The OpenGL Framebuffer Object Extension

Simon Green
NVIDIA Corporation
Overview

• Why render to texture?
• P-buffer / ARB render texture review
• Framebuffer object extension
• Examples
• Future directions
Why Render To Texture?

- Allows results of rendering to framebuffer to be directly read as texture
- Better performance
  - avoids copy from framebuffer to texture \((\text{glCopyTexImage2D})\)
  - uses less memory – only one copy of image
  - but driver may sometimes have to do copy internally
    - some hardware has separate texture and FB memory
    - different internal representations
- Applications
  - dynamic textures – procedurals, reflections
  - multi-pass techniques – anti-aliasing, motion blur, depth of field
  - image processing effects (blurs etc.)
  - GPGPU – provides feedback loop
WGL_ARB_pbuffer

- Pixel buffers
- Designed for off-screen rendering
  - Similar to windows, but non-visible
- Window system specific extension
- Select from an enumerated list of available pixel formats using
  - ChoosePixelFormat()
  - DescribePixelFormat()
Problems with PBuffers

- Each pbuffer usually has its own OpenGL context
  - (Assuming they have different pixel formats)
  - Can share texture objects, display lists between pbuffers using wglShareLists()
  - Painful to manage, causes lots of bugs
- Switching between pbuffers is expensive
  - wglMakeCurrent() causes context switch
- Each pbuffer has its own depth, stencil, aux buffers
  - Cannot share depth buffers between pbuffers
WGL_ARB_render_texture

- Allows the color or depth buffer of a pbuffer to be bound as a texture
  - `BOOL wglBindTexImageARB(HPBUFFERARB hPbuffer, int iBuffer`
  - `BOOL wglReleaseTexImageARB(HPBUFFERARB hPbuffer, int iBuffer)`

- Window system specific
  - GLX version of specification was never defined
  - MacOS X - APPLE_pixel_buffer

- Texture format is determined by pixel format of pbuffer

- Portable applications need to create a separate pbuffer for each renderable texture
Pbuffer Tricks

• The front and back buffers of a double-buffered pbuffer can be bound as separate textures
  
  ```
  glDrawBuffer(GL_FRONT); // draw to front
  glDrawBuffer(GL_BACK);  // draw to back
  // bind front and back buffers as textures
  wglBindTexImageARB(pbuffer, WGL_FRONT_LEFT_ARB);
  wglBindTexImageARB(pbuffer, WGL_BACK_LEFT_ARB);
  ```

• This gives you two buffers that you can switch between without incurring context switching cost

• On systems that support multiple render targets (ARB_draw_buffers) you can also use AUX buffers
Render To Texture And Anti-Aliasing

- Render to texture doesn’t work with multi-sample anti-aliasing
  - current texture hardware isn’t capable of reading from a multi-sampled buffer
  - could be implemented in driver using copy
- Common problem with post-processing effects in games
- Solution: create a normal multi-sampled pbuffer, and use `glCopyTexImage2D` to copy from this to a texture
  - the copy performs the down-sampling automatically
- Also possible to do your own super-sample anti-aliasing in the application
  - much more expensive than multi-sampling
Anti-Aliasing with Post Processing

Without AA

With AA
The Framebuffer Object Extension

- Specification finally published!
- Available in beta drivers from NVIDIA
Framebuffer Object Advantages

- Only requires a single OpenGL context
  - switching between framebuffers is faster than switching between pbuffers (wglMakeCurrent)
- No need for complicated pixel format selection
  - format of framebuffer is determined by texture or renderbuffer format
  - puts burden of finding compatible formats on developer
- More similar to Direct3D render target model
  - makes porting code easier
- Renderbuffer images and texture images can be shared among framebuffers
  - e.g. share depth buffers between color targets
  - saves memory
EXT_framebuffer_object

- OpenGL framebuffer is a collection of logical buffers
  - color, depth, stencil, accumulation
- Framebuffer object extension provides a new mechanism for rendering to destinations other than those provided by the window system
  - window system independent
- Destinations known as “framebuffer-attachable images”. Can be:
  - off-screen buffers (Renderbuffers)
  - textures
Terminology

