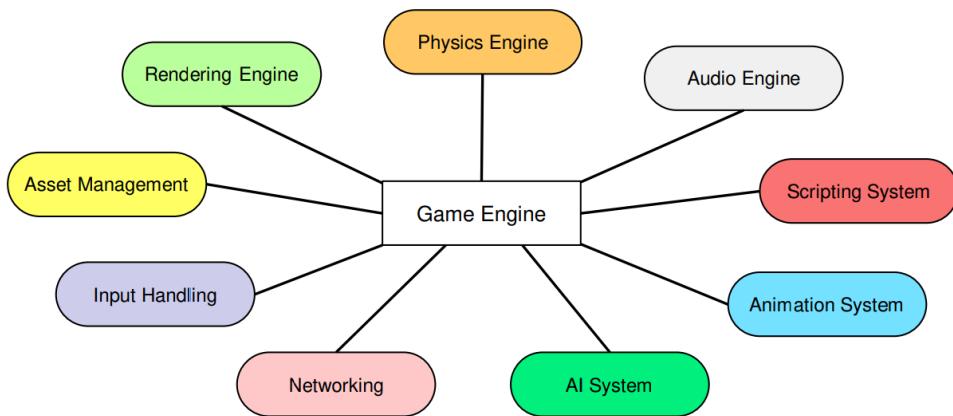
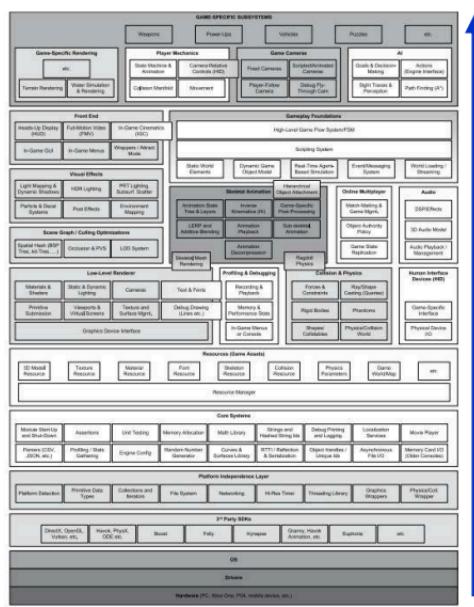


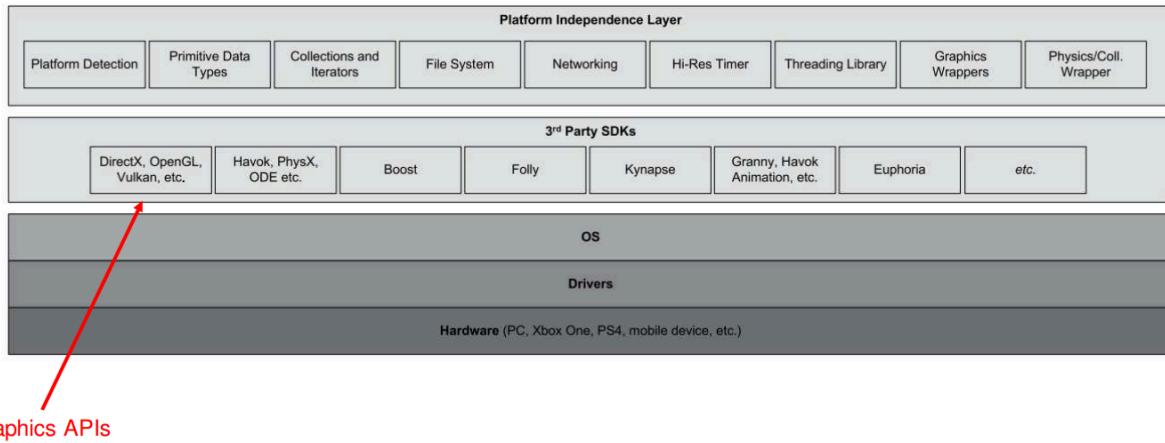
Lezione 5 20/03/2025

Un **game engine** è un framework software o una piattaforma progettata per facilitare la creazione e lo sviluppo di videogiochi. Fornisce una raccolta di strumenti, librerie e funzionalità che semplificano il processo di sviluppo, permettendo agli sviluppatori di concentrarsi sul design e sul gameplay invece di creare i sistemi di base da zero.



Vedremo questa struttura di game engine partendo dal basso:





L'**hardware** è guidato da **drivers**. Al di sopra sta il **sistema operativo**.

Poi troviamo le **SDK** (Software Development Kit).

Abbiamo **Graphics API** come OpenGL, un'API di basso livello per interagire con la scheda grafica.

La controparte, **DirectX**, include oltre alla parte di rendering anche API per comandare gli input, l'audio... è quella più diffusa perchè sviluppata da microsoft.

Più o meno tutte le API si equivalgono, dipende dall'hardware che devono comandare (come quelle per la playstation sono specializzate per quell'hardware).

Recentemente le API OpenGL sono state sostituite dalla API **Vulkan**, che sono ancora più basso livello, pensate per essere più efficienti.

Abbiamo poi delle **librerie esterne**, per aggiungere funzionalità specifiche. **Havok** gestisce la **fisica**. Open Dynamic Engine è specializzata per la simulazione degli oggetti rigidi, permette di simulare per esempio gli esseri umani in modo realistico. Bullet è una libreria per la fisica, è stata premiata nel 2015.

Poi abbiamo una serie di librerie per lo sviluppo (Boost, Folly Kynapse)

La fisica è anche legata pesantemente all'**animazione**. Queste sono spesso librerie molto specifiche, per compiere delle azioni animate seguendo una certa logica. Tra queste c'è anche Havok.

Un altro tool importante è Blender che serve per la modellazione 3D ma contiene anche tool per animare gli oggetti. Vedremo poi come queste animazioni possono

essere fatte usando il rigging, ovvero lo scheletro, di modo che le animazioni siano coerenti tra loro.

Abbiamo poi gli **asset**, degli elementi da inserire nella scena, sono spesso fatti con Blender.

