Ray-casting, engine 3D e videogame

L'articolo parla del ray-casting, tecnica usata per generare ambienti tridimensionali utilizzabili per giochi in soggettiva. Saranno illustrati un motore grafico e un programma dimostrativo realizzati applicando le tecniche descritte.

di Antonino Calderone

I motori grafici basati sull'algoritmo di ray-casting un tempo erano molto popolari. Il celebre gioco della ID Software Wolfenstein 3D, basato su uno di questi motori grafici, era in grado di girare in maniera fluida su macchine dotate di processori Intel 286. Col passar del tempo, e con l'aumento delle capacità computazionali, sono stati rimpiazzati da soluzioni più raffinate, che si avvantaggiano delle CPU di nuova generazione e di GPU dedicate. Oggi, chi volesse cimentarsi nello sviluppo di motori grafici per l'elaborazione di immagini in tempo reale non può ignorare il supporto offerto da Direct3D, OpenGL e librerie equivalenti. Queste librerie forniscono l'indispensabile supporto per sfruttare in modo trasparente la potenza dei nuovi dispositivi hardware, ottenendo così prestazioni altrimenti impossibili.
Se per i PC di oggi non c'è più spazio per il ray-casting, rimane un crescente numero di calcolatori inscatolati in telefonini di nuova generazione, palmari, ed elettrodomestici di vario tipo, dove l'intrattenimento "collaterale" può certamente giovarsi di tecniche che pensavamo ormai tramontate. Lo testimonia il fatto che in rete si trovano parecchi "remake" in Java di giochi per telefonini e palmari che ne fanno ampio uso.
Per comprendere e valutare fino a che punto potevamo spingerci con questa tecnica ed ottenere risultati apprezzabili (in termini di qualità e fluidità), abbiamo implementato un piccolo motore grafico e realizzato con esso un programma dimostrativo (denominato *WinRaycast*), interattivo, che gira in ambiente Windows.

Antonino Calderone acalderone@infomedia.it

Lavora come software engineer presso Marconi SpA società del gruppo Ericsson. Socio fondatore della DigiFacta SW Engineering S.r.l. Si occupa principalmente di sistemi e protocolli per le telecomunicazioni, firmware, device driver, internals di sistemi operativi.

Partiamo dal ray-tracing

Data la relazione di parentela tra ray-casting e ray-tracing ci è comodo partire proprio da una breve descrizione di quest'ultimo.
Il ray-tracing è una tecnica usata per la generazione di immagini foto-realistiche che sfrutta le proprietà dei materiali che formano gli oggetti e i corpi di una scena, grazie all'applicazione di leggi della fisica e ai modelli matematici che si rifanno alla realtà. I programmi di *3d modelling* (come per esempio *POV-Ray* [2] o *blender* [3]) partendo dalla definizione di un mondo tridimensionale (*world model*), ovvero un insieme di oggetti nello spazio tridimensionale, calcolano una scena, secondo un processo denominato *rendering,* rispetto a un punto di osservazione (o *camera*) che si trova all'interno di questo spazio. L'algoritmo di ray-tracing adotta un criterio elementare: per ogni punto appartenente ad un oggetto vengono generati i raggi relativi alle sorgenti luminose.

> **I ray tracing è una tecnica usata per la generazione di immagini foto-realistiche**

Questi raggi di luce vengono riflessi e/o rifratti dagli oggetti definiti nella scena, e in ultimo colpiscono gli occhi dell'osservatore (o l'obiettivo della camera). Gli algoritmi implementati, denominati comunemente backward ray-tracing, in pratica funzionano con un percorso inverso, poiché gli unici raggi di luce che sono utili al rendering sono quelli che arrivano all'osservatore. Questo processo è di tipo ricorsivo e può ripetersi in modo da consentire il raggiungimento di un livello di dettaglio arbitrario.
Il ray-tracing, benché semplice ed efficace dal punto di vista dei risultati, richiede una potenza di elaborazione

notevole: anche in presenza di scene non complesse, il numero di raggi da calcolare è elevatissimo. Malgrado la potenza dei calcolatori di oggi, il ray-tracing non può essere usato per l'elaborazione di scene tridimensionali in tempo reale. I realtime 3d engine, specialmente quando usati nel campo dei videogame, fanno uso di "compromessi" che consentono un livello di realismo qualitativamente accettabile, mantenendo così prestazioni utili al rendering delle scene in tempo reale.

