Buffers, textures, compositing, shaders

- Buffers and images
- Textures
- Compositing and blending
- Vertex and buffer shaders

Buffer

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

OpenGL Frame Buffer

- stencil buffer
- accumulation buffer
- overlay planes
- auxiliary buffers
- color indices
- depth buffer
- back buffer
- front buffer
OpenGL Buffers

- Color buffers that can be displayed
  - Front
  - Back
  - Auxiliary
  - Overlay
- Depth
- Accumulation
  - High resolution buffer
- Stencil
  - Holds masks

Writing in Buffers

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

Writing Model

Read destination pixel before writing source

Bit Writing Modes

- Source and destination bits are combined bitwise
- 16 possible functions (one per column in table)
The Pixel Pipeline

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

Raster Position

- OpenGL maintains a *raster position* as part of the state
- Set by `glRasterPos*()`
  - `glRasterPos3f(x, y, z);`
- The raster position is a geometric entity
  - Passes through geometric pipeline
  - Eventually yields a 2D position in screen coordinates
  - This position in the frame buffer is where the next raster primitive is drawn

Buffer Selection

- OpenGL can draw into or read from any of the color buffers (front, back, auxiliary)
- Default to the back buffer
- Change with `glDrawBuffer` and `glReadBuffer`
- Note that format of the 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
  - Drawing and reading can be slow

Bitmaps

- OpenGL treats 1-bit pixels (*bitmaps*) differently from multi-bit pixels (*pixelmaps*)
- Bitmaps are masks that determine if the corresponding pixel in the frame buffer is drawn with the *present raster color*
  - $0 \Rightarrow$ color unchanged
  - $1 \Rightarrow$ color changed based on writing mode
- Bitmaps are useful for raster text
**Drawing Bitmaps**

```c
glBitmap(width, height, x0, y0, xi, yi, bitmap)
```

- `width` and `height`: Size of the bitmap.
- `x0` and `y0`: Offset from raster position.
- `xi` and `yi`: Increments in raster position after bitmap drawn.

**Pixel Maps**

- OpenGL works with rectangular arrays of pixels called pixel maps or images.
- Pixels are in one byte (8 bit) chunks:
  - Luminance (gray scale) images: 1 byte/pixel
  - RGB images: 3 bytes/pixel
- Three functions:
  - Draw pixels: processor memory to frame buffer
  - Read pixels: frame buffer to processor memory
  - Copy pixels: frame buffer to frame buffer

**OpenGL Pixel Functions**

```c
glReadPixels(x, y, width, height, format, type, myimage)
```

- `x` and `y`: Start pixel in frame buffer.
- `width` and `height`: Size.
- `format` and `type`: Type of image and type of pixels.
- `myimage`: Pointer to processor memory.

```c
byte myimage[] = new byte[512*512*3];
gl.glReadPixels(0, 0, 512, 512, GL.GL_RGB, GL.GL_UNSIGNED_BYTE, myimage);
```

```c
glDrawPixels(width, height, format, type, myimage)
```

- `width` and `height`: Size.
- `format` and `type`: Type of image and type of pixels.

**Image Formats**

- We often work with images in a standard format (JPEG, TIFF, GIF).
- How do we read/write such images with OpenGL?
- No support in OpenGL:
  - Java Image IO
  - Next jogl version: [com.sun.opengl.util.texture](http://com.sun.opengl.util.texture)
Texture Mapping

The Limits of Geometric Modeling

• Although graphics cards can render over 10 million polygons per second, that number is insufficient for many phenomena
  – Clouds
  – Grass
  – Terrain
  – Skin

Modeling an Orange

• Consider the problem of modeling an orange (the fruit)
• Start with an orange-colored sphere
  – Too simple
• Replace sphere with a more complex shape
  – Does not capture surface characteristics (small dimples)
  – Takes too many polygons to model all the dimples

Modeling an Orange (2)

• Take a picture of a real orange, scan it, and “paste” onto simple geometric model
  – This process is known as texture mapping
• Still might not be sufficient because resulting surface will be smooth
  – Need to change local shape
  – Bump mapping
Types of Mapping

- Texture Mapping
  - Uses images to fill inside of polygons
- Environment (reflection mapping)
  - Uses a picture of the environment for texture maps
  - Allows simulation of highly specular surfaces
- Bump mapping
  - Emulates altering normal vectors during the rendering process
- Normal mapping, displacement mapping, parallax mapping

Texture Mapping

- geometric model
- texture mapped

Environment Mapping

- Bump Mapping

- geometric model
- texture mapped
Where does mapping take place?

