yu, S. Zhang, K. Wang, Y. Xia, and H. Zhang, "An efficient and fast GPU-based algorithm for visualizing large volume of 4D data from virtual heart simulations," *Biomedical Signal Processing and Control*, vol. 35, pp. 8–18, 2017.

[37] J. Schneider and R. Westermann, "Compression domain volume rendering," in *Proc. IEEE Vis.*, 2003, pp. 293–300.

[38] R. Parys and G. Knittel, "Giga-voxel rendering from compressed data on a display wall," in *Proc. WSCG*, 2009, pp. 73–80.

[39] M. Elad, *Sparse and Redundant Representations*. Springer, 2008.

[40] S. Suter, J. Iglesias Guitián, F. Marton, M. Agus, A. Elsener, C. Zollikofer, M. Gopi, E. Gobbetti, and R. Pajarola, "Interactive multiscale tensor reconstruction for multiresolution volume visualization," *IEEE TVCG*, vol. 17, no. 12, pp. 2135–2143, 2011.

[41] R. Ballester-Ripoll, P. Lindstrom, and R. Pajarola, "TTHRESH: Tensor compression for multidimensional visual data," *arXiv preprint arXiv:1806.05952*, 2018.

[42] Lindstrom, "Fixed-rate compressed floating point arrays," *IEEE TVCG*, vol. 20, no. 12, pp. 2674–2683, 2014.

[43] H.-W. Shen and C. R. Johnson, "Differential volume rendering: A fast volume visualization technique for flow animation," in *Proc. IEEE Vis*, 1994, pp. 180–187.

[44] S. Guthe and W. Straßer, "Real-time decompression and visualization of animated volume data," in *Proc. IEEE Vis*. IEEE, 2001, pp. 349–572.

[45] E. B. Lum, K.-L. Ma, and J. Clyne, "A hardware-assisted scalable solution for interactive volume rendering of time-varying data," *IEEE TVCG*, vol. 8, no. 3, pp. 286–301, 2002.

[46] L. Ibarria, P. Lindstrom, J. Rossignac, and A. Szymczak, "Out-of-core compression and decompression of large n-dimensional scalar fields." *Computer Graphics Forum*, vol. 22, no. 3, pp. 343–348, 2003.

[47] J. Woodring, C. Wang, and H.-W. Shen, "High dimensional direct rendering of time-varying volumetric data," in *Proc. IEEE Vis*, 2003, pp. 417–424.

[48] H. Wang, Q. Wu, L. Shi, Y. Yu, and N. Ahuja, "Out-of-core tensor approximation of multidimensional matrices of visual data," *ACM TOG*, vol. 24, no. 3, pp. 527–535, Jul. 2005.

[49] R. Westermann, "Compression domain rendering of time-resolved volume data," in *Proc.IEEE Vis*, 1995, pp. 168–175.

[50] K.-L. Ma and H.-W. Shen, "Compression and accelerated rendering of time-varying volume data," in *Proc. International Workshop on Computer Graphics and Virtual Reality*, 2000, pp. 82–89.

[51] C. Wang, J. Gao, L. Li, and H.-W. Shen, "A multiresolution volume rendering framework for large-scale time-varying data visualization," in *Proc. Volume Graphics*, 2005, pp. 11–19.

[52] H.-W. Shen, "Visualization of large scale time-varying scientific data," *Journal of Physics*, vol. 46, no. 1, pp. 535–544, 2006.

[53] C.-L. Ko, H.-S. Liao, T.-P. Wang, K.-W. Fu, C.-Y. Lin, and J.-H. Chuang, "Multi-resolution volume rendering of large time-varying data using video-based compression," in *Proc. IEEE Pacific Vis*, 2008, pp. 135–142.

[54] J. Mensmann, T. Ropinski, and K. Hinrichs, "A GPU-supported lossless compression scheme for rendering time-varying volume data." in *Proc. Volume Graphics*, 2010, pp. 109–116.

[55] D. Nagayasu, F. Ino, and K. Hagihara, "Two-stage compression for fast volume rendering of time-varying scalar data," in *Proc. GRAPHITE*, 2008, pp. 275–284.