Il ray-casting

Le nozioni fondamentali usate dall'algoritmo di ray tracing possono essere impiegate con una tecnica denominata *ray-casting,* che al prezzo di alcuni "compromessi", consente l'elaborazione di immagini tridimensionali in tempo reale. Il termine *ray-casting* è talvolta usato per descrivere una parte del processo di rendering del *ray-tracing*.

L'algoritmo di ray casting che presentiamo fa uso di un *word model* notevolmente semplificato, denominato *occupancy grid*. L'occupancy grid è descritto completamente da una una mappa reticolare. Ogni cella della mappa identifica una regione di spazio tridimensionale ben precisa (generalmente chiamata blocco o cella).

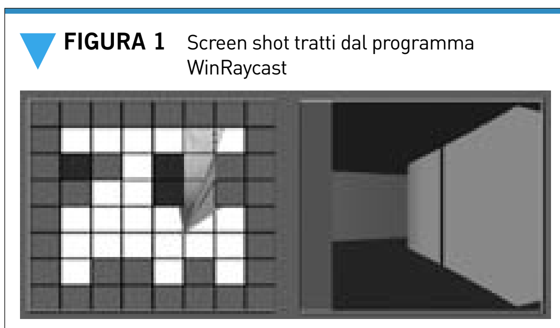
Immaginiamo di costruire una mappa come quella visibile in **Figura 1**. Questa mappa può essere implementata impiegando una matrice. Ogni elemento di questa matrice può rappresentare (per difetto) un blocco vuoto o non vuoto della mappa. Identifichiamo con *camera* (o *player*) un oggetto in grado di muoversi all'interno della mappa, nella regione di spazio occupata dai blocchi vuoti. Per descrivere la camera (il punto di osservazione) ci serviamo di un versore giacente sul piano della mappa, del quale sono significativi direzione, verso e punto di applicazione (coincidente con la sua posizione, in termini di coordinate cartesiane, entro i confini della mappa).

La direzione del versore è la bisettrice dell'angolo di visuale che caratterizza il *Field of Vision* o *FOV*. Nella **Figura 1** il FOV coincide con la regione di piano all'interno della quale sono rappresentati uscenti dalla camera un numero finito di linee (che possiamo denominare anche *ray o raggi*). L'ampiezza dell'angolo della visuale dipende da alcuni fattori: sperimentalmente si trova che un valore che produce buoni risultati in fase di rendering, per quest'angolo, è 60 gradi. Il numero di raggi dipende invece dalla risoluzione orizzontale del *piano di proiezione* (in altre parole la dimensione della porzione di schermo sulla quale viene visualizzata la scena).

A ogni raggio, dunque corrisponde una "striscia" verticale dell'area di rendering. Quando un raggio "incontra" un blocco non vuoto della mappa, ne viene calcolata la lunghezza, corrispondente alla distanza di quella porzione di blocco rispetto all'osservatore. Il rendering della striscia verticale del blocco "colpito" dal raggio produrrà un segmento con lunghezza inversamente proporzionale a quella del raggio stesso.

Partendo da questi elementi, e iterando per ogni raggio appartenente al FOV, è possibile costruire una vista prospettica simile a quella rappresentata nella parte destra di **Figura 1**.

FIGURA 1 Screen shot tratti dal programma WinRaycast



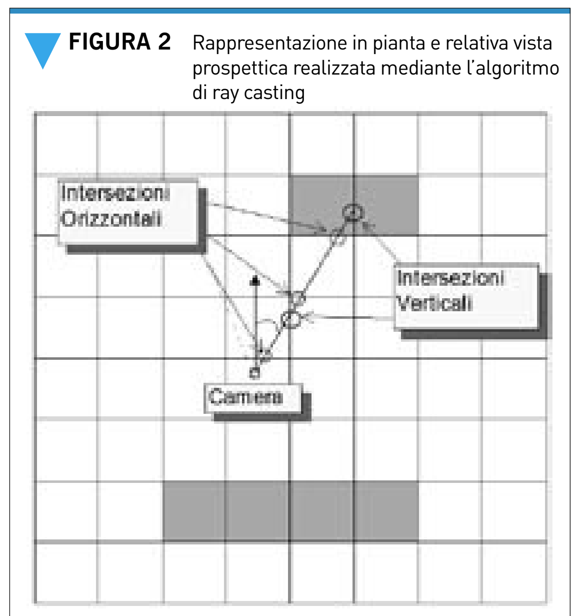
L'implementazione

L'implementazione di un motore grafico che applica l'algoritmo di *ray-casting* non richiede complesse trasformazioni nello spazio tridimensionale, ma solo alcune trasformazioni di geometria piana. Per questo motivo viene spesso definito un engine 2D e mezzo.