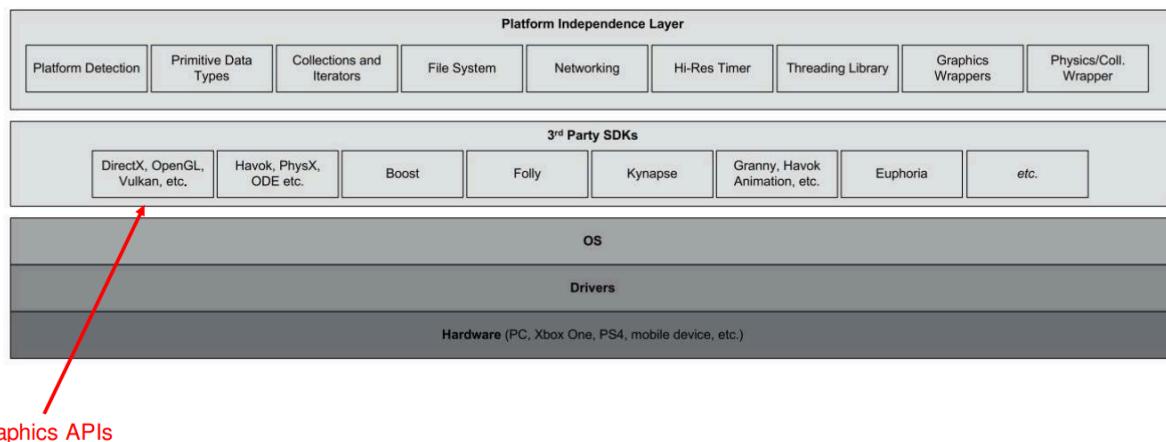
Per esempio SpeedTree ha usato tecniche di fotogrammetria per vendere asset di vegetazione, animandoli e creandoli in modo procedurale (ogni albero è diverso).

Oodle è pensata per ottimizzare in diverse condizioni la **compressione dei dati**, specialmente per trasmettere dati in rete.

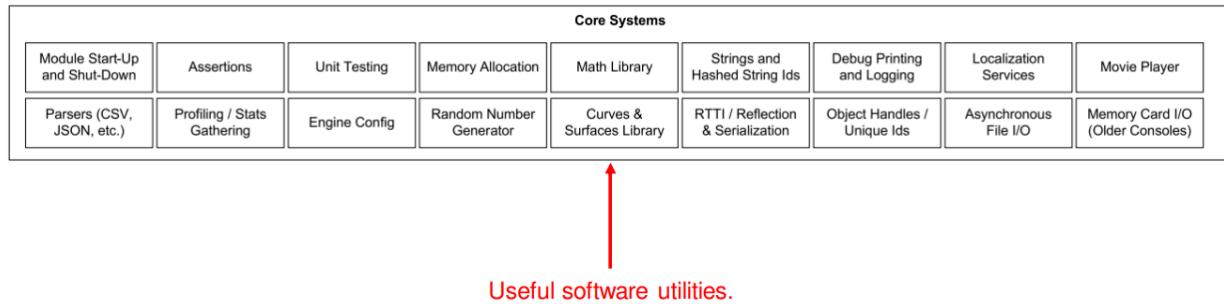
Poi ci sono librerie che gestiscono gli asset legati all'**audio**. OpenAL è una di queste librerie, in grado di gestire diversi formati audio, dando la sensazione di audio tridimensionale, compressione, riproduzione...

Un'altra parte non banale è quella di video, o meglio **formati multimediali** che non abbiamo già visto.

Per esempio Wwise permette anche di trasformare, mixare flussi audio, applicare post processing in base alla situazione di gioco.



Nella parte sopra serve per mascherare l'esistenza di queste librerie, sono dei wrapper scritti a doc per gestire le librerie sottostanti, con interfacce, linguaggi, gestione dati...



Sopra alle librerie troviamo funzionalità software che servono per sviluppare il gioco stesso. Quindi abbiamo utilità generiche.

Per esempio serve una **gestione degli errori**, non solo di programmazione ma anche di uso. Gli errori che si possono verificare **durante una partita** possono essere causati da un comportamento inaspettato dell'utente, per esempio sta andando in un'area di gioco non ammessa. Queste cose non devono fare crashare il gioco, ma devono essere gestite nel gioco stesso.



Gli errori che accadono **durante lo sviluppo** vanno invece resi il più evidente possibile.



Gli errori di **programmazione** invece vengono gestiti con le tecniche classiche: exceptions, asserts, try catch.

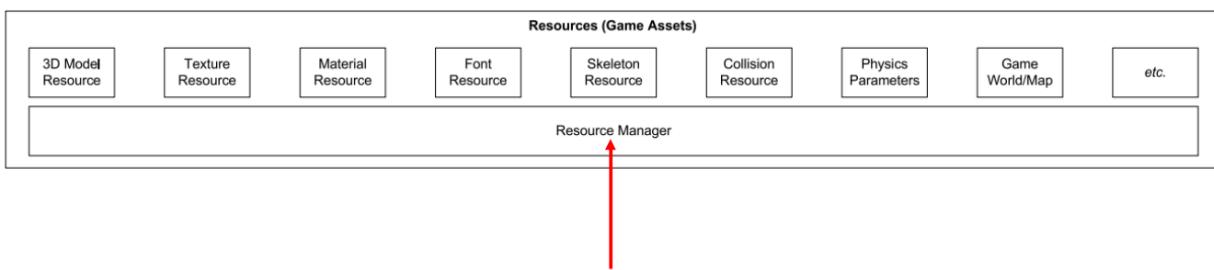
Un'altra cosa che va gestita è la **memoria**. Per esempio in C/C++ quando dobbiamo allocare grossi dati, dobbiamo chiedere al sistema operativo di allocare la memoria RAM. Questo sistema richiede tempo, i giochi allocano molta memoria, non possono chiedere ogni volta al sistema operativo. Tipicamente i giochi pre-allocano un'area di memoria grande più o meno quanto dovrebbe essere sufficiente. Vanno quindi a fare una finta allocazione. Ridefiniscono il concetto di allocazione di memoria, senza parlare al sistema operativo. Devono quindi avere questo **gestore di memoria**.

Poi abbiamo le **configurazioni**, ovvero quei parametri per esempio di qualità video. Tutte queste informazioni vanno gestite, vanno rese persistenti, tipicamente in file di testo o file binari, anche messi in cloud.

Poi bisogna gestire la localizzazione del **testo**. I giochi devono essere localizzati nelle varie lingue. Tutto deve essere progettato già dall'inizio per supportare più lingue.

Id	English	French
p1score	“Player 1 Score”	“Joueur 1 Score”
p2score	“Player 2 Score”	“Joueur 2 Score”
p1wins	“Player one wins!”	“Joueur un gagne!”
p2wins	“Player two wins!”	“Joueur deux gagne!”

Questo vale anche per l'audio, per i sottotitoli...

