POLITECNICO DI TORINO

Collegio di Ingegneria Informatica, del Cinema e Meccatronica

Corso di Laurea Magistrale
in Ingegneria del Cinema e dei Mezzi di Comunicazione

Tesi di Laurea Magistrale

DESTRUCTION

Relatore
prof. Andrea Giuseppe Bottino

Candidato
Matteo Luison

Luglio 2017

Indice

Introduzione

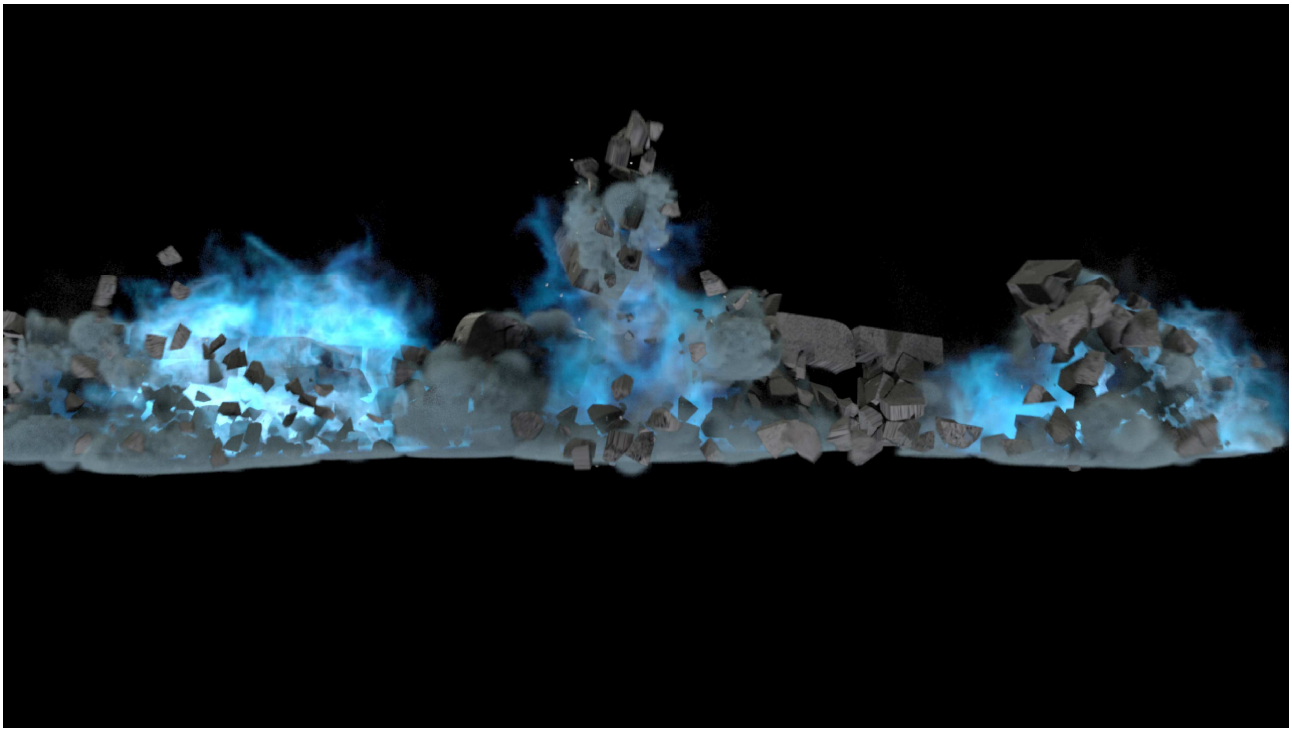
Il mondo dei *VFX* è vastissimo, sempre in continua espansione ed evoluzione. La tecnologia e lo sviluppo hanno permesso agli *"artisti"* che svolgono questo mestiere di farlo al meglio e sempre in modo più facile, veloce e intuitivo.
In questa trattazione viene discusso l'argomento dei *Destruction*, in accordo anche con l'esperienza svolta all'interno di un'azienda milanese, la *EDI*, che lavora nel settore della post produzione video e in particolare proprio nel campo degli effetti visivi. È stato concordato, appunto con la *EDI*, di sviluppare il tema dei *Destruction,* una tipologia di effetti visivi di grande rilevanza in questi anni in quanto sempre più richiesti ed utilizzati, soprattutto in ambito cinematografico, ma non solo. I *Destruction* sono tutti quegli effetti legati al mondo delle distruzioni che, inseriti nei prodotti audiovisivi, contribuiscono a rendere sempre più stupefacenti le sequenze nelle quali essi sono introdotti; si tratta, in sostanza, di avvenimenti che si verificano a seguito di altre motivazioni, come ad esempio le esplosioni, i terremoti, ecc., in sintesi di eventi che si possono verificare a causa di catastrofi naturali e non. I *Destruction* sono oramai degli effetti importanti nella filmografia odierna per generi come il *Fantastico*, la *Fantascienza*, l'*Azione*, il *Catastrofico*, lo *Storico*, l'*Horror* e molti altri, largamente utilizzati anche nei *Commercial*, questo perché, molto spesso, ricreare questo genere di fenomeni senza l'ausilio del digitale, e quindi riprodurli senza l'aiuto dei *VFX*, sarebbe una cosa impossibile o molto costosa, soprattutto quando si tratta di avvenimenti che si verificano su larga scala o che si riferiscono a un qualcosa di non esistente. Integrati nei prodotti audiovisivi, negli anni, si è cercato di rendere questi effetti sempre più verosimili e spettacolari, grazie anche allo sviluppo tecnologico che ha permesso la creazione di strumenti, anche dedicati, in grado di svolgere questo tipo di lavorazioni.
Il film che segna un punto di svolta nel settore degli effetti visivi che riguardano i *Destruction* è *2012*. Grazie all'uscita di questa pellicola, molte aziende del settore hanno incominciato a sviluppare pipeline di lavoro sempre più grandi e robuste, arrivando ad ottenere simulazioni sempre più precise e complesse, lavori di grande qualità effettuati anche su simulazioni di grande scala grazie, soprattutto, all'incremento della potenza di calcolo dei computer. La *Digital Domain* è una delle case di produzione che ha partecipato alla realizzazione degli *VFX* del film *2012*. L'azienda in questione è stata fondata nel 1993, ed è ora uno dei più grandi e influenti studi di post produzione per gli effetti visivi del mondo. Il lavoro svolto dalla *Digital Domain* si basa essenzialmente su *Bullet*, il motore di simulazione fisico che viene anche utilizzato per la realizzazione del prodotto audiovisivo riferito a questo testo, il cui compito è principalmente quello di effettuare il rilevamento delle collisioni, risolvere le collisioni e altri vincoli e fornire le trasformate del sistema mondo aggiornate per tutti gli oggetti presenti all'interno di una simulazione.

Bullet è semplice, veloce, stabile e open source, inoltre, nello specifico, nella sequenza audiovisiva di *Destruction* riferita al progetto oggetto di questa trattazione, è stata utilizzata per gestire la dinamica di simulazione, in particolare per gestire gli aspetti riguardanti l'operazione di rilevamento delle collisioni: un passo fondamentale nella costruzione di questo genere di simulazioni. Fino alla realizzazione di *2012* gli studi, le istituzioni accademiche e progetti open source hanno sempre dato il loro contributo allo sviluppo di strumenti per il lavoro di *Destruction* e non solo. In questo scenario, è stato quindi utile, in queste pagine, affrontare un ampio discorso sugli strumenti attualmente a disposizione che consentono di migliorare la velocità e la qualità del lavoro "dell'artista" che si occupa di questa tipologia di effetti visivi. Si tratta per lo più in generale di plugin, ovvero applicativi integrati all'interno di altre piattaforme utilizzate in questo ambito, come and esempio *Maya*, ma non solo, anche come lo stesso *Houdini,* software utilizzato per concretizzare la costruzione della simulazione di *Destruction* oggetto di questa tesi, che appunto sono in grado di offrire, come dice *Peter Kyme*, un operatore del settore, stabilità, velocità e flessibilità. Tutte caratteristiche fondamentali, queste, per lo sviluppo di strumenti che possono in qualche modo semplificare e velocizzare il lavoro di *"un'artista"* di *Destruction*.
In questo documento viene descritto, nel dettaglio, un possibile progetto di *Destruction*, l'obiettivo del lavoro prodotto dall'esperienza di tirocinio alla *EDI* che integra appunto questo scritto, come dimostrazione del processo che ogni *"artista"* che lavora su questa tipologia di effetti deve affrontare per poter portare a compimento questo genere di lavori e consiste, come accennato in precedenza, nella realizzazione di un prodotto audiovisivo. In particolare, in questo contesto, vengono discusse le scelte effettuate, le difficoltà affrontate e le soluzioni messe in atto per risolvere le problematiche incontrate durante il percorso. Vengono affrontate, quindi, tutte le fasi che compongono il processo creativo di *Destruction* in questione e la sua effettiva implementazione, dalla fase iniziale di ideazione alla fase finale di *Compositing*, passando dalla progettazione e modellazione a quella vera e propria di *Destruction*, che si divide ulteriormente in prefratturazione o fratturazione, creazione dei vincoli o constraint, simulazione e *Rendering*. Per affrontare il processo di creazione del video di *Destruction,* della durata di pochi secondi, oggetto di questa trattazione, è stato necessario l'utilizzo di tre differenti tipologie di software: *Blender* per la modellazione, *Houdini* per la parte di *Destruction* e *Nuke* per la fase di *Compositing*.
La seguente tesi si propone, inoltre, di mettere in evidenza gli approcci o le metodologie impiegate per fare *Destruction,* approfondendo inoltre, tramite una panoramica dettagliata, i pacchetti o gli strumenti attualmente in commercio utilizzati per compiere questo genere di lavorazioni. La simulazione di *Destruction* che accompagna questo scritto appartiene alla dinamica dei corpi rigidi (*RBS* o *RBD*). Nei *VFX* e nella *CGI (Computer Generated Imagery*) l'approccio *RBS* o *RBD* è ancora il più rilevante, anche per quanto riguarda la realizzazione di questa tipologia di *VFX*, ma non è l'unico.

Nello specifico, gli approcci utilizzati in questo settore sono essenzialmente due: *RBS* (Rigid Body Simulations) o *RBD* (Rigid Body Dynamics) e *FEA (Finite Element Analysis)*. Il primo, ovvero il metodo utilizzato per lo sviluppo del progetto raccontato in queste pagine, è un approccio che si può costruire attorno a librerie fisiche esistenti, come *Bullet*, *PhysX* o *ODE*, oppure attraverso l'utilizzo di librerie fisiche specifiche, come *PhysBAM,* della Stanford University. Questo metodo ha il vantaggio di essere quello più veloce, tuttavia rispetto a *FEA* porta con sé la perdita di una grossa fetta di realismo, inoltre, questo approccio, vincola il controllo *"dell'artista"* sull'aspetto finale della frattura. Il secondo metodo citato, invece, in un primo momento legato solamente al modo dell'ingegneria, si è esteso, ora, anche al mondo dei *VFX*, permettendo la realizzazione di simulazioni molto più precise rispetto a quelle che si possono ottenere con il metodo *RBS* o *RBD*. *FEA* richiede inoltre un tempo di calcolo maggiore, tuttavia, a differenza del precedente, produce una frattura dinamica che raggiunge un grande livello di realismo e che prevede da parte dell'operatore una minor attenzione ai dettagli, in quanto, in questo caso, costui deve solamente regolare dei parametri senza intervenire in maniera aggressiva controllando e gestendo le interazioni all'interno della simulazione. In sostanza, seguendo questi due approcci, un oggetto può essere fratturato in due modi: si possono segnare, in una fase precedente, i punti di rottura, per poi, in una fase successiva, distruggere l'oggetto, secondo questi punti, in base a una serie di regole, utilizzando quindi il metodo *RBS* o *RBD*, oppure si può calcolare la frattura in modo dinamico, utilizzando il metodo *FEA*, per determinare i carichi o le tensioni interne dell'oggetto da sottoporre a rottura. In ogni caso tutti gli approcci sono ugualmente validi, non sono altro che differenti percorsi di sviluppo che convergono in un unico punto, la risoluzione di uno stesso problema, ovvero la realizzazione di simulazioni sempre più stupefacenti e complesse con l'intento di spingersi sempre più verso i confini del realismo.
Questo lavoro si propone, inoltre, di dare delle precise regole per affrontare un qualsiasi lavoro di *Destruction* costruito su una pipeline di lavoro che segue un approccio di tipo *RBS* o *RBD*. In generale, questi punti fissi su quali si può costruire un qualunque lavoro di *Destruction,* proprio come è stato fatto all'interno del progetto oggetto di questa trattazione, sono essenzialmente tre: frattura degli oggetti, in questa fase viene preparato il modello, ovvero la geometria che verrà sottoposta alla rottura, settaggio dei vincoli e costruzione della dinamica di simulazione, in questa fase vengono creati i legami, ovvero le relazioni tra i pezzi che influenzeranno lo stato della simulazione ed esecuzione della simulazione e risoluzione della procedura di *Collision Detection*, ovvero l'operazione che si occupa del rilevamento delle collisioni per determinare i punti di contatto tra i corpi che interagiscono fra di loro all'interno della simulazione.

Oggigiorno, le simulazioni che gestiscono la dinamica dei corpi rigidi, permettono di gestire sempre più oggetti in contemporanea e consentono di fornire risultati attraverso calcoli sempre più veloci. Gli operatori coinvolti con questo genere di simulazioni devono fare i conti, quindi, con il compromesso precisione/prestazione. Spesso, siccome il vincolo prestazionale non consente il calcolo di soluzioni precise, e quindi qualitative, è necessario bilanciare le proprietà di precisione e stabilità per poter soddisfare il vincolo sulla prestazione. Per questo gli *"artisti"* del settore sono sempre alla ricerca di nuovi metodi per far si che queste simulazioni siano sempre più veloci e robuste e per poter lavorare su scenari sempre più complessi, come ad esempio una simulazione di *Destruction* su larga scala che deve sottostare a forti vincoli prestazionali. Le simulazioni di cui ci si sta occupando in questo testo sono delle simulazioni interattive, che oramai sono diventate importanti per molti strumenti informatici moderni impiegati in una vasta gamma di settori di applicazione, come ad esempio i videogiochi, i software di animazione per la produzione digitale, gli effetti speciali per il cinema e i film di animazione, la robotica e la prototipazione virtuale. Nel settore della produzione cinematografica, ad esempio, molto simile a quello dei videogiochi, le simulazioni di corpo rigido hanno bisogno di essere plausibili piuttosto che fisicamente corrette. Il discorso sul realismo di simulazione è di conseguenza molto importante, in alcuni casi, quindi, il movimento degli oggetti può essere completamente guidato dalla dinamica del corpo rigido, ovvero fisicamente corretto ma complesso da gestire a livello computazionale, più comunemente, invece, si può raggiungere il realismo di simulazione guidando e adattando il movimento degli oggetti attraverso l'utilizzo della cinematica. In questo contesto, a livello progettuale, si possono definire una vasta gamma di tecnologie: i *Simulatori Offline*, che possono richiedere anche ore o giorni per l'elaborazione dei dati con il vantaggio di fornire risultati di elevata qualità, *Simulatori Veloci*, che sono in grado di fornire risultati plausibili in tempi molto brevi, *Bullet* ad esempio è uno di questi, e *Simulatori Moderatamente Veloci*, che permettono di raggiungere risultati discreti e forniscono un buon realismo di simulazione.
Per quanto riguarda il tempo di esecuzione riferito alle simulazioni impiegate nel settore dei *VFX*, è importante dire che non è necessario che questo sia un tempo reale, ovvero non è necessario che queste simulazioni avvengano in tempo reale, nonostante ciò sono comunque preferiti metodi di simulazione veloci, in quanto spesso subentrano problemi effettivi di costo e di tempo di simulazione e di conseguenza di produzione. Il primo metodo che implementa la dinamica del corpo rigido è stato quello discusso da *Baraff* nel *1994*. Questo metodo, però, nella gestione delle dinamiche di simulazione costituite da più di circa cento oggetti perdeva la sua interattività. Questo perché quando il numero di oggetti interagenti aumentava, e superava quindi il centinaio, cresceva anche il costo computazionale associato alle simulazioni. Con il tempo, grazie allo sviluppo della tecnologia, sono stati trovati dei metodi migliori e la comunità che ruota attorno alle simulazioni interattive si è spostata su motori di simulazione più avanzati, come *Bullet* ad esempio, che non a caso, è anche quello impiegato nella simulazione di *Destruction* realizzata all'interno di questo documento.

Il seguente lavoro, quindi, si concentra, nello specifico, sul processo di realizzazione di un prodotto audiovisivo di *Destruction*. La tesi, seguendo lo sviluppo di questo progetto, il cui risultato si incarna in una sequenza interamente ricostruita in digitale che mostra il crollo di un edificio, vuole essere una sorta di guida di accompagnamento per il lettore che vorrà approcciarsi, per la prima volta, a questo tipo di lavorazioni, cecando quindi di fornire delle indicazioni generali che gli consentiranno di affrontare un qualsiasi lavoro di *Destrution*, su qualsiasi piattaforma disponibile che supporti la creazione di questa tipologia di *VFX*.



Capitolo uno: la teoria

Introduzione

Nel seguente capitolo verranno discussi gli aspetti principali connessi ad una pipeline di post produzione di *Destruction*, per la quale saranno richiamati tutti i concetti teorici fondamentali atti a comprenderne il suo funzionamento e la sua costruzione in vista delle operazioni che saranno svolte nel terzo capitolo, all'interno del software *Houdini*, nel quale verrà raccontato, nel dettaglio, il progetto che accompagna questo documento.
Il capitolo, nello specifico, racconta le metodologie e i procedimenti impiegati per realizzare simulazioni di *Destruction,* che sono essenzialmente due: *RBS (Rigid Body Simulations)* o *RBD (Rigid Body Dynamics)* e *FEA (Finite Element Analysis)*. Questi sono quindi due differenti pipeline di lavoro, quelle più utilizzate, per realizzare questo genere di effetti. Il metodo *RBS* o *RBD,* legato alla fisica dei copri rigidi, è quello che verrà spiegato più nel dettaglio nelle pagine successive in quanto è stato quello utilizzato per la realizzazione del progetto di *Destrutcion* oggetto di questo testo sviluppato tramite l'utilizzo del software in *Houdini* e del quale verranno raccontati i concetti matematici e fisici alla base, la pipeline di lavoro e le fasi che costituiscono un processo di *Destruciton* sviluppato attraverso questa metodologia. L'approccio *RBS* o *RBD* ha inoltre una caratteristica, ovvero quella di vincolare il controllo *"dell'artista"* sull'aspetto finale della frattura. *"L'artista"*, quindi, potrebbe realizzare un modello di frattura realistico, facendo in modo che il punto da lui indicato dal quale si genererà la frattura, ovvero il centro della frattura, sia vicino al punto di impatto, oppure potrebbe realizzare un modello meno realistico costruendo una frattura uniforme non centrata attorno ad alcun punto. Questo perché, questo metodo, implica che colui che realizzerà la simulazione di *Destruction*, ovvero *"l'artista"*, debba specificare i punti, all'interno del modello, dai quali si genererà la frattura. Il capitolo si conclude con una breve descrizione degli approcci *PhysBAM*, anch'esso basato su una pipeline di simulazione di corpo rigido, e *FEA (Finite Element Analysis)*.

RBS (Rigid Body Simulations) o RBD (Rigid Body Dynamics)

I concetti matematici e fisici

La simulazione del moto di un sistema di corpi rigidi si basa su un noto sistema di equazioni differenziali, le equazioni di *Newton* e *Eulero*. Queste equazioni si applicano solo quando i corpi vengono osservati da un sistema di coordinate inerziali o globale, ovvero privo di accelerazione, e quando in un punto di contatto tra due corpi la forza applicata da un corpo sul secondo è di uguale grandezza, opposta direzione e collineare alla forza applicata dal secondo sul primo. Queste due implicazioni, per la *seconda legge di Newton*, che afferma che la variazione nel tempo del moto di un corpo è uguale alla forza applicata, permettono di ottenere le equazioni differenziali del moto.
Mentre la seconda legge si applica solo alle particelle, *Eulero* ha esteso questo concetto al caso dei corpi rigidi, visualizzandoli come una collezione di un numero infinito di particelle. Ed è questo il motivo per il quale queste formule sono conosciute come equazioni di *Newton* e *Eulero*.
Realizzare una simulazione di corpo rigido, quindi, significa risolvere un sistema di equazioni differenziali ordinarie non lineari, per le quali non esistono soluzioni in forma chiusa e che costituiscono un modello a tempo istantaneo che viene discretizzato nel tempo, per produrre un modello a tempo discreto approssimato, costituito tipicamente da un sistema di equazioni algebriche e disequazioni. Il modello a tempo discreto in questione, viene poi risolto per ogni momento di interesse, ovvero a ogni istante di tempo $(t_0, \dots, t_N)$.
In sostanza il modello a tempo istantaneo, costituito da un sistema di equazioni algebriche differenziali e disuguaglianze, che descrive i continui moti dei corpi rigidi, può essere riformulato in un problema di complementarità non lineare differenziale detto anche *dNCP*. Siccome il *dNCP* non può essere risolto in forma chiusa o direttamente, allora viene discretizzato nel tempo, producendo così una sequenza di *NCP,* le cui soluzioni approssimano lo stato e le traiettorie delle forze di contatto del sistema.
Prima di presentare le equazioni di *Newton* e *Eulero*, è necessario introdurre una serie di concetti provenienti dalla meccanica classica. Si partirà dalle caratteristiche coinvolte nel moto di una particella per poi estendere il discorso al caso dei corpi rigidi, immaginandoli costituiti da un insieme di punti o particelle.

Il moto di una particella

Si supponga di voler simulare il moto di una particella data una posizione nello spazio $x(t)$ nel sistema di riferimento globale, ovvero lo spazio che la particella occupa durante la simulazione, al tempo t. Il vettore $x(t)$ descrive la traslazione della particella a partire dall'origine. La funzione $v(t) = x'(t)$, ovvero la derivata rispetto al tempo della posizione, determina la velocità della particella al tempo t. Lo stato della particella al tempo t è indicato dalla sua posizione e dalla sua velocità. Si può quindi definire un vettore di stato $X(t)$ per una singola particella come:

$$X(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}$$

$X(t)$ può essere descritto come un array di sei numeri: generalmente i primi tre si riferiscono $x(t)$ mentre i secondi tre a $v(t)$. Per un sistema con n particelle il vettore di stato $X(t)$ è:

$$X(t) = \begin{pmatrix} x_1(t) \\ v_1(t) \\ \cdots \\ x_n(t) \\ v_n(t) \end{pmatrix}$$

Nel moto di una particella si devono anche considerare le forze che agiscono su di essa al tempo t. $F(t)$ è quindi la somma di tutte le forze che agiscono sulla particella. Se la particella ha una massa m, il cambiamento di $X(t)$ nel tempo è dato da $X(t)$ derivato rispetto al tempo. Ad occuparsi dell'esecuzione di questo calcolo è tipicamente un solver, quindi, per conoscere la posizione al tempo t_1 della particella, il solver riceverà la posizione iniziale di partenza e il tempo t_0e t_1.

Il corpo rigido

I corpi rigidi sono più complicati rispetto alle particelle perché, oltre alla traslazione, hanno anche una rotazione, che le particelle, essendo assunte come dei punti, non hanno. Per posizionare un corpo rigido nello spazio di un sistema di coordinate globali, viene utilizzato un vettore $x(t)$ che descrive la sua traslazione. Viene poi definita una matrice di rotazione $R(t)$ e vengono chiamati $x(t)$ e $R(t)$ le variabili spaziali del corpo rigido. Un corpo rigido a differenza di una particella ha una forma e occupa un volume nello spazio. Si supponga che $R(t)$ specifichi la rotazione del corpo rispetto al suo centro di massa, che per il momento si suppone posto nell'origine del sistema di riferimento, si supponga poi un vettore r che fa ruotare il corpo nel sistema di coordinate globale al tempo t. Si ottiene $R(t) * r$. Quindi:

$$p(t) = R(t) * p_0 + x(t)$$

Considerando la traslazione e la rotazione insieme, un corpo rigido ha sei gradi di libertà. Come precedentemente detto, possiamo pensare a un corpo rigido come costituito da un numero infinito di particelle o punti, un oggetto solido nel quale la distanza tra ogni coppia di punti, al suo interno, non cambierà mai anche se vengono applicate delle forze di grande portata.

La velocità lineare o velocità traslazionale

Si supponga che $x(t)$ e $R(t)$ siano la posizione e l'orientamento del corpo al tempo t. Qui si vuole definire come questi due parametri cambiano nel tempo; $x(t)$ è la posizione del centro di massa nel sistema di coordinate globale, mentre $x'(t)$, ovvero la sua derivata, determina la velocità del centro di massa nel sistema di coordinate globali. La velocità lineare è definita quindi come:

$$v(t) = x'(t)$$

e indica la velocità della traslazione. La velocità lineare è un attributo di un punto, non di un corpo, perché quando un corpo ruota non tutti i punti che lo compongono assumono la stessa velocità. Tuttavia, la velocità di ogni punto può essere determinata a partire dalla velocità di un punto di riferimento, che nella dinamica del corpo rigido è tipicamente il centro di massa, e dalla velocità angolare del corpo.

La velocità angolare

Un corpo rigido, quindi, oltre a traslare può anche ruotare. Si può identificare la rotazione del corpo tramite il vettore $w(t)$. La direzione di $w(t)$ indica la direzione dell'asse rispetto al quale il corpo sta ruotando. La grandezza o ampiezza di $w(t)$, ovvero $|w(t)|$, indica quanto veloce il corpo sta ruotando. $|w(t)|$ si riferisce all'angolo attraverso il quale il corpo ruoterà su un periodo di tempo dato se la velocità angolare rimane costante. La quantità $w(t)$ è detta, appunto, velocità angolare. Mentre la velocità di rotazione può cambiare nel tempo, in ogni istante, ogni punto su un corpo rigido ha esattamente la stessa velocità di rotazione.

La relazione tra la matrice di rotazione e la velocità angolare

$R(t)$ è una matrice e $w(t)$ è un vettore. Le colonne di $R(t)$ determinano la direzione della trasformazione rispetto agli assi x, y e z al tempo t, quindi derivando $R(t)$ si ottiene $R'(t)$, le cui colonne descrivono le velocità rispetto ai vari assi. Dato un corpo rigido con velocità angolare $w(t)$, un vettore $r(t)$ al tempo t specificato nel sistema di coordinate globali, si ottengono:

$$R'(t) = w(t) * R(t)$$
$$r'(t) = w(t) * r(t)$$

L'orientamento di un corpo rigido è definito come il l'orientamento del sistema di coordinate del corpo fisso rispetto al sistema di coordinate inerziale o globale.

Il centro di massa

Un corpo rigido può essere pensato come a un insieme di tante piccole particelle che vanno da 1 a N. La massa di ogni singola particella è pari a m_i e ogni particella ha una posizione r_{0i}. La posizione di ogni particella nello spazio globale al tempo t è identificata da $r_i(t) = R(t) * r_{0i} + x(t)$. La massa totale del corpo è quindi $M = \sum_1^N m_i$. La velocità $r_i'(t)$ dell'i-esima particella è ottenuta utilizzando la relazione $R'(t) = w * R(t)$, quindi $r_i'(t) = w * R(t) * r_{0i} + v(t) = w(t) * (r_i(t) - x(t)) + v(t)$. Il centro di massa di un corpo nello spazio globale è definito da:

$$\frac{\sum m_i r_i(t)}{M}$$

dove m_i sono le masse individuali delle particelle mentre M quella del corpo.

La forza e il momento torcente

$F_i(t)$ denota la forza totale proveniente dalle forze esterne che agiscono sulla particella i-esima al tempo t. Il momento torcente esterno $\tau_i(t)$ che agisce sulla i -esima particella può essere definito come $\tau_i(t) = \big(r_i(t) - x(t)\big) * F_i(t)$. Il momento torcente differisce dalla forza in quanto il momento torcente su una particella dipende dalla posizione $r_i(t)$ della particella relativa al centro di massa $x(t)$. Intuitivamente si può pensare la direzione di $\tau_i(t)$ come l'asse del corpo che potrebbe ruotare a causa di $F_i(t)$. La forza totale esterna $F(t)$ che agisce sul corpo è la somma delle $F_i(t)$, mentre il momento torcente totale per un corpo è definito come:

$$\tau(t) = \sum \tau_i(t) = \sum \big(r_i(t) - x(t)\big) * F_i(t)$$

$F(t)$ non da alcuna informazione su dove le varie forze agiscono sul corpo, $\tau(t)$, invece, da qualche informazione sulla distribuzione delle forze $F_i(t)$ nel corpo.

Il momento lineare

Il momento lineare p di una particella con massa m e velocità v è definito come $p = m * v$. Il momento lineare totale $P(t)$ di un corpo rigido è la somma dei prodotti delle masse e delle velocità di ogni particella:

$$P(t) = \sum m_i(t) r_i'(t)$$

La velocità di ogni singola particella è $r_i'(t) = v(t) + w(t) * \big(r_i(t) - x(t)\big)$. Il momento lineare totale è quindi:

$$P(t) = M * v(t)$$

Da questa formula si ottiene che il momento lineare totale del corpo rigido è lo stesso, come se il corpo fosse semplicemente composto da una particella con massa M e velocità $v(t)$. Il momento lineare esprime l'effetto della forza totale $F(t)$ su un corpo rigido. Il cambiamento del momento lineare è equivalente alla forza totale che agisce su un corpo.

Il momento angolare

Si supponga un corpo che galleggia in uno spazio senza un momento torcente che agisce su di esso, il momento angolare del corpo è quindi costante. Anche se il momento angolare di un corpo è costante, la sua velocità angolare potrebbe non esserlo. Di conseguenza la velocità angolare del corpo può variare anche quando nessuna forza agisce su di esso. Il momento angolare totale $L(t)$ di un corpo rigido è:

$$L(t) = I(t) w(t)$$

Dove $I(t)$ è una matrice 3×3 chiamata tensore inerziale che descrive come la massa di un corpo è distribuita relativamente al suo centro di massa. Il tensore $I(t)$ dipende dall'orientamento del corpo, ma non dipende dalla traslazione del corpo.

Le equazioni di Newton e Eulero

Sia il momento lineare che quello angolare sono delle funzioni lineari della velocità. $L(t)$ è indipendente da qualsiasi effetto di traslazione mentre $P(t)$ è indipendente da qualsiasi effetto di rotazione.

$$P'(t) = F(t)$$
$$L'(t) = \tau(t)$$

Queste formule si ottengono derivando i momenti. A questo punto si hanno in mano tutti gli strumenti per poter presentare e le equazioni di *Newton* e *Eulero.* Queste equazioni in sostanza si ricavano applicando la legge di *Newton* per due volte, una volta per il moto traslazionale e una per quello rotazionale. Se queste equazioni vengono adattate al caso di un corpo rigido si possono ricavare, appunto, la forza e il momento torcente del corpo.

Il tensore inerziale

Il tensore inerziale $I(t)$ è il fattore di scala tra il momento angolare $L(t)$ e la velocità angolare $w(t)$.

$$I(t) = R(t)I_{body}R(t)^T$$

I_{body} rimane costante durante una simulazione, quindi, se I_{body} viene calcolato, per un corpo, prima che abbia inizio la simulazione, si ottiene facilmente il tensore inerziale $I(t)$.

$$I^{-1}(t) = R(t)I_{body}^{-1}R(t)^T$$

$$R(t)^T = R(t)^{-1}$$

$$(R(t)^T)^T = R(t)$$

Chiaramente anche I_{body}^{-1} rimane costante durante la simulazione.

Le equazioni del moto di un corpo rigido

Le componenti del vettore di stato $X(t)$ per un corpo rigido sono quindi:

$$X(t) = \begin{pmatrix} x(t) \\ R(t) \\ P(t) \\ L(t) \end{pmatrix}$$

Lo stato di un corpo rigido è dato dalla sua posizione e il suo orientamento, che descrivono le informazioni spaziali, e dal momento lineare e angolare, che descrivono le informazioni di velocità. M e I_{body} rimangono costanti, quindi vengono calcolati prima che la simulazione abbia inizio. Ad ogni istante di tempo, invece, vengono calcolati:

$$v(t) = \frac{P(t)}{M}$$

$$I(t) = R(t)I_{body}R(t)^T$$

$$w(t) = I(t)^{-1}L(t)$$

La pipeline di simulazione della dinamica del corpo rigido

Introduzione

Ora si dispongono di tutti gli elementi necessari per poter descrivere la pipeline di simulazione della dinamica del corpo rigido. Nello specifico, la simulazione coinvolta nella dinamica di un corpo rigido, si compone di tre fasi che si ripetono sempre nello stesso ordine, in un loop o ciclo, e sono:

1. *Collision Detection*
2. *Contact Handling e Collision Resolution*
3. *Time Integration*

Il simulatore *RBS* o *RBD* è complesso, e per questo, come appena accennato, è stato suddiviso in blocchi ben definiti, ciascuno incaricato di risolvere un singolo semplice compito, inseriti in un ciclo di simulazione. Nello specifico, il ciclo di simulazione inizia facendo un'interrogazione per rilevare le collisioni, e quindi per trovare i punti di contatto tra i vari oggetti, o corpi, interagenti all'interno della simulazione *(Collision Detection)*. I punti di contatto trovati sono necessari per determinare le leggi fisiche che governano il moto dei corpi e per determinare le forze di contatto che restituiscono gli adeguati effetti di attrito di contatto e impediscono ai corpi di compenetrare *(Contact Handling).* Ogni contatto trovato implica la presenza di una collisione ed è accompagnato da delle forze impulsive, ovvero delle forze con grandezza infinita agenti su periodi di tempo infinitesimali. Queste forze impulsive causano cambiamenti istantanei nelle velocità dei corpi e vengono gestite separatamente *(Collision Resolution).* Dopo aver calcolato tutte le forze di contatto, le posizioni e le velocità dei corpi, esse vengono integrate in avanti nel tempo prima che inizi il nuovo ciclo di simulazione *(Time Integration).*

Collision Detection

Le simulazioni di corpo rigido si basano su due elementi fondamentali, il primo è il motore fisico, il secondo è il componente che si occupa di rilevare le collisioni, il quale permette di intercettare gli oggetti che confinano o compenetrano tra loro e di calcolare gli effetti che queste collisioni generano sugli oggetti coinvolti. La fase di *Collision Detection* è molto importante perché, al fine di ricavare le leggi fisiche corrette per la scena che si sta simulando, tutti i contatti tra i vari corpi all'interno della simulazione devono essere trovati. Se ci sono N corpi all'interno di una scena allora si avranno $O(n^2)$ coppie di corpi da testare per rilevare le eventuali collisioni. Per evitare che il rilevamento delle collisioni diventi impraticabile a livello computazionale, questo viene suddiviso in fasi.

Le fasi che compongono la maggior parte delle pipeline che si occupano del rilevamento delle collisioni sono:

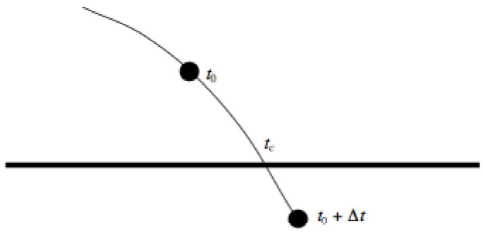
1. *Broad Phase*. Nella *Broad Phase*, eseguita in un sistema di coordinate globali o inerziali, i corpi o gli oggetti vengono approssimati da semplici primitive geometriche, permettendo l'esecuzione veloce dei calcoli di distanza fra i corpi, e quindi il rilevamento delle collisioni. La *Broad Phase* viene gestita solitamente tramite gerarchie di volumi limitanti, come ad esempio gli *AABB* (Axis Aligned Bounding Box), o le sfere di delimitazione. Attraverso le gerarchie di *Bounding Box* è possibile, quindi, ridurre il numero di coppie collisione/contatto da valutare, nello specifico, se due di questi riquadri di selezione non si sovrappongono, allora non sono necessari ulteriori controlli che coinvolgano il contenuto dei *Box*. Questo discorso vale in generale per qualunque primitiva si utilizzi per approssimare i corpi, o gli oggetti, presenti all'interno della simulazione. I *Bounding Box* sono la forma più semplice per gestire questa prima fase del processo di rilevamento delle collisioni. Il vantaggio di questa metodologia è che, in questo modo, se tutti gli oggetti presenti all'interno della simulazione sono contenuti all'interno dei *Box*, diventa relativamente semplice evitare che questi si sovrappongano e quindi le collisioni. Lo svantaggio, invece, è che nella maggior parte dei casi questi box non sono precisi, ovvero sono troppo grandi rispetto agli oggetti racchiusi al loro interno, e quindi, spesso, vengono generate delle simulazioni di bassa qualità. La *Broad Phase* si occupa dunque di ridurre il numero totale degli oggetti potenzialmente interagenti in base alla sovrapposizione dei volumi di delimitazione o limitanti.
2. *Narrow Phase*. Nella *Narrow Phase* vengono recuperate le geometrie all'interno dei volumi di delimitazione e utilizzate per determinare le caratteristiche dei corpi a contatto e la posizione dei punti di contatto. Questa è un'operazione costosa, quindi viene eseguita solo per le coppie di corpi per cui era stata trovata una sovrapposizione nella precedente fase di *Broad Phase*. Quando si ha a che fare con oggetti dall'aspetto complesso, ad esempio geometrie concave che collidono fra loro, il calcolo diventa molto complesso e dispendioso, proprio perché le intersezioni concave sono difficili da calcolare. Una delle soluzioni più popolari per risolvere questo tipo di situazione è quella di suddividere l'oggetto master in un insieme di oggetti più piccoli, tutti singolarmente convessi. Questa operazione, eseguita per il rilevamento delle collisioni, è detta *Convex Hull Collsion Detection*. *Bullet* esegue la *Convex Hull Collsion Detection* in modo molto robusto rispetto a *ODE*. Inoltre, questa metodologia, come vedremo in seguito proposta da *Erwin Coumans*, è ottimale quando si tratta di realizzare simulazioni di oggetti che devono essere fratturati. Un metodo generale per calcolare le sovrapposizioni tra oggetti convessi, quindi i punti vicini, e le distanze tra forme convesse, è chiamato *GJK* (Gilbert Johnson Keerthi), un algoritmo che permette di eseguire il passo di Narrow Phase consentendo la gestione di una vasta gamma di tipologie di forme di collisione. *GJK* funziona solo quando gli oggetti sono separati, è quindi necessario possedere un algoritmo che consenta il calcolo della sovrapposizione, come ad esempio l'*EPA* (Expanding Polytope Algorythm).

ODE (Open Dynamics Engine)

Il *Collision Detection* permette, quindi, di rilevare la compenetrazione tra i corpi che si muovono all'interno di un ambiente di simulazione. Per semplicità, si supponga di simulare il lancio di un punto materiale, quindi di una singola particella, verso un piano fisso.
Se si considera il piano flessibile, questo potrebbe essere deformato nei pressi in cui la particella va ad impattare. Se invece non si considera il piano flessibile si vuole che si verifichi alcuna compenetrazione. Questo significa che nell'istante in cui la particella viene a contatto con il piano, dovrà verificarsi un cambiamento improvviso della velocità della particella. Per un corpo flessibile, invece, tipo una palla di gomma, la collisione può essere considerata come un'occorrenza che si verifica in modo graduale. Quindi, su alcuni piuttosto piccoli, ma non nulli, intervalli di tempo, una forza potrebbe agire tra la palla e il piano e cambiare la velocità della palla. Durante questo intervallo di tempo la palla potrebbe deformarsi a causa di una forza. Più rigida è la palla, meno la palla potrà deformarsi, e più veloce questa collisione potrebbe occorrere. Nel caso limite, la palla è infinitamente rigida, e non può deformarsi, a meno che la velocità di discesa della palla venga interrotta istantaneamente, allora la palla potrebbe compenetrare un po' il piano.
Questo discorso è utile per capire che nella dinamica del corpo rigido le collisioni vengono considerate come occorrenze che si verificano istantaneamente.
Esistono due tipologie di contatto:

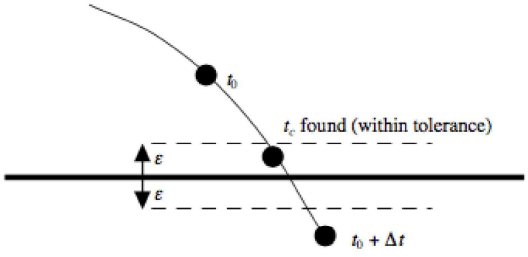
1. *Contatto Di Collisione*. Si verifica quando due corpi sono a contatto ad un certo punto p e hanno delle velocità che vanno una verso l'altro. Il contatto di collisione richiede un cambiamento istantaneo di velocità. Ogni volta che si verifica una collisione, lo stato di un corpo, il quale descrive entrambe posizione e velocità, subisce una discontinuità in velocità. Le routine numeriche che risolvono *ODE* lo fanno sotto l'ipotesi che lo stato $X(t)$ vari sempre senza problemi. Chiaramente, richiedendo che $X(t)$ cambi in modo discontinuo, quando si verifica una collisione, viola tale ipotesi. Per risolvere il problema, quindi, se si verifica una collisione al tempo t_c, viene detto al risolutore *ODE* di fermarsi. Viene poi preso lo stato in questo momento, $X(t_c)$, e viene calcolato come le velocità dei corpi coinvolti nella collisione devono cambiare. Lo stato che riflette queste nuove velocità è detto $X(t_c)^+$. Il solver numerico viene poi riavviato con il nuovo stato $X(t_c)$, e aggiornato per simulare in avanti partendo dal tempo t_c.
2. *Contatto di Riposo*. Ogniqualvolta i corpi si posano uno sull'altro ad un certo punto p, si immagini ad esempio una particella in contatto con il pavimento con velocità zero, si dice che i corpi sono a contatto di riposo. In questo caso, viene calcolata la forza che ostacola l'accelerazione verso il basso della particella; questa forza è il peso della particella causato dalla gravità, o qualsiasi altra forza esterna che agisce sulla particella. La forza tra la particella e il pavimento è detta forza di contatto. Il contatto di riposo chiaramente non richiede di arrestare e riavviare *ODE* per risolvere ogni istante.

Ci sono quindi due problemi da affrontare: il calcolo del cambiamento di velocità per un contatto di collisione e il calcolo delle forze di contatto che impediscono la compenetrazione. Per quanto riguarda la questione geometrica riferita al rilevamento di un contatto fra i corpi, si prenda il caso di una particella che cade su un pavimento. Mentre viene eseguita la simulazione, viene calcolata la posizione della particella a valori di tempo specifici man mano che questa si avvicina verso il pavimento.



(inter-penetration detected)

Si consideri la particella al tempo t_0, $t_0 + \Delta t$, $t_0 + 2\Delta t$ etc. e si supponga il momento della collisione t_c, in cui la particella colpisce effettivamente il pavimento, compreso tra t_0 e $t_0 + \Delta t$. Al tempo t_0, la particella si trova sopra il pavimento, ma al successivo passo, $t_0 + \Delta t$, la particella sta sotto il pavimento, il che significa che si è verificata una compenetrazione. Se si vuole interrompere e riavviare il simulatore al tempo t_c, sarà necessario t_c. t_c si trova quindi tra t_0 e $t_0 + \Delta t$. In generale, la risoluzione per t_c è difficile. Un modo semplice di determinare t_c è utilizzare il metodo numerico di bisezione.



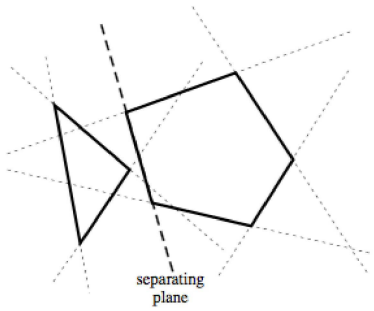
(inter-penetration detected)

Se al tempo $t_0 + \Delta t$ si rileva una compenetrazione, si informa il risolutore *ODE* che si vuole riprendere indietro al tempo t_0 e simulare in avanti fino al tempo $t_0 + \frac{\Delta t}{2}$. Se il simulatore raggiunge $t_0 + \frac{\Delta t}{2}$ senza incontrare una compenetrazione, si può determinare che il tempo t_c di collisione si trova tra $t_0 + \frac{\Delta t}{2}$ e $t_0 + \Delta t$. In caso contrario, t_c è inferiore a $t_0 + \frac{\Delta t}{2}$ e si cercherà di simulare da t_0 a $t_0 + \frac{\Delta t}{4}$. Il tempo t_c di collisione viene calcolato entro una certa appropriata tolleranza numerica. L'accuratezza con la quale viene trovato t_c dipende dalle routine di rilevamento di collisione. Le routine di rilevamento delle collisioni hanno un certo parametro ε. A questo punto si dichiara che la particella è in contatto con il pavimento. Il metodo di bisezione è un po' lento, ma è facile da implementare e abbastanza robusto.

Un metodo più rapido comporta effettivamente di predire il momento della collisione t_c sulla base della valutazione di $X(t_0)$ e $X(t_0 + \Delta t)$. Il processo di simulazione quindi consiste nel calcolo ripetuto della derivata del vettore di stato, $\frac{d}{dt}X(t)$, in vari tempi t. Il solver numerico *ODE,* un solver fisico come può essere *Bullet*, è responsabile della scelta dei valori di t in cui la derivata del vettore di stato deve essere calcolata. Per ogni simulazione ragionevolmente complicata, i valori di t scelti sono tali che il vettore di stato X non cambia notevolmente per valori successivi interpolati di t**.** Di conseguenza, vi è quasi sempre grande coerenza geometrica tra fasi temporali successive. Ad un passo $t_0 + \Delta t$ di tempo, l'idea è quella di sfruttare i risultati del rilevamento di collisione, calcolati in base al precedente passo temporale t_0.

I Poliedri Convessi

Si suppongano due poliedri convessi A e B. I due poliedri non avranno compenetrazione se e solo se esiste un piano di separazione tra di loro. Un piano di separazione tra due poliedri è un piano tale che ogni poliedro si troverà su un lato diverso del piano. Dato un piano, quindi, si può verificare che sia un piano di separazione verificando che tutti i vertici di A e B si trovino sui lati opposti del piano. In questo modo, un piano di separazione è una testimonianza del fatto che: due poliedri convessi non compenetrano. Se questo piano di separazione non esiste allora i poliedri si compenetrano. Un semplice modo per trovare inizialmente un piano di separazione è il seguente. Se una coppia di poliedri convessi è disgiunta o a contatto, ma non compenetrante, allora esiste un piano di separazione con la seguente struttura: o il piano contiene una faccia di uno dei poliedri, oppure il piano stesso contiene uno dei bordi appartenenti a uno dei due poliedri. Inizialmente, vengono controllate tutte le possibili combinazioni di facce e spigoli per vedere se tale combinazione forma un piano di separazione.



Anche se questo processo è inefficiente, viene svolto così di rado che l'inefficienza passa in secondo piano. Per i successivi passi temporali è necessario verificare che il piano di separazione della fase di tempo precedente, sia ancora valido. Una volta che il piano separazione è stato trovato, viene determinata la zona di contatto tra i due poliedri, assumendo che i poliedri non siano disgiunti. I punti di contatto tra i due poliedri possono avvenire solo sul piano di separazione. Dato il piano di separazione, i punti di contatto

possono essere rapidamente ed efficientemente determinati confrontando solo le facce, spigoli e vertici dei poliedri, che sono coincidenti con il piano di separazione.
Se non esiste alcun piano di separazione allora i due poliedri devono essere compenetranti. Quando due poliedri compenetrano, nella cache verrà memorizzato il testimone, o più, della compenetrazione. Dal momento che questo indica una collisione a qualche tempo prima, il simulatore verrà eseguito per tentare di calcolare $\frac{d}{dt}X(t)$ a qualche tempo prima. Quando il momento della collisione è determinato, la prima azione intrapresa sarà quella di verificare i testimoni presenti nella cache per vedere se questi indicano una compenetrazione. Si procede in questo modo fino a trovare il tempo di collisione.

Voronoi e Delaunay

Introduzione

Prima di procedere con la spiegazione delle fasi che compongono il processo di *Destruction,* che si sviluppa sulla pipeline *RBS* o *RBD,* è necessario aprire una parentesi per definire cosa sono i *Diagrammi Di Voronoi* e la tecnica della *Triangolazione Di Delaunay*. Questo è necessario perché il *Diagramma Di Vornoi* sarà uno strumento essenziale per poter eseguire la fase di rottura, o prefatturazione, della geometria all'interno di una simulazione di *Destruction*. Partendo dal *Diagramma Di Voronoi* è possibile ottenere la *Triangolazione Di Delaunay* definendo quindi una tecnica di triangolazione, in quanto il *Diagramma Di Voronoi* è il duale della *Triangolazione Di Delaunay*. Le *"triangolazioni"* sono delle griglie costituite esclusivamente da triangoli in $2D$ o da tetraedri in $3D$. Si supponga, ad esempio, di voler determinare le connessioni di un insieme di N punti P_i predeterminati. Esistono molti modi per connettere gli N punti in modo da formare una triangolazione, e la *Triangolazione Di Delaunay* è una tra le molte possibili. È inoltre necessario introdurre anche alcune definizioni importanti che saranno utili, appunto, per comprendere questi due argomenti:

1. Un *triangolo* è rappresentabile attraverso una terna ordinata di vertici P_i.
2. Il cerchio circoscritto ad un triangolo, o *circumcerchio*, ha un ruolo fondamentale negli algoritmi di triangolazione. Questo cerchio è completamente descritto dal centro, *circumcentro*, e dal raggio, *circumraggio*. Ci sono due modi per calcolare il *circumcerchio*, o si risolve un sistema lineare o si utilizza l'equazione del cerchio.
3. Il *tetraedro* è un poligono con quattro facce triangolari, ed è definito da una lista ordinata di quattro vertici P_i: $K = (P_1, P_2, P_3, P_4)$. Una faccia di K è una terna ordinata (sono permesse tutte le possibili permutazioni). Analogamente, gli spigoli di K sono implicitamente definiti come coppie ordinate. Ad ogni *tetraedro* è associata la sua *circumsfera*, ossia la sfera circoscritta ai suoi vertici. Il raggio, o *circumraggio*, e il centro, o *circumcentro*, di questa sfera possono essere ottenuti analogamente al caso bidimensionale, o usando un'equazione, oppure risolvendo un sistema di equazioni lineari per trovare il *circumcentro*.

4. Un insieme di punti S in $\mathbb{R}^d (d = 2 \, o \, d = 3)$ è in *posizione generale*, in due dimensioni, se non esistono tre punti collineari o 4 punti conciclici, cioè che appartengono alla stessa circonferenza. In tre dimensioni, significa che non devono esistere 4 punti complanari o cinque punti che appartengono alla stessa sfera.
5. Sia S un insieme di punti in *posizione generale* e sia Ω il dominio in $\mathbb{R}^d$ definito da $Conv(S)$.

Il diagramma di Voronoi

Alla fine del 19° secolo *Dirichlet* dimostrò che è possibile, per un insieme di punti in due dimensioni, dividere il piano in celle basandosi su dei criteri di vicinanza. Più tardi *Voronoi*, estese questo criterio allo spazio tridimensionale. Un *Diagramma Di Voronoi* di una serie di siti, o punti, è una collezione di regioni che dividono un piano o in generale uno spazio. Ogni regione corrisponde ad uno dei siti, e tutti i punti in una regione sono più vicini al sito corrispondente rispetto a qualsiasi altro sito. Tutte le *Regioni Di Voronoi* sono dei poligoni convessi, alcuni di loro sono infiniti, questi ultimi corrispondono ai siti *Convex Hull*. Il confine tra due regioni adiacenti è un segmento di linea, e la linea che le contiene è la bisettrice perpendicolare del segmento che unisce i due siti. Di solito, le *Regioni Di Voronoi* si incontrano tre alla volta nei *Punti Di Voronoi*. Se tre siti determinano le *Regioni Di Voronoi* che si incontrano in un *Punto Di Voronoi*, il cerchio che attraversa questi tre siti è centrato in quel *Punto Di Voronoi* e non ci sono altri siti nel cerchio. Quindi un *Diagramma Di Voronoi*, anche detto *Tasselatura Di Dirichlet*, è un particolare tipo di decomposizione di uno spazio metrico determinato dalle distanze rispetto a uno specifico insieme discreto di elementi dello spazio, ad esempio un insieme finito di punti. Il caso più semplice e comune è quello di un piano. Dato un insieme finito di punti S, il *Diagramma Di Voronoi* per S è la partizione del piano che associa una regione $V(p)$ ad ogni punto $p \in S$, in modo tale che tutti i punti di $V(p)$ siano più vicini a p che ad ogni altro punto in S. In ogni insieme discreto S di punti in uno spazio euclideo e per quasi ogni punto x, c'è un punto in S che è il più vicino a x. Il *"quasi"* è una precisazione necessaria dato che alcuni punti x possono essere equidistanti da 2 o più punti di S. In generale, l'insieme dei punti più vicini ad un punto $c \in S$ che ad ogni altro punto di S: è la parte interna di un politopo detto *Dominio Di Dirichlet* o *Cella Di Voronoi* di c. L'insieme di tali politopi è una tassellatura dell'intero spazio e viene detta *Tassellatura Di Voronoi,* corrispondente all'insieme S. Se la dimensione dello spazio è solo 2, è facile rappresentare graficamente la *Tassellatura Di Voronoi* ed è a questo caso che si riferisce solitamente l'accezione *Diagramma Di Voronoi*. Quindi dato S un insieme di n punti, o siti, P_i, $i = 1, \dots, n$ in dimensione d, una *Cella Di Voronoy*, $V(A)$, di un sottosinsieme A di S è il luogo dei punti di $\mathbb{R}^d$ equidistanti da ogni sito in A e più vicino a A che ad ogni altro sito che non è in A. Un *Diagramma Di Voronoi*, V, è una collezione di *Celle Di Voronoi*, non vuote, $V(A)$, per ogni sottoinsieme A di S. Le *Celle Di Voronoi* sono poligoni (poliedri in tre dimensioni) chiusi e convessi.

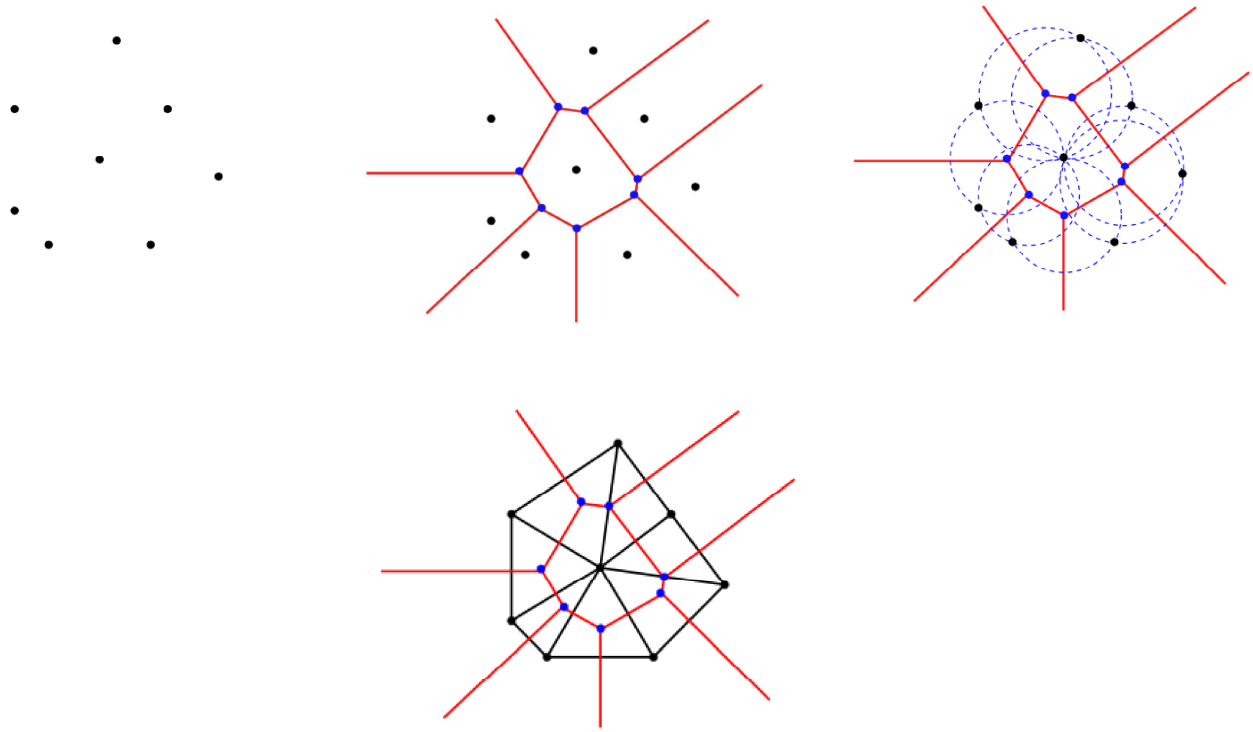
Un algoritmo semplice utilizzato per la costruzione del *Diagramma Di Voronoi* è quello che consente di inserire i vari punti uno alla volta all'interno del diagramma. Ogni volta che un nuovo punto viene inserito è necessario eseguire tre operazioni:

1. Capire quale delle *Celle Di Voronoi* esistenti contiene il nuovo sito, ovvero il punto.
2. Andare al confine della *Regione Di Voronoi* del nuovo sito e inserire i nuovi bordi all'interno del diagramma.
3. Cancellare i vecchi bordi che sporgono all'interno della nuova regione.

Nel caso peggiore il tempo complessivo speso da questo algoritmo è $O(n^2)$, se però i punti vengono inseriti in modo casuale, il tempo previsto sarà $O(n * \log(n))$ a prescindere da quale insieme di punti viene dato in input.

Da Voronoi a Delaunay

Si consideri un insieme di N punti P_i. Ad ogni punto P_i è associata una regione dello spazio V_i. L'unione dei lati di contorno delle regioni di V_i è detto *"Diagramma Di Voronoi"*.
Il lato del *Diagramma Di Voronoi* appartenente a due regioni V_i e V_j adiacenti è il luogo dei punti equidistanti da P_i e P_j e quindi giace sull'asse del segmento che connette i punti P_i e P_j. Il vertice del *Diagramma Di Voronoi* appartenente a tre regioni V_i, V_j e V_k è equidistante dai tre punti P_i, P_j e P_k e quindi coicide con il centro del cerchio passante per quei tre punti. Per ottenere la *Triangolazione Di Delaunay* dal *Diagramma Di Voronoi* è necessario introdurre la seguente definizione: due punti P_i e P_j si dicono *"contigui"* se le due regioni V_i e V_j hanno un lato di contorno in comune. La *Triangolazione Di Delaunay* si ottiene connettendo punti *"contigui"*.



Dualità per il caso 2D:

punto della triangolazione di Delaunay ⇔ regione di Voronoi
lato della triangolazione di Delaunay ⇔ lato di Voronoi
triangolo di Delaunay ⇔ vertice di Voronoi

In 3D le *Regioni Di Voronoi* sono dei poliedri delimitati da facce poligonali equidistanti da punti contigui. Le facce poligonali si intersecano lungo spigoli che sono equidistanti da tre punti contigui a due a due. Gli spigoli si incontrano in vertici che sono equidistanti da quattro punti, e quindi coincidono con il centro della sfera passante per quei quattro punti.

Dualità caso 3D:

punto della triangolazione di Delaunay ⇔ regione di Voronoi
spigolo della triangolazione di Delaunay ⇔ faccia di Voronoi.
faccia della triangolazione di Delaunay ⇔ spigolo di Voronoi.
tetraedro di Delaunay ⇔ vertice di Voronoi

La triangolazione di un insieme di punti

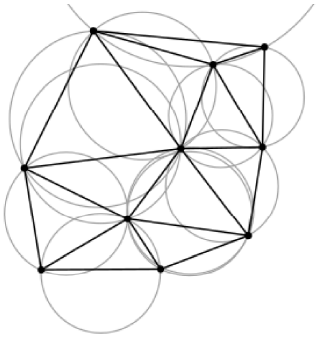
Una triangolazione di un insieme finito di punti $P \subset \mathbb{R}^2$ è una collezione $\mathcal{T}$ di triangoli. Dato un generico insieme di punti, esiste almeno una triangolazione. Proprietà:

1. Ogni insieme $P \subseteq \mathbb{R}^2$ di $n \geq 3$ punti ammette una triangolazione a meno che tutti i punti in P siano collineari.
2. Ogni triangolazione di un insieme $P \subseteq \mathbb{R}^2$ composto da n punti ha esattamente $3n - h - 3$ spigoli. Dove h è il numero dei vertici di $Conv(P)$.
3. Ogni triangolazione di un insieme $P \subseteq \mathbb{R}^2$ composto da n punti ha esattamente $2n - h - 2$ facce. Dove h è il numero dei vertici di $Conv(P)$.

La triangolazione di Delaunay

La *Triangolazione Di Delaunay* per un gruppo di punti P su un piano è una triangolazione $DT(P)$ tale che nessun punto appartenente a P sia all'interno del circumcerchio di ogni triangolo in $DT(P)$. Per un gruppo di punti su una stessa linea non esiste una triangolazione. Per un gruppo di quattro o più punti su una stessa circonferenza, ad esempio i vertici di un rettangolo, la triangolazione non è unica. Se si considera il caso di quattro punti non allineati, si scopre che esistono due possibili triangolazioni, ma che in generale solo una tra queste è: una *Triangolazione Di Delaunay* valida; se, invece, i quattro punti giacciono sulla stessa circonferenza, allora esistono due possibili *Triangolazioni Di Delaunay*. Partendo da un *Diagramma Di Voronoy*, è possibile ottenere la *Triangolazione Di Delaunay*. Sia $A \subset S$ e $V(A) \neq 0$ una *Cella Di Voronoy*. Si definisce *Cella Di Delaunay*, $T(A)$, la chiusura convessa

di A. Una *Triangolazione Di Delaunay* $\mathcal{T}$ è la collezione di tutte le *Celle Di Delaunay* $T(A)$, $\forall A \subset S$ con $V(A) \neq 0$.



La *Triangolazione Di Delaunay* possiede le seguenti proprietà:

1. Nella triangolazione ciascun punto è collegato al suo vicino più prossimo da un bordo.
2. Poiché la *Triangolazione Di Delaunay* è un grafo planare, si hanno al massimo $3n - 6$ bordi e al massimo $2n - 5$ triangoli, dove n è il numero di siti. Quindi, una volta costruita la *Triangolazione Di Dealunay*, se si vuole trovare la coppia più vicina di siti, bisogna solo confrontare $3n - 6$ coppie invece di tutte le possibili coppie.
3. La proprietà del cerchio vuoto afferma che se si disegna un cerchio attraverso i vertici di un qualsiasi *Triangolo Di Delaunay*, nessun altro sito sarà all'interno del cerchio.
4. Ogni *Triangolazione Di Delaunay* massimizza il più piccolo angolo interno tra tutte le triangolazioni possibili, ovvero l'angolo minimo di ogni *Triangolo Di Delaunay* è il più grande possibile.

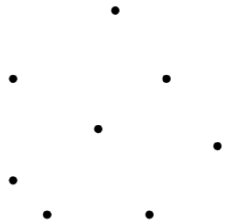
Sia S un insieme di punti. $\mathcal{T}_r$ è una *Triangolazione Di Delaunay* di $\Omega = Conv(S)$ se il circumcerchio o la circumsfera (aperta) associata ai suoi elementi è vuota. Il vantaggio della *Triangolazione Di Delaunay* consiste nella possibilità di generare la triangolazione *Convex Hull* di insiemi arbitrari di punti in modo efficiente e robusto. Per costruire la *Triangolazione Di Delaunay* si può disegnare un segmento di linea tra ogni coppia di siti le cui *Regioni Di Voronoi* condividono un bordo. Questa procedura suddivide il *Convex Hull* dei siti in triangoli. Esistono molti algoritmi per poter eseguire la *Triangolazione Di Delaunay*, ad esempio l'algoritmo di *Bowyer-Watson* e quello di *Green-Sibson* consentono di calcolare la *Triangolazione Di Delaunay* T_{n+1} di $n + 1$ punti a partire da una preesistente *Triangolazione Di Delaunay* T_n di n punti. Si tratta quindi di algoritmi che consentono di calcolare le modifiche da apportare ad una *Triangolazione Di Delaunay* preesistente per poter inserire un nuovo punto. Questi due metodi permettono inoltre di ottenere la *Triangolazione Convex Hull*, ovvero è possibile ottenere questo tipo di triangolazione a partire da quella di *Delaunay*.

La triangolazione Convex Hull

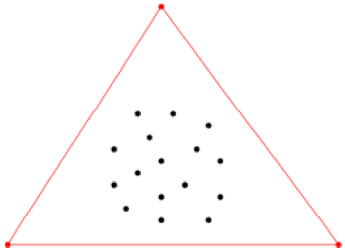
Per calcolare la *Triangolazione Convex Hull* di un insieme arbitrario di punti si possono eseguire questi semplici passi:

1. Si considera una triangolazione iniziale *"di servizio"* che contiene tutti i punti dell'insieme dato (ad esempio un unico triangolo).
2. Si inseriscono ad uno ad uno tutti i punti dell'insieme.
3. Si cancellano tutti i triangoli che hanno un vertice coincidente con uno dei tre vertici della triangolazione di servizio iniziale.

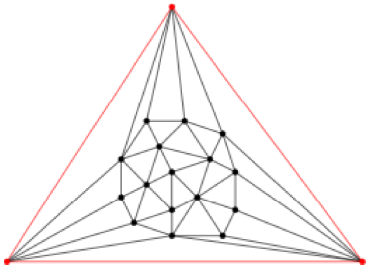
1) Dato un insieme di punti



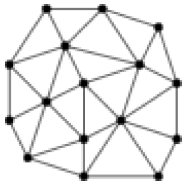
2) Si esegue una triangolazione di servizio



3) Si opera un'inserzione dei punti



4) Si crea il Guscio Convesso



Destruction

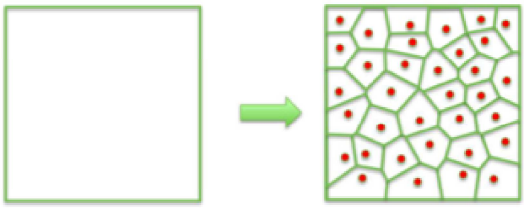
Fase 1: frattura degli oggetti

In questa prima fase che caratterizza l'approccio *RBS* di una pipeline di *Destruction* viene preparata la geometria, ovvero si decide come fratturarla e quindi come romperla: in quanti pezzi, la forma di questi pezzi, ecc. *Erwin Coumans*, nel 2011, al *SIGGRAPH* di Vancouver, in Canada, ha delineato quattro modi per frantumare un modello. Si tratta di una serie di metodologie che permettono di preparare la geometria sulla quale verrà eseguita la frattura. *Erwin Coumans* è il principale autore e sviluppatore di *Bullet*, il motore fisico open source utilizzato per lavori di rilevazione delle collisioni e per la gestione delle dinamiche dei corpi rigidi. *Erwin Coumans* ha creato la libreria fisica open source *Bullet* e le attività correlate ad essa e ha incoraggiato il suo utilizzo presso gli studi cinematografici, come la Dreamworks Animation, la Sony Pictures, la Digital Domain, le case di video game, come la Rockstar, la Disney Interactive and Sony Computer Entertainment e in molte altre aree. Prima di *Bullet* il motore fisico open source più popolare era *ODE (Open Dynamics Engine)*. I metodi di frantumazione elencati da *Coumans* sono i seguenti:

1. Diagrammi di Voronoi.
2. Tetraedralizzazione.
3. Operatori Booleani.
4. Decomposizione Convessa.

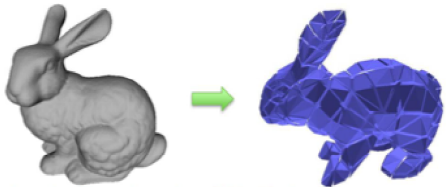
I diagrammi di Voronoi

Il *Diagrammma Di Voronoi* è uno dei metodi più utilizzati per avviare la fase di fratturazione che costituisce l'approccio *RBS* standard; esso deriva da un concetto matematico moderatamente semplice, che consente di produrre un aspetto naturale da conferire agli oggetti che verranno sottoposti all'operazione di frattura. I *Diagrammi Di Voronoi*, in sintesi, funzionano nel seguente modo: vengono create delle particelle all'interno del modello 3D, che possono essere generate in molti modi, incluso quello casuale, e poi, da questi punti, vengono costruite le *Regioni Di Voronoi* che li racchiudono; da queste regioni verranno poi generati i pezzi che andranno a costituire la geometria frantumata. È importante notare che la distribuzione dei punti iniziali è responsabile della forma e delle dimensioni di queste celle. Un modo largamente utilizzato per generare questi punti iniziali è eseguire una semplificazione efficace di un oggetto in una nuvola di punti, per poi produrre una sotto geometria costituita da un insieme di poligoni fratturati; questo è anche il metodo che verrà utilizzato per realizzare la frantumazione della geometria all'interno del progetto.



