Slides courtesy of Dave Shreiner, Ed Angel and Vicki Shreiner (SIGGRAPH 2001).
Pixel-based primitives

- **Bitmaps**
  - 2D array of bit masks for pixels
    - update pixel color based on current color

- **Images**
  - 2D array of pixel color information
    - complete color information for each pixel

- OpenGL doesn’t understand image formats
Programmable pixel storage and transfer operations

glBitmap(), glDrawPixels()
**Positioning Image Primitives**

```c
glRasterPos3f( x, y, z )
```

- raster position transformed like geometry
- discarded if raster position is outside of viewport
  - may need to fine tune viewport for desired results
glBitmap( width, height, xorig, yorig, xmove, ymove, bitmap )

- render bitmap in current color at
  \[
  \begin{bmatrix}
  x - xorig \\
  y - yorig
  \end{bmatrix}
  \]
- advance raster position by
  \[
  (xmove \quad ymove)
  \]
  after rendering
Rendering Fonts using Bitmaps

- OpenGL uses bitmaps for font rendering
  - each character is stored in a display list containing a bitmap
  - window system specific routines to access system fonts
    - glXUseXFont()
    - wglUseFontBitmaps()
Rendering Images

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

- render pixels with lower left of image at current raster position
- numerous formats and data types for specifying storage in memory
  - best performance by using format and type that matches hardware
Pixel Operations

- **Read Pixel**

  \[ \text{glReadPixels(x, y, width, height, format, type, pixels)} \]
  
  - read pixels form specified \((x, y)\) position in framebuffer
  - pixels automatically converted from framebuffer format into requested format and type

- **Framebuffer pixel copy**

  \[ \text{glCopyPixels(x, y, width, height, type)} \]
Pixel Zoom

`glPixelZoom(x, y)`

- expand, shrink or reflect pixels around current raster position
- fractional zoom supported

`glPixelZoom(1.0, -1.0);`
Storage and Transfer Modes

- **Storage modes control accessing memory**
  - byte alignment in host memory
  - extracting a subimage

- **Transfer modes allow modify pixel values**
  - scale and bias pixel component values
  - replace colors using pixel maps
Texture

- Motivation: to model realistic objects need surface detail: wood grain, stone roughness, scratches that affect shininess, grass, wallpaper.
- Use geometry, model surface detail with polygons; good for large scale detail, too expensive otherwise.
- Improvement: map an image of the details onto simple geometry
Texture mapping

- Texture mapping: adding surface detail by mapping texture patterns to the surface
Texture mapping methods

- 2D texture mapping: paint 2D pattern onto the surface
- Environmental (reflection) mapping
- Bump mapping: perturb surface normals to fool shading algorithms
- Procedural texture mapping, 3D texture
Examples
Environment Mapping

Yoshihiro Mizutani and Kurt Reindel
Texture array is a 2D image pattern

- With elements texels
- Value at a texel affects surface appearance
- The “texture map” determines how the pattern lies on the surface
2D texture mapping overview

Rendering uses the texture mapping

- Find surface that is front most at current pixel
- Find the surface patch corresponding to the pixel
- Find the part of the texture pattern corresponding to the surface patch
- Use that part of the texture pattern in setting the pixel color
2D texture mapping

- Source: 2D pattern from drawing, photo, procedure
- Destination: any surface, easier if surface given in parametric form
- The map from 2D texture coord to 3D object
- Texture mapping transformation: 2D screen coord $\rightarrow$ 3D object coord $\rightarrow$ 2D texture coord and back (see previous slide)
• Apply a 1D, 2D, or 3D image to geometric primitives

• Uses of Texturing
  – simulating materials
  – reducing geometric complexity
  – image warping
  – reflections
Texture Mapping

Geometry

Screen

Image
Images and geometry flow through separate pipelines that join at the rasterizer
- “complex” textures do not affect geometric complexity
2D Texture mapping in OpenGL

Pixel pipeline

Texture map done at rasterization stage
Texture Mapping in OpenGL

- Get hold of texture array
- Create texture object and specify texture for it
- Specify mode for applying texture to pixels
- Enable texture mapping
- Draw the scene providing both texture and geometric coordinates
Create texture object and texture for it

