EECS 487: Interactive Computer Graphics

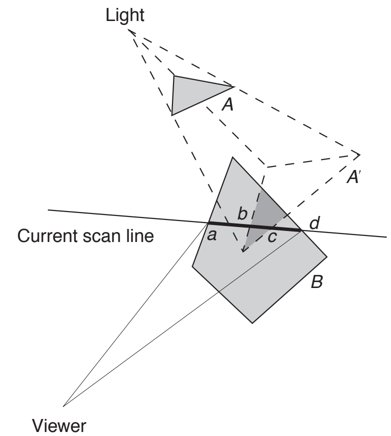
Lecture 31: Interactive Visual Effects

- Stencil Buffer
- Framebuffer Object (see sample code: http://web.eecs.umich.edu/~sugih/courses/eecs487/common/notes/gl3+webgl.tgz)

Clipped Projected Shadows

Once the projection matrix is determined:

- draw receiving planar polygon
- disable z-buffering
- draw projected occluder
  - in some dark color
  - but only where receiver is drawn
    - using stencil buffer

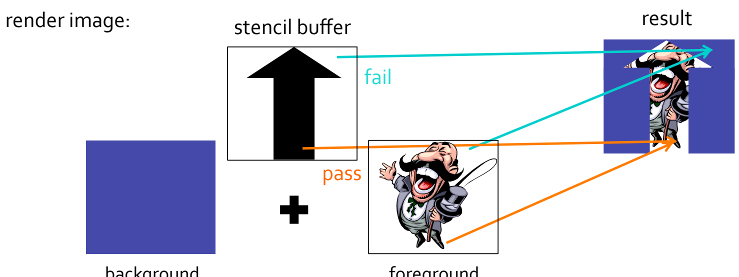


Foley et al.

Stencil Buffer

Restrict drawing to certain portion of the screen

- stencil test: for each fragment, check the corresponding stencil buffer content before rendering
- main idea: fragment rendering depends on contents of the stencil buffer passing the test
  - (not on "content of the fragment passing the test")



Akenine-Möller02

Stencil Buffer

Stencil buffer usually 8 bits/pixel

Not all stencil buffer bits are tested, only those corresponding to the fragment bits

Several actions are possible depending on outcome of stencil test

- including modifying the stencil buffer contents themselves

Stencil Buffer

First specify:

- criterion for passing
- the reference value to test against the stencil buffer content

```
void
glStencilFunc(GLenum func, GLint ref, GLuint mask);
```

- `mask`: which bits of `ref` and stencil buffer content to perform the test on
- `func`:
  - `GL_NEVER`
  - `GL_LESS`: passes if `(ref & mask) < (stencil & mask)`
  - `GL_EQUAL`: passes if `(ref & mask) == (stencil & mask)`
  - ...
  - `GL_ALWAYS`

Stencil Buffer

`glStencilFunc()` and `glStencilOp()` can be used together to initialize stencil buffer content

Next specify what to do to stencil buffer content if:

- `fail`: the test fails stencil test, or
- `zfail`: passes stencil test but fails depth test, or
- `zpass`: passes both stencil and depth tests

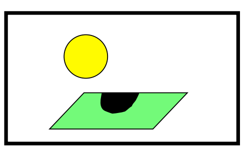
```
void
glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

- actions:
  - `GL_KEEP` keep the current value of stencil buffer
  - `GL_ZERO` set the stencil buffer value to zero
  - `GL_REPLACE` set the stencil buffer value to `ref`, as specified with `glStencilFunc()`
  - `GL_INCR` increment the current stencil buffer value (clamped to max)
  - `GL_DECR` decrement the current stencil buffer value (clamped to 0)
  - `GL_INVERT` bitwise invert the current stencil buffer value

Stencil Test Example

```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

Want:

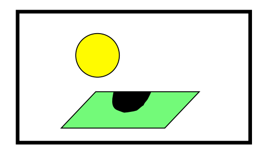


`GL_REPLACE`: set the stencil buffer value to `ref`

both color and stencil buffers

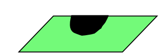
```
// draw lit receiver on both color and stencil buffers
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_ZERO, GL_ZERO, GL_REPLACE);

glCallList(receiver);   // set stencil to 1 everywhere receiver is drawn
```