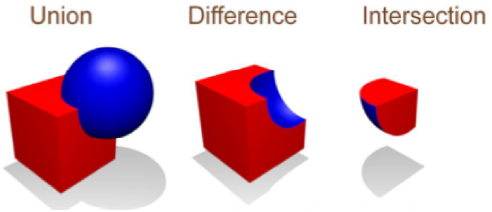
La tetraedralizzazione

La tetraedralizzazione permette di convertire un modello 3D in un insieme di tetraedri. Una delle metodologie più utilizzate per eseguire questa operazione è la *Triangolazione di Delaunay*. Suddividere la geometria in triangoli è un modo per poter frantumare un modello creando una rottura plausibilmente realistica.



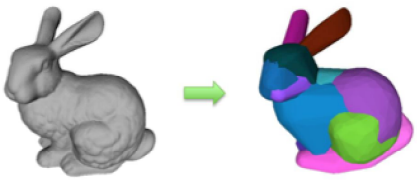
Gli operatori booleani

Il metodo che utilizza le operazioni booleane, anche conosciuto come approccio *CSG* (Constructive Solid Geometry), è un modo per eseguire operazioni volumetriche tra modelli 3D. Queste operazioni permettono di rompere il modello 3D in piccoli pezzi e quindi di poter eseguire la rottura della geometria. Questa tecnica si basa sull'utilizzo, anche congiunto, di semplici operazioni, come l'addizione, la sottrazione e la disgiunzione esclusiva di forme molto semplici (detta anche *XOR*, un operatore appartenente *all'algebra di Boole* o *algebra booleana*. L'or esclusivo è un operatore logico. L'operazione logica in questione restituisce il risultato vero se, e solo se, uno solo dei due operandi è vero).



La decomposizione convessa

Un altro modo per eseguire l'operazione di fratturazione si ottiene mediante la decomposizione convessa, che può essere eseguita dagli artisti manualmente, utilizzando semplici primitive convesse come box, sfere e capsule, o utilizzando degli strumenti automatici. Se ho una geometria complessa costituita da elementi concavi è possibile, tramite questo approccio, suddividerla in parti più piccole eliminando le concavità, quindi in elementi più piccoli tutti singolarmente convessi, e poi incollare tutti questi pezzi insieme in un unico corpo rigido combinato. Un modo che permette di eseguire questa operazione è proprio la decomposizione convessa.



Fase 2: settaggio dei vincoli e costruzione della dinamica di simulazione

Una volta che la geometria è stata preparata e fratturata in pezzi, questi hanno bisogno di essere vincolati insieme. Una geometria che segue l'approccio *RBS,* quindi il metodo del *Singolo Corpo Rigido Composito*, nel quale gli oggetti sono perfettamente rigidi prima e dopo la frattura, necessita di connessioni tra i vari pezzi, altrimenti questi rischierebbero immediatamente di allontanarsi, all'interno della simulazione, solo a seguito della forza di gravità. Esistono molti metodi per eseguire la seguente procedura. Una metodologia fra queste è quella di definire connessioni fra tutti i pezzi, quindi stabilire dei legami che mettono in relazione ogni pezzo ad ogni altro pezzi. Questa è una procedura che consente di raggiungere il massimo controllo peggiorando, però, le prestazioni, le quali possono essere molto rallentate a causa, appunto, della presenza delle molteplici connessioni. Un altro modo che permette l'esecuzione di questa operazione è il calcolo automatico delle connessioni in base al rilevamento delle collisioni: vengono calcolati i punti di contatto tra i pezzi adiacenti, e solo quando questo si verifica viene creata una connessione. È possibile, inoltre, creare una soglia di rottura per queste connessioni. Una volta che i pezzi sono stati incollati in un singolo corpo rigido, si è in grado di determinare il tempo di esecuzione della frattura. Se si verifica un impatto di collisione, viene calcolato l'impulso. Se l'impulso è maggiore di una certa soglia scelta, esso si propaga attraverso le connessioni, facendo in modo che questi legami si rompano o si indeboliscano. Questo metodo è stato molto utilizzato all'interno del progetto di *Destruction* attraverso l'uso di uno specifico solver, presente all'interno del software *Houdini*. Dopo aver definito la tipologia di legami da stabile tra i vari pezzi che compongono i singoli oggetti fratturati e quelli tra gli elementi appartenenti a oggetti differenti, è possibile, ulteriormente, definire e correggere la dinamica di simulazione al fine di ottenere il risultato desiderato.

Fase 3: esecuzione della simulazione e rilevamento delle collisioni

In questa terza fase viene avviato il processo di simulazione. In questo contesto viene messo in esecuzione il passo di *Collision Detection*, ovvero il rilevamento delle collisioni, un'operazione che consente di evitare il fatto che si verifichino delle compenetrazioni o che occorra un'errata dinamica tra gli oggetti che si muovono nella scena, all'interno della simulazione. Quindi, dopo aver rotto la geometria in pezzi e stabilito le connessioni tra questi, si procede con la simulazione, facendo molta attenzione alle collisioni che avvengono durante il tempo di simulazione. In una pipeline *RBS* o *RBD* è fondamentale ricordare che la fase di rilevamento delle collisioni è molto importante, in quanto, in questo contesto, la compenetrazione degli oggetti, se questi sono corpi rigidi, non è consentita.
In sintesi, in questa parte, la simulazione di *Destruction* deve essere eseguita sottostando a delle regole che sono state create in passaggi precedenti, quindi deve rispettare, ad ogni istante di tempo, queste regole facendo, però, attenzione alla compenetrazione degli oggetti, utilizzando il processo di *Collision Detection,* al fine di riprodurre una simulazione priva di errori.

I vincoli settati nella seconda fase del procedimento di *Destruction*, sono in sostanza delle equazioni e disequazioni che cambiano il modo in cui le coppie di corpi possono muoversi l'una rispetto all'altra. Dal momento che sono restrizioni cinematiche, influenzano anche le dinamiche. Mentre la cinematica è lo studio del moto senza interesse per le forze, i momenti, o le masse del corpo, la dinamica, invece, è lo studio di come le forze producono i movimenti. Quindi dal momento che i moti dinamici devono essere cinematicamente fattibili, la cinematica è un elemento essenziale della dinamica.
A proposito di forze, ne esistono di varie tipologie nella dinamica del corpo rigido: quella del vento e la forza di gravità sono solo alcune di esse. Tuttavia, le forze più difficili da affrontare, ma anche estremamente importanti nella simulazione interattiva, sono le forze di vincolo e di attrito. La forza di gravità agisce allo stesso modo su ogni particella costituente un corpo rigido. Tuttavia, la forza di gravità, si può mostrare come una singola forza di grandezza mg la cui linea di azione passa per il centro di massa del corpo. Questo perché l'effetto della gravità che agisce su un intero corpo è equivalente a una singola forza di grandezza mg che agisce nel suo centro di massa. Quando una coppia di corpi si scontra, quei corpi, e ogni altro corpo che si tocca, applica delle forze molto elevate di durata molto breve. Nel caso di corpi rigidi ideali, le grandezze di forza diventano infinite e la durata diventa infinitesimale. Queste forze sono indicate come forze impulsive o urti. Gli urti causano infinite accelerazioni, il che rende l'integrazione numerica diretta delle equazioni di Newton Eulero impossibile.

PhysBAM

PhysBAM è una libreria fisica specifica creata alla *Stanford University*. In particolare, è in grado di simulare corpi rigidi e deformabili, comprimibili e non, solidi e liquidi insieme, corpi rigidi e deformabili insieme, corpi rigidi e esseri umani articolati, fratture, fuoco, fumo, capelli, vestiti, muscoli e altri fenomeni naturali.
PhysBAM è il nome del progetto avviato nel 1998 da *Ronald Fedkiw*, la cui intenzione era di scrivere un codice che avrebbe potuto essere condiviso da tutti i vari progetti in cui era stato coinvolto in quel periodo. Nel 2000 *Fedkiw* arrivò alla *Stanford* e insieme a un gruppo di studenti impostò una struttura organizzativa per l'analisi numerica, il calcolo scientifico e la fisica basata sulla simulazione grafica. Nello specifico, *Fedkiw* e la sua squadra, si sono concentrati sulla progettazione di nuovi algoritmi di calcolo per una vasta gamma di applicazioni, tra cui la fluidodinamica computazionale e meccanica dei solidi, la computer grafica, la computer vision e la biomeccanica computazionale. L'*ILM (Industrial Light & Magic),* nell'ultimo decennio, è stato il principale utilizzatore di *PhysBAM*.
Alla *ILM*, per la quale *Fedkiw* è stato consulente, l'approccio *PhysBAM* è, come detto nella parte introduttiva, simile a quello *RBD* o *RBS*, in particolare, l'azienda può avvalersi di strumenti personalizzati come ad esempio il metodo degli insiemi di livello.
Il rilevamento delle collisioni del software di simulazione *PhysBAM* è tutto basato sull'approccio degli insiemi di livello; nello specifico utilizza la struttura dati degli insiemi di livello per eseguire il passo di *Narrow Phase Collision Resolution*.

Questo metodo è molto veloce, inoltre ha la proprietà di fornire ricerche veloci quando viene calcolata la distanza fra le particelle rispetto a una superficie geometrica, in sostanza, in qualsiasi punto dello spazio, si può dire esattamente quanto lontano dalla superficie ci si trova, costruendo quindi un buon modello consultativo di collisione. Il lato negativo, invece, è che ci vuole molto tempo per generare ed effettuare il calcolo, per di più, lo spazio di memoria di un insieme di livello aumenta in modo cubico.
Il sistema *RBS* o *RBD* creato alla *ILM,* basato sul metodo degli insiemi di livello, è in grado di generare una struttura dati volumetrica all'inizio della simulazione per gli oggetti che collidono correttamente, durante la simulazione verranno poi utilizzate le particelle al di fuori delle superfici dei corpi rigidi per valutare la profondità degli insiemi di livello degli oggetti che potrebbero interagire tra loro.
La struttura dati degli insiemi di livello usata da *PhysBAM,* divide il dominio spaziale della geometria in cellule di box, e per ogni cella memorizza la sua distanza rispetto alla superficie (phi). Dalla collezione di cellule vicine si può poi calcolare un gradiente di campo che produce il vettore normale che punta verso la superficie della geometria. Dalle normali e dalle funzioni phi dell'insieme di livello, si può calcolare la distanza esatta rispetto alla superficie della geometria. In questo modo, osservando i vettori e le normali, è possibile dire se si è all'interno o all'esterno di un oggetto quando si esegue il rilevamento delle collisioni. L'*ILM* utilizza il metodo degli insiemi di livello anche per eseguire la fase di fratturazione per le sequenze di *Destruction*. Nell'operazione di fratturazione vengono rappresentati un gruppo di oggetti detti *zero iso contours*, i quali rappresentano i punti in superficie dell'insieme di livello. Questo strumento, rispetto al *Voronoi*, che solitamente è quello più utilizzato per frantumare la geometria, non fa altro che prendere i punti *Voronoi* per poi trovare quelli che stanno effettivamente sulla superficie del modello, dal momento che il *Voronoi* normalmente non conosce la superficie. Il sistema che utilizza *ILM* ha la capacità di non dover ricreare la geometria in base all'insieme di livello, ma lo utilizza direttamente per ridurre l'elevata risoluzione della geometria.
Le macchine produrranno sempre dei limiti alle simulazioni, è per questo che l'approccio di simulazione che utilizza il metodo degli insiemi di livello è ottimo se si vuole simulare un milione di corpi, quindi ad esempio per realizzare simulazioni su larga scala, perché si tratta di simulare queste sequenze in più passaggi. Questo, quindi, può essere anche un modo per spingersi fino ai confini del realismo nel settore degli effetti cinematografici.

FEA (Finite Element Analysis)

Finite Element Analysis è l'approccio fisicamente più accurato per svolgere lavori di *Destruction*. È stato sviluppato nel 1943 da *Richard Courant,* e solo di recente ha trovato il suo spazio nel mondo degli effetti visivi, la ragione molto probabilmente è dovuta al fatto che solo ora la tecnologia utilizzata in questo campo è abbastanza all'avanguardia da permettere il suo uso. L'approccio ad elementi finiti, che si basa sulla meccanica dei continui, viene impiegato per risolvere le equazioni che governano la dinamica dei materiali elastici e quindi per simulare deformazioni e fratture.

Un'importante implementazione di FEA, è quella sviluppata da *Pixelux Entertainment* all'interno del suo sistema *DMM (Digital Molecular Matter)*, il quale utilizza appunto l'approccio ad elementi finiti per realizzare simulazioni di corpi morbidi. A differenza dei motori di simulazione tradizionali che lavorano in tempo reale, basati sulla cinematica del corpo rigido, *FEA* consente a *DMM* di simulare un vasto insieme di caratteristiche fisiche, anche molto velocemente. Questo consente di far si che l'oggetto si comporti come farebbe nel mondo reale, permettendo quindi di raggiungere il realismo di simulazione. *FEA* utilizza un complesso sistema di punti, chiamati nodi, per formare una griglia 3D chiamata mesh, programmata per contenere le proprietà strutturali e materiali che definiscono in che modo la struttura di un oggetto reagirà a determinate condizioni di carico. Ai nodi della mesh viene quindi assegnata una certa densità a seconda dei livelli di carico previsti per una particolare area. Di conseguenza le aree sottoposte a grandi quantità di carico avranno una densità di nodo superiore rispetto a quelle in cui verrà esercitato poco o nessun carico. Da ogni nodo della mesh, poi, si estende un altro elemento che si collega a ciascuno dei nodi adiacenti, formando la rete di vettori che esercita le proprietà del materiale dell'oggetto preso in considerazione. Tutto questo, però, complica l'approccio rendendolo molto costoso a livello computazionale.
In FEA il modello 3D viene approssimato da un insieme di elementi, solitamente da tetraedri. Questa approssimazione viene utilizzata ad esempio per il rilevamento delle collisioni. Nello specifico, in FEA, il rilevamento delle collisioni viene eseguito proprio sulla mesh, ad esempio tetraedrica, che avvolge completamente l'oggetto. Il *Tet Collision Detection,* ovvero il rilevamento delle collisioni operato fra tetraedri, è molto più semplice rispetto a quello eseguito fra elementi convessi. L'utilizzo dei tetraedri, o dei tet, è molto più facile rispetto all'utilizzo dell'approccio convex hull, inoltre il rilevamento delle collisioni in *FEA* è migliore di quello convex hull.
Per quanto riguarda la fase di creazione dei vincoli, i tetraedri che compongono la mesh, vengono collegati fra loro fino a formare un grande gruppo che possiede un punto nodale comune, per il quale vengono definite una certa forza e una robustezza strutturale al fine di determinare una soglia, che, se viene superata, fa si che la connessione si rompa.
Per quanto riguarda la fase di frantumazione della geometria, invece, è utile ricordare che in *FEA* non si esegue una fase di prefratturazione, quindi non è necessario alcun elemento che svolga questa operazione, come ad esempio fa il *Voronoi* nell'approccio *RBS* o *RBD*, che quindi rompa l'oggetto prima di simularne la rottura, in quanto la frattura vera e propria si basa solo sulla fisica, quindi più realistica, facendo in modo che i pezzi risultanti dalla rottura di un oggetto siano quelli che si otterrebbero da una rottura reale. Con *FEA* la frattura dell'oggetto si esegue durante lo step di simulazione. In sintesi non è necessario che l'artista *VFX* progetti la frattura, deve solo definire in modo adeguato le forze interne agli oggetti, permettendo così di effettuare una frattura efficace ed efficiente.
Concludendo, si può affermare che *FEA*, nonostante sia più complesso da gestire a livello computazionale, consente di accelerare l'intero flusso di lavoro di una pipeline di *Destruction* costruendo delle simulazioni di grande qualità.

Capitolo due: gli strumenti

Introduzione

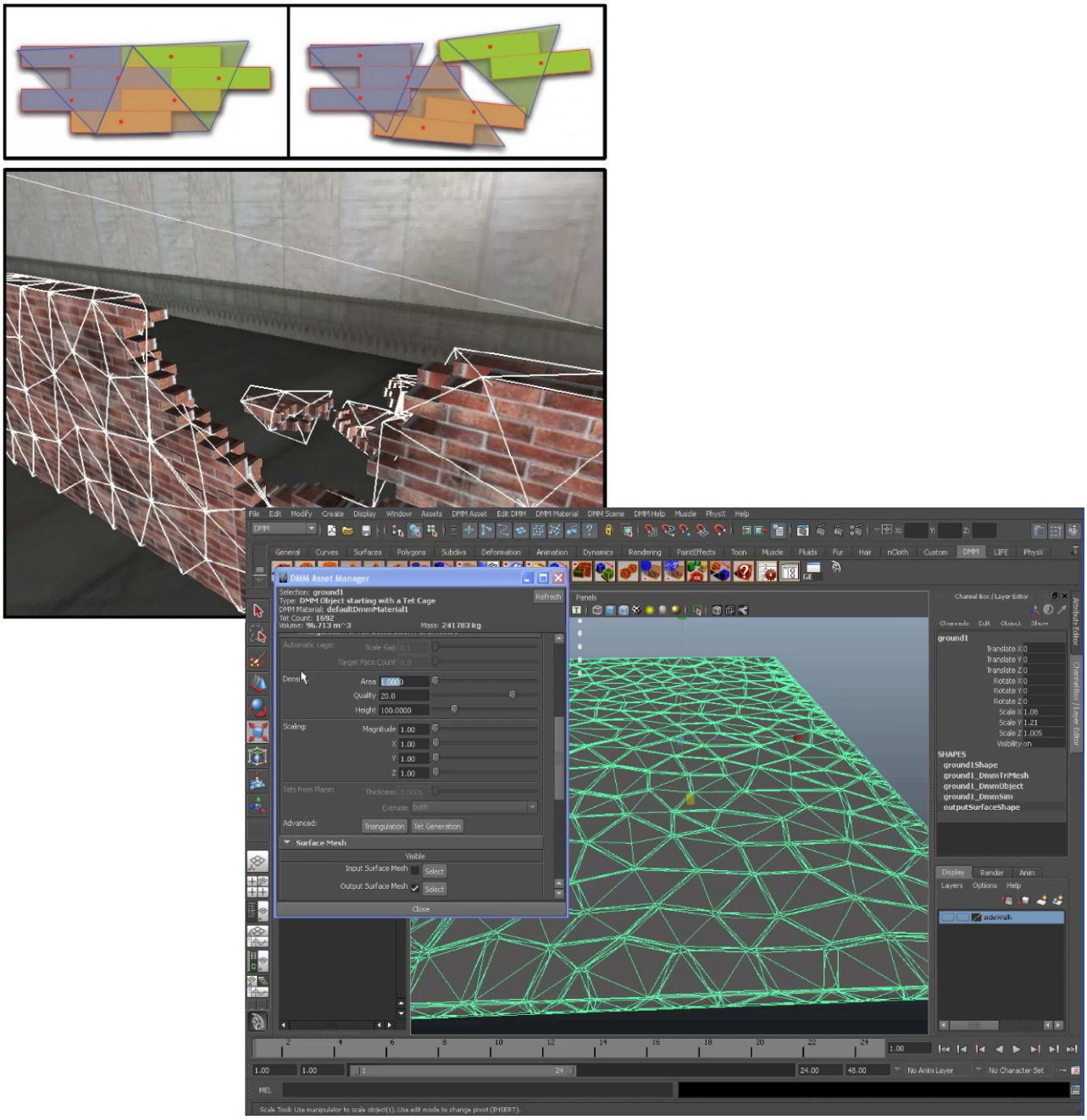
In questo secondo capitolo, chiamato *"gli strumenti"*, verranno approfonditi alcuni tool, attualmente a disposizione sul mercato, che consentono di effettuare lavori di *Destruction*. I software o plugin trattati in questa sezione, sono quelli più utilizzati dai più importanti studi di *VFX* in quanto, molto spesso, sono degli applicativi integrati all'interno dei più famosi software utilizzati in questo ambito, come and esempio *Maya*, o *Houdini*. Quello che gli *"artisti"* del *Destruction*, ma non solo loro, vogliono da questi strumenti sono, come dice *Peter Kyme*, stabilità, velocità e flessibilità. Che quindi sono diventati anche gli aspetti principali sui quali la progettazione di questi applicativi cerca di fare attenzione e quindi cerca di sviluppare. Nello specifico, gli strumenti trattati in questa sezione sono i seguenti:

1. *DMM (Digital Molecular Matter)*
2. *Fracture FX*
3. *Momentum*
4. *Rayfire*
5. *Realflow*
6. *ThinkingParticles*

Per ciascuno di essi viene descritta, anche brevemente, il tipo di approccio utilizzato, la storia, il confronto con *Houdini*, le caratteristiche principali e il prezzo, un aspetto, quest'ultimo, che comunque va tenuto in considerazione. Dopo una panoramica dettagliata su alcuni degli strumenti di *Destruction* disponibili, attraverso una comparazione, verranno raccontate le caratteristiche del software *Houdini*, i suoi aspetti principali e motivi per i quali è stato scelto per affrontare lo sviluppo del progetto oggetto di questo testo. Sempre in questo contesto, verrà aperta poi una parentesi su *Bullet,* in quanto è la libreria fisica che *Houdini* sfrutta all'interno delle dinamiche di simulazione.

DMM (Digital Molecular Matter)

Nel febbraio del 2004, a Ginevra, viene fondata la *Pixelux Entertainment S.A.*, il cui intento era quello di sviluppare una tecnologia che avrebbe potuto automatizzare la creazione di asset, ovvero attività, per i videogiochi, attraverso delle simulazioni avanzate. In questo contesto, viene creato il plugin *DMM (Digital Molecular Matter)*, ovvero una tecnologia per la gestione di simulazioni realtime della fisica dei materiali, il cui team esecutivo era composto da *Mitchell Bunnel*, *Vikram Sohal* e *Eric Parker*, divenuti gli attuali amministratori delegati della compagnia. *DMM* è stato utilizzato per la prima volta nel mondo degli effetti visivi cinematografici nel 2011 per il film *Avatar*, diretto da *James Cameron*.
DMM è un plugin avanzato che sfrutta il metodo *FEA* per la creazione delle sequenze di *Destruction*. Un aspetto chiave della tecnologia *FEA* è che gli oggetti, o i corpi, all'interno della simulazione, possono comportarsi proprio come farebbero nel mondo reale
A differenza del metodo *RBS* o *RBD*, quindi della fisica dei corpi rigidi, nel quale il comportamento degli oggetti, o dei corpi, è spesso gestito ingannando l'utente, in *FEA,* questo comportamento, viene regolato da dei parametri che vengono collegati al modello e al suo materiale, che quindi consentono il controllo della realtà fisica, dalla plasticità alla durezza, dalla preservazione della forma a molti altri parametri, conferendo un alto grado di veridicità alla simulazione.
DMM, in sostanza, permette la creazione di simulazioni fisiche realistiche, una caratteristica molto rilevante perché di fondamentale importanza per il suo sviluppo e il suo successo e che, inoltre, ha consentito la realizzazione del gioco *Star Wars: The Force Unleashed*, per il quale *DMM* è stato creato: era necessaria una tecnologia che consentisse la creazione di simulazioni realtime che avrebbero potuto cambiare a seconda delle azioni intraprese dal giocatore. Questo funzionamento, nello specifico, è stato riportato nel documento *"Real-Time Deformation and Fracture in a Game Environment"*, che descrive, appunto, il sistema di simulazione sviluppato per modellare deformazioni e fratture di oggetti solidi in un contesto di gioco in tempo reale.
Il plugin *DMM* è diventato negli anni il cuore della pipeline di *Destruction Kali* all'interno della *MPC*, un'azienda coinvolta nel settore della produzione degli effetti visivi cinematografici. *Kali* permette la creazione di simulazioni basate su eventi. Questo risulta particolarmente utile nella realizzazione di simulazioni secondarie o nella generazione di particelle. Questa metodologia permette di rilevare quando un evento di simulazione si verifica e quando termina. Nello specifico, *DMM* consente la suddivisione del modello sottoposto a rottura mediante una mesh tetraedrica. In *Kali,* infatti, l'età di qualsiasi tetraedro va da zero a un termine, che può essere un qualsiasi valore, quando questo valore viene resettato, tornado a zero, si verifica che la parte del modello corrispondente al tetraedro il cui valore è stato resettato si rompe. È importante, però, che qualunque modello sottoposto alla metodologia *FEA* venga costruito nel modo corretto, al fine di superare il processo di fratturazione e consentire una reazione corretta alle forze per riprodurre la realtà fisica.



DMM utilizza un approccio completamente diverso rispetto a quello *RBS* o *RBD* utilizzato da *Houdini* per la gestione del progetto oggetto di questa tesi, quindi è difficile poterli mettere a confronto. Nonostante questo si possono fare alcune considerazioni generali: *DMM* è molto più facile da utilizzare e da apprendere, in quanto è molto più intuitivo rispetto ad *Houdini*, contiene al suo interno una grande libreria che permette la gestione di diverse tipologie di materiali, consente di lavorare in maniera procedurale; rispetto a *Houdini* il processo di creazione dei vincoli, quindi delle connessioni fra i pezzi appartenenti all'oggetto che è stato fratturato, è molto più semplice, in quanto, con *DMM*, basta selezionarli da un menù, inserirli all'interno del modello e se necessario modificarli a seconda del risultato che si vuole raggiungere attraverso la creazione di regioni passive e attive, un passaggio, questo, che con *Houdini* sarebbe più lungo e non così intuitivo.

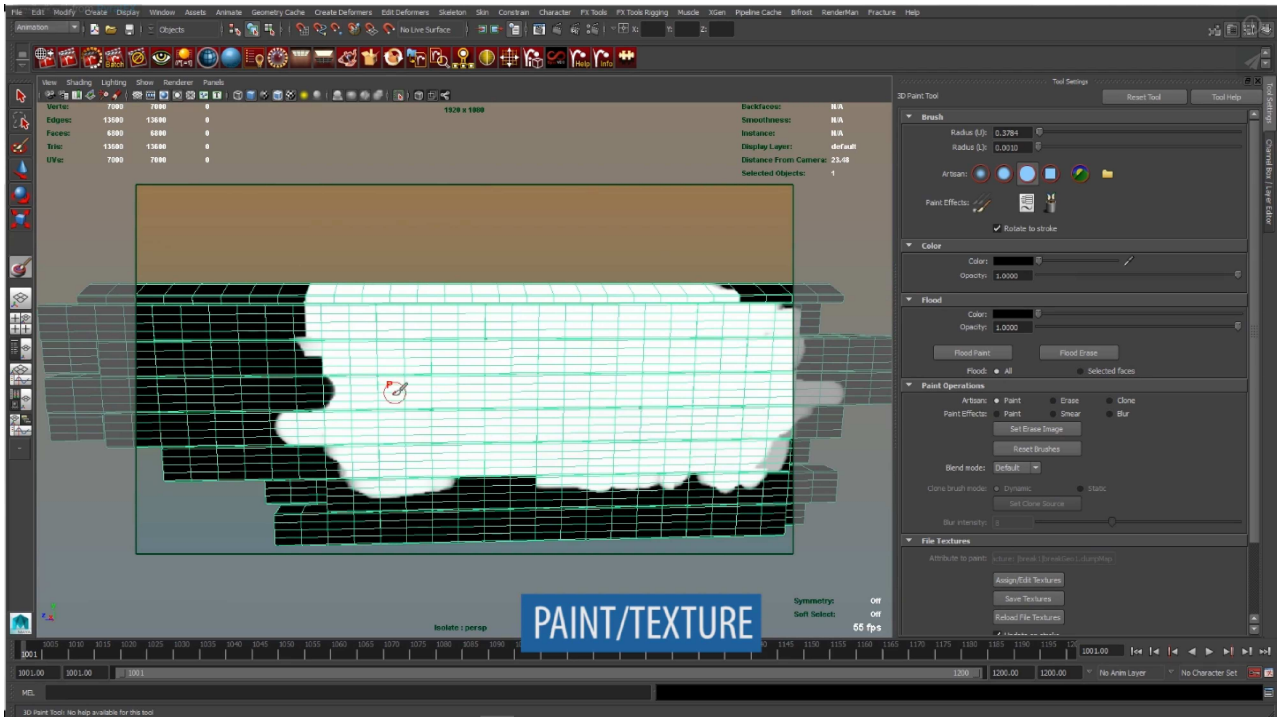
La caratteristica che forse è in grado di esprimere al meglio le potenzialità e le qualità di *DMM* è la naturalezza del comportamento dei materiali che produce delle simulazioni realistiche a livello fisico, cosa che non sarebbe possibile tramite l'approccio *RBS* o *RBD* utilizzato da *Houdini*.
Per quanto riguarda il prezzo, la versione più recente del plugin *DMM* per *Maya*, utilizzato per fare *Destruction*, ha un costo di *500$.* Anche in questo caso non è possibile comparare il prezzo con quello del software *Houdini,* in quanto quest'ultimo non è solo un plugin ma un vero e proprio software, completo, che quindi consente la realizzazione di ogni sorta di simulazione, non solo quelle legate al *Destruction*.

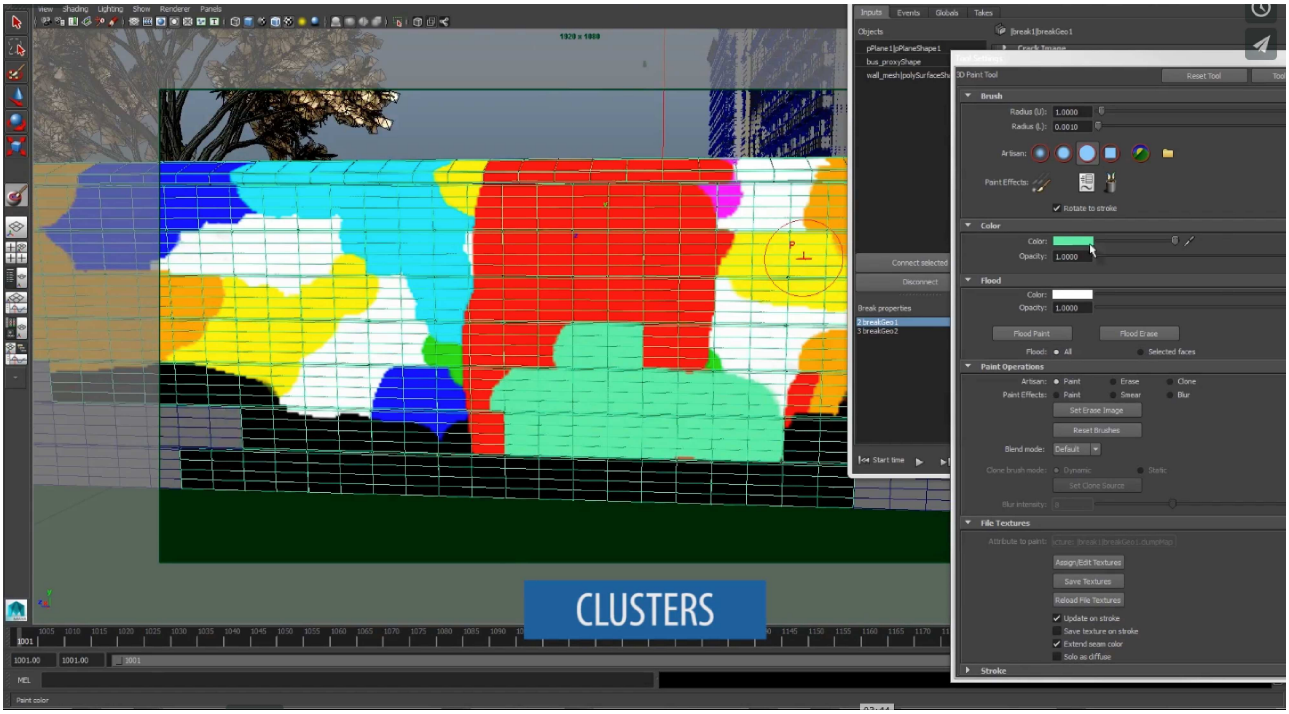
Fracture FX

Fracture FX è uno strumento, basato sull'approccio *RBS* o *RBD,* utilizzato per fare *Destruction*. Per realizzare una simulazione di *Destruction* con *Fracture FX* è necessario seguire alcuni determinati passaggi, che a livello concettuale non sono altro che le fasi, precedentemente spiegate, che compongono la pipeline *RBS* o *RBD* di *Destruction*:

1. Rompere l'oggetto, o gli oggetti, che dovrà, o che dovranno, essere simulato, o simulati.
2. Creare una simulazione, come se fosse anch'essa un oggetto.
3. Connettere l'oggetto alla simulazione.
4. Settare le proprietà dell'oggetto.
5. Settare le proprietà dell'oggetto rotto.
6. Settare le proprietà del solver all'interno della simulazione, ovvero quello che si dovrà occupare del rilevamento delle collisioni.

Fracture FX è una soluzione cross platform per *Autodesk Maya* che consente, anche a chi si approccia per la prima volta al suo utilizzo, di prendere subito il completo controllo delle simulazioni in modo semplice e veloce. *Fracture Fx* utilizza una simulazione basata su eventi e inoltre è costruito su un'architettura di tipo procedurale.
L'applicativo, rispetto ad *Houdini,* è molto più facile, veloce e intuitivo, come d'altronde tutti i plugin dedicati a questo genere di lavorazioni, in quanto il loro compito è semplificare e velocizzare il flusso di lavoro. Come *Houdini*, *Fracture FX* utilizza il solver *Bullet* per gestire la dinamica del corpo rigido e quindi anche il rilevamento delle collisioni. Proprio perché utilizza lo stesso approccio, ovvero quello della fisica dei corpi rigidi, *Fracture FX* può confrontarsi in maniera molto più leale rispetto al precedente *DMM*, che invece utilizza il metodo dell'analisi ad elementi finiti.
Fracture FX consente l'utilizzo di quattro differenti tipologie di *Voronoi* per poter eseguire l'operazione di frattura, questo consente, rispetto ad *Houdini*, di svolgere questa fase in modo molto più semplice e veloce, per esempio è possibile creare una rottura attraverso l'utilizzo di un'immagine. Inoltre, a questo proposito, il software possiede diversi strumenti per il controllo preciso della frattura, anche su aree localizzate, ovvero zone di interesse che si vogliono distruggere magari evitando, ad esempio, di romperne il contorno.





Per quanto riguarda il prezzo, la versione più recente del plugin *Fracture FX* per *Maya*, utilizzato per fare *Destruction*, ha un costo di circa 400$.

Momentum

Exocortex Momentum 3 è un simulatore multifisico che opera ad alta velocità, uno strumento utilizzato per la realizzazione di simulazioni di *Destruction*. *Momentum* fornisce un grande ambiente di simulazione semplice ed accessibile, nel quale è possibile creare simulazioni affidabili in breve tempo, lasciando quindi più spazio alle eventuali modifiche e correzioni.
Exocortex Momentum 3 introduce la frattura al suo già vasto motore multifisico. Nello specifico, il motore che gestisce la fratturazione degli oggetti è stato progettato per fornire agli *"artisti"* il pieno controllo di questa operazione. Si parla quindi di una frattura flessibile, in quanto il motore che gestisce l'operazione di fratturazione supporta ad esempio la rottura dinamica, ovvero quella che si verifica in risposta alle collisioni, tipica di un approccio come può essere quello *FEM*, e la prefratturazione statica, quindi specificata dall'utente, ovvero quella tipica dell'approccio *RBS* o *RBD*.

Exocortex Momentum 3, inoltre, offre quattro nuovi mezzi, implementati su richiesta degli utilizzatori, che consentono un controllo più preciso delle simulazioni:

1. *Controlli ICE*. Validi per tutti gli elementi di simulazione; questi controlli consentono agli *"artisti" VFX* di realizzare più facilmente le loro idee.
2. *Force Sculpting*. È un modo molto semplice e intuitivo di dirigere le esplosioni attraverso gli *ICE*. Quindi le forze vengono scolpite attraverso la definizione di una mesh deformata, le cui dimensioni rappresentano le varie forze che agiscono in ciascuna direzione. Basta quindi modificare la mesh per poter cambiare le forze che agiscono nella parte di mesh che è stata modificata.
3. *Motion Modifier*. Questo modificatore consente la copia del movimento di un oggetto sulle parti di un oggetto che viene simulato.
4. *Time Warping*. Tramite questo strumento è possibile modificare il tempo di una simulazione al fine di creare specifici effetti, come ad esempi lo slow motion, il bullet time e la riproduzione all'indietro.

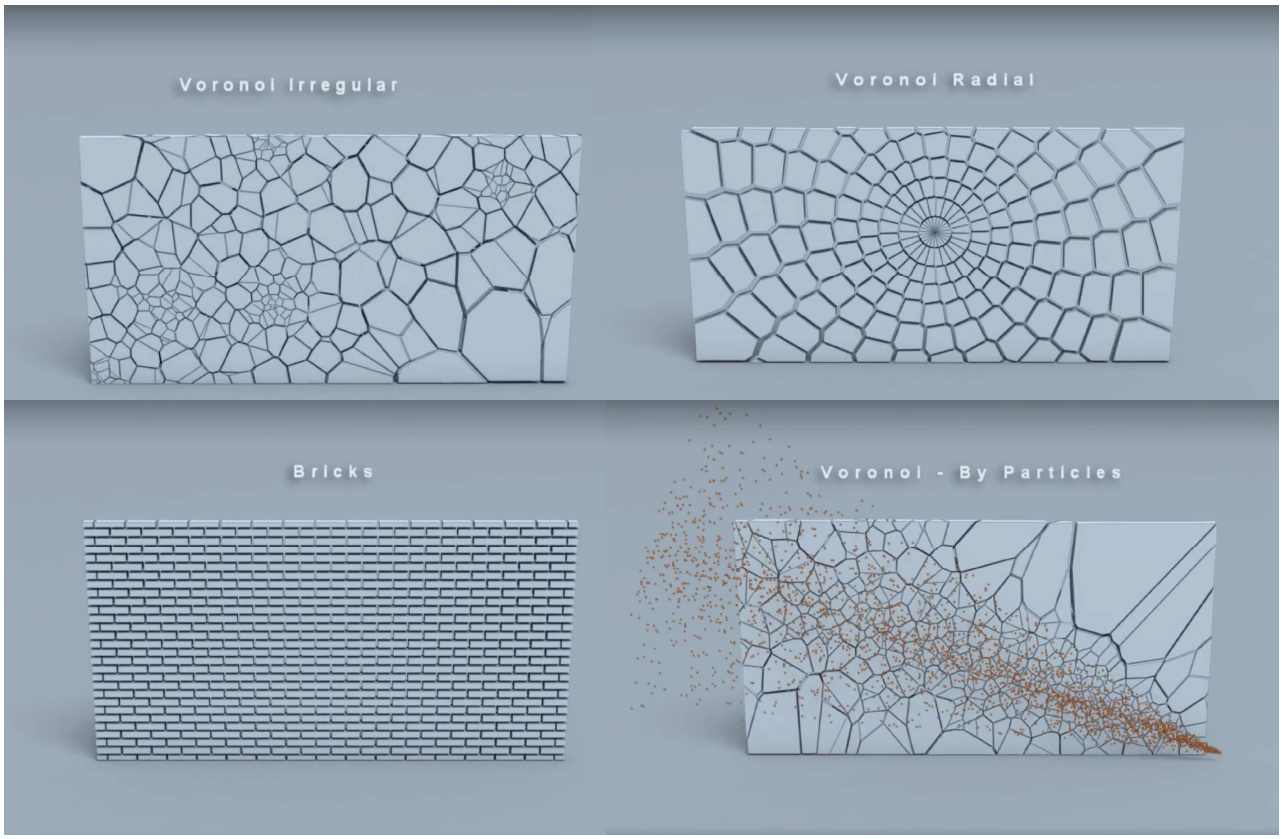
Momentum (quando si utilizza l'approccio *RBS* o *RBD*, come è stato fatto per *Houdini)* si basa sulle caratteristiche del solver *Bullet*, che poi possono essere estese, per aggiungerne di nuove, per gestire ad esempio le aree da sottoporre a frattura, i controlli delle simulazioni, la qualità del *Rendering* e il flusso di lavoro in generale. *Momentum 3,* rispetto a *Houdini,* permette di realizzare la fase di creazione dei vincoli in modo molto più facile e veloce, in quanto è semplicemente sufficiente selezionare gli oggetti da vincolare e il tipo di vincolo che si vuole applicare. Dal punto di vista grafico, *Momentum 3* ha un'interfaccia vecchio stampo, nella quale, però, i settaggi risultano ben chiari, questo lo rende, appunto, più intuitivo rispetto ad *Houdini,* che invece risulta molto più complesso. *Momentum* consente un controllo molto più preciso delle simulazioni utilizzando, inoltre, un sistema nodale, il quale consente un approccio procedurale, come appunto avviene anche all'interno del software *Houdini*. Per ciò che concerne il prezzo, l'ultima versione del software *Exocortex Momentum 3,* disponibile per *Autodesk Softimage,* ha un costo di circa 500$.

RayFire

RayFire è un plugin, sviluppato per *Autodesk 3ds Max,* che consente la progettazione di scene di *Destruction* in maniera molto semplice e intuitiva rispetto a quello che si potrebbe fare con *Houdini*. *RayFire*, appunto, è un software molto efficiente, ideato con l'intento di migliorare il flusso di lavoro, rendendolo più veloce. Rispetto ad *Houdini, RayFire* ha un'interfaccia, delle funzionalità e delle impostazioni molto facili da imparare e da utilizzare, e forse sono proprio questi i tratti distintivi che lo caratterizzano. Il flusso di lavoro all'interno di *RayFire* è reso così semplice grazie ai modificatori di cui esso dispone, in particolare, ad esempio, possiede degli strumenti specifici che sono adatti a realizzare differenti tipologie di fratture, cosa che in *Houdini* non è presente, o meglio, all'interno di *Houdini* si possono ottenere gli stessi risultati ma in modo più lungo e complesso, proprio per il fatto che non esistono degli strumenti dedicati come appunto avviene in *RayFire*. In sostanza, è grazie ai

modificatori, gli elementi che lo contraddistinguono, che *RayFire* riesce a rendere il tutto più immediato da realizzare. Alcuni dei modificatori che implementano le funzionalità di *RayFire* sono:

1. *Cluster*. Questo modificatore permette il raggruppamento di frammenti in insiemi più complessi. In *Houdini* questa funzionalità è implementabile all'interno del nodo di *Voronoi*, nonostante questo *Rayfire* è molto più semplice da utilizzare.
2. *Bricks*. Questo modificatore è uno strumento molto avanzato che *Houdini* non mette a disposizione, almeno non in questa forma, già implementato. In *Rayfire*, questo tool permette di scegliere una vasta gamma di opzioni per il controllo del layout dei mattoni, o in generale per muri fatti con una struttura simile.
3. *Voronoi*. Questo modificatore è molto più immediato da utilizzare rispetto a quello presente in *Houdini* grazie alla presenza di impostazioni molto più semplici da gestire e selezionare. A differenza di *Houdini,* in *Rayfire,* il *Voronoi* lavora in real time in modo interattivo, mentre quello offerto da *Houdini* è interattivo solo quando la geometria è semplice.
4. *Asperity*. Questo modificatore permette di aggiungere un ulteriore livello di dettaglio ai semplici frammenti, conferendone grande realisticità.
5. *Cracks*. Questo modificatore consente di realizzare delle fratture, molto realistiche, per gli oggetti che sono costituiti da un materiale rifrangente, come ad esempio il vetro.
6. *Trace*. Questo modificatore permette all'utente di disegnare una frattura attraverso l'utilizzo di immagini che vengono sovrapposte all'oggetto che si vuole sottoporre al *Destruction*.

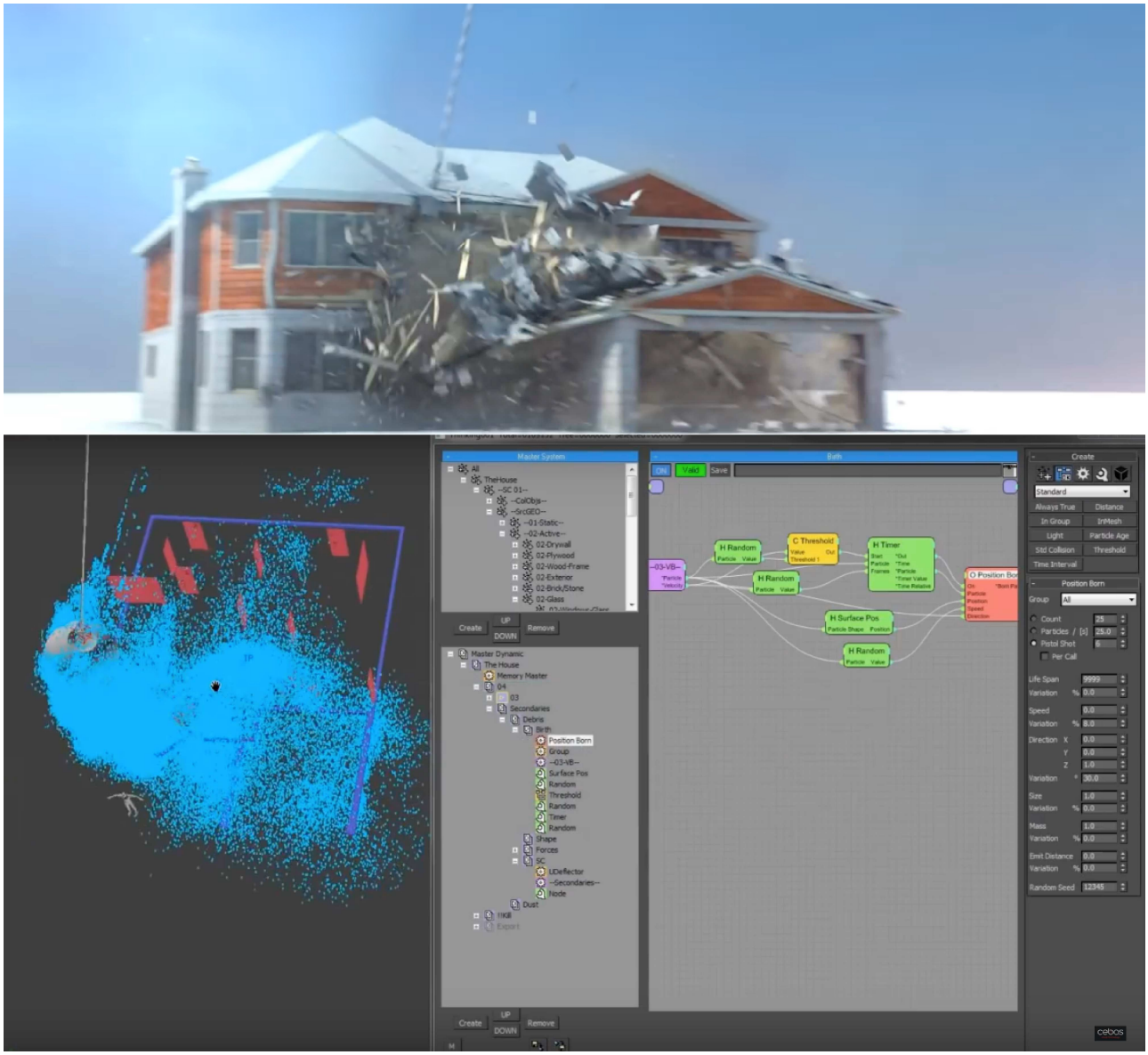


RayFire, inoltre, possiede un sistema di demolizione, o distruzione, interattivo che consente di realizzare simulazioni dinamiche e di *Destruction* in accordo con il materiale e la forza di collisione degli oggetti. Come per *Houdini*, anche *RayFire* può lavorare con la fisica dei corpi rigidi all'interno delle simulazioni di *Destruction*, gli oggetti vengono quindi influenzati da forze, ad esempio, e vengono creati dei vincoli che possono essere rotti o indeboliti. Proprio come *Houdini, RayFire* consente di simulare diverse tipologie di oggetti, come ad esempio quelli dinamici, statici, animati cinematicamente, a riposo, ecc.

Per quanto riguarda il prezzo, la versione più recente del plugin *Rayfire* per *Autodesk 3ds Max* ha un costo di circa 400$.

thinkingParticles

thinkingParticles è uno dei plugin di punta, per *3ds Max,* della *Cebas Visual Technology*.
La struttura architetturale di *thinkingParticles* è stata sviluppata con l'intento di offrire un sistema particellare completamente integrato, potente e aggiornabile in grado, appunto, di produrre effetti particellari avanzati di grande spettacolarità, basati su un elevato grado di precisione fisico.
Il software offre la possibilità di creare infinite simulazioni combinate che coinvolgono la dinamica dei fluidi, dei corpi morbidi e dei corpi rigidi, fino ad ottenere dei risultati che raggiungono elevati livelli di realismo.
L'obiettivo della *Cebas* è sempre stato quello di fornire agli *"artisti" VFX* e all'industria dell'animazione un plugin capace di soddisfare qualsiasi esigenza che fosse allo stesso tempo potente, sofisticato, fisicamente accurato e quindi indispensabile. Tutto questo si incarna nell'ultimo aggiornamento dell'applicativo alla versione 6.4.
Grazie al suo completo approccio procedurale e il suo elevato livello di accuratezza fisico raggiunto all'interno delle simulazioni, *thinkingParticles* è ampiamente conosciuto e utilizzato da i più importanti studi cinematografici coinvolti nella realizzazione degli effetti visivi, ma non solo anche dal settore della pubblicità e dei videogame. A questo proposito, le pellicole più recenti che sfruttano il software per la loro realizzazione sono: *White house down, After earth, Star trek: into the darkness, Oblivion, Die hard 5: a good day to die hard, Snow white and the huntsman, The avengers, Twilight saga: breaking dawn II, Battleship, Red tails, Harry Potter and the deathly hallows: part I and II, Transformers 3D, Green lantern, Thor 3D, 2012, Alice in wonder land* e molti altri. Mentre per quanto riguarda il settore dei videogame, alcuni titoli che hanno fatto uso di *thinkingParticles* all'interno delle loro produzioni sono: *League of legends, The witcher 2, Classic transformers, Command and conquer, Diablo, Star craft* e *Tomb Raider.*
thinkingParticles consente agli utenti di costruire sistemi particellari non basati su eventi, ma piuttosto su regole e condizioni, e soprattutto non legati a fattori che sfruttano il tempo per attivare specifici comportamenti.
Per quanto riguarda le simulazioni di *Destruction* e l'utilizzo dell'approccio *RBS* o *RBD,* anche in questo contesto, all'interno di *thinkingParticles,* viene sfruttata la libreria fisica *Bullet*. Siccome *Bullet*, in origine, è stata sviluppata all'interno di un contesto che è quello dei videogame, ha dei grossi limiti nel produrre delle simulazioni che risultino realistiche, per questo, in *thinkingParticles,* le sue librerie sono state riscritte per, appunto, essere in grado di raggiungere una soddisfacente accuratezza visiva in grado di produrre un elevato livello di realismo, nello specifico sono state riscritte le parti che riguardano la fisica delle collisioni e della dinamica dei corpi rigidi.
Per poter eseguire l'operazione di fratturazione, all'interno di *thinkingParticles* è presente il nodo *volumeBreak*, il quale fa in modo che, anche se i pezzi dell'oggetto fratturato vengono spostati, il volume totale del corpo rimanga invariato. Il software è in grado di agire sulle parti che vengono sottoposte a frattura con molta accuratezza, aumentando la velocità di reazione del software.



Anche in questo caso, le differenze con *Houdini* stanno nella semplicità dell'accesso alle impostazioni, *thinkingParticles* consente l'esecuzione dei lavori di *Destruction* in modo molto più veloce perché di più facile utilizzo, inoltre, dispone di un'interfaccia abbastanza chiara e pulita, meno confusionale rispetto a quella di *Houdini.*
Per quanto riguarda il prezzo, la versione del plugin più recente di *thinkingParticles* per *3ds Max* ha un costo di circa 540$.

RealFlow

RealFlow è un software molto potente utilizzato per realizzare ogni sorta di simulazione, comprese quelle di *Destruction*. In particolare, *RealFlow* è indicato per lo più per la creazione di simulazioni di fluido, infatti consente di ottenere elevate performance in questo genere di effetti raggiungendo alti livelli di simulazione e semplificando il flusso di lavoro.

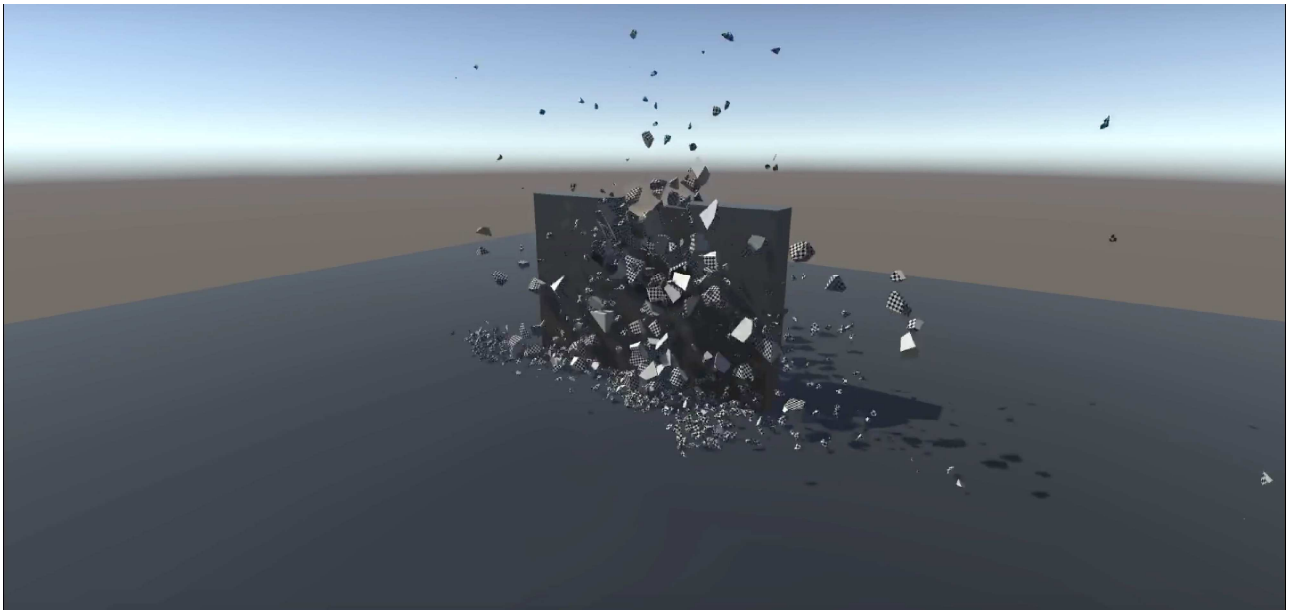
RealFlow, inoltre, è uno strumento completamente integrato, veloce e facile da utilizzare, compatibile con tutte le maggiori piattaforme 3D:

Houdini 3D ANIMATION TOOLS · MAYA · CINEMA 4D · 3DS MAX · LightWave · SOFTIMAGE

RealFlow si compone di diversi solver, ciascuno dei quali si occupa di un compito specifico. *Dyverso* è appunto il solver che consente di ottenere il meglio nelle simulazioni di fluido, facilitando il flusso di lavoro. *Dyverso*, essendo potenziato *Maxwell Render*, il più avanzato motore fisico di render, permette di raggiungere un alto livello di realismo: basta semplicemente importare la scena, aggiungere il fluido e poi sfruttare *Maxwell* per renderizzare il tutto.

Per la realizzazione delle simulazioni di fluido su media e grande scala, *RealFlow* mette a disposizione il solver *Hybrido (HyFLIP),* che offre infinite possibilità. Il solver di fluido *Hybrido (Hybrid Large Dimension Liquid Solver),* di *RealFlow,* da la possibilità di simulare qualsiasi cosa, dalle inondazioni alle scene in cui è presente l'oceano, queste sono delle scene molto difficili da ricreare con l'emettitore di particelle tradizionale di *RealFlow*. Con *Hybrido*, appunto, gli *"artisti"* dei fluidi sono in grado di simulare scatti impressionati, come oceani con onde che si infrangono, enormi inondazioni, o il viaggio di navi attraverso acque turbolenti durante un'enorme tempesta. *Hybrido* permette inoltre di ottenere dei filp di simulazione molto velocemente, un rilevamento più accurato delle collisioni e una migliore gestione della memoria, specialmente quando si ha a che fare con geometrie complesse e oggetti che si muovono velocemente.
A differenza di *Houdini*, che sfrutta il solver *Bullet* per gestire la dinamica dei corpi rigidi, *RealFlow* si appoggia sul solver *Caronte,* che implementa, inoltre, la dinamica dei corpi morbidi. Per poter conferire le capacità dinamiche agli oggetti è necessario attivare la funzione *"Dynamics"* dal menù situato all'interno del software. Attraverso questa funzione è possibile scegliere se creare un corpo rigido passivo, o attivo, oppure un corpo morbido. Una volta operata questa scelta viene aperto un nuovo pannello attraverso il quale è possibile regolare tutte le proprietà fisiche dell'oggetto appena creato. Una delle caratteristiche più sofisticate, è la possibilità di creare un'iterazione fra solver diversi, si possono quindi far interagire fra loro corpi morbidi e rigidi all'interno di una simulazione che contiene, ad esempio, anche dei fluidi. All'interno di *RealFlow* è proprio il solver *Caronte,* che consente la simulazione dei fenomeni di *Destruction* e non solo, anche esplosioni e deformazioni di strutture complesse.
Per quanto riguarda gli oggetti non deformabili, dei quali si vuole simulare il comportamento e il movimento, esistono due opzioni:

1. È possibile creare un corpo rigido passivo, che descrive un oggetto il cui moto non è influenzato dagli altri elementi all'interno della scena, questo però non significa necessariamente che un corpo rigido passivo debba essere immobile. Nel pannello che gestisce i parametri della fisica di un corpo rigido passivo, alcuni di essi, come ad esempio la massa e la velocità, non sono accessibili. Questo perché possono essere considerati oggetti con massa infinita nei quali non sono presenti velocità iniziali e rotazioni e, inoltre, non richiedono un centro di gravità.
2. Quando, invece, viene creato un corpo rigido attivo sono presenti tutti i gradi di libertà del caso, possono essere spostati e sostituiti, hanno una velocità iniziale e una rotazione, un centro di gravità che previene il suo ribaltamento e vere impostazioni fisiche, come l'elasticità e l'attrito che influiscono non solo sul comportamento del corpo ma anche sulla credibilità.



Per quanto riguarda l'uso dei corpi prefratturati per simulare i fenomeni di *Destruction*, *RealFlow* offre diverse possibilità, nello specifico diversi strumenti e funzioni per rompere un oggetto in tanti pezzi e collegare questi elementi fino a quando non si verifica un evento che rompe queste connessioni. Rispetto ad *Houdini*, *RealFlow* offre quattro metodi per realizzare differenti tipologie di rotture, come il legno, il vetro, il cemento ecc. Per ricollegare i singoli pezzettini, originati dalla frattura dell'oggetto, usa un sistema detto *MultiJoint* che permette a questi pezzi di mantenere la loro disposizione e la posizione originale. Nello specifico, il set di strumenti di frattura di cui *RealFlow* dispone è costituito da quattro applicazioni distinte:

1. *Voronoi Uniform*
2. *Voronoi By Steering Geometry*
3. *Voronoi By Points*
4. *Voronoi Radial*

Il grande vantaggio è che questi metodi creano un solo oggetto detto *MultiBody*. I singoli pezzi vengono quindi raggruppati in un singolo *MultiBody*, più facile da gestire, che è in grado di ripercuotere tutte le modifiche che l'utente effettua su di esso su tutti gli elementi che contiene al suo interno. Il processo di frammentazione è molto facile da controllare e da avviare rispetto a quello che si dovrebbe seguire in *Houdini,* basta selezionare l'oggetto che si vuole frammentare, avviare lo strumento *Fracture* dalla barra degli strumenti e poi cliccare su ok. Come avviene anche per *Houdini* il processo in questione non è distruttivo, gli oggetti originali vengono conservati.

Esiste un'opzione particolare, per alcuni strumenti di frattura, che consente di ottenere effetti particolari. Questa opzione è detta *"Apply globally"*, e può essere applicata solo su un *MultiBody*. Questo consente di non dover necessariamente applicare il *Voronoi* per ogni oggetto singolarmente, ma di propagare la rottura fra oggetti diversi, all'interno di un *MultiBody*. Questo parametro può essere utilizzato per la creazione delle schegge di legno, in quanto attualmente gli strumenti di frattura offerti da *RealFlow* non forniscono una modalità dedicata a questo tipo di frammentazione. In linea di principio è la stessa operazione che viene svolta su *Houdini* per ottenere il medesimo risultato, ma su *RealFlow* può essere eseguita in meno passaggi: l'idea è quella di ridurre, in questo caso, il *MultiBody* lungo un asse, applicare la frammentazione a livello globale e poi ingrandire il *Multibody* fino alla dimensione originale dell'oggetto di riferimento.
Il *MultiBody* non viene solo utilizzato nelle operazioni di fratturazione, ma è anche molto comodo quando si deve importare centinaia, o migliaia, di oggetti all'interno della scena, in quanto il processo di importazione diventa molto più veloce. All'interno di un *MultiBody* gli oggetti esistono ancora, mantenendo proprietà come il volume e la forma. I *MultiBody*, come qualsiasi oggetto, possono essere definiti come corpi rigidi attivi, o passivi, o corpi morbidi. La differenza sostanziale tra un oggetto normale e un *MultiBody* è che: non esiste una massa ma un parametro detto densità. Quando un oggetto viene creato o importato, *RealFlow* determina il suo volume e calcola la sua massa da questo valore. La densità è definita come la massa per unità di volume o, in unità *SI*, chilogrammo per metro cubo. Dal momento che *RealFlow* conosce il volume di ogni nodo all'interno di un *MultiBody*, e la densità è data come $\frac{1000\ kg}{m^3}$, per impostazione predefinita, è facile calcolare le masse individuali degli oggetti. Nel pannello delle impostazioni di un *MultiBody* è sufficiente aumentare o diminuire il parametro *density* per alterare le masse dei nodi presenti al suo interno.
Per quanto riguarda, invece, la gestione dei vincoli, il cui campo di applicazione principale è la gestione degli oggetti prefratturati, quindi quello del *Destruction*, in *RealFlow* viene utilizzato il *MultiJoints*. Su un *MultiJoints* possono agire delle forze, se la forza con cui vengono tenuti insieme i pezzi è maggiore rispetto a una possibile forza esterna che invece cerca di attirare i pezzi a sé, come quella di gravità, allora i pezzi rimarranno attaccati insieme, altrimenti alcune di queste connessioni, o tutte, si romperanno. Anche in questo caso la gestione dei vincoli diventa molto più semplice rispetto a quello che si potrebbe fare su *Houdini,* con i suoi strumenti interni.
Il prezzo della versione standard di *RealFlow* si aggira all'incirca sui 1000 euro, mentre quello riferito al solver *Caronte*, utilizzato per i *Destruction*, intorno ai 95 euro.
In questo caso il prezzo si riferisce, come nel caso di *Houdini,* all'acquisto di un software completo, più specializzato su una determinata tipologia di effetti, ovvero quello di fluido, ma comunque confrontabile in maniera molto equa con le possibilità offerte da *Houdini*.
Caronte essendo un solver che facilità parecchio le operazioni di *Destruction* è ciò che lo contraddistingue da *Houdini*.

Houdini

L’azienda

Per quasi trent'anni la *SideFX* ha provveduto a fornire agli *“artisti”* degli effetti visivi strumenti progettati per poter raggiungere risultati cinematografici di altissima qualità.
La *SideFX* è un’azienda leader nella produzione di software procedurali sin dal 1978. In principio *Kim Davidson* e *Greg Hermanovic* hanno dato origine alla *Omnibus*, una società pioneristica nel mondo, allora emergente, della computer grafica. In questo contesto hanno dato il via allo sviluppo di un software proprietario per la realizzazione degli effetti visivi, tanto che nel 1987 crearono la *Side Effects Software* e rilasciarono *PRISMS*, un’applicazione di grafica procedurale che gettava le fondamenta per la costruzione di *Houdini*. Da allora la *SideFX,* con *Houdini*, ha incominciato a prendere parte agli effetti visivi di molte pellicole cinematografiche, e non solo, vincendo molti premi per l’eccellente lavoro svolto dal suo team di sviluppo. La *SideFX*, nello specifico, è stata premiata dalla *Academy Motion Picture, Arts and Sciences,* per ben tre volte, per la grande tecnologia, basata su un sistema procedurale, offerta da *Houdini*, ricevendo, inoltre, numerosi premi, come l’*Academy Awards,* per la grande spettacolarità degli effetti visivi prodotti dal questo software. Questi sono dei grandi riconoscimenti per l’enorme lavoro svolto dalla *SideFX*, che non smette mai di investire nella ricerca e nello sviluppo, prestando, inoltre, molta attenzione alle parole e alle richieste dei propri utilizzatori, direzionando, in questo modo, la sua crescita e le sue esigenze produttive.
Mentre l’innovazione fa salti da gigante, la *SideFX* può dirsi orgogliosa del ruolo di leadership assunto negli anni in questo settore così frenetico e della sua costante ricerca di modi per sostenere il processo creativo. Alla *SideFX* si da valore all’innovazione, all'integrità e all'impegno andando incontro a tutte le esigenze del cliente.
Date le profonde radici nella produzione, l’azienda e il suo team sanno che cosa ci vuole per far si che gli *“artisti” CG* riescano ad ottenere il meglio dai loro lavori, per questo sono costantemente attivi nel cercare di aiutare i propri clienti a raggiungere i propri traguardi.
La *SideFX* è, inoltre, impegnata nella crescita delle conoscenze, come dimostra la versione *Houdini Apprentice* che dà, appunto, essendo il suo utilizzo gratuito, agli *“artisti”* digitali la possibilità di costruire le proprie competenze e abilità e creare quindi i propri contenuti. Questo è un modo che l’azienda utilizza per coltivare talenti, che in questo modo possono arrivare preparati quando entreranno nella vera realtà lavorativa della produzione.

