Corso di *Tecniche Avanzate per la Grafica*

Render-to-texture

Docente:

Massimiliano Corsini

Laurea Specialistica in Informatica

Facoltà di Scienze MM. FF. NN.
Università di Ferrara

Overview

- Render-to-texture
  - glReadPixels(...), glDrawPixels(...)
  - pBuffer (pixels-buffer)
  - Framebuffer Object (FBO)

Introduzione

- Per ***render-to-texture*** si intende la creazione di texture maps “al volo”, ossia durante l’esecuzione dell’applicazione grafica.

- Tipicamente si ha un loop di questo tipo:
  1. Rendering di un immagine.
  2. Si crea una texture da tale immagine.
  3. Si utilizza la texture così ottenuta.

Applicazioni delle texture dinamiche

- Impostors
- Dynamic Cube / Environment map generation
- Dynamic Normal map generation
- Dynamic Volumetric Fog
- Procedural Texturing
- Dynamic Image Processing

Collo di bottiglia

- Sempre riferendosi al ciclo base:
  1. Rendering di un immagine.
  2. **Si crea una tessitura da tale immagine.**
  3. Si utilizza la texture così ottenuta.
- Si può evidenziare che il punto 2 presenta un collo di bottiglia, ossia il passare da ciò che si è appena renderizzato alla texture.

Performances

- Come passare dall'immagine renderizzata alla texture?
  - glReadPixels() → glTexImage*() ?
    - Lento.
  - glCopyTexImage*()
    - Meglio.
  - glCopyTexSubImage*()
    - Ancora meglio.
  - Rendering direttamente sulla texture
    - Elimina I passaggi di memoria → potenzialmente ottimo

- Il rendering diretto su texture non fa parte del core dell'OpenGL ma è un'estensione ARB.

- Estensioni in gioco:
  - WGL_ARB_extensions_string
  - WGL_ARB_render_texture
  - WGL_ARB_pbuffer
  - WGL_ARB_pixel_format

- Dai modelli GeForce (28.40 driver) in poi queste estensioni sono sempre disponibili

Render-to-texture (pbuffer)

- pBuffer sta per pixels-buffer
- Idea è che il pbuffer è legato ad una texture
- Innanzitutto si deve creare una texture
- Poi si deve creare una “Render Texture” (ossia un pbuffer)
- Loop:
  - Settare il pbuffer come il rendering target corrente
  - Renderizzare l’immagine desiderata
  - Settare la finestra il rendering target corrente
  - Collegare il pbuffer alla texture
  - Usare la texture come qualsiasi altra texture
  - Scollegare il pbuffer dalla texture
- Clean Up

Creare la texture

- Analogamente a come si crea un normale texture – l'unica differenza è che i dati della texture non sono specificati
-
```
glGenTextures(1, &render_texture);
glBindTexture(GL_TEXTURE_2D, render_texture);
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR);
glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_MAG_FILTER, GL_LINEAR );
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,
    GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
    GL_CLAMP_TO_EDGE );
```

Creazione del pbuffer

- Step 1: Ottenere un ***device context*** (DC) valido
- Step 2: Trovare un formato dei pixel adatto
- Step 3: Creare un pbuffer con tale formato
- Step 4: Creare un DC per il pbuffer
- Step 5: Creare un rendering context (RC) OpenGL ed associarlo al pbuffer

Ottenere il Device Context (DC)

- ***Step 1: ottenere un device context (DC) valido*** significa ottenere un ID (handle) al dispositivo di rendering.
- Nel caso che una finestra OpenGL sia già stata inizializzata è sufficiente utilizzare il comando:
  - `HDC hdc = wglGetCurrentDC();`

Scegliere il formato dei pixels

- ***Step 2: scegliere il formato dei pixels***
- In pratica, si deve settare quali attributi desideriamo che il nostro pbuffer abbia; a questo punto si può richiedere all'hardware una lista dei formati che hanno gli attributi richiesti (ogni formato è specificato da un numero intero).
- Per attributi si intende il numero di bit delle varie componenti: ad esempio possiamo volere 32-bit x colore (RGBA) e 24 bits per l'informazione di depth.
- La funzione che restituisce i formati disponibili con gli attributi richiesti è **`wglChoosePixelFormatARB()`**.
- Gli attributi richiesti devono essere memorizzati in un vettore come coppia attributo-valore.

