UCLA

CS174A : Introduction to Computer Graphics

Royce 190
TT 4-6pm

Scott Friedman, Ph.D
UCLA Institute for Digital Research and Education

Buffers

- Frame (Color) buffer
  - RGB color values we see on screen.
    - Sometimes configured as RGBA
- Depth buffer
  - Normalized z values from clip volume.
  - Used for hidden surface elimination.
  - Allows us to draw objects without respect to their position in space. Order independent.

Depth Buffer

- When the depth buffer is enabled
  - A z value is written into buffer for every pixel.
  - If an incoming pixel has a z value less than the value already in the buffer
    - The pixel is written into the frame (color) buffer.
    - The z value is updated in the depth buffer.
  - Else
    - The pixel is rejected and not written to the frame buffer.
    - The depth buffer is not modified.

Depth Buffer

- What happens without it?
  - Last write to the frame buffer “wins”.
  - The order that objects are drawn then *matters*.
  - Objects have to be rendered back to front.
    - Why?
  - Potentially problematic cases
    - For inter-penetrating objects
    - Moving objects

UCLA

Depth Buffer

- There is very little to do in order to use
  - Request a depth buffer (usually during initialization).
  - Enable the depth buffer.

```
function init()
{
    ...
    gl.enable( gl.DEPTH_TEST );
}
```

Depth Buffer

- Why do we need to enable?
  - OpenGL is a state machine, remember.
  - Also have to *clear* the depth buffer when starting a new image.

```
function display()
{
    gl.Clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT );
    ...
}
```

Frame Buffer

- Typically we use double buffering
    - Allocates two color buffers - front and back.
    - Rendering is performed on the back buffer.
    - Swapping makes the back the front.
  - This is handled by the browser in WebGL!

```
void display( void )
{
    ...
    glutSwapBuffers( );
}

main( )
{
    ...
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
    ...
    glutDisplayFunc( display );
}
```

Frame Buffer

- Front and back just describe
  - Which buffer is written to and which is displayed.
  - Notice that we use GLUT to do the swap
    - Because this is a windowing system operation.

```
void display( void )
{
    ...
    glutSwapBuffers( );
}

main( )
{
    ...
    glutInitDisplayMode( GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH );
    ...
    glutDisplayFunc( display );
}
```

Frame Buffer

- Transparent object rendering
  - The 'A' in RGBA – so far it has always been 1.0
  - Just setting it to something < 1.0 does not work.
  - There is a bit more to it, however…

```
function display()
{
    ...
    gl.enable( gl.BLEND );
    gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
    gl.depthMask( gl.FALSE );
    ... Draw trasparent geometry ...
    gl.depthMask( gl.TRUE );
    gl.disable( gl.BLEND );
}
```

Frame Buffer

- Transparent object rendering
  - First, we must *enable* blending in order for the pipeline to even consider the A values.
  - The OpenGL state is then set to perform blending.

```
function display()
{
    ...
    gl.enable( gl.BLEND );
    gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
    gl.depthMask( gl.FALSE );
    ... Draw trasparent geometry ...
    gl.depthMask( gl.TRUE );
    gl.disable( gl.BLEND );
}
```

Frame Buffer

- Transparent object rendering
  - Blending involves the RGBA value of the pixel in the pipeline
    - The *source* pixel.
  - and the RGBA value of the pixel already in the frame buffer.
    - The *destination* pixel.
  - There is a predefined set of functions internal to the pipeline that determines how these values interact.