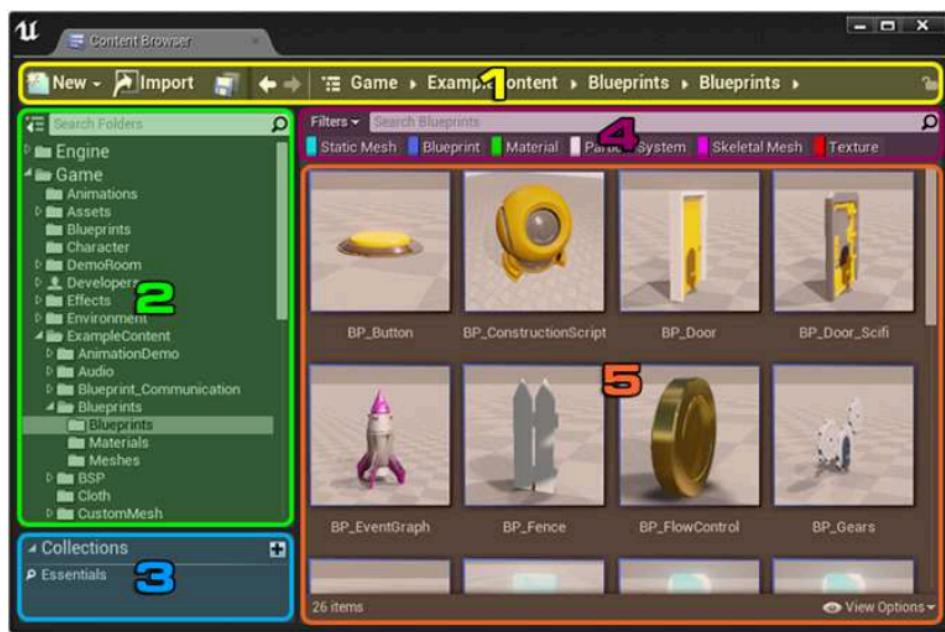


Un game engine deve poter **gestire dati di diversi formati**: modelli 3D, textures, materiali, il testo, font, animazioni con gli scheletri, fisica, streaming della scena.

La parte di grafica non è solo il come rappresento i dati nell'ambiente 3D, ho bisogno di informazioni geometriche sulla forma degli oggetti, si usa un reticolo di punti, ovvero un insieme di vertici che collegati formano un'approssimazione della forma dell'oggetto. Questo però non basta per renderizzare la scena, devo anche scegliere come disegnare la geometria. L'apparenza viene gestita per esempio da textures. Quindi si devono definire anche i riflessi, i materiali, che cambiano l'apparenza.

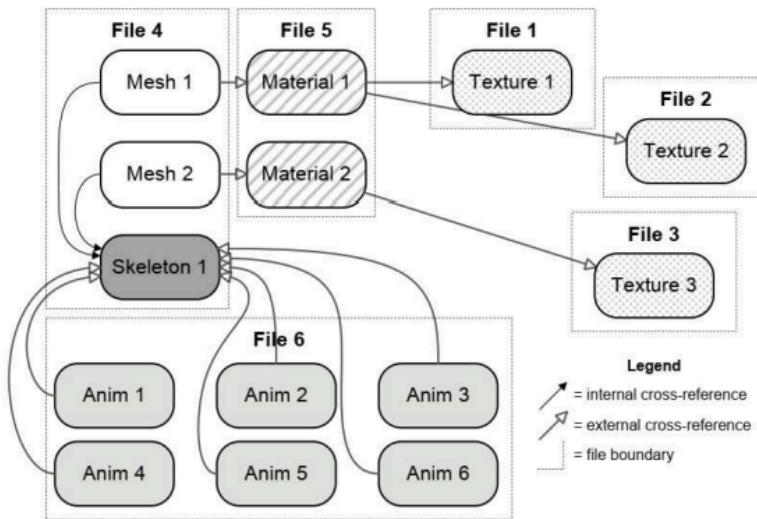
Tipicamente esiste un layer chiamato **resource manager** che deve capire tutti questi diversi dati e importarli con un'interfaccia comune di modo che siano usabili.

Inoltre, tutto questo serve a runtime, ma spesso i game engine sono anche usati durante la fase di sviluppo, quindi ci possono essere dei tool per gestire questi asset, come un database per gestire tutte le textures, anche quelli che magari non finiscono nel gioco.

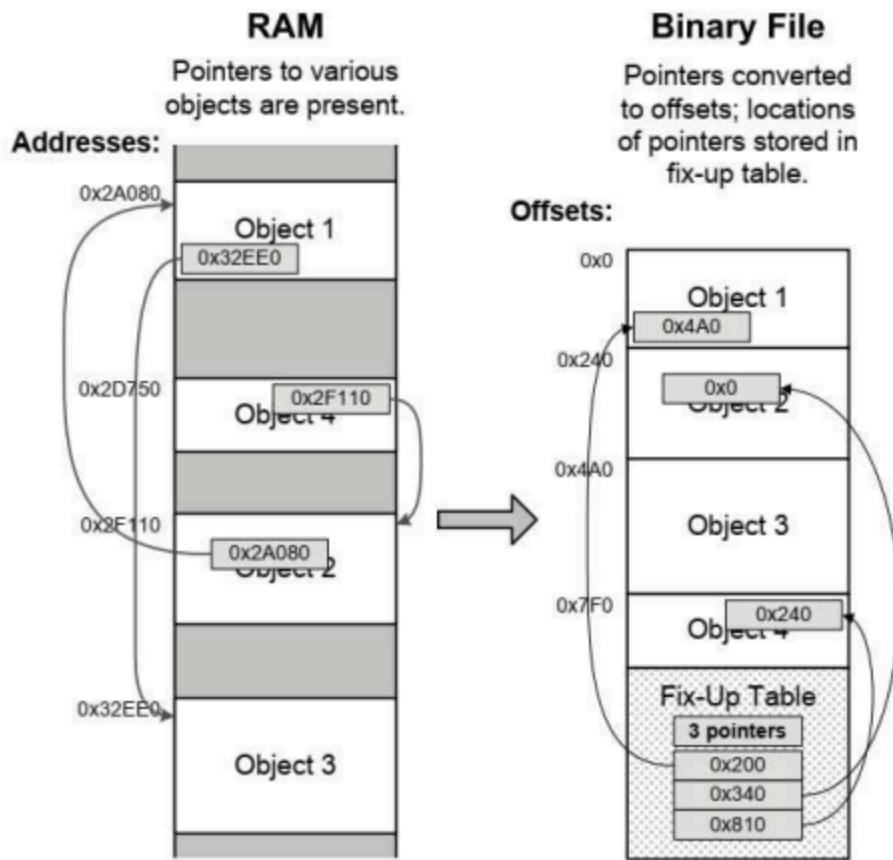


Le **risorse** non sono isolate, indipendenti, spesso dipendono l'una dall'altra. Per esempio la geometria di un oggetto deve essere associata ad un materiale, ad

un'animazione che ha delle regole... queste dipendenze sono sia **logiche** ma anche **di dati**.

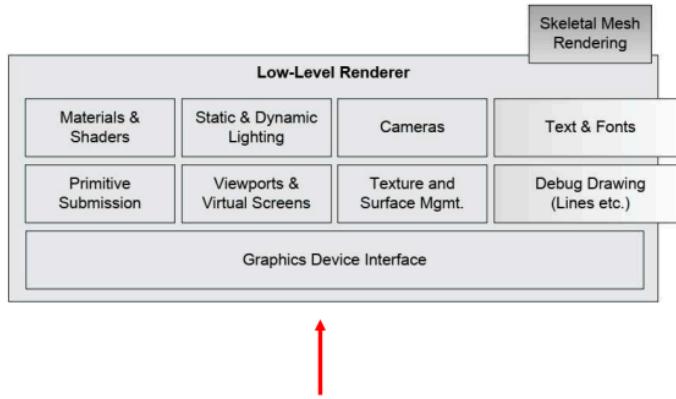


Dato che i dati sono pre-allocai, queste dipendenze che possiamo vedere come puntatori vanno bene quando sono in memoria, ma quando sono su disco come trasformiamo un puntatore?