Creare il pbuffer

- ***Step 3: creare il pbuffer***
- Una volta trovato il formato dei pixel si può procedere alla creazione vera e propria del pbuffer con il comando:
  - **`HPBUFFERARB hbuffer = wglCreatePbufferARB( hdc, format, iwidth, iheight, iattribs );`**
  - **`hdc`** è il device context ottenuto al passo 1
  - **`format`** è l'intero che identifica il formato dei pixel scelto
  - **`iattribs`** è un codice per ottenere ulteriori settaggi

Creazione di un DC per il pbuffer

- ***Step 4: creazione di un DC per il pbuffer***
- Per creare un device context per il pbuffer si utilizza la chiamata:
  - **`HDC hpbufdc = wglGetPbufferDCARB( hbuffer );`**
  - **`hpbufdc`** è un handle al DC creato

Creazione di un rendering context (RC) per il pbuffer

- ***Step 5: associare un rendering context (RC) OpenGL al pbuffer***
- E' il passo finale e consiste nel creare un contesto di rendering OpenGL (RC) ed associarlo al pbuffer creato al passo 4:
  - `pbufglctx = wglCreateContext(hpbufdc);`

PBuffer Binding

- Una volta creato ed inizializzato il pbuffer lo si può utilizzare in qualsiasi momento per effettuare un rendering off-screen.
- Prima di utilizzare il pbuffer si deve renderlo il contesto di rendering OpenGL attivo:
  - `wglMakeCurrent( hpbufdc, pbufglctx );`
- Per ripristinare il contesto di rendering originale si deve chiamare di nuovo la wglMakeCurrent() con il DC e l'RC corrispondenti alla finestra su cui stavamo visulizzando la nostra applicazione.

PBuffer Binding

- Dopo aver effettuato il ***bind*** della texture che vogliamo collegare al pbuffer
- Si deve effettuare il binding del pbuffer:
  - **`BOOL wglBindTexImageARB(HPBUFFERARB hPbuffer, int iBuffer)`**
  - **`iBuffer`** determina quale buffer viene usato per il rendering sulla texture (WGL_FRONT_LEFT_ARB, WGL_BACK_LEFT_ARB)
- Prima di effettuare un nuovo rendering sulla texture è necessario scollegare il pbuffer dalla texture
  - **`BOOL wglReleaseTexImageARB ( HPBUFFERARB hPbuffer, int iBuffer)`**

Render-to-Texture

- Il meccanismo render-to-texture permette di specificare su quale parte della texture vogliamo fare il rendering:
  - Un determinato livello di una texture con mipmap
  - Una specifica faccia di una cube map texture

- Per specificare la parte della texture possiamo usare la funzione wglSetPbufferAttribARB(…):
  - **`BOOL wglSetPbufferAttribARB (HPBUFFERARB hPbuffer, const int *piAttribList)`**

Non-power-of-two Textures

- OpenGL supporta textures di risoluzione 2^m x 2^n
- Tuttavia, in molti casi farebbe comodo poter maneggiare texture “Non-Power-of-Two” (ossia la larghezza o l’altezza non è una potenza del due)
  - Ad esempio se vogliamo creare uno sfondo di uno schermo (es. 800x600)
- Quando occorre è possibile gestire texture di dimensione arbitraria grazie ad opportune estensioni (ARB_texture_non_power_of_two):
  - Le coord. texture sono mappate diversamente dal solito:
    - s,t range: [0,Width], [0,Height] respectively instead of usual [0,1], [0,1] range.
  - Mipmap non è supportato
  - BORDER or REPEAT texture wrap non è supportato