- Renderbuffer image - 2D array of pixels. Part of a renderbuffer object.
- Framebuffer-attachable image - 2D array of pixels that can be attached to a framebuffer. Texture images and renderbuffer images are examples.
- Attachment point - State that references a framebuffer-attachable image. One each for color, depth and stencil buffer of a framebuffer.
- Attach - the act of connecting one object to another. Similar to “bind”.
- Framebuffer attachment completeness
  Framebuffer completeness
Framebuffers and Renderbuffers

• Introduces two new OpenGL objects:
  • “Framebuffer” (FBO)
    – collection of framebuffer-attachable images
      (e.g. color, depth, stencil)
    – plus state defining where output of GL rendering is directed
    – equivalent to window system “drawable”
  • “Renderbuffer” (RB)
    – contains a simple 2D image
      • no mipmaps, cubemap faces etc.
    – stores pixel data resulting from rendering
    – cannot be used as textures
Framebuffer Objects

- When a framebuffer object is bound its attached images are the source and destination for fragment operations
  - Color and depth textures
    - Supports multiple color attachments for MRT
  - Color, depth or stencil renderbuffers
Framebuffer Object API

```c
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 and Renderbuffers

• Creating and destroying FBOs (and Renderbuffers) is easy - similar to texture objects

  void GenFramebuffersEXT(sizei n, uint *framebuffers);
  void DeleteFramebuffersEXT(sizei n, const uint *framebuffers);
  void BindFramebufferEXT(enum target, uint framebuffer);

• Can also check if a given identifier is a framebuffer object (rarely used)

  boolean IsFramebufferEXT(uint framebuffer);
Binding an FBO

```c
void BindFramebufferEXT(enum target, uint framebuffer);
```

- **Makes given FBO current**
  - all GL operations occur on attached images
- **target** must be FRAMEBUFFER_EXT
- **framebuffer** is FBO identifier
  - if `framebuffer` == 0, GL operations operate on window-system provided framebuffer (i.e. the window). This is the default state.
- **Set of state that can change on a framebuffer bind:**
  - AUX_BUFFERS, MAX_DRAW_BUFFERS, STEREO, SAMPLES, X_BITS, DOUBLE_BUFFER and a few more
Attaching Textures to a Framebuffer

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

- Attaches image from a texture object to one of the logical buffers of the currently bound framebuffer
- `target` must be `FRAMEBUFFER_EXT`
- `attachment` is one of:
  - `COLOR_ATTACHMENT0_EXT` ...
  - `COLOR_ATTACHMENTn_EXT`, `DEPTH_ATTACHMENT_EXT`, `STENCIL_ATTACHMENT_EXT`
- `textarget` must be one of:
  - `TEXTURE_2D`, `TEXTURE_RECTANGLE`, `TEXTURE_CUBE_MAP_POSITIVE_X` etc.
- `level` is the mipmap level of the texture to attach
- `texture` is the texture object to attach
  - if `texture`==0, the image is detached from the framebuffer
- Other texture dimensions are similar
  - for 3D textures, `z-offset` specifies slice
Renderbuffer API

```c
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);
```
**Defining RenderBuffer Storage**

```c
void RenderbufferStorageEXT(enum target, 
    enum internalformat, sizei width, sizei height);
```

- Defines format and dimensions of a **Renderbuffer**
  - similar to TexImage call, but without image data
  - can read and write data using Read/DrawPixels etc.
- **target** must be RENDERBUFFER_EXT
- **internalformat** can be normal texture format (e.g. GL_RGBA8, GL_DEPTH_COMPONENT24) or:
  - STENCIL_INDEX1_EXT
  - STENCIL_INDEX4_EXT
  - STENCIL_INDEX8_EXT
  - STENCIL_INDEX16_EXT
void FramebufferRenderbufferEXT(enum target, enum attachment, enum renderbuffertarget, uint renderbuffer);

• Attaches given renderbuffer as one of the logical buffers of the currently bound framebuffer
• target must be FRAMEBUFFER_EXT
• attachment is one of:
  – COLOR_ATTACHMENT0_EXT ...
    COLOR_ATTACHMENT\(n\)_EXT
    • Maximum number of color attachments implementation dependent - query MAX_COLOR_ATTACHMENTS_EXT
  – DEPTH_ATTACHMENT_EXT
  – STENCIL_ATTACHMENT_EXT
• renderbuffertarget must be RENDERBUFFER_EXT
• renderbuffer is a renderbuffer id
Generating Mipmaps

void GenerateMipmapEXT(enum target);

- Automatically generates mipmaps for texture image attached to \textit{target}
- Generates same images as \texttt{GL\_SGIS\_generate\_mipmap} extension
  - filtering is undefined, most likely simple 2x2 box filter
- Implemented as new entry point for complicated reasons (see spec).
Framebuffer Completeness

- **Framebuffer is complete if all attachments are consistent**
  - texture formats make sense for attachment points
    - i.e. don’t try and attach a depth texture to a color attachment
  - all attached images must have the same width and height
  - all images attached to COLOR_ATTACHMENT0_EXT - COLOR_ATTACHMENTn_EXT must have same format

- **If not complete, glBegin will generate error INVALID_FRAMEBUFFER_OPERATION**
Checking Framebuffer Status