Il cuore dell'algoritmo si basa come già detto sulla emissione di un certo numero di raggi, in base alla dimensione orizzontale del piano sul quale viene effettuata la proiezione prospettica. Ognuno dei raggi emessi incrocerà il reticolo della mappa in più punti o intersezioni, che per comodità definiamo orizzontali e verticali, a seconda che siano prodotte da intersezioni con le linee orizzontali o verticali della mappa. Il raggio terminerà la propria corsa quando incontrerà il bordo di una cella non vuota o il limite invalicabile del confine della mappa stessa.

In **Figura 2** è rappresentato uno dei raggi emessi dal punto di osservazione. È possibile calcolare la prima intersezione orizzontale e la prima intersezione verticale

FIGURA 2 Rappresentazione in pianta e relativa vista prospettica realizzata mediante l'algoritmo di ray casting



tenendo conto che sono noti: l'angolo del raggio (chiamiamolo *alpha*) rispetto al versore della *camera*, quindi la sua *inclinazione* M, e l'origine dello stesso (ovvero le coordinate della camera x_p, y_p). L'inclinazione M del raggio è calcolata come M=*tan(alpha)*. Sappiamo dalla trigonometria che $M=(y-y_p)/(x-x_p)$ con *x,y* coordinate del punto di intersezione. Se chiamiamo x_c, y_i le coordinate della prima intersezione verticale e x_i, y_c quello della prima intersezione orizzontale, dall'equazione precedente si ricava $y_i=(x_c-x_p)*M+y_p$ e $x_i=(y_c-y_p)*M^{-1}+x_p$; x_c, y_c dipendono solo dalla posizione della camera e dalla dimensione delle celle della mappa. Il codice del **Listato 1**, tratto da *WinRaycast*, mostra l'implementazione dei metodi *first_horz_intersection* e *first_vert_intersection* della classe *RaycastEngine*, che applicano il criterio appena descritto per determinare la prima intersezione di ciascun raggio con le linee del reticolo della mappa.

Si noti che per determinare x_c, y_c, si controlla verso quale quadrante è rivolto l'osservatore. Ovviamente, abbiamo scelto un orientamento della mappa che ci

LISTATO 1 Calcolare la prima intersezione di ciascun raggio con il reticolo della mappa

```
void
RaycastEngine::
first_horz_intersection(WorldMap& wmap,
                        int ray,
                        const REAL& M1,
                        pair<REAL, REAL>& point) const throw()
{
  REAL xp = _camera.get_x();
  REAL yp = _camera.get_y();
  REAL yc, xi;

 if (ray>=_camera.deg180() && ray <_camera.deg360()) {
    yc = ((REAL)wmap.get_camera_cell_pos().second) * wmap.get_dy_cell();
  }
  else {
    yc = ((REAL)wmap.get_camera_cell_pos().second+1) * wmap.get_dy_cell();
  }

 xi = M1*(yc - yp) + xp;

  point.first = xi;
  point.second = yc;
}

void
RaycastEngine::
first_vert_intersection(WorldMap& wmap,
                        int ray,
                        const REAL& M,
                        pair<REAL, REAL>& point) const throw()
{
  REAL xp = _camera.get_x();
  REAL yp = _camera.get_y();
  REAL yi, xc;

  if (ray>=_camera.deg90() && ray <_camera.deg270()) {
    xc = ((REAL)wmap.get_camera_cell_pos().first) * wmap.get_dx_cell();
  }
  else {
    xc = ((REAL)wmap.get_camera_cell_pos().first+1) * wmap.get_dx_cell();
  }

 yi = M*(xc - xp) + yp;

  point.first = xc;
  point.second = yi;
}
```