CleanUp

- Per rilasciare le risorse di memoria utilizzate dal pbuffer si deve effettuare essenzialmente tre operazioni:

1. Cancellare il RC del pbuffer (hpbufglrc)
2. Rilasciare il DC del pbuffer (hpbufdc)
3. Distruggere il pbuffer

```
wglDeleteContext(hpbufglrc);
wglReleasePbufferDCARB(hbuf,
  hpbufdc);
wglDestroyPbufferARB(hbuf);
```

Framebuffer Object (FBO)

PBuffer Review

- I ***Frambuffer Object*** sono la tecnologia più moderna per fare render-to-texture
- Abbiamo appena visto i pixels-buffer (pbuffer)
  - Estensione WGL_ARB_pbuffer
  - Disegnati per fare off-screen rendering come se fossero delle finestre (ma invisibili)
  - Lavorano su un insieme predefinito di formati

Svantaggi dei pbuffer

- Ogni pbuffer ha il suo contesto OpenGL
  - Difficile da maneggiare, fonte di bugs
- Switching tra pbuffers è lento
  - wglMakeCurrent() è un'operazione costosa
- Each pbuffer ha il suo depth buffer, il suo stencil buffer, i suoi auxiliary buffers
  - Non posso condividere un depth buffer tra diversi pbuffers → non c'è un'ottimizzazione delle risorse di memoria