```c
enum CheckFramebufferStatusEXT(enum target);
```

- Should always be called after setting up FBOs
- Returns enum indicating why framebuffer is incomplete:
  - `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_INCOMPLETE_READ_BUFFER_EXT`
  - `FRAMEBUFFER_UNSUPPORTED`
  - `FRAMEBUFFER_STATUS_ERROR`

- Completeness is implementation-dependent
  - if result is “FRAMEBUFFER_UNSUPPORTED”, application should try different format combinations until one succeeds
FBO Performance Tips

- Don’t create and destroy FBOs every frame
- Try to avoid modifying textures used as rendering destinations using TexImage, CopyTexImage etc.
FBO Usage Scenarios

- FBO allows several ways of switching between rendering destinations

- In order of increasing performance:
  - **Multiple FBOs**
    - create a separate FBO for each texture you want to render to
    - switch using `BindFramebuffer()`
      - can be 2x faster than `wglMakeCurrent()` in beta NVIDIA drivers
  - **Single FBO, multiple texture attachments**
    - textures should have same format and dimensions
    - use `FramebufferTexture()` to switch between textures
  - **Single FBO, multiple texture attachments**
    - attach textures to different color attachments
    - use `glDrawBuffer()` to switch rendering to different color attachments
#define CHECK_FRAMEBUFFER_STATUS() \ 
{ \ 
 GLenum status; \ 
 status = glCheckFramebufferStatusEXT(GL_FRAMEBUFFER_EXT); \ 
 switch(status) { \ 
 case GL_FRAMEBUFFER_COMPLETE_EXT: \ 
  break; \ 
 case GL_FRAMEBUFFER_UNSUPPORTED_EXT: \ 
  /* choose different formats */ \ 
  break; \ 
 case GL_FRAMEBUFFER_UNSUPPORTED_EXT: \ 
  /* choose different formats */ \ 
  break; \ 
 default: \ 
  /* programming error; will fail on all hardware */ \ 
  assert(0); \ 
 } \ 
 } \ 
}
Simple FBO Example

GLuint fb, depth_rb, tex;

// create objects
glGenFramebuffersEXT(1, &fb);   // frame buffer
glGenRenderbuffersEXT(1, &depth_rb); // render buffer
glGenTextures(1, &tex);          // texture
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);

// initialize texture
glBindTexture(GL_TEXTURE_2D, tex);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA8, width, height, 0,
            GL_RGBA, GL_UNSIGNED_BYTE, NULL);

// (set texture parameters here)

// attach texture to framebuffer color buffer
glFramebufferTexture2DEXT(GL_FRAMEBUFFER_EXT,
                          GL_COLOR_ATTACHMENT0_EXT, GL_TEXTURE_2D, tex, 0);
Simple FBO Example

// initialize 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);
CHECK_FRAMEBUFFER_STATUS();

...

// render to the FBO
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, fb);
// (draw something here, rendering to texture)

// render to the window, using the texture
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
glBindTexture(GL_TEXTURE_2D, tex);
Future Directions

• Currently an EXT extension
  – will be promoted to an ARB extension once the design is proven
• Got feedback?
  – Give it to the OpenGL ARB!
• Future extensions
  – Render to vertex attribute
    • likely built on top of Renderbuffers
  – Format groups
    • like pixel formats, defines groups of formats that work together for a given implementation
  – Multisampling, accumulation buffer support
Thanks

- Jeff Juliano, Mike Strauss and the rest of the NVIDIA OpenGL driver team
- Jeremy Sandmel, Jeff Juliano and the rest of the ARB Superbuffers working group