LISTATO 2 Calcolare le intersezioni successive

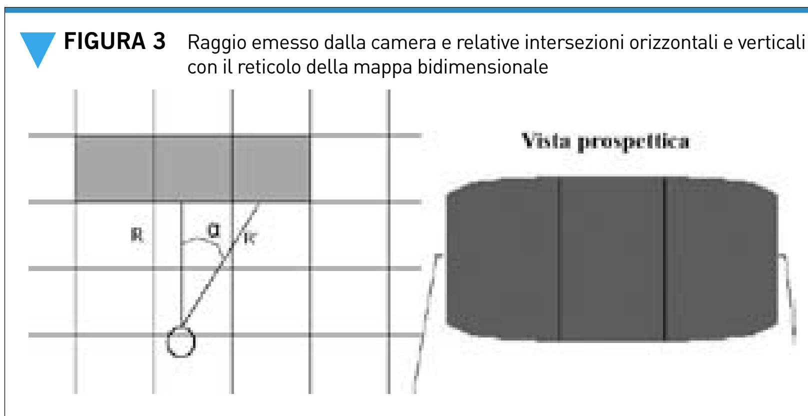
```
void
RaycastEngine::
vert_intersection(WorldMap& wmap,
                  const pair<REAL, REAL>& first_int,
                  int ray,
                  const REAL& M,
                  pair<REAL, REAL>& point) const throw()
{
  REAL xi, yi;

 if (ray>=_camera.deg90() && ray <_camera.deg270()) {
    yi = first_int.second - M * wmap.get_dx_cell();
    xi = first_int.first - wmap.get_dx_cell();
  }
  else {
    yi = first_int.second + M * wmap.get_dx_cell();
    xi = first_int.first + wmap.get_dx_cell();
  }

  point.first = xi;
  point.second = yi;
}
```

consente di pensare in termini di quadranti e di linee “orizzontali” o “verticali”.
La prima intersezione non assicura di incontrare un blocco “non vuoto”, quindi è necessario determinare le successive intersezioni. Dato che la dimensione delle celle e la pendenza del raggio non cambiano durante il calcolo delle intersezioni per lo stesso raggio, è possibile procedere ricorsivamente sommando alla precedente coordinata dell’intersezione l’ampiezza della cella, modulata dalla pendenza del raggio. Nel far questo si deve tenere conto del quadrante verso il quale è rivolta la camera. Il codice del **Listato 2** è usato dal motore grafico per il calcolo delle intersezioni successive alla prima.
Il metodo *horz_intersection*, usato per le intersezioni con le linee orizzontali, è duale al precedente.
Il processo prosegue ricorsivamente finché non si interseca una cella della mappa non vuota (o si raggiunge il limite dei confini della mappa). Trovata la cella non vuota, se ne determina la distanza dalla camera. Ovviamente si otterranno (di norma) due intersezioni e quindi due distanze (una, nel caso di raggio parallelo alle linee del reticolo). Queste due grandezze vengono confrontate, e viene scelta quella inferiore (o l’unica determinata), poiché la porzione di cella più distante, dal punto di vista dell’osservatore, è coperta da quella più vicina. Determinata la distanza minima dall’osservatore, si traccia sul piano di proiezione, la “striscia di blocco” della cella (cioè se ne effettua il rendering), in modo che la stessa intersechi nel centro una linea immaginaria chiamata *centro di proiezione*. La lunghezza della striscia è calcolata in modo inversamente proporzionale alla distanza dall’osservatore. La dimensione della striscia dipende inoltre da un fattore di scala (una costante). Questa costante viene scelta tenendo in considerazione alcuni fattori: la dimensione del piano di proiezione (ovvero la risoluzione dell’immagine *renderizzata*), la relazione con la dimensione della mappa e quella di ciascuna cella, ma soprattuto il risultato visivo che si vuole ottenere. Come per la scelta dell’angolo di visuale, si è operato in modo empirico.
Per completare i conti, in ultimo, bisogna considerare la distorsione dovuta alla differenza introdotta dalla geometria dei blocchi. In **Figura 3** è illustrato il fenomeno. Dato che l’emissione dei raggi è radiale, i raggi periferici che incontrano una porzione di blocco sono più lunghi rispetto quelli centrali; questo si riflette per quanto detto sull’altezza della striscia di blocco renderizzata, creando una distorsione che può essere definita “proiettiva”. Questo fenomeno non è trascurabile, e va corretto. La distorsione può essere eliminata normalizzando i raggi rispetto quello a minor distanza R dall’osservatore: basti osservare, guardando all’immagine in **Figura 3**, che R=R’ cos α. Quindi cos α può essere usato come fattore correttivo.
Nella realizzazione dell’algoritmo di rendering, è conveniente considerare il *centro di proiezione* non fisso (per difetto è il centro del piano di proiezione) e farlo dipendere dall’*inclinazione* della camera. In pratica agendo su questo parametro si simula l’orientamento dell’osservatore verso l’alto o verso il basso (**Figura 4**).