Il software

Houdini è un software sviluppato dalla *Side Effects*. Attualmente è quello più utilizzato per gestire le simulazioni che coinvolgono la dinamica dei corpi rigidi. Nonostante la sua complessità architetturale, e spesso anche di utilizzo, è un software completo. *Houdini* fornisce controllo, flessibilità e velocità, e per questo, a differenza dei tool dedicati disponibili in commercio, come quelli di *Destruction* trattati in questo capitolo, viene adottato come parte del flusso di lavoro dalle migliori aziende che lavorano nel settore degli effetti visivi. Il bisogno di raggiungere risultati sempre più al limite del realismo in tempi sempre più brevi, porta gli *"artisti" VFX* ad avere la necessità di utilizzare degli strumenti sempre più al passo con queste esigenze. *Houdini*, in questo contesto, velocizza il processo produttivo e da grande spazio alla creatività.
Houdini, soprattutto dopo il rilascio della versione 11, ha conquistato una grande porzione del mercato delle aziende che operano nel mercato dei *VFX*. Con la versione 12 ha introdotto grandi miglioramenti: l'implementazione di *Bullet* ha fatto si che il suo successo si espandesse in modo ancora più esplosivo. La fratturazione tramite *Voronoi*, introdotta nella versione 11, e l'integrazione di *Bullet,* inserita nella versione 12, sono solo alcuni dei miglioramenti implementati su suggerimento dei clienti e utilizzatori del software. Con la versione 12, e poi quelle successive, sono stati introdotti dei forti miglioramenti anche a livello prestazionale, infatti, ora, la dinamica dei corpi rigidi è in grado di gestire grandi quantità di oggetti, insieme, all'interno di una stessa simulazione.
Uno dei punti di forza dell'ambiente dinamico di *Houdini*, ovvero quello che corrisponde alle simulazioni, è il fatto che a livello progettuale è stato pensato per poter gestire più solver tra loro interagenti in modo naturale ed intuitivo, quindi, ad esempio, è possibile far interagire dei fluidi con dei corpi rigidi. Come molti strumenti di questo tipo, anche *Houdini* punta, non solo al raggiungimento di un maggiore realismo di simulazione ma anche ad un incremento delle performance, come anche alla capacità di gestire sempre più grandi insiemi di dati.
In *Houdini*, è possibile rimuovere, aggiungere e riorganizzare i nodi in qualsiasi momento, il che rende davvero facile fare modifiche. I sistemi basati su nodi, come *Houdini*, sono sistemi procedurali con un flusso di lavoro non distruttivo. Procedurale significa quindi che viene impostato un processo al posto di apportare delle modifiche statiche di fila, mentre non distruttivo è una conseguenza dell'approccio procedurale, significa pertanto che qualsiasi cambiamento effettuato può essere invertito in qualunque momento all'interno di questo processo.
Bullet e l'approccio *RBS* o *RBD* nativo di *Houdini* offrono alcuni vantaggi agli "artisti" che li utilizzano, tra questi vi sono la capacità di gestire forme geometriche complesse, intersezioni non facili e condizioni iniziali complesse, e l'aumento delle prestazioni, che consente una gestione più flessibile delle reti di vincoli. L'integrazione di *Bullet* in *Houdini* è veramente ottima, oramai è diventato un sistema stabile.

Per quanto riguarda il rilevamento delle collisioni, *Houdini*, nello specifico, utilizza un numero di differenti tecniche a seconda del tipo di geometria e delle condizioni che si verificano nel sistema fisico. La soluzione primaria per il rilevamento delle collisioni offerta dal software utilizza il metodo *SDF (Signed Distance Fields),* comunemente usato nel metodo *LSM (Level Set Methods)* per rappresentare i dati volumetrici. Per eseguire la frattura dinamica *Houdini* si avvale del *Voronoi,* strumento che permette di calcolare velocemente le interazioni, gestire bene le parti concave e consentire una direzione artistica illimitata. L'operatore di *Vornoi* all'interno di *Bullet* lavora in modo trasparente con la pipeline *RBS* o *RBD* e il motore di render. Il metodo *RBS* o *RBD,* è normalmente indeformabile. Mentre questo è vero in generale, il sistema *RBS* o *RBD* presente in *Houdini* è in grado di gestire anche corpi rigidi deformabili. Per deformare una geometria, *Houdini* ricalcola il *SDF* per ogni fotogramma, facendo in modo che le collisioni e le forze di impatto vengano calcolate correttamente per tutte le forme che devono essere deformate. *Houdini* permette, inoltre, anche l'implementazione della plasticità, dell'elasticità e delle proprietà di strappo da poter inserire all'interno di alcune simulazioni di corpo rigido deformabile.
Per il rilevamento delle collisioni *Convex Hull* è fondamentale un modello, o una mesh, che circonda un oggetto in cui sono state rimosse tutte le parti concave. Il modello diventa quindi una forma convessa, quindi più gestibile, in quanto la fase di *Collision Detection* fra oggetti concavi è molto più complessa rispetto a quella che utilizza un oggetto convesso. La procedura *Convex Hull* è attualmente molto efficiente e stabile all'interno di *Houdini*. *Houdini* permette di eseguire l'operazione di rilevamento delle collisioni anche tramite il metodo *LSM* che rispetto al metodo *Convex Hull* è più lento. Il metodo *LSM* è utile quando si tratta di capire se si è all'interno o all'esterno dell'oggetto, le prestazioni comunque, in questo caso, non sono di qualità e in generale è un metodo, sia a livello di tempo sia di scala, non veloce.
Houdini include, inoltre, una vasta gamma di opzioni e strumenti di modellazione per la suddivisione delle superfici. L'approccio procedurale in *Houdini* presente consente, tra le altre cose, di creare scenari complessi, come ad esempio degli ambienti.
Con *Houdni* si ha anche la possibilità di animare qualsiasi parametro attraverso l'utilizzo dei keyframe, questa opzione è stata impiegata, ad esempio, per muovere gli oggetti utilizzati per rompere la struttura del modello del palazzo e generare il crollo all'interno del progetto oggetto di questo testo; questi oggetti non sono stati renderizzati per questioni di stile, in quanto si voleva generare un'esplosione, ricreata a parte attraverso un'altra simulazione differente da quella di *Destruction* in cui era presente il crollo.
Per quanto riguarda il discorso dell'illuminazione, *Houdini* fornisce un ambiente flessibile e potente per la gestione delle luci, la costruzione di shader e la definizione del look di uno scatto. Per il processo di *Rendering*, *Houdini* può contare su *Mantra,* un robusto e integrato motore di *Rendering,* oppure su un supporto personalizzato utilizzato per sfruttare una vasta gamma di motori di render di terze parti. *Houdini* include un sistema di *Compositing* basato sui nodi e gode anche di un importante proprietà, ovvero l'interoperatibilità: importazione ed esportazione di una varietà di formati, tra cui *Alembic*, *FBX, OBJ, OpenEXR* e molti altri.

Con *Houdini* non c'è alcuna necessità di utilizzare plugin aggiuntivi, perché tutto il necessario è già al suo interno: è un pacchetto completo. Tutti i principali pacchetti disponibili sul mercato hanno più o meno pari capacità, la differenza riguarda principalmente la difficoltà che, all'interno di ognuno di questi, si deve superare per arrivare al desiderato risultato finale.
Houdini è uno degli strumenti più flessibili e trasparenti, le sue caratteristiche lo rendono la scelta migliore che si possa fare nell'ambito degli effetti speciali. *Houdini* consente inoltre un più facile accesso ai blocchi di costruzione di livello inferiore
In una produzione cinematografica, ad esempio, *Houdini* viene in genere utilizzato in particolare dal reparto *FX*, tutti gli altri reparti *3D*, con alcune eccezioni naturalmente, di solito sono dominati dall'utilizzo di *Maya*.
I vari approcci, i vari strumenti e le varie pipeline di lavoro impiegate all'interno dei *Destruction* hanno degli aspetti positivi ma nessuno di questi è veramente completo e formidabile rispetto agli altri. Ogni azienda, proprio per questa ragione, sceglie liberamente quale software utilizzare e spesso ne utilizza una combinazione. Spesso è volentieri anche la combinazione può non essere utile e soddisfare i bisogni richiesti, quindi a seconda del problema viene o acquistato uno strumento dedicato oppure viene richiesto a un team o a uno sviluppatore di elaborare un codice specifico atto a risolverlo.
In generale per poter eseguire una comparazione corretta fra tutti i programmi presentati in questo capitolo, è necessario operare un'ulteriore suddivisione rispetto a quelli che sono i programmi completi e quelli che sono i plugin, ovvero dei tool integrabili all'interno di software indipendenti, come ad esempio può essere *Houdini*.
I plugin sono:

1. *DMM* il cui prezzo è di circa 500$.
2. *Fracture FX* il cui prezzo è di circa 400$.
3. *Momentum* il cui prezzo è di circa 500$.
4. *RayFire* il cui prezzo è di circa 400$.
5. *thinkingParticles* il cui prezzo è di circa 540$.

I software completi e indipendenti sono:

1. *RealFlow* con un prezzo di circa 1050$ con *Caronte* con un prezzo di circa 100$.
2. *Houdini Base* con un prezzo di circa 1995$ e *Houdini FX* con un prezzo di circa 4495$.

Il primo gruppo è composto da strumenti che, anche se magari differiscono dal tipo di approccio, consentono l'esecuzione del lavoro di *Destruction* in modo molto più facile, anche se magari alcuni sono meglio di altri sotto certi aspetti, sono abbastanza simili tra loro, anche a livello di costo di utilizzo, come si può notare dai prezzi riportati.
Il secondo gruppo di strumenti contiene due software abili nella gestione di ogni tipo di simulazione anche se ognuno di questi è più specializzato in un diverso settore, come ad esempio può essere *RealFlow* con le simulazioni di fluido.

Caronte, di *RealFlow,* è il solver specializzato negli effetti di *Destruction*, per così dire, e rispetto ad *Hodini* semplifica questo lavoro rendendolo più facile e intuitivo.
Nonostante tutto questo, che insomma potrebbe mettere in cattiva luce le caratteristiche e le potenzialità di *Houdini*, quest'ultimo continua a rimanere il software più completo e più utilizzato perché in grado, magari in modo più complesso di altri, di concretizzare ogni sorta di diavoleria: è flessibile, è veloce e fornisce tutti gli strumenti per fare qualsiasi cosa. L'unico difetto che possiede *Houdini* è che, visto la sua poca intuitività, bisogna conoscere molto a fondo il software per poter diventare degli agili *"artisti"* nel suo utilizzo.

Houdini FX

La versione *Houdini FX* creata dalla *SideFX* permette di realizzare ogni sorta di effetto. Houdini FX incrementa il livello delle prestazioni, fornisce una potente e accessibile esperienza 3D agli *"artisti" VFX* per la creazione di film, spot, videogiochi o in generale di prodotti audiovisivi in cui questa tipologia di lavori sono richiesti. *Houdini FX* include un set di strumenti completo che si diversifica dal pacchetto base in quanto include tutte le caratteristiche *FX,* come la dinamica, i fluidi, ecc., che *Houdini Base* appunto non possiede. Nello specifico le attività che consente di realizzare *Houdini* nella versione *Base* sono:

1. *Modeling*
2. *Character*
3. *Animation*
4. *Lighting*
5. *Compositing*
6. *Volumes*

La versione *FX,* oltre a queste feature, contenute anche nella versione Base, aggiunge:

1. *Pyro FX*
2. *Fluids*
3. *Rigid bodies*
4. *Particles*
5. *Cloth dynamics*
6. *Wire dynamics*

È questa la ragione per la quale per la realizzazione del progetto oggetto di questa trattazione è stata utilizzata la versione *FX* del software. Questa consentiva, rispetto a quella *Base*, di poter realizzare gli effetti di *Destruction* e anche tutte le altre simulazioni che, sempre all'interno del progetto, sono state costruite.

Bullet

Introduzione

Bullet è la libreria fisica utilizzata all'interno del progetto di *Destruction*, oggetto di questo documento, sviluppato attraverso il software *Houdini*. *Bullet*, nello specifico, nella sequenza del crollo dell'edificio, è stata utilizzata per gestire la dinamica di simulazione, in particolare per sviluppare gli aspetti riguardanti l'operazione di rilevamento delle collisioni.
Bullet Physics è quindi una libreria professionale, open source, utilizzata per le operazioni di rilevamento delle collisioni che usano forme che includono modelli concavi e convessi e tutte le primitive di base, e per la dinamica dei corpi rigidi e morbidi, scritta in *C++*, dotata di una licenza *ZLib*, gratuita per qualsiasi uso commerciale su qualunque piattaforma. *Bullet* è stata principalmente progettata per essere poi utilizzata all'interno delle simulazioni create per i videogames, per i visual effects o per la robotica. Il compito principale di un motore fisico è quello di effettuare il rilevamento delle collisioni, risolverle e porre soluzioni ad altri vincoli, e fornire le trasformate del sistema mondo aggiornate per tutti gli oggetti. *Bullet* è stato progettato per essere personalizzabile e modulare, quindi componibile. In particolare lo sviluppatore può ad esempio:

1. Utilizzare solo il componente atto al rilevamento delle collisioni.
2. Utilizzare il componente che gestisce la dinamica del corpo rigido senza il componente che esegue la dinamica del corpo morbido.
3. Utilizzare solo piccole parti della libreria ed estendere questa a seconda delle necessità.
4. Scegliere di usare una versione della libreria a singola precisione o doppia precisione. Al fine di consentire di compilare la libreria in singola precisione in virgola mobile e doppia precisione, in *Bullet* viene utilizzato il tipo di dato *btScalar*. Un *btScalar* è una parola elegante utilizzata per esprimere un numero in virgola mobile.

La pipeline della fisica del corpo rigido di Bullet

In questa sezione vengono descritte le strutture dati e le fasi computazionali presenti all'interno della pipeline fisica di *Bullet*. In *Bullet,* tutto questo è rappresentato da un *btDynamicsWorld*. L'implementazione del *btDynamicsWorld* di default, per *Bullet,* è il *btDiscreteDynamicsWorld*. In questo contesto può essere creato o un *btDiscreteDynamicsWorld* o un *btSoftRigidDynamicsWorld*. Queste due classi, che derivano da *btDynamicsWorld*, forniscono un'interfaccia ad alto livello che permette la gestione degli oggetti fisici e dei vincoli, consentendo inoltre anche l'implementazione dell'aggiornamento di tutti gli oggetti per ogni frame.

Può essere costruito un *btRigidBody* o un *btCollisionObject* da aggiungere al *btDynamicsWorld* fornendo:

1- Una massa: positiva per la dinamica degli oggetti in movimento e pari a zero per gli oggetti statici.
2- Un *CollisionShape*: come un box, una sfera, un cono, un convex hull o una mesh triangolare.
3- Delle proprietà dei materiali come l'attrito (*friction*) e la restituzione (*restitution*).

Per aggiornare la simulazione a ogni frame, viene chiamato lo *stepSimulation* sul *btDynamicsWorld.* Il *btDiscreteDynamicsWorld* tiene automaticamente in considerazione un passo temporale variabile eseguendo un'interpolazione invece di simulare per i piccoli passi temporali. Esso utilizza un passo temporale interno fisso di 60 Hertz. Lo *stepSimulation* esegue il rilevamento delle collisioni e la simulazione fisica. Viene poi aggiornata la trasformazione del mondo per gli oggetti attivi attraverso il *setWorldTransform* del *btMotionState*. Il bt*MotionStates* è un modo per *Bullet* di ottenere la trasformata del sistema di coordinate del mondo degli oggetti che vengono simulati nel corso della fase di *Rendering* all'interno del programma. Quindi, per ogni oggetto, deve essere aggiornata la sua posizione per poi essere renderizzata e *Bullet* utilizza il bt*MotionStates* per eseguire questa operazione.
Bullet utilizza un sistema di coordinate destrorso. Spesso è importante che i dati siano allineati a 16 byte. *Bullet* fornisce di default degli allocatori di memoria che gestiscono l'allineamento, gli sviluppatori, però, possono fornire un proprio allocatore di memoria. Tutte le allocazioni di memoria in *Bullet* usano:

1- *btAlignedAlloc*, che permette di specificare la dimensione e l'allineamento.
2- *btAlignedFree*, che libera la memoria allocata da *btAlignedAlloc*.

Per sovrascrivere l'allocatore di memoria di default, è possibile scegliere tra:

1- *btAlignedAllocSetCustom*, che viene utilizzato quando l'allocatore personalizzato non supporta l'allineamento.
2- *btAlignedAllocSetCustomAligned*, che può essere usato per impostare l'allocatore allineato di memoria personalizzato.

Per garantire che una struttura o una classe venga automaticamente allineata è possibile utilizzare la macro: *ATTRIBUTE_ALIGNED16(type) variablename*, che crea una variabile allineata su 16 byte.
Per mantenere un array di oggetti, in origine, *Bullet* utilizzava una struttura dati di tipo *std::vector*, in seguito, per ragioni di portabilità e compatibilità, è stata costruita una classe apposita di tipo vettore, ovvero il *btAlignedObjectArray*, che ricorda molto da vicino *std::vector*. Questa nuova classe utilizza un allocatore allineato per garantire, appunto, l'allineamento. Il *btAlignedObjectArray* utilizza diversi metodi per ordinare l'array, tra questi troviamo il *quick sort* o il *heap sort*.

Il rilevamento delle collisioni

Per quanto riguarda il rilevamento delle collisioni, le principali strutture dati che *Bullet* mette a disposizione sono le seguenti:

1- *btCollisionObject*, ovvero l'oggetto che ha una trasformata del mondo e una forma di collisione.
2- *btCollisionShape*, che descrive la forma di collisione di un oggetto di collisione, come un box, una sfera, un guscio convesso o una mesh triangolare. Una singola forma di collisione, inoltre, può essere condivisa tra oggetti di collisione multipli.
3- *btGhostObject*, ovvero uno speciale *btCollisionObject* utile per le interrogazioni di collisione localizzate veloci.
4- *btCollisionWorld,* che memorizza tutto il *btCollisionObject* e fornisce un'interfaccia per migliorare le interrogazioni.

Il processo di rilevamento delle collisioni, nella fase *Broad Phase,* in *Bullet,* fornisce la struttura di accelerazione necessaria per respingere rapidamente coppie di oggetti che basano la loro sovrapposizione su sistemi di bounding box allineati agli assi (*AABB*). *Bullet* offre diverse strutture di accelerazione per la fase *Broad Phase*:

1- *btDbvBroadphase,* che utilizza una rapida gerarchia di bounding volume dinamici basati su alberi *AABB*.
2- *btAxisSweep3* e *bt32BitAxisSweep3*.
3- *btSimpleBroadphase*, ovvero un'implementazione di riferimento a forza bruta. È lento ma facile da capire e utile per il debug e il test di una *Broad Phase* più avanzato.

Bullet, per la fase *Broad Phase* del *Collision Detection*, nello specifico, offre la possibilità di scegliere come implementare questa fase a seconda del risultato che si vuole raggiungere. La principale implementazione utilizzata si basa su un sistema di gerarchie di volumi limitanti dinamico, che viene aggiornato in modo incrementale durante la simulazione e quindi durante lo spostamento dei *Bounding Box,* aggiungendo o rimuovendo oggetti. La *Broad Phase* aggiunge e rimuove le coppie sovrapposte da una cache. Le coppie sovrapposte sono persistenti nel tempo e possono memorizzare nella cache informazioni quali precedenti forze di vincolo di contatto. Il *btPersistentManifold* è la cache che consente la memorizzazione i punti di contatto per una data coppia di oggetti.

Bullet a seconda delle necessità utilizzerà quindi un certo algoritmo di collisione. A questo proposito esiste, appunto, il *Collision Dispatcher*, ovvero una classe che consente di avviare un processo che si ripete su ogni coppia di *Bounding Box* che si sovrappone, cercando un algoritmo di collisione che trovi il match in base ai tipi di oggetti coinvolti, eseguendo l'algoritmo di collisione e calcolando i punti di contatto. *GJK* sta per *Gilbert, Johnson* e *Keerthi*, ovvero le persone dietro l'algoritmo per il calcolo della distanza convessa. Questo è combinato con l'*EPA,* utilizzato per il calcolo della profondità di penetrazione. *EPA* è l'acronimo di *Expanding Polythope Algorithm* di *Gino Van Den Bergen*. *Bullet* ha una propria implementazione libera dello *GJK* e dell'*EPA*, e offre tre semplici modi per garantire che solo certi oggetti collidano tra di loro:

1. *Masks.* Se le *Masks* sono sufficienti si possono utilizzare in quanto meglio funzionanti e di più semplice utilizzo.
2. *Broadphase Filter Callbacks*
3. *Near Callbacks*.

La dinamica del corpo rigido viene implementata sulla parte superiore del modulo di rilevamento delle collisioni, aggiungendo le forze, la massa, l'inerzia, la velocità e i vincoli.

1- *btRigidBody* è utilizzato per simulare singoli oggetti in movimento con 6 gradi di libertà. *btRigidBody* deriva dal *btCollisionObject* ed eredita il suo *World Transform*, la frizione e la restituzione e aggiunge la velocità lineare e angolare.
2- *btTypedConstraint* è la classe base per i vincoli di corpo rigido, e contiene vincoli tra cui il *btHingeConstraint*, il *Point2PointConstrant*, il *btConeTwistConstraint*, il *btSliderConstraint* e il *btGeneric6DOFconstraint*.
3- *btMultiBody* è una rappresentazione alternativa di una gerarchia di corpi rigidi.

In *Bullet* esistono 3 differenti tipi di oggetti:

1- I corpi rigidi dinamici con massa positiva, per i quali ad ogni frame di simulazione la dinamica verrà aggiornata attraverso il *World Transform.*
2- I corpi rigidi statici con massa pari a zero, i quali non si possono muovere, ma solo collidere.
3- I corpi rigidi cinematici con massa pari a zero, i quali possono essere animati dall'utente e nei quali è possibile solo un modo di interazione, ovvero gli oggetti dinamici sono spinti ma non sono presenti alcune influenze provenienti dagli oggetti dinamici.

Tutti questi oggetti devono essere aggiunti alle dinamiche del sistema di coordinate del mondo. A un corpo rigido, inoltre, può essere assegnata una forma collisione. Questa forma di collisione può essere utilizzata per calcolare la distribuzione della massa, detta anche tensore inerziale.

Le forme di collisione

Bullet supporta una grande varietà di forme di collisione differenti, è possibile inoltre aggiungerne di proprie. Per ottimizzare le prestazioni e la qualità è importante scegliere la forma di collisione che soddisfa i propri scopi.

Le primitive convesse

Molte forme primitive sono centrate attorno all'origine del loro sistema di coordinate locali:

1. *btBoxShape*.
2. *btSphereShape*.
3. *btCapsuleShape*.
4. *btCylinderShape*.
5. *btConeShape*.
6. *btMultiSphereShape*.

Le forme composte

Diverse forme convesse possono essere combinate in una forma composta utilizzando il *btCompoundShape*. Questa è una forma concava costituita da sottoparti convesse dette *child shapes*. Ogni *child shape* ha una propria *local offset transform*, rispetto al suo *btCompoundShape*. Questa è un'ottima idea per approssimare forme concave con una collezione di gusci convessi e memorizzarli in un *btCompoundShape*. È possibile regolare il centro di massa del *btCompoundShape* utilizzando il metodo di utilità *btCompoundShape::calculatePrincipalAxisTransform*. In *Bullet* il *World Transform* di un corpo rigido è sempre pari al suo centro di massa e la sua base definisce anche il sistema di coordinate locali di inerzia. Il tensore di inerzia locale dipende dalla forma, e la classe *btCollisionShape* fornisce un metodo per calcolare l'inerzia locale data una massa. Il *World Transform* deve essere un *Rigid Body Transform*, il che significa che non deve contenere la possibilità ad esempio di ridimensionamento. Se invece si desidera scalare un oggetto è possibile scalarne la forma di collisione. Altre trasformazioni, quali ad esempio il taglio, possono essere applicate ai vertici di una mesh triangolare se necessario. Nel caso la forma collisione non sia allineata con la trasformata del centro di massa, essa può essere spostata per il match. Per fare questo è possibile utilizzare, appunto, il *btCompoundShape*.

Le forme di gusci convessi

Bullet supporta diversi metodi per rappresentare modelli triangolari convessi. Il modo più semplice è quello di creare un *btConvexHullShape* e passargli un vettore di vertici. In alcuni casi la mesh grafica contiene troppi vertici per essere utilizzata direttamente come *btConvexHullShape*. In tal caso, è necessario cercare di ridurre il numero di vertici.

Le mesh triangolari concave

Un modo molto efficiente per rappresentare mesh triangolari statiche è quello di utilizzare un *btBvhTriangleMeshShape*. Quando si dispone di più istanze della stessa mesh triangolare, ma con diversa scala, è possibile, ad esempio, utilizzare un *btBvhTriangleMeshShape* più volte, usando il *btScaledBvhTriangleMeshShape*. La *btBvhTriangleMeshShape* è in grado di memorizzare parti di mesh multiple.

La decomposizione convessa

Idealmente le mesh concave dovrebbero essere utilizzate solo per le immagini statiche. In caso contrario, i suoi gusci convessi devono essere utilizzati passando la mesh al *btConvexHullShape*. Se una singola forma convessa non è abbastanza dettagliata, più parti convesse possono essere combinate in un oggetto composto chiamato *btCompoundShape*. La decomposizione convessa quindi può essere utilizzata per decomporre la mesh concava in più parti convesse.

Il margine di collisione

Bullet utilizza un piccolo margine di collisione per le forme di collisione, questo per migliorare le prestazioni e l'affidabilità del rilevamento delle collisioni. Generalmente il margine di collisione espande l'oggetto. Questo crea quindi un piccolo spazio. Per compensare questo, alcune forme possono sottrarre il margine dalla dimensione reale. Per gli oggetti con gusci convessi, invece, vi è un metodo per rimuovere il gap introdotto dal margine che consiste nel contrarre l'oggetto.

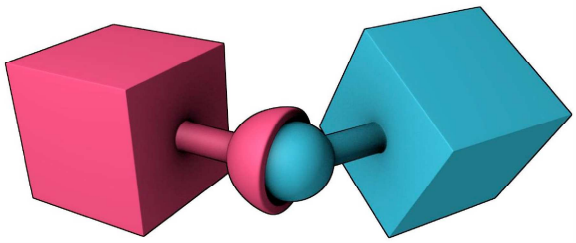
I Vincoli

Ci sono diversi vincoli che possono essere implementati in *Bullet*. Tutti i vincoli, compresi il *btRaycastVehicle*, derivano dalla classe *btTypedConstraint*. La legge di vincolo che esiste tra due *rigidbodies* impone che almeno uno di essi debba essere dinamico.
Tra i vincoli disponibili in *Bullet* sono presenti:

1. *Point To Point Constraint.*
2. *Hinge Constraint.*
3. *Slider Constraint.*
4. *Cone Twist Constraint.*
5. *Generic 6DOF Constraint.*

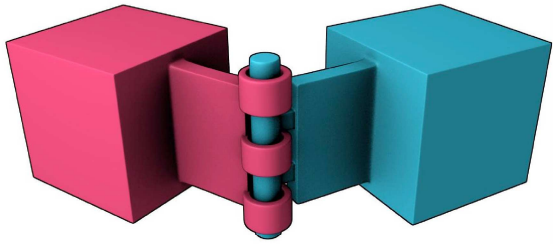
Point to point

Il vincolo punto a punto limita la traslazione in modo che i pivot locali dei due corpi rigidi facciano match nel *WorldSpace*. Una catena di corpi rigidi può essere quindi collegata tramite questo vincolo.



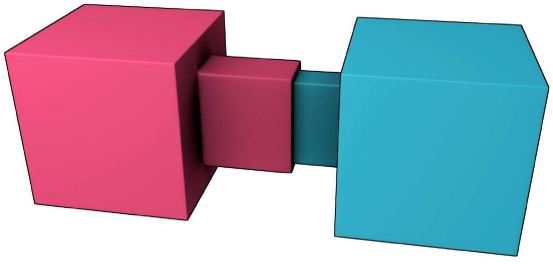
Hinge

Il vincolo a cerniera, o giunto rotoidale, limita due gradi di libertà angolari supplementari, in modo che il corpo possa solo ruotare attorno a un asse, ovvero l'asse cerniera. Questo può essere utile per rappresentare porte o ruote girevoli attorno a un asse. L'utente può specificare i limiti e motore per la cerniera.



Slider

Il vincolo cursore permette al corpo di ruotare attorno a un asse e traslare lungo questo asse.



Cone Twist

Per creare dei corpi articolati il *Cone Twist Constraint* è molto utile. Questo è uno speciale vincolo punto a punto che aggiunge dei limiti sull'asse di *Cone* e *Twist*. L'asse x solitamente funge da asse di torsione.

Generic 6DOF

Questo vincolo generico può emulare una varietà di vincoli standard, consentendo la configurazione di ciascuno dei 6 gradi di libertà (*DOF*). I primi 3 assi *DOF* sono assi lineari, che rappresentano la traduzione di *rigidbodies*, e gli ultimi 3 assi *DOF* rappresentano il movimento angolare. Ogni asse può essere bloccato, libero o limitato. Quando viene creato un nuovo *btGeneric6DofSpring2Constraint* tutti gli assi sono bloccati, in seguito l'asse può essere riconfigurato. Nonostante tutto molte combinazioni che includono gradi di libertà angolari liberi e/o limitati non possono essere definite.

La dinamica dei corpi morbidi

La dinamica del corpo morbido si appoggia sulla dinamica del corpo rigido già esistente in *Bullet*. Tra i corpi morbidi, i corpi rigidi e gli oggetti di collisione, esiste un'interazione bidirezionale. Il *btSoftBody* è l'oggetto principale utilizzato per rappresentare un corpo morbido. Esso deriva dalla classe *btCollisionObject*. A differenza dei corpi rigidi, nei i corpi morbidi ogni nodo, o vertice, viene specificato nel sistema di coordinate del mondo. Il *btSoftRigidDynamicsWorld*, invece, è il contenitore dei corpi morbidi, dei corpi rigidi e degli oggetti di collisione. Il *btSoftBody Helpers::CreateFromTriMesh* è in grado di creare automaticamente un corpo morbido partendo da una mesh triangolare.
Per impostazione predefinita, nei corpi morbidi si esegue l'operazione di rilevamento delle collisioni tra i vertici (nodi) e tra i triangoli (facce). Questa procedura richiede una tassellazione densa, altrimenti le collisioni potrebbero essere perse. Esiste un metodo migliore che utilizza una decomposizione automatica in cluster deformabili convessi.

Capitolo tre: il progetto

Ideazione

In questa prima fase di progettazione è stato scelto lo scatto di riferimento sul quale poi si sarebbero eseguite tutte le fasi di elaborazione successive. L'idea era quella di riprodurre il crollo di un edificio, tratto da uno scatto panoramico reale. È stato quindi dato il via alla fase di ricerca, operata su *Internet*, di una foto, la cui scelta è stata eseguita tenendo a mente l'obiettivo che si voleva raggiungere: l'immagine in questione sarebbe dovuta essere interessante dal punto di vista prospettico e pratica, ovvero sulla quale fosse stato poi possibile ricreare qualcosa di plausibilmente realistico che abbattesse le difficoltà di modellazione, in quanto questa era una fase che si voleva risolvere in breve tempo, e funzionale alla credibilità della scena del crollo, in quanto non si sarebbe operata un'intera ricostruzione della scena 3D e quindi non si sarebbero potute riprodurre tutte le possibili interazioni dovute al crollo con i vari elementi presenti all'interno della scena. Qui di seguito viene riportato lo scatto panoramico scelto alla fine di tutte queste considerazioni.

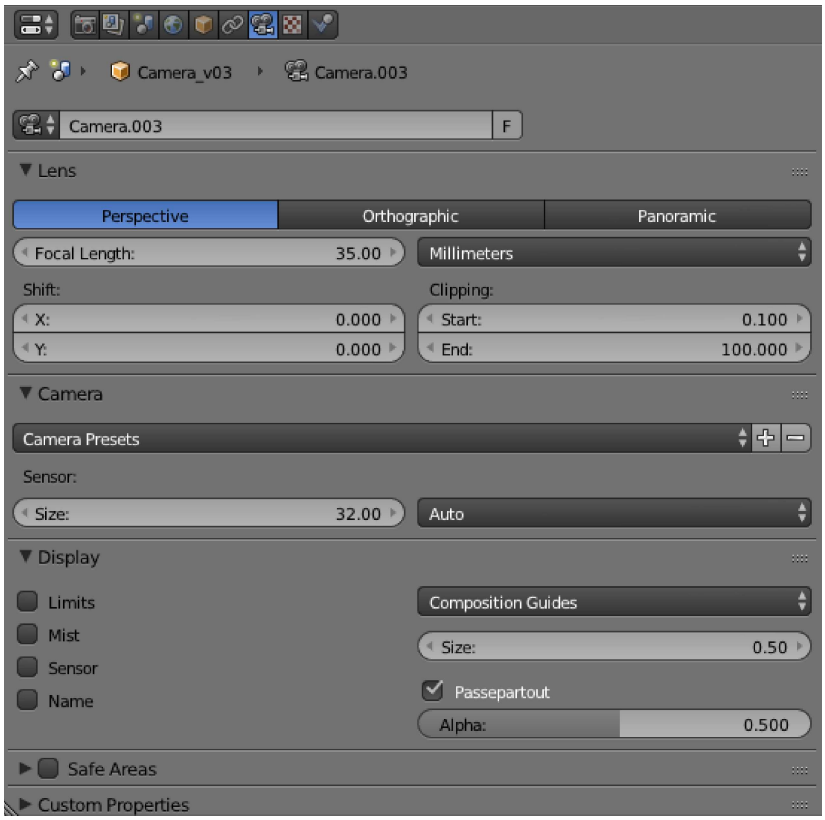
Una volta scelta l'immagine di riferimento, sono state necessarie due ulteriori operazioni, una di *"pulizia"* e una di *"ripopolamento"*, per poi poter procedere con le successive fasi del processo di creazione del prodotto visivo. Nello specifico, in prima battuta, sono stati rimossi tutti gli elementi di disturbo presenti sulla superficie del palazzo, come ad esempio i ponteggi, al fine di creare una superficie omogenea per ottimizzare l'operazione di proiezione della texture sulla superficie del modello 3D e facilitare la fase di modellazione. In seconda battuta, invece, è stata effettuata un'operazione di ripopolamento, ovvero si è cercato di ricostruire il *Backgraund* che sarebbe rimasto scoperto dopo il crollo dell'edificio.

Nello specifico, prima sono stati rimossi il palazzo e la sua ombra, i quali sarebbero stati ricreati in digitale in momenti successivi, e poi sono stati aggiunti degli elementi coerenti con il paesaggio presente all'interno della foto per riempire i vuoti rimasti scoperti.
Tutte le operazioni che fanno parte di questa fase di ideazione e progettazione sono state realizzate in *Photoshop*. Di seguito vengono riportati gli scatti ottenuti a seguito dell'esecuzione di queste due importanti operazioni di elaborazione dell'immagine.

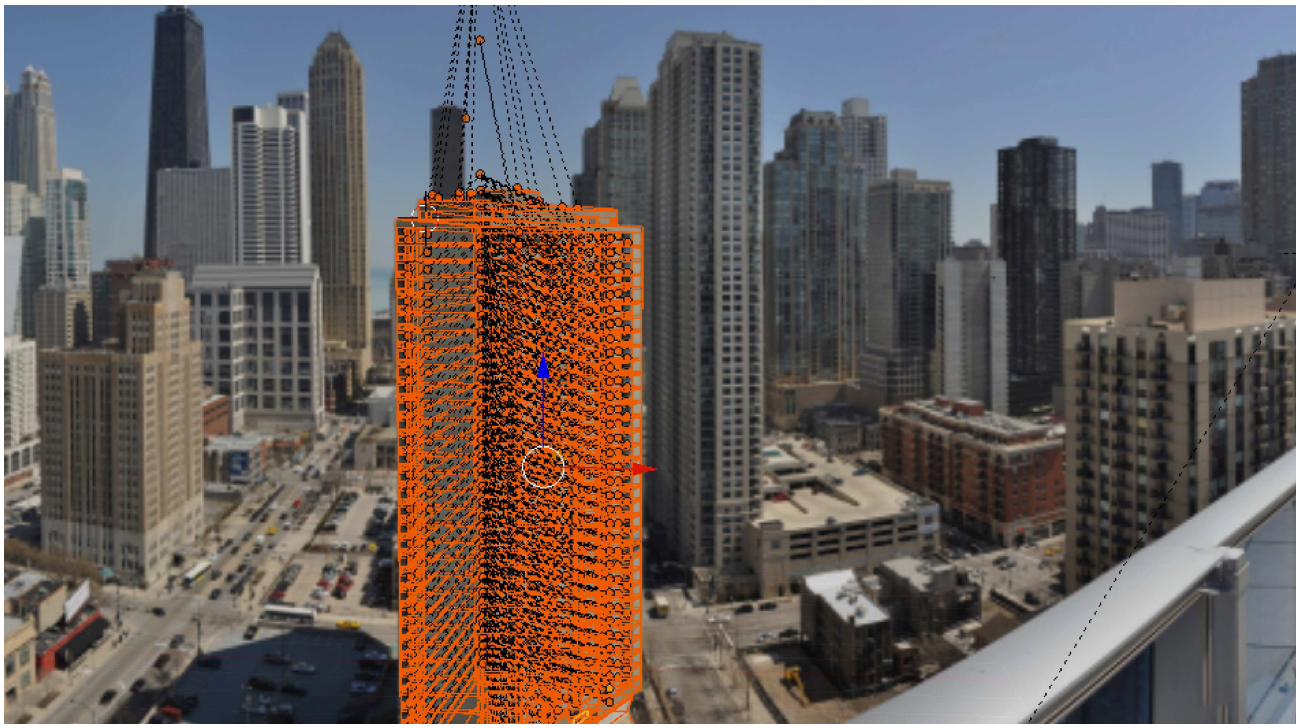
Grandi difficoltà, in questa fase, non c'è ne sono state; è stata, però, impegnativa, dal punto di vista creativo, l'operazione di ricostruzione del *Backgraund*, ovvero ciò che sarebbe rimasto visibile, dietro il palazzo, dopo il suo crollo. Nello specifico, quindi, sono state complicate la ricerca degli elementi da inserire e la disposizione di questi all'interno dello scatto. A livello di tempistiche lavorative, le procedure di elaborazione dell'immagine, effettuate in questa fase, si sono concluse nell'arco di una settimana.

Modellazione

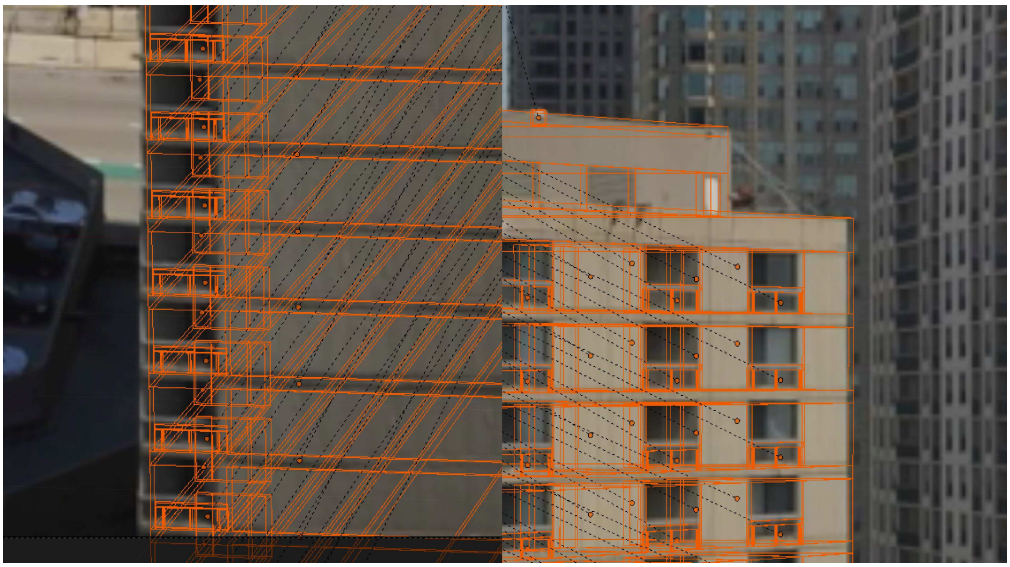
In questa fase è stato sviluppato il modello 3D dell'oggetto scelto per la scena di *Destruction* partendo dalla configurazione delle impostazioni della camera virtuale, passaggio fondamentale per poter eseguire il *Projection Mapping,* ovvero la proiezione dell'immagine, che fungerà da texture, sul modello del palazzo. Questa operazione è stata eseguita per conferire alla superficie dell'edificio un aspetto più realistico possibile, in quanto si sarebbe basato su una texture reale e non su una costruita manualmente, su misura, in modo, quindi, artificioso. Le operazioni di proiezione delle texture e l'applicazione dei materiali sono però state svolte in un momento successivo, in questa fase è stato solamente realizzato il modello 3D del palazzo, modellato a partire da specifiche impostazioni della camera virtuale coincidenti con il punto di vista della camera dell'immagine di riferimento. Questa parte del processo è stata realizzata tramite l'utilizzo del software *Blender* e ha richiesto circa due settimane di lavoro, in quanto si sono riscontrati dei grossi problemi nel far combaciare le due prospettive, quella della scena 3D e quella dell'immagine reale, e quindi il punto di vista della camera reale con quello della camera virtuale. Il passo fondamentale prima di procedere con la modellazione è stato appunto la ricerca del match tra la camera virtuale e quella reale. Qui di seguito viene riportato il set delle impostazioni della camera virtuale che ha garantito un buon compromesso di match tra queste due prospettive.



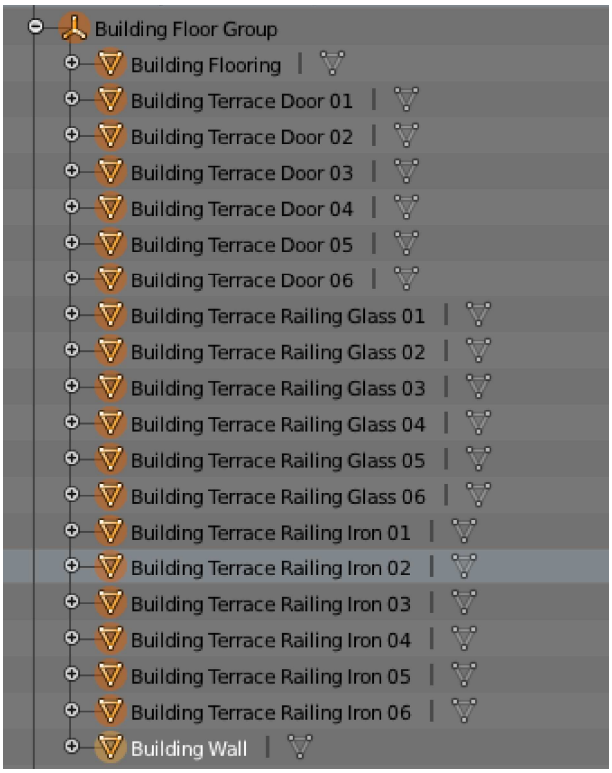
Come è stato detto in precedenza, questa operazione è stata molto difficoltosa, in quanto lo scatto di riferimento riportava una grande distorsione, probabilmente ricreata in modo artificiale, il cui centro era situato proprio nella parte centrale dell'immagine, dove si trovava, appunto, l'edificio scelto per il *Destruction*, del quale si sarebbe dovuto ricreare il modello 3D. L'operazione di proiezione delle texture riesce al meglio quando questo match è più preciso possibile e siccome questa distorsione non era di aiuto ed era molto marcata si è cercato di trovare comunque un compromesso che riducesse al minimo questo errore. Di seguito viene mostrato quanto detto tramite un'immagine esplicativa che mostra il compromesso di match trovato.



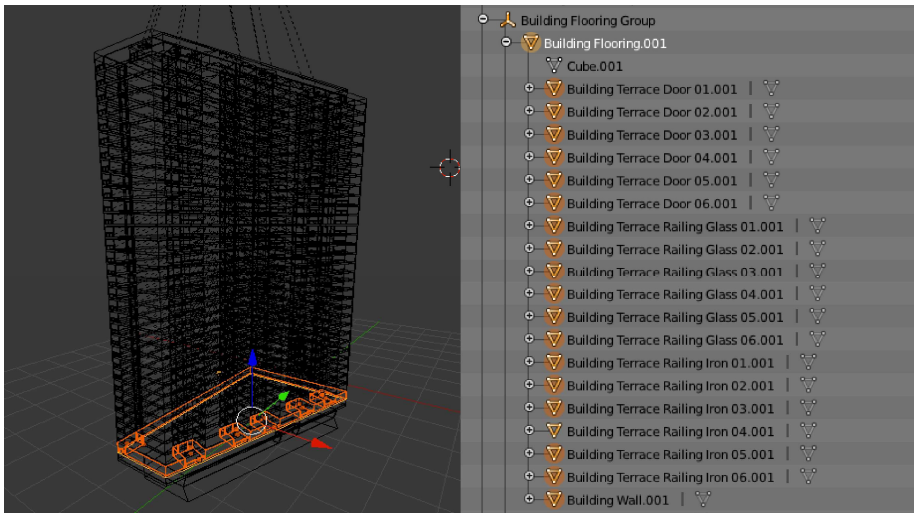
Come si può notare dalle immagini seguenti, se ci si avvicina facendo uno zoom sui bordi dell'edificio, si può osservare come il modello del palazzo non sia esattamente coincidente con quello reale dell'immagine sottostante.



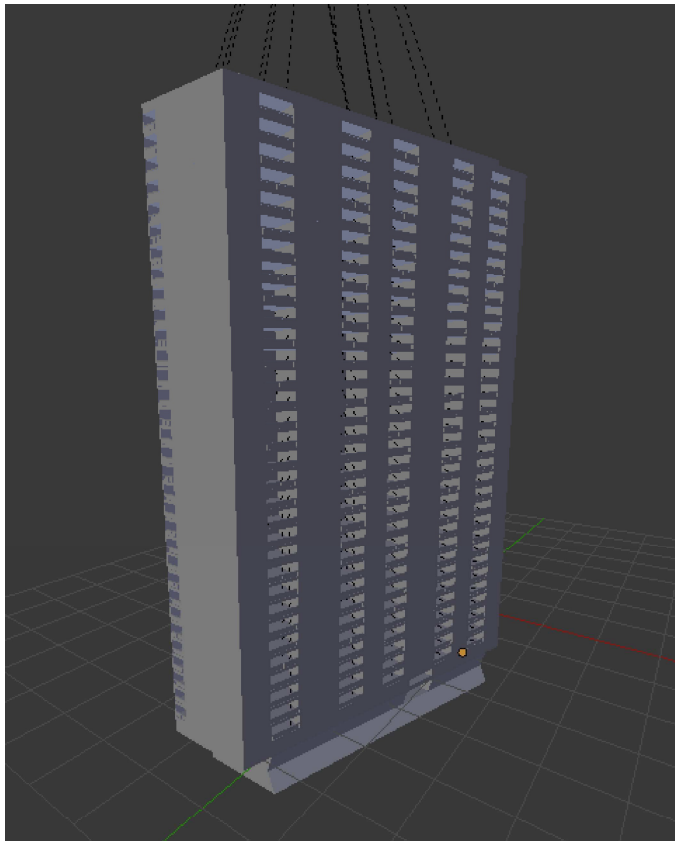
Nonostante questa problematica, l'operazione di proiezione delle texture, eseguita successivamente, non ha dato effetti negativi, soprattutto perché da lontano questi artefatti non si notano ad occhio nudo. Il passo successivo è stato la costruzione vera e propria del modello 3D del palazzo, eseguita attraverso una semplice modellazione poligonale basata su geometria solida primitiva, ovvero si è partiti da un cubo e poi tramite successive fasi di estrusione sono stati realizzati tutti gli elementi che compongono l'edificio. Siccome tutti i piani, eccetto la base e la cima, si ripetevano sempre uguali, per realizzare i 32 piani centrali è stato utilizzato il modificatore array, che praticamente ha realizzato 32 copie di uno stesso piano traslandole verso l'asse z, che in questo caso, era l'asse che puntava verso l'alto. Quindi è stato costruito un piano completo costituito dagli elementi riportati qui sotto opportunatamente nominati, un'operazione, quella della nomenclatura, di fondamentale importanza perché in questo modo è stato possibile esportare i singoli elementi su *Houdini* e operare su di essi il controllo della frattura e altre proprietà senza dover eseguire prima una suddivisione degli elementi in gruppi, perché già effettuata su *Blender*.



Una volta realizzato un piano completo di tutti i suoi elementi, questo è stato dato in pasto al modificatore array che ha creato tutti gli altri 32, i quali riportano una nomenclatura come quella visibile nella figura sottostante.



Ovvero a ogni piano è stato aggiunto un codice finale, in questo caso nella foto è lo *001*, che indica l'effettivo numero nel piano: tutti piani quindi hanno un codice finale progressivo che li identifica insieme a tutti gli elementi che lo costituiscono, quindi il primo piano riporta il codice *001*, il secondo lo *002* e così via. Questo ha permesso su *Houdini*, come già detto, di suddividere i vari elementi perché facenti parte di gruppi già costituiti in *Blender* attraverso questo tipo di nomenclatura.



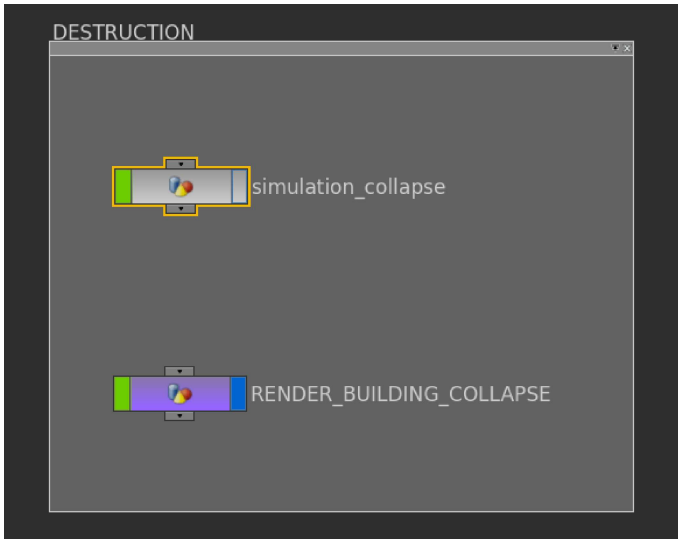
La modellazione, a differenza dell'impostazione delle proprietà della camera virtuale, non ha creato difficoltà particolari. L'unica cosa su cui si è dovuto fare attenzione, mentre si modellava, è stata la creazione degli elementi componenti i vari piani. Ciascuno di essi avrebbe dovuto essere modellato come una mesh unica, evitando la presenza di facce interne, questo per facilitare le operazioni di frattura su *Houdini*. Nello specifico nella fase di fratturazione è importante non spezzare il volume interno degli oggetti, per questo in fase di modellazione è stata posta attenzione nell'evitare di generare facce interne. L'ultima operazione eseguita su *Blender* prima di procedere con la fase di *Destruction* in *Houdini* è stata l'esportazione. Il modello 3D del palazzo è stato esportato come file *".obj"* mentre le impostazioni della camera sono state esportate come file *".abc"*. Il file *Wavefront ".obj"*, anche conosciuto come *Wavefront Object OBJ*, è un formato di file non compresso, usato per memorizzare e scambiare dati 3D. Viene utilizzato per la memorizzazione di oggetti geometrici composti da linee, poligoni, curve e superfici. Il file *".obj"* non richiede alcun tipo di intestazione, benché sia pratica comune iniziare il file con una linea di commento. Un aspetto importante è che questo file mantiene in memoria anche i gruppi generati grazie alla nomenclatura operata in fase di modellazione, è quindi molto utile perché queste sono informazioni che si possono recuperare una volta importato il file su *Houdini*. *Alembic* è un formato di file aperto per lo scambio di geometrie e animazioni. È stato utilizzato per memorizzare le impostazioni della camera virtuale che sono state importante in *Houdini* per essere utilizzate per la visualizzazione corretta della scena durante la costruzione delle simulazioni e per la successiva fase di *Rendering*.

Configurazioni iniziali

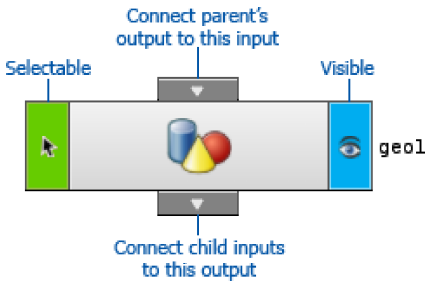
Prima di importare i file in *Houdini* è stato necessario impostare il progetto di *Destruction*. Come concordato in fase di ideazione si sarebbe dovuto realizzare il crollo di un palazzo. Per realizzare ciò, e rendere il tutto più credibile e godibile per gli occhi di chi lo avrebbe guardato, sono state integrate diverse simulazioni, quella di *Destruction*, quella dei *Debris*, per aggiungere maggiore dettaglio, quella dello *Smoke*, per accompagnare il crollo, e quella di *Fuel*, ovvero dell'esplosione, per dare una spiegazione sull'origine del crollo, tutto per arricchire la simulazione centrale di *Destruction*. Oltre a realizzare queste simulazioni sono stati creati ulteriori blocchi per la gestione delle impostazioni della camera, dell'illuminazione, delle ombre e dei materiali per la realizzazione dei *Rendering* finali.

Prefratturazione

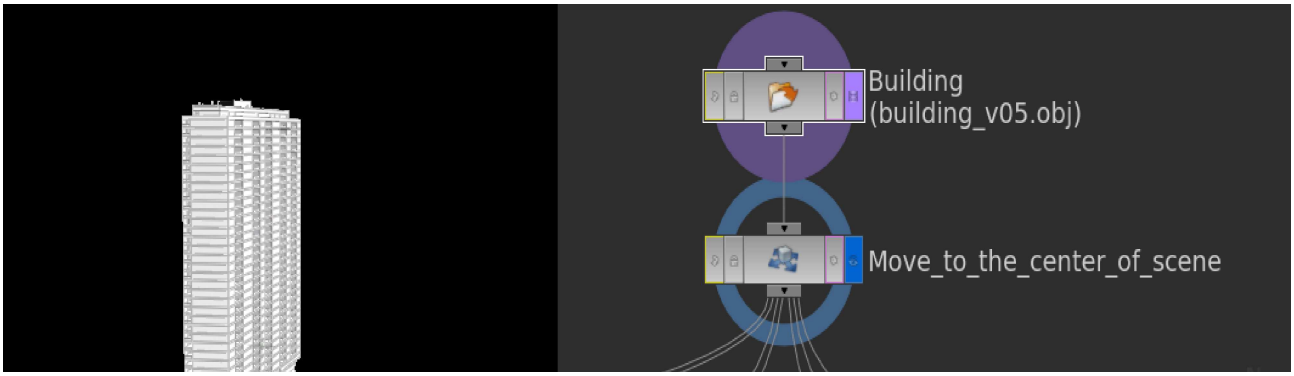
Prima di procedere con il primo step del processo di distruzione è stato necessario importare il file *".obj"* del modello del palazzo e quello *".abc"* della camera. Prima ancora di questa operazione sono stati creati due nodi di geometria che avrebbero contenuto, uno (*simulation_collpase*) tutto ciò che avrebbe costituito la simulazione, e l'altro (*RENDER_BUILDING_COLLAPSE*) tutta la parte di visualizzazione utilizzata poi nella fase finale di *Rendering*. Per una più chiara leggibilità del piano di lavorazione sono stati scelti dei colori che avrebbero permesso di distinguere le varie tipologie di nodi utili nel processo di *Destruction*: il colore viola per i nodi di render, il rosa per i nodi di visualizzazione, il giallo per i nodi che contengono dei parametri sui quali è possibile effettuare una modifica, l'azzurro per i nodi di simulazione e il nero per i nodi di output o uscita.



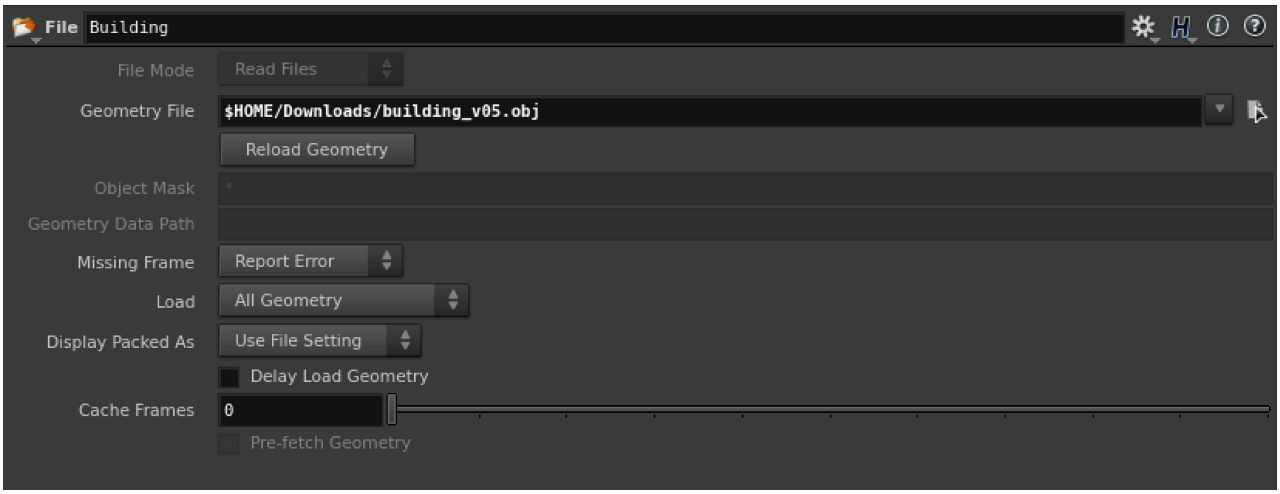
Il primo livello di nodi, quello più superficiale, per essere più chiari, mostrato nella foto sovrastante, è composto da due *Object Node*, il cui contenuto è stato già citato prima, i quali per definizione rappresentano gli oggetti presenti all'interno della scena, nello specifico sono due *Geometry Node*.



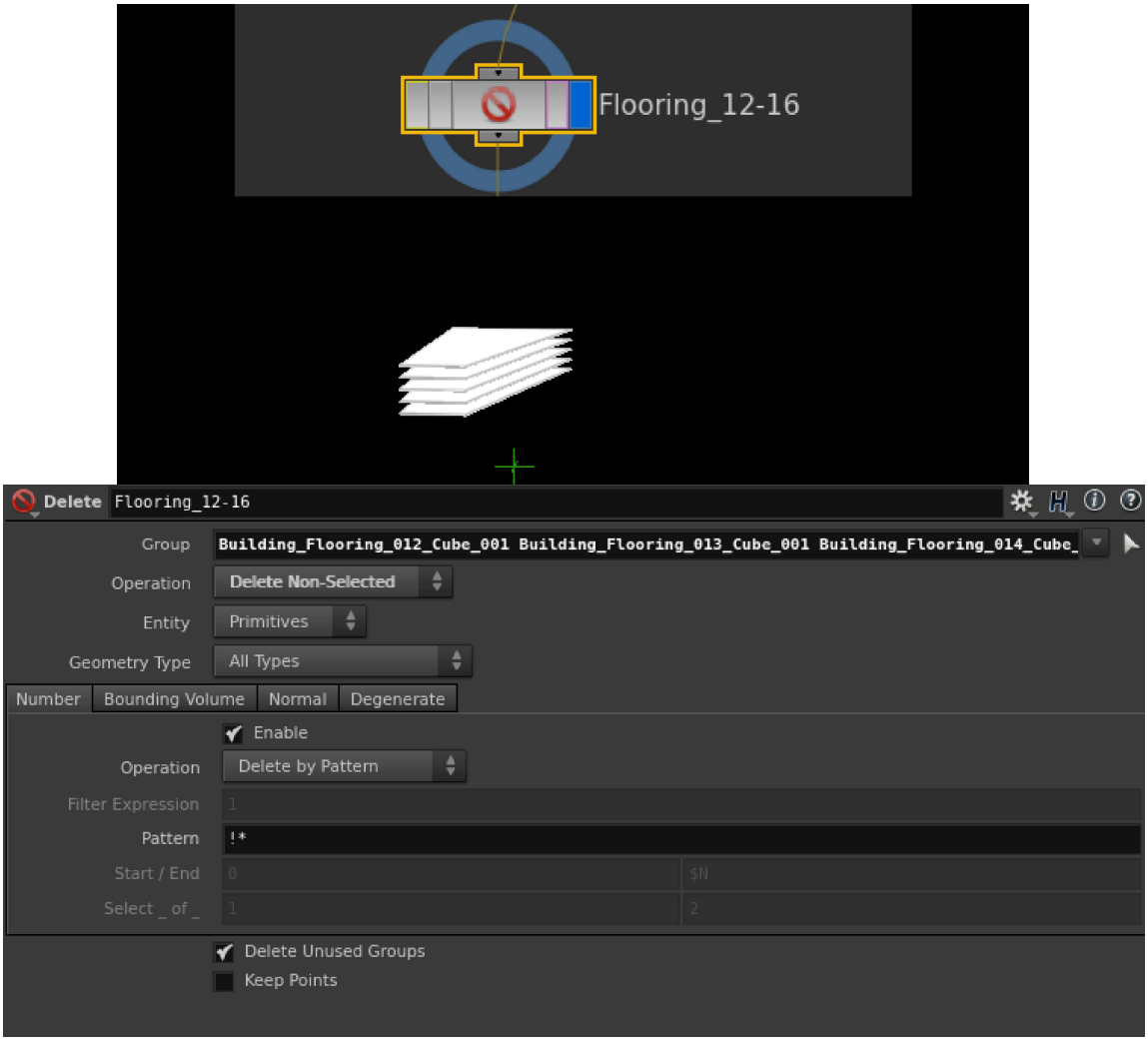
I *Geometry Node* sono dei contenitori per operatori geometrici (*SOP*) che definiscono un oggetto modellato. Un importante distinzione da discutere è la differenza tra operatori *SOP* e *DOP*, i primi, anche detti *Surface Operator* o *Geometry Node*, sono situati in un livello oggetto, vengono usati per costruire e modificare la geometria ed ereditano qualsiasi trasformazione esercitata a livello oggetto, i secondi, anche detti *Dynamic Operator* o *Simulation/Solver Node*, vengono utilizzati per realizzare simulazioni, prelevano la geometria dai nodi di *SOP* e passano questi dati ai *DOP* e settano le condizioni e le regole che gestiscono la dinamica di simulazione. Per affrontare il discorso che riguarda la prefratturazione è necessario scendere di livello, quindi entrare dentro il nodo che porta il nome *simulation_collapse*, in questo caso, per una questione di comodità, contiene al suo interno anche il nodo di simulazione. Prima di incominciare a rompere il modello è stato necessario importarlo. L'importazione del modello 3D del palazzo è stata eseguita attraverso il nodo di *File*.



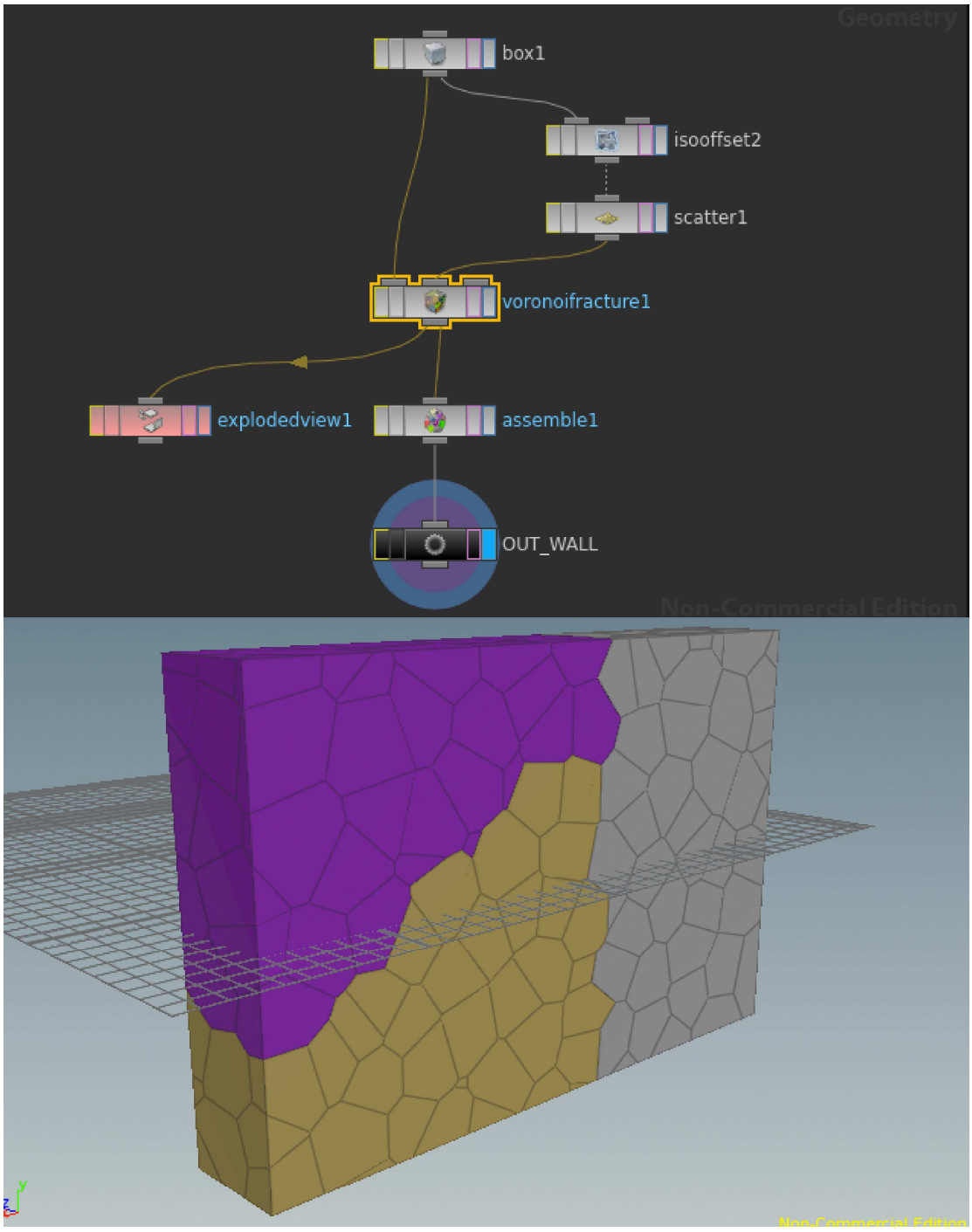
Il nodo di *File* legge, scrive o fa la cache della geometria su disco. Questo nodo ha due modi di operare, se c'è un input connesso si può scegliere un *FILE MODE* per controllare come il nodo legge, scrive o esegue la cache della geometria su disco, se non c'è nessun input connesso si può specificare un file di geometria da leggere da disco e spedirlo all'uscita del nodo stesso. Nel caso di specie il nodo di *File* non ha alcun input connesso, ma legge il file *"obj"* del modello esportato da *Blender* memorizzato su disco.



