Buffers and Fragment Tests

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science

Buffers

Image Buffer

Define a buffer by its spatial resolution \((n \times m)\) and its depth (or precision) \(k\), the number of bits/pixel.

WebGL Frame Buffer

Color buffers

Where are the Buffers?

• HTML5 Canvas
  - Default front and back color buffers
  - Under control of local window system
  - Physically on graphics card
• Depth buffer also on graphics card
• Stencil buffer
  - Holds masks
• Most RGBA buffers 8 bits per component
• Latest are floating point (IEEE)

Other Buffers

• Desktop OpenGL supported other buffers
  - auxiliary color buffers
  - accumulation buffer
  - these were on application side
  - now deprecated
• GPUs have their own or attached memory
  - texture buffers
  - off-screen buffers
  - not under control of window system
  - may be floating point
Images

- Framebuffer contents are unformatted
  - usually RGB or RGBA
  - one byte per component
  - no compression
- Standard Web Image Formats
  - jpeg, gif, png
- WebGL has no conversion functions
  - Understands standard Web formats for texture images

Writing into Buffers

- WebGL does not contain a function for writing bits into framebuffer
  - Use texture functions instead
- We can use the fragment shader to do bit level operations on graphics memory
- Bit Block Transfer (BitBlt) operations act on blocks of bits with a single instruction

BitBlt

- Conceptually, we can consider all of memory as a large two-dimensional array of pixels
- We read and write rectangular block of pixels
  - Bit block transfer (BitBlt) operations
- The frame buffer is part of this memory

The (Old) Pixel Pipeline

- OpenGL has a separate pipeline for pixels
  - Writing pixels involves
    - Moving pixels from processor memory to the framebuffer
    - Format conversions
    - Mapping, Lookups, Tests
  - Reading pixels
    - Format conversion

Packing and Unpacking

- Compressed or uncompressed
- Indexed or RGB
- Bit Format
  - little or big endian
- WebGL (and shader-based OpenGL) lacks most functions for packing and unpacking
  - use texture functions instead
  - can implement desired functionality in fragment shaders

Buffer Reading

- WebGL can read pixels from the current framebuffer with gl.readPixels
  - Returns 8 bit or integer RGBA values
  - In general, the format of pixels in the frame buffer is different from that of processor memory and these two types of memory reside in different places
    - Need packing and unpacking
    - Reading can be slow
  - Drawing through texture functions and off-screen memory (frame buffer objects)
**WebGL Pixel Function**

```javascript
var myimage[512*512*4];
gl.readPixels(0, 0, 512, 512, gl.RGBA, gl.UNSIGNED_BYTE, myimage);
```

**Formats & Types**

- `gl.RGBA`
- `gl.UNSIGNED_BYTE`
- `gl.RGB`
- `gl.BYTE`
- `gl.ALPHA`
- `gl.FLOAT`
- `gl.RED`
- `gl.HALF_FLOAT`
- `gl.RG`
- `gl.SHORT`
- `gl.RED_INTEGER`
- `gl.UNSIGNED_SHORT`
- `gl.RG_INTEGER`
- `gl.RGB_INTEGER`
- `gl.RGBA_INTEGER`
- `gl.UNSIGNED_BYTE`
- `gl.BYTE`
- `gl.FLOAT`
- `gl.HALF_FLOAT`
- `gl.SHORT`
- `gl.UNSIGNED_SHORT`
- `gl.UNSIGNED_BYTE`

**Clearing Buffers**

- A clear (default) value may be set for each buffer
  - `gl.clearColor()`
  - `gl.clearDepth()`
  - `gl.clearStencil()`
- `gl.clear(Glbitfield mask)`
- Clears the specified buffer
  - `gl.COLOR_BUFFER_BIT, gl.DEPTH_BUFFER_BIT, gl.STENCIL_BUFFER_BIT`
- Can be or'ed together in one `clear()` call