Frame Buffer

- Transparent object rendering
  - The blending function adds the two pixels together after multiplying each by its own blending factor.
    - There are several options, see the API
    - The common case for this purpose is below

```
function display()
{
    ...
    gl.enable( gl.BLEND );
    gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
    gl.depthMask( gl.FALSE );
    ... Draw trasparent geometry ...
    gl.depthMask( gl.TRUE );
    gl.disable( gl.BLEND );
}
```

Frame Buffer

- Transparent object rendering
  - This combination keeps the color value in the frame buffer from saturating (going above 1.0)

$$(R_{d'}, G_{d'}, B_{d'}, \alpha_{d'}) = (\alpha_s R_s + (1-\alpha_s) R_d, \alpha_s G_s + (1-\alpha_s) G_d, \alpha_s B_s + (1-\alpha_s) B_d, \alpha_s \alpha_d + (1-\alpha_s) \alpha_d).$$

```
function display()
{
    ...
    gl.enable( gl.BLEND );
    gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
    gl.depthMask( gl.FALSE );
    ... Draw trasparent geometry ...
    gl.depthMask( gl.TRUE );
    gl.disable( gl.BLEND );
}
```

Frame Buffer

- Transparent object rendering
  - But this is not enough for a correct result.
  - While we want the benefit of the depth buffer we do not want to update it.
    - Think about why…

```
function display()
{
    ...
    gl.enable( gl.BLEND );
    gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
    gl.depthMask( gl.FALSE );
    ... Draw trasparent geometry ...
    gl.depthMask( gl.TRUE );
    gl.disable( gl.BLEND );
}
```

Frame Buffer

- Transparent object rendering
  - We can control writing to the depth buffer with the depth mask.
    - – Effectively makes the depth buffer read-only.
  - What does this do?

```
function display()
{
    ...
    gl.enable( gl.BLEND );
    gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
    gl.depthMask( gl.FALSE );
    ... Draw trasparent geometry ...
    gl.depthMask( gl.TRUE );
    gl.disable( gl.BLEND );
}
```

Frame Buffer

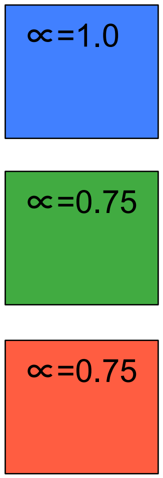
- Transparent object rendering
  - Imagine we want to render a frosted window.
  - What do we want to happen…
    - See what is behind the window.
    - Not see the window if it is behind other opaque objects.
    - Achievable by performing depth test but not writing depth value of window.
  - This has some implications

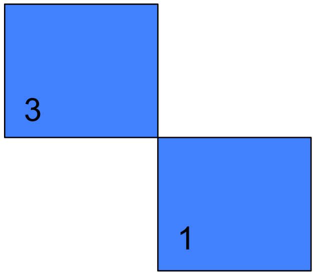
Frame Buffer

- Transparent object rendering
  - Implications for correct results
    - Have to render all transparent objects after opaque objects.
    - Must sort and render transparent object in back to front order.
  - The ordering is needed because we are not updating the depth buffer
    - Allows for correct transparent/opaque interaction.

Frame Buffer

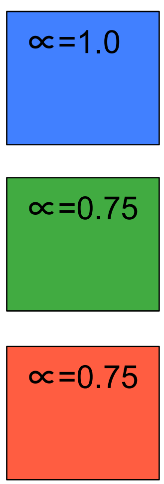
- Transparent object rendering
  - **Order** of transparent objects **matters**
  - Draw some opaque objects (# is depth)
  - Set the depth mask to FALSE