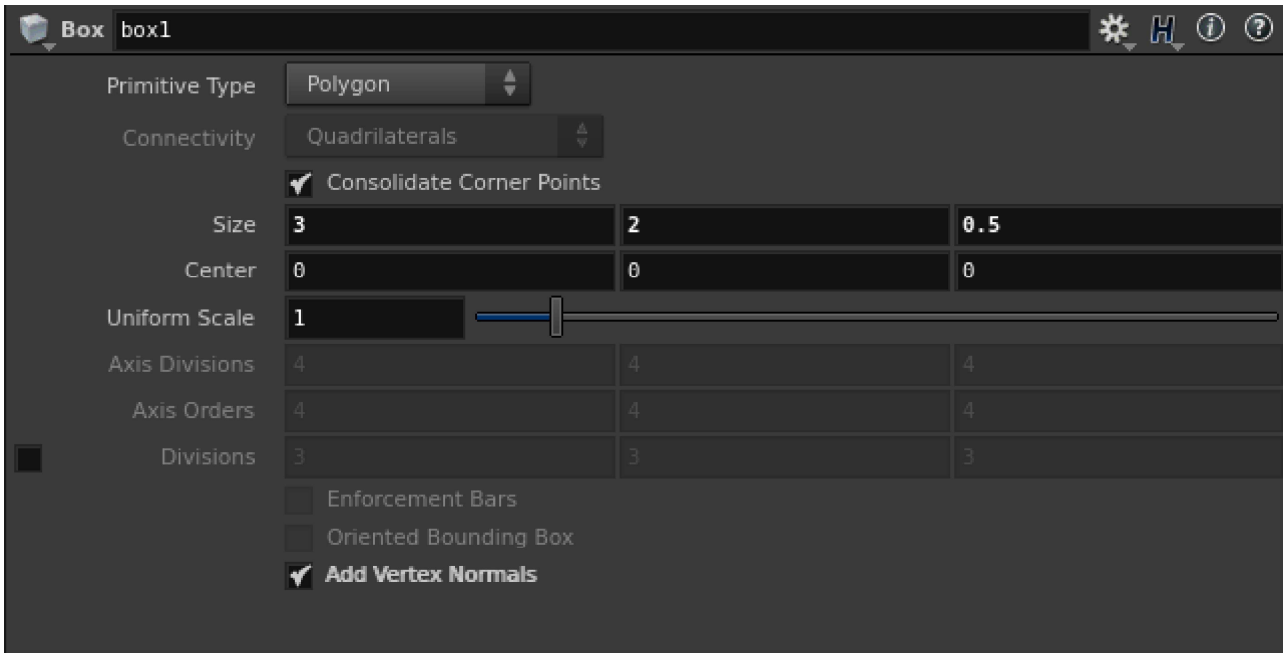
Il nodo di *File* è seguito da un nodo di *Transform* che, come dice la descrizione a fianco ad esso, è stato usato per spostare il palazzo nel centro della scena, un'operazione non necessaria ma comoda solo dal punto di vista della visualizzazione. L'operazione di *Transform* trasforma la geometria della sorgente nello spazio oggetto utilizzando una matrice di trasformazione. In questo caso la matrice di trasformazione è stata utilizzata esclusivamente per traslare la geometria, appartenente al modello del palazzo, riportandola al centro del sistema di riferimento, nei pressi dell'origine degli assi.
A questo punto si può procedere con la fase di prefratturazione. Nello specifico sono stati inseriti dei nodi di *Delete* per poter operare la rottura delle singole parti distintamente, questo si è potuto eseguire perché quando è stato importato il modello si sono anche importati tutti i dati appartenenti alla nomenclatura operata in precedenza. Tramite l'operatore di *Delete* sono stati selezionati solo i componenti del palazzo che si volevano sottoporre a frattura. Questo ha permesso di generare delle fratture differenti e garantire una gestione diversa sul controllo di queste rotture. Quanto detto si può notare nello scatto successivo che riporta una selezione eseguita, appunto, tramite l'operatore di *Delete*.



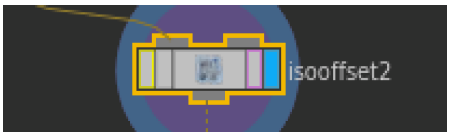
Il nodo di *Delete* è un nodo di geometria, questo operatore può eliminare la geometria in input utilizzando diverse metodologie, in questo caso, all'interno del progetto, in questa prima parte, è stato usato per eseguire una selezione tramite gruppi. Nello specifico, in questo caso, attraverso il nodo di *Delete* dell'immagine precedente, sono stati selezionati solo i piani dal dodicesimo al sedicesimo, eliminando tutte le parti che non sarebbero state coinvolte nella frattura che avrebbe interessato questi elementi. La prefratturazione può essere eseguita attraverso diverse metodologie. Qui di seguito viene descritto uno schema di fratturazione base che include degli operatori essenziali per questo tipo di lavorazioni.



Il nodo più importante in questo blocco è l'operatore evidenziato in giallo, quello che compie effettivamente la procedura di fratturazione. Il primo nodo che compone questo semplice schema è quello che porta l'etichetta *"box1"*.

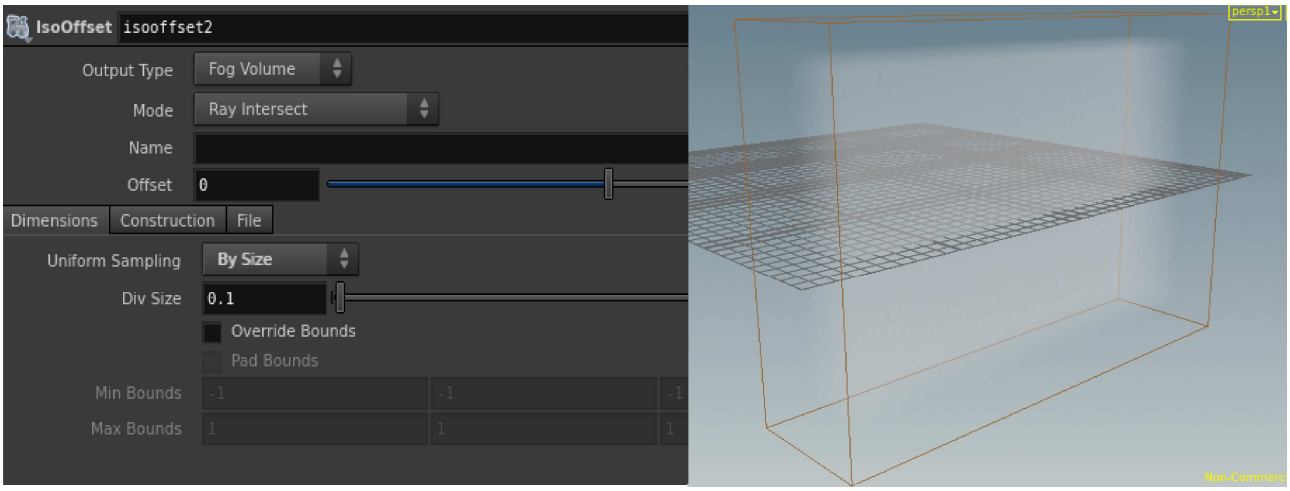


L'operatore *Box* è un nodo di geometria il cui compito è quello di creare un cubo, o un box rettangolare, a sei facce. È anche possibile utilizzare questo operatore, come poi verrà utilizzato in una fase successiva nel progetto di *Destruction*, come *Bounding Box*, racchiudendo un oggetto geometrico al suo interno, sul quale poi verranno effettuate delle operazioni di controllo. Il cubo, all'interno di questo schema di base, ha le seguenti caratteristiche: è un poligono, le cui dimensioni, lungo i tre assi, sono state regolate al fine di creare una scatola rettangolare con un piccolo livello di profondità. Il nodo successivo, presente in questo schema è quello che riporta l'etichetta *"isooffset2"*. L'operatore *Isooffset* è un nodo di geometria che costruisce una superficie di compensazione, o equivalente, partendo da una geometria sorgente, quindi una superfice che differisce rispetto a quella originale di un valore di offset.

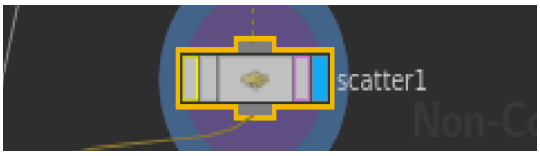


Questo operatore costruisce una funzione implicita data una geometria in input. Questa funzione viene utilizzata per creare un involucro ad un offset, o a una distanza, fissato, dalla superficie originale. Le modalità di produzione del volume permettono alla funzione implicita di essere creata direttamente come un volume primitivo senza ulteriori fasi di elaborazione.

Il settaggio utilizzato, nello specifico, prevede come *Output Type*, ovvero il controllo di quello che viene fatto con la superficie implicita generata, un *Fog Volume*, ovvero il volume primitivo sarà settato a uno all'interno dell'oggetto e a zero al suo esterno, mentre le celle di confine avranno un valore interpolato tra questi estremi, come *Mode*, ovvero il parametro che governa il metodo che viene usato per generare il campo di distanza da riempire, il *Ray Intersect*, ovvero i raggi saranno sparati sulla geometria provenendo da varie direzioni per poter determinare dove è situata la superficie, facendo si che il campo risultante venga riempito, così un offset di zero genererà una isosuperficie separando l'interno dall'esterno. In parole più semplici questo operatore genera un volume all'interno del box di partenza regolabile tramite dei parametri che gli consentono di adattarsi, nel modo desiderato, alla geometria di partenza.



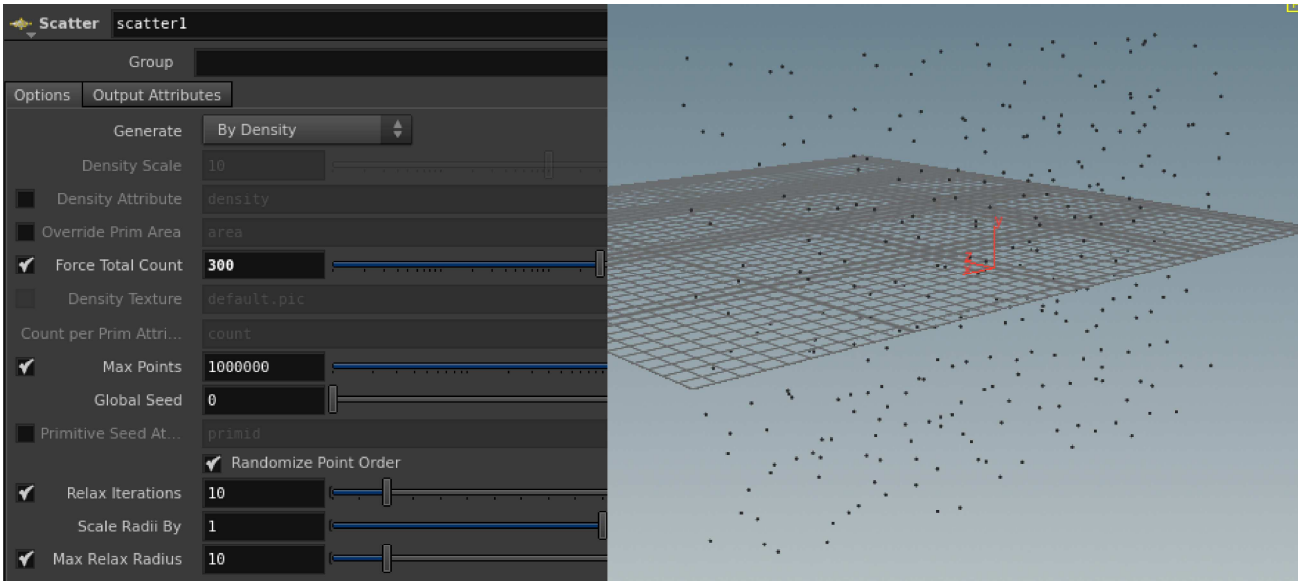
L'operatore successivo è lo *Scatter*, anch'esso un nodo di geometria, che genera punti in modo casuale nella superficie o nel volume.



Questo nodo, nello specifico, distribuisce i punti in modo approssimativamente uniforme consentendo di limitare le aggregazioni e la creazione di buchi. È possibile utilizzare i punti generati per una varietà di scopi: per esempio possono essere utilizzati per specificare delle posizioni da cui possono originarsi delle particelle o per identificare i punti di celle per la creazione di una geometria fratturata.

Tramite questo operatore è possibile specificare la densità, il numero di punti per unità di superficie, la distribuzione dei punti sulla superficie della geometria. I parametri principali che sono stati settati in questo schema, per questo nodo, sono i seguenti:

1- *Randomize Point Order*. Se viene abilitata questa opzione i punti saranno posizionati in ordine casuale, lo svantaggio è che, se più punti vengono generati in un'unica zona cambierà l'ordine dei punti in altri settori anche se le posizioni e gli attributi possono rimanere coerenti.
2- *Relax Iterations*. Quando è attivato questo parametro, i punti saranno spinti distanti dagli altri per evitare raggruppamenti. Questo viene fatto gradualmente per evitare un comportamento caotico. Una distribuzione dei punti in cui essi sono ben separati tra loro è chiamata *"rumore blu"*.
3- *Max Points*. Questo è un limite che si pone in modo opzionale sul numero di punti che vengono generati per evitare che si verifichino delle problematiche, ad esempio quelle legate al calcolo computazionale che potrebbe non essere supportato dalla macchina.
4- *Force Total Count*. Questo parametro, se abilitato, permette di generare esattamente il numero di punti che vengono specificati in questo campo. Questa è un'opzione che si può abilitare solo quando si imposta *Density* come parametro *Generate*.
5- *Generate*. Questo parametro specifica l'approccio generale che viene utilizzato quando si generano dei punti. Se in questo campo viene specificata la modalità *By Density* i punti verranno distribuiti in modo indipendente per ogni primitiva avente una densità.

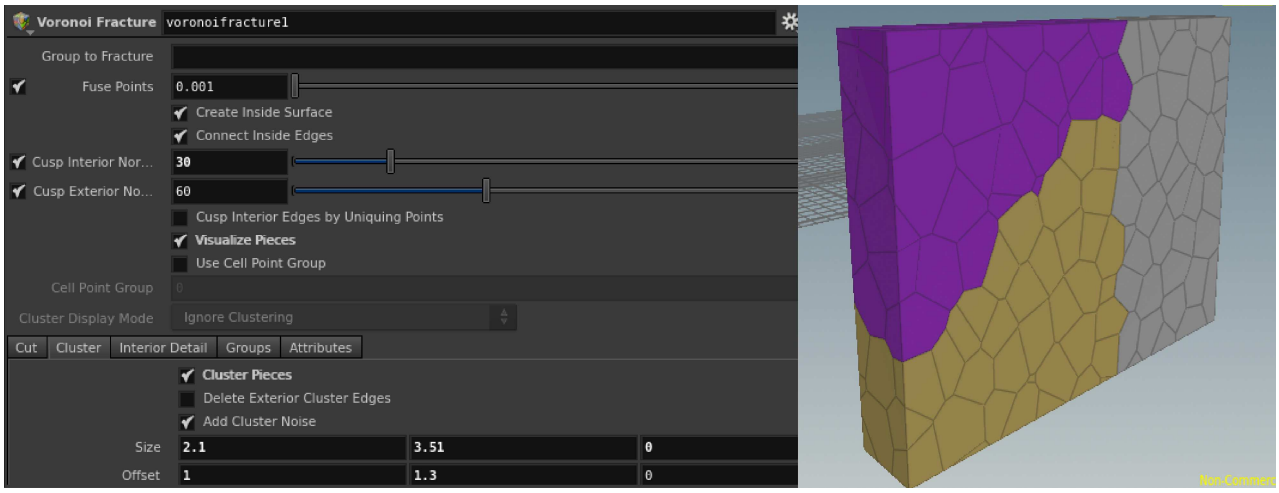


In sintesi, questo operatore, ha permesso, dato un volume in ingresso, di generare 300 punti distribuiti all'interno del volume in maniera uniforme. L'obiettivo della generazione di punti, in questo schema, è stato utile per operare la frattura. Ogni punto è diventato quindi il centro di un pezzo fratturato dal *Voronoi*.

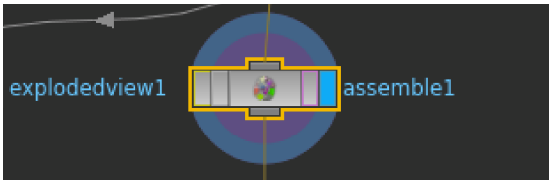
Per l'appunto il blocchetto successivo allo *Scatter* è proprio il nodo di *Voronoi* che ha bisogno di due input per produrre una geometria fratturata, come si può notare nello schema più sopra, uno è la geometria o la mesh da fratturare, l'altro è l'insieme di punti attorno ai quali costruire ogni cella *Voronoi*.

Il *Voronoi Fracture* è un nodo di geometria che esegue l'operazione di frattura, data una geometria in input, mediante una decomposizione *Voronoi* dello spazio attorno a dei punti, dati anch'essi come input. Maggiore è la complessità della mesh o del modello da fratturare, più lenta è l'operazione di fratturazione. I parametri che sono stati impostati all'interno del *Voronoi Fracture*, per lo schema di base preso in analisi, sono i seguenti:

1- *Fuse Points*. I punti della geometria in ingresso sono fusi insieme. Questo permette il giusto rilevamento di oggetti chiusi e fornisce una corretta frantumazione degli oggetti solidi. Tuttavia, questo parametro, può anche cambiare la topologia della mesh in entrata, quindi se la geometria è già stata correttamente fusa insieme, questo parametro non deve essere utilizzato.
2- *Create Inside Surfaces*. Crea delle facce interne.
3- *Connect Inside Edges*. Collega le facce interne della geometria fratturata alle corrispondenti facce esterne.
4- *Cusp Interior Normals*. Calcola le normali ai vertici dei bordi della geometria interna, in modo che essi abbiano un aspetto a forma di cuspide.
5- *Cusp Interior Normals Angle*. Calcola le normali ai vertici dei bordi della geometria interna aventi degli angoli maggiori del valore di questo parametro, in modo che essi abbiano un aspetto a forma di cuspide.
6- *Visualize Pieces*. Consente di visualizzare la geometria fratturata mediante l'applicazione di un colore casuale per ogni pezzo di cui è composta.
7- *Cluster Pieces*. Fonde i singoli pezzi in gruppi più grandi in base ai loro punti di ingresso, i quali condividono un valore comune, diverso da zero, come attributo di cluster. I valori di questo attributo possono provenire da attributi sui punti di ingresso, o da un rumore che viene generato, settato e regolato in questa sezione.

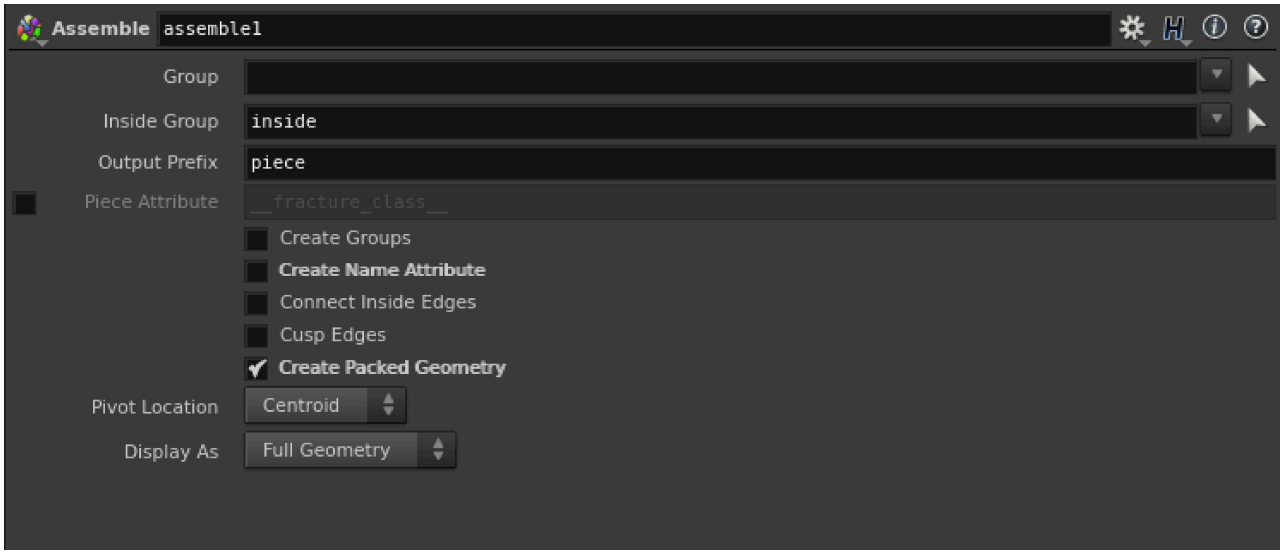


In sostanza quello che fa l'operatore di *Voronoi* è l'operazione di prefratturazione. In questo caso, data la geometria del cubo e i 300 punti in ingesso, genera una geometria fratturata composta da 300 pezzi. L'ultimo importante operatore che conclude la spiegazione di questo schema di base è l'*Assemble*.



Questo blocchetto è un nodo di geometria. L'*Assemble* viene utilizzato per terminare il processo di rottura di un pezzo di geometria. Esso utilizza i gruppi e i pezzi connessi creati dal *Voronoi* per produrre un insieme di pezzi sconnessi. Nello specifico le impostazioni che sono state utilizzate in questo schema, per questo operatore, sono le seguenti:

1- *Inside Group*. Questo è il parametro che riporta il nome del gruppo corrispondente alle facce interne create dall'operatore di *Voronoi* precedente. Questo nome dovrebbe rimanere coerente per tutte le fratture che interessano la stessa geometria.
2- *Output Prefix*. Questo indica il prefisso applicato all'attributo *"nome"* e ai nomi dei gruppi creati dall'operatore. Ad esempio, un oggetto suddiviso in 3 parti che ha un prefisso di uscita del pezzo avrà tre gruppi di nome piece0, piece1 e piece2.
3- *Create Packed Geometry*. Se la geometria di uscita contiene un attributo primitivo chiamato *"nome"*, verrà creato un frammento impacchettato per ogni valore univoco dell'attributo *"nome"*. Inoltre, un attributo di punto chiamato "nome" verrà creato per identificare il pezzo che il frammento impacchettato contiene. In caso contrario, la geometria di uscita sarà imballata in una primitiva impacchettata, incastrata e il prefisso di uscita verrà utilizzato per creare l'attributo di punto.

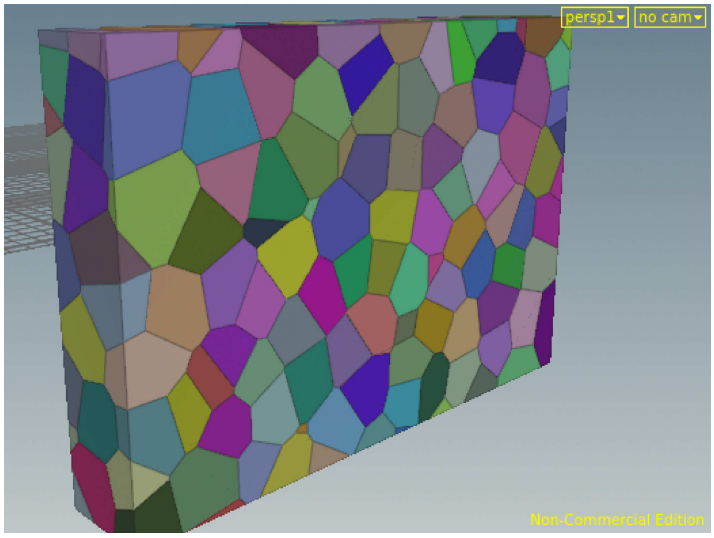


```
Full Name:     /obj/Ball_Against_Wall/Wall/box1
Operator type: box

     8 Points
     6 Primitives
    24 Vertices
     6 Polygons

  Size:    3,  2,   0.5
Center:    0,  0,     0
Bounds:  1.5,  1,  0.25
       -1.5, -1, -0.25

     1 Point Attributes: P[3fp]

     1 Vertex Attributes: N[3fn]

Memory: 7.33 KB; Unique: 6.40 KB

Last Cook Time: 906.72 ms
   Total Cooks: 1

Created  Time: 30 Mar 2016 03:46 PM
Modified Time: 20 Nov 2016 11:02 AM
Time Dependent Cook: No
```

```
Full Name:     /obj/Ball_Against_Wall/Wall/assemble1
Operator type: assemble
Version:       1
Defined by:    /Library/Frameworks/Houdini.framework/Versions/15.0.416/Resources/houdini/otls/OPlibSop.hda?Sop/assemble

   300 Points
   300 Primitives
   300 Vertices
   300 Packed Fragments

  Size:    3,  2,   0.5
Center:    0,  0,     0
Bounds:  1.5,  1,  0.25
       -1.5, -1, -0.25

     2 Point Attributes: P[3fp], name[1s;300]

     1 Primitive Attributes: path[1s;300]

Memory: 310.40 KB (Instanced)
Subnetwork output from: assemble1/switch_pack
Contains 23 OPs

Synchronized with definition

Last Cook Time: 0.06 ms
   Total Cooks: 4

Created  Time: 30 Mar 2016 04:00 PM
Modified Time: 22 Nov 2016 10:49 AM
Time Dependent Cook: No
```

```
Full Name:     /obj/Ball_Against_Wall/Wall/voronoifracture1
Operator type: voronoifracture
Version:       3
Defined by:
/Library/Frameworks/Houdini.framework/Versions/15.0.416/Resources/houdini/otls/OPlibSop.hda?Sop/voronoifracture

 5,994 Points
 3,597 Primitives
17,982 Vertices
 3,597 Polygons

     2 Primitive Groups:
3,264 primitives in inside
333 primitives in outside

  Size:    3,  2,   0.5
Center:    0,  0,     0
Bounds:  1.5,  1,  0.25
       -1.5, -1, -0.25

     1 Point Attributes: P[3fp]

     1 Vertex Attributes: N[3fn]

     2 Primitive Attributes: Cd[3fc], name[1s;300]

     1 Detail Attributes: varmap[0s;1]

Memory: 1017.80 KB (Instanced)
Subnetwork output from: voronoifracture1/OUT
Contains 59 OPs

Synchronized with definition

Last Cook Time: 2.44 ms
   Total Cooks: 2

Created  Time: 30 Mar 2016 03:49 PM
Modified Time: 22 Nov 2016 11:34 AM
Time Dependent Cook: No
```

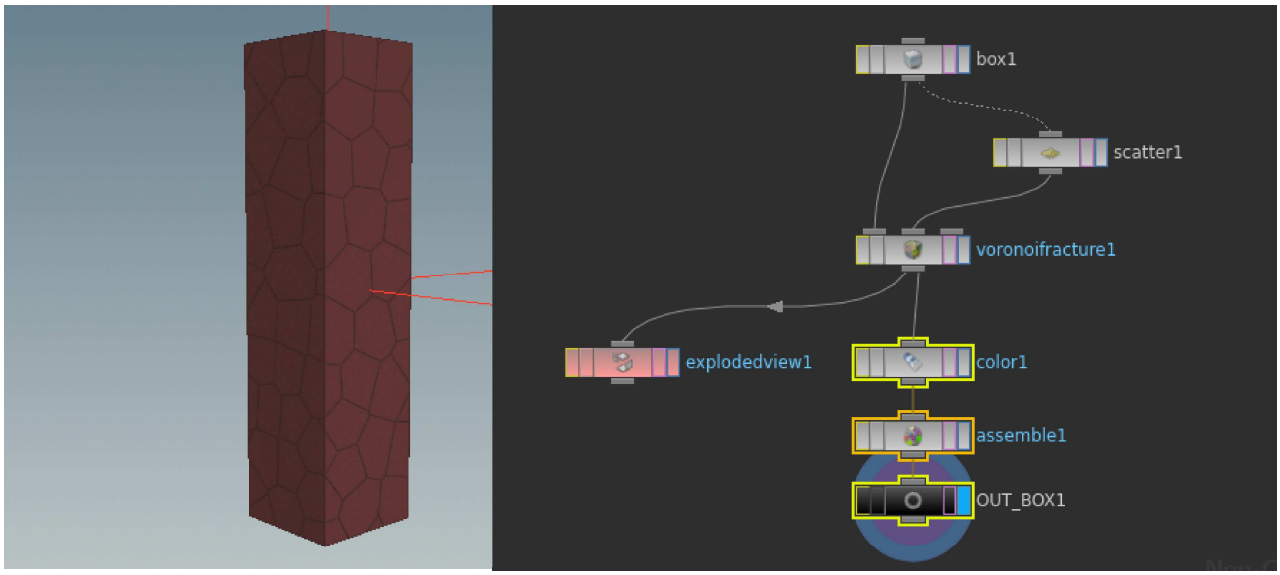
```
Full Name:     /obj/Ball_Against_Wall/Wall/isooffset2
Operator type: isooffset

     1 Points
     1 Primitives
     1 Vertices
     1 Volume

  Size:  3.4,  2.4,   0.9
Center:    0,    0,     0
Bounds:  1.7,  1.2,  0.45
       -1.7, -1.2, -0.45

     1 Point Attributes: P[3fp]

Volume Resolutions:
  0 : [34, 24, 9] Voxel Count: 7,344,

Memory: 34.84 KB; New: 34.41 KB

Last Cook Time: 1.135s
   Total Cooks: 1

Created  Time: 30 Mar 2016 03:52 PM
Modified Time: 20 Nov 2016 11:06 AM
Time Dependent Cook: No
```




```
Full Name:     /obj/Ball_Against_Wall/Wall/scatter1
Operator type: scatter::2.0

   300 Points
     0 Primitives
     0 Vertices

  Size:    3.04544,    2.06111,   0.563899
Center: 0.00834972, 0.00194198, 0.00123085
Bounds:    1.53107,     1.0325,    0.28318
         -1.51437,   -1.02861,  -0.280718

     1 Point Attributes: P[3fp]

Memory: 8.91 KB; New: 8.48 KB; Unique: 4.95 KB

Last Cook Time: 488.88 ms
   Total Cooks: 1

Created  Time: 30 Mar 2016 03:50 PM
Modified Time: 22 Nov 2016 10:24 AM
Time Dependent Cook: No
```

In sostanza l'*Assemble* permette di impacchettare la geometria e ridurre il suo peso, questo per poter eseguire operazioni successive che magari richiedono più complessità di gestione, come ad esempio le simulazioni. Si prenda in considerazione l'esempio precedente e le immagini di riferimento qui sopra e si osservi più nel dettaglio cosa generano i vari operatori per capire nel profondo il lavoro svolto dall'*Assemble*. Si analizzi questa procedura deselezionando la voce cluster nel *Voronoi* che, come si vede nella foto, permette di vedere i 300 pezzi che sono stati creati durante l'operazione di frattura. Ora si osservi il flusso partendo dal primo operatore: il box riporta in uscita una geometria che ha 8 punti, 6 primitive, 24 vertici e 6 poligoni, questo perché è stata creata una scatola composta da 6 facce. Escludendo l'*Isoofset*, che in questa considerazione è poco rilevante, si passi allo *Scatter,* che genera esattamente 300 punti. Il *Voronoi* crea ben 5991 punti, 3597 primitive, 17982 vertici e 3597 poligoni. In questo caso se si volesse eseguire una simulazione senza impacchettarla si avrebbe a che fare con un sacco di elementi da calcolare. Per questa ragione viene utilizzato l'*Assemble*, difatti si può notare che una volta eseguito l'impacchettamento ci si ritrova con 300 punti, 300 primitive, 300 vertici e 300 frammenti impacchettati. Questi frammenti impacchettati corrispondono esattamente al numero di pezzi creati dalla fratturazione della geometria in ingresso. Questa è un'operazione molto importante quando si affrontano questo tipo di lavorazioni. Un'altra operazione che svolge il nodo di *Assemble* è quella di inserire un attributo identificativo che permette una migliore gestione dei vincoli e della simulazione quando si ha a che fare con un progetto di *Destruction* composto da molti elementi da fratturare con diverse metodologie a seconda della frattura che si vuole ottenere. In un progetto di questa portata ogni geometria dovrà avere un diverso attributo per poter essere identificata correttamente. Prima di continuare il racconto del processo di *Destruction* svolto all'interno del progetto in questione, viene aperta, di seguito, una piccola parentesi per descrivere alcune metodologie per la gestione e il controllo delle rotture. *Houdini* offre diversi operatori per affrontare il lavoro di *Destruction*, di seguito vengono riportati alcuni esempi di collegamenti fra nodi che permettono di gestire, appunto, il lavoro di prefratturazione.

1) In questo primo esempio è possibile notare uno schema ancora più basico di quello visto in precedenza. Qui è presente solo uno *Scatter* che crea 100 punti distribuiti nella geometria in modo uniforme. Difatti se si deselezionasse la voce *Relax iterations* la frattura avrebbe un aspetto più casuale e meno regolare. Gli altri operatori, come tipologia di parametri settati, hanno più o meno le stesse caratteristiche dello schema precedente.

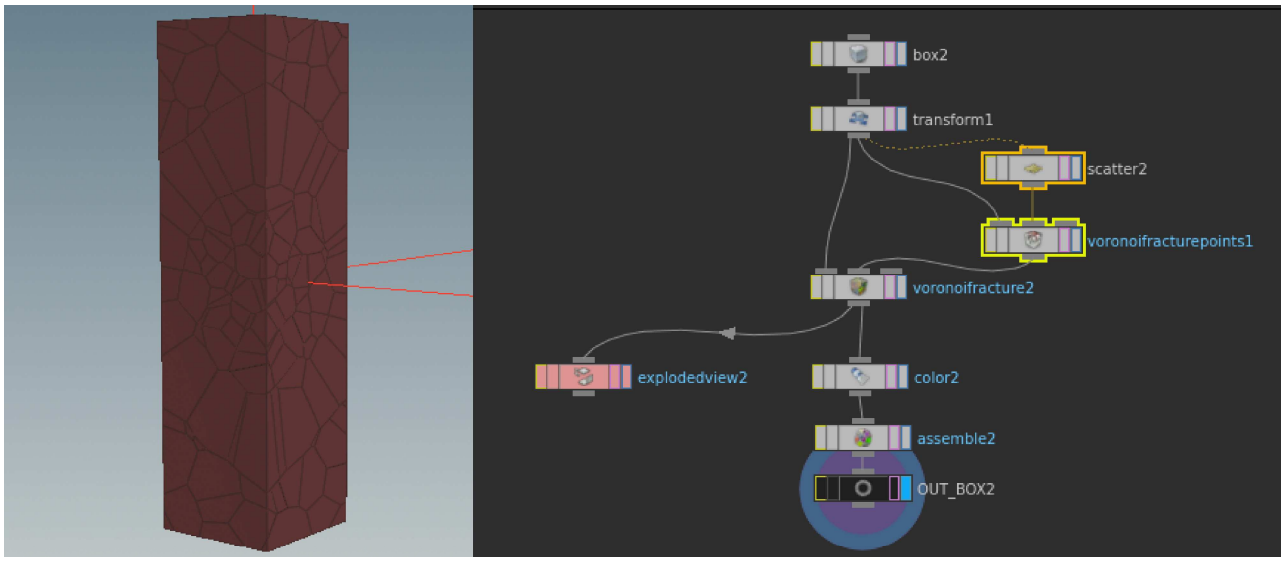


2) Questo secondo esempio differisce dal precedente solo per l'aggiunta di un operatore: il *Voronoi Fracture Points*. Questo è un nodo di geometria che è stato posizionato subito dopo lo *Scatter*. Questo blocchetto riceve in ingresso una geometria e dei punti di impatto, che in questo caso sono tre, e genera un insieme di punti che vengono usati come input per il *Voronoi Fracture* per eseguire la fratturazione della geometria. Questo operatore crea un volume di impatto composto da metaballs che vengono posizionate su ciascun punto d'impatto, in questo caso avendo tre punti sono state generate tre metaballs. Il secondo ingresso può anche accettare direttamente una primitiva *MetaBall*, che sarà poi fusa con le metaballs create dal *Voronoi Fracture Points* per formare il volume d'impatto. Questo volume viene poi utilizzato per dividere l'oggetto di input in tre regioni: la regione di superficie è la superficie dell'oggetto che interseca il volume impatto. la regione interna è l'intersezione tra il volume dell'oggetto e il volume impatto, la regione esterna è il volume dell'oggetto che è fuori dal volume d'impatto.

In poche parole, questo operatore, come già accennato, crea delle metaball, delle sfere, che vengono inserite nell'intorno dei punti in ingesso che sono stati distribuiti sulla geometria tramite lo *Scatter*. Questo permette di concentrare la frattura della geometria in ingresso maggiormente in queste regioni cercando di creare una frattura meno regolare. Il settaggio utilizzato per l'operatore *Voronoi Fracture Points* ha i seguenti parametri:

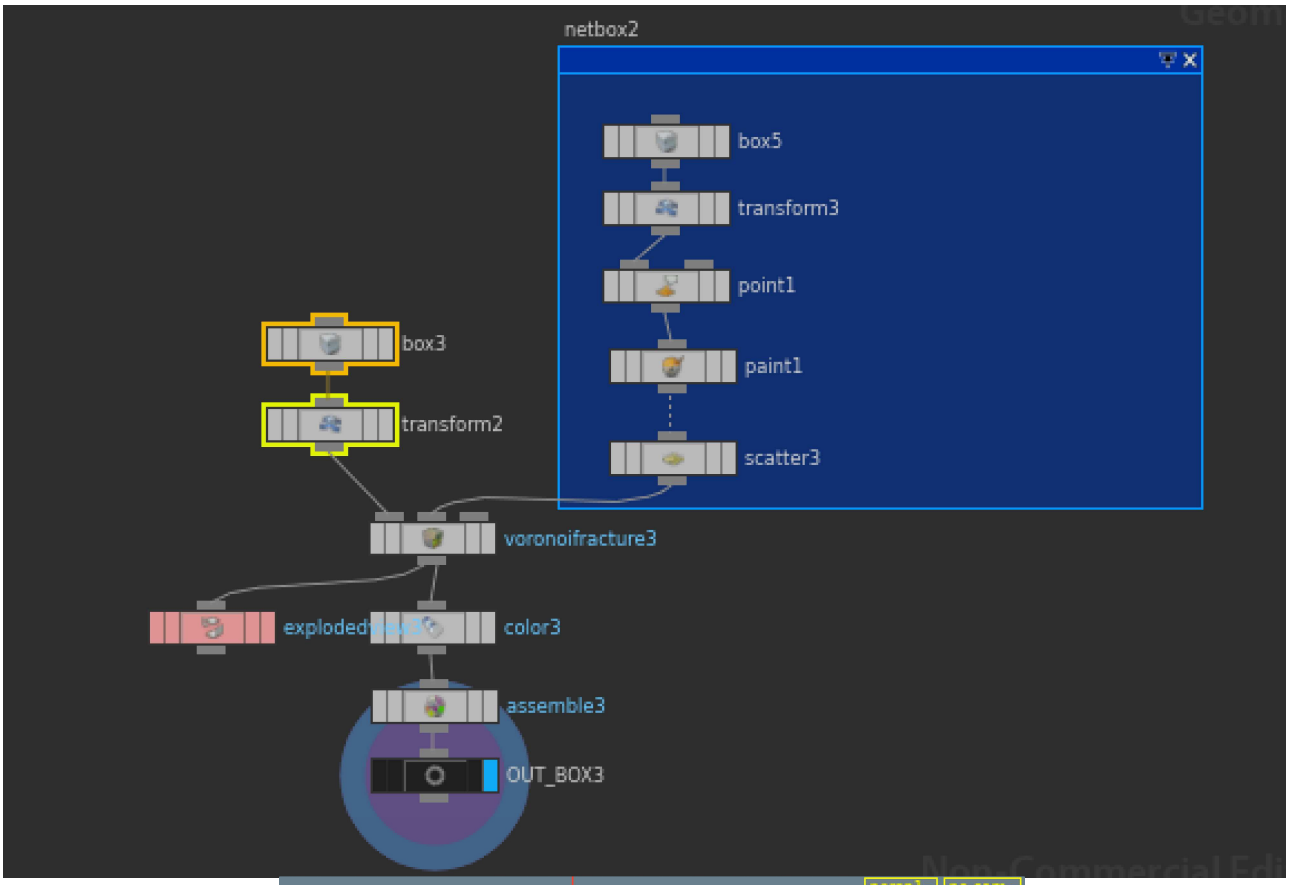
1- *Raggio di impatto*. Qui viene specificato il raggio delle metaball.
2- *Point per Area*. Viene specificato il numero di punti per unità dell'area della superficie, cioè i punti che stanno all'interno delle metaball.

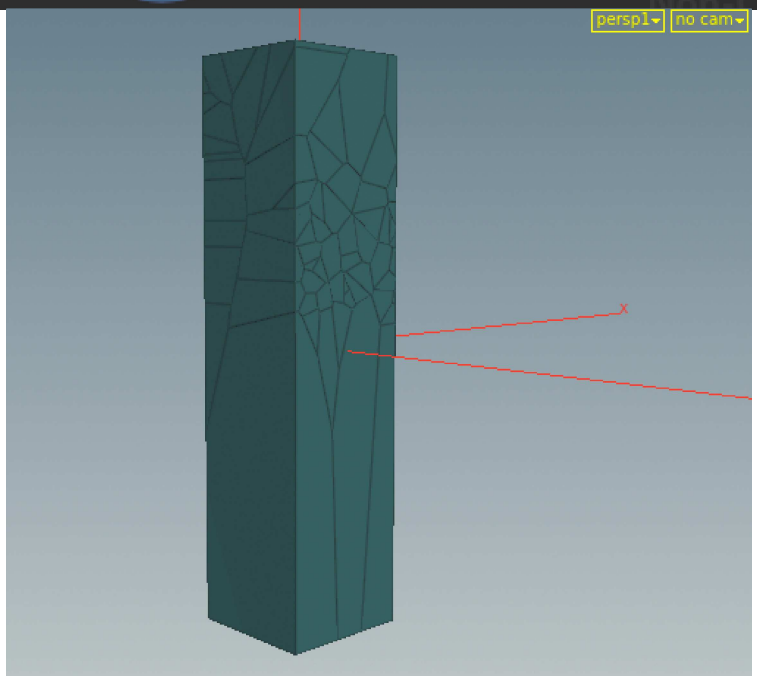
Poi si vanno a determinare i parametri per la costruzione dei punti per le regioni di superficie, interna ed esterna, attivando il parametro *Clustering*, il cui effetto potrà essere modificato attivando il parametro *Cluster Pieces* all'interno del *Voronoi Fracture*.



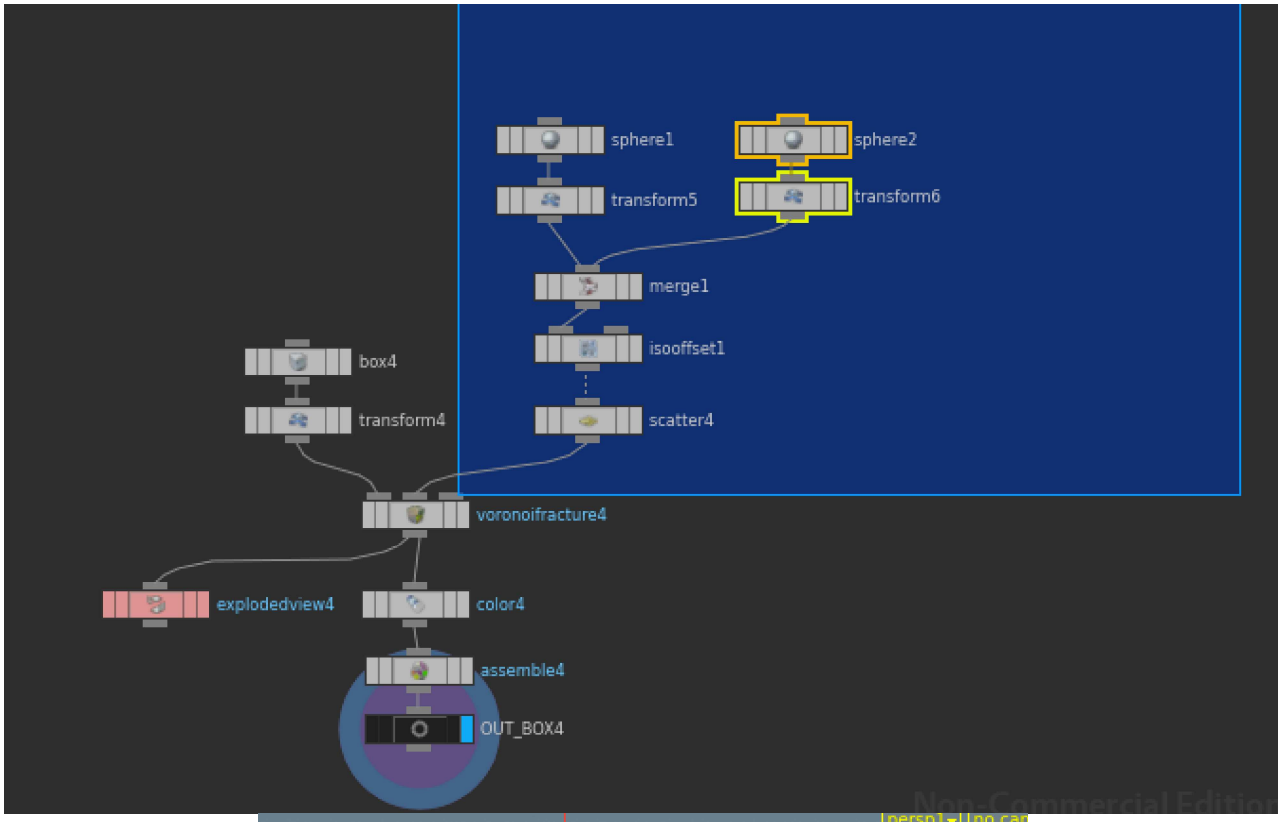
3) In questo blocco la parte che differisce rispetto agli esempi precedenti è tutto il ramo che precede lo *Scatter*. Con il nodo di *Box* è stata creata una mesh poligonale la cui superficie è stata suddivisa in quadratini. Il nodo di *Trasform* ha permesso di creare una riproduzione esatta della geometria in input al *Voronoi Fracture* posizionata alle sue stesse coordinate, quindi la copia è stata sovrapposta esattamente alla geometria in input al *Voronoi* da fratturare. È stato aggiunto l'operatore *Point* che ha permesso la creazione di un attributo di punto: è stato aggiunto come attributo di punto il colore nero. L'operatore *Paint* successivo ha consentito di colorare la superficie della mesh creata e colorata di nero con un altro colore diverso da questo. In questo modo, tramite lo *Scatter*, si sono potuti distribuire i punti solo nella parte di mesh colorata sovrascrivendo il colore nero generato con la creazione dell'attributo di punto.

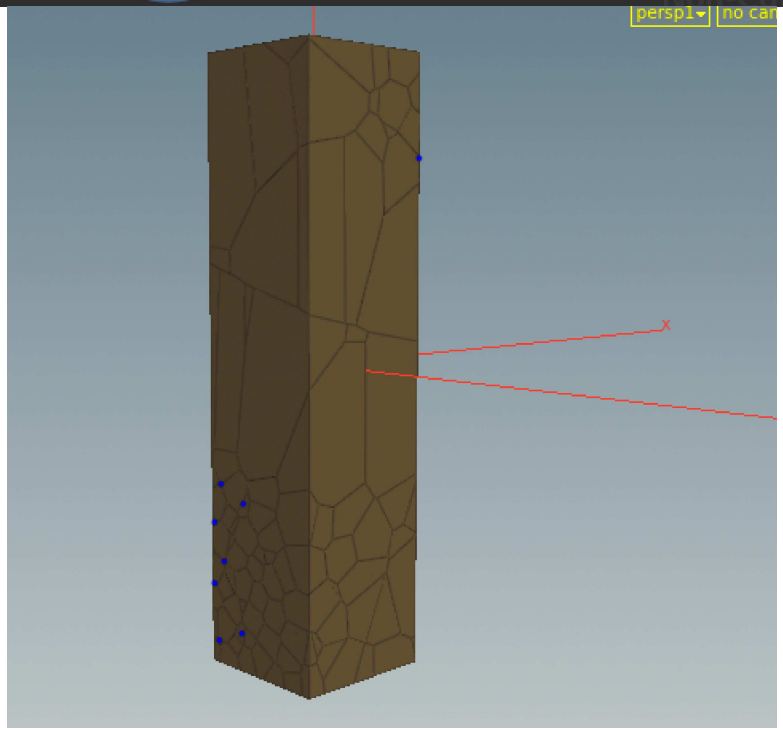
Per fare in modo che lo *Scatter* funzioni correttamente è necessario attivare il parametro *Density Attribute* specificando che si sta lavorando sul colore, inserendo quindi in questo campo la dicitura *Cd* che specifica, appunto, l'attributo associato al colore. Poi si possono andare a determinare il numero di punti da generare come è sempre stato fatto negli schemi precedenti.



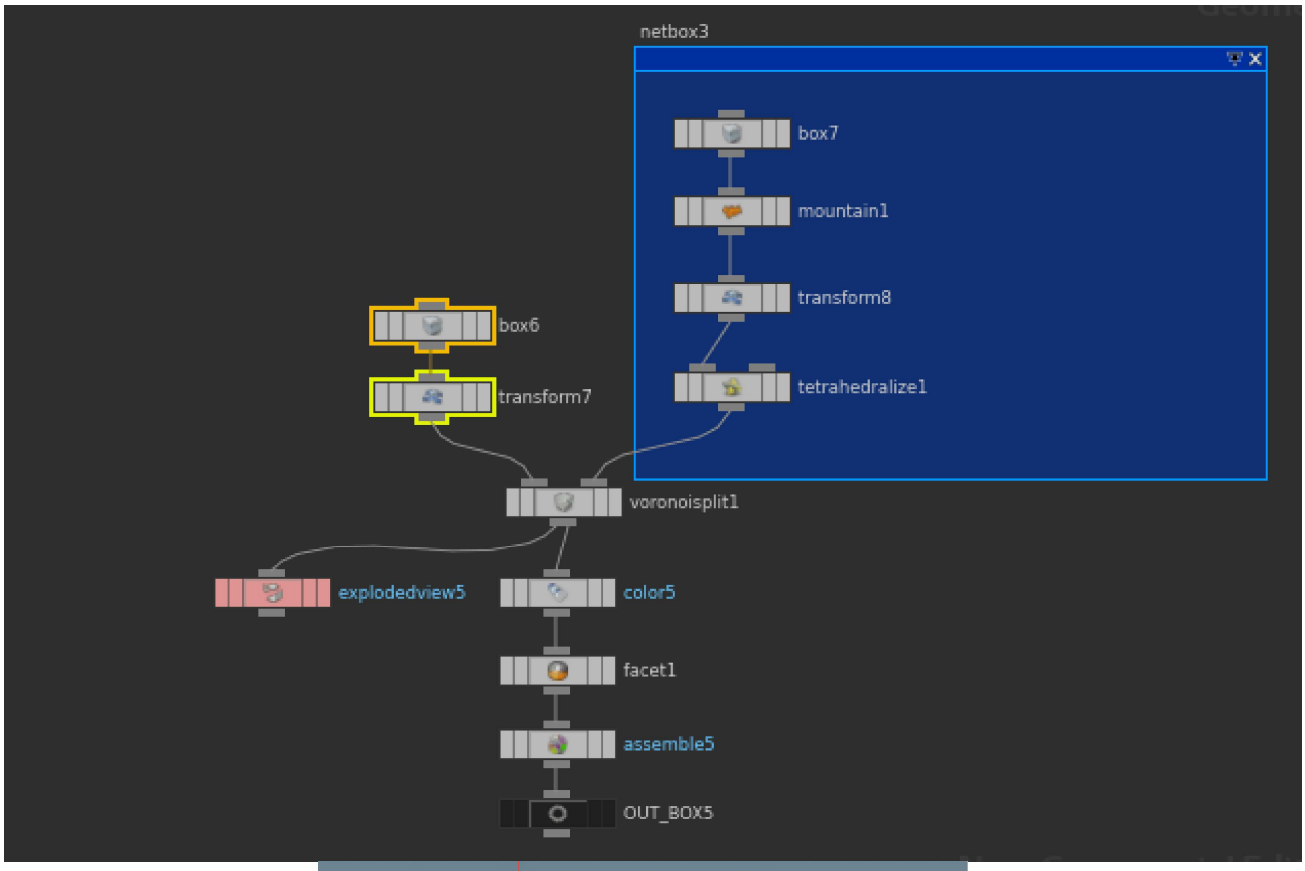


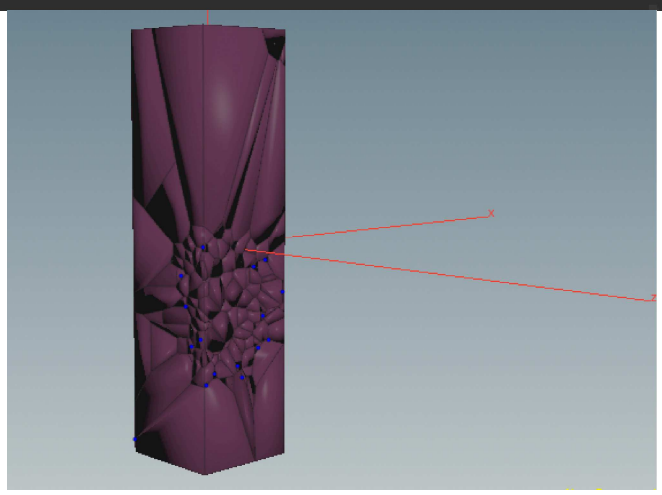
4) Anche in questo schema, quello che differisce dai precedenti è il ramo che precede lo *Scatter*. In questo settaggio sono state create due sfere che sono state posizionate agli estremi della geometria da fratturare in input al *Voronoi*. Le sfere sono delle primitive che differiscono l'una dall'altra in fatto di dimensioni, operazione eseguita tramite il nodo di *Trasform*. Il nodo di *Merge* unisce in un'unica primitiva le sfere precedenti. L'*Isooffset* permette di creare il volume all'interno delle sfere nelle quali poi lo *Scatter* creerà i punti che verranno dati in pasto al *Voronoi* per eseguire la frattura.



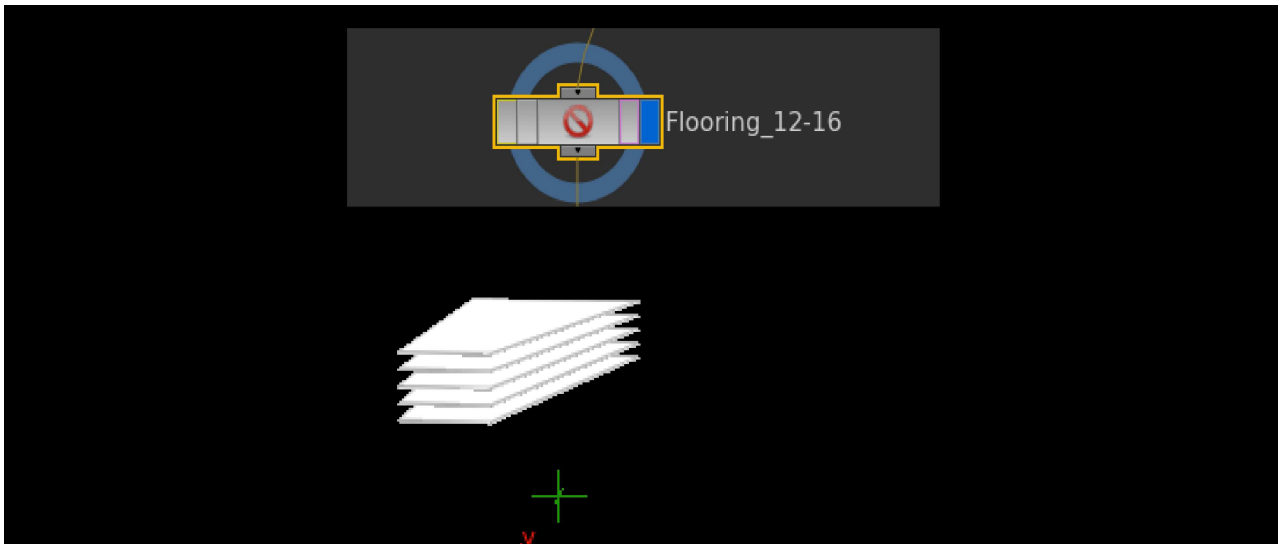


5) In questo schema ci sono molte differenze rispetto a quelli precedenti. Sono presenti due rami che entrano in un nodo che non è il solito operatore di *Voronoi Fracture* ma è il *Voronoi Split*. Quest'ultimo è un nodo di geometria, esso taglia la geometria in piccoli pezzi, in accordo con un insieme di tagli definiti da delle polilinee. Il *Voronoi Split* taglia la geometria in base a una serie di polilinee quindi, queste linee possono essere create al meglio con l'operatore di *Tetrahedralize*. Il nodo *Voronoi Split* è simile al nodo di *Vornoi Fracture*; esso riceve in ingresso una geometria da fratturare e dall'altro l'insieme delle polilinee che sono state create, come la regola dice, tramite il nodo di *Tetrahedralize*. Con questa metodologia è stato possibile fratturare la geometria. Il nodo di *Voronoi Fracture* però è molto più completo rispetto al *Voronoi Split* quindi per eseguire questo tipo di lavorazioni sarebbe meglio sempre utilizzare l'operatore di *Vornoi Fracture*.

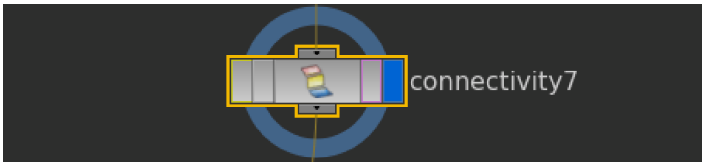




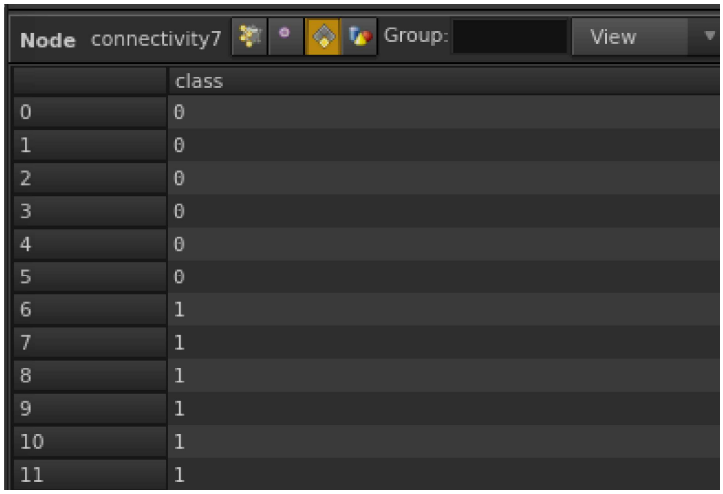
Queste sono, come già detto, alcune metodologie che permettono di eseguire l'operazione di fratturazione e che consentono di esercitare un diverso tipo di controllo sulla rottura della geometria. Tornando al progetto di *Destruction*, ora vengono analizzate le diverse tipologie di fratturazioni che sono state utilizzate per rompere i diversi elementi che compongono la geometria del modello 3D del palazzo.



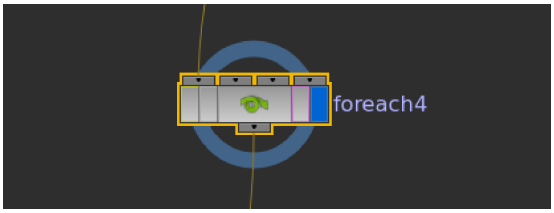
Vengono selezionati, come si può notare nella foto qui sopra, alcuni piani del modello del palazzo, nuovamente tramite il nodo di *Delete,* e poi viene analizzata la gestione del processo di fratturazione per questi elementi.



Il nodo di *Connectivity* è un nodo di geometria. Esso crea un attributo con un valore unico per ogni insieme di primitive o punti connessi. Il nome predefinito per questo attributo è *Class*. Il parametro *Connectivity Type* determina se l'attributo sarà un attributo di punto o di primitiva, in questo caso è stato scelto di creare questo attributo, chiamato *Class*, sulle primitive, in modo tale che ciascun piano riportasse un identificativo univoco. Ogni piano è composto da un insieme di primitive, nello specifico 6, infatti se si osservano le proprietà si hanno in totale 30 primitive, proprio perché sono presenti 5 piani. Questo nodo non fa altro che assegnare un numero che identifica in maniera univoca le primitive che compongono ciascun piano: le primitive che costituiscono lo stesso piano hanno lo stesso valore, mentre le primitive che si trovano su piani diversi hanno un valore differente. Questo discorso risulta chiaro nell'immagine successiva, dove si può notare che per l'attributo *Class* si hanno per le prime sei primitive che appartengono allo stesso piano il valore zero mentre per le successive sei il valore uno e così via.



Questa operazione, eseguita tramite il nodo di *Connectivity*, è stata fondamentale per eseguire tutte quelle successive: è stata utile per poter eseguire un loop di fratturazione. Infatti il nodo che segue è il *For Each Subnetwork*.



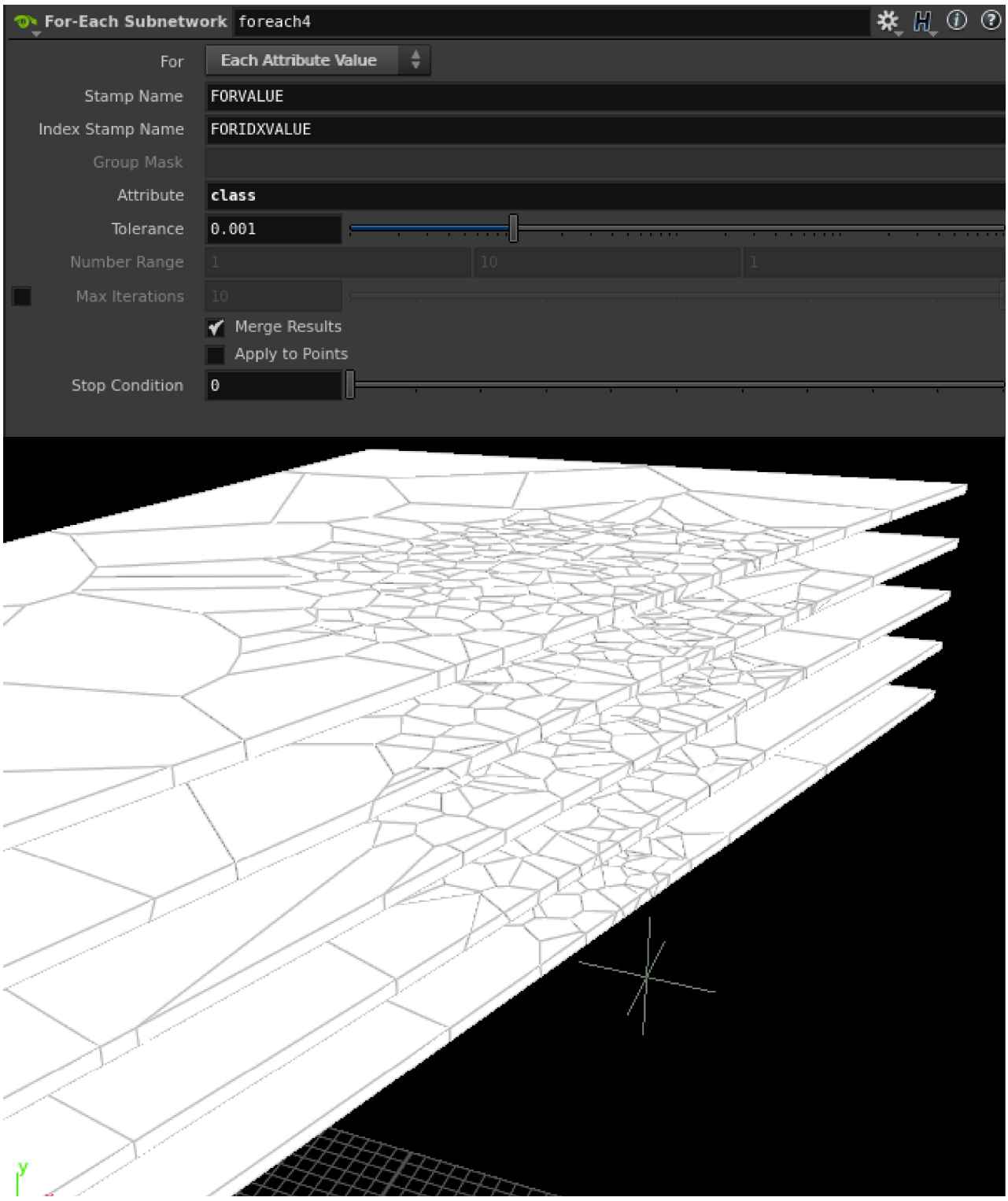
Questo è un nodo di geometria che in modo iterativo applica uno o più operatori, contenuti al suo interno, a una geometria in ingresso. Questo operatore ha due funzioni principali:

1- *For Each Group/Attribute*. Che rompe la geometria in ingresso in pezzi che corrispondono a ciascun gruppo, o al valore di attributo, ed esegue i *SOP* per ciascuno di questi pezzi. Il risultato viene poi fuso insieme per creare un output.
2- *For Each Number*. Che ripete l'insieme dei *SOP* per un determinato numero di volte.

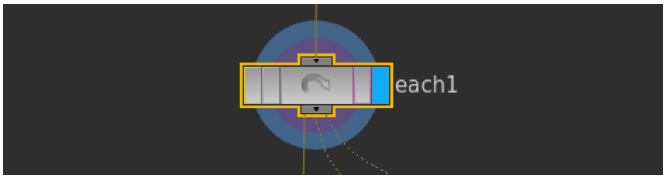
Il *SOP For Each* in tutti i casi lavora con gruppi di primitive o attributi di primitive. Altri gruppi o attributi devono essere promossi a *Primitive* prima che possano essere utilizzati. Questo giustifica il passaggio precedente nel quale è stato creato un attributo di primitiva chiamato *Class*. Il settaggio utilizzato in questo nodo è il seguente:

1- *For Each Attribute Value*. Per l'attributo primitivo dato, questo parametro determina quanti valori unici ha nella geometria di ingresso. Esegue i *SOP* contenuti per ogni valore di attributo unico, con la stampa impostata sul valore dell'attributo. La tolleranza è utilizzata per determinare ciò che è considerato unico.
2- *Stamp Name*. Per ogni iterazione del *For Each*, il valore corrente dell'iterazione sarà stampato in questa variabile.
3- *Stamp Index Name*. L'attuale iterazione del *For Each* sarà stampata in questa variabile. I passi di stampa partono da zero.
4- *Attribute*. Questo è il nome dell'attributo di primitiva utilizzato quando si utilizza il *For Each Attribute Value*.