[56] C. Wang, H. Yu, and K.-L. Ma, "Application-driven compression for visualizing large-scale time-varying data," *IEEE CGA*, vol. 30, no. 1, pp. 59–69, 2010.

[57] Y. Cao, G. Wu, and H. Wang, "A smart compression scheme for GPU-accelerated volume rendering of time-varying data," in *Proc. IEEE ICVRV*, 2011, pp. 205–210.

[58] J. Pulido, D. Livescu, K. Kanov, R. C. Burns, C. Canada, J. P. Ahrens, and B. Hamann, "Remote visual analysis of large turbulence databases at multiple scales," *J. Parallel Distrib. Comput.*, vol. 120, pp. 115–126, 2018.

[59] M. Aharon, M. Elad, and A. Bruckstein, "K-SVD: An algorithm for designing overcomplete dictionaries for sparse representation," *IEEE TSP*, vol. 54, no. 11, pp. 4311–4322, 2006.

[60] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proc. PPoPP*, 2006, pp. 48–57.

[61] "ZFP compression library," https://computation.llnl.gov/projects/floating-point-compression/zfp-versions, 2018, [accessed: 2018-10-31].

[62] P. S. Efraimidis, "Weighted random sampling over data streams," in *Algorithms, Probability, Networks, and Games*, 2015, pp. 183–195.

[63] P. Sinha and A. A. Zoltners, "The multiple-choice knapsack problem," *Operations Research*, vol. 27, no. 3, pp. 503–515, 1979.

[64] R. Rubinstein, M. Zibulevsky, and M. Elad, "Efficient implementation of the K-SVD algorithm using batch orthogonal matching pursuit," CS Technion, Tech. Rep., 2008.

[65] "CUDA toolkit," https://developer.nvidia.com/cuda-toolkit, [accessed: 2018-10-31].

[66] T. Foley and J. Sugerman, "KD-tree acceleration structures for a GPU raytracer," in *Proc. Graphics hardware*, 2005, pp. 15–22.

[67] E. Gobbetti, F. Marton, and J. A. Iglesias Guitián, "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets," *The Visual Computer*, vol. 24, no. 7–9, pp. 797–806, 2008.

[68] W.-H. Hsu, J. Mei, C. D. Correa, and K.-L. Ma, "Depicting time evolving flow with illustrative visualization techniques," in *International Conference on Arts and Technology*. Springer, 2009, pp. 136–147.

[69] A. Brambilla, R. Carnecky, R. Peikert, I. Viola, and H. Hauser, "Illustrative flow visualization: State of the art, trends and challenges," *Proc. EG STAR*, 2012.

[70] P. Holub, M. Srom, M. Pulec, J. Matela, and M. Jirman, "GPU-accelerated DXT and JPEG compression schemes for low-latency network transmissions of HD, 2K, and 4K video," *Future Generation Computer Systems*, vol. 29, no. 8, pp. 1991–2006, 2013.

[71] D. Laney, P.-T. Bremer, A. Mascarenhas, P. Miller, and V. Pascucci, "Understanding the structure of the turbulent mixing layer in hydrodynamic instabilities," *IEEE TVCG*, vol. 12, no. 5, pp. 1053–1060, 2006.

[72] "ASTC compression library," https://github.com/ARM-software/astc-encoder, 2017, [accessed: 2018:10:31].

[73] "CUDACOMPRESS compression library," https://github.com/m0bl0/cudaCompress, 2013, [accessed: 2019-02-01].

[74] S. Di and F. Cappello, "Fast error-bounded lossy HPC data compression with SZ," in *Proc. IEEE IPDPS*, 2016, pp. 730–739.

[75] "SZ compression library," https://github.com/disheng222/SZ, 2018, [accessed: 2018-10-31].

[76] Z. Wang, A. Bovik, H. Sheikh, and E. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE TIP*, vol. 13, no. 4, pp. 600 –612, 2004.

[77] J. Diffenderfer, A. Fox, J. Hittinger, G. Sanders, and P. Lindstrom, "Error analysis of ZFP compression for floating-point data," *SIAM Journal on Scientific Computing*, 2019, to appear.