**Masking Buffers**

- A buffer may be masked, i.e. enabled or disabled
  - `gl.colorMask(red, green, blue, alpha)`
  - Arguments are booleans
  - `gl.depthMask(flag)`
  - `gl.stencilMask(mask)`
  - `gl.stencilMaskSeparate(face, mask)`
- Stencil specific sides (front & back) of triangles

**Render to Texture**

- GPUs now include a large amount of texture memory that we can write into
- Advantage: fast (not under control of window system)
- Using frame buffer objects (FBOs) we can render into texture memory instead of the frame buffer and then read from this memory
- - Image processing
- - GPGPU

**Framebuffer Objects**
Objectives

• Look at methods that use memory on the graphics card
• Introduce off screen rendering
• Learn how to create framebuffer objects
  - Create a renderbuffer
  - Attach resources

Discrete Processing in WebGL

• Recent GPUs contain large amounts of memory
  - Texture memory
  - Framebuffer
  - Floating point
• Fragment shaders support discrete operations at the pixel level
• Separate pixel (texel) pipeline

Accessing the Framebuffer

• Pre 3.1 OpenGL had functions that allowed access to the framebuffer and other OpenGL buffers
  - Draw Pixels
  - Read Pixels
  - Copy Pixels
  - BitBlt
  - Accumulation Buffer functions
• All deprecated

Going between CPU and GPU

• We will see that we can write pixels as texels to texture memory
• Texture objects reduce transfers between CPU and GPU
• Transfer of pixel data back to CPU slow
• Want to manipulate pixels without going back to CPU
  - Image processing
  - GPGPU

Framebuffer Objects

•Framebuffer Objects (FBOs) are buffers that are created by the application
  - Not under control of window system
  - Cannot be displayed
  - Can attach a renderbuffer to a FBO and can render off screen into the attached buffer
  - Attached buffer can then be detached and used as a texture map for an on-screen render to the default frame buffer

Render to Texture

• Textures are shared by all instances of the fragment shader
• If we render to a texture attachment we can create a new texture image that can be used in subsequent renderings
• Use a double buffering strategy for operations such as convolution
Steps

- Create an Empty Texture Object
- Create a FBO
- Attach renderbuffers to create and store texture image
- Bind FBO
- Render scene
- Detach renderbuffer
- Bind window system frame buffer
- Bind texture
- Render with new texture

Empty Texture Object

```javascript
var texture = gl.createTexture();
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture);
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, 512, 512, 0, gl.RGBA, gl.UNSIGNED_BYTE, null);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.NEAREST_MIPMAP_LINEAR);
gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.NEAREST);
```

Creating a FBO

- We create a framebuffer object in a similar manner to other objects
- Creating an FBO creates an empty FBO
- Must add needed resources
  - Can add a renderbuffer to render into
  - Can add a texture which can also be rendered into
  - For hidden surface removal we must add a depth buffer attachment to the renderbuffer
  - Can also add a stencil buffer to FBO

Frame Buffer Object

```javascript
var framebuffer = gl.createFramebuffer();
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
framebuffer.width = 512;
framebuffer.height = 512;
renderbuffer = gl.createRenderbuffer();
gl.bindRenderbuffer(gl.RENDERBUFFER, renderbuffer);
gl.renderbufferStorage(gl.RENDERBUFFER, gl.DEPTH_COMPONENT16, 512, 512);
gl.framebufferRenderbuffer(gl.FRAMEBUFFER, gl.DEPTH_ATTACHMENT, gl.RENDERBUFFER, renderbuffer);
```

Rest of Initialization

- Same as previous examples
  - Allocate VAO
  - Fill VAO with data for render to texture
- Initialize two program objects with different shaders
  - First for render to texture
  - Second for rendering with created texture

Render to Texture

```javascript
var status = gl.checkFramebufferStatus(gl.FRAMEBUFFER);
if (status != gl.FRAMEBUFFER_COMPLETE) alert('Frame Buffer Not Complete');
```
Objectives