I dati su file vengono impacchettati in sequenza, e i puntatori diventano degli **offset** per ritrovare l'oggetto più avanti nel file.

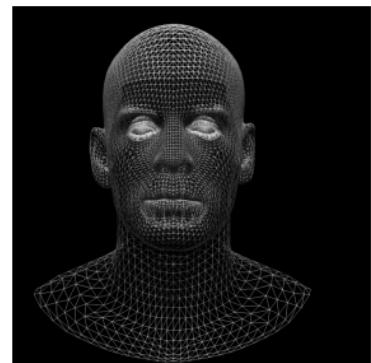
Questo insieme di byte viene quindi caricato in memoria, e quegli offset vengono ri-trasformati in puntatori. Questo serve per salvare su file degli asset dipendenti tra loro senza allocarli da zero. Vengono messi in un **singolo file di risorse**, che viene **caricato in blocco**.



Raw rendering facilities of the engine focused on rendering a collection of geometric primitives as quickly as possible.
Manages all parts of the rendering pipeline.

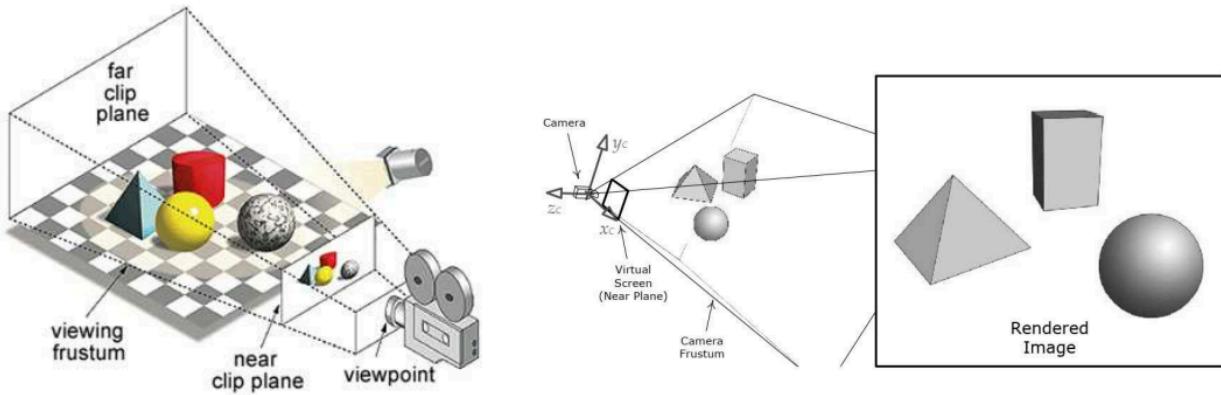
Una delle parti più importanti del game engine è il **rendering** della scena e degli asset, ovvero mostrare a schermo il mondo 3D.

Il renderer della scena deve gestire come la scena è codificata, tipicamente tramite **mesh**, dei reticolati di punti connessi tra loro che definiscono la struttura geometrica.



Dopo aver scelto come rappresentare la geometria di un oggetto (triangoli, quadrati...) serve una **camera**, ovvero il come vogliamo vedere il mondo 3D da un certo punto di vista. Questa può essere il punto di vista dell'utente, o in un gioco in terza persona sarà dietro all'utente.

La camera definisce una regione di spazio che viene inquadrata, proiettando il mondo in un'immagine 2D.



Un altro ingrediente sono le proprietà visuali delle superfici. Ovvero, ci serve qualcosa per colorare la geometria, ovvero le **textures**.

La regione dei capelli avrà un certo materiale, i capelli un altro...

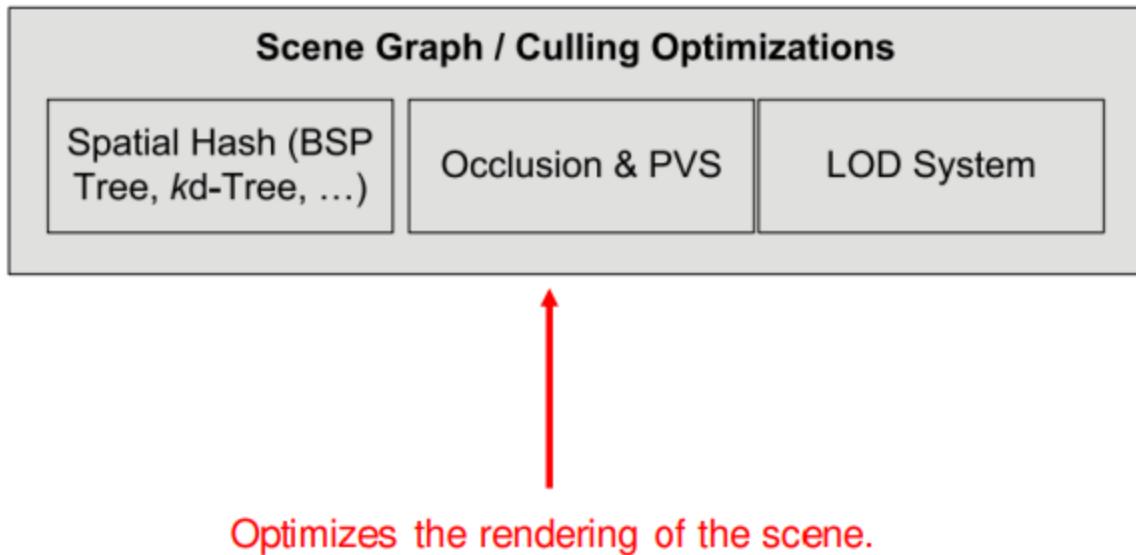


Servono poi le **luci**, perchè senza luci gli oggetti sono piatti, non si vede la geometria bene.



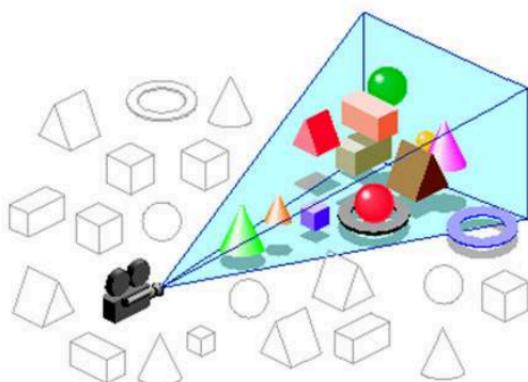
Quindi per renderizzare la scena 3D serve: **la geometria, una camera, delle proprietà associate ai materiali e delle sorgenti luminose.**

Queste informazioni vengono prese dalla pipeline di rendering che calcola ciascun pixel.