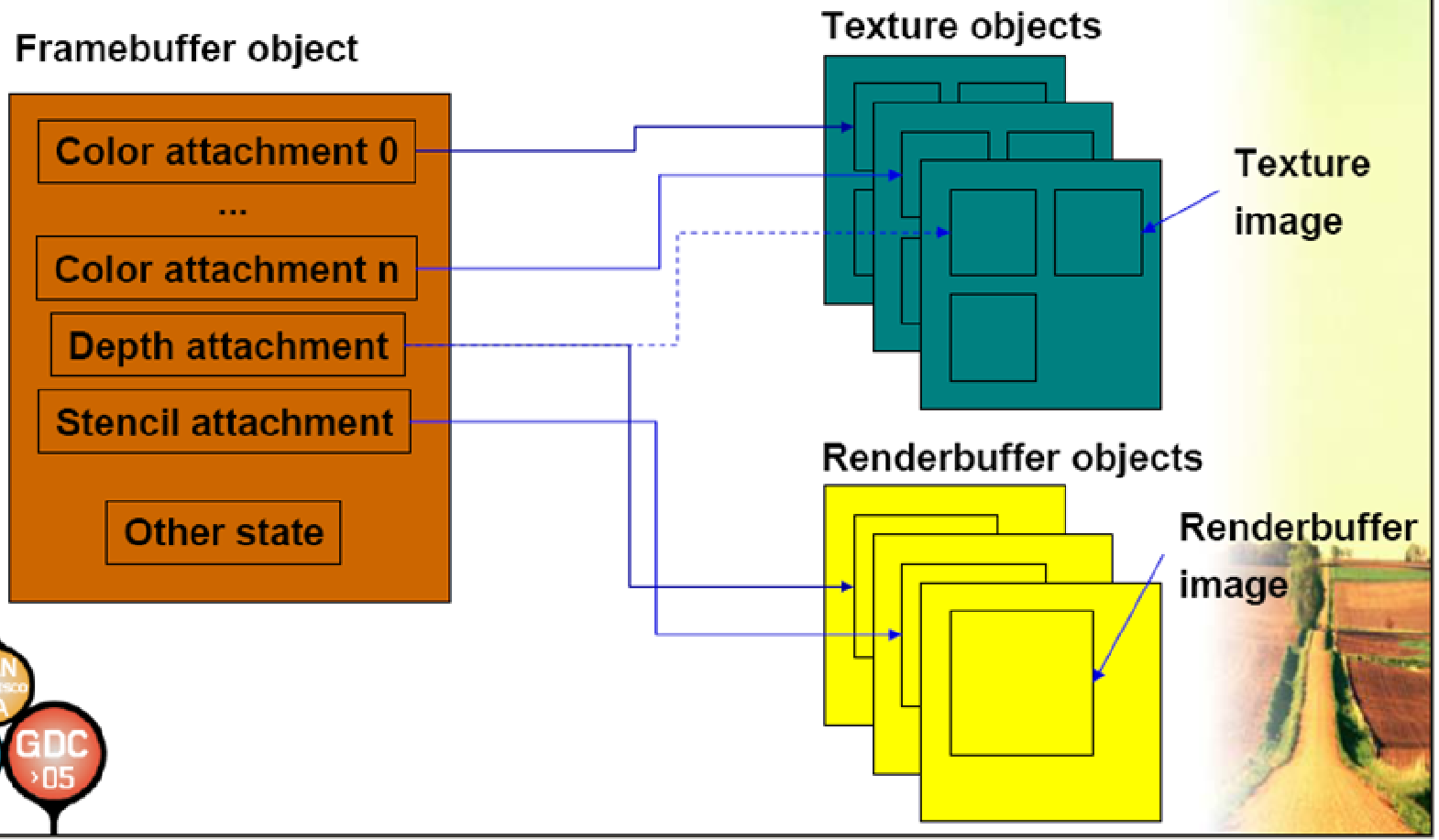
Framebuffer Objects

- I framebuffer sono una collezione di buffer logici (depth buffer, color buffer, aux buffer, ..)
- Estensione in gioco:
  - EXT_framebuffer_object
- Forniscono un meccanismo per fare rendering su un target diverso da una finestra del sistema operativo → indipendenti dal sistema operativo
- I target su cui fare rendering possono essere delle texture oppure dei buffer off-screen chiamati ***renderbuffers images***

GameDevelopers Conference

NVIDIA

Framebuffer Object Architecture



SAN FRANCISCO CA MAR 7-11 GDC 05

Terminologia

- ***Renderbuffer image:*** matrice di pixels. Parte di un renderbuffer object.
- ***Framebuffer-attachable image:*** matrice di pixels che può essere associata (attached) ad un framebuffer. SIa una texture che una renderbuffer image sono immagini collegabili ad un framebuffer object (framebuffer-attachable images).
- ***Attachment point:*** servono per referenziare framebuffer-attachable image di un framebuffer. C'è ne è uno per ogni depth buffer, color buffer e stencil buffer di un framebuffer.
- ***Attach:*** l'atto di connettere, di associare, un oggetto ad un altro (come la "bind").

Framebuffers e Renderbuffers

- Quindi abbiamo due oggetti OpenGL:
  - **Framebuffer Objects**
    - Contengono un insieme di framebuffer-attachable images (e.g. depth buffer, stencil buffer, color buffer, aux buffers)
    - Più alcune informazioni di stato (ad esempio dove l'OpenGL sta inviando il suo output)
  - **Renderbuffer Objects**
    - Immagini semplici
      - NO mipmaps, cubemap, ecc.
    - Immagazzinano i pixels provenienti in ouptut dal rendering
    - Non possono essere utilizzati come textures

Framebuffers e Renderbuffers

- Quando un framebuffer object è utilizzato le sue immagini “attached” sono utilizzate nelle operazioni per frammento.