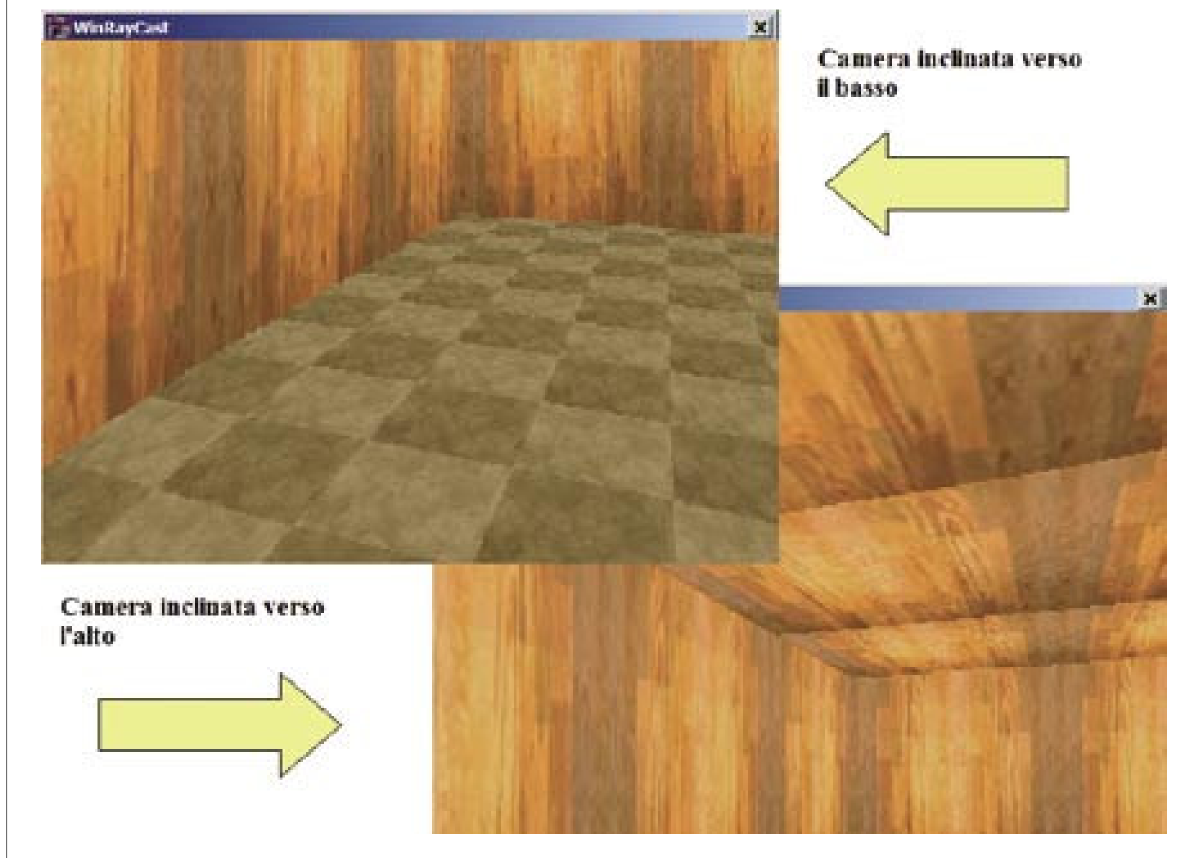
FIGURA 3 Raggio emesso dalla camera e relative intersezioni orizzontali e verticali con il reticolo della mappa bidimensionale



Texture mapping e shading

Il realismo di un motore grafico è notevolmente accentuato dal *texture mapping*. Fortunatamente, usando il ray-casting, l’algoritmo di “tessitura” dei blocchi è piuttosto semplice.
Si assume che siano definite le informazioni sulle texture e le bitmap