- Examples of render-to-texture
- Render a triangle to texture, then use this texture on a rectangle
- Introduce buffer pingponging

Program Objects and Shaders

- For most applications of render-to-texture we need multiple program objects and shaders
  - One set for creating a texture
  - Second set for rendering with that texture
- Applications that we consider later such as buffer pingponging may require additional program objects

Program Object 1 Shaders

pass through vertex shader:

```glsl```
in vec4 aPosition;
void main()
{
  gl_Position = aPosition;
}
```

fragment shader to get a red triangle:

```glsl```
precision mediump float;
out vec4 fColor;
void main()
{
  fColor = vec4(1.0, 0.0, 0.0, 1.0);
}
```

Program Object 2 Shaders

```glsl```
// vertex shader
in vec4 aPosition;
in vec2 aTexCoord;
out vec2 vTexCoord;
void main()
{
  gl_Position = aPosition;
  vTexCoord = aTexCoord;
}
```

```glsl```
// fragment shader
precision mediump float;
in vec2 vTexCoord;
out vec4 fColor;
uniform sampler2D uTexture;
void main()
{
  fColor = texture2D(uTexture, vTexCoord);
}
```

First Render (to Texture)

```javascript```
gl.useProgram(program1); // Outputs a constant color
var buffer1 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer1);
gl.bufferData(gl.ARRAY_BUFFER, flatten(pointsArray), gl.STATIC_DRAW);
// Initialize the vertex position attribute from the vertex shader
var aPosition = gl.getAttribLocation(program1, "aPosition");
gl.vertexAttribPointer(aPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(aPosition);
// Render one triangle to an FBO
gl.viewport(0, 0, 64, 64);
gl clearColor(0.5, 0.5, 0.5, 1.0);
gl clear(gl.COLOR_BUFFER_BIT);
gl drawArrays(gl.TRIANGLES, 0, 3);
```

Set Up Second Render

```javascript```
// Bind to default window system framebuffer using null argument
gl bindFramebuffer(gl.FRAMEBUFFER, null);
// Bind Renderbuffer object to RENDERBUFFER, null;
gl disableVertexAttribArray(aPosition);
gl useProgram(program2); // Outputs colors from texture map
// We have already set up a texture object with null texture image
tex Image was written into by the previous drawArrays command
gl activeTexture(gl.TEXTURE0);
gl bindTexture(gl.TEXTURE_2D, texImage);
// set up vertex attribute arrays for texture coordinates and rectangle as usual
```

---

```
Angel and Shreiner: Interactive Computer Graphics 7E © Addison-Wesley 2015
```
Data for Second Render

```javascript
var buffer2 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer2);
var vertices = [ [0,0], [1,0], [1,1], [0,1] ];
gl.bufferData(gl.ARRAY_BUFFER, flatten(vertices), gl.STATIC_DRAW);
var aPosition = gl.getAttribLocation(program2, "aPosition");
gl.vertexAttribPointer(aPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(aPosition);
```

Render a Quad with Texture

```javascript
var buffer3 = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, buffer3);
gl.bufferData(gl.ARRAY_BUFFER, flatten(texCoord), gl.STATIC_DRAW);
var aTexCoord = gl.getAttribLocation(program2, "aTexCoord");
gl.vertexAttribPointer(aTexCoord, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(aTexCoord);
```

Buffer Ping-ponging

- Iterative calculations can be accomplished using multiple render buffers
- Original data in texture buffer 1
- Render to texture buffer 2
- Swap buffers and rerender to texture

Picking by Color

- Use off-screen rendering for picking
- Example: rotating cube with shading
  - indicate which face is clicked on with mouse
  - normal rendering uses vertex colors that are interpolated across each face
  - Vertex colors could be determined by lighting calculation or just assigned
  - use console log to indicate which face (or background) was clicked