Stencil Test Example

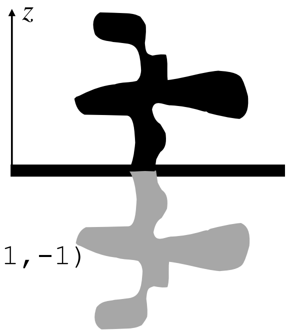


```
void glStencilFunc(GLenum func, GLint ref, GLuint mask);
void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass);
```

```
// draw unlit receiver in shadowed area, onto color buffer only
glDepthFunc(GL_LEQUAL);
glDisable(GL_LIGHTING);
glStencilFunc(GL_EQUAL, 1, 1);  // draw if corresponding stencil
                                // pixel is 1, else don't draw
glColor3f(0.0f, 0.0f, 0.0f);    // color it black
glPushMatrix();
glMultMatrixf((GLfloat*)shadowM); // shadow projection matrix
glCallList(occluder);  // transform+draw onto color buffer in black
                       // where stencil buffer is 1
glPopMatrix();
glDepthFunc(GL_LESS);
```



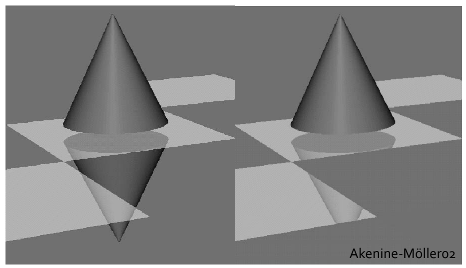
Planar Reflections



Reflections also influence visual perception of spatial relationships and help increase realism

For plane at $z = 0$, apply `glScalef(1,1,-1)`
- back facing polygons become front facing!
- lights must be reflected as well

When reflection surface is smaller than reflected image, reflected image need to be clipped (how?)

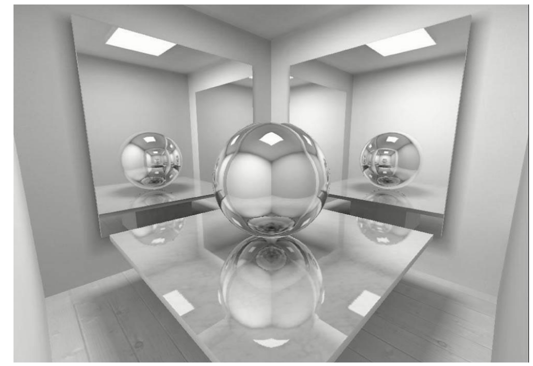


Akenine-Möller02

Rendering Planar Reflections

Render:
1. the mirror plane into the stencil buffer
2. the scaled $(1,1,-1)$ model, but masked with the stencil buffer
3. the mirror plane (semi-transparent)
4. the unscaled model

Alternate method: instead of scaling,
1. reflect the camera position and direction in the plane
2. render reflection image from there



Akenine-Möller02,Nielsen

Framebuffer Object

The accumulation buffer has been deprecated since OpenGL 3.1

Instead, use framebuffer object with floating-point pixel format (for the increased resolution)

OpenGL Default Framebuffer

Framebuffer: a collection of images that store information representing the image OpenGL eventually displays

OpenGL default framebuffer consists of:
- color buffer(s): contains info about the color of each pixel, there could be up to 4 color buffers: two for double buffering, which, together with the other 2, enable stereoscopic rendering
- depth (or z-) buffer: stores depth info of each pixel, allowing closer pixels to be drawn over those farther away
- stencil buffer: for masked rendering
- multisample buffer: for anti-aliasing
- accumulation buffer: for GFX
- auxiliary color buffer(s): for off-screen rendering

subsumed by FBO since OpenGL 3

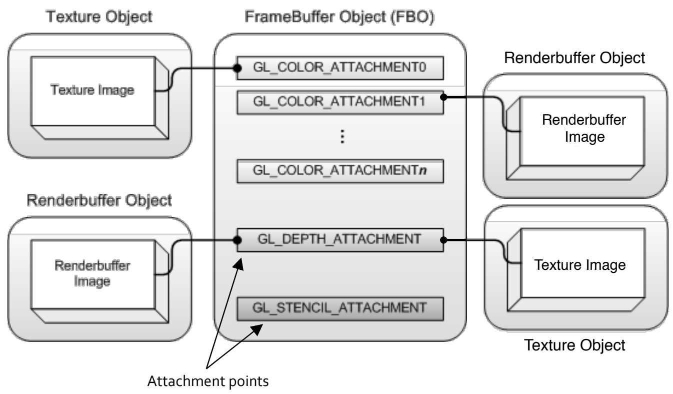
Framebuffer Object (FBO)