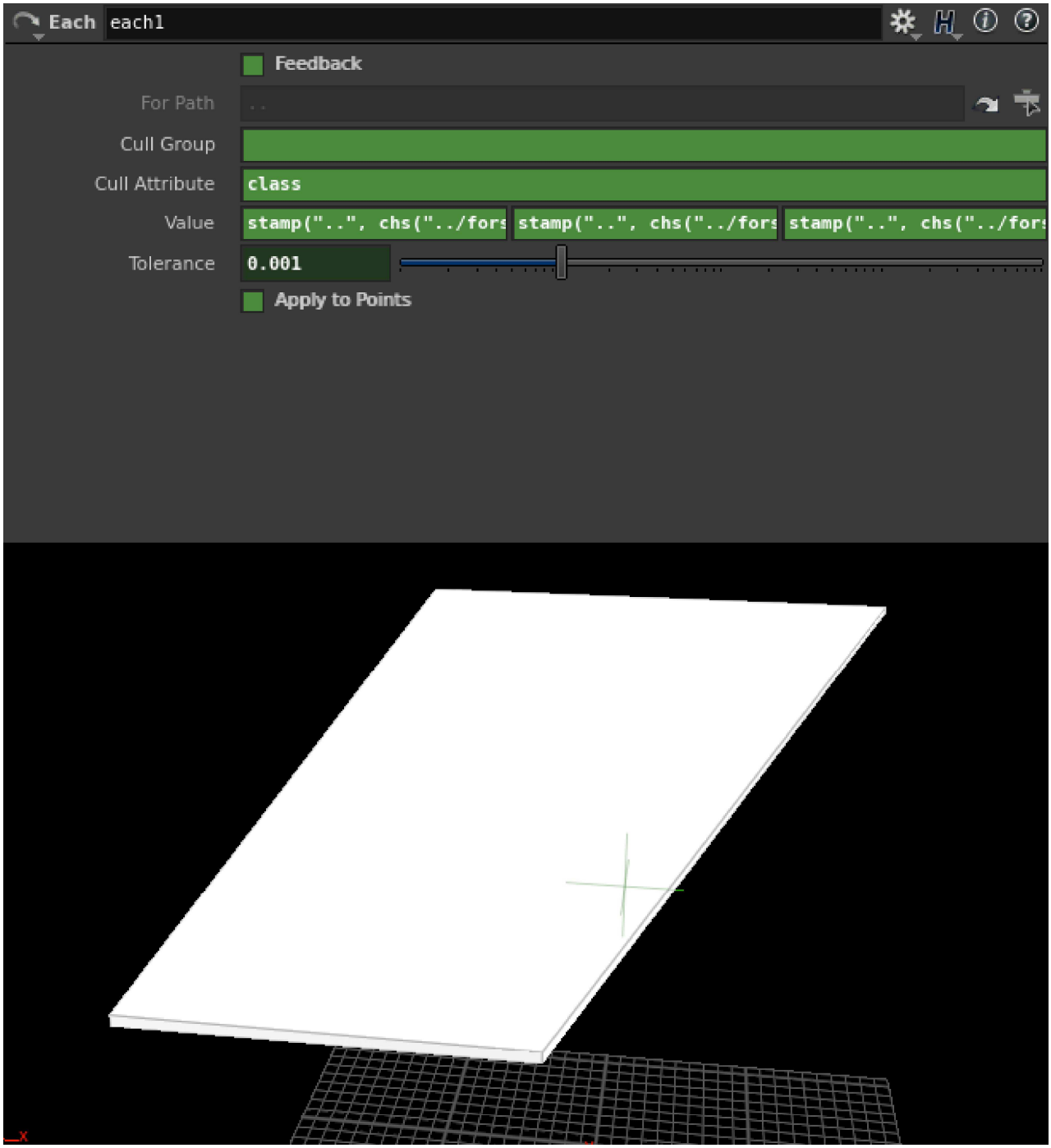
In sostanza questo nodo, per ogni iterazione eseguita sull'attributo *Class*, posto sulle primitive, esegue una serie di operazioni, in pratica è un modo veloce per eseguire in modo automatico l'operazione di fratturazione per quest'intero blocco di piani, come se fosse eseguita singolarmente per ogni piano. Tramite questa procedura è anche stato possibile eseguire una frattura leggermente diversa per ogni piano, modificando dei piccoli parametri nelle impostazioni dei nodi di *Scatter* posti all'interno del *For Each*.



Scendendo più in profondità all'interno del *For Each* si incontra il nodo di *Each*.



Questo è un nodo di geometria che esegue l'operazione di selezione della geometria di input in accordo con le specifiche del *For Each*, quindi in questo caso tramite l'attributo *Class*, per ogni iterazione, andrà a selezionare un piano alla volta sul quale verranno eseguite tutte le operazioni successive.



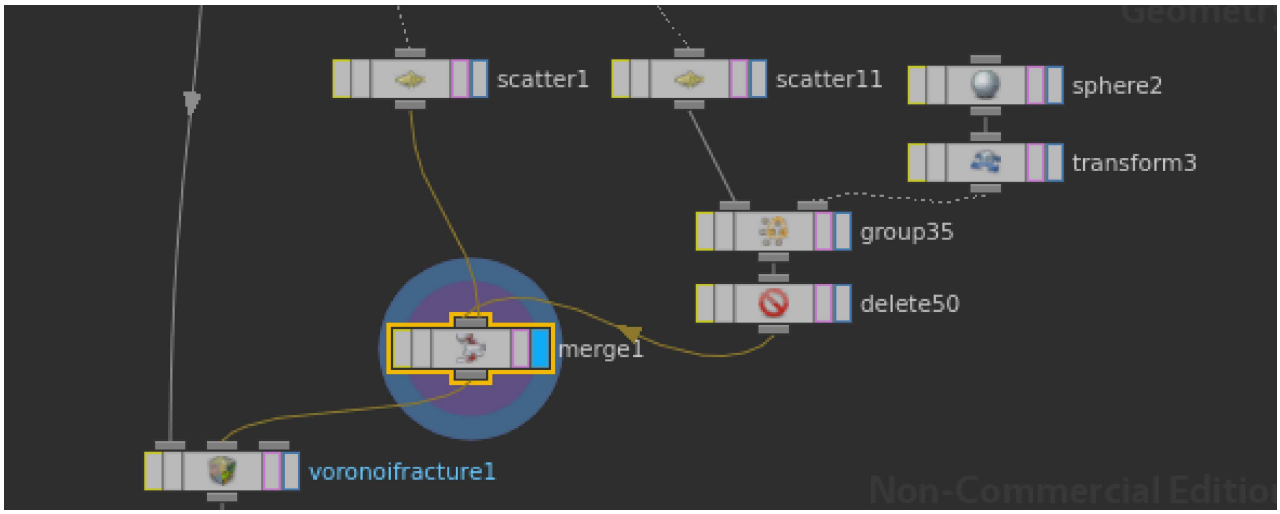
Nello specifico:

1- *Cull Attribute*. Se viene specificato un attributo di primitiva, solo le primitive il cui attributo corrisponde a quello specificato dentro la tolleranza data verranno copiate dall'ingresso.
2- *Value*. Il valore dell'attributo deve corrispondere per essere poi copiato dall'ingresso. Questo è comunemente un'espressione di stampa del tipo *Stamp("..", "FORVALUE", 0)*.

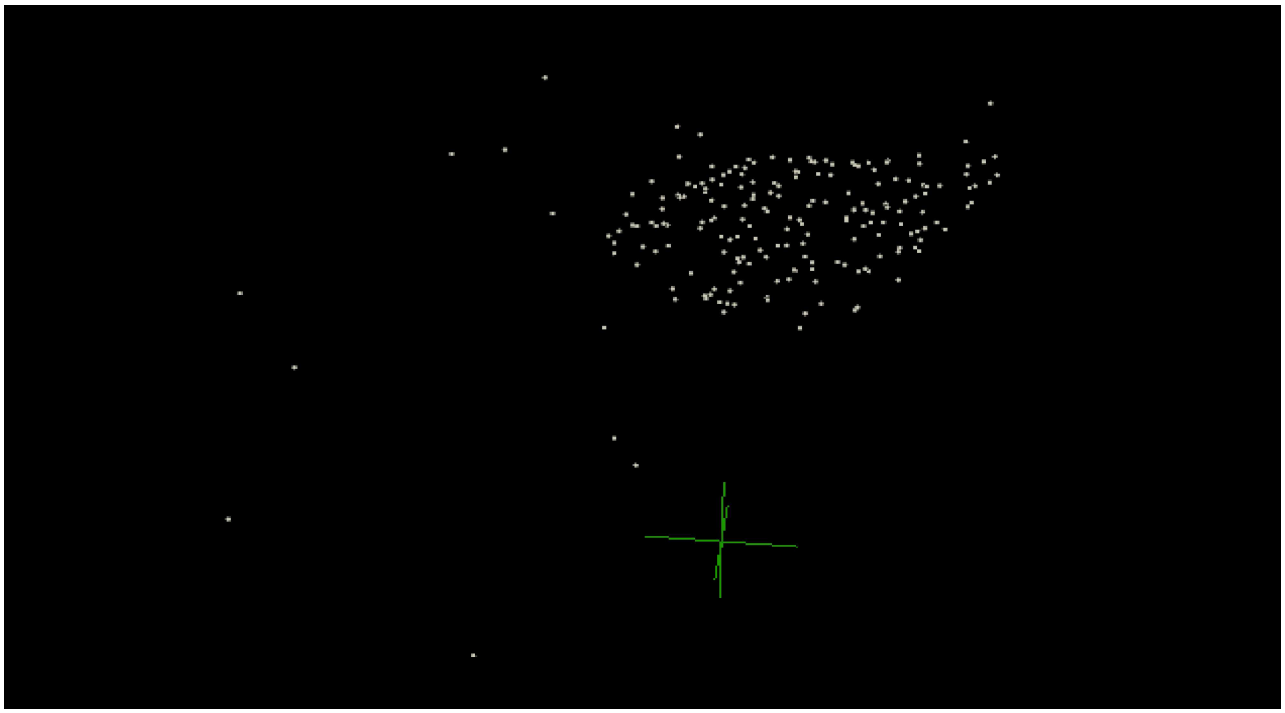
Per il parametro *Value* è presente, infatti, la seguente stringa:

$$stamp(\text{"..",chs(../forstamp)} + \text{"1"}, 0)$$

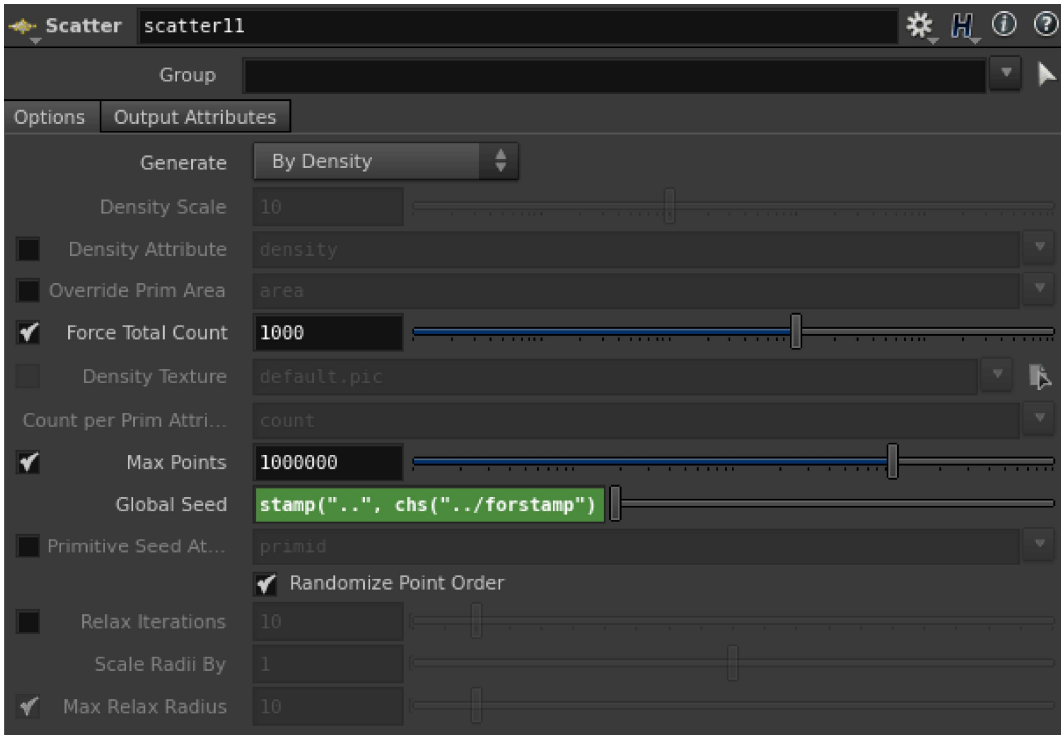
che riporta in pratica il valore dell'attributo *Class* presente all'iterazione corrente. In questo caso nella figura viene selezionato il piano che riporta il valore dell'attributo *Class* pari a zero perché in questo caso si stava eseguendo la prima iterazione.
Poi è presente il blocco che esegue effettivamente la procedura di fratturazione, composto come segue.



La figura qui sopra riporta un nodo di *Voronoi* che riceve i due input, da un lato la geometria da fratturare, selezionata dal nodo di *Each*, che cambierà ad ogni iterazione, e dall'altro l'insieme dei punti dai quali si origineranno le *Regioni Di Voronoi*. L'insieme dei punti che verrà passato al blocchetto *Voronoi Fracture* è stato originato dall'unione di due operazioni di *Scatter*, una eseguita su tutta la superficie in maniera non uniforme utilizzando un numero basso di punti e l'altra eseguita con un numero più alto di punti distribuiti sempre in modo non regolare solo su un'area della superfice, come si può notare dalla foto successiva.



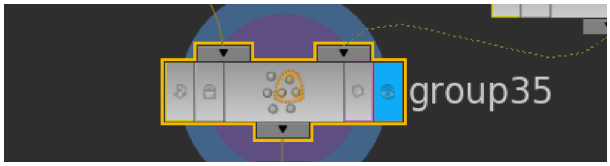
Nello specifico, per quest'ultima selezione più concentrata di punti, sono stati utilizzati: una sfera, inserita come primitiva e posizionata nella zona centrale del piano in una posizione leggermente decentrata, uno *Scatter*, eseguito su tutta la superficie in modo irregolare utilizzando un numero di punti molto elevato in modo tale da generare molti frammenti di piccole dimensioni, un nodo di *Group* che riceve la sfera, lo *Scatter* come input che genera un output che rappresenta il raggruppamento di un sottoinsieme dei punti generati dallo *Scatter*, e un *Delete* che elimina tutti quei punti che, generati dallo *Scatter*, non si trovano all'interno del gruppo creato dall'operatore di *Group*.



Nelle impostazioni dei due *Scatter*, per poter ottenere una diversa fratturazione per ciascun piano ad ogni iterazione, è stata aggiunta una linea di codice all'interno del parametro *Global Seed*, che viene utilizzato per determinare il numero di punti da generare su ciascuna primitiva in modo randomico:

$$stamp("..",\text{chs}(../\text{forstamp}) + "1", 0) * 3$$

Questa linea di codice prende il valore dell'attributo di *Class* corrente e lo moltiplica per tre e in questo modo permette di generare dei punti distribuendoli in maniera sempre diversa ad ogni iterazione. Ed è proprio attraverso quest'impostazione che si sono riuscite a creare cinque differenti fratture, che partono da un settaggio simile, per i vari piani in ingresso al nodo di *For Each*.

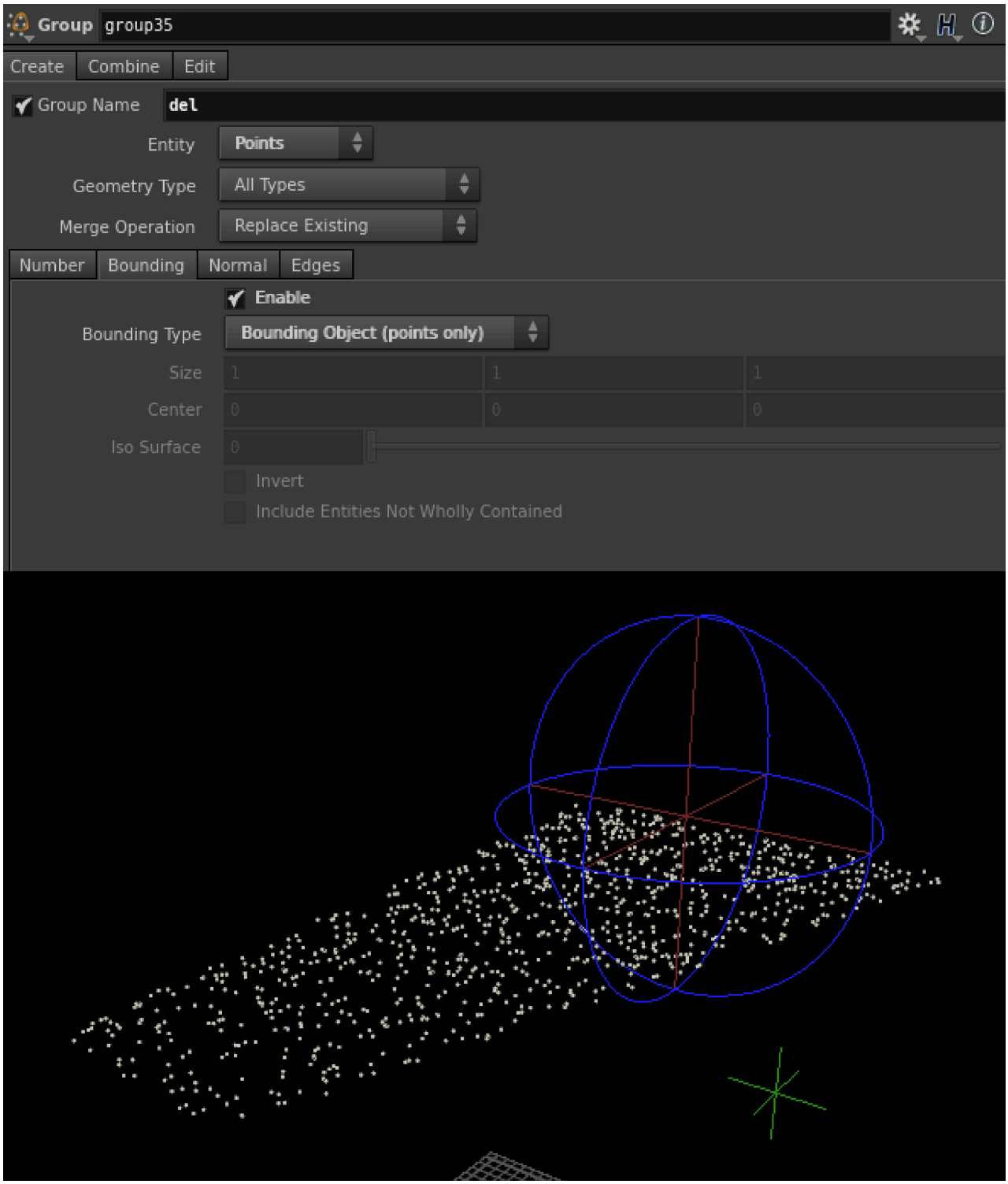


Il nodo di *Group* è anch'esso un nodo di geometria che genera gruppi di punti, primitive o vertici a seconda di vari criteri. Gli elementi possono essere incorporati in più di un gruppo. Questo operatore è molto potente ed è ideale per combinare geometria simile. In generale questo riceve due input, la geometria sulla quale creare dei gruppi e la geometria limitante che funziona solamente, come in questo caso, per gruppi di punti, quando si seleziona la modalità *Bounding Geometry*. I parametri che sono stati utilizzati, nella sezione generale, per questo operatore, sono i seguenti:

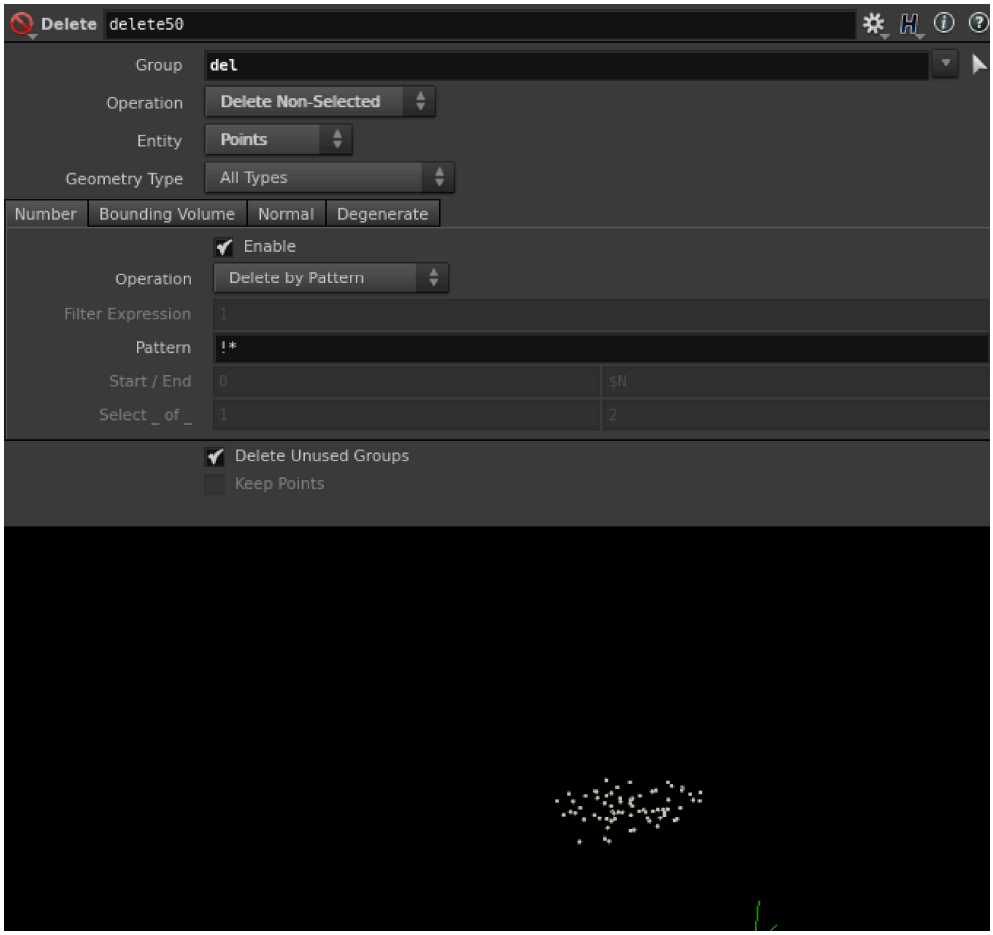
1- *Group Name*. È il nome del gruppo creato.
2- *Entity*. È utile sia per creare un gruppo di primitive o di punti.
3- *Geometry Type*. Indica che tipologia di primitive sono permesse all'interno del gruppo.
4- *Merge Operation*. Indica come risolvere le collisioni fra i nuovi gruppi creati con un gruppo già esistente.
5- *Replace Existing*. I nuovi gruppi creati sostituiranno i gruppi già esistenti.

Nella sezione *Bounding*, invece, sempre del nodo di *Group*, sono stati settati i parametri:

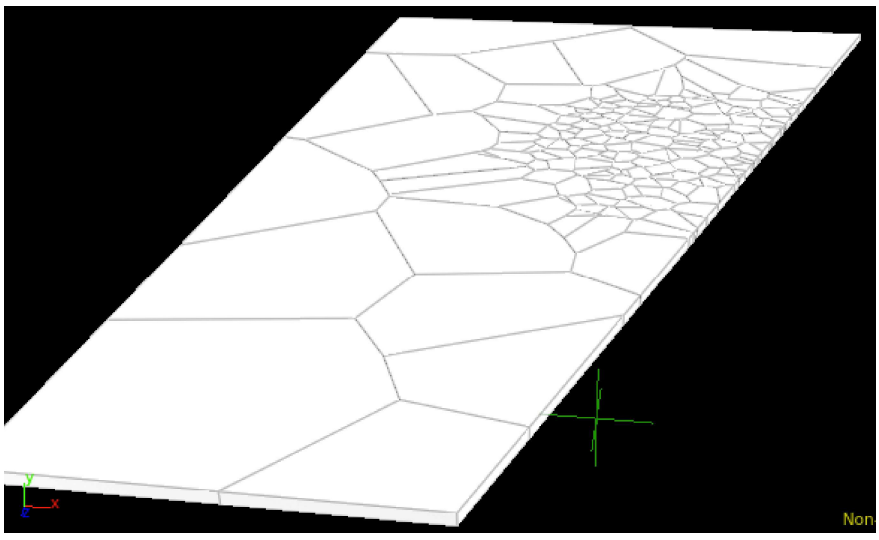
1- *Enable*. Che abilita il raggruppamento tramite *Bouding Volume* ovvero i volumi limitanti.
2- *Bounding Type*. Che indica la forma del *Bounding Volume*.



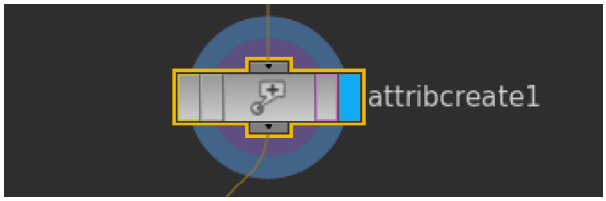
Grazie al nodo di *Group*, quindi, si sono andati a selezionare tutti i punti che si trovano all'interno della sfera, per poi, tramite il nodo di *Delete*, eliminare quelli che non si trovano all'interno del gruppo appena creato. Questa operazione si è potuta eseguire grazie al nominativo del gruppo. Come si può notare nell'immagine sottostante, l'operatore di *Delete* va ad eliminare, tramite il parametro *Operation*, tutti i punti che stanno all'esterno del gruppo *del*, selezionato tramite l'oggetto sfera utilizzato come *Bounding Object*.



Il risultato della prima iterazione, dopo l'operazione di fratturazione eseguita dal nodo di *Voronoi Fracture* su un singolo piano, è il seguente:



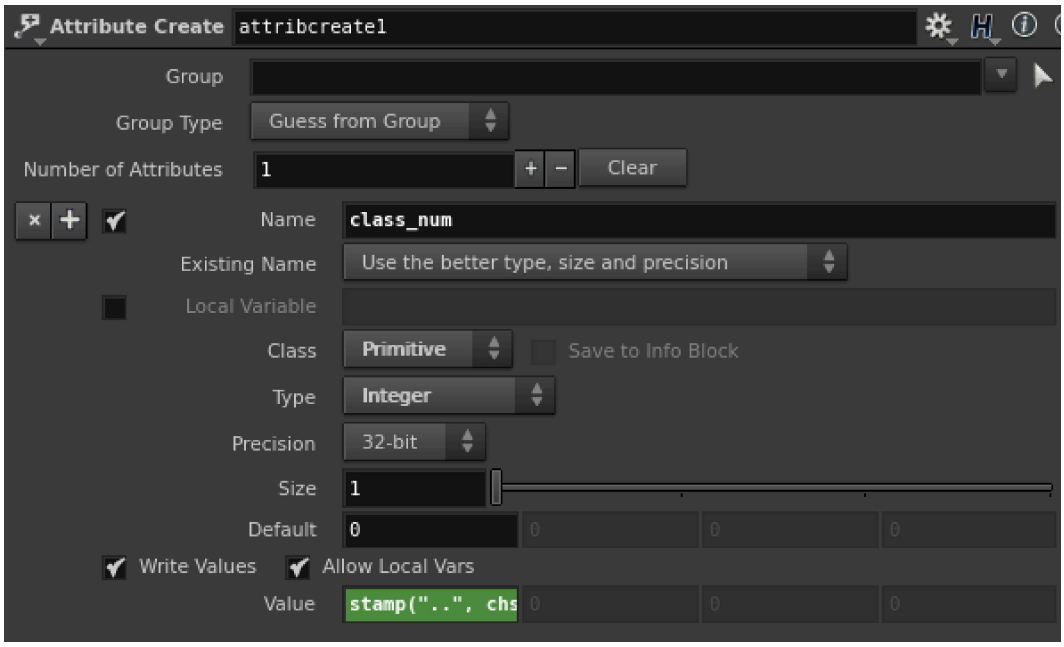
All'interno del *For Each*, dopo l'operatore di *Voronoi Fracture*, sono stati aggiunti due ulteriori nodi, che come si vedrà in seguito saranno utili per poter effettuare delle operazioni successive. Il primo nodo che è stato posto di seguito al *Voronoi Fracture* è quello di *Attribute Create*.



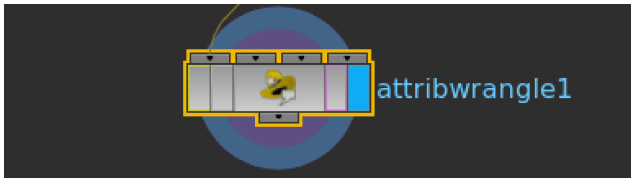
Questo è un nodo di geometria, che aggiunge o modifica degli attributi che vengono definiti dall'utente. L'attributo che verrà creato da questo operatore potrà essere di tipo *float*, *integer*, *vector* o *string*. I parametri rilevanti per questo nodo sono i seguenti:

1- *Number Of Attributes*. Indica il numero di attributi da aggiungere.
2- *Name*. È il parametro che porta il nome dell'attributo.
3- *Existing Name*. È il parametro che specifica quale azione dovrebbe essere intrapresa, se è già presente un attributo con lo stesso nome.
4- *Class*. È il parametro che indica dove si vuole aggiungere il nuovo attributo alla geometria.
5- *Type*. È il parametro che indica il tipo di attributo, ovvero se un *float*, *integer* o *string*.
6- *Precision*. Indica la precisione numerica che si desidera utilizzare, quando il parametro *Type* non è una stringa.
7- *Size*. Indica il numero di elementi presenti nell'attributo.
8- *Default*. È il parametro che riporta il valore dell'attributo di default.
9- *Value*. È il valore numerico da scrivere per questo attributo.

Nello specifico è stato creato un attributo di primitiva chiamato *class_num* che riporta come parametro *value* il valore dell'attributo *class* per l'iterazione corrente.



Dopo questa operazione ne è stata effettuata una simile utilizzando l'operatore *Attribute Wrangle*.



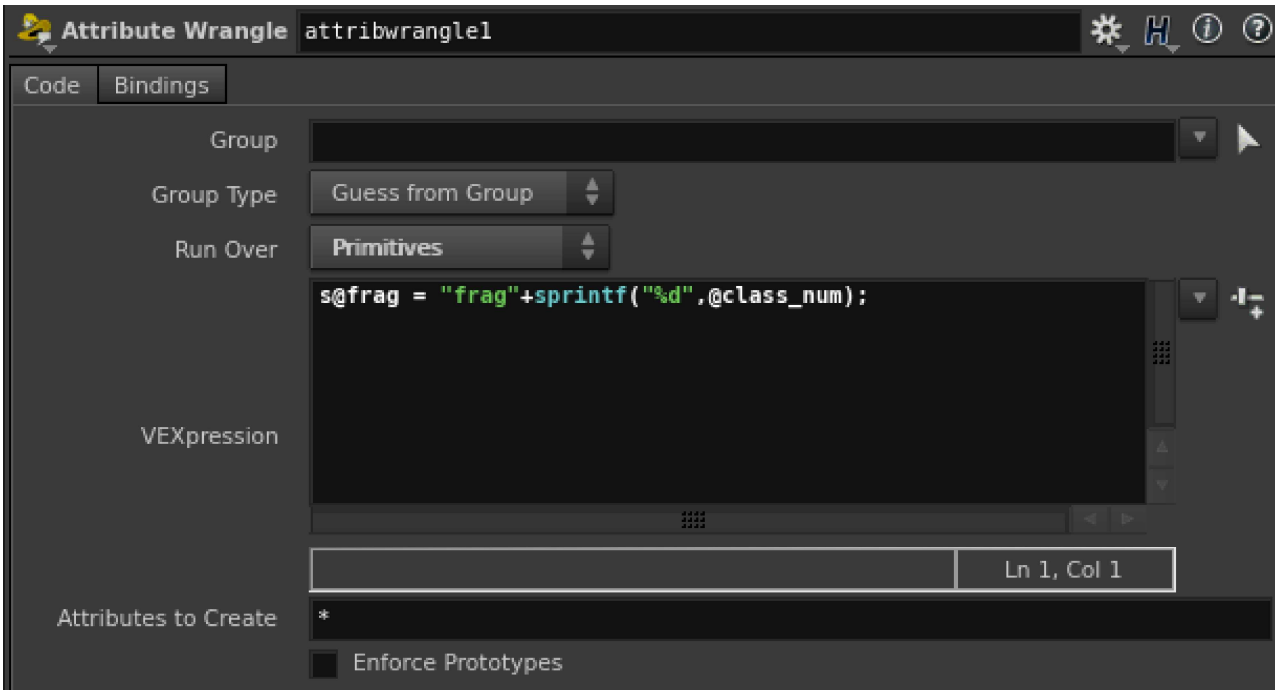
Questo è uno nodo di geometria, che esegue un frammento di codice *VEX* per modificare i valori degli attributi. Questo è un nodo molto potente, di basso livello, che consente di lavorare sugli attributi utilizzando frammenti di codice *VEX*. Esso esegue il frammento di codice sui *detail* o per ogni attributo di *point/primitive/vertex* nella geometria di ingresso. Il frammento di codice può modificare la geometria di ingresso cambiando gli attributi. Le impostazioni rilevanti per questo nodo sono le seguenti:

1- *Run Over*. Specifica quale classe per l'attributo di geometria verrà modificata.
2- *VEXpression*. Specifica il frammento di codice che si desidera utilizzare per eseguire una determinata operazione.

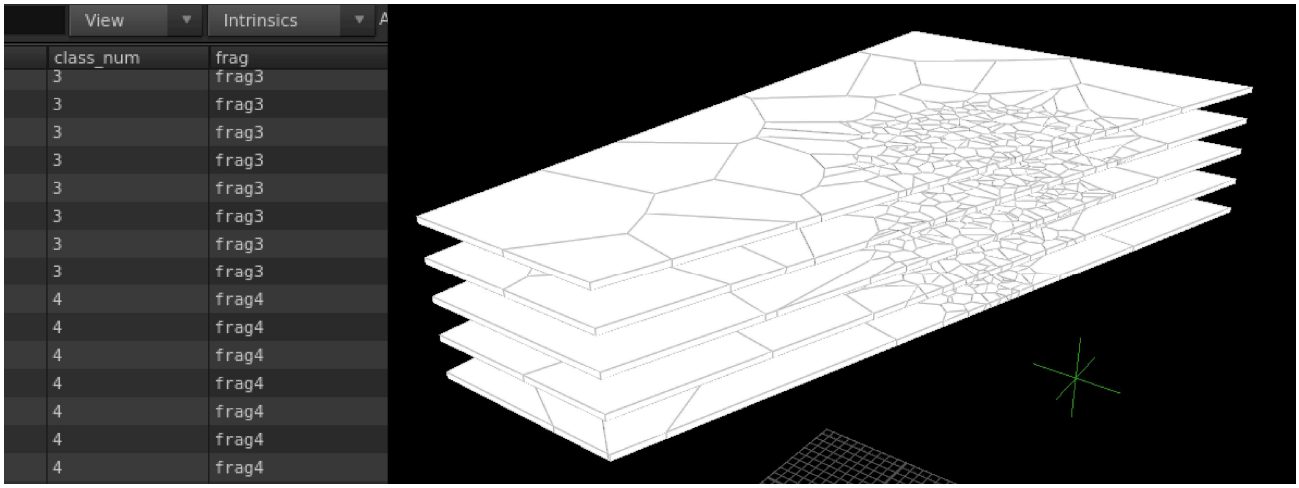
Come si può vedere nell'immagine sottostante, questo operatore è stato utilizzato per creare un ulteriore attributo da inserire sulle primitive. Il codice *VEX* scritto in questo nodo riporta la seguente espressione:

$$s@frag = \text{"frag"} + sprintf(\text{"\%d"}, @class\_num);$$

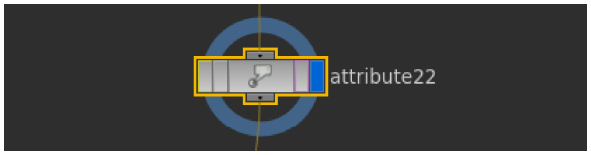
Il significato di questa riga di codice è il seguente: *s@frag*, indica che verrà creata una variabile di tipo stringa chiamata *frag*, mentre *"frag" + sprintf("%d", @class_num)*, indica che il valore di questo nuovo attributo riporterà la parola *frag* seguita da un valore numerico intero pari al valore dell'attributo *class_num* dell'iterazione corrente.



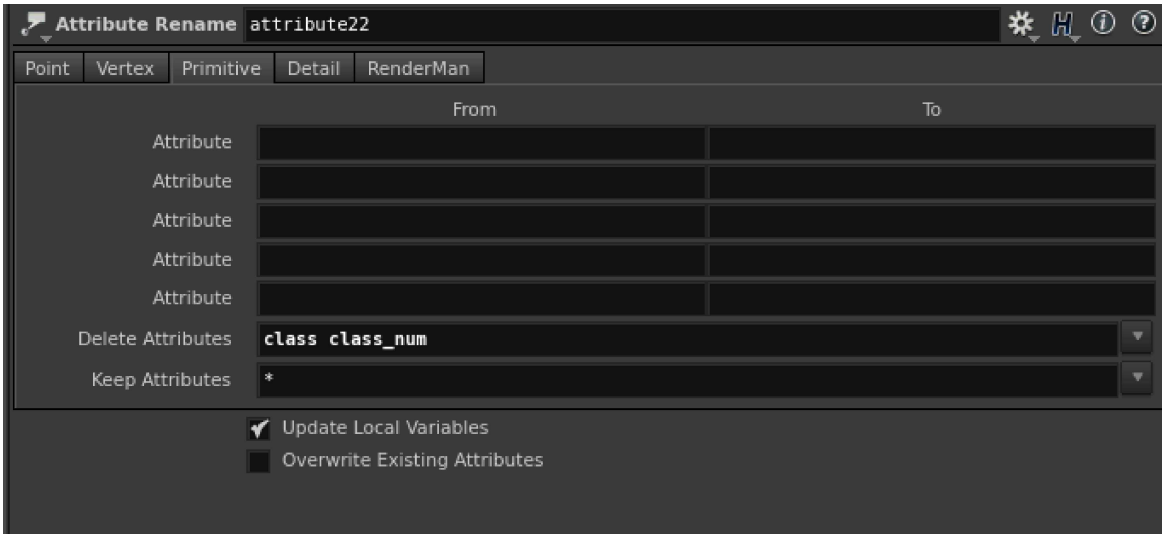
Tornando più in superfice si può notare, dalla foto successiva, il lavoro di fratturazione effettuato dall'operatore di *For Each*. Un occhio di riguardo va posto sulla foto che riporta le informazioni relative a questo nodo, in quanto saranno utili per comprendere le operazioni successive. L'immagine sottostante a destra, riporta un estratto che mostra chiaramente la creazione dei frammenti legata a ciascun piano, le primitive che appartengono alla terza iterazione, che riguardano lo stesso piano, hanno un attributo *frag* che è lo stesso per tutte, mentre, questo valore cambia all'interazione successiva che riguarda la frattura del quarto piano considerato nell'iterazione successiva.



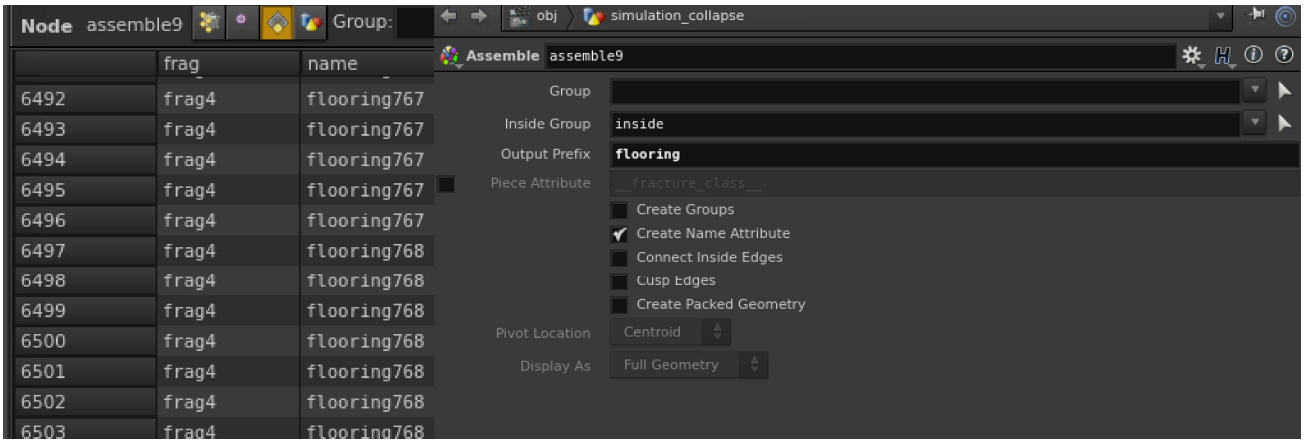
Il nodo che viene subito dopo quello di *For Each* è l'*Attribute Rename*.



Anche questo è un nodo di geometria, il cui compito è quello di rinominare o eliminare attributi di punto o di primitive, difatti è stato utilizzato per eseguire proprio quest'ultima operazione. Sono stati eliminati gli attributi di primitiva *class* e *class_num* che erano stati utilizzati per creare gli attributi di primitiva *frag* all'interno del nodo di *For Each*.

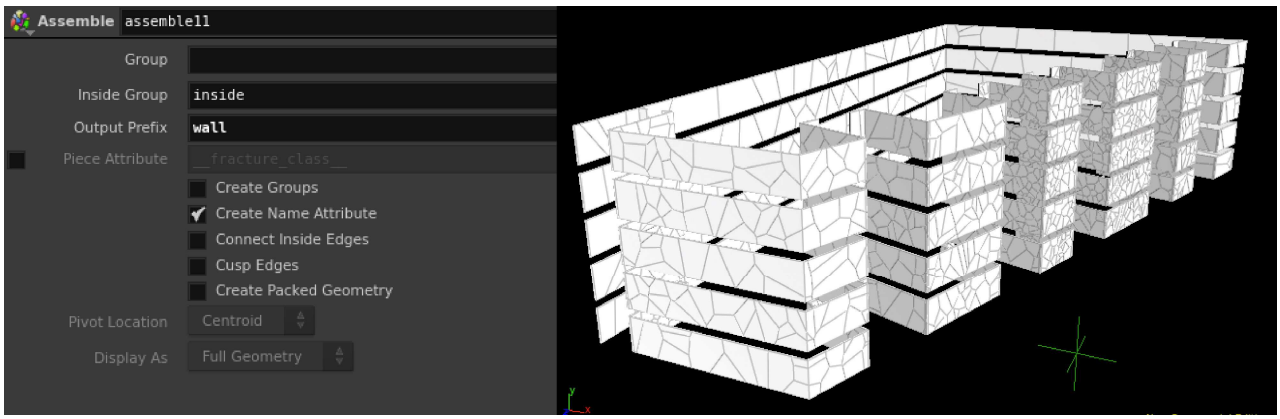


A questo punto è stato creato un altro attributo sulle primitive tramite il nodo di *Assemble* le cui proprietà sono già state spiegate in precedenza. In questo caso il nodo non viene utilizzato per impacchettare la geometria ma solo per creare un attributo che sarà poi utile insieme all'attributo *frag* per la fase di creazione dei vincoli.

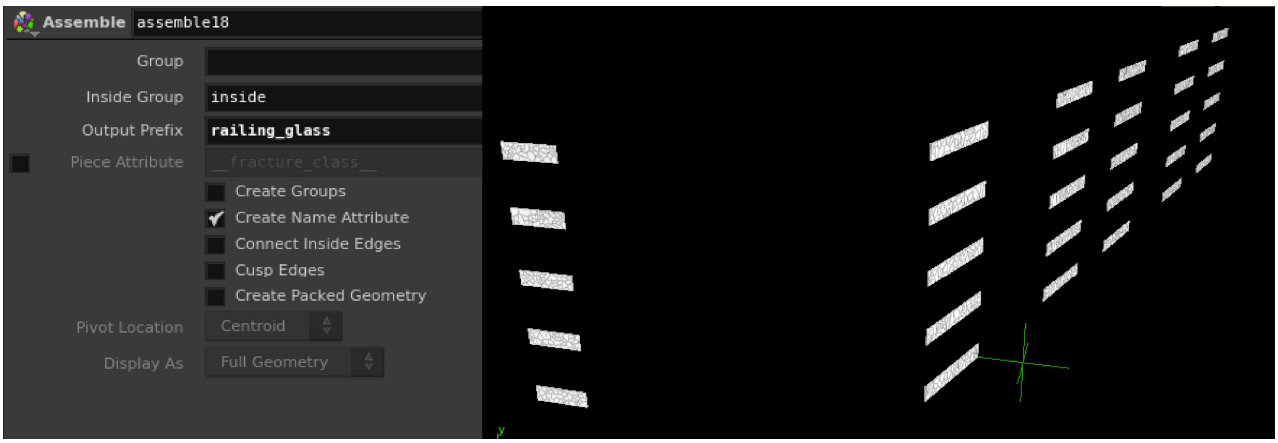


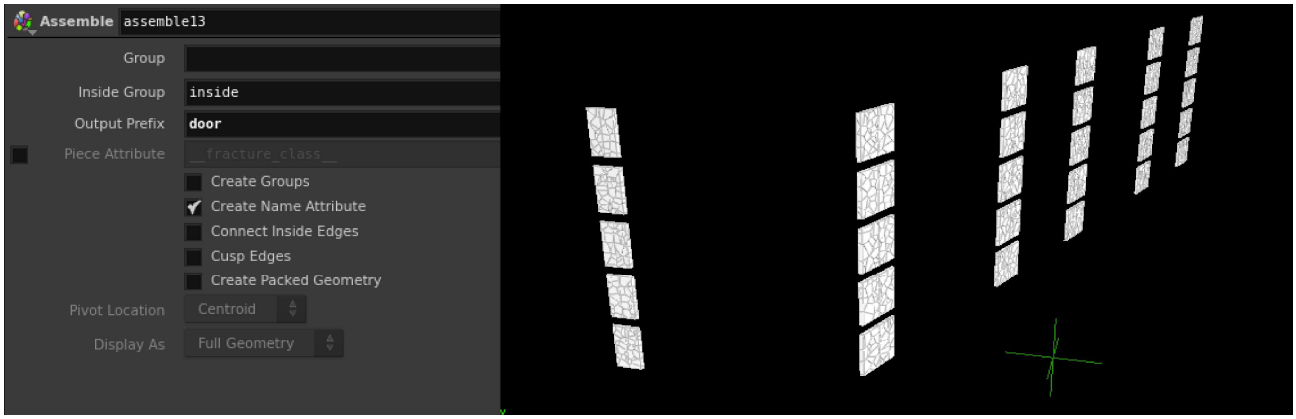
Come si può notare dallo scatto qui sopra è stato aggiunto l'attributo *flooring* su ciascuna primitiva costituente i piani fratturati dal precedente nodo di *For Each*. Da questo punto in poi si procede con la fase di creazione dei vincoli, ovvero si decide come tutti questi pezzi fratturati devono essere connessi fra loro per poter costruire la dinamica da conferire alla simulazione di *Destruction*. Prima di procedere con la spiegazione della parte che riguarda la creazione dei vicoli è utile osservare anche le procedure di fratturazione che riguardano gli altri elementi che compongono il palazzo, anche se sono tutte piuttosto simili fra loro.

Prendendo in riferimento questo blocco centrale, la cui procedura di fratturazione dei piani è stata appena spiegata, viene analizzato velocemente com'è stato diviso e come sono state fratturate tutte le altre parti che lo compongono ovvero i muri, le ringhiere in vetro, le ringhiere in ferro e le porte. Nell'immagine sottostante si può vedere la prefratturazione eseguita sui muri del blocco preso in considerazione. Qui, l'unica differenza è stata l'aggiunta di una sfera prima di entrare all'ingresso del nodo di *For Each* che è stata utilizzata per rompere in modo ancora più fine la geometria che si trova racchiusa al suo interno e posizionata all'incirca nella parte centrale visibile frontalmente nello scatto seguente. La procedura di fratturazione, a parte questa piccola, differenza è sostanzialmente identica a quella utilizzata per i piani. Il nodo di *Assemble* qui crea un attributo di primitive chiamato *wall*.

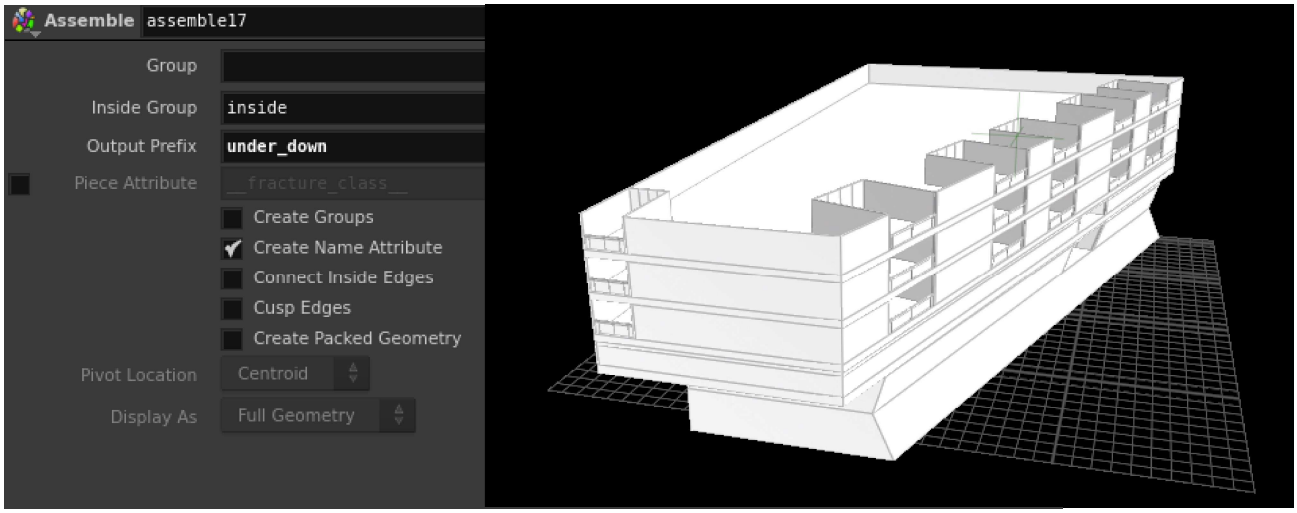


Per le porte e le ringhiere del blocco medesimo è stata utilizzata la stessa metodologia, per queste parti il nodo di *For Each* è un po' più semplice perché ha al suo interno solo uno *Scatter* prima di entrare in ingresso all'operatore di *Voronoi*. Anche in questo caso prima di effettuare la creazione dei vincoli è stato aggiunto un nodo di *Assemble* che tramite la creazione dell'attributo di primitive permette di distinguere le varie parti, ovvero le ringhiere in vetro, le ringhiere in ferro e le porte.

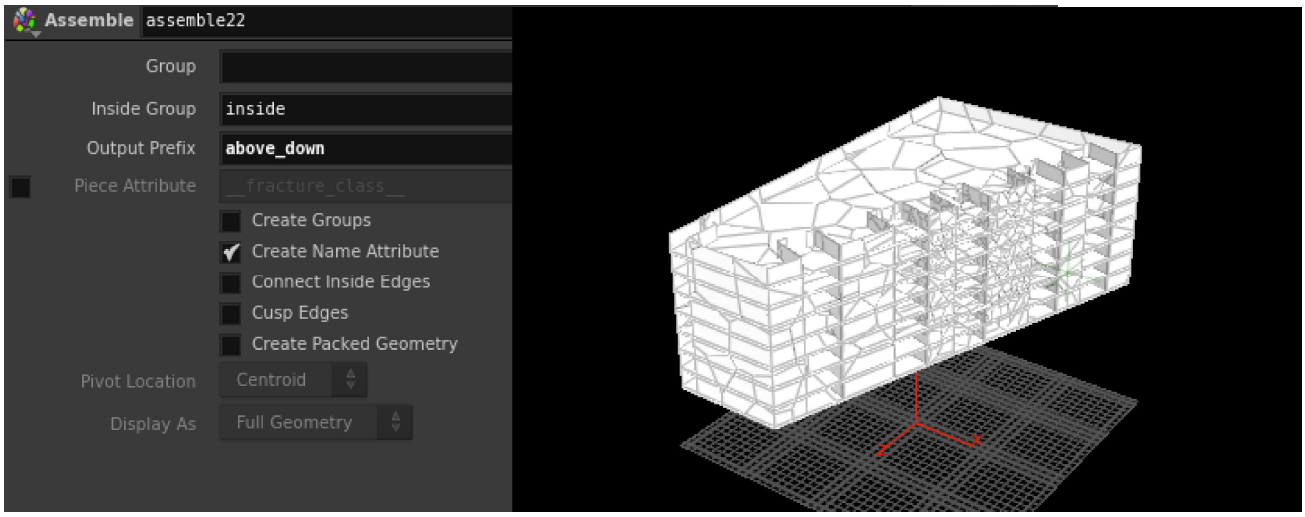




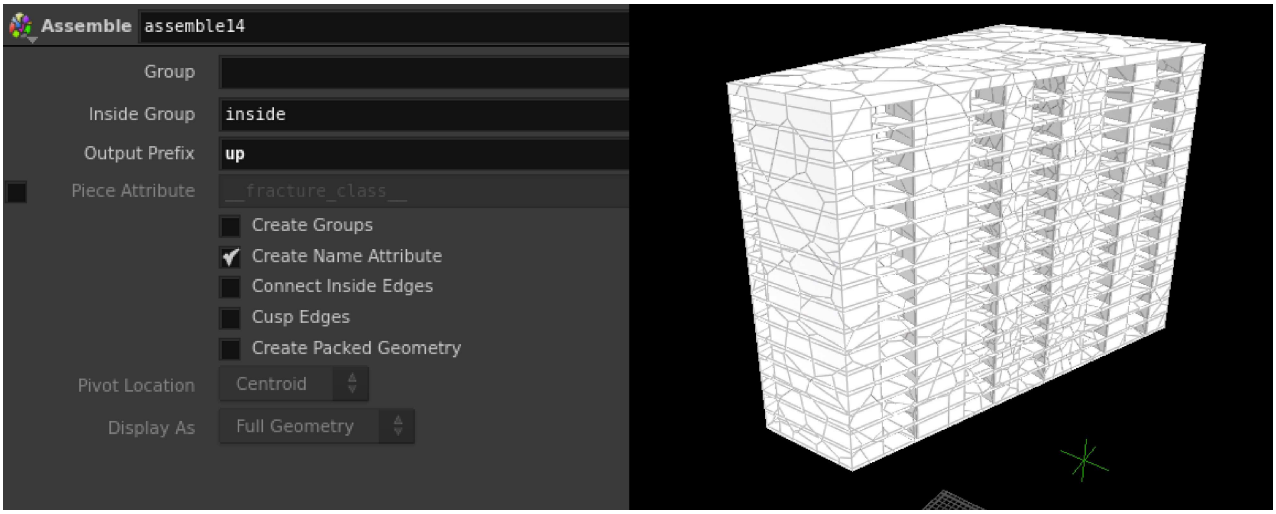
Si proceda ora con il dettaglio degli elementi costituenti il palazzo partendo dal base arrivando fino alla sua cima. La foto successiva mostra la base del palazzo, la quale, per evitare di complicare da un punto di vista computazionale la simulazione, è rimasta intatta, ovvero non si è eseguita alcun tipo di rottura della geometria. Questo anche perché, essendo inquadrata molto di sfuggita nel *Rendering* finale, non si sarebbe vista e quindi sarebbe stato inutile complicare in questo modo la simulazione.



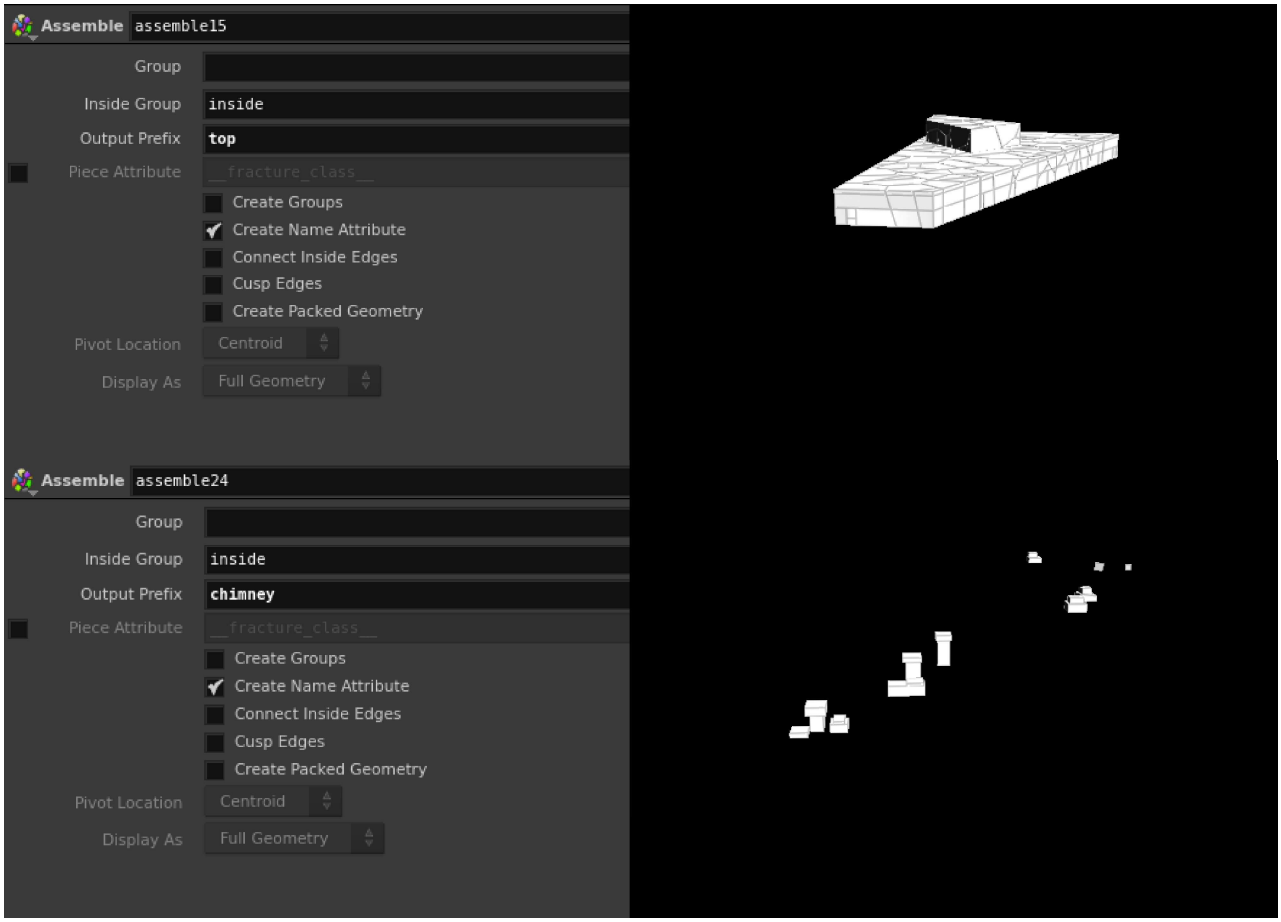
Salendo più sopra si trova un altro blocco di piani e di muri, il quale è stato preso, privato di porte e ringhiere, le quali sono state fratturate con il metodo spiegato in precedenza, con l'unica differenza che in questo caso il nodo di *Assemble*, dopo la prefratturazione, attribuisce un identificativo di primitive differente anche se si tratta sempre degli stessi oggetti, e successivamente integralmente sottoposto a una frattura unica.



La frattura della parte del palazzo visibile nella foto sottostante è stata eseguita senza l'utilizzo del *For Each*, in quanto la fratturazione si sarebbe dovuta eseguire sull'intero blocco, come nel caso precedente. Anche qui è stata eseguita la frattura delle porte e delle ringhiere separatamente sempre utilizzando la metodologia già spiegata per i blocchi precedenti in cui questi elementi erano presenti.



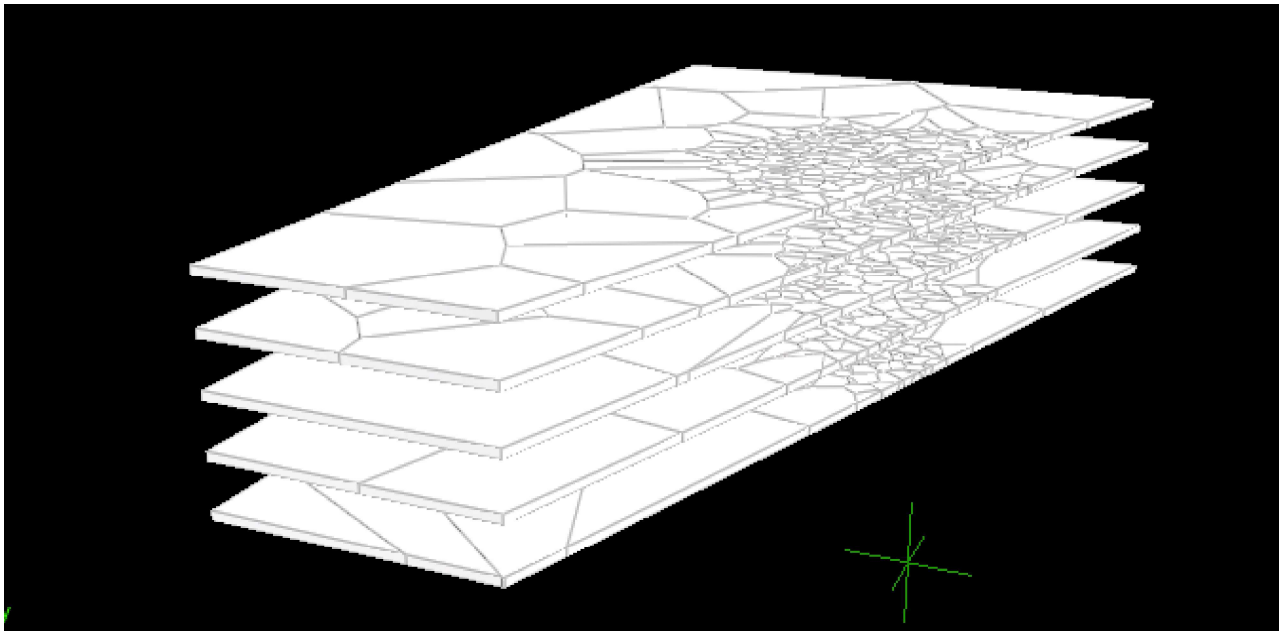
Anche per il tetto è stata utilizzata una procedura di fratturazione che ha preso in considerazione l'intero blocco privato dei comignoli, i quali essendo molto piccoli e non così visibili all'interno del *Rendering* finale non sono stati sottoposti ad alcuna procedura prefratturazione.



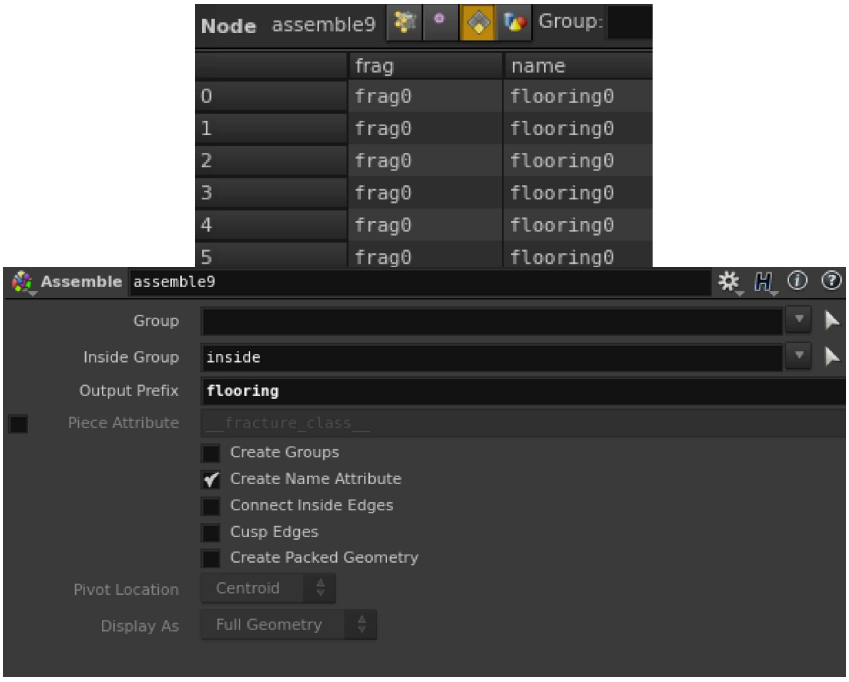
Come si può notare, in tutti questi scatti, a tutte le parti costituenti il palazzo, è stato assegnato un attributo di primitive diverso, un'operazione utile quando si ha a che fare con molti elementi i quali devono essere sottoposti a fasi di elaborazioni differenti. In questo caso tali attributi sono stati utili anche per la fase successiva di creazione dei vincoli. L'operazione di fratturazione è un'operazione complessa ma non così difficile da realizzare perché il programma offre tutti gli aiuti del caso per poterla eseguire al meglio. In questa fase le problematiche non sono state molte, è stato necessario però un lavoro di analisi delle varie procedure per poter scegliere quelle migliori per il progetto. La teoria e poi la pratica per questa fase sono durate circa un mese di lavoro complessivo.

Vincoli

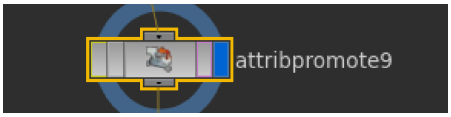
A questo punto inizia una fase molto importante perché sarà in questa che verrà decisa in qualche modo la dinamica da conferire alla simulazione, in quanto i vincoli sono il collante che mantiene insieme i pezzi fratturati della fase precedente. In realtà il tipo di collante verrà poi scelto all'interno della simulazione, qui vengono solamente preparate le reti di connessioni che poi dovranno essere attivate all'interno, appunto, del nodo di simulazione. Anche per questa tipologia di lavoro il software *Houdini* mette a disposizione un vasto set di operatori e garantisce quasi infinite possibilità di implementazione. La procedura più automatica si ottiene attraverso l'utilizzo di un nodo in particolare, di cui, proprio per le sue caratteristiche, è stato fatto largo utilizzo all'interno di questo progetto: il *Connect Adjacent Pieces*. Nel lavoro di *Destruction* che affianca questo documento, in particolare, sono state utilizzate due procedure differenti, entrambe aventi come pezzo centrale questo operatore, per la creazione delle connessioni tra i pezzi, che tra l'altro si sono rivelate quelle più adatte al tipo di risultato che si voleva raggiungere. La parte di progettazione delle reti di vincoli, quindi la procedura di montaggio dei pezzi, è durata circa due mesi, perché lo studio è stato piuttosto articolato; e l'applicazione della teoria ha incontrato parecchie difficoltà. Il problema più grande ha coinvolto la giusta creazione degli attributi da conferire a questi legami, che in un primo momento era sbagliata e non permetteva la creazione di queste relazioni tramite il *Connect Adjacent Pieces*. Una volta corretti questi attributi è stata poi facile la prosecuzione del lavoro. Di seguito vengono spiegate le procedure utilizzate all'interno del progetto, per affrontare il lavoro svolto in questa fase, che si prestano a svolgere il compito per cui sono state create rispondendo in maniera abbastanza buona a quelli che erano gli obiettivi per la costruzione della dinamica di simulazione. La fase di creazione dei vincoli inizia subito dopo il nodo di *Assemble* spiegato alla fine della fase di fratturazione precedente. Prendendo in considerazione la fratturazione dei piani centrali, spiegata nel dettaglio nel sottocapitolo precedente, si prosegue ora con la spiegazione della creazione delle reti di vincoli.