Objectives

- Use off-screen rendering for picking
- Example: rotating cube with shading
  - indicate which face is clicked on with mouse
  - normal rendering uses vertex colors that are interpolated across each face
  - Vertex colors could be determined by lighting calculation or just assigned
  - use console log to indicate which face (or background) was clicked

Algorithm

- Assign a unique color to each object
- When the mouse is clicked:
  - Do an off-screen render using these colors and no lighting
  - use gl.readPixels to obtain the color of the pixel where the mouse is located
  - map the color to the object id
  - do a normal render to the display
Shaders

- Only need one program object
- Vertex shader: same as in previous cube examples
  - includes rotation matrices
  - gets angle as uniform variable
- Fragment shader
  - Stores face colors for picking
  - Gets vertex color for normal render from rasterizer
- Send uniform integer to fragment shader as index for desired color

Fragment Shader

precision meduimp float;
uniform int uColorIndex;
in vec4 vColor;
out vec4 fColor;

void main()
{
vec4 c[7];
c[0] = vColor;
c[1] = vec4(1.0, 0.0, 0.0, 1.0);
c[2] = vec4(0.0, 1.0, 0.0, 1.0);
c[3] = vec4(0.0, 0.0, 1.0, 1.0);
c[4] = vec4(1.0, 1.0, 0.0, 1.0);
c[5] = vec4(0.0, 1.0, 1.0, 1.0);
c[6] = vec4(1.0, 0.0, 1.0, 1.0);
fColor = c[uColorIndex];
}

Event Listener

canvas.addEventListener("mousedown", function(){
gl.bindFramebuffer(gl.FRAMEBUFFER, framebuffer);
gl.clear(gl.COLOR_BUFFER_BIT);
for(var i=0; i<6; i++) {
gl.uniform1i(gl.getUniformLocation(program, "uColorIndex"), i+1);
gl.drawArrays(gl.TRIANGLES, 6*i, 6);
}
var x = event.clientX;
var y = canvas.height - event.clientY;  // Flipping y !!!
gl.readPixels(x, y, 1, 1, gl.RGBA, gl.UNSIGNED_BYTE, color);

if(color[0]==255)
if(color[1]==255)
console.log("yellow");
else if(color[1]==255)
console.log("magenta");
else if(color[2]==255)
console.log("cyan");
else if(color[2]==255)
console.log("green");
else if(color[0]==255)
console.log("blue");
else
console.log("background");

Event Listener

// return to default framebuffer
gl.bindFramebuffer(gl.FRAMEBUFFER, null);
//send index 0 to fragment shader
gl.uniform1i(gl.getUniformLocation(program, "uColorIndex"), 0);
//normal render
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays(gl.TRIANGLES, 0, 36);
Picking by Selection

- Possible with render-to-texture
- When mouse clicked do an off-screen rendering with new viewing conditions that render only a small area around mouse
- Or render full scene and just sample off-screen image at mouse click position
- Keep track of what gets rendered to this off-screen buffer
- Know what was picked by returned color
Pixel Tests

- **Scissor**
  - Only draw in a rectangular portion of screen
  - `gl.scissor()` – Specify rectangle
  - Default rectangle matches window

- **Depth**
  - Draw based on depth value and comparison function
  - `gl.depthFunc()` – Specify comparison function
  - Default is `gl.LESS`

- **Stencil**
  - Draw based on values in stencil buffer, if available and enabled
  - Used for drawing into an irregular region of color buffer
  - `gl.stencilFunc()` – Specifies comparison function, reference value and mask
  - `gl.stencilOp()` – Specifies how fragments can modify stencil buffer
  - Used for reflections, capping and stippling

Dithering

- **Dithering**
  - Dithering may be enabled (`gl.DITHER`) on some systems with limited color resolution
  - System/hardware-dependent

Opacity and Transparency using Blending

- **Opaque surfaces** permit no light to pass through
- **Transparent surfaces** permit all light to pass
- **Translucent surfaces** pass some light
  - Translucency = 1 – opacity ($\alpha$)