A mechanism for rendering to other than the default framebuffer, e.g., render-to-texture, as accumulation buffer, or other intermediate buffers for GFX

Each FBO can have texture object or renderbuffer object attached to it

Attachment is different from binding:
- binding binds an object to a context, the states of the context are mapped to the states of the object (changing one changes the other)
- attachment simply connects two objects together

FBO Graphicallly



[Ahn]

Texture vs. Renderbuffer Object

A texture object (we're familiar with from texturing):
- contains one or more images
- the images must all have the same format
- but could be of different sizes (for mipmapping, e.g.)
- used for render-to-texture
  - can be used to render from/with
  - can be bound to shader variables

A renderbuffer object:
- contains a single 2D image, no mipmaps, cubemap faces, etc.
- optimized to be used as render target
- can only be attached to an FBO and be rendered to
- mostly used as depth and stencil buffers
- also for offscreen-rendering and for pixel transfer (see PBO)
  - cannot be used to render from/with
  - cannot be bound to shader variables

Framebuffer Object (FBO)

Similar to the default framebuffer, an FBO have attachment points for:
- n (≥ 1) color-buffers (`GL_COLOR_ATTACHMENT`i)
  - `glGetFramebufferAttachmentParameter(…, GL_MAX_COLOR_ATTACHMENTS, …)` for value of n
- 1 depth-buffer (`GL_DEPTH_ATTACHMENT`)
- 1 stencil-buffer (`GL_STENCIL_ATTACHMENT`)
- also `GL_DEPTH_STENCIL_ATTACHMENT`
- (all may be multisampled)
- (no accumulation buffer)

Different attachment points impose different limitations on the format of attachable image

Framebuffer Object Setup

As with other OpenGL objects, first call `glGen*()`:

```
glGenFramebuffers(GLsizei n, GLuint *fbods);
```

Next bind FBO descriptor to a type of framebuffer

```
glBindFramebuffer(target, fbod);
// target is GL_FRAMEBUFFER (for read/write),
// GL_DRAW_FRAMEBUFFER, or
// GL_READ_FRAMEBUFFER, allowing for
// glReadPixels() and glDraw*() to operate
// on separate framebuffers
// fbod=0 is reserved for the default framebuffer, use fbod=0
// to unbind current framebuffer and revert to the default framebuffer
```

Subsequently, all rendering goes to the bound framebuffer

- `glViewport(0, 0, width, height)` render to the whole buffer

Renderbuffer Object Setup

To set up a renderbuffer object as the render target:

```
int rbod;
glGenRenderbuffers(1, &rbod);
glBindRenderbuffer(GL_RENDERBUFFER, rbod);
```

allocate storage for the renderbuffer:

```
glRenderbufferStorage(GL_RENDERBUFFER,
   internalformat, width, height);
// internalformat: depending on attachment: GL_RGBA, GL_RGB32F, etc.
// or GL_DEPTH_COMPONENT, GL_STENCIL_INDEX, GL_DEPTH_STENCIL
// see http://www.opengl.org/wiki/Image_Format#Required_formats
// width, height: must be < GL_MAX_RENDERBUFFER_SIZE
//    use glGet(GL_MAX_RENDERBUFFER_SIZE, …)
```

and attach it to the framebuffer

```
glFramebufferRenderbuffer(target, attachment_point,
                          GL_RENDERBUFFER, rbod);
```

Texture Object Setup

To set up a texture object as the render target:

```
int tod;
glGenTextures(1, &tod);
glBindTexture(GL_TEXTURE_2D, tod);
glTexImage2D(GL_TEXTURE_2D, level, internalformat,
   width, height, border, format, GL_UNSIGNED_BYTE, 0);
// the last argument is 0, no texture needs be copied
// level: can render to different levels of a mipmap, but no auto mipmap
//     with TexParam GL_GENERATE_MIPMAP b/c no texture is copied,
//     instead use glGenerateMipmap() after base image is modified
```

and attach it to the framebuffer:

```
glFramebufferTexture2D(target, attachment_point,
                       GL_TEXTURE_2D, tod, level);
// target: GL_FRAMEBUFFER (== GL_DRAW_FRAMEBUFFER, not read & write)
//         or GL_READ_FRAMEBUFFER
// tod==0 detaches texture object
```