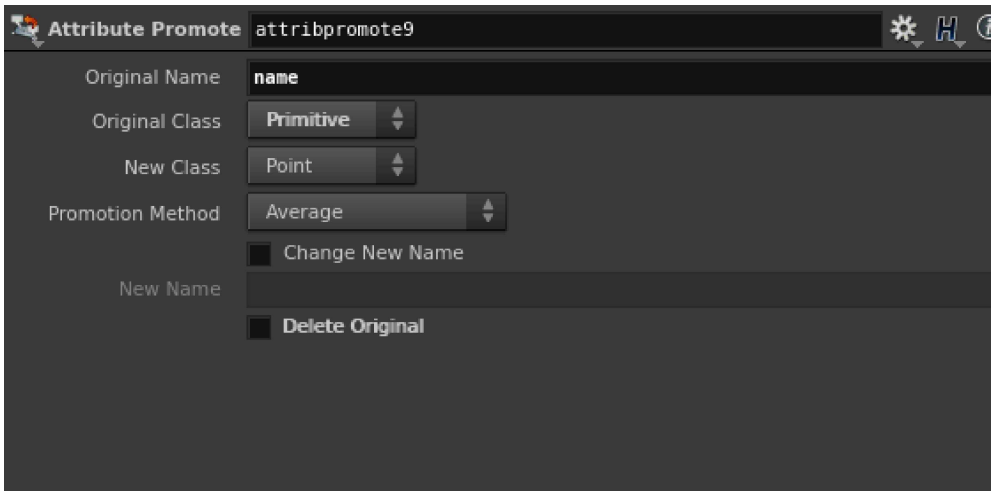
Dopo ciascun nodo di *Assemble* sono stati creati due tipi di vincoli, uno per l'uscita della geometria proveniente da questo nodo e l'altro che preleva le geometrie provenienti da differenti nodi di *Assemble* e le collega insieme. A questo punto del lavoro ci si trova una geometria che ancora non è stata impacchettata, dotata di due attributi di primitiva: *frag* e *name*.



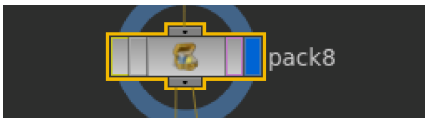
Dopo il nodo di *Assemble*, è stato utilizzato l'operatore *Attribute Promote* per spostare l'attributo *name* dalle primitive ai punti.



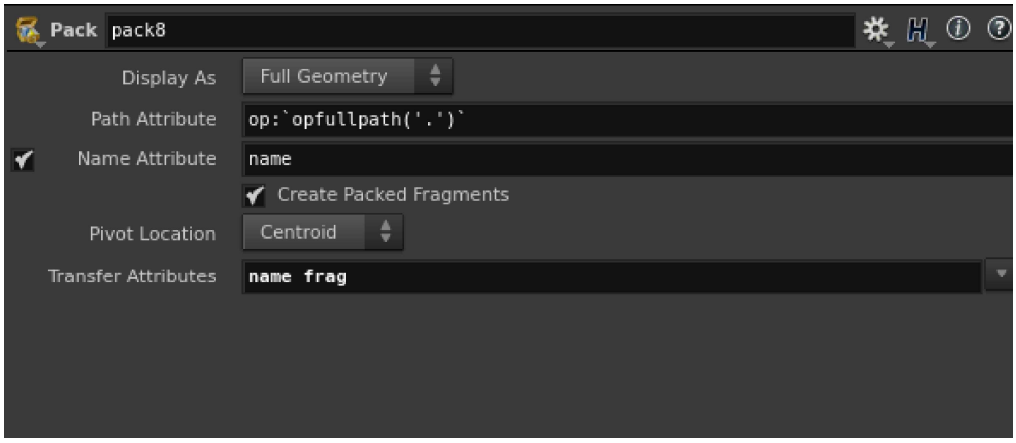
L'*Atribute Promote* è un nodo di geometria che promuove o declassa attributi da un livello di geometria a un altro. Questo fornisce un modo semplice per convertire un attributo da una classe all'altra.



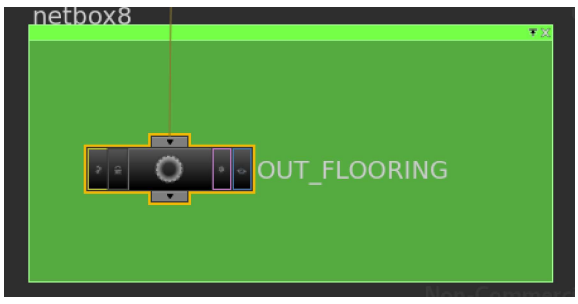
Tramite l'operatore di *Pack* è stata poi eseguita la necessaria operazione di impacchettamento della geometria.



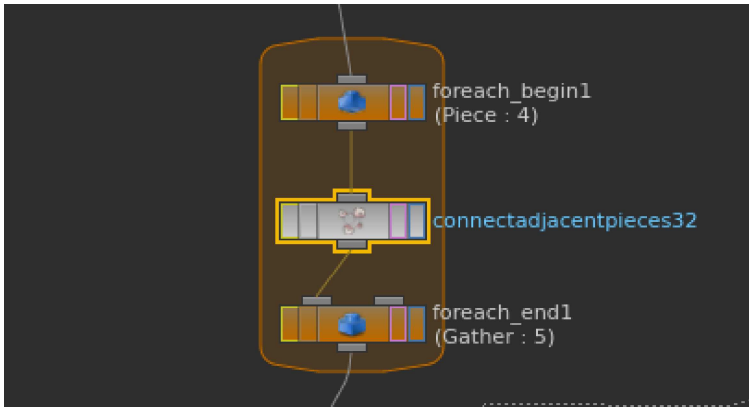
Questo è un nodo di geometria che impacchetta, appunto, la geometria in una primitiva assemblata, ovvero imballa la geometria in un'unica primitiva. Questa operazione permette un notevole risparmio di memoria. Tramite l'utilizzo di questo operatore vengono generate delle *Packed Primitives,* le quali possiedono tutte le informazioni riguardanti la geometria incorporata al loro interno. *Mantra*, il *viewport*, i *solver* e molti altri componenti di *Houdini*, sanno come interpretare le informazioni impacchettate e possono renderarle, mostrarle e soprattutto lavorare con la geometria così trasformata più efficientemente. Le *Packed Primitives* non possono essere modificate, se si desidera farlo prima è necessario estrarre la geometria impacchettata, e poi modificarla, e se lo si richiede impacchettarla nuovamente.
Tramite il parametro *Transfer Attributes* del nodo di *Pack* è stato possibile specificare una lista di attributi da trasferire alla geometria impacchettata, come si può notare dalla foto successiva sono stati trasferiti, in questo caso, gli attributi *name* e *frag*.



A questo punto del grafo è presente una biforcazione, un ramo porta alla creazione di un primo vincolo, il secondo, riportato nello scatto seguente, rappresenta invece l'uscita della geometria appena impacchettata, ovvero l'output che poi verrà utilizzato in ingresso alla simulazione per importare al suo interno la geometria sulla quale lavorare e di conseguenza da simulare.



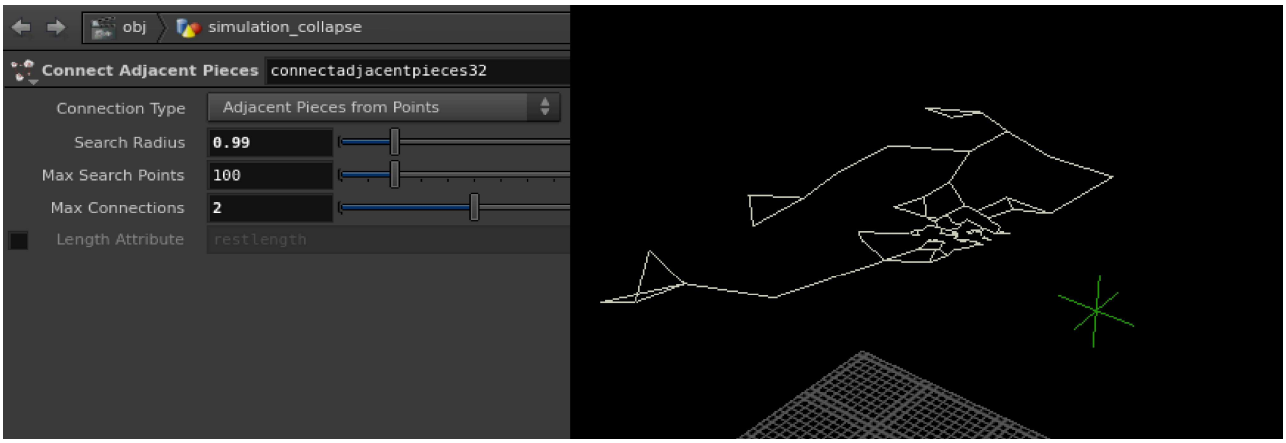
L'altro ramo, ovvero quello che contiene uno dei vincoli, è invece strutturato come segue: è presente, come riportato nello scatto sottostante, un blocco di *Loop* costituito da un inizio e una fine. Il blocco di inizio semplicemente preleva ogni pezzo in base ad ogni iterazione eseguita sull'attributo *frag* specificato nel blocco di fine, quindi in sostanza ripete l'operazione all'interno del blocco per ogni piano preso in considerazione ad ogni iterazione.



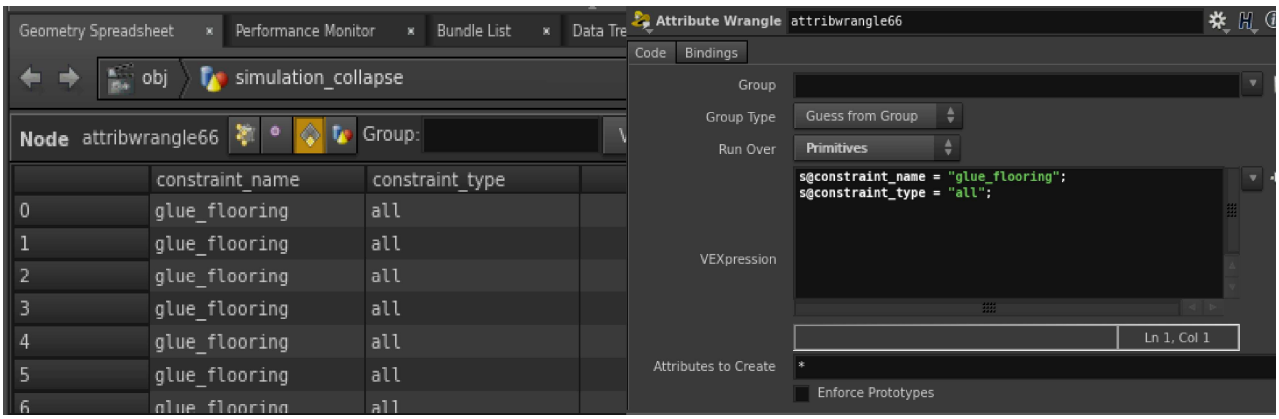
L'operazione svolta all'interno del blocco per ogni piano è quella che consente la creazione della rete di vincoli eseguita dal nodo *Connect Adjacent Pieces*. Questo è un nodo di geometria che crea, appunto, delle linee o delle connessioni tra pezzi adiacenti. Questo operatore crea un insieme di poligoni che connettono insieme i punti provenienti da pezzi vicini. Questo è utile per creare una rete di vincoli che limita e incolla insieme oggetti vicini all'interno di una simulazione. I parametri specificati, per questo operatore, sono stati i seguenti:

1- *Connection Type*. Che specifica come le connessioni tra i pezzi adiacenti saranno create. In questo caso il parametro specificato è *Adjacent Pieces from Surface Points* che permette il collegamento dei centroidi dei pezzi adiacenti.
2- *Search Radius*. È il parametro che specifica la distanza massima consentita durante la ricerca dei punti vicini per creare le connessioni.
3- *Max Search Points*. Determina un limite superiore al numero di punti adiacenti ispezionabili.
4- *Max Connections*. Specifica un limite superiore al numero di pezzi a cui ogni punto può essere connesso

In questo caso è stata creata una rete in cui il massimo delle connessioni tra i punti che si potevano creare sarebbero state due, per permettere una rete più stabile in quanto se si fosse ridotto il *Search Radius* si sarebbe creata una rete troppo fitta di vincoli e nel nostro caso si volevano tenere insieme i pezzi ma senza creare una rete troppo forte perché si sarebbe dovuta rompere facilmente e soprattutto si sarebbe dovuta rompere all'inizio della simulazione.

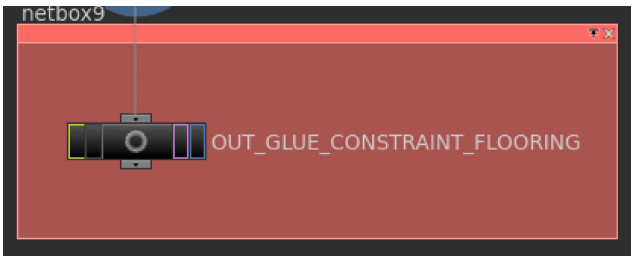


L'operazione successiva è stata quella di ridurre le dimensioni della rete per evitare che fossero inclusi i pezzettini più piccoli situati al centro dei piani per farli crollare proprio all'inizio della simulazione e dare quindi l'impressione che l'inizio della rottura avvenisse a seguito di un eventuale esplosione. In questo modo appunto si voleva dare l'idea che l'esplosione avesse creato una rottura istantanea di questa parte e che poi da qui il resto della struttura del palazzo fosse stata trascinata giù di conseguenza. L'altro nodo importante, in questa fase, prima della creazione dell'output, è l'*Attribute Wrangle*, utilizzato per la creazione degli attributi necessari per l'attivazione di questi vincoli all'interno della simulazione.

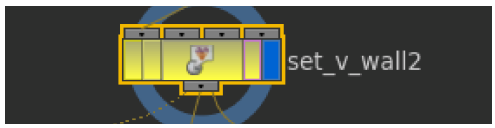


Qui vengono creati gli attributi di primitiva *contraint_name* e *constraint_type*.

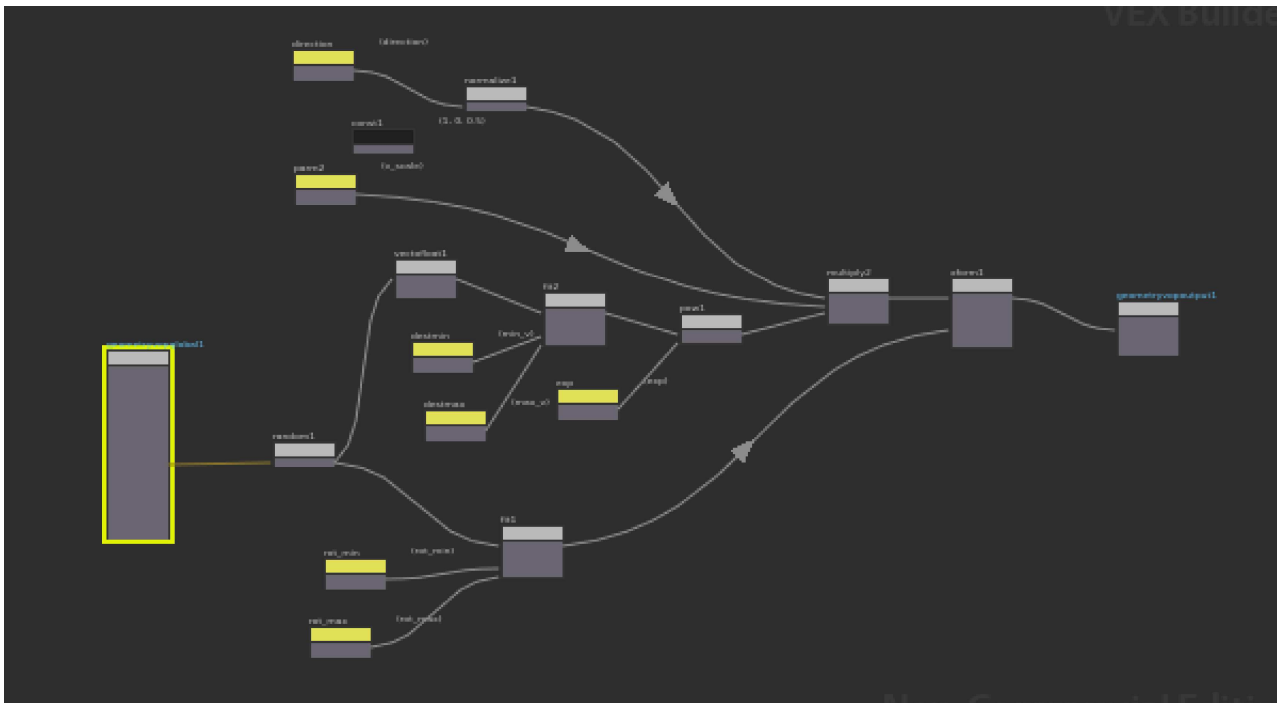
La creazione dell'output riguardante la rete di connessioni appena creata, riportato nell'immagine qui sotto, come vale per quello riferito alla geometria impacchettata, è quello che poi verrà richiamato all'interno del nodo di simulazione.



A questo punto è stata descritta una delle due procedure utilizzate per la creazione dei vincoli. Prima di procedere con la spiegazione della seconda, si osservi velocemente l'applicazione di questa prima metodologia sugli altri blocchi dell'edificio. Non ci sono delle grosse differenze, ma si può notare che per alcuni elementi sono stati utilizzati degli operatori aggiuntivi, come ad esempio quello che permette l'iniezione di una velocità iniziale, utilizzato per dare l'illusione che l'esplosione, presente nei primi frame, sia in grado di generare una forza interna tale da poter scagliare pezzi di muro lontano, come sarebbe potuto accadere nella realtà. Per quanto riguarda il blocco dei muri, riferito a quello dei piani centrali, la cui creazione dei vincoli è stata appena trattata, è presente una struttura che, come già detto, possiede grossomodo le stesse fattezze. Infatti, subito dopo il nodo di *Assemble*, come prima, anche qui è presente il nodo di *Attribute Promote* e poi quello di *Pack*. Inoltre è sempre presente un ramo corrispondente all'output della geometria impacchettata, sottoposto poi a ulteriori lavorazioni, in particolare, come già accennato, è stato aggiunto, a specifiche aree dell'oggetto muro, un attributo che conferisce la possibilità di aggiungere una velocità. Tramite il nodo di *Group*, quindi, sono state selezionate delle parti di muro e poi tramite il nodo *Attribute VOP* è stata generata la velocità con i relativi controlli per poterla regolare ed aggiustare prima di entrare nella simulazione.

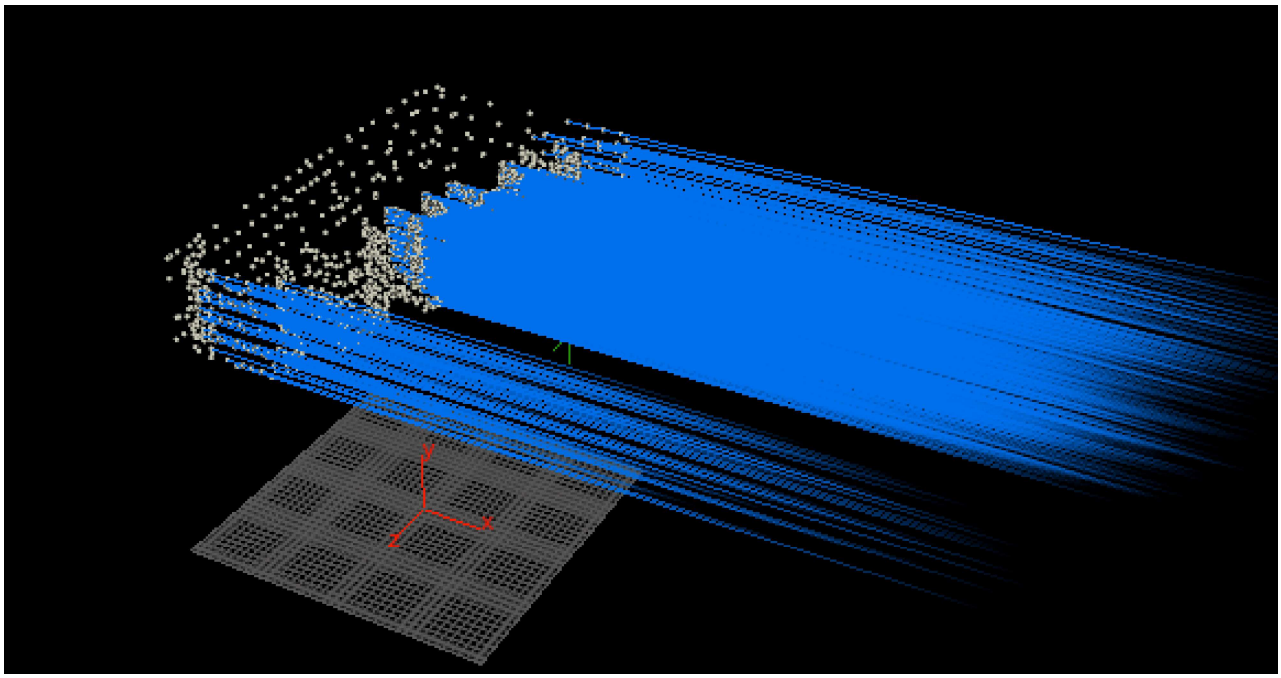


L'*Attribute VOP* è un nodo di geometria che esegue una rete di *VOP* per modificare gli attributi di geometria. La rete di nodi di *VOP* è specificata al suo interno.

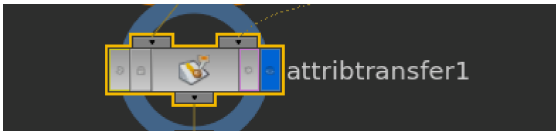


Sostanzialmente questo network ha permesso di generare dei parametri per il controllo della velocità assegnata a ogni primitiva: è presente un parametro che ne regola l'ampiezza e uno ad esempio che ne regola la direzione. In sostanza questi sono stati i parametri maggiormente utilizzati per attribuire la giusta direzione ai pezzi durante la simulazione.

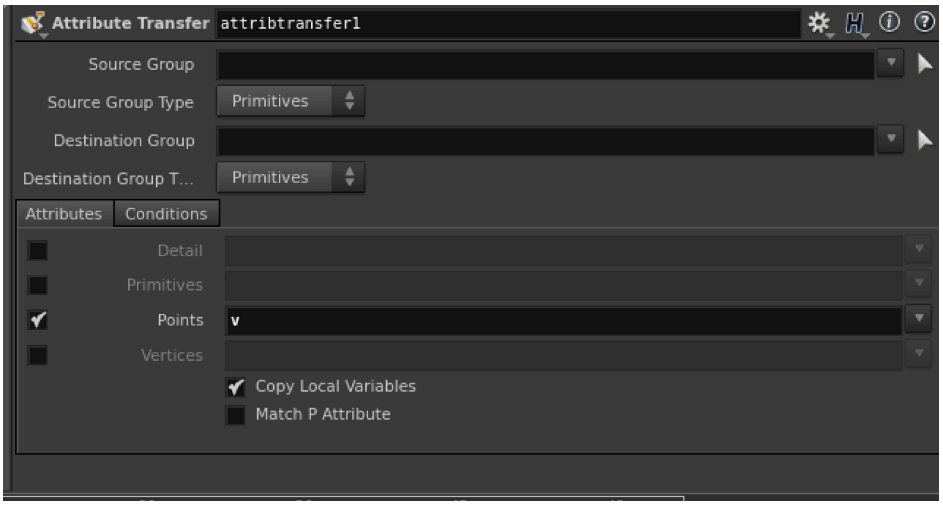
In questo caso, per ogni gruppo racchiuso nel nodo di *Group,* è possibile di decidere la rotazione di ciascuna velocità associata a ogni primitiva, cambiando i parametri *Rotation Min* e *Rotation Max*. La *Direction* permette di dare una direzione al raggio. *V Scale* decide l'ampiezza, quindi la lunghezza del raggio blu, come mostrato nella foto, che rappresenta la velocità. *Min Vel Percent* e *Max V Percent* rappresentano la possibilità di cambiare, in modo randomico, la potenza della velocità associata alle varie primitive, in questo caso si è data la possibilità di variare minimamente questo valore in modo tale da conferire un aspetto più casuale alla simulazione e quindi in modo tale che ogni pezzo avesse una velocità diversa per donare, appunto, un aspetto più realistico al crollo.



Lo scatto qui sopra è utile solamente per comprendere come sia stato possibile regolare i raggi associati alla velocità graficamente, in quanto visibili nella *Viewport*.
Nell'altro ramo, subito dopo la creazione delle connessioni, tramite sempre un loop che ad ogni muro riferito a un piano diverso crea i collegamenti fra i pezzi attraverso il nodo *Connect Adjacent Pieces,* è stato aggiunto un nodo di *Attribute Transfer*.

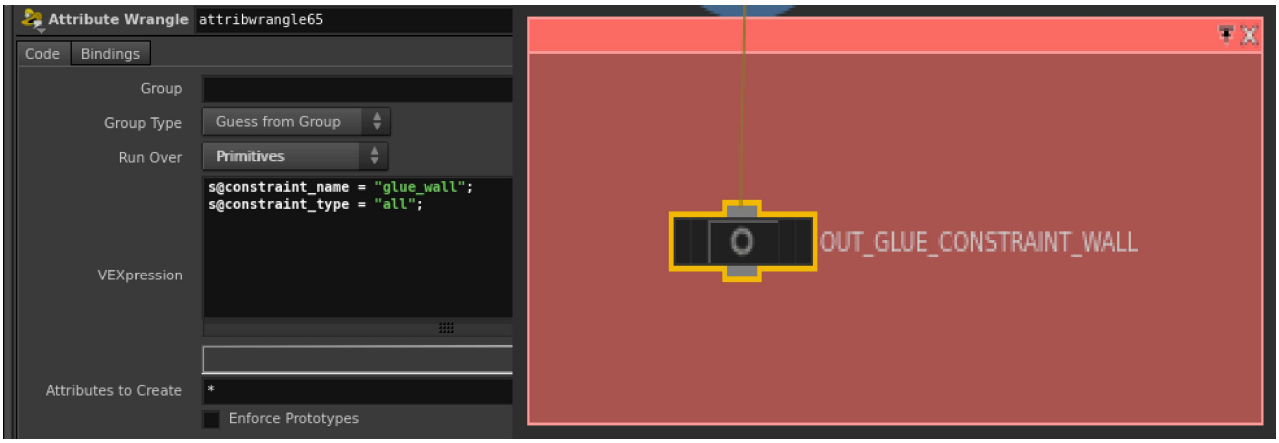


Questo è un nodo di geometria che trasferisce attributi di vertici, punti, primitive e detail tra due modelli. In questo caso, detto operatore, serve per trasferire gli attributi di velocità che si sono creati nel passaggio precedente.

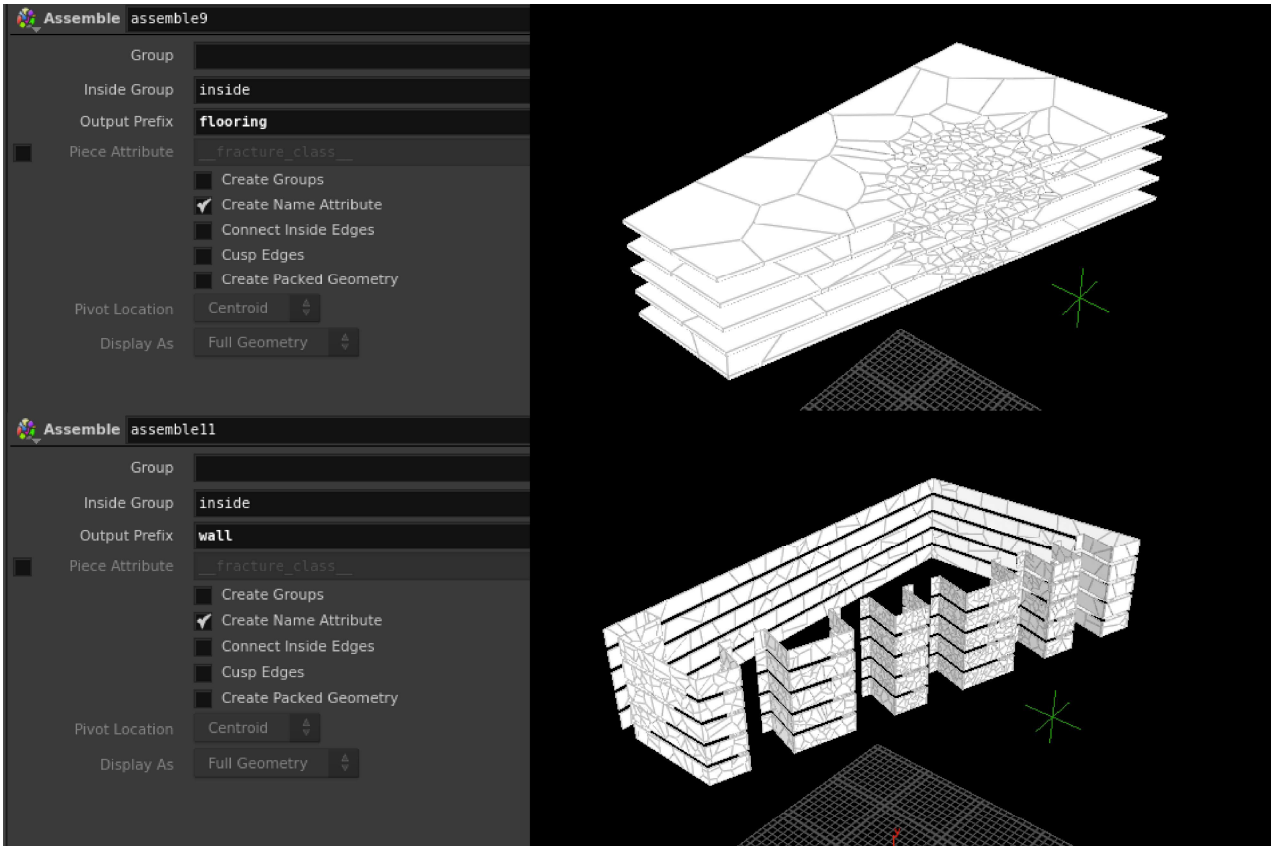


In seguito è stato inserito un operatore con l'intento di indebolire la rete di vincoli appena creata con il blocco di loop. Infatti il nodo che segue è appunto il nodo di *Attribute Promote* che trasmette la velocità dai punti alle primitive. Con il successivo nodo di *Attribute Wrangle,* invece, è stata operata l'eliminazione di tutte le connessioni dotate di una velocità, quelle che si volevano distruggere attraverso la forza dell'esplosione presente all'inizio della simulazione. Per poter compiere questa operazione si sono dovuti eliminare, appunto, i vincoli che la fase precedente di loop aveva creato.

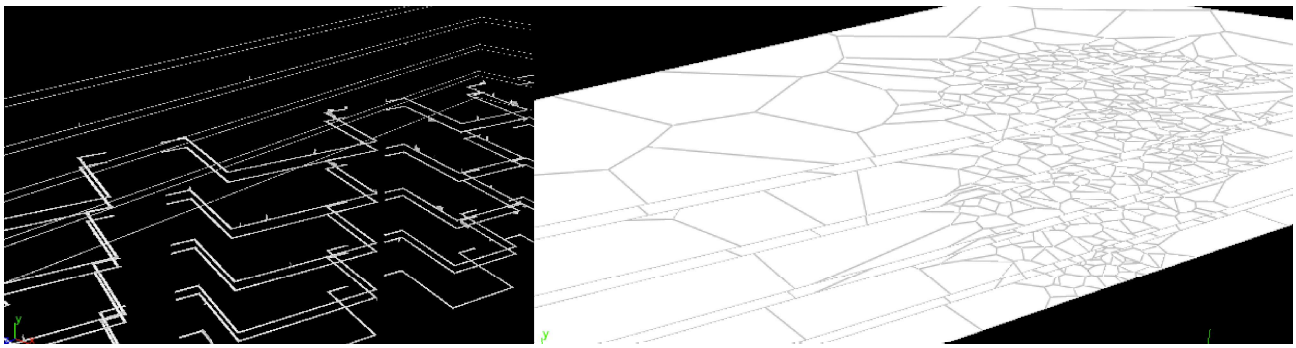
Questo operatore fa proprio questo, rimuove le primitive che hanno una velocità superiore a due. In questo modo non si rimuovono tutte completamente, ma un numero tale da poter permettere di eliminare quei vincoli che avrebbero impedito, ai pezzi racchiusi all'interno dei gruppi creati, la velocità di movimento. Quindi in questo modo si eliminano delle connessioni riducendo il network di vincoli. In questa parte è stata eseguita ancora un'operazione per regolare, sempre prima dell'ingresso in simulazione, la forza di questi vincoli. È stato quindi aggiunto un altro *Attribute Wrangle* e tramite la stringa *f@strength* si è regolata la forza di queste connessioni. Un importante passo successivo è stato la creazione dell'output riguardante la rete di vincoli appena creata. Per l'appunto, il nodo più importante che conclude qualsiasi composizione riguardante una rete di vincoli è l'*Attribute Wrangle* che, tra le altre cose, consente la creazione degli attributi che regolano l'attivazione dei vincoli all'interno della simulazione.



Anche per le porte e per le ringhiere è stata svolta lo stesso tipo di procedura, per ogni gruppo sono stati creati due rami uno per la geometria impacchettata e l'atro per la rete di vincoli, al fine di creare gli output da richiamare all'interno della simulazione. Ovviamente per quanto riguarda la creazione degli attributi che controllano l'attivazione dei vincoli, sono stati creati degli attributi che avessero dei nomi differenti, quindi univoci, per evitare che si potessero creare dei problemi una volta giunti all'interno della simulazione. La stessa operazione è stata eseguita per ogni blocco in cui era stata divisa la geometria di partenza. A questo punto si può passare alla seconda procedura di creazione dei vincoli che coinvolge i collegamenti tra gli elementi costituenti i diversi blocchi. Si prendano in considerazione i piani centrali e i rispettivi muri partendo dai corrispondenti nodi di *Assemble* che, come si ricorda, non avevano il compito di impacchettare la geometria ma solo di creare un attributo.

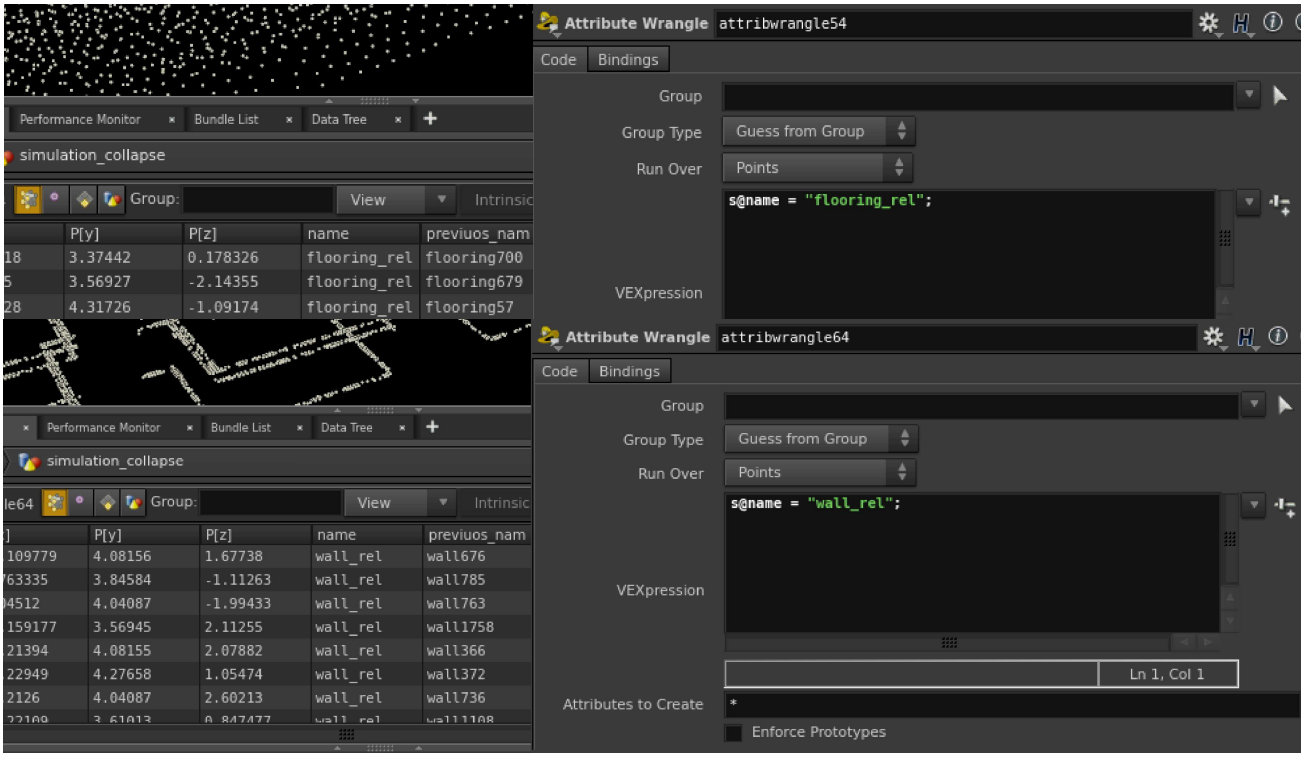


Per entrambi i rami, dopo il nodo di *Assemble*, è stato aggiunto l'operatore di *Attribute Promote* che promuove l'attributo *name* dalle primitive ai punti. È stata poi eseguita, su entrambi i lati, un'operazione che ha consentito di eliminare le parti che non si volevano includere nell'operazione di creazione delle connessioni, in quanto si volevano mettere in relazione solo tra il piano e la parte di muro che lo avrebbe toccato quindi sono state eliminate tutte le altre parti, come appunto si può osservare nello scatto successivo.

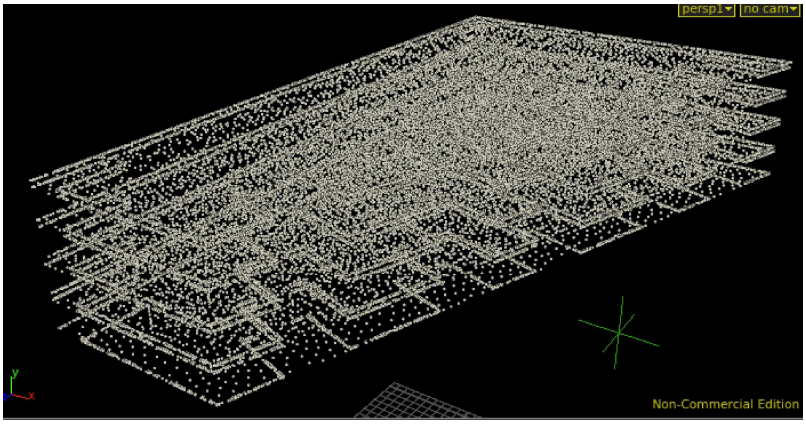


A questo punto su entrambi i rami è stato applicato un nodo di *Scatter* ricoprendo di punti le superfici ottenute dall'operatore precedente. Successivamente è stato cambiato, tramite l'*Attribute Rename*, l'attributo *name*, questo per non interferire con i precedenti attributi e lavorare su un attributo che è lo stesso ma porta un nome diverso, in *previous_name*. In realtà questo è stato un passo importante perché è stato necessario lavorare su attributo avente un nome diverso da *name*. È stata quindi eseguita questa operazione per modificarlo, per poi con un'operazione successiva creane uno nuovo, questo perché *name*

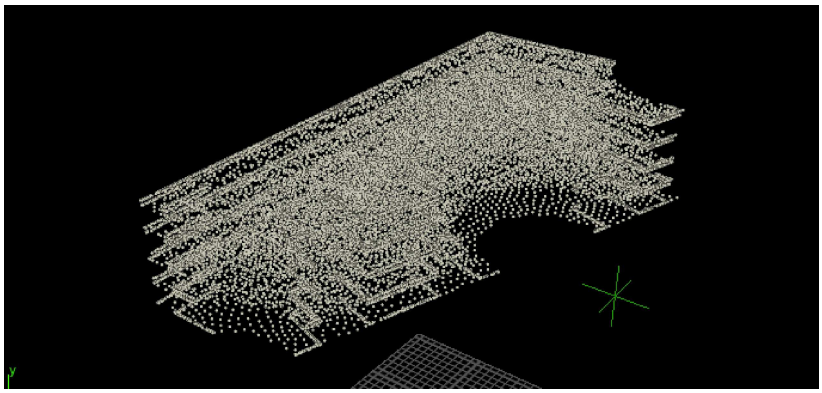
è l'attributo fondamentale per il riconoscimento dei vincoli all'interno della simulazione. Il nodo successivo è un *Attribute Wrangle*, nodo che ha permesso l'impostazione del valore per il nuovo attributo *name*.



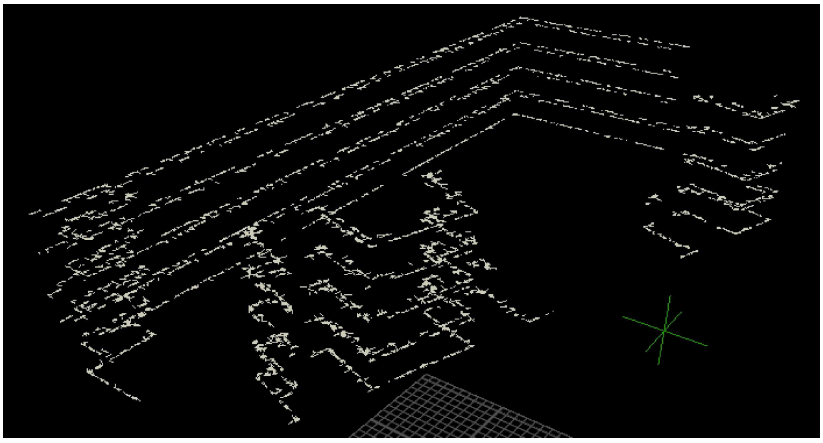
Come si può notare negli scatti precedenti gli attributi citati sono due attributi di punto. Ora è possibile mettere tutto insieme tramite il nodo di *Merge*, il cui risultato si può osservare nella foto successiva.



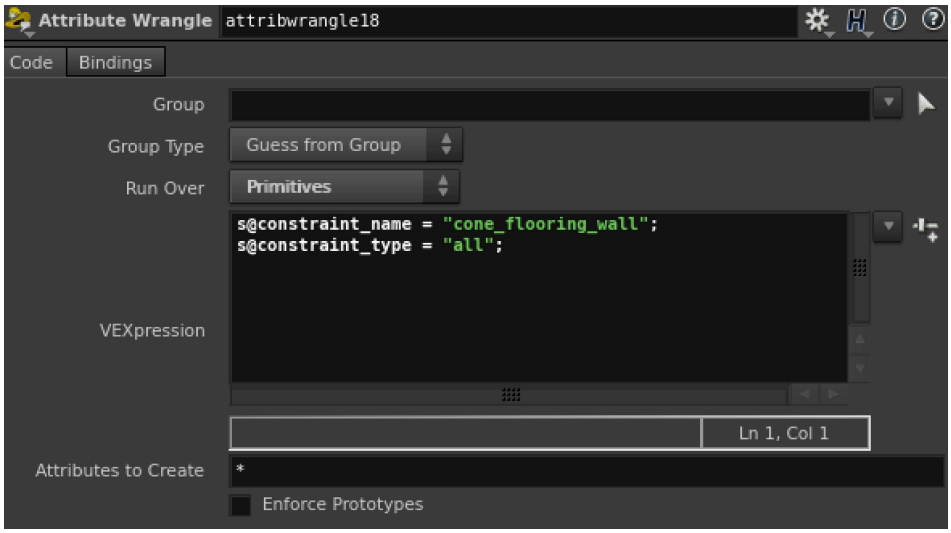
A questo punto sono presenti tutti gli ingredienti per poter completare la ricetta e quindi la costruzione di tutto il network di relazioni. Prima di creare le connessioni con il nodo *Connect Adjacent Pieces*, però, sono state eliminate alcune aree di punti per evitare che si creassero delle connessioni forti in tutto il blocco, utilizzando quindi prima un nodo di *Group* e poi un nodo di *Delete* per eliminare i gruppi di punti creati. Il risultato di questa operazione si può notare nello scatto successivo, che in sostanza è servita a rendere il blocco di elementi più debole e quindi più facile da rompere.



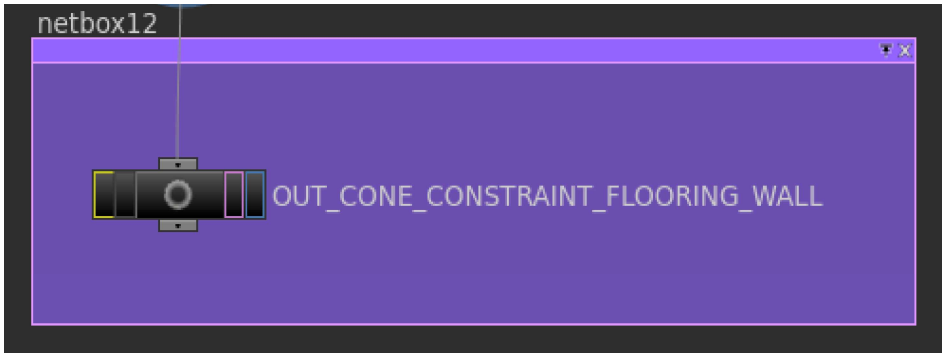
Con l'applicazione dell'operatore successivo, ovvero il *Connect Adjacent Pieces,* sono state create le connessioni solo in quei punti in cui si era interessati a creare dei legami fra pezzi.



Il nodo *Connect Adjacent Pieces* ha consentito la costruzione di tutti i collegamenti tra i due blocchi di elementi unendo i punti dei piani con quelli dei muri. L'operazione successiva è stata l'eliminazione dell'attributo *name* e il cambio del nome dell'attributo *previous_name* in *name*. In modo tale da far rimanere un solo attributo di punto che portasse l'identificativo *name*. Il nodo successivo è l'*Attribute Wrangle,* utilizzato per associare i nuovi attributi che identificano la nuova rete di vincoli. Per far si che si possano attivare i vincoli appena creati all'interno della simulazione bisogna far si che questi attributi vengano creati come attributi di primitive.



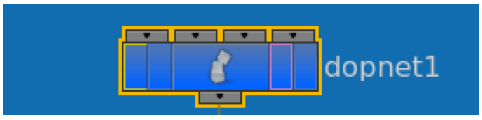
Il processo si può concludere con l'inserimento del nodo di output da richiamare all'interno della simulazione, riportato nell'immagine qui sotto.



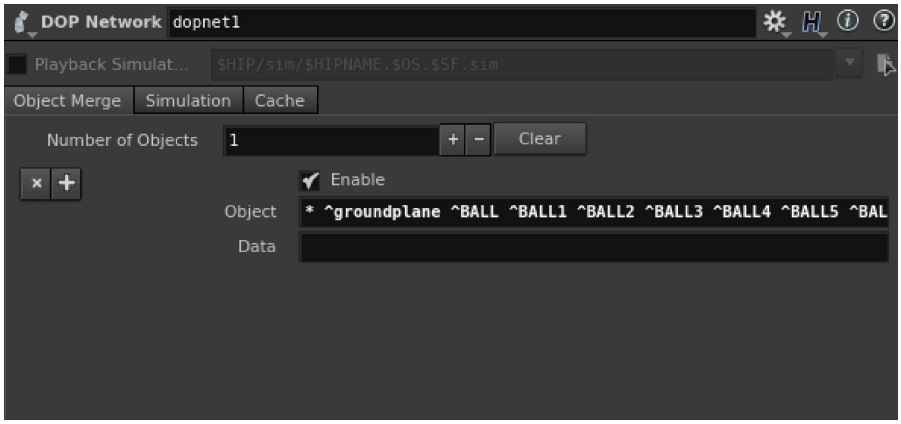
Questa seconda metodologia è stata ripetuta per unire tutti gli elementi appartenenti a blocchi differenti creati dalla prima divisione del modello iniziale del palazzo. Una volta conclusa anche questa operazione di creazione dei vincoli è possibile proseguire con la successiva fase di simulazione.

Simulazione

In questa fase si capirà come utilizzare i network dinamici del software *Houdini* per creare delle simulazioni. Nello specifico, ci si concentrerà, come al solito, sulla simulazione di *Destruction*. In *Houdini,* è possibile creare degli oggetti di simulazione ed applicare ad essi dei solver che ne determinano il comportamento. *Houdini* offre degli oggetti e dei solver per costruire simulazioni di ogni genere, proprio come le cinque differenti simulazioni che costituiscono il progetto oggetto di questa trattazione. All'interno di *Houdini,* esistono, inoltre, dei solver specializzati nell'integrazione dinamica con gli altri componenti del programma. Una tipica configurazione per una simulazione dinamica, come nel caso di quella di *Destruction,* consiste in un singolo oggetto di simulazione sostenuto da un solver, come più spesso accade, e si può notare all'interno della simulazione di *Destruction* creata per questo lavoro, essa può possedere al suo interno più oggetti di simulazione. La simulazione di *Destruction* qui trattata, nello specifico, contiene più oggetti di simulazione, gestiti dai nodi *RBD Packed Object,* uniti in un unico oggetto da un nodo di *Merge*, al quale è poi stato applicato un singolo solver: il *Bullet Solver*. Alle simulazioni dinamiche possono essere applicati dei vincoli, i quali, possono limitare la posizione di un oggetto rispetto a quella di un altro, come nel caso dei vincoli utilizzati per mettere in relazione, e quindi in sostanza tenere insieme, i pezzi degli oggetti che sono stati rotti nel processo di fratturazione, fino al punto di rottura. A questo proposito, e quindi possibile applicare delle forze agli oggetti di simulazione, in particolare, gli oggetti di simulazione possono avere solver multipli, i quali possono condividere le stesse forze. Si possono, inoltre, applicare più forze ad ogni oggetto. I vincoli sono quindi un tipo di relazione che può essere associata a degli oggetti di simulazione, i quali possono avere delle relazioni di collisione. Le reti dinamiche create stabiliscono degli alberi di oggetti ai quali vengono applicati dei dati. A questo proposito, la *viewport* mostra la geometria animata creata dalla simulazione, mentre la *details view* mostra tutti i dati di simulazione applicati a ciascun oggetto in un formato comparabile a un foglio di calcolo chiamato *spreadsheet*. L'oggetto utilizzato per gestire la simulazione di *Destruction,* è l'operatore *DOP Network,* il quale, appunto, contiene al suo interno la simulazione della dinamica del crollo, quindi una simulazione *DOP*. Il nodo in questione contiene al suo interno tutti i pezzi per poter creare un'animazione della geometria, ovvero, gli oggetti di simulazione che si riferiscono alla geometria *SOP*, il solver e i vincoli, ovvero i *Constraint Network* che sono stati utilizzati per scegliere il tipo di vincolo e la sua soglia di rottura. In sostanza, gli oggetti di simulazione sono costruiti e controllati dai nodi *DOP* contenuti nel *DOP Network*.
Il nodo che racchiude la simulazione è, quindi, il nodo *DOP network.*



Questo oggetto contiene al suo interno una simulazione dinamica. Le impostazioni di questo nodo sono state modificate solo nella sezione *Object Merge*, andando ad eliminare, tramite il comando *"* ^"*, che significa *"tutto, tranne"*, tutti quegli oggetti che non si sarebbero voluti visualizzati in simulazione, come gli elementi aggiunti per rompere ulteriormente la struttura, che riportano, in questo caso, come si può notare nell'immagine successiva, il nome *BALL*.



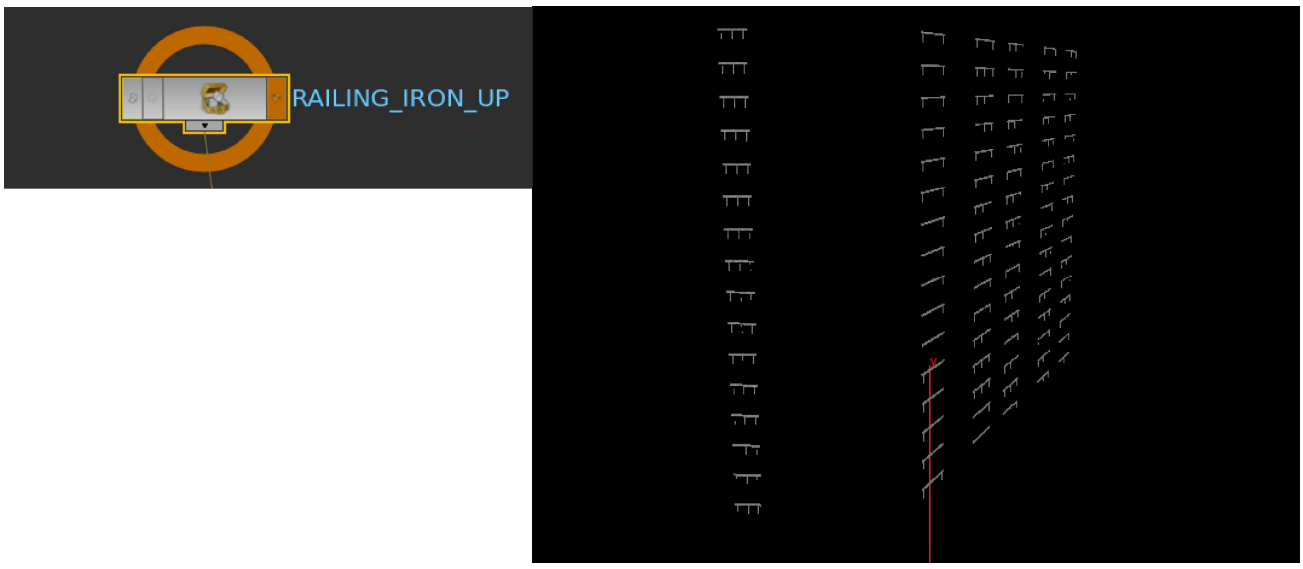
Il *DOP Network* contiene, al suo interno, tutti gli operatori *DOP* che costituiscono la simulazione. Per capirne il funzionamento e la struttura, è necessario scendere di livello, quindi entrare all'interno di questo nodo. I primi operatori che si incontrano, una volta scesi a livello *DOP*, sono i nodi dinamici *RBD Packed Object,* che racchiudono, al loro interno, i pezzi di geometria impacchettata confezionati, a livello oggetto, in precedenza. Il *DOP Network,* quindi, crea un singolo oggetto *DOP,* dalla geometria *SOP,* che rappresenta un numero di oggetti *RBD*. I parametri utilizzati per la seguente simulazione, riferiti al nodo *DOP Network,* sono:

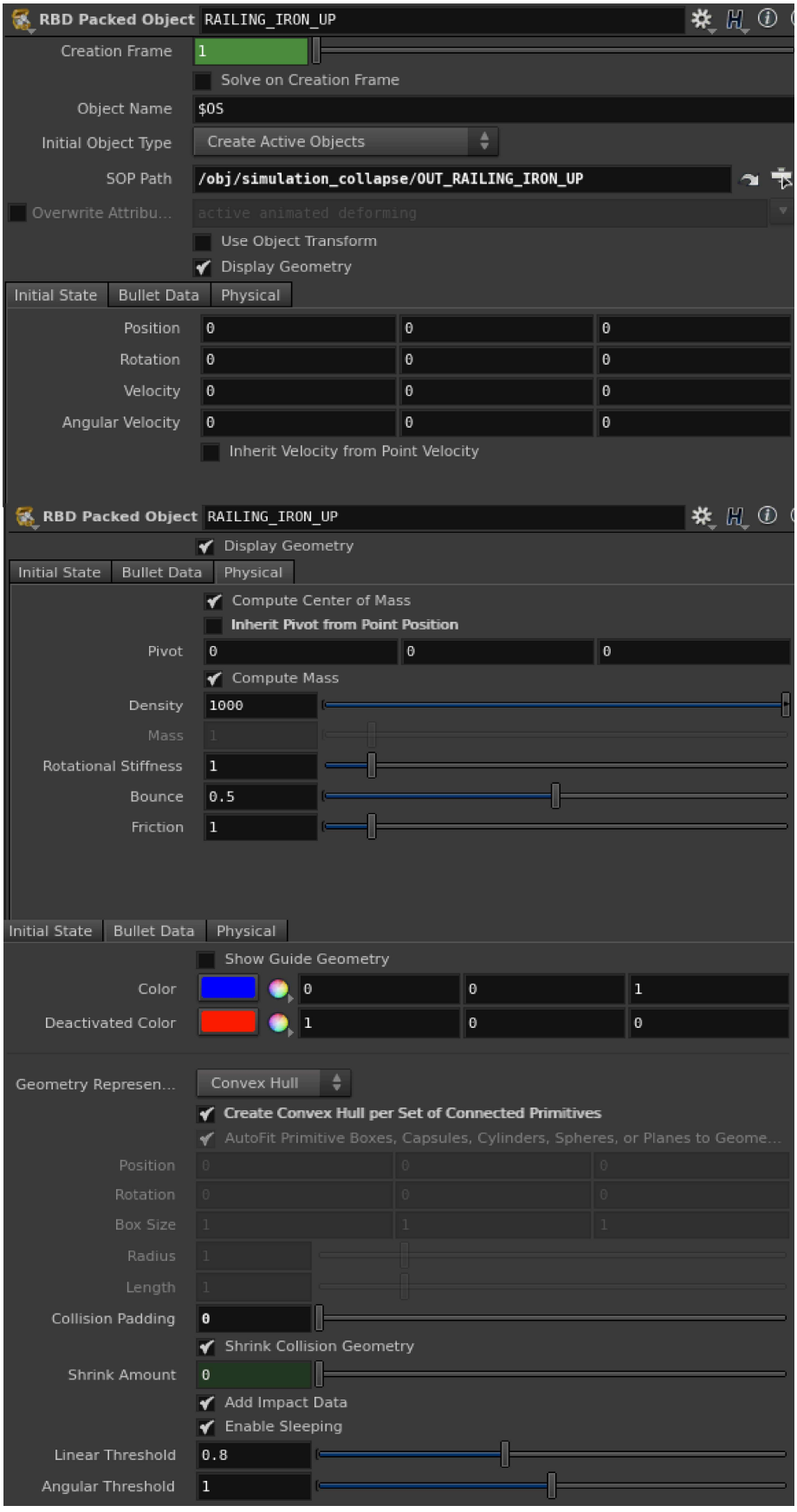
1- *Initial Object Type*. Questo campo specifica lo stato iniziale dell'oggetto. Per la simulazione di *Destruction,* sono stati utilizzati, essenzialmente, due tipologie di oggetti. Il *Create Active Objects*, che fa in modo che gli oggetti vengano simulati e reagiscano in risposta agli altri oggetti all'interno della simulazione, e il *Create Static Objects,* che fa in modo che l'oggetto non si possa muovere, o reagisca, in risposta agli altri oggetti presenti all'interno della simulazione.
2- *SOP Path*. Questo parametro indica il percorso di un nodo *SOP* che contiene la geometria dell'oggetto. Quindi, in questo caso, è stato specificato, il percorso della geometria impacchettata.

Questi due parametri sono presenti nella sezione generale del nodo. Esiste, poi, la sezione che riguarda lo stato iniziale da attribuire all'oggetto. In particolare, questa sezione, all'interno della simulazione di *Destruction*, consente di modificare la velocità, la posizione, la rotazione e la velocità angolare dell'oggetto. Sempre nella scheda che riguarda lo stato iniziale da attribuire ad un oggetto, è, inoltre, presente il parametro *Inherit Velocity from Point Velocity,* il quale è stato utilizzato per far si che a livello *DOP* l'oggetto ereditasse la velocità già settata a livello *SOP*. Nella sezione *Bullet Data,* riferita al nodo *RBD Packed Object,* vengono presentati i parametri che gestiscono le impostazioni riguardanti le collisioni:

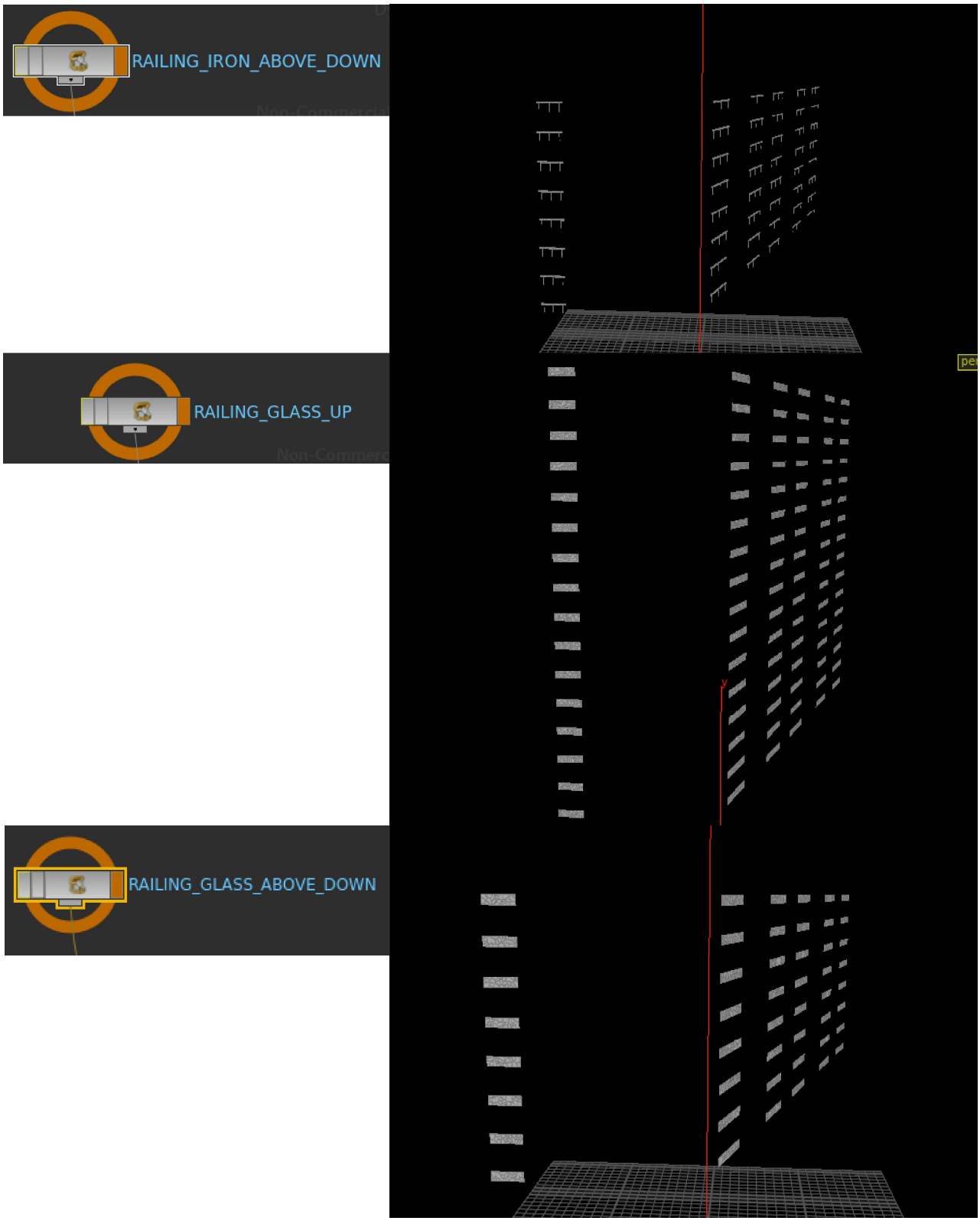
1- *Geometry Representation*. Questo parametro specifica la forma utilizzata dal motore *Bullet* per rappresentare l'oggetto. Nel caso della simulazione di *Destruction* in esame è stata scelta, per tutti gli oggetti presenti, l'opzione *Convex Hull,* che rappresenta la forma di default per l'oggetto. Il *Bullet Solver,* quindi, crea una forma di collisione del guscio convesso della geometria dei punti.
2- *Create Convex Hull Per Set Of Connected Primitives*. Quando la rappresentazione geometrica è *Convex Hull,* come in questo caso, il *Bullet Solver* crea una forma composta che contiene una forma di collisione *Convex Hull* per ogni insieme di primitive connesse all'interno della geometria.

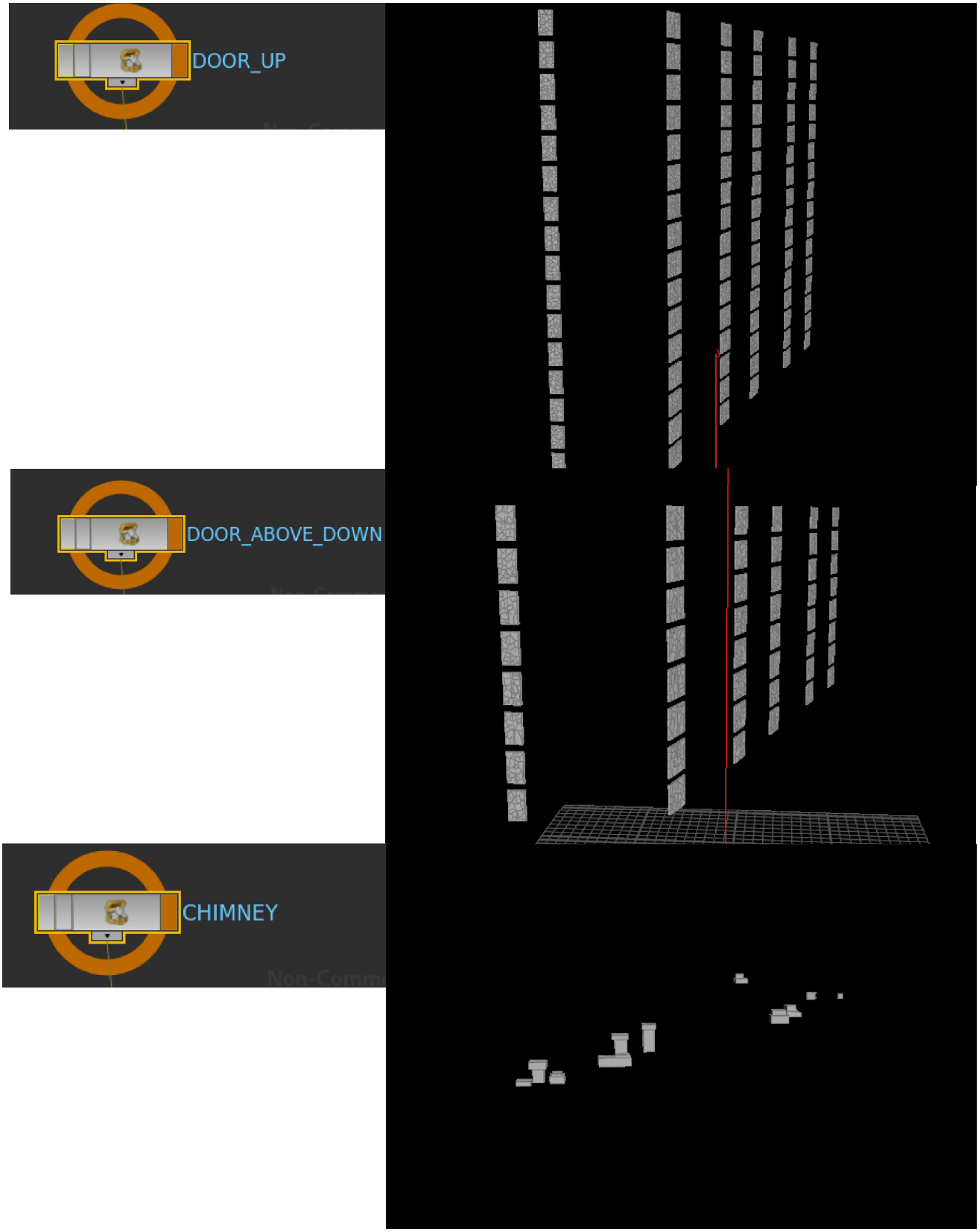
Questi sono essenzialmente i parametri più rilevanti di questa sezione, gli atri sono serviti per regolare la forma di collisione che *Bullet* utilizza per racchiudere gli oggetti. Per gli operatori *RBD Packed Object* sono utilizzate le impostazioni più compatibili con il lavoro svolto da *Bullet* per ridurne il livello di complessità, ovvero quelle che il solver *Bullet* avrebbe gestito con più facilità. Nella sezione dell'operatore che riguarda la fisica dell'oggetto sono state attribuite le caratteristiche che ne determinano il comportamento, come ad esempio la massa. Per ottenere la dinamica desiderata queste impostazioni sono state piuttosto alterate rispetto a quelle che possono essere le misure di un oggetto reale, tutto per conferire un grado di dinamica di simulazione realistica. Si presenta, di seguito, una panoramica di tutti gli *RBD Packed Object* utilizzati all'interno del *DOP Network,* giusto per tenere in memoria gli elementi presenti all'interno della simulazione di *Destruction*.