Physical Models

- Dealing with translucency in a physically correct manner is difficult due to
  - the complexity of the internal interactions of light and matter
  - Using a pipeline renderer

Writing Model for Blending

- Use A component of RGBA (or RGBu) color to store opacity
- During rendering we can expand our writing model to use RGBA values
Blending Equation

• We can define source and destination blending factors for each RGBA component

\[ s = [s_r, s_g, s_b, s_a] \]
\[ d = [d_r, d_g, d_b, d_a] \]

Suppose that the source and destination colors are

\[ b = [b_r, b_g, b_b, b_a] \]
\[ c = [c_r, c_g, c_b, c_a] \]

Blend as

\[ c' = s \cdot b + d \cdot c \]
\[ c' = [b_r s_r + c_r d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_a s_a + c_a d_a] \]

WebGL Blending

• Must enable blending and set source and destination factors

\[ \text{gl.enable}(\text{gl.BLEND}) \]
\[ \text{gl.blendFunc} \text{(source_factor, destination_factor)} \]

• Only certain factors supported

- \text{gl.ZERO}, \text{gl.ONE}
- \text{gl.SRC_ALPHA}, \text{gl.ONE_MINUS_SRC_ALPHA}
- \text{gl.DST_ALPHA}, \text{gl.ONE_MINUS_DST_ALPHA}
- See WebGL spec for complete list

Example

• Suppose that we start with the opaque background color \((R_0, G_0, B_0, 1)\)
  - This color becomes the initial destination color
• We now want to blend in a translucent polygon with color \((R_1, G_1, B_1, a_1)\)
• Select \text{gl.SRC_ALPHA} and \text{gl.ONE_MINUS_SRC_ALPHA} as the source and destination blending factors

\[ R_0' = a_1 R_1 + (1 - a_1) R_0, \ldots \]

• Note this formula is correct if polygon is either opaque or transparent

Clamping and Accuracy

• All the components (RGBA) are clamped and stay in the range \((0,1)\)
• However, in a typical system, RGBA values are only stored to 8 bits
  - Can easily lose accuracy if we add many components together
  - Example: add together \(n\) images
    - Divide all color components by \(n\) to avoid clamping
    - Blend with source factor = 1, destination factor = 1
    - But division by \(n\) loses bits

Order Dependency

• Is this image correct?
  - Probably not
  - Polygons are rendered in the order they pass down the pipeline
  - Blending functions are order dependent

Opaque and Translucent Polygons

• Suppose that we have a group of polygons some of which are opaque and some translucent
• How do we use hidden-surface removal?
  - Opaque polygons block all polygons behind them and affect the depth buffer
  - Translucent polygons should not affect depth buffer
  - Render with \text{gl.depthMask}(\text{false}) which makes depth buffer read-only
• Sort polygons first to remove order dependency!
• Draw back to front
Blending and HTML

• In desktop OpenGL, the A component has no effect unless blending is enabled.
• In WebGL, an A other than 1.0 has an effect because WebGL works with the HTML5 Canvas element.
• A = 0.5 will cut the RGB values by ½ when the pixel is displayed.
• Allows other applications to be blended into the canvas along with the graphics.

Fragment Tests and Operations

• Tests and operations are performed in the following order:
  - Scissor test
  - Stencil test
  - Depth test
  - Blending
  - Dithering
• On/off `gl.enable()`, `gl.disable()`

Other Buffer Applications

Anti-aliasing

• Aliasing – artifacts produced from inadequate sampling
  - Jagged edges
  - Missing thin objects/features
• Anti-aliasing – removing artifacts via super-sampling, filtering, blurring, smoothing
• OpenGL offers a number of ways to perform anti-aliasing.
• More limited in WebGL.