- Mapping techniques are implemented at the end of the rendering pipeline
  - Very efficient because few polygons make it past the clipper

Is it simple?

- Although the idea is simple---map an image to a surface---there are 3 or 4 coordinate systems involved

Coordinate Systems

- Parametric coordinates
  - May be used to model curves and surfaces
- Texture coordinates
  - Used to identify points in the image to be mapped
- Object or World Coordinates
  - Conceptually, where the mapping takes place
- Window Coordinates
  - Where the final image is really produced

Mapping Functions

- Basic problem is how to find the maps
- Consider mapping from texture coordinates to a point a surface
- Appear to need three functions
  \[ x = x(s,t) \]
  \[ y = y(s,t) \]
  \[ z = z(s,t) \]
- But we really want to go the other way
Backward Mapping

- We really want to go backwards
  - Given a pixel, we want to know to which point on an object it corresponds
  - Given a point on an object, we want to know to which point in the texture it corresponds
- Need a map of the form
  \[ s = s(x,y,z) \]
  \[ t = t(x,y,z) \]
- Such functions are difficult to find in general

Two-part mapping

- One solution to the mapping problem is to first map the texture to a simple intermediate surface
- Example: map to cylinder

Box Mapping

- Easy to use with simple orthographic projection
- Also used in environment maps

Second Mapping

- Map from intermediate object to actual object
  - Normals from intermediate to actual
  - Normals from actual to intermediate
  - Vectors from center of intermediate
**Aliasing**

- Point sampling of the texture can lead to aliasing errors

- Miss blue stripes

- Point samples in $u,v$ (or $x,y,z$) space

- Point samples in texture space

**Area Averaging**

A better but slower option is to use area

Note that preimage of pixel is curved

**OpenGL Texture Mapping**

**Basic Strategy**

Three steps to applying a texture

1. specify the texture
   - read or generate image
   - assign to texture
   - enable texturing

2. assign texture coordinates to vertices
   - Proper mapping function is left to application

3. specify texture parameters
   - wrapping, filtering
Texture Mapping

Texture Example

- The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective.

Texture Mapping and the OpenGL Pipeline

- Images and geometry flow through separate pipelines that join at the rasterizer.
  - “Complex” textures do not affect geometric complexity.

Specifying a Texture Image

- Define a texture image from an array of texels (texture elements) in CPU memory.
  - `byte myTexels[512*512]; // (Or ByteBuffer)`
- Define as any other pixel map.
  - Scanned image.
  - Generate by application code.
- Enable texture mapping.
  - `gl.glEnable(GL.GL_TEXTURE_2D)`
  - OpenGL supports 1-4 dimensional texture maps.
Define Image as a Texture

```c
glTexImage2D( target, level, components, w, h, border, format, type, texels );
```

- **target**: type of texture, e.g. `GL_TEXTURE_2D`
- **level**: used for mipmapping (discussed later)
- **components**: elements per texel
- **w, h**: width and height of `texels` in pixels
- **border**: used for smoothing (discussed later)
- **format and type**: describe `texels`
- **texels**: pointer to `texel` array

```c
gl.glTexImage2D(GL.GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL.GL_RGB, GL.GL_UNSIGNED_BYTE, myTexels);
```

Converting A Texture Image

- OpenGL requires texture dimensions to be powers of 2
- If dimensions of image are not powers of 2
  - `gluScaleImage();`
  - `ImageIO` in Java

Mapping a Texture

- Based on parametric texture coordinates
- `glTexCoord*()` specified at each vertex

```
(s, t) = (0.2, 0.8)
```

```
(s, t) = (0.4, 0.2)
```

```
(s, t) = (0.8, 0.4)
```