- Texel values could be up to 4D (R,G,B,A)
- Texturing is expensive. Texture objects similar to display lists: faster to bind(reuse) stored texture than load with glTexImage*D()
The texture (below) is a 256 x 256 image that has been mapped to a rectangular polygon which is viewed in perspective.
Applying Textures I

- Three steps
  1. specify texture
     - read or generate image
     - assign to texture
  2. assign texture coordinates to vertices
  3. specify texture parameters
     - wrapping, filtering
Applying Textures II

- specify textures in texture objects
- set texture filter
- set texture function
- set texture wrap mode
- set optional perspective correction hint
- bind texture object
- enable texturing
- supply texture coordinates for vertex
  - coordinates can also be generated
Using texture objects

- **Create texture names**, `glGenTextures()`,
- **Returns available id’s in second parameter which is passed by reference**
  
  ```c
  static GLuint texName;
  glGenTextures(1, &texName);
  ```

- **Create and use texture objects**:  
  ```c
  glBindTexture(GL_TEXTURE_2D, texName);
  ```

- **In first use: new texture object is created and assigned the name; subsequent uses: activate the texture object**
Using texture objects

- `glBindTexture()`, "sets" texture state, subsequent calls to `glTexImage`, `glTexParameters`, etc., store data in the active texture object.
- The data may include: texture image (mipmaps), width, height, border, internal format, texture properties (minmag filters, wrapping modes, ...).
2D texture mapping in OpenGL

- Provide texture array
  
  
  ```
  Glubyte my_texels[512][512][3];
  ```

- Specify which texture array to use
  
  ```
  glTexImage2D( GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB, GL_UNSIGNED_BYTE, my_texels);
  ```

- Enable texture mapping
  
  ```
  glEnable(GL_TEXTURE_2D);
  ```

- Specify map from texture array to object.
  
  - \( 0 \leq s, t \leq 1 \)
  
  - Assign texture coordinates to vertices
    
    ```
    glTexCoord2f(s, t);
    ```
Mapping texture coordinates to object

Example: map the array to quadrilateral

```cpp
glBegin(GL_QUAD);
glTexCoord2f(0.0,0.0);
glVertex3f(x1,y1,z1);
glTexCoord2f(1.0,0.0);
glVertex3f(x2,y2,z2);
glTexCoord2f(1.0,1.0);
glVertex3f(x3,y3,z3);
glTexCoord2f(0.0,1.0);
glVertex3f(x4,y4,z4);
glEnd();
```
Texture Objects

- Like display lists for texture images
  - one image per texture object
  - may be shared by several graphics contexts

- Generate texture names

  `glGenTextures( n, *texIds );`
Create texture objects with texture data and state
   
   `glBindTexture(target, id);`

Bind textures before using
   
   `glBindTexture(target, id);`
Define a texture image from an array of texels in CPU memory

\[
\text{glTexImage2D}( \text{target}, \text{level}, \text{components}, \ w, \ h, \ \text{border}, \ \text{format}, \ \text{type}, \ \ast\text{texels} );
\]

- dimensions of image must be powers of 2

Texel colors are processed by pixel pipeline
- pixel scales, biases and lookups can be done
Converting A Texture Image

- If dimensions of image are not power of 2
  
  ```c
  gluScaleImage( format, w_in, h_in,
                  type_in, *data_in, w_out, h_out,
                  type_out, *data_out );
  ```

  - *in is for source image
  - *out is for destination image

- Image interpolated and filtered during scaling
Specifying a Texture: Other Methods

- Use frame buffer as source of texture image
  - uses current buffer as source image
  - `glCopyTexImage2D(...)`
  - `glCopyTexImage1D(...)`

- Modify part of a defined texture
  - `glTexSubImage2D(...)`
  - `glTexSubImage1D(...)`

- Do both with `glCopyTexSubImage2D(...), etc.`
Mapping a Texture

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

![Diagram showing texture mapping and correspondence between texture space and object space with coordinates and vertices labeled a, b, c.](image)
Generating Texture Coordinates

- Automatically generate texture coords
  
  \[ \text{glTexGen}\{\text{ifd}\}[v]() \]