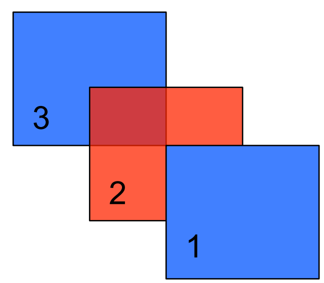




Frame Buffer

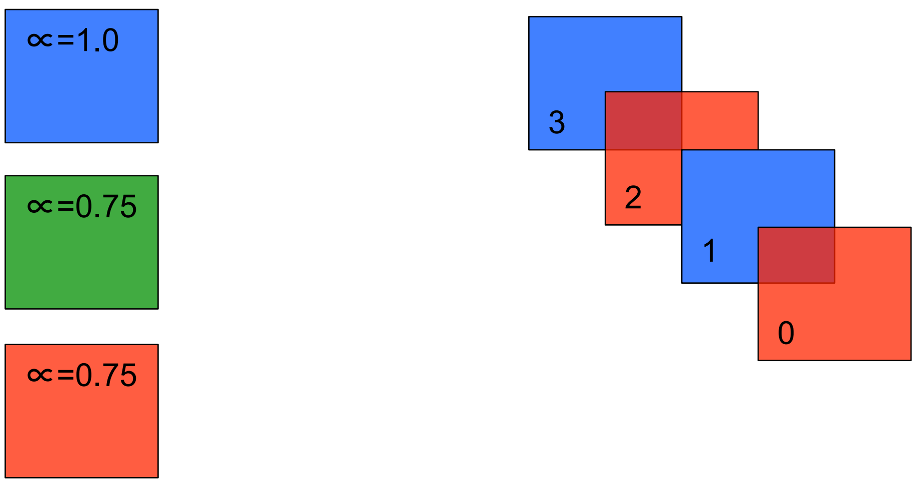
- Transparent object rendering
  - Draw a 25% transparent blue object at depth 2
  - It's occluded by the blue object at depth 1
  - Blue object is blended with red object





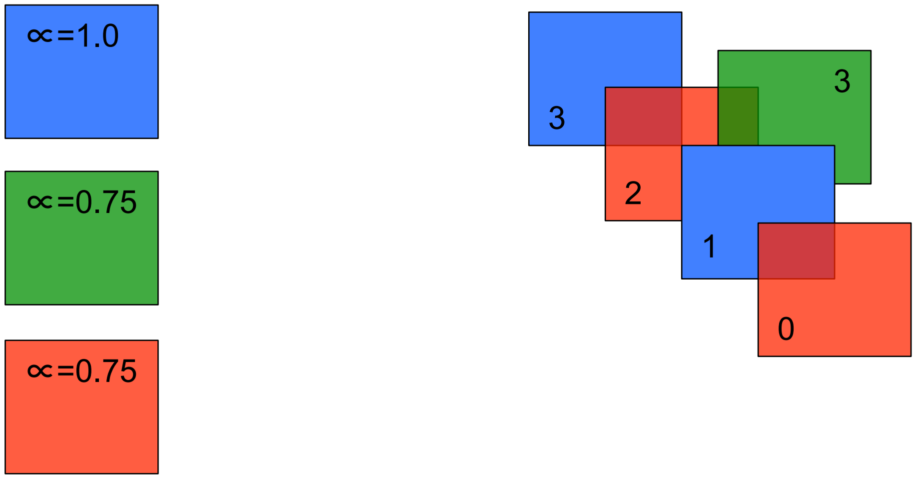
Frame Buffer

- Transparent object rendering
  - Red object at depth 0, blends with blue object
  - Depth buffer is still off so only values 1 and 3 have been written into the depth buffer.



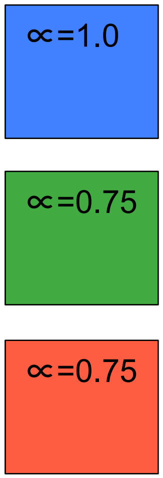
Frame Buffer

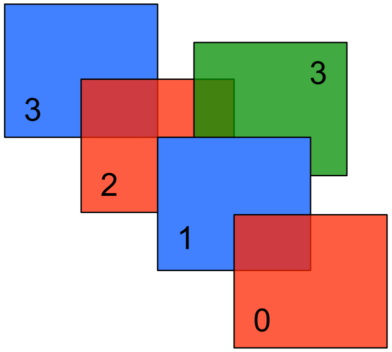
- Transparent object rendering
  - Draw green object at depth 3 gives **incorrect** result
  - It appears behind the opaque blue object at depth 1
  - Blending with red object at depth 2 is **wrong**.