FIGURA 4 Effetto della distorsione proiettiva

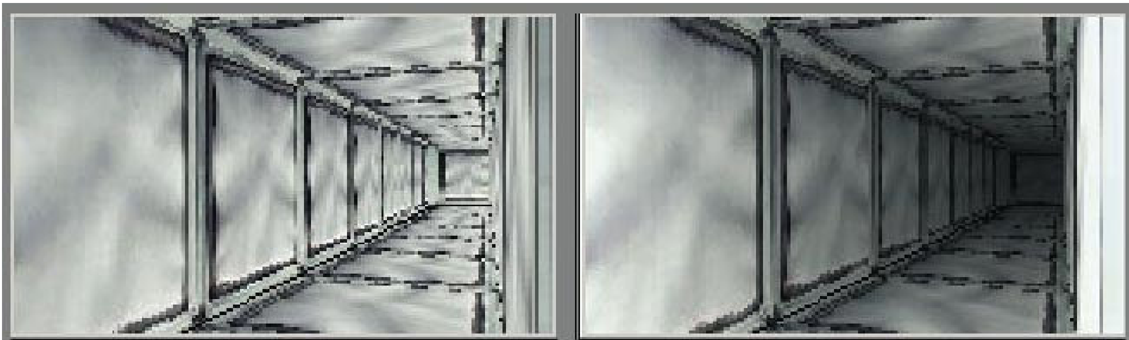


corrispondenti siano presenti in memoria. Le texture sono associate ai blocchi, usando una meccanismo di codifica. Per semplicità, in *WinRaycast* si è scelto di associare una bitmap per blocco, usando come chiave un numero intero rappresentabile con un singolo byte. Ovviamente sono possibili soluzioni diverse, che richiedono tuttavia maggiore complessità nella descrizione della singola cella della mappa. Durante la fase di rendering della striscia di un blocco, per ogni punto si determina quale sia la porzione di bitmap corrispondente, tenendo conto del fattore di scala, determinato sia dalla dimensione della striscia che della risoluzione della bitmap della texture.

Riportiamo nel **Listato 3** l'implementazione del metodo *shadingStretchBtl* della classe *RaycastEngine* che effettua il rendering di una striscia di blocco applicando il texture mapping.

Si noti che tra gli argomenti del metodo *shadingStretchBtl* esiste un parametro denominato *depth_param*. Questo numero dipende dalla distanza della striscia dall'osservatore, ed è utilizzato per realizzare lo *shading* (o sfumatura) dell'immagine durante la fase di rendering. Ciascun

FIGURA 5 Applicando lo shading, gli oggetti distanti appaiono più scuri



LISTATO 3 Rendering con il texture mapping

```
void
RaycastEngine::
shadingStretchBtl(HDC dest_hdc,
                  int x_dest, int y_dest,
                  int height_dest,
                  int x_source, int y_source,
                  int height_source, int width_source,
                  int max_visible_y_coord,
                  REAL depth_param, HBITMAP hBitmap)
{
  height_dest +=2;

  REAL step = REAL(height_source)/REAL(height_dest);

 REAL ys = REAL(y_source);
  int yd = y_dest-1;
  int max_yd = min(max_visible_y_coord, height_dest+y_dest);

  REAL Rcomp, Gcomp, Bcomp;

 if (yd<0) {
    ys += (-yd)*step;
    yd = 0;
  }

  TextureBuffer* tx_buf =
    getTextureMap(dest_hdc, hBitmap, width_source, height_source);

  while (yd < max_yd && ys < height_source) {
    COLORREF c = tx_buf->getPixel(x_source % width_source, int(ys) % height_source);

    if (depth_param<REAL(1.0)) {
      Rcomp = depth_param*( GetRValue(c) );
      Gcomp = depth_param*( GetGValue(c) );
      Bcomp = depth_param*( GetBValue(c) );

      DDrawPixel32(_videoBuffer, x_dest, yd, RGB(Rcomp, Gcomp, Bcomp));
    } // if
    else {
      DDrawPixel32(_videoBuffer, x_dest, yd, c);
    }

  ++yd;
    ys += step;
  }
}
```

pixel da proiettare viene scomposto nelle tre componenti *Red*, *Green*, *Blue* (*RGB*) del colore (operazione lecita lavorando in *true-color*). Ognuna di queste viene moltiplicata per un numero positivo razionale (*depth_param*) compreso tra 0 e 1. Il paramtro *depth_param* è calcolato in funzione della distanza: i punti appartenenti a strisce più distanti dall'osservatore hanno un *depth_param* più piccolo rispetto a quelle più vicine; ne risulta che gli oggetti distanti appaiano gradualmente più scuri rispetto quelli vicini. La differenza tra un'immagine prodotta senza l'applicazione del *depth shading* e la stessa immagine con lo *shading* abilitato è mostrata in **Figura 5**.