Framebuffer Object API

- `void GenFramebuffersEXT(sizei n, uint *framebuffers);`
- `void DeleteFramebuffersEXT(sizei n, const uint *framebuffers);`
- `boolean IsFramebufferEXT(uint framebuffer);`
- `void BindFramebufferEXT(enum target, uint framebuffer);`
- `enum CheckFramebufferStatusEXT(enum target);`

Framebuffer Object API

- `void FramebufferTexture1DEXT(enum target, enum attachment, enum textarget, uint texture, int level);`
- `void FramebufferTexture2DEXT(enum target, enum attachment, enum textarget, uint texture, int level);`
- `void FramebufferTexture3DEXT(enum target, enum attachment, enum textarget, uint texture, int level, int zoffset);`
- `void FramebufferRenderbufferEXT(enum target, enum attachment, enum renderbuffertarget, uint renderbuffer);`
- `void GetFramebufferAttachmentParameterivEXT(enum target, enum attachment, enum pname, int *params);`
- `void GenerateMipmapEXT(enum target);`

Managing FBOs e Renderbuffer

- Creare e distruggere FBOs (e Renderbuffers) è semplice – è molto simile a gestire una texture standard
  - `void GenFramebuffersEXT(sizei n, uint *framebuffers);`
  - `Void DeleteFramebuffersEXT(sizei n, const uint *framebuffers);`
  - `void BindFramebufferEXT(enum target, uint framebuffer);`
  - `Boolean IsFramebufferEXT (uint framebuffer);`
- È possibile controllare se un certo handle è un framebuffer object con il comando (raramente utile):
  - `Boolean IsFramebufferEXT(uint framebuffer);`

FBO Binding

- Come per una texture standard prima dell'utilizzo deve essere eseguito una bind sul framebuffer object:
  - **`void BindFramebufferEXT(enum target, uint framebuffer);`**
  - **`target`** deve valere FRAMEBUFFER_EXT
  - Tutte le operazioni openGl vengono eseguiti sugli "attachment" del framebuffer object

Attach di una Texture ad un Framebuffer

- **`void FramebufferTexture2DEXT(enum target, enum attachment, enum textarget, uint texture, int level);`**

- **`target`** deve valere obbligatoriamente FRAMEBUFFER_EXT
- **`attachment`** può valere **`COLOR_ATTACHMENT0_EXT, COLOR_ATTACHMENT1_EXT, ..., COLOR_ATTACHMENTn_EXT, DEPTH_ATTACHMENT_EXT, STENCIL_ATTACHMENT_EXT`**
- **`textarget`** può assumere i valori **`TEXTURE_2D, TEXTURE_RECTANGLE, TEXTURE_CUBE_MAP_POSITIVE_X ecc.`**
- **`Level`** è il livello di mipmap della texture su cui eseguire *l'attach*
- **`texture`** è l'handle della tessitura
  - Se vale 0 viene eseguito il detach

Renderbuffer API

- `void GenRenderbuffersEXT(sizei n, uint *renderbuffers);`
- `void DeleteRenderbuffersEXT(sizei n, const uint *renderbuffers);`
- `boolean IsRenderbufferEXT(uint renderbuffer);`
- `void BindRenderbufferEXT(enum target, uint renderbuffer);`
- `void RenderbufferStorageEXT(enum target, enum internalformat, sizei width, sizei height);`
- `void GetRenderbufferParameterivEXT(enum target, enum pname int *params);`

Attach di un Renderbuffer ad un Framebuffer

- **`void FramebufferRenderbufferEXT(enum target, enum attachment, enum renderbuffertarget, uint renderbuffer);`**

- **`target`** deve valere FRAMEBUFFER_EXT
- *attachment può assumere i valori:*
  - COLOR_ATTACHMENT0_EXT
  - ...
  - COLOR_ATTACHMENTn_EXT
  - DEPTH_ATTACHMENT_EXT
  - STENCIL_ATTACHMENT_EXT
- **`renderbuffertarget`** deve valere RENDERBUFFER_EXT
- **`renderbuffer`** è l'id del Renderbuffer