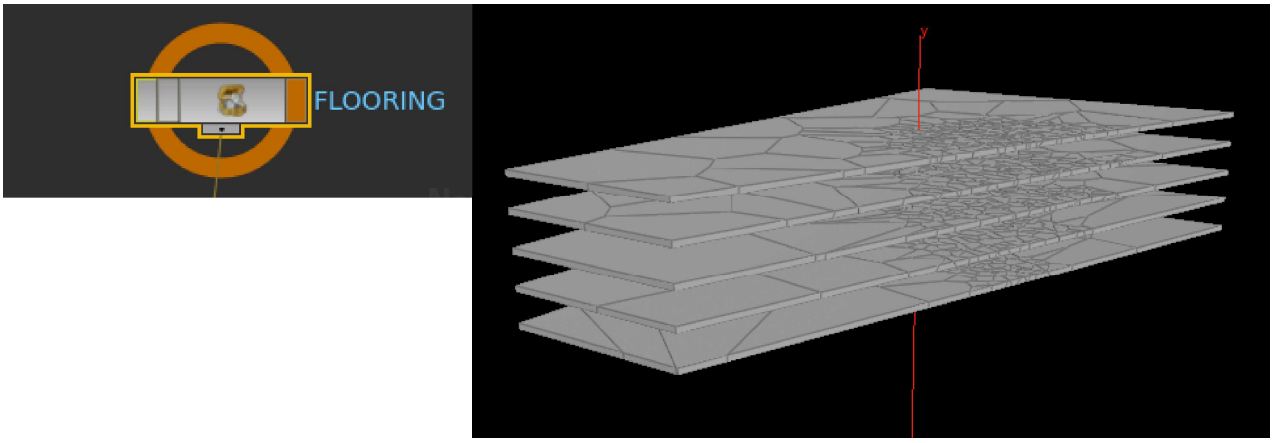




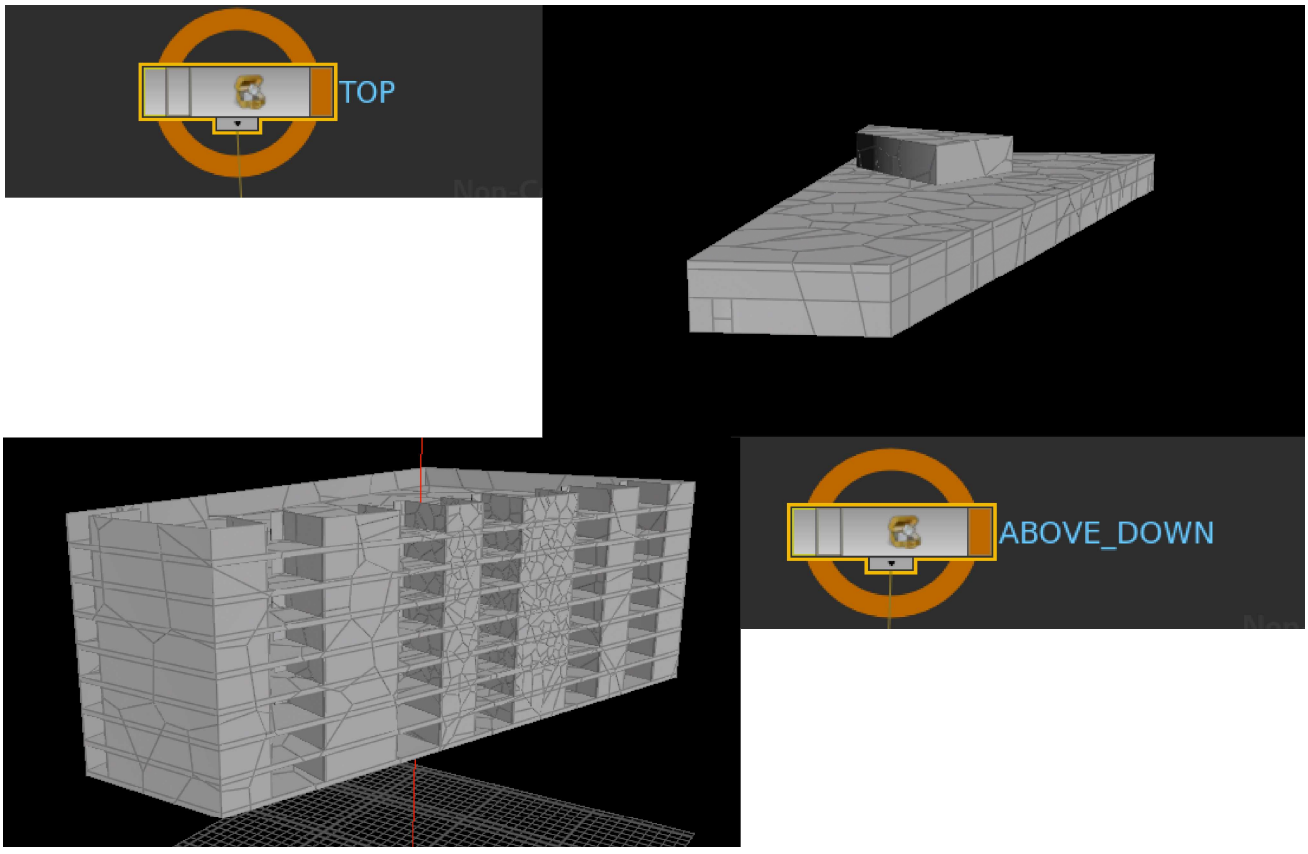
Le impostazioni qui sopra si riferiscono all'oggetto *RAILING_IRON_UP* e definiscono, in questo caso, l'oggetto come attivo, come si può notare nella prima schermata.


RAILING_IRON_ABOVE_DOWN
RAILING_GLASS_UP
RAILING_GLASS_ABOVE_DOWN

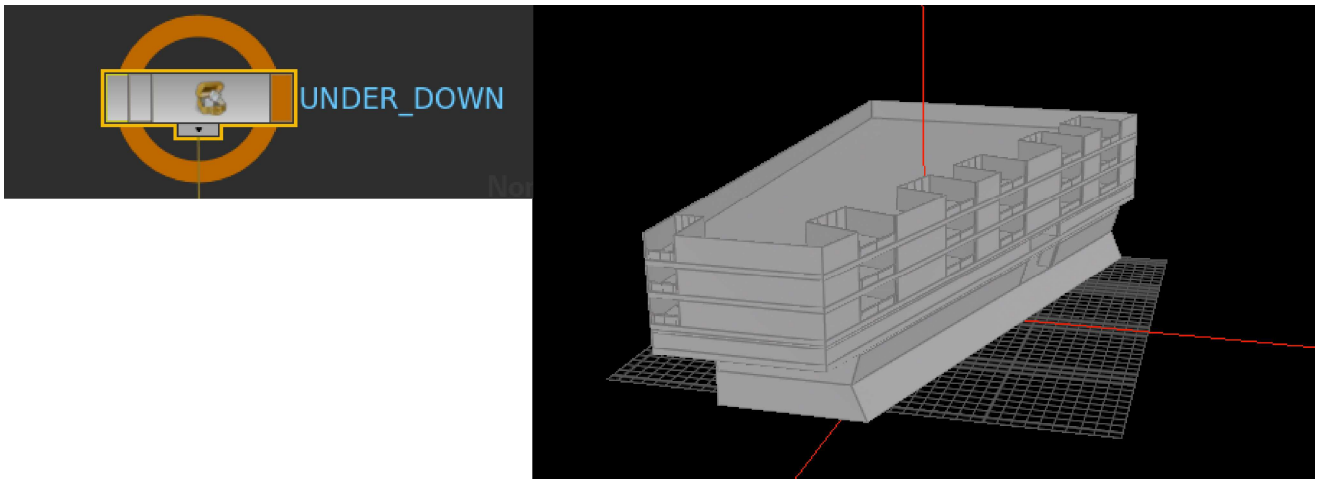

DOOR_UP
DOOR_ABOVE_DOWN
CHIMNEY



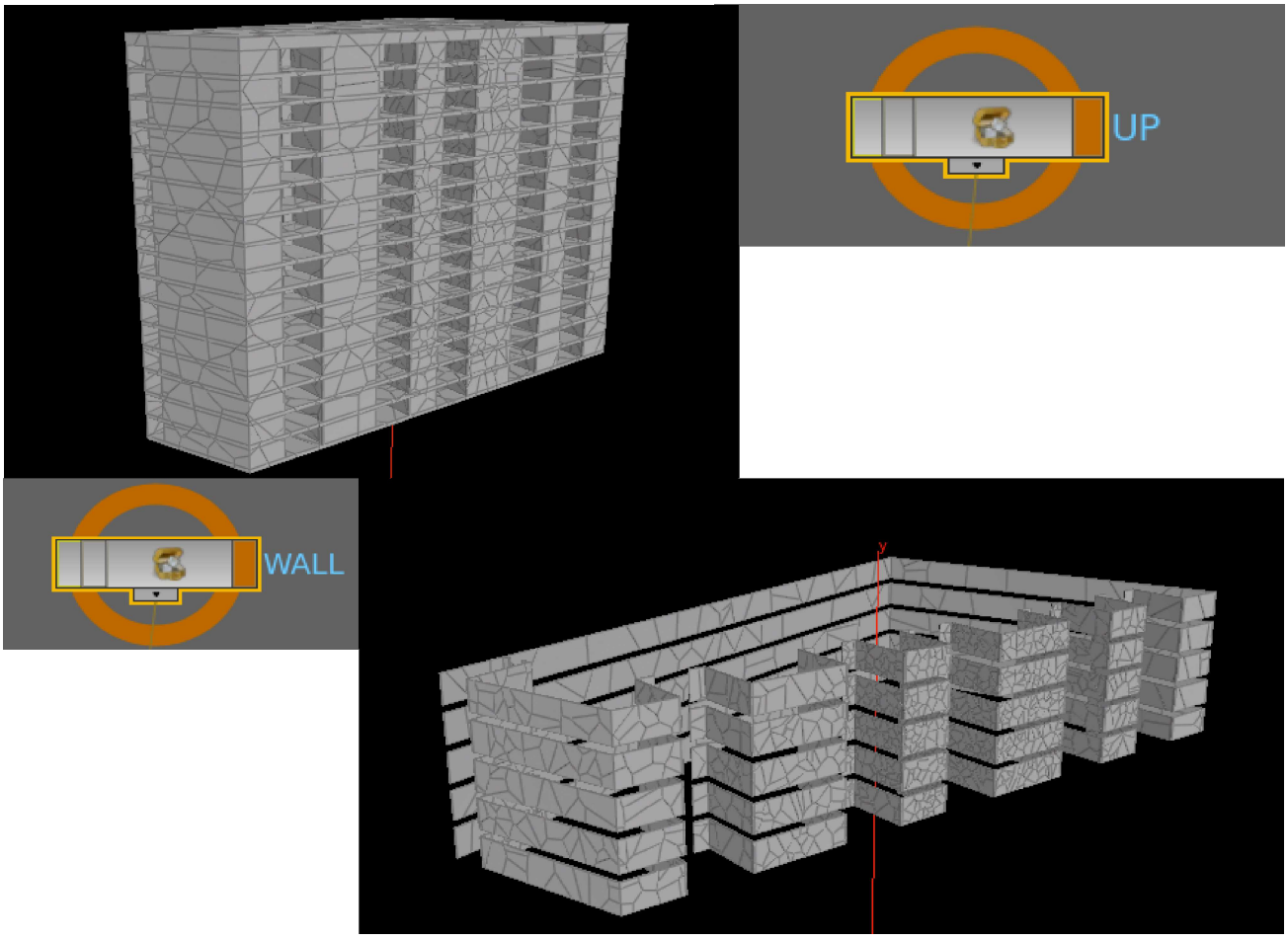
Per tutti gli elementi fino a d'ora presentati le impostazioni sono le stesse del nodo *RAILING_IRON_UP*. Gli oggetti seguenti hanno più o meno le stesse impostazioni dei precedenti, l'unica differenza sta nel parametro *Density* della sezione riferita alla fisica dell'oggetto, la quale è stata aumentata.

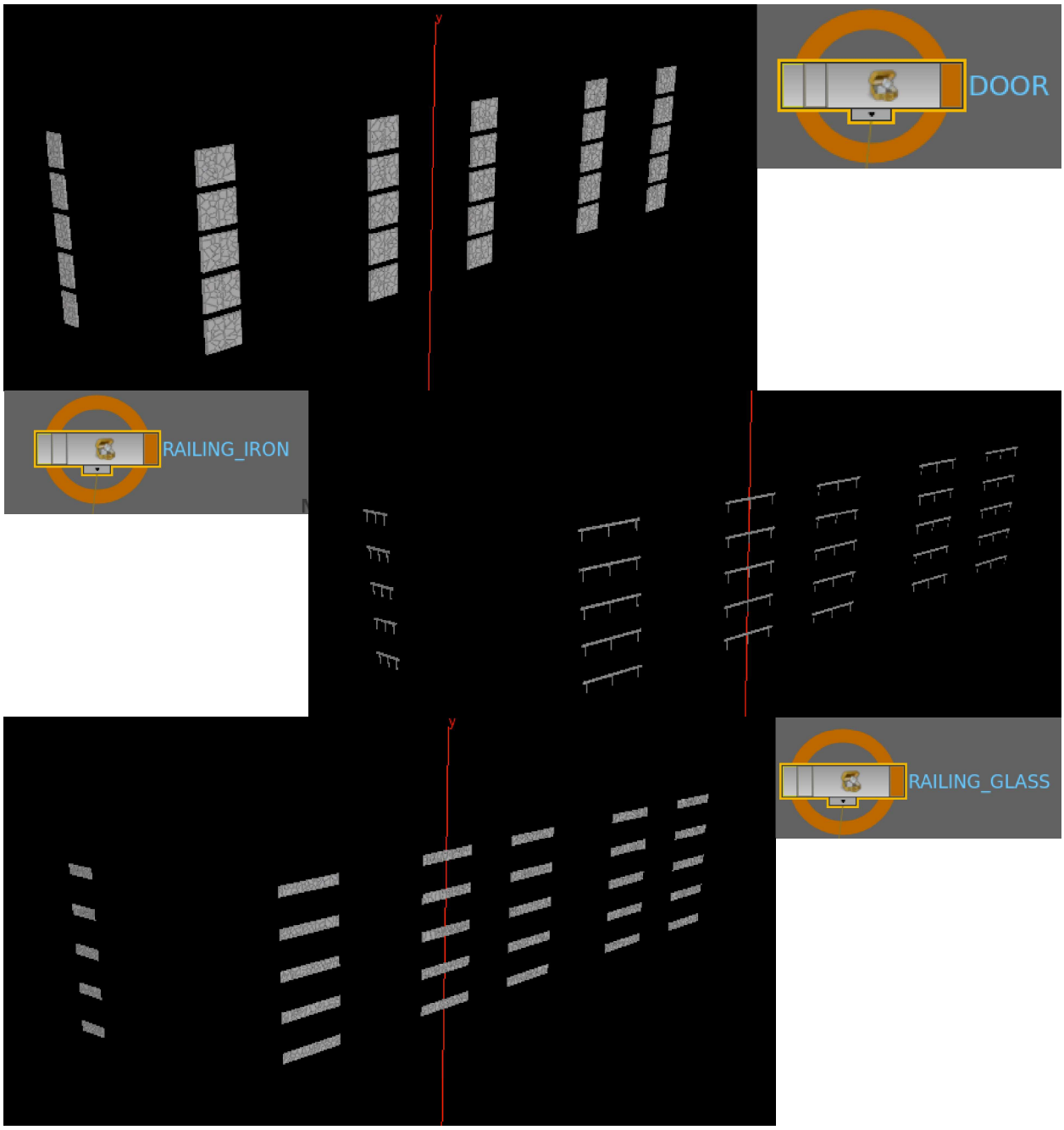


L'oggetto successivo, invece, come stato detto in fasi precedenti, è statico, in quanto rendere l'elemento statico non avrebbe influito sulla dinamica di simulazione e, inoltre, a livello pratico, non sarebbe stato visualizzato nel render perché non compreso all'interno della vista della camera utilizzata nella scena, quindi non incluso.

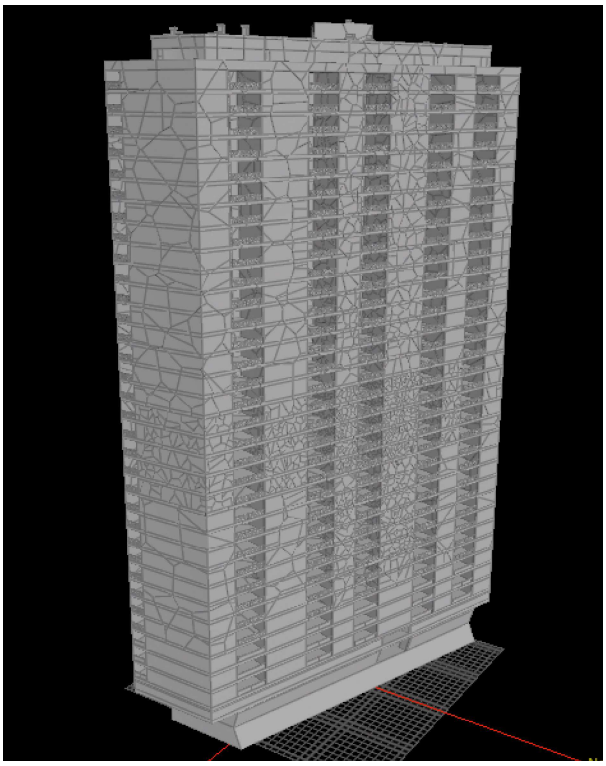


Di seguito, invece, viene presentata la carrellata di elementi ai quali è stata attivata l'impostazione *Inherit Velocity from Point Velocity*, che permette di ereditare la velocità dalla geometria *SOP* senza doverne iniettare una iniziale.

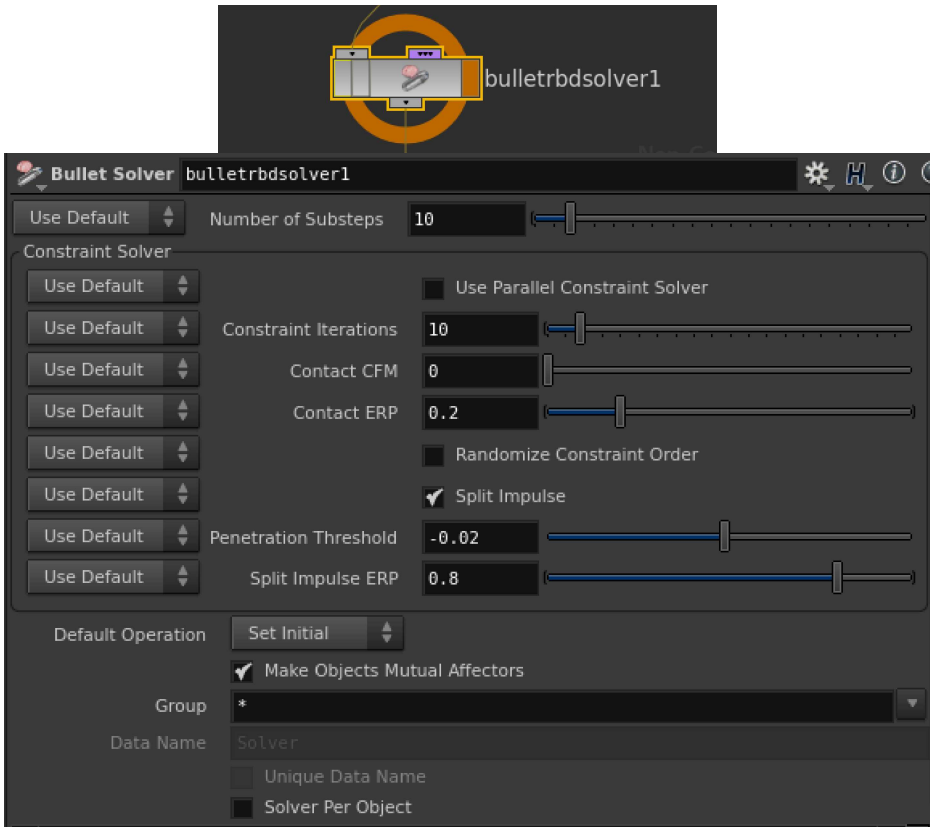



DOOR
RAILING_IRON
RAILING_GLASS

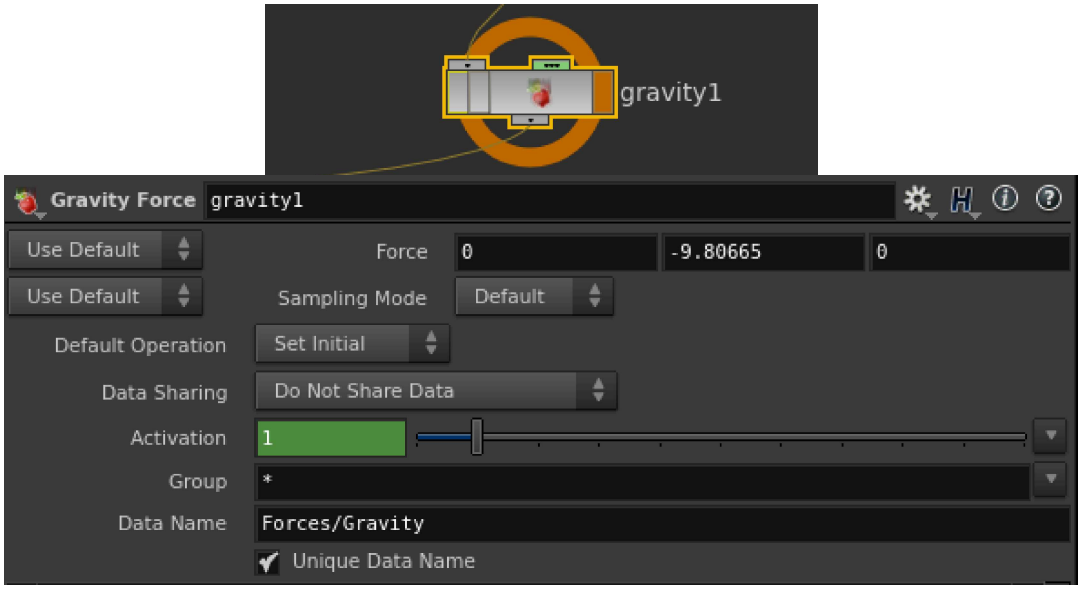
Tutti questi oggetti sono stati uniti, tramite il nodo di *Merge*, per poter affrontare tutte le operazioni successive riguardanti per lo più l'attivazione dei vincoli.



Subito dopo l'operatore di *Merge* è stato connesso il *Bullet Solver*, forse il nodo più importante, il quale determina la dinamica di simulazione e configura, appunto, il *Bullet Dynamics solver.*

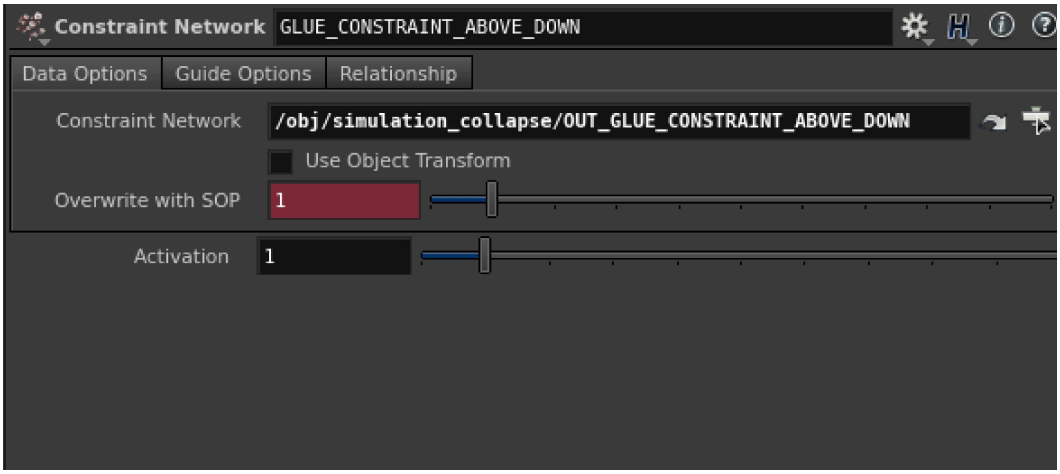


Dopo il *Bullet Solver*, che gestisce la dinamica e le collisioni fra gli oggetti all'interno della simulazione, è stato inserito il nodo di *Gravity*, il quale è anch'esso un nodo di dinamica che applica la gravità come forza all'oggetto, infatti se non vengono settate le relazioni fra i pezzi fratturati, questi cadono verso il basso separandosi perché sottoposti alla sua influenza.

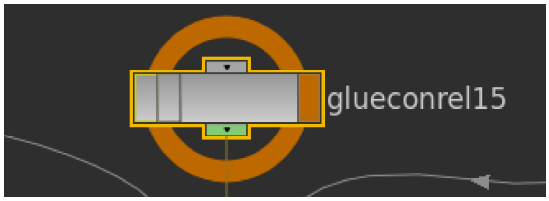


A questo punto non rimane da applicare che la rete di vincoli creata nella fase precedente. Per poter attivare i vincoli è necessario inserire il nodo *Constraint Network*. Questo è un nodo di dinamica composto da tre ingressi: quello che riceve gli oggetti, quello che specifica il tipo di vincolo che deve essere settato e un ingresso opzionale che consente l'inserimento di un ulteriore *solver* che può essere utilizzato, come è stato fatto in questo progetto, per rompere ulteriormente alcuni legami. L'operatore *Constraint Network* vincola delle coppie di oggetti *RBD* insieme, in accordo con una rete di poligoni, ovvero quella che è stata appunto creata tramite il nodo *Connect Adjacent Pieces* a livello *SOP*.

Una delle impostazioni importanti di questo operatore è il parametro *Constraint Network* che specifica la geometria *SOP* da utilizzare come rete di vincolo.



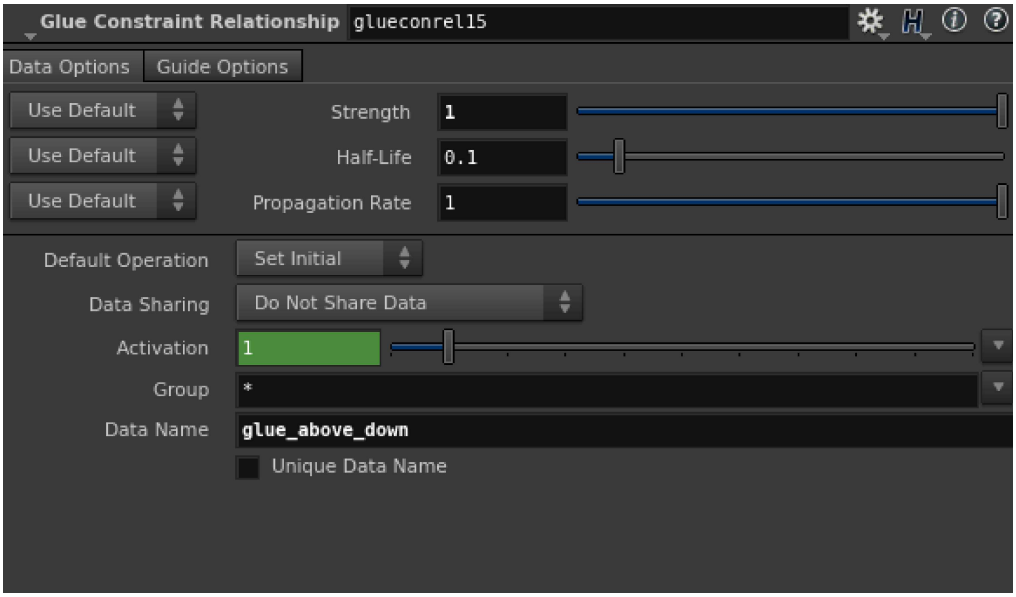
Come tipo di relazione da attribuire al *Constraint* è stato scelto, per semplicità, un vincolo di *Glue*. È stato poi modificato il suo comportamento tramite l'ausilio di un *solver* aggiuntivo che ha indebolito ulteriormente i vincoli rendendoli, di conseguenza, più facili da rompere.



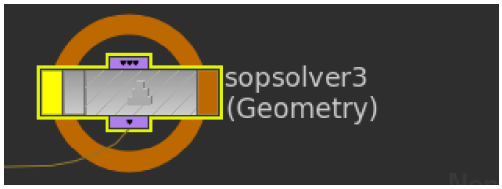
L'operatore *Glue Constraint Relationship* è un nodo di dinamica. Con questo tipo di relazione gli oggetti collegati fra loro si muovono come un unico intero oggetto fino a quando non viene applicata una forza sufficiente da rompere il vincolo. Essenzialmente i parametri che sono stati settati per questo operatore riguardano la resistenza del vincolo:

1- *Strength*. Specifica la forza del legame. Se viene indicato un valore pari a -1 il legame viene considerato indistruttibile. In alcuni casi, per alcuni blocchi, è stato specificato un valore pari a 1, questo perché nella fase *SOP*, per questi blocchi, era già stato creato un valore di *strength*. Se si fosse specificato un valore diverso da uno, questo sarebbe stato moltiplicato per quello determinato nel nodo *SOP* e siccome non si voleva alterare questo numero, nella fase *DOP* è stato impostato un valore neutro.

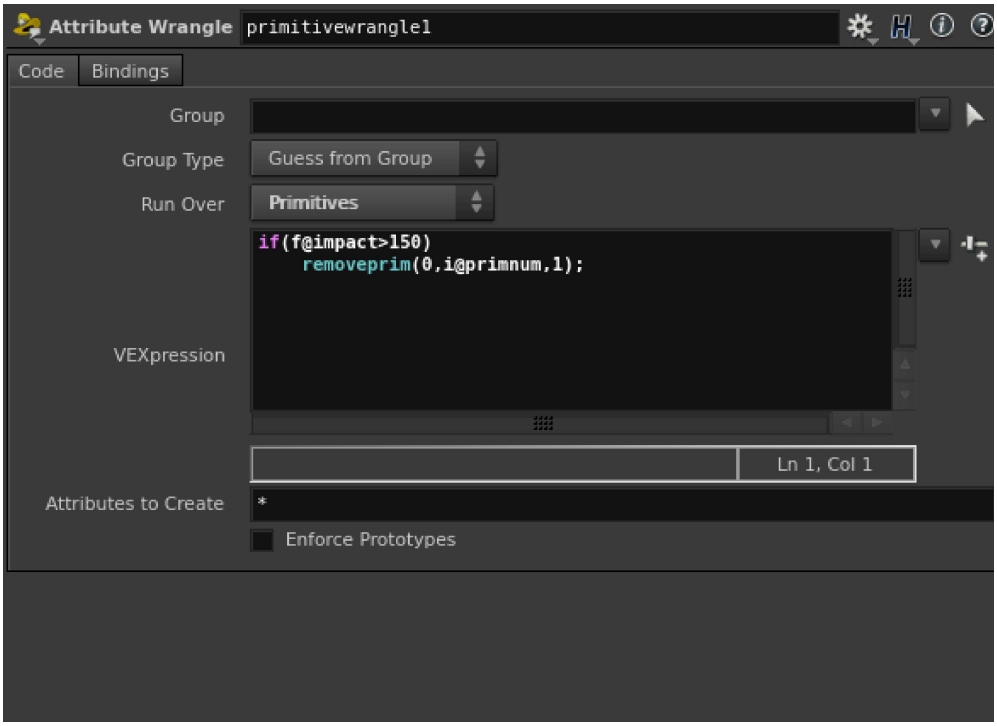
2- *Data Name*. Qui viene specificato il nome utilizzato per attivare il legame. In questo caso questo parametro si riferisce all'attributo creato nella fase finale del processo di creazione dei vincoli gestito nei *SOP*.



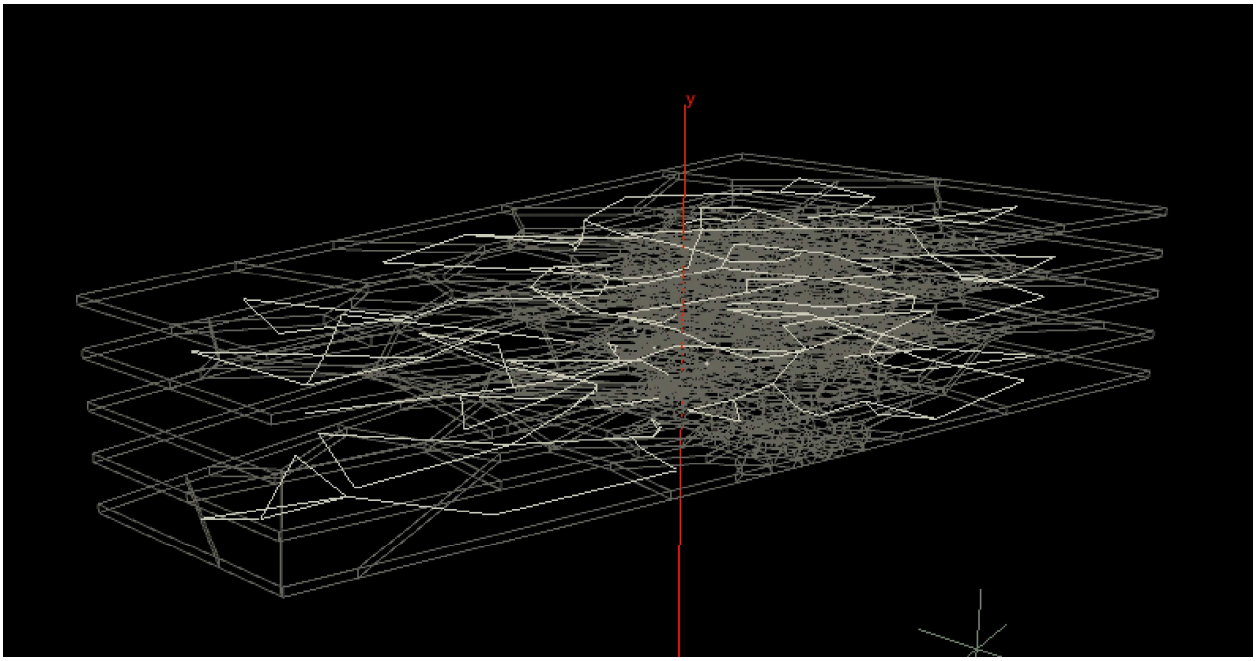
Ad alcuni operatori *Constraint Network* è stato aggiunto un ulteriore nodo di *Sop Solver*.



Il *Sop Solver* è un operatore dinamico. Al suo interno è stato modificato un parametro per gestire alcune relazioni all'interno della geometria tramite un nodo di *Attribute Wrangle*.



In questo caso, il codice presente nell'immagine precedente, rimuove le primitive sottoposte a una forza di impatto pari a 150. I nodi che compongono il resto del network, che costituisce la simulazione all'interno del nodo *DOP,* da questo momento in poi sono essenzialmente tutti dei nodi di *Constraint Network,* che quindi consentono di attivare tutto il resto dei vincoli presenti a livello *SOP*, i quali si riferiscono a tutti gli altri oggetti presenti a livello di geometria. Le impostazioni utilizzate all'interno di questi operatori sono all'incirca le stesse e i *Constraint* creati sono tutti vincoli di *Glue,* anche per le relazioni tra gli elementi che corrispondo a oggetti o blocchi differenti.
La fase di simulazione è stata completata in circa un mesetto. La difficoltà principale è stata, per lo più, gestire la dinamica di simulazione, e quindi fare in modo che il palazzo crollasse secondo una logica specifica. Per ottenere, appunto, una dinamica che soddisfacesse le aspettative di realismo ci è voluto un tempo molto più lungo rispetto a quello utilizzato per impostare la struttura di nodi all' interno del network di simulazione, per la quale ci sono voluti solamente un paio di giorni. È utile affrontare, ora, una panoramica un po' più ampia per conoscere le varie tipologie di vincoli offerti da *Houdini* e quindi dal sistema che gestisce la dinamica dei corpi rigidi, ovvero *Bullet*.
Un vincolo è una relazione che si crea tra due oggetti, o in questo caso tra pezzi che possono appartenere allo stesso oggetto o a oggetti differenti. *Houdini* mette a disposizione delle configurazioni automatiche che l'utente può utilizzare in modo molto semplice e veloce per creare dei *Constraint*, ma in ogni caso è preferibile costruire queste strutture di vincoli manualmente per averne il pieno controllo, gestendo ad esempio, come nel caso del progetto, la soglia di rottura all'interno di una simulazione di *Destruction*.
All'interno di un nodo di simulazione è possibile gestire i vincoli attraverso l'operatore *Constraint Network* che utilizza la geometria per stabilire al suo interno delle relazioni, nello specifico all'interno della geometria vengono create delle polilinee, come si può notare nell'immagine successiva, che rappresentano appunto i vincoli.



All'interno dell'operatore *Constraint Network* è stato poi gestito il parametro *constraint_name* per attivare le connessioni definite nell'*Attribute Wrangle* a livello *SOP*. Utilizzare il nodo *Constraint Network* è l'unico modo per creare dei *Constraint*, ovvero delle relazioni fra i pezzi quando abbiamo a che fare con *RBD Packed Objects*, visto che non si possono utilizzare dei normali vincoli dinamici. Le principali tipologie di vincoli che si possono definire attraverso il nodo *Constraint Network* mediante il software *Houdini* sono:

1- *Pin Constraint*. Questo *Constraint* vincola un oggetto a stare ad una certa distanza da un punto di ancoraggio. L'oggetto, in pratica, viene costretto ad un punto nello spazio, il quale viene utilizzato come un perno. Il *Pin Constraint* è anche detto *Hard Constraint* e consente in sostanza di fissare un elemento ad un altro. Questo tipo di legame non si può rompere, esso definisce quindi una relazione di vincolo che deve sempre essere soddisfatta.

2- *Spring Constraint*. Questa tipologia di vincolo applica delle forze per cercare di mantenere un oggetto ancorato ad una certa distanza da un punto di ancoraggio. In sostanza vincola un oggetto a un punto nello spazio utilizzando una molla come connessione e quindi può essere utilizzato per simulare, appunto, delle molle. Il parametro *Spring Strength,* presente nelle impostazioni dello *Spring Constraint,* controlla quanto la molla vincola l'oggetto.

3- *Cone Twist Constraint*. Costringe un oggetto a rimanere una certa distanza dal vincolo e ne limita la rotazione. Può essere usato per creare dei giunti con un limitato intervallo di movimento, come ad esempio una spalla.

4- *Slider Constraint*. Costringe un oggetto a ruotare e a traslare lungo un unico asse, limitandone, su tale asse, la rotazione e la traslazione. Può essere usato per creare oggetti capaci di muoversi su un unico asse, come ad esempio dei pistoni.

5- *Glue Constraint*. Vincola due elementi a muoversi come fossero un oggetto unico fino al momento della rottura della connessione che li lega.

Il risultato ottenuto dalla fase di simulazione è stato un *Flipbook*, attraverso il quale è stato possibile verificare l'accuratezza della simulazione, per poi passare alle successive fasi, da ultimare sempre all'interno di *Houdini*, di ottimizzazione, di applicazione dei materiali e di *Rendering*. Di seguito è riportato il *Flipbook* della simulazione di *Destruction*.

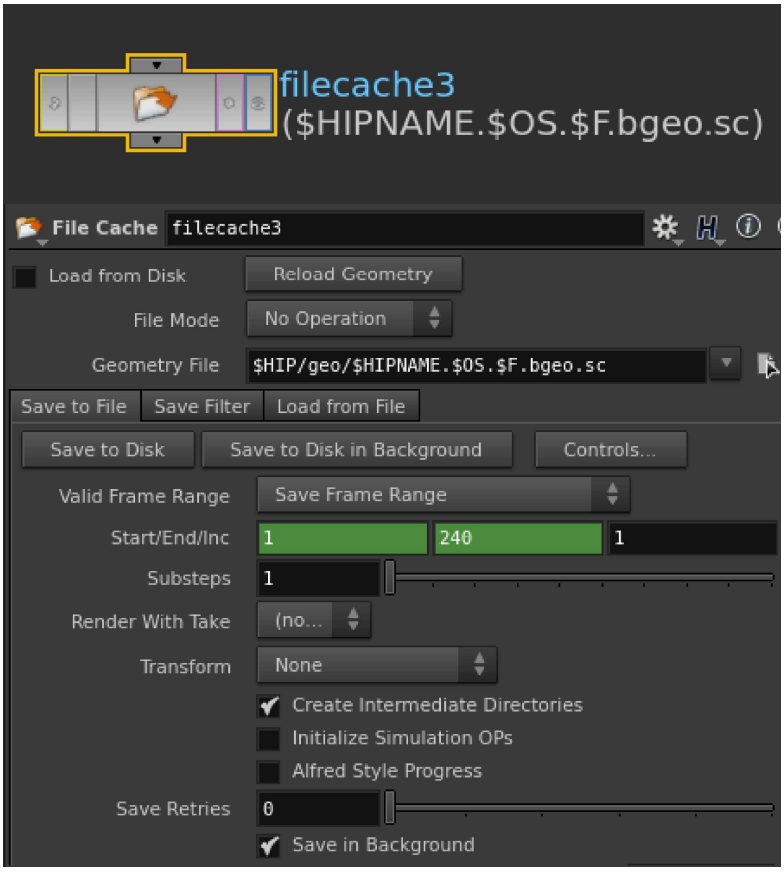
Preparazione pre-Rendering: Ottimizzazione, Materiali e Luci

Introduzione

Questo passo è stato necessario per poter ottenere come risultato i render finali, in quanto la scena è stata preparata al fine di ricostruire la sua veridicità e per meglio farla assomigliare all'immagine dello scatto di riferimento. Sono quindi stati aggiunti i materiali e un'illuminazione adeguata. Per quanto riguarda invece le operazioni di ottimizzazione sono state necessarie per facilitare il flusso di lavoro e ridurre il livello di complessità dato dai lunghi calcoli necessari.

Ottimizzazione

Una volta creato il nodo di simulazione e costruito la giusta dinamica per il crollo dell'edificio sono state eseguite alcune operazioni di ottimizzazione per il proseguimento delle elaborazioni successive. In prima battuta, è stato utilizzato un nodo File Cache per memorizzare la simulazione su disco

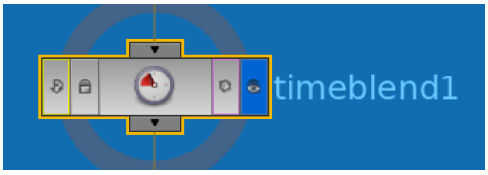


Attraverso il parametro *Geometry File* si può specificare il nome della sequenza di immagini che verrà salvata partendo dalla simulazione. Il file salvato è di tipo *.bgeo*

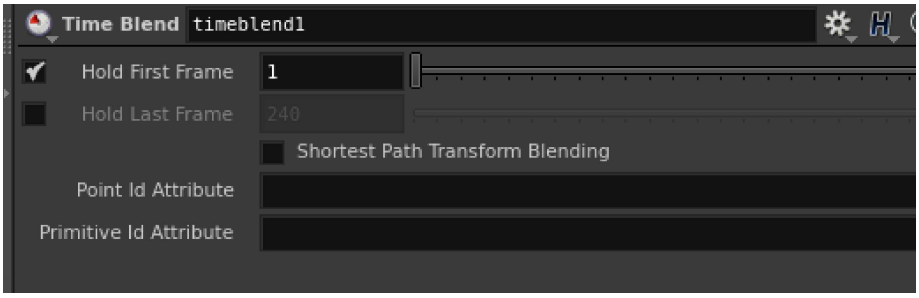
File Cache è un *Geometry Node*: scrive e legge una sequenza di geometria su disco. L'operatore *File Cache* fornisce un modo semplice per mettere in cache una fase intermedia della rete *SOP* in una sequenza di file su disco. Le impostazioni più rilevanti riferite a questo nodo sono:

1. *Load From Disk*. Quando questa opzione è selezionata il nodo deciderà se caricare la geometria salvata su disco o se invece salvarla su disco.
2. *File Mode*. Questo parametro è abilitato solo quando il parametro precedente è disabilitato e specifica se il nodo scriverà o leggerà la geometria su o da file su disco. Tra le opzioni disponibili come valori di questo parametro si trovano:
   - *Automatic*. Significa che scriverà il file se questo non esiste, mentre lo leggerà se esiste. Quindi se si vuole forzare la scrittura di un aggiornamento del file sarà necessario eliminare manualmente il file su disco.
   - *Read Files*. Legge la geometria da un file.
   - *Write Files*. Scrive la geometria in input su disco.
   - *No Operation*. Non sarà eseguito nessun accesso al file.
3. *Geometry File*. Indica il percorso nel quale caricare la geometria quando viene abilitata l'opzione *Load From Disk*.

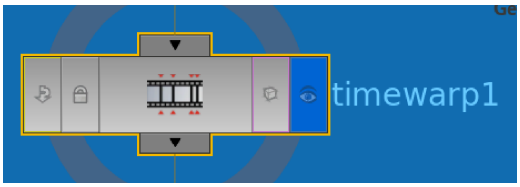
Per creare un effetto che potesse fare in modo di conferire una sensazione di pesantezza, di qualcosa di massiccio, è stato deciso di estendere la durata della simulazione. Per realizzare quindi una simulazione rallentata sono stati utilizzati determinati operatori.
Nello specifico, per eseguire questa operazione, sono stati utilizzati due operatori, il primo è un *Time Blend*, il secondo è il *Time Warp*. Il *Time Blend* è un nodo di geometria che miscela i valori *intraframe* della geometria.



Il *Time Blend* riceve sia l'inizio e che la fine del *frame* intero più vicino. I valori vengono poi miscelati secondo la posizione frazionaria all'interno del fotogramma. Questo operatore funziona solo se la topologia della geometria rimane la stessa passando da un fotogramma all'altro. Uno dei parametri più importanti del *Time Blend* è *Hold First Frame* che determina se il primo *frame* debba essere bloccato o meno.



Il Time Warp, invece, è un nodo di geometria che ricalcola l'ingresso per fornire in uscita un differente intervallo di tempo tramite un'operazione di *Mapping*.



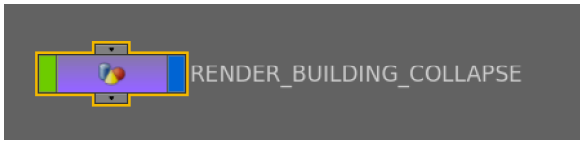
Questo operatore può essere utilizzato, ad esempio, per selezionare delle clip animate o, mediate l'opzione *Cycle*, generare un ciclo di camminata. Un'importante aspetto da tenere in considerazione, come è stato fatto in questo caso, è che se l'animazione è definita solamente su valori di *frame* interi, come ad esempio per un flusso *.bgeo* letto da un operatore *File*, può risultare necessario utilizzare prima un nodo *Time Blend* che consente di ottenere dei sotto frame interpolati. Il settaggio dei parametri riferito a questo nodo è il seguente:

1. *Input Range*. Viene indicato l'intervallo di *frame* in ingresso da modificare, il quale può anche essere frazionale.
2. *Output Range*. Qui viene settato l'intervallo di *frame* che si vuole ottenere in uscita.
3. *Pre Extend*. Usato per impostare la regola da applicare ai *frame* presenti prima dell'*Output Range*. L'opzione *Hold* blocca l'*Output Range*, *Extend* estende linearmente l'*Input Range*, e *Cycle* ripete l'*Input Range*.
4. *Post Extend*. Qui viene impostata la regola da applicare ai *frame* presenti dopo l'*Output Range*. L'opzione *Hold* blocca l'*Output Range*, *Extend* estende linearmente l'*Input Range*, and Cycle ripete l'*Input Range*.

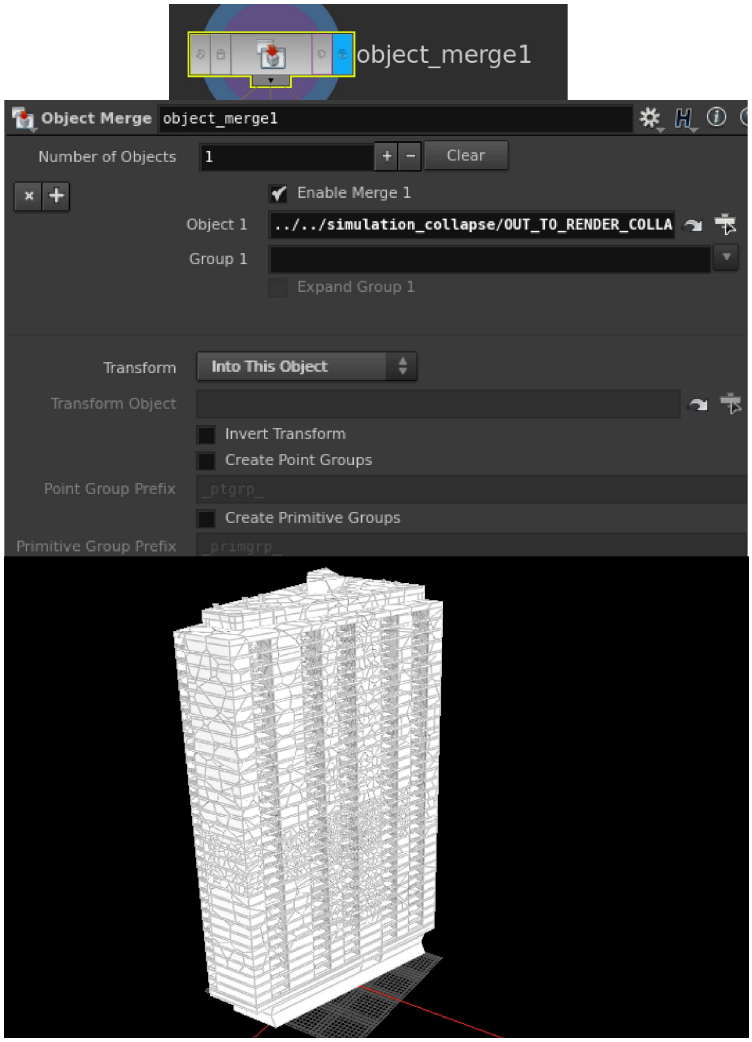
Dopo aver rallentato la simulazione è stato inserito un nodo *Null* per creare l'effettivo output di simulazione finale, utilizzato nelle successive fasi del processo di *Destruction*, dopo essere stato nuovamente memorizzato su disco per ottimizzare le tempistiche di simulazione e poi di *Rendering*. Un'operazione, a questo punto del lavoro, necessaria, data la complessità della dinamica di simulazione, in quanto se non si fosse implementata il software avrebbe dovuto eseguire tutti i calcoli per ogni *frame* ogni qual volta la simulazione fosse stata avviata, mentre così facendo il software dovrebbe solamente caricare il file *.bgeo* memorizzato su disco.

Materiali

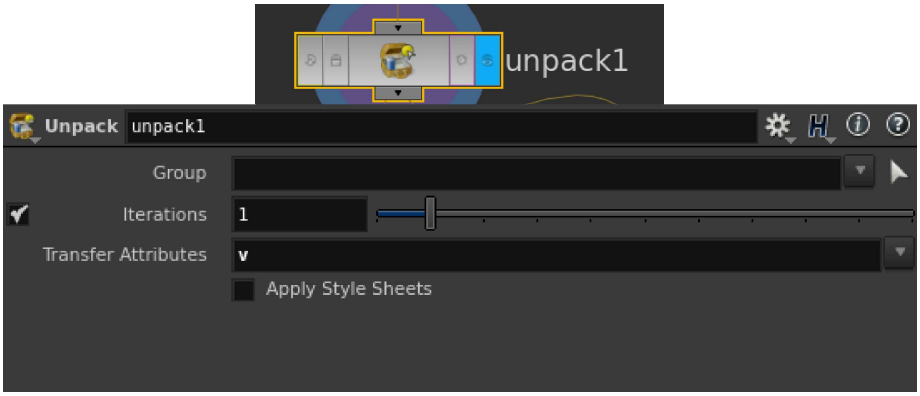
Prima di passare alla fase di *Rendering* finale sono state necessarie due ulteriori operazioni, ovvero l'applicazione dei materiali e la gestione dell'illuminazione. In particolare, per gestire il look dei render sono stati utilizzati dei nodi di geometria che al loro interno, tra le altre cose, contengono gli operatori riguardanti l'inserimento dei materiali.



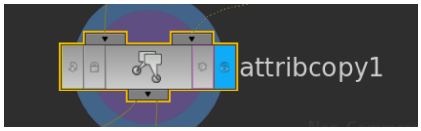
Tramite il nodo di *Object Merge* è stato possibile prelevare la geometria di simulazione memorizzata su disco e inserirla all'interno del nodo di geometria *RENDER_BUILDING_COLLAPSE,* all'interno del quale è stata effettuata la procedura di inserimento dei materiali.



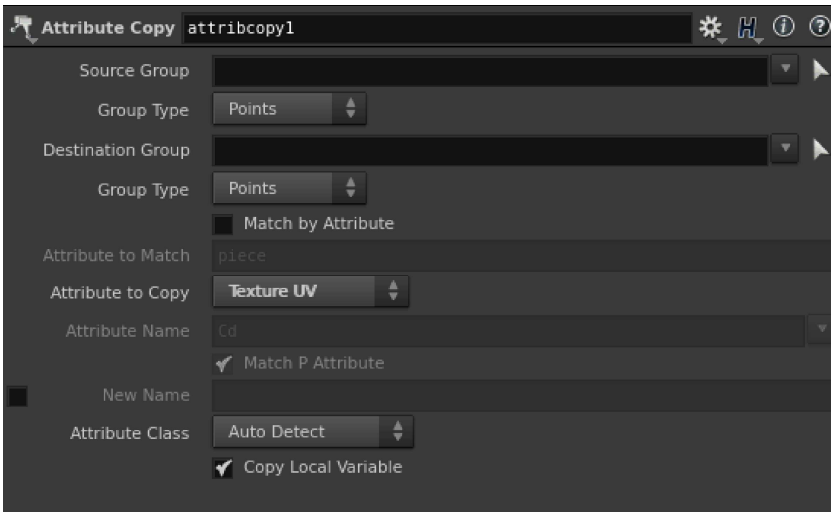
Prima dell'inserimento dei materiali, dopo il nodo *Object Merge,* è stato utilizzato l'operatore di *Unpack* per consentire di spacchettare la geometria e recuperare l'attributo *velocity*. Tutto questo per eseguire prima la proiezione delle texture sul modello 3D e poi l'aggiunta dei materiali.



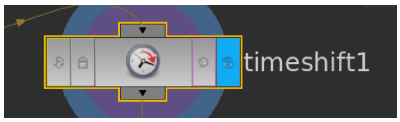
L'operatore *Unpack* è stato poi dato in ingresso al nodo di geometria *Attribute Copy*, il cui compito è quello di copiare gli attributi tra gruppi di vertici, punti o primitive, insieme al nodo di geometria *Timeshift*.



Nello specifico, tramite il nodo *Attribute Copy*, gli attributi vengono copiati da un gruppo sorgente a un gruppo destinazione. Questi due gruppi devono però essere dello stesso tipo, anche se comunque non ci sono delle restrizioni sulla classe dell'attributo. L'ordine, con il quale i gruppi sorgente e destinazione sono specificati, influenza il risultato della copia. Un esempio di utilizzo dell'operatore *Attribute Copy* è quello che consente la copia delle cordinate di texture UV, ovvero ciò che è stato fatto proprio all'interno del progetto di *Destruction*. L'opzione *Attribute to Copy* presente in questo operatore specifica l'attributo da copiare. Per convenienza, per questa voce, sono forniti gli attributi di colore e texture UV, nonostante questo tutti gli altri attributi possono essere specificati mediante un nome selezionando l'opzione *Other Attribute*.

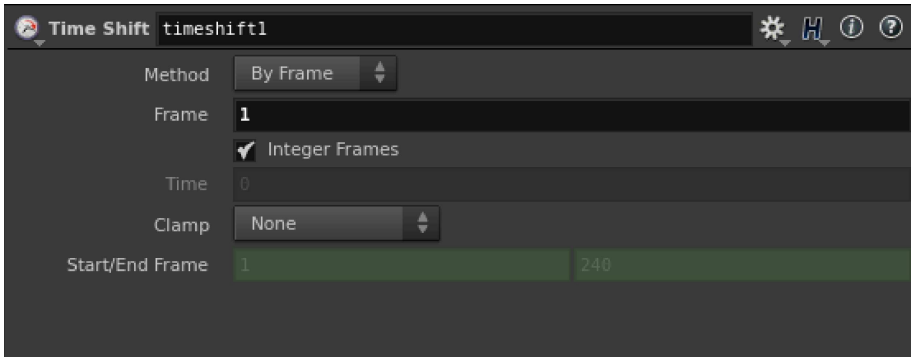


Il nodo di *Timeshift*, invece, permette ad esempio di ottenere l'accesso a frame precedenti o successivi.

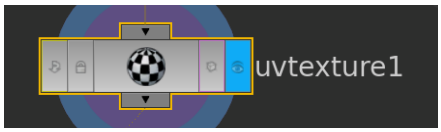


Le impostazioni più rilevanti per questo nodo sono:

1. *Method*. Determina se l'ingresso potrebbe essere fornito ad un determinato numero di frame o a un dato tempo.
2. *Frame*. Indica il numero del *frame* da fornire in ingresso.

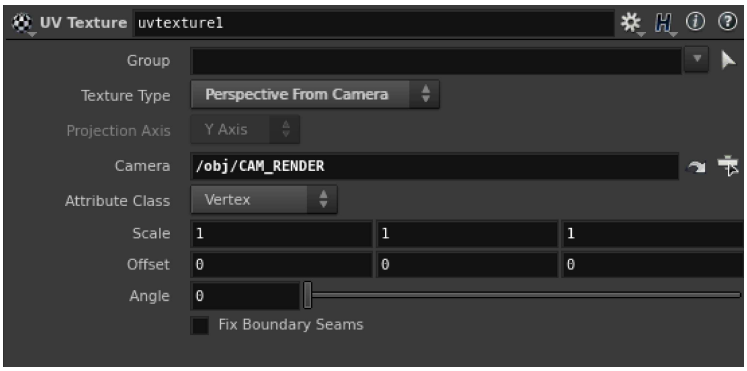


In sostanza con questo operatore vengono copiate le cordinate di texture UV, prelevate tramite il precedente nodo *Attribute Copy*, corrispondenti al primo frame di simulazione, come si può notare nell'immagine delle impostazioni del nodo *Timeshift*, per poi essere proiettate sul modello 3D del palazzo attraverso il nodo successivo *UV Texture*. L'operatore *UV Texture* è un nodo di geometria.



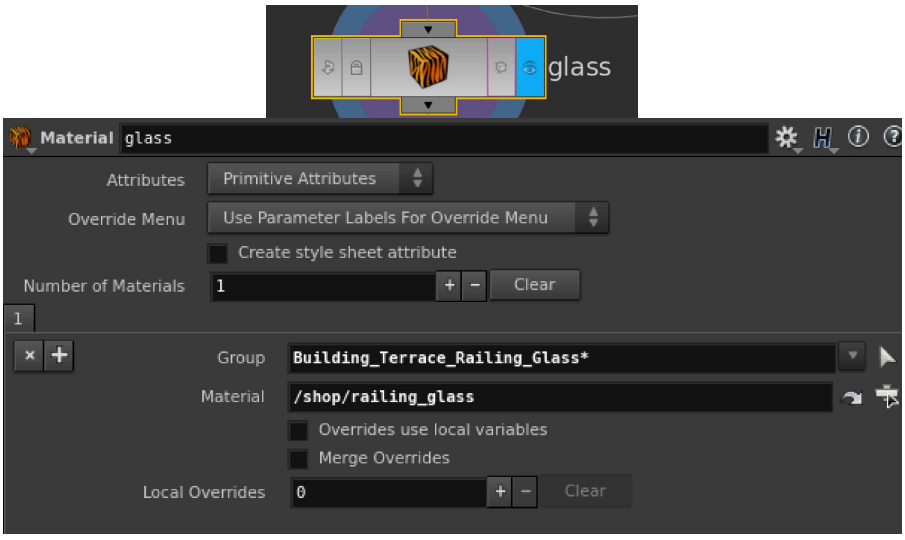
Questo operatore assegna le coordinate di texture UV alla geometria. L'effetto di questa operazione è meglio visualizzato nella viewport UV, o con le texture attive nella viewport 3D. Lo strumento *UV Texture* consente di scegliere una tipologia di texture dal menu a scomparsa presente nell'editor dei parametri.
Per l'appunto i parametri utilizzati in questo contesto sono:

1. *Texture Type*. Ovvero il tipo di proiezione che si vuole utilizzare per avviare il processo di proiezione delle texture. Tra i vari valori selezionabili è stata scelta l'opzione *Perspective From Camera* che consente di assegnare le coordinate delle texture, questo significa che lo spazio globale dell'oggetto può essere texturizzato adattandosi esattamente alla proiezione eseguita della camera.
2. *Camera*. Qui viene specificato il percorso della Camera da cui poter eseguire la proiezione delle coordinate prospettiche.



Per poter visualizzare tutto questo, e quindi la proiezione della texture sul modello 3D, è stato necessario un ulteriore passaggio eseguito attraverso il nodo di visualizzazione *UV Quick Shade*, tramite il quale è stato possibile selezionare la texture da proiettare attraverso il parametro *Texture Map*, nel quale e quindi stato possibile specificare il percorso di memorizzazione dal quale poter prelevare l'immagine da applicare come texture.

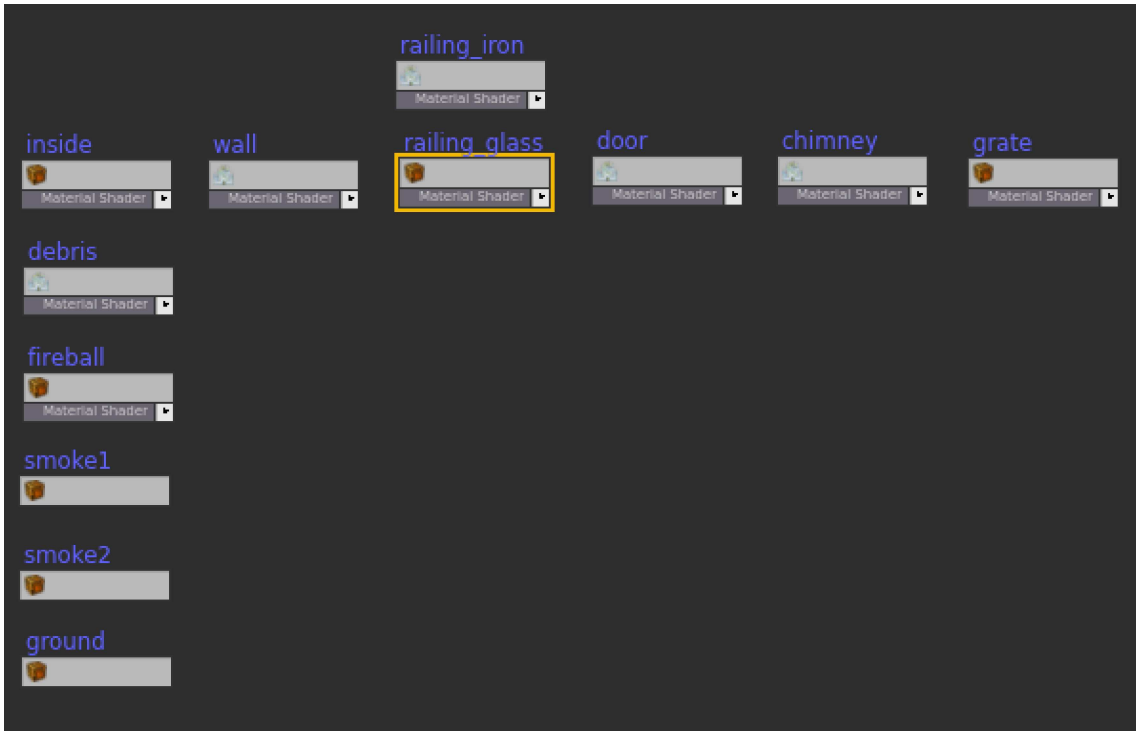
Dopo questa importante operazione di proiezione sono stati aggiunti i materiali a tutti gli elementi costituenti il modello del palazzo. Nello specifico sono stati utilizzati sette operatori di tipo *Material*.



Il nodo *Material* è un nodo geometria e ha il compito di assegnare uno o più materiali. Questo operatore quindi assegna un materiale a una lista, o gruppi, di primitive o punti. Il nodo crea gli attributi *shop_materialpath*, nel quale viene impostato il percorso dal quale prelevare il materiale, *material_override*, *material_stylesheet*. *Mantra* utilizza tutti questi attributi, ma la *viewport* supporta solamente lo *shop_materialpath*. I parametri utilizzati per questo operatore sono i seguenti:

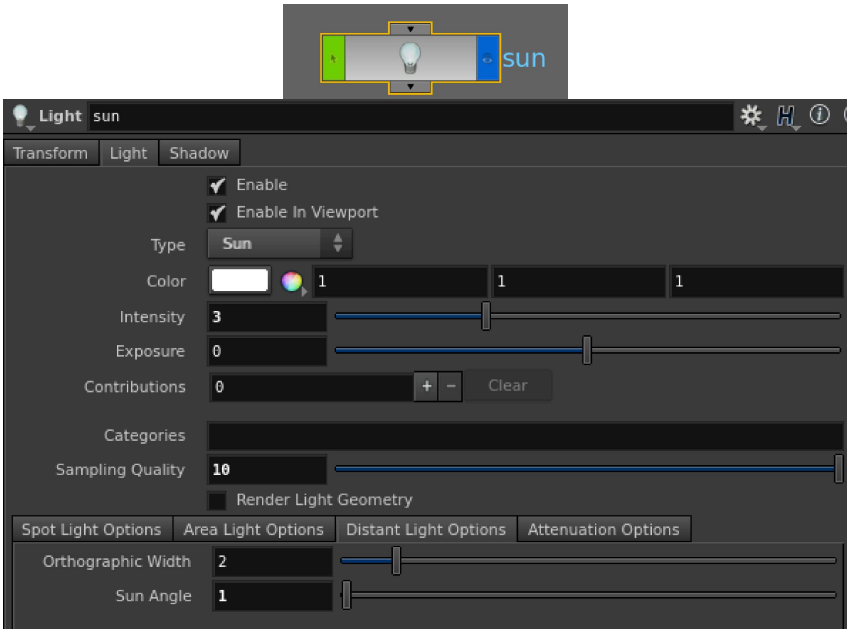
1. *Attributes*. Che specifica il livello al quale assegnare il materiale, per il quale è stata scelta la voce *Primitive Attributes* che appunto assegna il materiale al livello primitive quindi alle facce.
2. *Group*. Qui è possibile specificare la lista delle primitive, o il nome del gruppo, a cui assegnare il materiale.
3. *Material*. Questo è il parametro nel quale viene specificato il percorso del materiale da assegnare. Attraverso questa opzione è stato possibile selezionare il giusto materiale aggiunto nella sezione *Shop*.

Nella sezione *Shop* sono stati creati i vari materiali da applicare ai vari elementi del palazzo e non solo anche al fumo, ai detriti e all'esplosione.