Framebuffer Check

Before using the framebuffer target, check that it is set up properly and all objects are correctly attached:

```
GLenum glCheckFramebufferStatus(GL_FRAMEBUFFER);
```

you want to see `GL_FRAMEBUFFER_COMPLETE` returned

If the framebuffer is not complete, any reading/writing command will fail

See the wiki page for completeness rules and corresponding error messages: http://www.opengl.org/wiki/Framebuffer_Object

Render-to-Texture

Used to generate dynamic texture, e.g., for reflection effect, dynamic environment maps, shadow maps

Remember to:

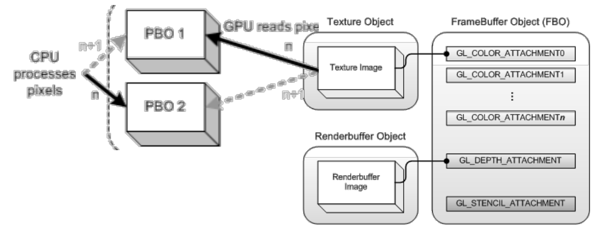
- `glEnable(GL_TEXTURE_2D)` before applying the texture (use `glPushAttrib(GL_ENABLE_BIT)` and `glPopAttrib()`)
- set the texture parameters for minification (and magnification and texture coordinate wrap around behavior, as necessary)
- and set the texture application mode: `GL_REPLACE`, `GL_BLEND`, etc.

Render-to-Texture

Even if you only need the color buffer, you may have to provide depth and stencil buffers if the rendering process needs them

- unless you want to store a shadow map, the depth buffer is usually a renderbuffer (faster)

Can be combined with PBO for post-processing FX such as image-based motion blur and depth-of-field



[Ahn]

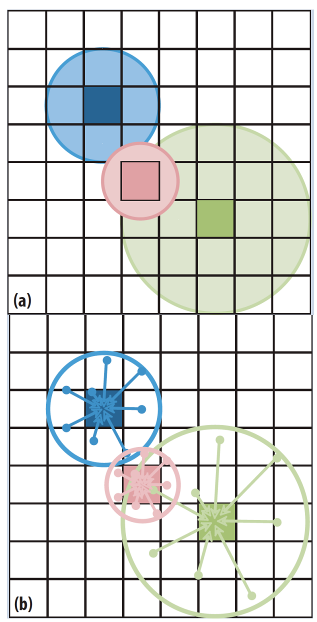
Post-processing FX

Creating multiple images takes time

Instead, simulate depth of field and motion blur as image post-processing

Depth of field, for depths away from focal distance:

a. forward mapping: color of a pixel is spread out to its circle of confusion as a function of depth
b. reverse mapping: color of a pixel is averaged from neighboring pixels, neighborhood size a function of depth



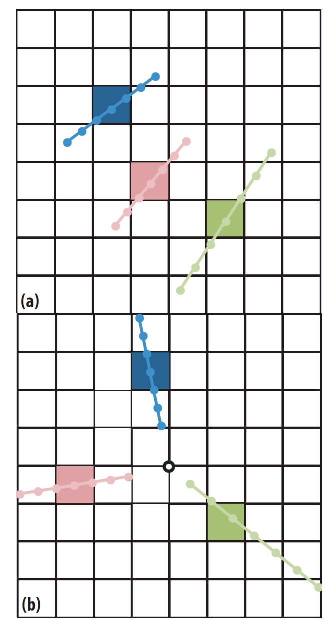
Yang, Yip, Xuog

Post-processing FX

Motion blur:

- during rendering, render to a velocity buffer the screen-space velocity of object at each pixel
- during post-processing, each pixel is blurred by averaging pixels in a line segment with equally spaced sampling point
- the direction and length of the line segment is a function of the velocity