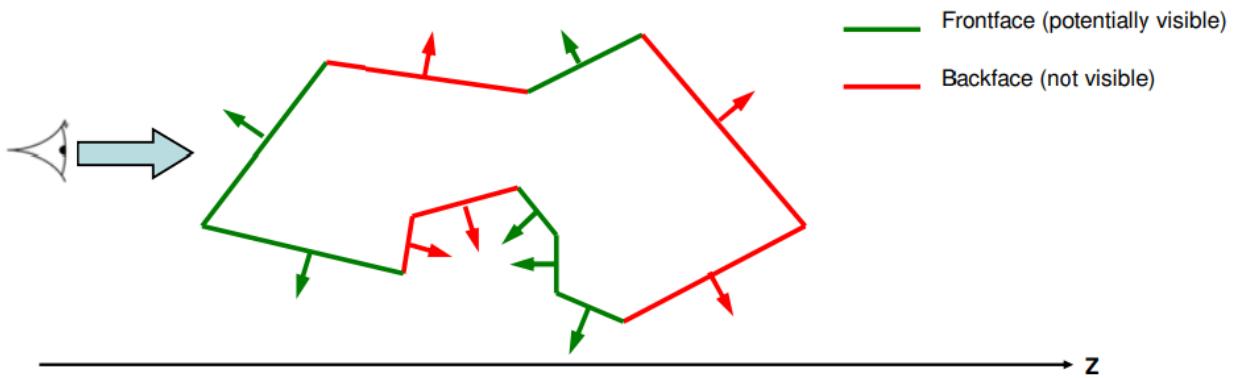


Abbiamo poi la fase di **occlusion detection**, per capire se tutta la scena che sto guardando è **necessaria per il rendering**, e il **livello di dettaglio** di ciascun oggetto.

Quindi il game engine, prima di passare i dati al renderer grafico, deve togliere tutto quello che non è necessario.



La camera è in grado già di suo di capire cosa vede e cosa no. Il game engine sa cosa vede la camera e quindi può eliminare tutto quello che sta fuori. Questa tecnica si chiama **frustum culling**.

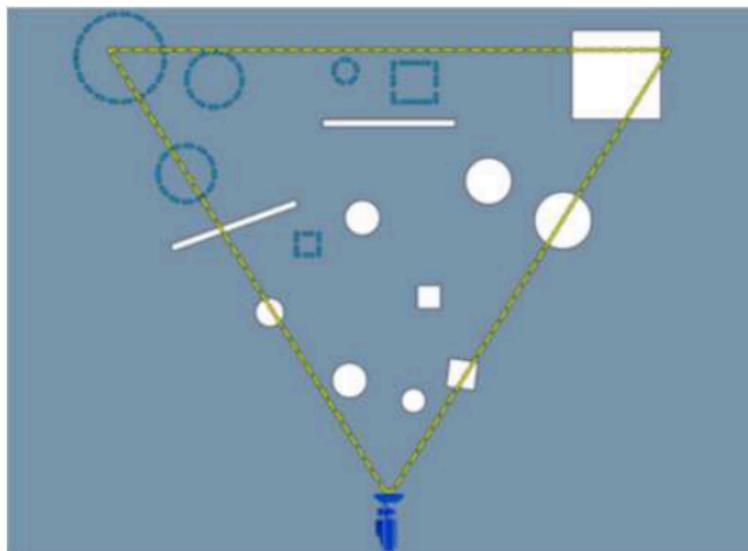


Un'altra cosa che va rimossa è il retro degli oggetti dato che non li vedo. Abbiamo quindi delle facce nascoste rispetto al punto di vista della camera. Questo si chiama **backface culling**.

Geometricamente parlando, se scomponiamo le componenti delle frecce rosse vediamo che vanno da sinistra a destra, come il punto di vista dell'utente.

Quindi tutte le facce di un oggetto che hanno le normali "opposte" rispetto al punto divista sono potenzialmente visibili (se non occultate).

Come faccio a trovare quelle **occluse**? Uso l'**occlusion culling**.



Qui le forme tratteggiate sono occluse.

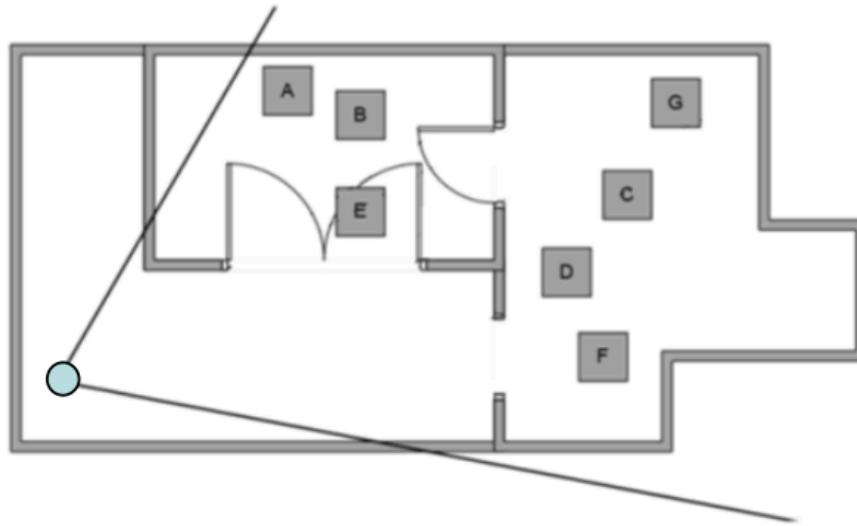
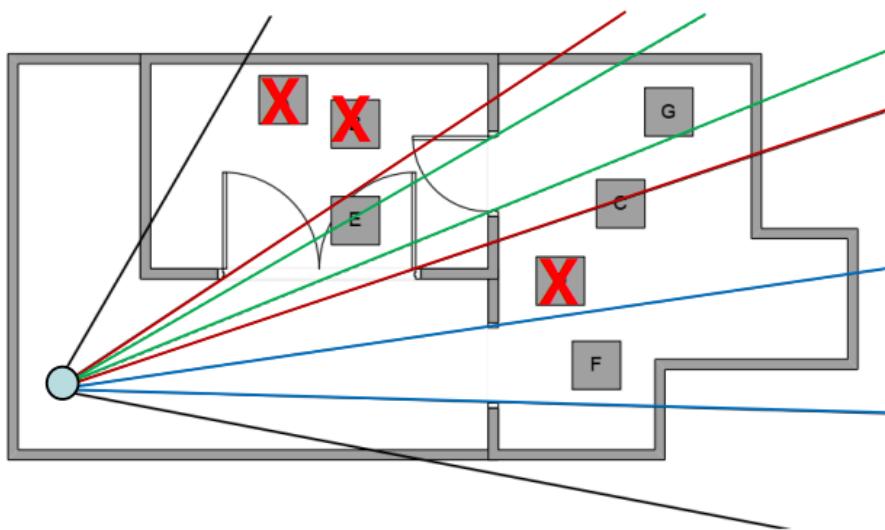