Luci

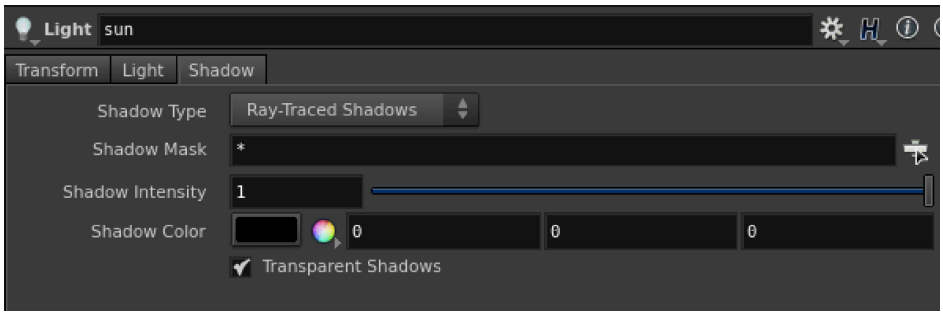
A questo punto è stata creata la corretta illuminazione al fine di ricreare la stessa atmosfera dello scatto panoramico di riferimento scelto all'inizio di tutto questo lavoro. Sono state quindi aggiunte delle luci. Nello specifico sono stati utilizzati due diverse tipologie di operatori. Il nodo di *Light* è un nodo di geometria il cui compito è emettere luce sugli altri oggetti presenti all'interno della scena.



Attraverso questo operatore è possibile controllare il colore, le ombre, l'atmosfera e la qualità del render degli oggetti che vengono illuminati da questa luce. Attraverso la sezione *Transform* la luce è stata posizionata al fine di ricreare la corretta illuminazione, quindi cercando di ottenere la stessa quantità di luce e la stessa provenienza presente nell'immagine sorgente. Nella sezione *Light*, invece, i parametri più rilevanti sono:

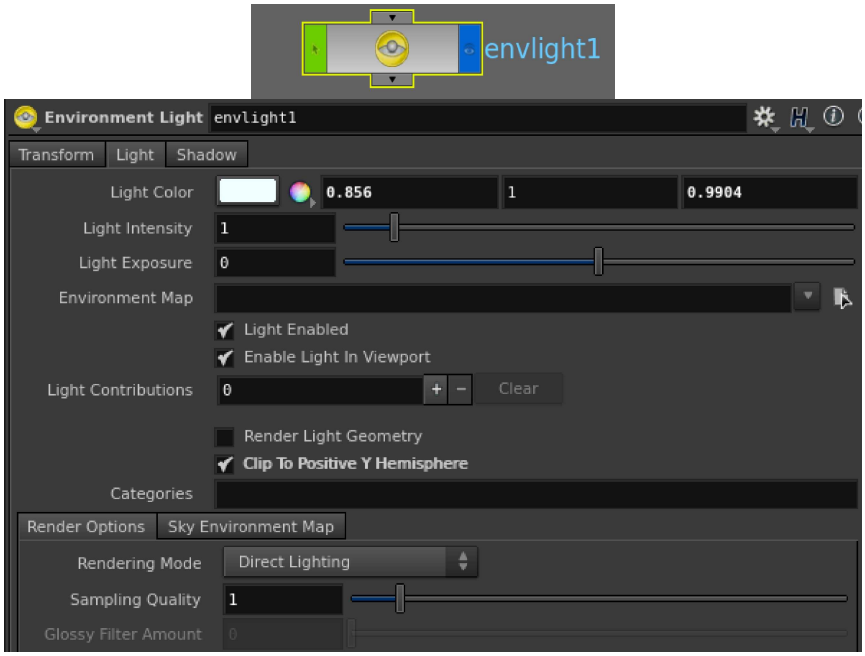
1. *Enable*. Deselezionare questa voce è equivalente a settare l'intensità a zero.
2. *Enable In Viewport*. Deselezionare questa voce significa rimuovere la luce all'interno della *viewport*.
3. *Type*. Come valore per questo parametro, che specifica il tipo di luce utilizzata, è stata scelta l'opzione *Sun*, che indica una sorgente di luce direzionale di dimensione finita infinitamente lontana dalla scena.
4. *Color*. Che indica il colore da attribuire alla sorgente di luce.
5. *Intensity*. Questo parametro specifica l'intensità lineare della sorgente luminosa. Se l'intensità è pari a zero significa che la luce è disabilitata.

Nella sezione *Shadow* i parametri più importanti sono:



1. *Shadow Type*. Per questo parametro è stata selezionata la voce *Ray-Traced Shadows*, che significa che verrà utilizzato il *Ray-tracing* per calcolare le ombre prodotte da questa sorgente luminosa.
2. *Shadow Intensity*. Che in sostanza si occupa di rendere più o meno marcate le ombre.
3. *Shadow Color*. Specifica il colore utilizzato al posto del nero per le ombre completamente opache. Questo valore viene combinato con il parametro precedente per produrre il valore finale delle ombre.

L'operatore *Environment Light*, invece, è un tipo di luce che fornisce un'illuminazione di sfondo da dare alla scena.



Con questo si completa anche la fase riguardante l'illuminazione che precede quella di *Rendering* finale.

Rendering

Introduzione

In una normale pipeline di post produzione che richiede un'ulteriore fase di rimaneggiamento delle riprese o delle immagini, come nel caso del progetto oggetto di questa tesi, il processo di *Rendering* è l'ultimo importante stadio che conferisce l'aspetto finale, o il look, alla scena, o alla sequenza visiva, o al prodotto audiovisivo, in sostanza è ciò che sarà sottoposto allo sguardo dello spettatore. Il progetto di *Destruction,* qui trattato, ha richiesto una fase molto massiccia per quanto riguarda lo stadio di elaborazione delle immagini, in quanto è stata necessaria l'integrazione di elementi ricreati digitalmente in uno spazio, o in un ambiente, fotorealistico. Il *Rendering*, quindi, è l'operazione il cui obiettivo consiste, attraverso dei calcoli, di ottenere, a partire da un file che descrive una scena 3D in ingresso, in questo caso una simulazione completa di geometria, luci, materiali e molto altro, una o più immagini.

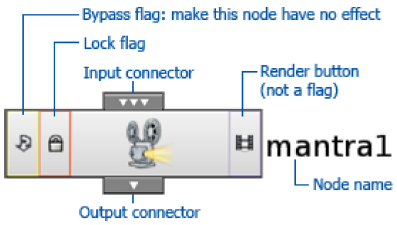
Cos'è il Rendering

Il *Rendering* è quindi il processo di generazione di un'immagine, o una serie di immagini, a partire da una descrizione matematica di una scena tridimensionale, la quale viene interpretata da algoritmi che definiscono il colore per ogni punto dell'immagine digitale che sarà generata. Il motore di *Rendering* è ciò che consente di avviare il processo di *Rendering*. Un motore di *Rendering* consiste, in sostanza, in una combinazione di metodi che sfruttano le proprietà dell'ottica, della percezione visiva, della matematica e dell'ingegneria per ottenere una conversione il cui risultato è un'immagine. Un grosso fattore limitante in tutto ciò è la lunghezza di tale processo, in quanto l'operazione di *Rendering* è un tipo di elaborazione che richiede, spesso, molto tempo per poter essere ultimata: è un'operazione complessa, molto onerosa a livello computazionale. Nel caso di una pipeline utilizzata per la creazione dei *VFX*, come ad esempio per quella di *Destruction*, in linea teorica non dovrebbero sussistere dei limiti di tempo, in quanto l'obiettivo non è ottenere un *Rendering* in tempo reale ma raggiungere un'elevata qualità visiva che tocca uno sbalorditivo grado di realismo; nella pratica, invece, si cercano comunque delle soluzioni che riducano il tempo di *Rendering* al fine di abbattere i costi di produzione. Per quanto riguarda il *Rendering* in *Houdini,* esso richiede, in generale, una camera che definisce la vista da cui osservare la scena e dalla quale avviare il processo, delle luci che vengono utilizzate per illuminare la scena, e un nodo di render che rappresenta il motore di *Rendering* utilizzato. Con *Houdini* è anche possibile eseguire un render senza tutto questo, quindi senza l'utilizzo di una camera, di luci o nodi di render. Come impostazione predefinita *Houdini* utilizza *Mantra* per gestire il processo di *Rendering*: un motore di *Rendering* altamente avanzato.

Mantra

Mantra è il motore e il nodo di render utilizzato per gestire l'importante fase di *Rendering* in *Houdini* del progetto oggetto di questa trattazione. Il processo di *Rendering* della scena, con *Mantra,* consente la generazione di immagini nel formato di esportazione standard *IFD*. Il file *IFD* è, appunto, il formato di descrizione della scena prodotto da *Houdini* e utilizzato da *Mantra* per la realizzazione di immagini o sequenze animate renderizzate. *Mantra* è altamente efficiente nel *Rendering* di *packed primitives* e *volumi*, che sono, tra l'altro, gli oggetti utilizzati all'interno di questo progetto per gestire la fase di simulazione che, in questo frangente, sarebbe stata sottoposta all'operazione di *Rendering*. *Houdini* permette, inoltre, l'integrazione di motori di render di terze parti, come ad esempio *RenderMan*, *Arnold*, *Mental Ray*, e *Maxwell*, e l'implementazione di *Shader* sfruttando reti *VOP* oppure codice in linguaggio *VEX*. *Mantra* dispone di diverse tipologie di motori di render, i quali utilizzano, a loro volta, differenti algoritmi per la produzione di immagini. In generale è sempre consigliato il *Physically Based Rendering* a meno che non si abbia l'esigenza di utilizzare un motore di render differente. Di seguito è mostrata un'immagine che illustra i pulsanti che compongono il nodo di render *Mantra,* che nel dettaglio sono:

1- *Bypass*.Il nodo viene sorpassato.
2- *Lock*. Il nodo viene disabilitato.
3- *Render*. Attivando questo pulsante il nodo risulta attivo e quindi pronto per avviare il processo di *Rendering*.



Prima di procedere con la spiegazione di ciò che è stato fatto in questa fase, all'interno del progetto oggetto di questo studio, è necessario, aprire una parentesi a proposito del lavoro che *Mantra* svolge, per comprendere al meglio, anche, l'insieme dei parametri da impostare all'interno del nodo di render per configurare in modo adeguato il *Rendering*, e capire come tutto questo influenzi le prestazioni e la qualità del motore di render che viene utilizzato. Come è stato già accennato, *Mantra* consente di scegliere il motore di render che si vuole utilizzare per il *Rendering* della scena. Nello specifico le possibili scelte che *Mantra* mette a disposizione per il parametro *Rendering Engine* sono:

1- *Micropolygon Rendering.*
2- *Ray Tracing.*
3- *Micropolygon PBR.*
4- *Physically Based Rendering (PBR).*
5- *Photon Map Generation.*

Il *Physically Based Rendering* e il *Raytracing (RT)* producono risultati molto simili a livello di immagini, in quanto si basano entrambi su una logica, che a livello di shader, è per la maggior parte la stessa e di conseguenza producono uno *Shading* molto simile nel colore finale dei pixel. *Mantra,* però, ha essenzialmente due modalità di funzionamento:

1- *Physically Based Rendering* (*PBR*).
2- *Micropolygon Rendering*.

In generale, all'interno della pipeline di *Rendering*, *Mantra* divide l'immagine in tessere e avvia il processo di *Rendering* per ciascuna di esse singolarmente. Se l'operazione di *Rendering* viene eseguita su tessere di piccole dimensioni si potrebbe avere una riduzione del livello complessivo delle prestazioni. Una particolarità del processo di *Rendering,* che avviene o in *MPlay* o nella *Render View*, è la possibilità di indicare, tramite puntatore, le tessere che *Mantra* dovrà renderizzare per prima, in modo da poter visualizzare un'anteprima delle aree di interesse più velocemente. Dopo questa fase di suddivisione in tessere dell'immagine*, Mantra* carica la geometria, che viene normalmente salvata in un file *IFD* contenuto nella *RAM*. Prima di approfondire la differenza tra il *Micropolygon* e il *PBR Renderign* è utile anche capire quali tipi di shader *Mantra* mette a disposizione per avviare il processo di *Shading*.

Shader

Gli shader sono dei componenti che vengono eseguiti per specificare i vari aspetti che accompagnano il processo di *Rendering,* come ad esempio il colore delle superfici. Per *Mantra,* gli shader possono essere scritti in linguaggio *VEX* o creati attraverso i nodi *VOP,* i quali vengono poi convertiti automaticamente in codice *VEX* da *Houdini*. *Mantra* gestisce nello specifico cinque tipi di shader, tra questi, i più importanti sono:

1- *Displacement*. Questi shader vengono eseguiti per primi e modificano i valori delle variabili globali *P* (posizione del vertice) e *N* (vertice della normale). Un parametro importante del *Displacement Shading* è *Displacement Bouding*. Il *Displacement* è un processo che cambia la forma della geometria e quindi può rischiare di spingerla fuori dal suo *Bouding Box* originale. Per evitare questo problema, quindi, il parametro *Displacement Bouding* riferisce a *Mantra* la distanza massima che si può accettare dal *Displacement,* in unità di *Houdini*, per spostare la geometria. *Mantra* utilizza questo valore per aumentare la quantità di geometria da tenere in memoria, è necessario quindi impostare questo valore al suo massimo senza superarlo. Infatti se questo valore è troppo basso la geometria sottoposta al *Displacement* viene tagliata, se invece questo valore è troppo alto viene riscontrato un aumento della complessità.

2- *Surface*. Gli shader di superficie vengono eseguiti dopo quelli di *Displacement*. Possono causare l'esecuzione dei *Light* e degli *Shadow Shader* se chiamano un ciclo di illuminamento.
3- *Fog*. I *Fog Shader* vengono eseguiti dopo gli shader di superficie. Questi shader possono modificare il valore finale *Cf* e *Of* di ogni superficie. È possibile aggiungere un *Fog Shader* alla scena creando un oggetto atmosfera a livello oggetto e specificando lo shader nei parametri dell'oggetto atmosfera. In generale, però, i volumi renderizzati sono un modo molto più preciso e flessibile per simulare effetti atmosferici rispetto all'utilizzo dei *Fog Shader*.

Dopo l'esecuzione degli shader, *Mantra* riassume i contributi dei campioni riferiti a ogni pixel per ottenere il colore finale di questo. In questo contesto esiste inoltre un filtro che controlla quanto ogni campione contribuisce al valore finale del pixel. L'impostazione della larghezza di questo filtro controlla il raggio entro il quale sono considerati i campioni che contribuiscono a dare il colore al pixel. Il valore predefinito che regola questo parametro è un filtro *Gaussiano* 2×2.

Micropolygon Rendering

Il *Micropolygon* è un algoritmo che è stato molto utilizzato in passato perché forniva un buon compromesso prestazionale, è stato successivamente soppiantato, nelle moderne configurazioni di *Rendering*, dal *Raytracing*. In particolare, il *Micropolygon*, è stato progettato per rendere più efficiente l'utilizzo della memoria, in quanto la geometria viene tagliata a dadini, sottoposta all'operazione di *Shading* una volta sola e scartata quando risulta non più necessaria. Oggi, però, con l'arrivo di modelli che consentono di eseguire calcoli poligonali molto più complessi e di macchine con grande disponibilità di memoria, sono preferibili, in quanto più efficienti, algoritmi come il *Raytracing* o il *PBR*.
Uno dei motivi che potrebbe spingere ad utilizzare ancora il *Micropolygon*, è il fatto che esso consente di ottenere un *Rendering* efficiente delle scene in cui viene coinvolto un sacco di pelo e/o fumo, a patto che non vengano utilizzate delle impostazioni *Raytracing* nello *Shader*. Un'altra ragione per cui potrebbe essere meglio utilizzare il Micropolygon, in quanto più veloce e con un efficiente motion blur, si verifica quando dobbiamo renderizzare una scena priva di riflessioni, rifrazioni, o ombre.

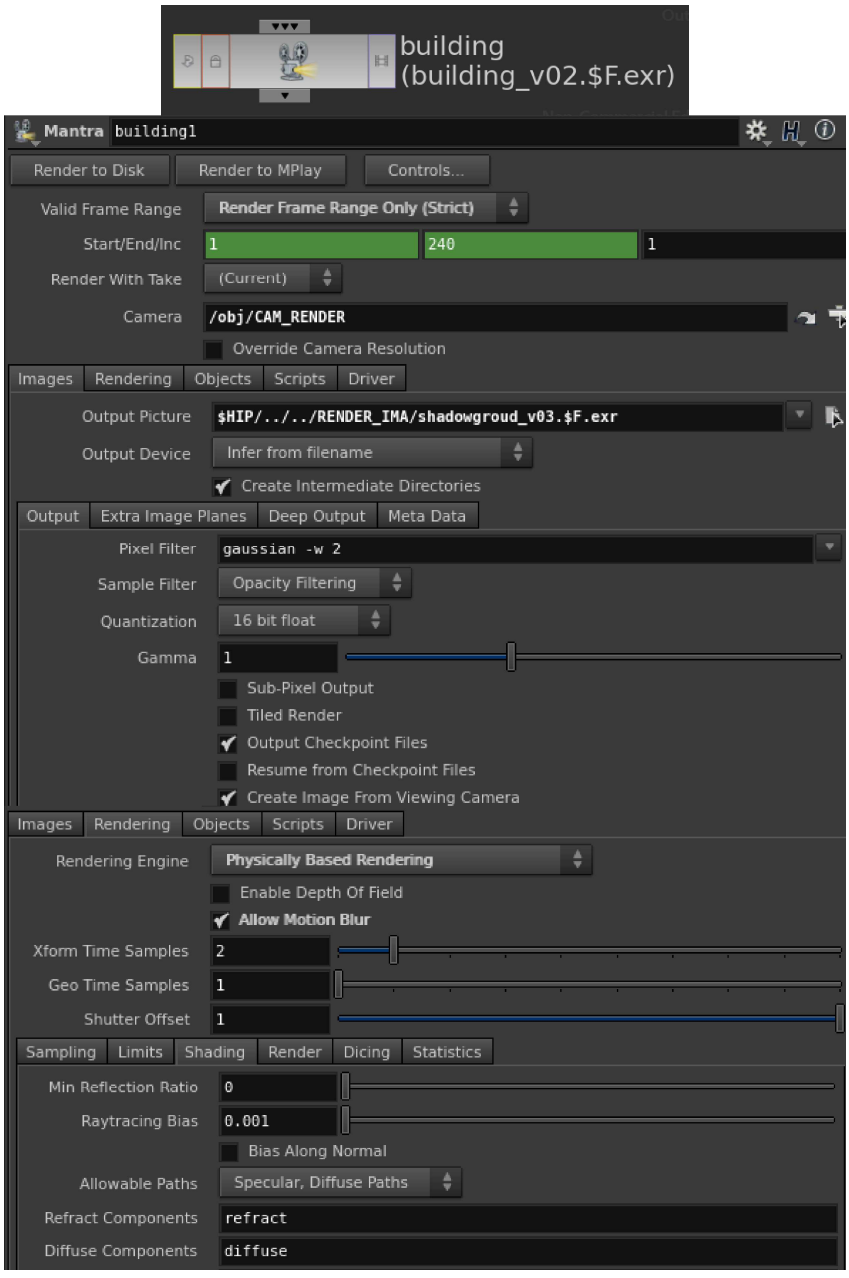
Physically Based Rendering

Il *Physically Based Rendering (PBR)* è un algoritmo che viene utilizzato per il *Rendering* di differenti tipologie di scene, il processo di *Shading* in esso presente consente un'accurata simulazione fisica di luci, ombre, sfumature, riflessi e molto altro in modo semplice senza il bisogno di soluzioni alternative o di scrivere shader complessi. Il *PBR* è un motore di *Rendering* molto più realistico rispetto ad altri, però ha un aspetto negativo, ovvero si basa su un campionamento dell'immagine casuale e quindi tende a creare delle immagini con più rumore rispetto ad altri motori di render. Il campionamento è un processo che influenza in modo massiccio l'algoritmo di *Rendering*, aumentare il numero di campioni per pixel può essere un metodo per incrementare la fedeltà dell'immagine ma che ha lo svantaggio di prolungare il tempo di *Rendering*.
Per quanto riguarda il processo di *Shading* è possibile scrivere per il *PBR* degli shader di superficie i quali comportano la progettazione di una funzione *BSDF (Bidirectional Scattering Distribution Function)*. Questa funzione descrive il modo con cui la luce viene diffusa da una superficie; questo fenomeno è generalmente diviso in componenti riflesse e trasmesse, che vengono poi trattate separatamente attraverso le funzioni *BRDF (Bidirectional Reflectance Distribution Function)* e *BTDF (Bidirectional Transmittance Distribution Function)*.

$$BSDF = BRDF + BTDF$$

Il concetto che sta alla base di tutte le funzioni BxDF potrebbe essere descritto come una scatola nera la cui uscita indica il valore che definisce il rapporto tra l'energia luminosa in ingresso e quella in uscita per una data coppia di angoli, uno per i raggi (incidente) in entrata e l'altro per i raggi di uscita (riflessa o trasmessa) rispetto a un dato punto della superficie. Il contenuto di questa scatola può essere una formula matematica che cerca più o meno accuratamente di modellare e approssimare il comportamento della superficie oppure, un algoritmo che produce un'uscita in base al valore dei campioni discreti misurati.
In *Houdini* il valore calcolato dalla *BSDF* è connesso alla variabile globale *F,* un tipo di dato *VEX* introdotto nella versione 9 di *Houdini*, che a sua volta è connessa allo shader. In sostanza il *PBR* non esegue un *Rendering* tradizionale, come può essere quello eseguito dal *Raytracing,* ovvero ignora i calcoli relativi alle variabili globali *Cf* (colore della superficie), *Af* (la trasparenza alpha del pixel) e *Of* (opacità della superficie) e valuta solamente la variabile *F(BSDF)*. Nello *Shading* eseguito dal *PBR*, quindi, lo shader è il responsabile del calcolo della *BSDF*, il quale imposta *F* per un valore *BSDF*.

La fase di *Rendering* in *Houdini*, è stata gestita mediante l'utilizzo degli operatori *Mantra*. Ogni simulazione è stata associata a un nodo di render differente in modo tale da poter gestire i diversi processi separatamente. Nello specifico, sono stati creati cinque nodi: uno per il *Rendering* della simulazione di *Destruction*, uno per il *Rendering* dello smoke, uno per il *Rendering* dell'esplosione, uno per il *Rendering* dei detriti e uno per il *Rendering* delle ombre. Ciascuno di questi operatori è stato configurato secondo delle impostazioni abbastanza simili tra loro, l'unica sostanziale differenza riguarda solamente i parametri che gestiscono la visualizzazione, ovvero quali elementi o oggetti abilitare all'interno della scena per poter essere renderizzati. Per quanto concerne il render della simulazione di *Destruction*, vengono riportati qui di seguito i parametri più importanti che ne costituiscono la configurazione.



In alto si trovano due modi possibili per avviare un render:

1- *Render To Disk*. Salva i render delle immagini nel percorso specificato in *Output Picture* utilizzando le impostazioni di controllo.
2- *Render To MPlay*. Sfrutta il render tramite *MPlay*, invece che specificare un determinato percorso, utilizzando le impostazioni di controllo. *MPlay* non è altro che un visualizzatore di immagini e animazioni.

Il parametro *Render Control* permette, quindi, di modificare le impostazioni del render prima di eseguirlo. Gli altri parametri importanti presenti, come i precedenti, nella sezione generale, sono i seguenti:

1- *Valid Frame Range*. Per questo parametro è stata scelta l'opzione *Render Frame Range (strict)* che in sostanza avvia il *Rendering* iniziando e terminando il processo in base ai fotogrammi di inizio e di fine specificati.
2- *Start/End/Inc*. Qui viene indicato l'intervallo di frame da renderizzare, vengono specificati i fotogrammi di inizio, di fine e l'incremento. Il suddetto range è inclusivo. Nello specifico, per il *Rendering* del palazzo, come poi per tutti i render successivi, è stato scelto un intervallo che parte dal frame 1 e termina al 240 con un incremento di 1, vengono quindi realizzati 240 frame totali.
3- *Camera*. Con questo parametro viene specificato il percorso riferito all'oggetto camera, usato per renderizzare la scena, già utilizzato durante il passo di esportazione in *Blender*.

Nella sezione *Images*, invece, il parametro *Output Picture* viene specificato per indicare il percorso nel quale salvare il file su disco. Anche se è stato utilizzato *MPlay*, il percorso in cui è stato salvato il render dopo l'esecuzione tramite *MPlay* è lo stesso di quello specificato da questo parametro, il quale viene sfruttato quando il render è avviato tramite l'opzione *Render To Disk*. Come formato di esportazione è stato scelto il *.exr*. *OpenEXR* è un formato di file immagine di alta gamma dinamica *(HDR)* sviluppato dall'*Industrial Light & Magic* per l'uso in applicazioni di *Imaging Computer*. *OpenEXR* è utilizzato dalla *ILM* in tutti i film attualmente in produzione. I primi film a impiegare il formato *OpenEXR*, prodotti dalla *ILM*, sono stati *Harry Potter e la pietra filosofale*, *Men in Black II*, *Gangs of New York*, e *Signs*. Da allora, *OpenEXR* è diventato il formato di file immagine principale utilizzato presso l'*Industrial Light & Magic*. Le caratteristiche principali incluse nel formato *OpenEXR* sono:

1- Gamma dinamica superiore ed elevata precisione del colore.
2- Include molti algoritmi di compressione delle immagini sia lossless che lossy. Tutti i codec lossy presenti in *OpenEXR* sono stati progettati per garantire grande qualità visiva ed elevate prestazioni di decodifica.
3- *OpenEXR* è un formato di file estensibile, ovvero i nuovi codec di compressione e i vari tipi di formati immagine possono essere facilmente aggiunti estendendo le classi *C++* incluse nelle librerie di distribuzione del software *OpenEXR*.

Nella sezione *Rendering,* i parametri che si sono rivelati fondamentali per settare le giuste configurazioni del render, sono i seguenti:

1- *Rendering Engine*. Per questo parametro è stato scelto il *Physically Based Rendering*.
2- *Allow Motion Blur*. *Mantra*, attivando questo parametro, renderizzerà l'immagine utilizzando il motion blur.

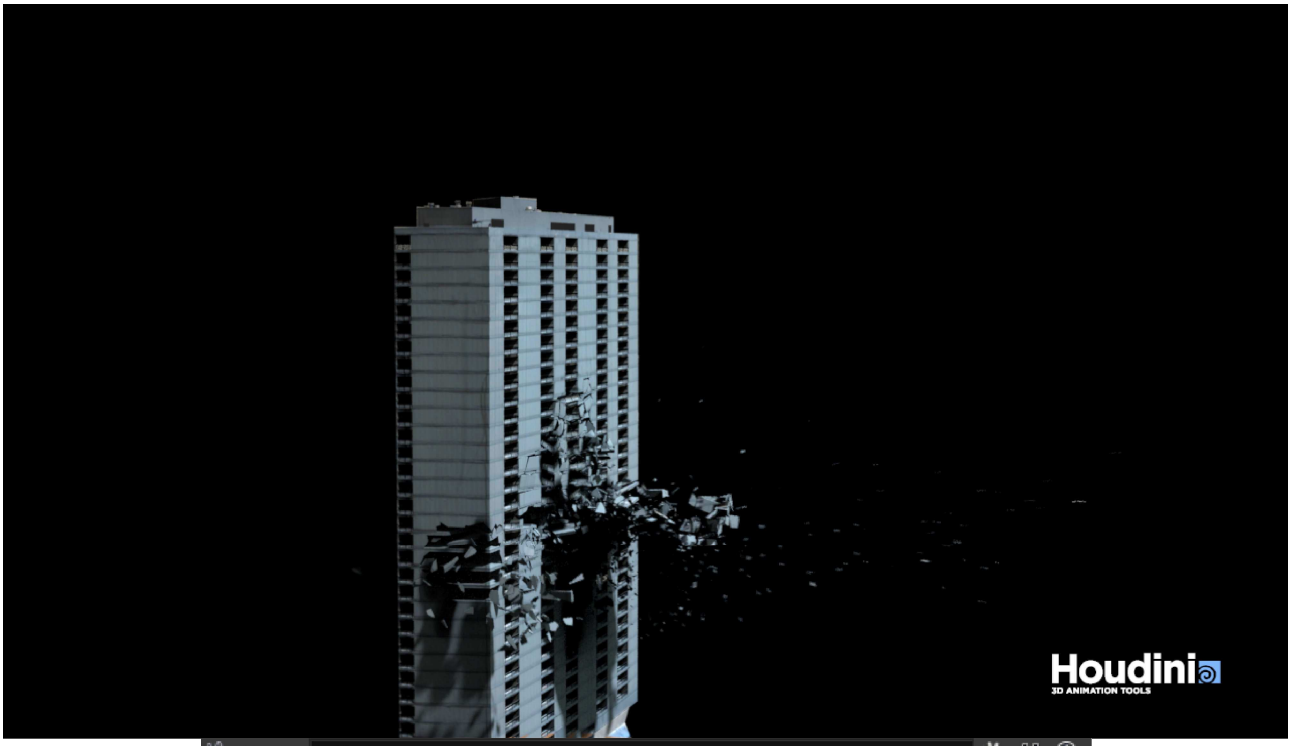
Nella sezione *Objects* sono presenti i parametri che determinano quali oggetti e quali luci vengono inclusi all'interno del file *IFD* prodotto dal render. *Mantra* nel dettaglio processa questi parametri nell'ordine seguente:

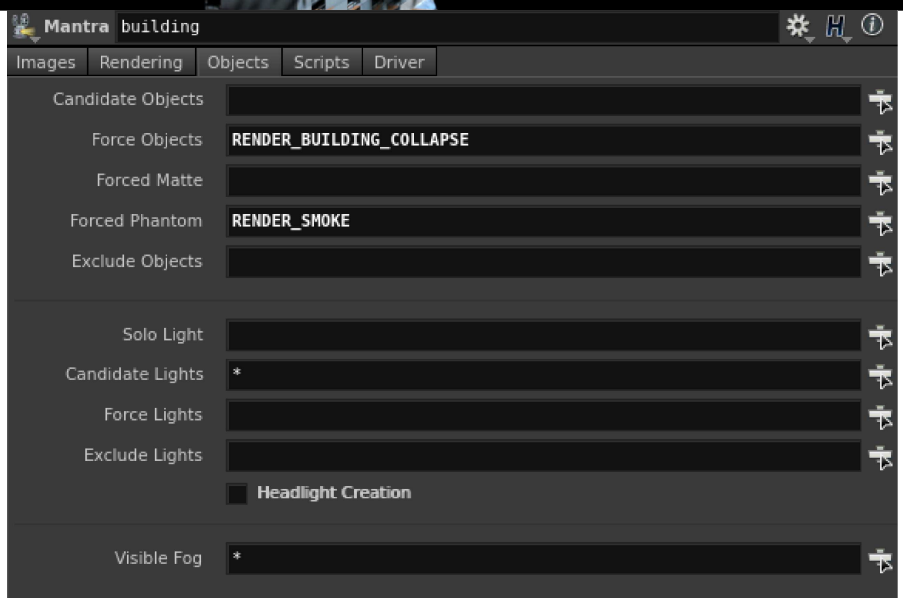
1- Sono selezionati i *Candidate objects/lights*.
2- Sono aggiunti i *Forced objects/lights*.
3- Gli *Objects/Lights* che fanno match con il parametro *Exclusion* sono rimossi.

La lista completa e il dettaglio dei parametri presenti nella sezione *Objects* sono riportati di seguito:

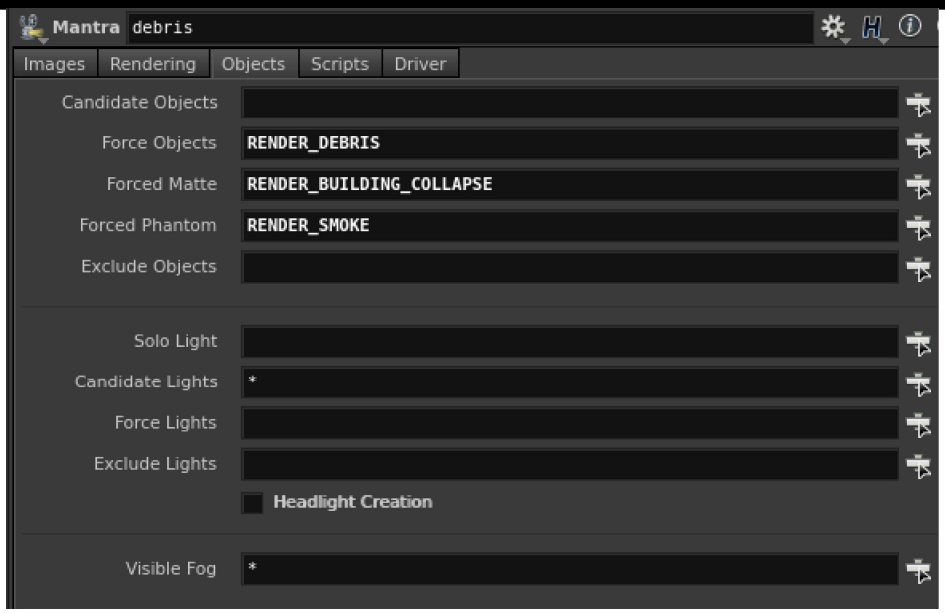
1- *Candidate Objects*. Gli oggetti presenti in questo parametro vengono inclusi nel file *IFD* solo se vengono abilitati e resi visibili.
2- *Force Objects*. Gli oggetti in questo parametro vengono aggiunti al file *IFD* indipendentemente dal loro stato, quindi indipendentemente dal fatto che vengano abilitati e resi visibili.
3- *Force Matte*. Gli oggetti vengono forzati ad essere degli *Matte Objects*.
4- *Forced Phantom*. Gli oggetti vengono forzati ad essere dei *Phantom Objects*.
5- *Exclude Objects*. Gli oggetti in questo parametro vengono esclusi dalla scena, indipendentemente dal fatto che vengano selezionati in *Candidate Objects* o *Force Objects*.
6- *Solo Light*. Solo le luci riportante in questo parametro vengono incluse all'interno del file IFD. Questo include la generazione di *Shadow Map* e dell'illuminazione. Se questo viene impostato, i parametri Candidate, Forced e Exclusion vengono ignorati. Usando questo parametro congiuntamente alle proprietà della *render_viewcamera* viene fornito un modo veloce per generare delle *Shadow Map* per le luci selezionate.
7- *Candidate Lights.* Ogni luce specificata in questo parametro viene aggiunta al file *IFD* solo se il canale dell'oscuratore della luce non è pari a zero. Infatti la luce standard imposta il canale oscuratore a zero quando essa non viene abilitata.
8- *Force Lights*. Le luci in questo parametro vengono aggiunte al file *IFD* indipendentemente dal valore riportato nei loro canali oscuratori.
9- *Exclude Lights.* Le luci qui specificate vengono escluse dalla scena, anche se sono presenti nei parametri *Candidate Lights* o *Force Lights*.
10- *Headlight Creation*. Se non sono presenti luci all'interno della scena viene creato un headlight, o proiettore, di default. Per disabilitare questa impostazione basta deselezionare il checkbox di attivazione.
11- *Visible Fog*. Gli oggetti nebbia o atmosfera presenti in questo parametro vengono inclusi nel file *IFD* solo se vengono abilitati e resi visibili.

Le principali differenze fra i render delle varie simulazioni sono riportate nella sezione *Object,* nella quale, per ogni simulazione, vengono appunto selezionati gli elementi presenti all'interno della scena da visualizzare. Di seguito viene riportata una carrellata di immagini riferita alle diverse impostazioni della sezione *Object* per il *Rendering* di ciascuna simulazione accompagnata da degli scatti tratti dai render corrispettivi.
Nella sezione *Objects* del render della simulazione di *Destruction* le impostazioni sono visualizzabili qui di seguito.

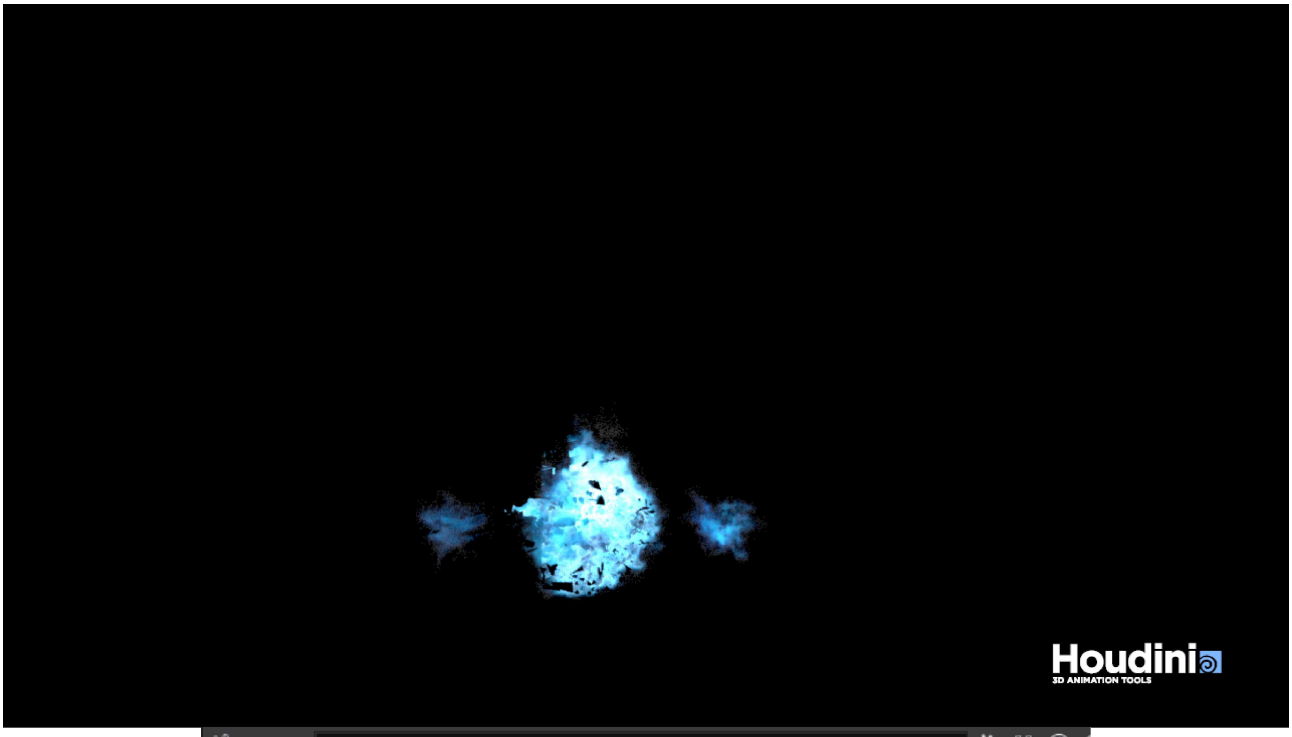


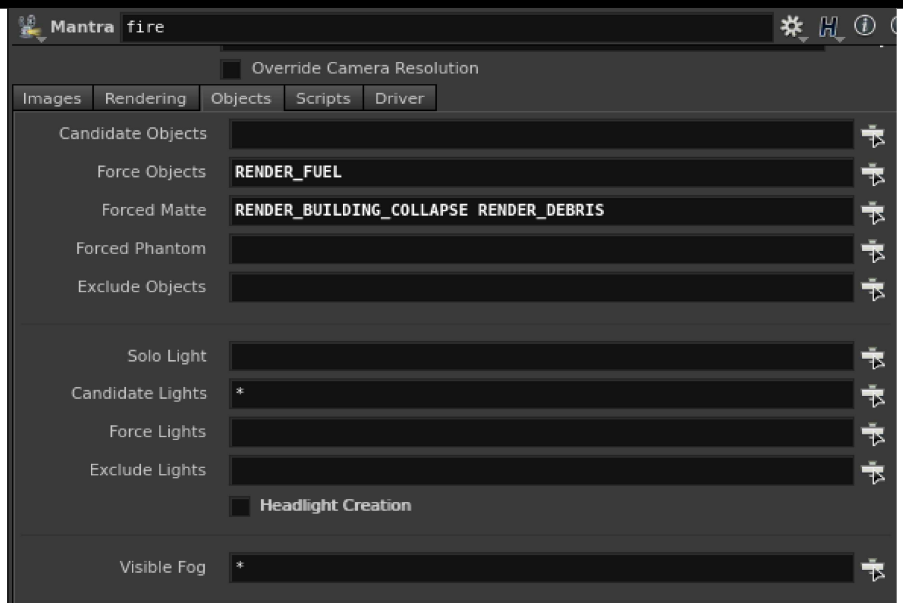


Le impostazioni mostrano che il render includerà la simulazione di *Destruction* e quella del fumo, quest'ultima però sotto forma di *"fantasma"*, ovvero se ne percepisce la presenza, perché interagisce con quella del palazzo, ma non è concretamente visibile. Per quanto riguarda le luci, sono state incluse tutte quelle abilitate a livello oggetto.
A proposito della simulazione dei detriti, come si può notare nelle immagini successive, sono state selezionate la simulazione del fumo, quella di *Destruction* in modalità *Matte*, quella del fumo in modalità *Phantom* e tutte le luci abilitate a livello oggetto.

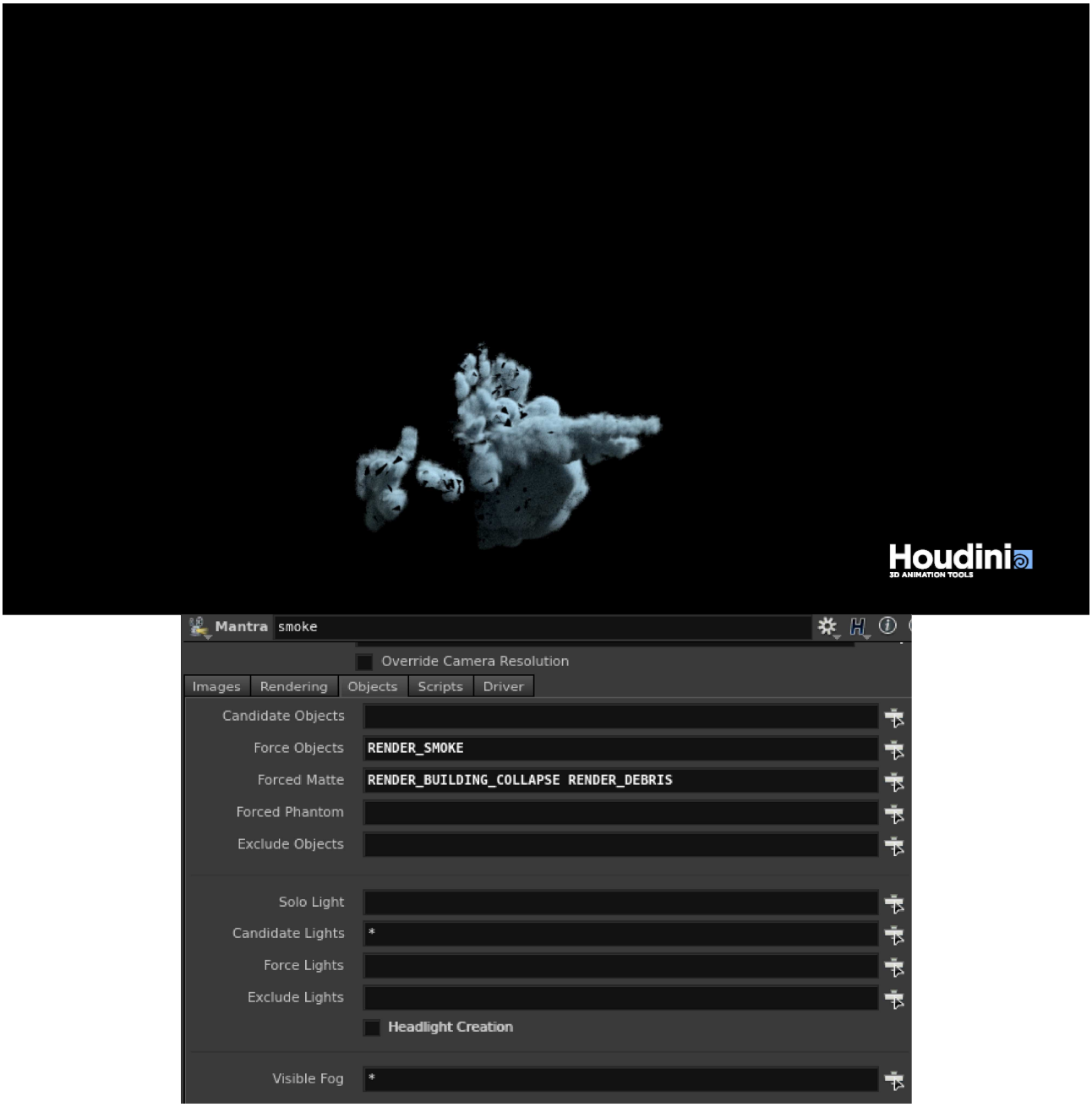


Le impostazioni del render riferito alla simulazione dell'esplosione indicano che verranno visualizzate: la simulazione dell'esplosione, quella di *Destruction* e dei detriti in modalità *Matte* e le tutte le luci abilitate a livello oggetto.





Per quanto riguarda il render riferito alla simulazione del fumo, le sue impostazioni mostrano che verranno renderizzate: il fumo, il palazzo e i detriti in modalità *Matte* e tutte le luci presenti nella scena abilitate a livello oggetto.



Per il *Rendering* di tutte queste simulazioni, in generale, non si è riscontrato alcun problema, è solamente stato un processo lungo che ha richiesto circa una settimana per poter essere ultimato.

Compositing

Introduzione

I grandi studi di produzione, i vari team creativi e anche i singoli artisti impiegati nella realizzazione dei *VFX* e non solo, sono sempre più alla ricerca di esperienze visive innovative e di elevata qualità, al passo con le esigenze che il mercato oggi richiede. Il *Compositing* costituisce solo una piccola parte in questo enorme processo e contribuisce a rendere i prodotti audiovisivi incredibili e sempre più realistici. Oggigiorno, il *Compositing,* è diventato una parte fondamentale in tutte le pipeline di produzione implementate in quasi tutti i settori coinvolti con la creazione di contenuti audiovisivi e non solo per la realizzazione di *VFX* e quindi, nel nostro caso, in questa trattazione, per i *Destruction*.

Cos'è il Compositing

Il *Compositing* si può definire come un processo creativo, un'operazione che consente di assemblare e combinare insieme elementi provenienti da sorgenti anche molto differenti fra loro. Nel caso del progetto di *Destruction* oggetto di questo testo, si tratta della combinazione di elementi visivi, ricreati digitalmente, integrati in un ambiente reale, rappresentato dallo scatto panoramico di riferimento scelto all'inizio di tutto questo processo. Attraverso, quindi, l'operazione di *Compositing,* si cerca di fornire a all'osservatore, attraverso una sequenza di immagini in movimento e fisse, in quanto l'immagine di sfondo è statica, in movimento perché il palazzo che crolla e tutti gli altri elementi come il fumo, l'esplosione e i detriti sono stati ottenuti attraverso il *Rendering* di simulazioni dinamiche, l'illusione che quello che vede è reale, vero, plausibile e perfettamente coerente.
Dal punto di vista prettamente più tecnico, il *Compositing* può essere di due tipi: basato su livelli o basato su nodi. Il *Compositing* basato sui nodi, mostra un flusso di lavoro avente una struttura ad albero, in cui tutti gli elementi, ovvero gli effetti e gli oggetti multimediali, sono legati insieme in una mappa procedurale che segue un processo che parte da un inizio grezzo per arrivare a ottenere un prodotto finito elaborato, come appunto si vedrà in seguito, quando verrà affrontata, nel dettaglio, la parte dedicata al *Compositing*, gestita, all'interno di questo progetto di *Destruction*, attraverso il software *Nuke*. Il *Compositing* basato sui livelli, invece, è costituito da un flusso di lavoro, nel quale ogni oggetto multimediale è rappresentato da un livello, inserito all'interno di una timeline o linea temporale. Il *Compositing* basato su livelli è limitato quando si ha a che fare con dei flussi di lavoro molto complessi, in particolare quando vengono coinvolti degli elementi 3D, per i quali la soluzione basata su nodi è migliore o comunque in grado di gestire il lavoro molto più facilmente.

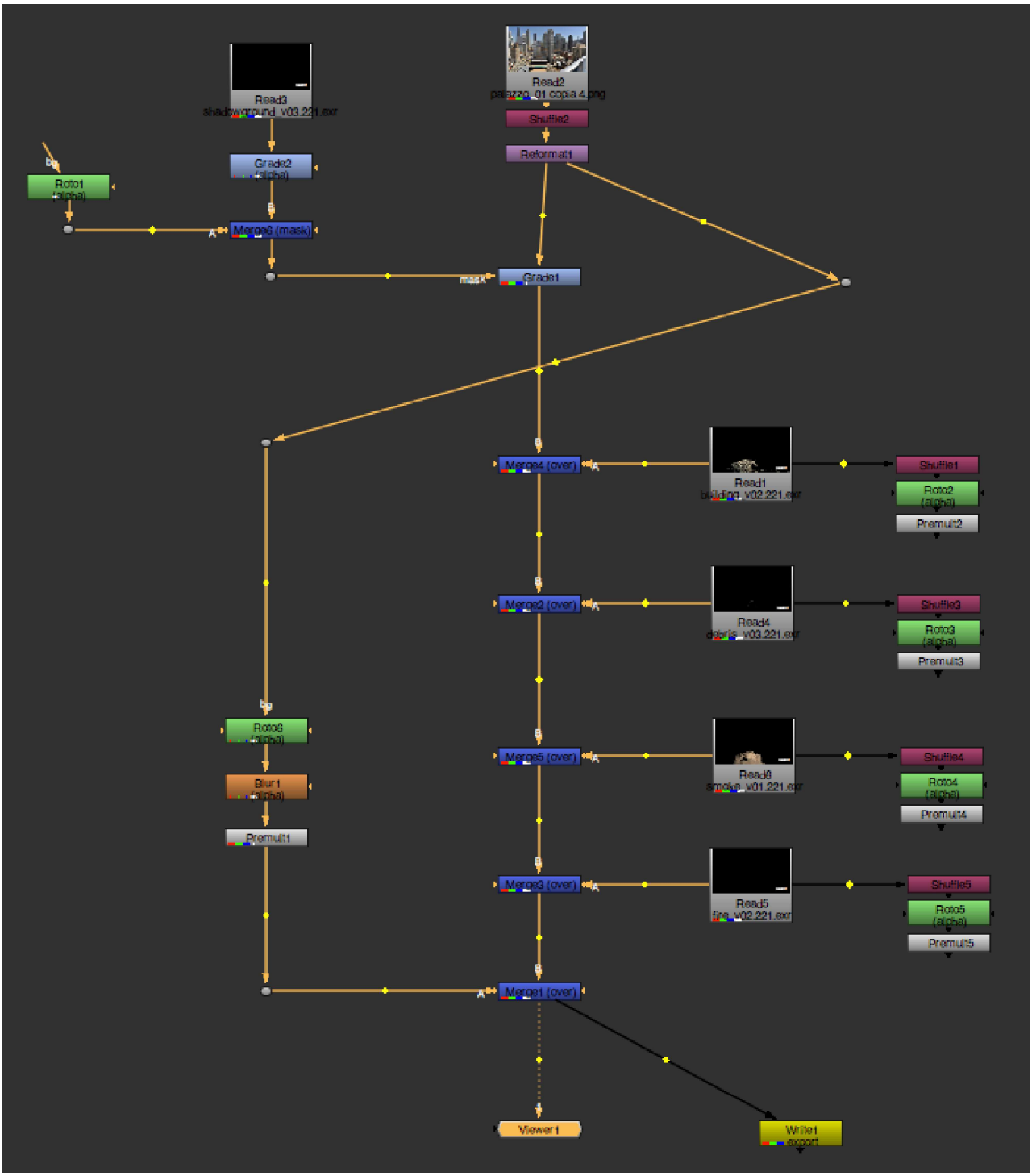
The Foundry

Nuke nasce dalla sempre più crescente richiesta di composizioni sempre più complesse, che ha dato il via allo sviluppo di soluzioni software sempre più avanzate e sempre più user friendly. *The Foundry*, produttrice del software di *Compositing Nuke*, basato su un sistema a nodi, è praticamente il leader in questo settore. *The Foundry* è stata fondata nel 1996, ed è oggi una società di software a livello mondiale con sede a Londra e uffici a Manchester, Los Angeles e nella Silicon Valley. L'azienda si propone di sviluppare strumenti, tecnologie e processi che consentano agli *"artisti"* di portare le loro idee creative alla luce in modo efficace e veloce. Il principale obiettivo di *The Foundry* è quello di credere in un mondo in cui le persone possono continuamente sviluppare e accrescere il loro potenziale creativo. A questo proposito l'azienda, grazie alle versioni non commerciali di molti dei suoi prodotti, tra cui *Nuke*, che è stata utilizzata all'interno di questo progetto, offre l'opportunità, a chiunque ne abbia voglia, di far conoscere i propri prodotti e strumenti. L'utente in questo modo, senza alcune limitazioni temporali, in quanto ha a disposizione tutto il tempo che desidera per studiare il programma e prenderne confidenza, può affinare le proprie abilità, sperimentare o semplicemente divertirsi ad esplorare il software e appassionarsi a questo mondo. L'obiettivo, in questo senso, che l'azienda si propone di portare avanti è quello di fornire un modo per formare i futuri *"artisti"*, aiutandoli a sviluppare le competenze necessarie per andare incontro alle esigenze delle aziende e formare delle figure professionali pronte ad entrare nel mondo del lavoro. *The Foundry,* inoltre, è in continua espansione ed è costantemente alla ricerca di nuovi talenti da aggiungere al suo team. Infatti, un altro importante elemento che caratterizza *l'azienda* è, appunto, la qualità del suo team, sul quale si è costituita la sua credibilità e la reputazione.

Nuke

Nuke è il software utilizzato per le operazioni di *Compositing* da molti dei migliori studi di *VFX* nel mondo. Il suo avanzato set di strumenti, basato su nodi, comprende più di 200 nodi differenti. *Nuke,* attraverso la libertà e la flessibilità di un ambiente di lavoro sia 2D che 3D, apre grandi possibilità all'innovazione del mondo dei *VFX*. I prodotti di *The Foudry,* e in particolar modo *Nuke,* sono largamente utilizzati in ogni settore dell'industria dei media, dall'intrattenimento alla pubblicità, e da tutti, dal singolo *"artista" VFX* ai principali studi cinematografici. *Nuke,* e le differenti versioni che *The Foundry* mette a disposizone, ovvero *Nuke Studio*, *NukeX* e *Nuke*, è un software leader nel settore: offre un set di strumenti per il *Compositing* basato su nodi all'avanguardia che permette un flusso di lavoro facile, veloce e qualitativo, offrendo inoltre livelli di potenza e affidabilità senza precedenti. Per quanto riguarda le licenze di utilizzo del software, tra le molte disponibili, ne esiste una non commerciale che da la possibilità a chiunque, come già detto, di imparare il software e mettere alla prova le proprie abilità. Per il progetto oggetto di questo scritto è stata utilizzata, per l'appunto, la versione non commerciale di *Nuke*.

Nuke, quindi, è stato utilizzato per gestire la fase di *Compositing,* la parte che costituisce il passo finale del processo di *Destruction* e di quasi qualsiasi altro processo o lavoro di post produzione. Nello specifico, essa, consiste nella combinazione delle cinque differenti simulazioni, renderizzate, realizzate nel passo precedente, assemblate insieme secondo un ordine preciso di precedenza e integrate in un ambiente tratto da uno scatto fotografico panoramico reale. Tutto questo ha permesso di realizzare qualcosa di plausibilmente realistico, grazie anche all'aiuto di alcune operazioni di elaborazione delle immagini.



L'immagine qui sopra mostra il grafo a nodi utilizzato per la fase di *Compositing*. Come si può notare lo schema è molto semplice: è stata importata come prima cosa l'immagine dello scatto di riferimento ricostruita, ovvero quella che è stata modificata al fine di rimuovere le ombre e il palazzo che poi sarebbe dovuto essere sottoposto al *Destruction*; questa immagine è poi stata ulteriormente sottoposta ad alcune elaborazioni, nello specifico, è stata ricavata una maschera che ha permesso di creare una sorta di profondità al suo interno, portando in primo piano la balaustra presente in basso a destra e facendo in modo che la scena del crollo avvenisse dietro di essa, questo per rendere la scena più verosimile. Ovviamente per ottenere lo stesso risultato si poteva operare in modo diverso, essendo un sistema nodale, non esiste un solo modo per arrivare a una stessa soluzione. Questa è stata, in sostanza, l'unica elaborazione importante che si è eseguita sul materiale in questa fase. Durante il *Compositing* non sono state riscontrate molte difficoltà, anche dal punto di vista dell'utilizzo del software, la sua comprensione si è rivelata abbastanza semplice, in quanto esso dispone di un'interfaccia molto intuitiva e inoltre la gestione del sistema nodale non è molto complicata. L'unica vera difficoltà è stata quella di pensare a come sovrapporre i vari livelli di simulazione renderizzati, quindi a come creare il giusto livello di profondità per non incorrere in errori che avrebbero potuto disorientare l'occhio di chi lo avrebbe osservato. Insomma il *Compositing* avrebbe dovuto ricreare il senso della scena, conferendogli un aspetto realisticamente plausibile. Un altro aspetto limitatamente complicato da gestire, è stato l'utilizzo del canale alfa, usato nella creazione della maschera all'interno dello scatto di riferimento, ma soprattutto per gestire la visibilità della simulazione renderizzata delle ombre. Per quanto riguarda le tempistiche, il *Compositing* è durato circa una settimana, quindi non moltissimo, in quanto, come già detto, le operazioni eseguite in questa fase non sono state così complesse.

Montaggio e Musiche

Dopo aver conferito un senso e un look al progetto video di *Destruction* tramite la fase di *Compositing*, è stato realizzato un breve montaggio, accompagnato da una musica scelta all'interno del catalogo *Onetrackaday* della *Machiavelli*, un'azienda attiva nel panorama della vendita di brani musicali in Italia e all'estero, con l'intento di spiegare in quaranta secondi, circa, quello che è stato tutto il percorso di lavorazione al fine di ottenere un prodotto finito.

Conclusione

In questa tesi sono state ripercorse tutte le tappe fondamentali che accompagnano la realizzazione di un qualunque progetto di *Destruction*. I lavori di *Destruction* possono essere svolti con vari approcci, alcuni più credibili altri meno, ma ad oggi forniscono quasi tutti dei risultati altamente qualitativi soprattutto da un punto di vista prettamente visivo. In ogni caso ad oggi non esiste una metodologia in grado di sovrastare le altre con grande semplicità, ognuna di esse ha aspetti positivi e negativi e in ogni caso ciascuna è in grado di fare un qualcosa meglio delle altre, sono in un certo senso equiparabili ma anche complementari. Per comporre un progetto di questa portata sono stati necessari molteplici elementi che combinati tra loro hanno permesso di raggiungere un risultato finale soddisfacente o comunque quello desiderato, da qui l'importanza di tutti questi passaggi e della loro accurata integrazione.
L'approccio *RBS* o *RBD,* utilizzato per le simulazioni di *Destruction,* è tutt'oggi quello che va per la maggiore rispetto, ad esempio, ad un approccio di tipo *FEA,* il quale ha senz'altro un grado di accuratezza fisica migliore. Nonostante questo, ciò che gli garantisce di non essere spodestato da atri metodi sono i miglioramenti effettuati negli anni, lo sviluppo tecnologico, ma soprattutto la grande abilità degli *"artisti"* degli effetti visivi nel gestirlo e che consente di raggiungere alti livelli di godibilità.
L'approccio in questa trattazione utilizzato, quindi quello *RBS* o *RBD*, è comunque complesso da gestire, ma rispetto ad esempio a quello di tipo *FEA* lo è molto meno, questo è uno dei motivi principali per i quali viene utilizzato in molti studi che svolgono l'attività di post produzione per gli effetti visivi, in quanto il costo e la gestione del livello di complessità di questo genere di simulazioni è molto oneroso e molti studi non si possono permettere di lavorare su pipeline magari più spettacolari ma al di fuori dei loro costi di produzione quindi preferiscono lavorare con un approccio meno complesso e affidarsi all'abilità dei propri *"artisti"*. È proprio questo che fa la differenza in qualsiasi caso, la bravura degli artisti degli effetti visivi nel gestire questo tipo di simulazioni in confronto a quella di un software, o di un approccio, che magari può risultare più ottimo in confronto ad un altro.

In questo progetto, lavorare con delle simulazioni che avrebbero dovuto interagire fra loro ma che di fatto non lo avrebbero fatto davvero perché realizzate separatamente e quindi indipendenti l'una dall'altra, il grosso del lavoro è stato proprio quello di integrare le singole simulazioni e fare in modo che queste sembrassero tutte un'unica cosa, quindi ad esempio che la simulazione del fumo che si sarebbe dovuto generare dal crollo fosse plausibile all'interno della simulazione del crollo dell'edificio. Ogni simulazione pertanto è stata adattata e corretta più volte per raggiungere dei risultati di plausibilità realistica, senza dubbio se si fosse utilizzato un altro software, o meglio un differente approccio, come ad esempio quello *FEA,* la gestione di tutto questo sarebbe stata più facilmente implementabile. In sostanza nel lavoro di *Destruction*, come in molti altri in questo settore, l'abilità di un *"artista"* è molto importante perché combinata con le potenzialità del software permette di raggiungere risultati sorprendentemente stupefacenti anche nella realizzazione di simulazioni molto complesse come ad esempio quelle su grande scala.

Ringraziamenti

In questo frangente colgo l'occasione per ringraziare l'azienda di cui sono stato ospite durante il mio periodo di tirocinio, ovvero la *EDI (Effetti Digitali Italiani)* di Milano, che mi ha permesso di poter realizzare questo progetto di *Destruction* fornendomi il supporto e gli strumenti. Ringrazio tutti, uno ad uno, rivolgendo un grazie particolare a *Stefano Leoni* e ai suoi collaboratori tra cui *Daniele De Maio* e *Annamaria Nigro*.
Ringrazio la Machiavelli Music, che mi ha concesso gratuitamente l'uso delle sue musiche permettendomi di accompagnare il montaggio del video realizzato per presentare il lavoro svolto per questa tesi.
Ringrazio la mia famiglia i miei amici e tutti coloro che mi hanno sostenuto e aiutato in tutto questo tempo.
Un grazie ancora più caloroso a mia nonna *Marisa* che, anche se non potrà assistere alla presentazione di questo lavoro e alla conclusione dei miei studi, so che sarà fiera del traguardo a cui sono arrivato.

Riferimenti

MIKE SEYMOUR
http://www.fmx.de/about/history/fmx-2015/boards-2015/program-board/mike-seymour-co-founder-fxguidecom-fxphdcom/
https://www.linkedin.com/in/mikesfxguide
http://sydney.edu.au/business/staff/mike.seymour

SIMULAZIONI INTERATTIVE E DINAMICA DEL CORPO RIGIDO
https://www.cs.rpi.edu/twiki/pub/RoboticsWeb/LabPublications/BETCstar_part1.pdf
http://graphics.cs.cmu.edu/courses/15-869-F08/lec/14/notesg.pdf

INTRODUZIONE
http://www.treccani.it/enciclopedia/effetti-speciali_(Enciclopedia-del-Cinema)/

DESTRUCTION: METODOLOGIE E APPROCCI
https://www.fxguide.com/featured/art-of-destruction-or-art-of-blowing-crap-up/

ERWIN COUMANS
https://www.linkedin.com/in/erwincoumans
http://bulletphysics.org/siggraph2011/

DIGITAL DOMAIN
http://digitaldomain.com
https://www.linkedin.com/company/digital-domain

PHYSBAM
http://physbam.stanford.edu
http://physbam.stanford.edu/links/backhistdisclaimcopy.html
http://physbam.stanford.edu/links/rigidbodies.html
http://physbam.stanford.edu/~fedkiw/
http://physbam.stanford.edu/~fedkiw/papers/stanford2006-07.pdf
http://physbam.stanford.edu/~fedkiw/papers/stanford2009-02.pdf
http://physbam.stanford.edu/~fedkiw/papers/stanford2005-05.pdf
https://www.fxguide.com/featured/ilm-at-the-oscars/

VORONOI E DELAUNAY
http://mat.unicam.it/piergallini/home/tesi/benigni.pdf
http://profs.scienze.univr.it/~caliari/aa1213/calcolo_scientifico/Visentin.pdf
http://www2.mate.polimi.it/corsi/papers/7034d9c55ef1fa85ee8db41feb71.pdf
http://www.ics.uci.edu/~eppstein/gina/scot.drysdale.html

DMM
http://www.pixelux.com/DMMplugin.html
https://www.fxguide.com/featured/dmm-fea-for-vfx/
http://graphics.berkeley.edu/papers/Parker-RTD-2009-08/index.html
https://www.youtube.com/user/dmmjedi
http://www.moving-picture.com

FRACTURE FX
http://www.fracture-fx.com
https://vimeo.com/channels/928048

MOMENTUM
http://exocortex.com/products/momentum

RAYFIRE
http://rayfirestudios.com

THINKINGPARTICLES 6.4™
http://www.cebas.com/index.php?pid=productinfo&prd_id=187
https://www.fxguide.com/quicktakes/thinkingparticles-v6-is-released-with-new-pricing/

REAL FLOW
http://www.realflow.com
http://support.nextlimit.com/display/rf2014docs/Simulation+-+Caronte+Dynamics?_ga=1.33262647.864985537.1475567185
http://support.nextlimit.com/display/rf2014docs/Caronte+-+Rigid+and+Soft+Bodies
http://support.nextlimit.com/display/rf2014docs/Voronoi+Tools
http://support.nextlimit.com/display/rf2014docs/Caronte+-+Fracture+Tools
http://support.nextlimit.com/display/rf2014docs/Caronte+-+MultiBodies
http://support.nextlimit.com/display/rf2014docs/Caronte+-+Objects
http://support.nextlimit.com/pages/viewpage.action?pageId=24248985
https://www.fxguide.com/featured/science-of-fluid-sims-pt-2-realflow/

HOUDINI E SIDEFX
https://www.sidefx.com/filmtv/
https://www.sidefx.com/products/houdini/
https://www.sidefx.com/filmtv/products/
https://www.sidefx.com/company/about-sidefx/
https://www.sidefx.com/products/houdini-fx/
https://www.fxguide.com/featured/time-for-destruction-the-tech-of-quantum-break/
https://www.fxguide.com/quicktakes/fmx-2014-houdini-engine-for-cinema-4d-realflow-2014/

BULLET 2.83 PHYSICS SDK
http://bulletphysics.org/wordpress/

FORMATI: OPENEXR, ALEMBIC, OBJ

http://www.openexr.com
http://www.alembic.io
http://vintageapple.org/macbooks/pdf/Graphics_File_Formats_Second_Edition_1996.pdf
http://www.sidefx.com/docs/houdini/io/alembic
https://wiki.blender.org/index.php/Extensions:2.6/Py/Scripts/Import-Export/Wavefront_OBJ
https://wiki.blender.org/index.php/Dev:Ref/Release_Notes/2.78/Alembic

DOCUMENTAZIONE HOUDINI

http://www.sidefx.com/docs/houdini/_index
https://www.sidefx.com/docs/houdini15.0/
https://www.sidefx.com/forum/topic/43515/?page=1#post-195305

BSDF

https://measuretruecolor.hunterlab.com/2013/10/15/bsdf-brdf-btdf/
http://blender.stackexchange.com/questions/785/what-is-a-bsdf

NUKE E IL COMPOSITING

https://www.thefoundry.co.uk
https://www.thefoundry.co.uk/products/nuke/about-digital-compositing/