Frame Buffer

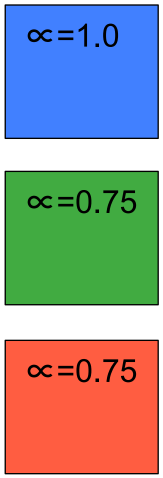
- Transparent object rendering
  - It appears on top of the red object when it should be behind it.

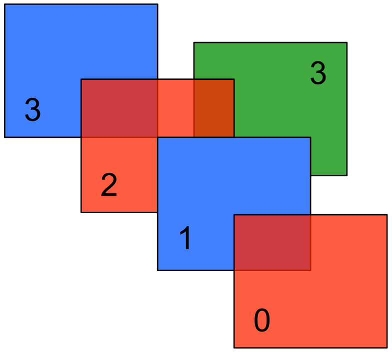




Frame Buffer

- Transparent object rendering
  - This is the correct result. (*Order matters!*)
  - Green on top results in color (.8125,.25,.0625)
  - Red on top results in color (.25,.8125.0625)





Frame Buffer

- Transparent object rendering
  - Once all transparent objects are rendered
    - – Allow writing to the depth buffer again
    - – Turn off blending.
    - – Things work with blending on but it is unnecessary and slow
      - » Think about blending with A=1.0 in src and dest.

```
function display()
{
    ...
    gl.enable( gl.BLEND );
    gl.blendFunc( gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA );
    gl.depthMask( gl.FALSE );
    ... Draw trasparent geometry ...
    gl.depthMask( gl.TRUE );
    gl.disable( gl.BLEND );
}
```

Mapping Methods

- Texture, Environment and Bump Mapping
  - Three variations on the same mechanism.
  - Ways to add detail and complexity
    - More and more complex geometry
    - Details assigned to vertex colors
  - This only can take things so far
    - Imagery can add detail and the appearance of complexity to geometry without actually modeling it.
    - Maps can be used to modify the shading directly by altering color values
    - or, can be used to alter the surface material or normal properties.

Mapping Methods

- Texture, Environment and Bump Mapping
  - Texture Mapping
    - Uses an image to alter a surfaces color values.
  - Environment Mapping
    - Gives objects the appearance of reflection.
  - Bump Mapping
    - Distort normal vectors during the shading process.
  - All three can be combined.
  - All are performed at the fragment stage.
  - All can be stored in 1, 2 or 3 dimensional buffers (maps).

Mapping Methods

- Texture mapping
  - We will consider the two-dimensional form (most common).
  - Are images, basically.
  - Individual pixels of the image are called ***texels***.
  - The texture map is described by $T(s,t)$.
    - The variables s and t are known as ***texture coordinates***.
    - Texture coordinates are defined over the interval [0,1].
  - In the strict sense of the term mapping
    - Given a parametric representation of a surface we are mapping from $T(s,t)$ onto a point $\mathbf{p}(u,v)$ on that surface.

Mapping Methods

- Texture mapping
  - In OpenGL we define this mapping by assigning texture coordinates to vertices.
    - Like color or normals.
    - These coordinates are then interpolated over the surface being rendered and made available in the fragment shader.
  - There are several steps to perform to use texture mapping.
    - Define and download an image to the GPU.
    - Assign texture coordinates to geometry
    - Apply the texture to fragments.

Mapping Methods

- Texture mapping
  - First step is to create a ***texture object***.
    - Container object that describes a texture.
    - We use the familiar `Gen` and `Bind` forms to create and activate our texture map when using desktop OpenGL.
    - Subsequent texture parameter specification applies to the currently bound texture.

```
function init()
{
    var tex;
    ...
    tex = gl.createTexture();
    ...
    gl.bindTexture( gl.TEXTURE_2D, tex );
    ...
}
```

UCLA

Mapping Methods

- Texture mapping
  - Once a ***texture object*** is created and bound (active)
    - We define the image data itself using `glTexImage2D()`.
    - Here we define the external and internal format of the image data. (*this is not jpeg etc. data, it's raw RGB)
    - After calling this function the data is copied onto the GPU.
  - WebGL has the ability to ingest normal images directly.

```
function init()
{
    Uint8Array texImage[512*512*3];
    …
    gl.bindTexture( gl.TEXTURE_2D, tex );
    gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB, 512,512, 0, gl.RGB, gl.UNSIGNED_BYTE, texImage );
    ...
}
```