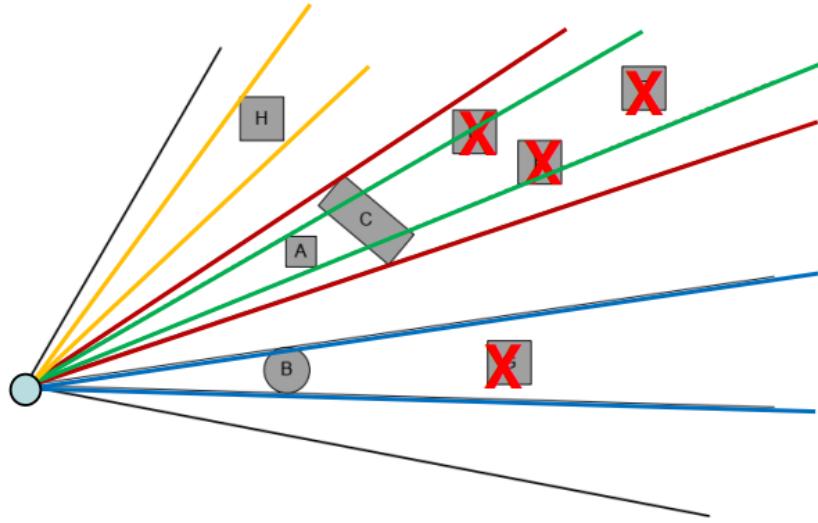
Figure II.48. Portals are used to define frustum-like volumes, which are used to cull the contents of neighboring regions. In this example, objects A, B and D will be culled because they lie outside one of the portals; the other objects will be visible.

In questo esempio, l'utente è il pallino. Se i quadratini sono i nemici, quali vedo? Ci sono dei muri, delle porte. Si usano i "portali", ovvero delle porte. Ovvero, sapendo che ci sono geometricamente delle porte che hanno dei vincoli, delle dimensioni, un'apertura... dal punto di vista del soggetto si lanciano dei "raggi", e sostanzialmente porta per porta si considerano i coni ridotti



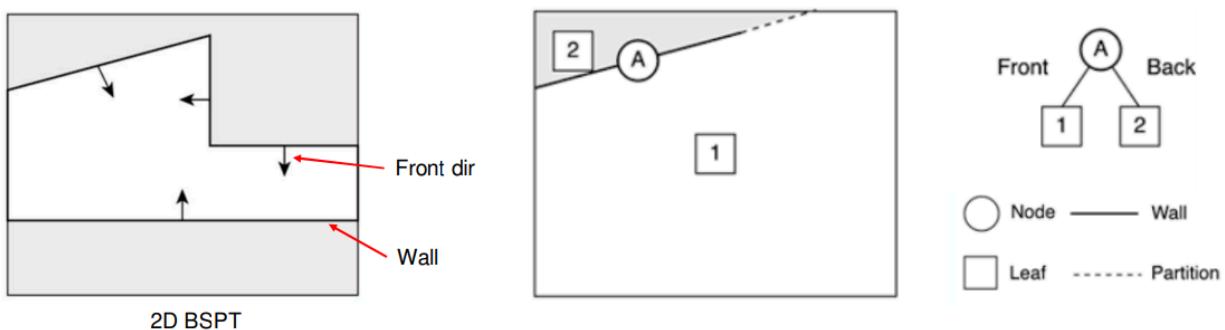
Nel caso generico dove non ho delle "porte" che mi permettono di restringere l'algoritmo, si usa un concetto di **anti-portale**. Per ogni oggetto, **partendo da quelli più vicini alla camera**, si vedono i suoi limiti geometrici e si lanciano dei raggi che sono tangenti alla posizione degli oggetti, e si butta tutto quello che vive nel cono di vista che ho individuato.

◆ Anti-portals



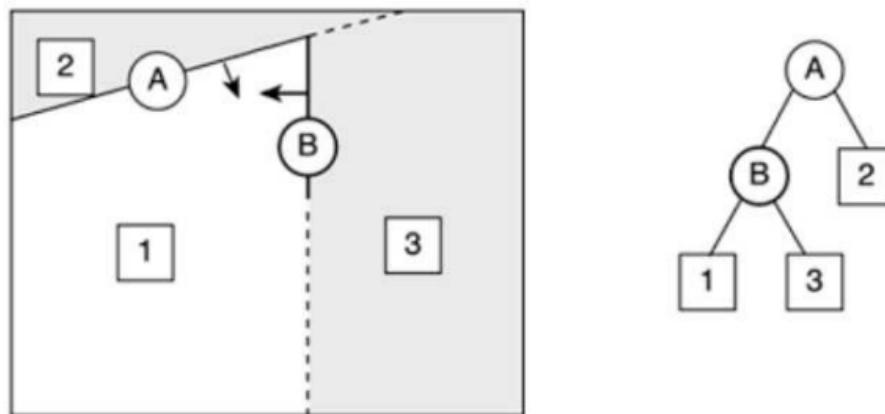
Quindi come faccio a trovare gli oggetti più vicini alla camera?

Con i **binary spatial partitioning tree**. Sono delle strutture dati che consentono di navigare una regione dello spazio in modo efficiente, e trovare gli oggetti più vicini all'utente.

