HPMS Engine

# Concept

Partendo dal prototipo sviluppato tramite LWJGL (JRD3) verrà sviluppato un motore analogo più leggero e ottimizzato in codice nativo (C++)

# Filesystem

Per avere compressione, trasparenza e velocità di caricamento delle risorse, verranno utilizzati dei formati binari e un filesystem virtuale.

Utilizzando la libreria *PhysFS*, tutte le risorse presenti in un archivio (formato *7z*, in questo caso rinominato *hpak*) vengono caricate in fase di startup e montate su una directory virtuale strutturata nel modo seguente:

data/  
|  
+*--- HPMS.cfg*|  
+*--- scripts/*|  
+*--- rooms/*|  
+*--- shaders/*|  
+*--- models/*|  
+*--- textures/*|  
+*--- screens/*|  
+*--- sounds/*

Gli scripts, shaders, immagini e audio rimarranno in formato standard, mentre i modelli 3D e i descrittori delle ambientazioni verranno compressi in un formato binario.

# Engine

Il motore grafico sarà una versione ottimizzata e leggera di JRD3 (prototipo del medesimo in LWJGL3).  
La struttura modulare permetterà di scegliere l’API grafica per il rendering (al momento supporterà solamente OpenGL). Per questo motivo ogni modulo e classe che contiene chiamate alla GPU sarà una precisa implementazione di una interfaccia generica. Ad esempio il motore utilizza una generica interfaccia *Renderer* per il disegno delle primitive e delle texture, e in base all’implementazione (*GLRenderer* nello specifico caso di OpenGL) verranno invocate le funzioni specifiche. I moduli specifici dipendenti dall’API grafica sono:

* Gestore del Rendering;
* Gestore degli Shader;
* Gestore della finestra;
* Gestore degli effetti post-rendering (Framebuffer).

La stessa *Pipeline* di disegno (quindi la sequenza dei render pass) è un’interfaccia, e si può scegliere la tecnia in base alla tipologia di scena che si vuole rappresentare per una determinata sessione, in particolare:

* Scena in 2.5D con sfondi pre-renderizzati, modelli 3D e maschera di profondità (*R25DPipeline*). Questa è la tecnica più comune utilizzata da questo tipo di motore;
* Scena completamente 3D (*Full3DPipeline*).Questa può essere utilizzata per dei piccoli minigiochi;
* Scena prevalentemente 2D senza profondità (*GUI2DPipeline*). Utilizzata per le GUI, permette comunque di aggiungere modelli 3D sopra di essa (ma senza alcuna maschera di profondità).

Alla pipeline (e successivamente al motore) verranno passati tutti gli oggetti inseriti all’interno del costrutto *Scene.* Qui è possibile aggiungere:

* Entità (ovvero riferimenti a modelli 3D, comprensivi di informazione su posizione e animazione);
* Immagini (che possono essere di background, foreground o maschere di profondità);
* Array di immagini (che in base alle loro proprietà verranno renderizzate come una gif, per dare l’effetto di sfondo animato);
* Varie (effetti particellari, testo ecc...)

La camera viene creata in automatico insieme alla scena, all’inizio di ogni sessione.

L’applicativo permette di interagire con la scena e la camera in diversi momenti della sessione:

* Inizializzazione, chiamata una sola volta all’inizio della sessione. Qui vengono solitamente creati gli oggetti da aggiungere alla scena;
* Input, chiamato ad ogni pressione di un tasto, è possibile gestire l’interazione con l’utente;
* Loop, chiamato ad ogni frame. Qui si gestiscono tutte le interazioni tra entità;
* Cleanup, chiamato alla fine della sessione. Tutte le entità e gli oggetti creati in fase di inizializzazione qui vanno rimossi dalla memoria.

Il passaggio da una sessione di logica all’altra viene gestito automaticamente, è sufficiente richiamare la funzione di cambio stato al verificarsi di una determinata condizione.   
Questa funzione è implementata in base alla macro-logica utilizzata. Sarà supportata una logica a script (dove ogni sessione logica è identificata da uno script, per cambiare sessione è quindi sufficiente invocare la funziona passando come parametro il nome dello script che contiene la logica della nuova sessione.

# Logica e Interazione

Considerando la pipeline principale (2.5D), per la sua natura mantiene la componente 3D minima, ovvero solamente nella rappresentazione di oggetti ed entità (ad esempio player e NPC).  
Non esistendo quindi una geometria per identificare le collisioni, o eventi per le interazioni varie o cambi di visuale, tutta la parte interattiva viene governata dal modulo *Floor,* ovvero una mappa composta da un certo numero di settori (triangoli adiacenti) che contengono determinate informazioni. Queste informazioni permettono di:

* Calcolare le collisioni con la scena (un’entita con collisioni può trovarsi solamente sul Floor);
* Adattare automaticamente l’altezza delle entità con collisioni;
* Calcolare le impostazioni per la camera corrente (ogni settore calpestato dal player attiva una determinata visuale, e di conseguenza attiverà un certo background e maschera di profondità);
* Calcolare percorsi ottimali automaticamente (IA);
* Attivare determinati eventi.