Mapping Methods

- Texture mapping
  - Now that a texture is defined, how do we use it?
  - As we said, textures coordinates are assigned to vertices statically. (or they can be computed in the shader)
    - They can be passed through just like color and normals.
    - The texture coordinates can be allocated buffer space in the same way we add color or normals.

```
void init( void )
{
    var tCoordArray = [ ... ];
    var tBuf = gl.CreateBuffer();
    gl.bindBuffer( gl.ARRAY_BUFFER, tBuf );
    gl.bufferData( gl.ARRAY_BUFFER, flatten(texCoordArray), gl.STATIC_DRAW );
    var texCoord = glGetAttribLocation( program, "texCoord" );
    gl.vertexAttribPointer(texCoord, 2, gl.FLOAT, false, 0, 0 );
    gl.enableVertexAttribArray(texCoord );
}
```

Mapping Methods

- Texture mapping
    - **Vertex** shader
        - Nothing needs to be done just pass texture coordinates through to fragment shader.
        - This allows the *s* and *t* coordinates to be interpolated.

```
...
attribute vec2 texCoord;

...
varying vec2 st;

void main( void )
{
    ...
    st = texCoord;
}
```

Mapping Methods

- Texture mapping
  - **Fragment** shader
    - First we need to set up a way to control which texture object we are using.
    - Second, we need to set the texture we want to use.

```
void init( void )
{
    var texLoc;
    ...
    texLoc = gl.getUniformLocation( program, “texMap” );
    ...
}

Void display( void )
{
    ...
    gl.uniform1i( texLoc, tex );  // tex is the texture object we want to use
    // uniform1 is the texture (hardware) unit we want to use
    ...
}
```

Mapping Methods

- Texture mapping
  - Fragment shader
    - We define the texture reference as a new type
      - » sampler2D
    - The built in function texture2D will sample the referenced texture at the coordinate specified returning a color.
    - The color and texture could be combined any way you wish or even combined with the normal and lit.

```
varying vec2 st;
varying vec4 color;
uniform sampler2D texMap;

void main( void )
{
    gl_fragColor = color * texture2D( texMap, st );
}
```

Mapping Methods

- Texture mapping
  - You may be wondering how texture coordinates..
    - map outside of [0,1]?
    - map between [0,1] and discrete texels?

UCLA

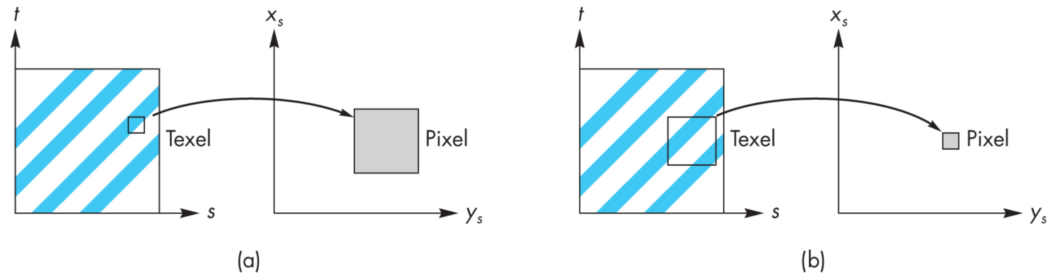
Mapping Methods

- Texture mapping
  - For (s,t) values outside the range of [0,1] we have two choices for *texture wrapping*.
    - Clamp (reuses edge texel, smears)
    - Repeat
  - These parameters are set separately for *s* and *t* and are performed when the texture is defined.

```
void init( void )
{
    var tex = gl.createTexture();
    ...
    gl.bindTexture( gl.TEXTURE_2D, tex );
    ...
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT );
    ...
}
```