Rendering di ceil e floor

Il rendering del soffitto (*ceil*) e del pavimento (*floor*) arricchiscono ulteriormente il realismo nella scena.
L'associazione con le bitmap che devono essere usate per il rendering di soffitto e pavimento sono codificate anch'esse nella mappa. Anche in questo caso sarebbero possibili soluzioni diverse. Una bitmap per cella rimane a nostro parere un buon compromesso. Per il soffitto inoltre, si può decidere di operare o meno il rendering a seconda che si vogliano rappresentare "spazi all'aperto" o no.
Nell'immagine della **Figura 6**, lo sfondo usato per rap-

FIGURA 6 Il cielo nello sfondo è realizzato mediante un'immagine senza soluzione di continuità

presentare il cielo all'orizzonte usa una tecnica presa in prestito dal cinema. La bitmap del cielo viene proiettata sullo sfondo del piano di proiezione. Quest'immagine non rimane immobile, ma scorre in corrispondenza della rotazione della camera: se la camera ruota in senso orario, l'immagine scorre verso sinistra, e viceversa. Per gli sfondi conviene usare immagini senza soluzione di continuità.
Lo sfondo di solito rappresenta un soggetto "assai distante" dall'osservatore, come il cielo o un paesaggio, ed è posto dietro qualunque altro oggetto della scena.
Per illustrare la tecnica di rendering del pavimento (per il soffitto si procede simmetricamente) possiamo usare le seguenti considerazioni. La striscia di blocco (ovvero di muro verticale) occupa una porzione del piano di proiezione. I pixel immediatamente sopra la striscia appartengono al soffitto (*ceil*) e quelli immediatamente sotto appartengono al pavimento (*floor*), purché visibili nel *FOV*. Si tratta di stabilire dunque a quale cella della mappa essi corrispondano per determinare la texture da associare e la loro posizione relativa rispetto l'osservatore.
La porzione di codice del **Listato 4**, effettua il rendering del pavimento (*floor rendering*) mostrando più in dettaglio come questo venga implementato in *WinRaycast*.

Trasparenza, effetti animati, sprite e mip mapping

Come per il ray-tracing, anche l'algoritmo di ray-casting può essere eseguito in modo ricorsivo. È possibile per esempio utilizzare tecniche che combinano il rendering di più blocchi usando porzioni di superfici "trasparenti" ai raggi.
In *WinRaycast* il rendering delle scene è effettuato usando l'algoritmo del pittore: si disegnano prima le superfici lontane dall'osservatore e progressivamente quelle più vicine. Quando sono applicate le texture, si controlla se il colore di ciascun pixel sorgente della texture da applicare coincide con quello scelto come attributo di trasparenza. In caso affermativo, l'immagine prodotta da una precedente iterazione dell'algoritmo non viene sovrascritta in quel punto, ottenendo una sovrapposizione che denominiamo "trasparenza".
Sia il riferimento alle bitmap nella mappa, usate per il texture mapping, che il contenuto delle bitmap stesse, può essere modificato dinamicamente per ottenere animazioni, come l'effetto fuoco, la superficie dell'acqua nella vasca, le luci intermittenti usati in *WinRaycast* (**Figura 7**).
Il ray-casting non può essere usato per il rendering di oggetti complessi, ma in un gioco 3D che si rispetti,

LISTATO 4 Porzione di codice che effettua il rendering del pavimento

```
for (int floor_ray = _camera.get_slope();
     floor_ray<(ceil_bottom+center_of_projection);
     ++floor_ray)
{
  TextureBuffer* tx_buf;

  REAL delta_c = ceil_bottom-floor_ray;
  if (delta_c<=0.0) continue;

  REAL distanceToPtOnCeiling = floor_scaled_distort_LUT/delta_c;

  int x_picture = (_camera.cos(rel_ray)*distanceToPtOnCeiling) + camera_pos_x;
  int y_picture = (_camera.sin(rel_ray)*distanceToPtOnCeiling) + camera_pos_y;

  int dx_cell = wmap.get_dx_cell();
  int dy_cell = wmap.get_dy_cell();

  int floor_key = 0;

  int row = y_picture/dy_cell;
  int col = x_picture/dx_cell;

  if ((row<wmap.row_count()) && col<wmap.col_count() && row>=0 && col>=0) {
    int map_key = wmap[row][col];
    if (map_key & 0xff) continue;
    floor_key = (map_key & 0x00ff0000)>>16;
  }
  else {
    continue;
  }

  if (floor_key == 0xFF) {
    continue;
  }

  tx_buf = getTextureMap(video_hdc, wmap.get_bmp(floor_key), dx_cell, dy_cell);

  REAL shading_attr = _ceil_floor_shading_param/REAL(distanceToPtOnCeiling);
  COLORREF c = tx_buf->getPixel(x_picture % dx_cell, y_picture % dy_cell);

  int y = _camera.get_slope()+_camera.get_y_projection_res()-floor_ray;

  if (shading_attr>=1.0) {
    DDrawPixel32(_videoBuffer, ray, y, c);
  }
  else {
    REAL Rcomp = shading_attr*( GetRValue(c) );
    REAL Gcomp = shading_attr*( GetGValue(c) );
    REAL Bcomp = shading_attr*( GetBValue(c) );

    DDrawPixel32(_videoBuffer, ray, y, RGB(Rcomp, Gcomp, Bcomp));
  } //else
} // for
```