Framebuffer Completeness

- Un Framebuffer Object si dice ***completo*** se ha le seguenti caratteristiche (vedetelo come sinonimo di ben inizializzato):
  - I formati delle texture hanno senso relativamente ai punti di attachment (ad esempio una depth texture non può essere attached ad un buffer di colore)
  - Tutte le immagini “attached” hanno la stessa altezza e la stessa larghezza
  - Tutte le immagini “attached” a COLOR_ATTACHMENT0_EXT COLOR_ATTACHMENTn_EXT devono avere lo stesso formato
- Se il Framebuffer Object non è ***complete*** la prima glBegin genera un errore di tipo INVALID_FRAMEBUFFER_OPERATION

Stato di un Framebuffer

- **`enum CheckFramebufferStatusEXT(enum target);`**
- Ci si dovrebbe sempre accertare delle condizioni del Framebuffer prima di utilizzarlo
- La funzione ritorna un codice che indica se e perchè il Framebuffer non è completo:
  - **`FRAMEBUFFER_COMPLETE`**
  - **`FRAMEBUFFER_INCOMPLETE_ATTACHMENT`**
  - **`FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT`**
  - **`FRAMEBUFFER_INCOMPLETE_DUPLICATE_ATTACHMENT`**
  - **`FRAMEBUFFER_INCOMPLETE_DIMENSIONS_EXT`**
  - **`FRAMEBUFFER_INCOMPLETE_FORMATS_EXT`**
  - **`FRAMEBUFFER_INCOMPLETE_DRAW_BUFFER_EXT`**
  - **`FRAMEBUFFER_UNSUPPORTED`**
  - **`FRAMEBUFFER_STATUS_ERROR`**

Performance Tips

- Bisogna evitare di creare e distruggere i FBOs ad ogni frame
- Bisogna evitare di modificare le textures del FBO usate come target di rendering (ad esempio TexImage, CopyTexImage, ecc.)

Usi tipici

- Il Framebuffer Object permette diverse modalità di switching della destinazione del rendering (in ordine di performances)

- **FBO Multipli**
  - Si crea un Framebuffer Object per ogni texture su cui voglio effettuare un rendering
  - Switch con **`BindFramebuffer()`**
  - Veloce il doppio che **`wglMakeCurrent()`**
- **FBO singolo, più texture attachments**
  - Le textures devono avere lo stesso formato e la stessa dimensione
  - Si usa **`FramebufferTexture()`** per “switchare” da una texture all’altra
- **FBO singolo, più texture attachments**
  - Si effettua l’attach a diversi color attachments
  - Si usa glDrawBuffer() per fare switch tra i diversi color attachments

FBO (esempio)

```
GLuint fb, depth_rb, tex;

// creazioni oggetti in gioco
glGenFramebuffersEXT(1, &fb);          // frame buffer
glGenRenderbuffersEXT(1, &depth_rb);   // render buffer
glGenTextures(1, &tex);                // texture
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

// initializzazione della texture
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
             GL_RGBA, GL_UNSIGNED_BYTE, NULL);
// setting dei parametri della texture...

// la texture viene “attached” al framebuffer color buffer
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
  GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, tex, 0);
```

FBO (esempio)

```
// initializazzione DEPTH renderbuffer
glBindRenderbufferEXT(GL_RENDERBUFFER_EXT, depth_rb);
glRenderbufferStorageEXT(GL_RENDERBUFFER_EXT,
    GL_DEPTH_COMPONENT24, width, height);

// attach renderbuffer to framebuffer depth buffer
glFramebufferRenderbufferEXT(GL_FRAMEBUFFER_EXT,
   GL_DEPTH_ATTACHMENT_EXT, GL_RENDERBUFFER_EXT,
   depth_rb);

// render to the FBO (BINDING)
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

// draw something here...

// ripristino il rendering verso la window
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);

// utilizzo la texture sulla quale ho appena effettuato
   un rendering
glBindTexture(GL_TEXTURE_2D, tex);
```

Domande?