Mapping Methods

- Texture mapping
  - Mapping between [0,1] and discrete texels
  - Sampling
    - Point, nearest neighbor
    - Linear, (bi-)linear interpolation
  - Minification and magnification
    - Can set for both cases



UCLA

Mapping Methods

- Texture mapping
  - Almost no reason not to use linear sampling.
    - All hardware is fast enough.
  - For minification OpenGL can create MipMaps.
    - Down sampled versions of the image.
    - Faster processing
    - User can create their own versions using high quality image resizing routines.
      - » Set using the *level* parameter of `gl.texImage2D()`
      - » Level 0 is the base image, 1 is the next smaller, etc.
      - » Each is half the size of the previous, down to 1x1.
      - » `gl.generateMipmap( gl.TEXTURE_2D )`

Mapping Methods

- Texture mapping
  - When mipmaps are present filtering the sample has additional options.
  - Here we sample across texels and mipmaps.
    - gl.NEAREST_MIP_NEAREST
    - gl.LINEAR_MIP_NEAREST
    - gl.NEAREST_MIP_LINEAR
    - gl.LINEAR_MIP_LINEAR (tri-linear interpolation)
      - » Interpolate between mipmap levels and then between texels. Best quality.

Mapping Methods

- Texture mapping
  - Which type of sampling to use?
  - Depends on desired result.
  - Quality and type of image used in texture map.
  - Avoidance of sampling artifacts. (moire patterns)
  - Experimentation is usually employed.
  - Resource usage must be considered.
    - – Mipmaps use 1/3 additional storage.

Mapping Methods

- Texture mapping
  - The full setup would look something like this.
  - Later, during rendering we would bind and pass a reference to the texture we wanted to sample.

```
void init( void )
{
    ...
    var tex = gl.CreateTexture()
    gl.bindTexture( gl.TEXTURE_2D, tex );
    gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB, 512,512, 0, gl.RGB, gl.UNSIGNED_BYTE, texImage );
    gl.generateMipmap( gl.TEXTURE_2D );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.REPEAT );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.REPEAT );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR_MIPMAP_LINEAR );
    gl.texParameteri( gl.TEXTURE_2D, gl.TEXTURE_MAG_FILTER, gl.LINEAR_MIPMAP_LINEAR );
    ...
}
```

Mapping Methods

- Texture mapping
  - Multi-texturing, GPUs today can process several textures at once.
    - We *activate* hardware texture unit then *bind* a texture to it.
    - Possibly need multiple sets of texture coordinates.
    - Up to fragment shader to determine how to process/combine the result.

```
void display( void )
{
    ...
    gl.activeTexture( gl.TEXTURE0 );
    gl.bindTexture( gl.TEXTURE_2D, tex1 );
    gl.uniform1i( texLoc1, tex1 );
    gl.activeTexture( gl.TEXTURE1 );
    gl.bindTexture( gl.TEXTURE_2D, tex2 );
    gl.uniform1i( texLoc2, tex2 );
    ...
}
```

Mapping Methods

- Texture mapping
  - Texture coordinate generation
  - This can be tricky if done by hand for anything other than simple things. (like in an assignment)
  - Mostly a tool like a modeling program are used to make fine adjustments.

Mapping Methods

- Texture mapping
  - Loading images from disk
  - OpenGL does not provide a mechanism for this!
  - You will need another library to help
    - libjpg, libpng, libtiff, DevIL, others.
    - WebGL is pretty easy (!)

```
void display( void )
{
    ...
    var texImage = Image();
    texImage.src = "kittens.gif"
    gl.pixelStorei( gl.UNPACK_FLIP_Y_WEBGL, true );
    gl.texImage2D( gl.TEXTURE_2D, 0, gl.RGB, gl.RGB, gl,UNSIGNED_BYTE, texImage );
    ...
}
```

UCLA

Mapping Methods

- Next time
    - Environment mapping
    - Bump mapping