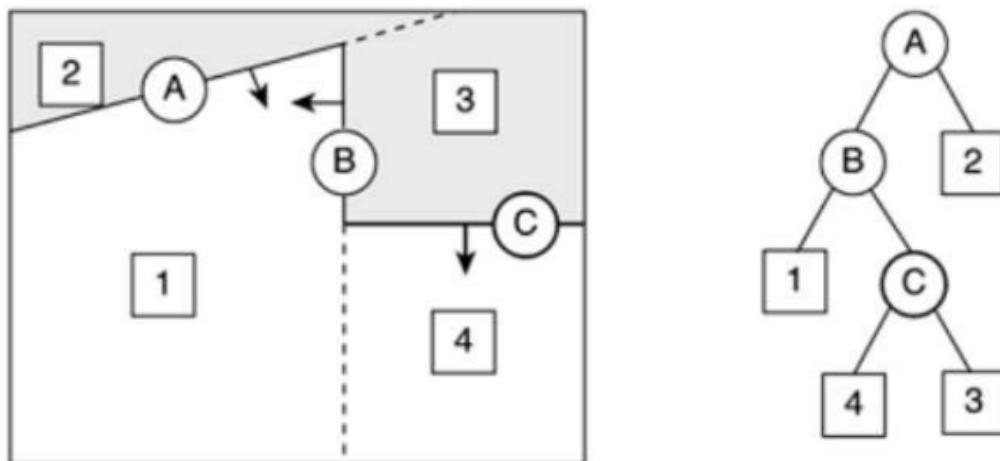


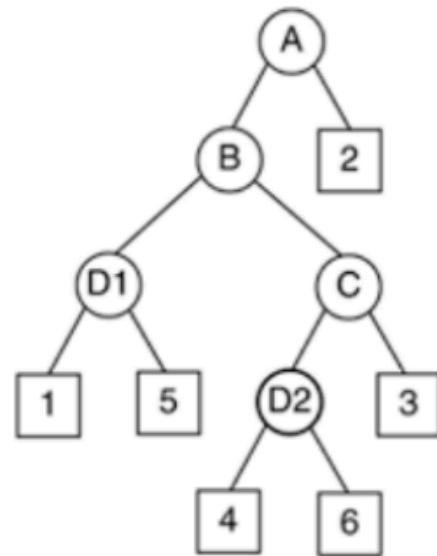
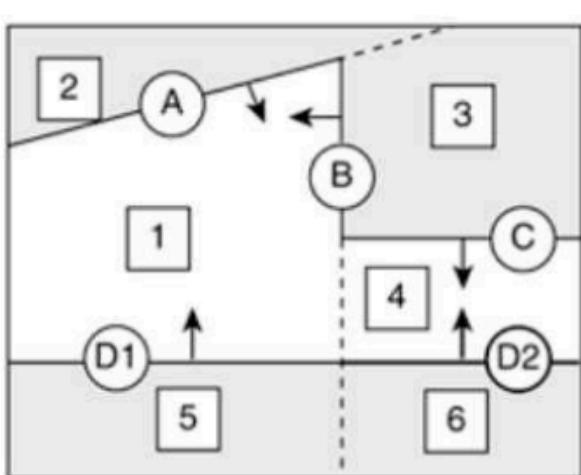
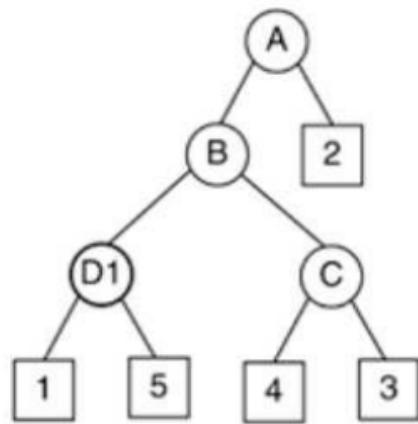
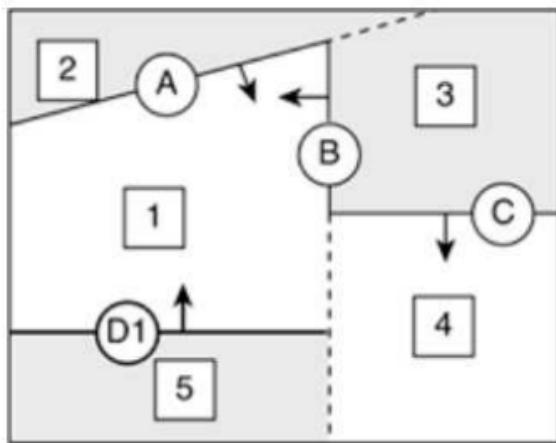
Per esempio se consideriamo questi 2 muri (non quelli a destra e sinistra), le frecce indicano qual è il lato del muro visibile. I binary spatial partitioning tree suddividono per piani uno spazio 3D, separandolo ogni volta in 2 dimensioni.

Per esempio prendiamo il muro in alto a sinistra come riferimento per tagliare in 2 lo spazio della stanza, che quindi diventa divisa nella regione 1 e la regione 2. L'elemento di separazione è indicato con A. La regione 1 vive nella parte frontale del muro (dove punta la freccia), la regione 2 sta dietro al muro. Quindi quando si costruisce l'albero binario, si mettono a sinistra le regioni frontali e a destra quelle dietro.



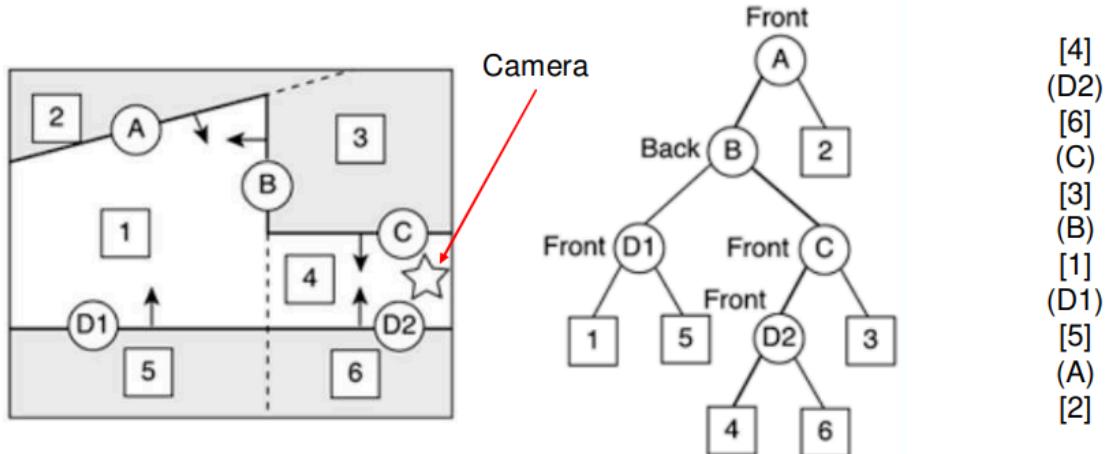
Poi si va a prendere un altro muro. Il muro B divide tutta la regione 1 di prima in 2 regioni, 1 (davanti) e 3 (di dentro).





Continuo ad aggiungere muri e arrivo a questo albero di partizionamento.

Navigando nell'albero riusciamo a capire come è fatta la regione davanti e dietro di ciascun muro.