qualche avversario qua e là ci vuole, e gli avversari non possono essere rappresentati da blocchi squadrati.
Una tecnica alternativa, usata anche in giochi tra i quali il celebre Doom, è quella di simulare oggetti tridimensionali proiettando bitmap bidimensionali o *sprite*, che se sapientemente animati possono fornire un buon surrogato a basso costo. Uno sprite lo si può rappresentare come una bitmap proiettata su un piano, la cui *normale* abbia la direzione del versore della camera. Ovviamente deve implementare la trasparenza ed essere scalabile, in

FIGURA 7 L'animazione del fuoco è ottenuta selezionando dinamicamente una delle quattro bitmap di destra

modo da avere una risoluzione accettabile se osservato da vicino, quanto da lontano. Una sola bitmap per sprite poi non basta, quando si intendano simulare punti di osservazione da angolazioni distinte.
Gli sprite possono essere parzialmente non visibili nel caso in cui siano "nascosti" all'osservatore dalle pareti dei blocchi. In pratica per ogni porzione di blocco proiettata è necessario controllare se esiste uno sprite che interseca il raggio, e quando questo avviene, verificare se la distanza dall'osservatore è maggiore o minore a quella della striscia di blocco che si sta tracciando. Così facendo è possibile ottenere una corretta integrazione con l'ambiente.
Per migliorare la qualità di queste bitmap, e in genere di quelle usate per il texture mapping, si può ricorrere alla tecnica conosciuta come *mip mapping* (mip deriva dal latino *multim in parvo* ovvero "molto in poco"). Questa tecnica consente di risolvere il problema dell'*effetto mosaico* delle texture senza l'ausilio di complesse interpolazioni. Consiste nel memorizzare differenti copie della medesima bitmap prescalata a diverse risoluzioni. La texture da applicare viene selezionata dinamicamente in modo da usare la più idonea alle dimensioni dell'oggetto nello spazio tridimensionale, ovvero in base alla distanza dal punto di osservazione.

Conclusioni

WinRaycast è stato costruito in modo da mantenere il giusto equilibrio tra prestazioni e comprensibilità del codice. Questo progetto è nato con l'obiettivo di fornire un banco di prova per sperimentare le tecniche descritte nell'articolo. Il lettore che voglia approfondire l'argomento (magari per semplice curiosità) troverà giovamento dall'analisi del codice. Potrà, se lo vorrà, estendere, correggere o modificare il programma aggiungendo molte caratteristiche strutturali che non sono state implementate, aggiungendo, per esempio, un miglior supporto per la gestione delle informazioni della mappa, delle texture, la gestione degli sprite e altro ancora.
Il progetto *WinRaycast* è compilabile usando l'ambiente di Microsoft Visual Studio 6.0 e l'SDK delle DirectX (va bene una qualunque versione dalla versione 7 in avanti). Abbiamo usato le *DirectDraw* in sostituzione delle *dib section* [1] della *GDI,* impiegate invece in una prima implementazione, per ragioni prestazionali. Dopo l'installazione dell'SDK si consiglia di copiare da questo, i file *.h* e *.lib* in sostituzione e/o in aggiunta a quelli originali, installati con l'ambiente di sviluppo di Microsoft. La pratica potrebbe sembrare poco ortodossa, ma è stata più volte sperimentata con successo dall'autore.
Nel parlare di *WinRaycast* abbiamo tralasciato molti dettagli (come per esempio la gestione delle collisioni con l'ambiente circostante, la gestione di tastiera o del joystick, e altro ancora), ritenendo che tutto sommato, siano relegabili a un ruolo marginale rispetto al resto della trattazione, e comunque desumibili dal codice del progetto allegato.

Bibliografia & Riferimenti

[1] Charles Petzold – "*Programming Windows, 5th Edition*", Microsoft Press, 1998
[2] http://www.povray.org
[3] http://www.blender.org

CODICE ALLEGATO

ftp.infomedia.it **Raycast**