Siccome i singoli settori potrebbero essere moltissimi per una singola scena, il singolo settore conterrà solo le informazioni geometriche di questo, mentre le impostazioni di visuale ed eventuali trigger in comune a più settori verranno memorizzate in un gruppo.  
La struttura di un Floor è la seguente (HPMS utilizzerà ovviamente un formato binario):

<?xml version="1.0" encoding="UTF-8"?>  
<room id="R01">  
 <sectorGroups size="1">  
 <sectorGroup id="G01">  
 <sectors size="3">  
 <sector id="S01" groupId="G01" x1="0.0" x2="0.0" x3="0.0" y1="0.5" y2="0.5" y3="0.5" z1="1.0" z2="1.0"  
 z3="1.0">  
 <perimetralSides size="2">  
 <perimetralSide idx1="0" idx2="1"/>  
 <perimetralSide idx1="1" idx2="2"/>  
 </perimetralSides>  
 </sector>  
 <sector id="S02" groupId="G01" x1="0.0" x2="0.0" x3="0.0" y1="0.5" y2="0.5" y3="0.5" z1="1.0" z2="1.0"  
 z3="1.0"/>  
 <sector id="S03" groupId="G01" x1="0.0" x2="0.0" x3="0.0" y1="0.5" y2="0.5" y3="0.5" z1="1.0" z2="1.0"  
 z3="1.0"/>  
 </sectors>  
 </sectorGroup>  
 </sectorGroups>  
 <triggers size="2">  
 <trigger>  
 <conditions size="1">  
 <call functionName="entity\_on\_sector\_group">  
 <params size="2">  
 <param sVal="PLAYER" type="0"/>  
 <param sVal="G01" type="0"/>  
 </params>  
 </call>  
 </conditions>  
 <actions size="1">  
 <call functionName="incr\_decr\_entity\_param">  
 <params size="3">  
 <param sVal="PLAYER" type="0"/>  
 <param sVal="HP" type="0"/>  
 <param iVal="-1" type="1"/>  
 </params>  
 </call>  
 </actions>  
 </trigger>  
 </triggers>  
</room>

Con questa configurazione di esempio, quando il giocatore entrerà in collisione con uno dei triangoli appartenenti al gruppo *G1*, cambierà la visuale, settando la camera con le coordinate nuove, e l’immagine di background e maschera di profondità definite rispettivamente in *backgroundImage* e *depthMask.* Se inoltre la variabile custom *GOT\_ITEM\_AMULETE* è false, verrà mostrato un messaggio sullo schermo, e il parametro HP del giocatore verrà decrementato di 1 unità ogni secondo (1000ms).Tutto questo è definito nei blocchi *triggers*, ovvero azioni che si compiono al verificarsi di una determinata condizione. Nei triggers si possono invocare una serie funzioni booleane dello script di scena, se una di queste funzioni restituisce *true,* allora vengono eseguite tutte le funzioni di azione. Le azioni possono essere ripetute in loop fin tanto che l’entità si trova in quel gruppo, oppure solamente la prima volta appena ci entra.

# Runtime

Il gameplay verrà realizzato interamente in LUA, per permettere di modificare e creare nuovi scenari di gioco senza modificare l’eseguibile.

Ogni scenario ha uno script associato, la struttura seguente permette di creare una semplice scena, con una stanza interattiva, una entità animata, immagine di background e maschera di profondità:

-- Scene script.  
scene = {  
 name = "Scene\_01",  
 version = "1.0.0",  
 mode = "R25D",  
 quit = *false*,  
 setup = **function**(world, camera)  
 -- Init function callback.  
 e = hpms.make\_entity('data/models/M01.hmdat')  
 b = hpms.make\_background('data/textures/B01\_B.png')  
 d = hpms.make\_depth\_mask('data/textures/B01\_D.png')

r = hpms.make\_room('data/rooms/R01.hrdat')

e.position = hpms.vec3(0, 0, 0)  
 e.rotation = hpms.from\_axis(0, 0, 0, 1)  
 e.scale = hpms.vec3(0.2, 0.2, 0.2)  
  
 camera.position = hpms.vec3(3.5, 1, -3)  
 camera.rotation = hpms.vec3(0, hpms.to\_radians(-120), 0)  
 world.ambient\_light = hpms.vec3(1, 1, 1)

hpms.add\_entity\_to\_world(e, world)  
 hpms.add\_picture\_to\_world(b, world)  
 hpms.add\_picture\_to\_world(d, world)  
 **end**,  
 input = **function**(key)  
 -- Input function callback.  
 **if** key ~= *nil* **then  
 if** key.key == 'ESC' **then** scene.quit = *true* **end  
 end  
 end**,  
 update = **function**(world, camera)  
 -- Update loop function callback.

hpms.room\_interact\_entity(r, e)

hpms.room\_update\_view(r, camera)

hpms.room\_update\_background(r, b)

hpms.room\_update\_depth(r, d)  
 **end**,  
 cleanup = **function**()  
 -- Close function callback.  
 hpms.delete\_entity(e)  
 hpms.delete\_background(b)  
 hpms.delete\_depth\_mask(d)

hpms.delete\_room(r)  
 **end**}

# Editor

L’editor verrà sviluppato in Java per velocizzare i tempi di realizzazione della GUI. L’applicativo permetterà di:

* Caricare la walkmap e le camere da un file esportato da blender;
* Caricare le immagini degli sfondi e le maschere di profondità;
* Poter visualizzare un’anteprima delle varie inquadrature con il proprio sfondo;
* Poter aggiuingere gli oggetti 3D della scena;
* Poter aggiungere comportamenti ai vari oggetti o alla scena stessa, tramite script LUA, usando un sistema di condizione/azione;
* Poter importare i vari modelli 3D e visualizzarne l’anteprima;
* Poter esportare dal progetto un HPAK contenente:
  + Un XML/JSON/BIN che descrive la scena (incluse camere, settori, scripts ecc…);
  + Le immagini ed eventuale audio;
  + I modelli 3D (che verranno prima convertiti nel formato binario usando una routine esterne scritta in C++, che importerà i modelli nelle strutture usate dal runtime e le serializzerà in relativi file binari).
* Poter salvare tutto il lavoro in formato *.hproj*