- specify a plane \[ Ax + By + Cz + D = 0 \]
  - generate texture coordinates based upon distance from plane

- generation modes
  - GL_OBJECT_LINEAR
  - GL_EYE_LINEAR
  - GL_SPHERE_MAP
Tutorial: Texture

GLfloat border_color[] = { 1.00, 0.00, 0.00, 1.00 };  
GLfloat env_color[] = { 0.00, 1.00, 0.00, 1.00 };  

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, border_color);  
glTexParameteri(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, env_color);  

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);  
glTexParameteri(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  

GLenum(GL_TEXTURE_2D);  
glBuild2DMipmaps(GL_TEXTURE_2D, 3, w, h, GL_RGB, GL_UNSIGNED_BYTE, image);  

glColor4fv( 0.60, 0.60, 0.60, 1.00 );  

begin(GL_POLYGON);  

glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, 0.0 );  
glTexCoord2f( 1.0, 0.0 ); glVertex3f( 1.0, -1.0, 0.0 );  
glTexCoord2f( 1.0, 1.0 ); glVertex3f( 1.0, 1.0, 0.0 );  
glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0, 1.0, 0.0 );  
glEnd();  

Click on the arguments and move the mouse to modify values.
Texture Application Methods

- **Filter Modes**
  - minification or magnification
  - special mipmap minification filters

- **Wrap Modes**
  - clamping or repeating

- **Texture Functions**
  - how to mix primitive’s color with texture’s color
    - blend, modulate or replace texels
Filter Modes

Example:

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

- Mipmap allows for prefiltered texture maps of decreasing resolutions

- Lessens interpolation errors for smaller textured objects

- Declare mipmap level during texture definition
  
  ```c
  glTexImage*D( GL_TEXTURE_*D, level, ... )
  ```

- GLU mipmap builder routines
  
  ```c
  gluBuild*DMipmaps( ... )
  ```

- OpenGL 1.2 introduces advanced LOD controls
Wrapping Mode

- Example:

  ```
  glTexParameteri( GL_TEXTURE_2D,
                  GL_TEXTURE_WRAP_S, GL_CLAMP )
  glTexParameteri( GL_TEXTURE_2D,
                  GL_TEXTURE_WRAP_T, GL_REPEAT )
  ```

  ![Texture Example]
Texture Functions

- Controls how texture is applied
  
  ```c
  glTexEnv{fi}[v]( GL_TEXTURE_ENV, prop, param )
  ```

- `GL_TEXTURE_ENV_MODE` modes
  - `GL_MODULATE`
  - `GL_BLEND`
  - `GL_REPLACE`

- 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”**

```c
glHint( GL_PERSPECTIVE_CORRECTION_HINT, hint )
```

where `hint` is one of

- `GL_DONT_CARE`
- `GL_NICEST`
- `GL_FASTEST`
Is There Room for a Texture?

- Query largest dimension of texture image
  - typically largest square texture
  - doesn’t consider internal format size

    glGetIntegerv( GL_MAX_TEXTURE_SIZE, &size )

- Texture proxy
  - will memory accommodate requested texture size?
  - no image specified; placeholder
  - if texture won’t fit, texture state variables set to 0
    - doesn’t know about other textures
    - only considers whether this one texture will fit all of memory
Texture Residency

- **Working set of textures**
  - high-performance, usually hardware accelerated
  - textures must be in texture objects
  - a texture in the *working set* is *resident*
  - for residency of current texture, check `GL_TEXTURE_RESIDENT` state

- **If too many textures, not all are resident**
  - can set priority to have some kicked out first
  - establish 0.0 to 1.0 priorities for texture objects
Immediate Mode versus Display Listed Rendering

- **Immediate Mode Graphics**
  - Primitives are sent to pipeline and display right away
  - No memory of graphical entities

- **Display Listed Graphics**
  - Primitives placed in display lists
  - Display lists kept on graphics server
  - Can be redisplayed with different state
  - Can be shared among OpenGL graphics contexts
Immediate Mode vs. Display Lists