Typical Code

```c
glBegin(GL_POLYGON);
g1Color3f(r0, g0, b0); // if no shading used
g1Normal3f(u0, v0, w0); // if shading used
g1TexCoord2f(s0, t0);
g1Vertex3f(x0, y0, z0);
gColor3f(r1, g1, b1);
g1Normal3f(u1, v1, w1);
g1TexCoord2f(s1, t1);
g1Vertex3f(x1, y1, z1);
```

Note that we can use vertex arrays to increase efficiency
Texture Parameters

- OpenGL has a variety of parameters that determine how texture is applied
  - Wrapping parameters determine what happens if s and t are outside the (0,1) range
  - Filter modes allow us to use area averaging instead of point samples
  - Mipmapping allows us to use textures at multiple resolutions
  - Environment parameters determine how texture mapping interacts with shading

Wrapping Mode

Clamping: if s,t > 1 use 1, if s,t < 0 use 0
Wrapping: use s,t modulo 1

```c
gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_S, GL.GL_CLAMP);
gl.glTexParameteri(GL.GL_TEXTURE_2D, GL.GL_TEXTURE_WRAP_T, GL.GL_REPEAT);
```

Magnification and Minification

More than one texel can cover a pixel (minification) or more than one pixel can cover a texel (magnification)

Can use point sampling (nearest texel) or linear filtering (2 x 2 filter) to obtain texture values

Filter Modes

Modes determined by

```c
glTexParameteri(target, type, mode);
```

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

```c
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```
Mipmapped Textures

- **Mipmapping** allows for prefiltered texture maps of decreasing resolutions
- Lessens interpolation errors for smaller textured objects
- Declare mipmap level during texture definition
  
  ```
  glTexImage2D( GL_TEXTURE_*D, level, …
  
  )
  
  ```
- GLU mipmap builder routines will build all the textures from a given image
  
  ```
  gluBuild*DMipmaps( … )
  
  ```

Texture Functions

- Controls how texture is applied
  
  ```
  glTexEnv{fi}[v]( GL_TEXTURE_ENV, prop, param )
  