Line Aliasing

• Ideal raster line is one pixel wide
• All line segments, other than vertical and horizontal segments, partially cover pixels
• Simple algorithms color only whole pixels
• Lead to the “jaggies” or aliasing
• Similar issue for polygons

Antialiasing

• Color a pixel by adding in a fraction of the fragment’s color
  - Fraction depends on percentage of pixel covered by object
  - Fraction depends on whether there is overlap
Area Averaging

• Use average $a_1 + a_2 - a_1 \cdot a_2$ as blending factor

OpenGL Antialiasing

• Not (yet) supported in WebGL
• Can enable separately for points, lines, or polygons
  // Compute fractional alpha values along edges
  // Based on pixel coverage
  glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); glDisable(GL_POINT_SMOOTH); glDisable(GL_LINE_SMOOTH); glDisable(GL_POLYGON_SMOOTH); glEnable(GL_BLEND); glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); /!

WebGL Antialiasing

• Full-screen antialiasing
• Multiple renderings with texture ping-pong
• Jitter view
• Average several jittered images together

Fog

• We can blend with a fixed color and have the blending factors depend on depth
  - Simulates a fog effect
• Blend source color $C_s$ and fog color $C_f$ by
  $C_s' = f \cdot C_s + (1 - f) \cdot C_f$
  - $f$ is the fog factor based on depth
  - Exponential
  - Gaussian
  - Linear (depth cueing)
• Hard-coded fog deprecated but can recreate

Fog Functions
Fog Effect

Interactive Depth-of-Field

- Jitter camera
- Each frustum has common plane “in focus”
- Accumulate & blend images

Interactive Depth-of-Field

Reflections

- One of the most noticeable effect of inter-object lighting
- Direct calculation of the physics (ray tracing) is too expensive
- Our focus is to capture the most significant reflection while minimizing the overhead via rendering the “virtual object”

Reflections

Planar Reflections

- The most common reflection – flat mirror, floor, wall, etc
- Creating virtual objects (or reflected objects) is much easier
- A view independent operation – only consider the relative position of the object and the reflector
- The virtual object is created by transforming the object across the reflector plane

Planar Reflections

Image vs. Object Space Methods

- Image space methods: create a texture from a view of the reflected objects and apply it to the reflector
  - Advantage: does not depend on the object geometry
  - Disadvantage: sampling issue and also only an approximation (environment mapping as an example)
- Object space methods: create the actual geometry of the object to be reflected and render it to the reflector
  - Disadvantage: Limited to planar reflections
  - Advantage: more accurate reflection (for nearby objects)
- Both methods need to create the virtual objects

Image vs. Object Space Methods
Planar Reflections

Render the Reflected Geometry

An important task: clip the reflected geometry so it is only visible on the reflector surface
- Beyond the reflector boundaries and in front of reflector

Clipping using the stencil
- The key is you only want the reflected geometry to appear on the reflector surface
- Use stencil buffer:
  - Clear the stencil buffer
  - Render the reflector and set the stencil
  - Render the reflected geometry only where the stencil pixels have been set
- The above algorithm uses the stencil buffer to control where to draw the reflection

Clipping using the stencil
- Another method: render the reflected object first, and then render the reflector to set the stencil buffer, then clear the color buffer everywhere except where the stencil is set
- This method is used to use the stencil buffer to control where to erase the incorrect reflection
- Advantage: when it is faster to use stencil to control clearing the scene than drawing the entire scene with stencil tests

The stencil erase algorithm

Reflection Effect
Motion Effects

HW9 Suggestions

• Create an off-screen frame buffer
  - With color(texture) and depth buffers
• Draw normally to on-screen frame buffer
• Allow user to click in the graphics window
• Draw your three objects to this frame buffer, with each object having a unique, constant color
  - This color will act as the object’s ID

• Read color at click point out of the off-screen frame buffer
• The color will tell you if an object was selected and which one
• Randomly change the diffuse color of the picked object
• YourEventListener should draw to both the off-screen and on-screen buffers