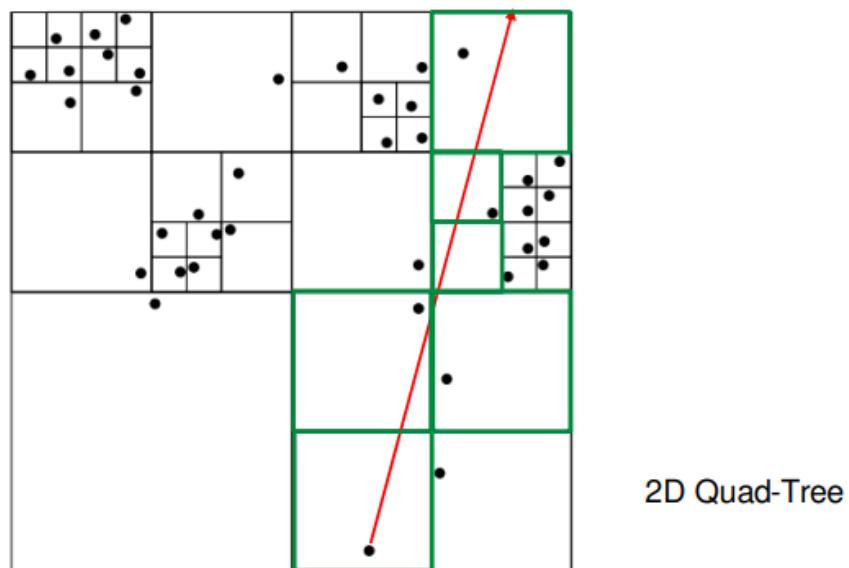


Per esempio se ci mettiamo nella regione 4 con la camera, se risaliamo l'albero incontriamo D2, quindi trovo la regione 6. Poi salgo, trovo C, e trovo la regione 3. etc

Questo è come verrebbe navigato l'albero partendo dalla regione 4, da questa sequenza possiamo vedere quali sono gli elementi più vicini alla camera.

L'idea è di costruire questa struttura dati per la scena con gli oggetti, e usarla per trovare gli oggetti più vicini alla camera, per esempio per fare quel test degli anti-portali.

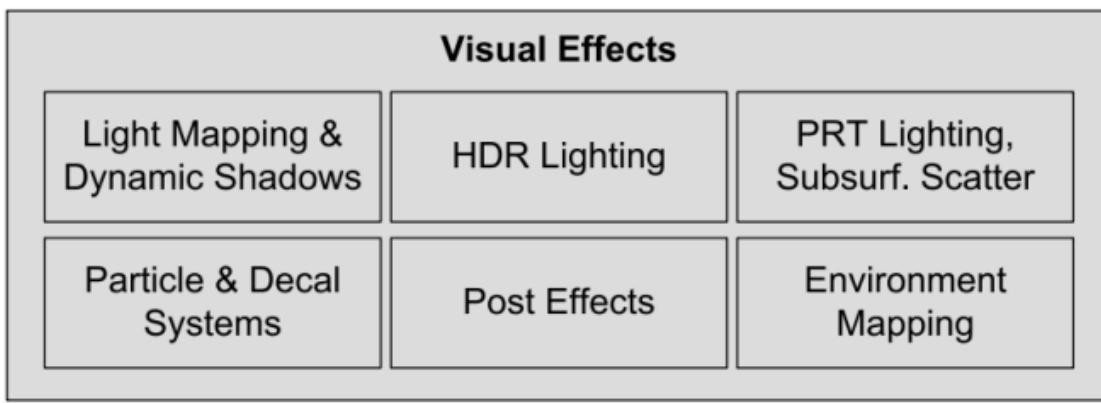
Noi potremmo quindi prendere l'intero mondo 3D e farlo a fette rispetto agli assi di distribuzione standard del mondo.



Si può usare la struttura dati **quad-tree**, partiziono lo spazio 2D in questo caso, in quadranti, e se un quadrante contiene più di un oggetto di interesse, continuo a partizionarlo finché ne contiene solo uno.

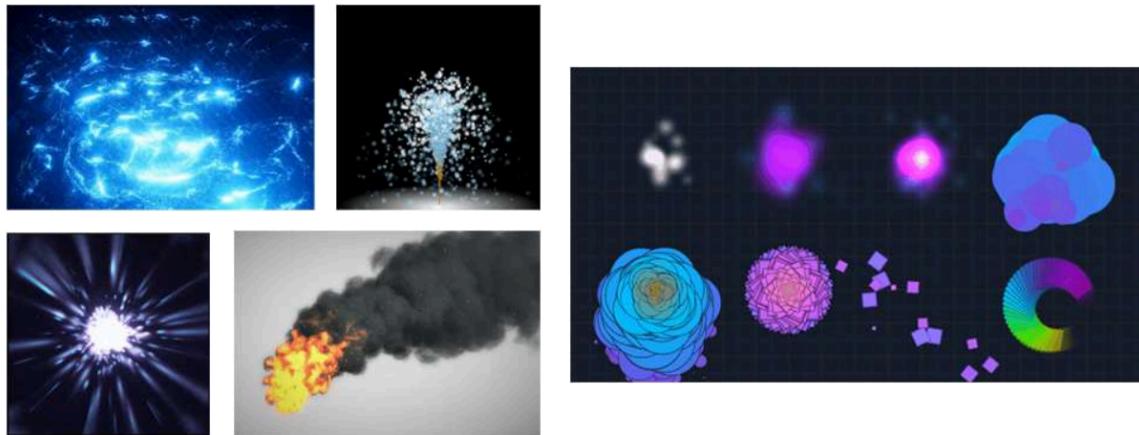
Se quindi "spariamo" in una direzione, al posto di controllare che oggetto incrocia controlliamo i quadranti che incrocia, e a quel punto vado a testare solo gli oggetti in quei quadranti. Questo riduce molto lo spazio di ricerca.

Quindi il game engine spesso deve **filtrare i dati** (ridurre lo spazio di ricerca), prima di vedere quali oggetti sono visibili o cosa viene colpito... sia per il rendering che per la logica di gioco



Manages specialized rendering needs of particles, decals and other visual effects.

Un'altra parte importante dei game engine è quali **effetti grafici** applicare alla scena che sta renderizzando. Spesso, ciò che stiamo renderizzando non è composto da oggetti macroscopici: abbiamo una stanza, posso modellarla mettendo banchi, sedie... sono elementi macroscopici che posso modellare a doce e piazzare nella scena. Ci sono nella realtà degli eventi o delle situazioni in cui non è possibile modellare in questo modo delle realtà, come quando abbiamo degli effetti particellari, che non sono generati da un singolo oggetto ma da una moltitudine di oggetti che interagiscono tra loro. Questi sono effetti tipo il fumo, il fuoco, l'acqua...



Questi si creano andando a comporre e animare delle texture di oggetti elementari non necessariamente 3D.

Più queste texture sono piccoli e più abbiamo un effetto realistico.



Abbiamo poi le **decalcomanie**, che sono delle modifiche a runtime delle texture. Per esempio disegnare l'apparenza di un buco sulla texture.



Poi abbiamo gli **effetti ambientali**, tipo le nuvole nel cielo, le increspature dell'acqua...



Poi abbiamo gli **overlay**, per esempio per aiutare a selezionare un oggetto.



Poi c'è il **motion blur**, per dare l'effetto di movimento quando si muove la camera, e il **depth of field** sfuocando gli oggetti lontani.



Poi abbiamo **vignette** e **colorization**.

Questi sono tutti effetti per comunicare qualcosa al giocatore.