  ```
- **GL_TEXTURE_ENV_MODE** modes
  
  - **GL_MODULATE**: modulates with computed shade
  
  - **GL_BLEND**: blends with an environmental color
  
  - **GL_REPLACE**: use only texture color
- Set blend color with **GL_TEXTURE_ENV_COLOR**

Perspective Correction Hint

- Texture coordinate and color interpolation
  - either linearly in screen space
  - or using depth/perspective values (slower)
- Noticeable for polygons “on edge”
- **glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )**
  
  where hint is one of
  
  - **GL_DONT_CARE**
  
  - **GL_NICEST**
  
  - **GL_FASTEST**
### Texture Objects

- Texture is part of the OpenGL state
  - If we have different textures for different objects, OpenGL will be moving large amounts of data from processor memory to texture memory
- Recent versions of OpenGL have **texture objects**
  - One image per texture object
  - Texture memory can hold multiple texture objects

### Other Texture Features

- **Environment Maps**
  - Start with an image of the environment through a wide angle lens
    - Can be either a real scanned image or an image created in OpenGL
  - Use this texture to generate a spherical map
  - Use automatic texture coordinate generation
- **Multitexturing**
  - Apply a sequence of textures through cascaded texture units

### Compositing and Blending

### Opacity and Transparency

- Opaque surfaces permit no light to pass through
- Transparent surfaces permit all light to pass
- Translucent surfaces pass some light

\[
\text{translucency} = 1 - \text{opacity} (\alpha)\]
**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**

- Use a component of RGBA (or RGBα) 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
  
  \[
  s = [s_r, s_g, s_b, s_\alpha]
  \]
  
  \[
  d = [d_r, d_g, d_b, d_\alpha]
  \]

  Suppose that the source and destination colors are
  
  \[
  b = [b_r, b_g, b_b, b_\alpha]
  \]
  
  \[
  c = [c_r, c_g, c_b, c_\alpha]
  \]

  Blend as
  
  \[
  c' = [b_r s_r + c_r, d_r, b_g s_g + c_g d_g, b_b s_b + c_b d_b, b_\alpha s_\alpha + c_\alpha d_\alpha]
  \]

**OpenGL Blending and Compositing**

- Must enable blending and pick source and destination factors
  
  \[
  glEnable(GL_BLEND)
  \]
  
  \[
  glBlendFunc(source_factor, destination_factor)
  \]

- Only certain factors supported
  - GL_ZERO, GL_ONE
  - GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA
  - GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA
  - See Redbook 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 \((R_1,G_1,B_1,\alpha_1)\)
• Select `GL_SRC_ALPHA` and `GL_ONE_MINUS_SRC_ALPHA` as the source and destination blending factors
  \[ R'_1 = \alpha_1 R_1 + (1-\alpha_1) R_0, \ldots \]
• Note this formula is correct if polygon is either opaque or transparent

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 `glDepthMask(GL_FALSE)` which makes depth buffer read-only
• Sort polygons first to remove order dependency

Accumulation Buffer

• Compositing and blending are limited by resolution of the frame buffer
  – Typically 8 bits per color component
• Adding several colors have problems:
  – Washed out colors
  – Loss of accuracy/color resolution if scaled
• The accumulation buffer is a high resolution buffer (16 or more bits per component) that avoids this problem
• Write into it or read from it with a scale factor
• Slower than direct compositing into the frame buffer
Applications

- Compositing
- Image Filtering (convolution)
- Whole scene antialiasing
- Motion effects

Vertex and fragment shaders

Introduction

- Recent major advance in real time graphics is programmable pipeline
  - First introduced by NVIDIA GForce 3
  - Supported by high-end commodity cards
    - NVIDIA, ATI, 3D Labs
  - Software Support
    - DirectX 8, 9, 10
    - OpenGL Extensions
    - OpenGL Shading Language (GLSL)
    - Cg

Background

- Two components
  - Vertex programs (shaders)
  - Fragment programs (shaders)
- Requires detailed understanding of two seemingly contradictory approaches
  - OpenGL pipeline
    - Real time
  - RenderMan ideas
    - offline
**Black Box View**

**Vertex Processor**
- Takes in vertices
  - Position attribute
  - Possibly color
  - OpenGL state
- Produces
  - Position in clip coordinates
  - Vertex color

---

**Fragment Processor**
- Takes in output of rasterizer (fragments)
  - Vertex values have been interpolated over primitive by rasterizer
- Outputs a fragment
  - Color
  - Texture
- Fragments still go through fragment tests
  - Hidden-surface removal

---

**Programmable Shaders**
- Replace fixed function vertex and fragment processing by programmable processors called **shaders**
- Can replace either or both
- If we use a programmable shader we must do *all* required functions of the fixed function processor
Vertex Shader Applications

• Moving vertices
  – Morphing
  – Wave motion
  – Fractals
• Lighting
  – More realistic models
  – Cartoon shaders

Fragment Shader Applications

Per fragment lighting calculations

Simple example (GLSL)

• Vertex shader
  – Wave motion
• Fragment shader
  – Final color: texture + red

Fragment Shader Applications

Texture mapping

Simple example (GLSL)

• Vertex shader
  – Wave motion
• Fragment shader
  – Final color: texture + red

Simple example (GLSL)
attribute float wave;

void main() {
   vec4 vertex = gl_Vertex;
   vertex.z = (sin(wave + (vertex.x * 3.0)) + 
               sin(wave + (vertex.y * 4.0))) / 12.5;
   gl_Position =
               gl_ModelViewProjectionMatrix * vertex;
   gl_TexCoord[0] = gl_MultiTexCoord0;
}

uniform sampler2D tex;

void main() {
   vec4 color = texture2D(tex, gl_TexCoord[0].st);
   gl_FragColor = color + vec4(0.3, 0.0, 0.0, 1.0);
}

Next time

- Pipeline implementation
  - Clipping lines and polygons
  - Hidden surface removal
  - Scan conversion