- can also be simulated in object space by stretching vertices over time



Yang, Yip, Xuog

Read and Render Targets

In the app, you can specify which buffer to draw to or read from per bound framebuffer using:

```
glDrawBuffer(GL_COLOR_ATTACHMENTi);
glReadBuffer(GL_COLOR_ATTACHMENTi);
```

or specify more than one draw buffers:

```
glDrawBuffers(#buffers, buffers[]);
```

For example, to use buffer 0 as texture to render to buffer 1:

```
glReadBuffer(GL_COLOR_ATTACHMENT0);
glDrawBuffer(GL_COLOR_ATTACHMENT1);
glDrawArrays(…);
```

Render Target in Shader

If you attach a texture object `tod` at mipmap level 0 to color attachment 1:

```
glFramebufferTexture(GL_FRAMEBUFFER,
      GL_COLOR_ATTACHMENT1, tod, 0);
```

your fragment shader specifies this render target with:

```
layout(location = 1) out vec3 color;
```

To render to multiple targets, attach multiple color attachments and specify a different location for each fragment shader variable, e.g., temperature, stress level, etc. rendered as false color into different targets

Framebuffer Blitting

Blitting ::= copying a rectangular area of pixels from one framebuffer to another

- can blit between FBOs
- can also blit between an FBO and the default framebuffer, in either direction
- blitting is more limited than pixel transfer in format conversion (see http://www.opengl.org/wiki/Framebuffer_Object)

Framebuffer Blitting Example

To copy from buffer 1 of your fbo to the default framebuffer, for example:

```
glBindFramebuffer(GL_DRAW_FRAMEBUFFER, 0);
glBindFramebuffer(GL_READ_FRAMEBUFFER, fbod);
glReadBuffer(GL_COLOR_ATTACHMENT1);
glBlitFramebuffer(srcX0, srcY0, srcX1, srcY1, dstX0,
dstY0, dstX1, dstY1, GLbitfield mask, GLenum filter);
// mask: GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, or
// GL_STENCIL_BUFFER_BIT
// filter: if the image needs to be stretched, interpolate by
// GL_NEAREST or GL_LINEAR
```

For color buffer, only `GL_READ_FRAMEBUFFER` is copied to `GL_DRAW_FRAMEBUFFER`

Multi renders if more than one `GL_DRAW_FRAMEBUFFER` is specified

Using FBO as Accumulation Buffer

What we need: a framebuffer object with:

- a texture object with `GL_RGBA` *`internalformat`* to be our per-frame color buffer (attachment 0)
- a renderbuffer object with `GL_RGB32F` *`internalformat`* to be our accumulation buffer (attachment 1)
- a renderbuffer object to serve as our depth (and stencil) buffer

Init: clear our "accumulation buffer" (to 0):

```
glDrawBuffer(GL_COLOR_ATTACHMENT1);
glClearColor(0.0, 0.0, 0.0, 0.0);
glClear(GL_COLOR_BUFFER_BIT);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
          // for per-frame rendering
```

Using FBO as Accumulation Buffer

After each frame is rendered to color buffer 0:

- draw a quad that fills the screen (modify model-view and projection matrices) onto the accumulation buffer, textured with color buffer 0 already bound to `GL_TEXTURE_2D`:

```
glPushAttrib(GL_ENABLE_BIT);
glDisable(GL_DEPTH_TEST); glDisable(GL_LIGHTING);
// enable blending as per below
glDrawBuffer(GL_COLOR_ATTACHMENT1);
glEnable(GL_TEXTURE_2D);
glDrawArrays(…);
glDrawBuffer(GL_COLOR_ATTACHMENT0);
glPopAtrib();
```

- blend color buffer 0 with content of "accumulation buffer":

```
glEnable(GL_BLEND);
glBlendColor(0.0, 0.0, 0.0, weight);
      // same weight used with glAccum()
glBlendFunc(GL_CONSTANT_ALPHA, GL_ONE);
glBlendEquation(GL_FUNC_ADD);
```

Using FBO as Accumulation Buffer

To display the accumulation buffer:

- bind our FBO to `GL_READ_FRAMEBUFFER`, set color attachment 1 as the read buffer
- bind the default FBO (0) to `GL_DRAW_FRAMEBUFFER`
- call `glBlitFramebuffer()`
- bind the default FBO (0) to `GL_FRAMEBUFFER` and display

See `fbo.cpp` for example