Immediate Mode

- Polynomial Evaluator
- Per Vertex Operations & Primitive Assembly

CPU

Display List

- Rasterization

Pixel Operations

- Texture Memory
- Per Fragment Operations

Frame Buffer
Display Lists

- Creating a display list
  ```c
  GLuint id;
  void init( void )
  {
    id = glGenLists( 1 );
    glNewList( id, GL_COMPILE );
    /* other OpenGL routines */
    glEndList();
  }
  ```

- Call a created list
  ```c
  void display( void )
  {
    glCallList( id );
  }
  ```
Display Lists

- Not all OpenGL routines can be stored in display lists
- State changes persist, even after a display list is finished
- Display lists can call other display lists
- Display lists are not editable, but you can fake it
  - make a list (A) which calls other lists (B, C, and D)
  - delete and replace B, C, and D, as needed
Display Lists and Hierarchy

- Consider model of a car
  - Create display list for chassis
  - Create display list for wheel

```c
glNewList( CAR, GL_COMPILE );
glCallList( CHASSIS );
glTranslatef( ... );
glCallList( WHEEL );
glTranslatef( ... );
glCallList( WHEEL );
...

GLfloat ENDList();
```
**Vertex Arrays**

- Pass arrays of vertices, colors, etc. to OpenGL in a large chunk

```c
glVertexPointer( 3, GL_FLOAT, 0, coords )
glColorPointer( 4, GL_FLOAT, 0, colors )
glEnableClientState( GL_VERTEX_ARRAY )
glEnableClientState( GL_COLOR_ARRAY )
glDrawArrays( GL_TRIANGLE_STRIP, 0, numVerts );
```

- All active arrays are used in rendering
Display Lists or Vertex Arrays?

- May provide better performance than immediate mode rendering
- Display lists can be shared between multiple OpenGL context
  - reduce memory usage for multi-context applications
- Vertex arrays may format data for better memory access
Alpha: the 4$^{th}$ Color Component

- **Measure of Opacity**
  - simulate translucent objects
    - glass, water, etc.
  - composite images
  - antialiasing
  - ignored if blending is not enabled

```gl
glEnable( GL_BLEND )
```
Blending

- Combine pixels with what’s in already in the framebuffer

$$\tilde{C}_r = src \; \tilde{C}_f + dst \; \tilde{C}_p$$

**Blending Equation**

- Fragment $(src)$
- Framebuffer Pixel $(dst)$
- Blended Pixel
- Per Vertex
- Texture
- Raster
- Frag
- FB
- CPU
- DL

**glBlendFunc** $(src, dst)$
Multi-pass Rendering

- Blending allows results from multiple drawing passes to be combined together
  - enables more complex rendering algorithms

Example of bump-mapping done with a multi-pass OpenGL algorithm
Bump mapping

- 2D Texture map creates odd looking rough surfaces
- Bump mapping: texture map that alters surface normals.
  - Use texture array to set a function which perturbs surface normals
  - Altered normals match a bumpy surface
  - Applying illumination model to the new normals shades the bumps correctly
Bump mapping

- Bump map is in texture array: $d(s,t) << 1$
- $p$ point on the surface corresponding to texture coordinates $s,t$.
- $N$ the normal at $p$
- $p'$ the bump point for $p$
  \[ p' = p + d(s,t)N \]

We actually do not "bump" the surface, just the normal at $p$.
- $N'$ the normal at $p'$. This normal used by the illumination model at $p$. 
Bump mapping

How to get $\mathbf{N}'$:

• given two vectors tangent to the bumpy surface, $\mathbf{N}'$ is their cross product

• The two vectors follow from the partial derivatives of the $\mathbf{p}'$ equation wrt $u,v$

  \[ \mathbf{p}' = \mathbf{p} + d(s,t)\mathbf{N} \]

• These partial derivatives expressed in terms of the derivatives of $d(s,t)$ as $s,t$ change
Mapping the 2D texture to the surface

- The map: 2D texture \((s,t)\) \(\rightarrow\) 3D object \((x,y,z)\)
- Mapping onto triangle is not difficult
- Mapping onto triangular mesh is more difficult (have to handle texture discontinuity)
- Mapping onto parametric surface is easier
- Alternative: use an intermediate parametric surface (cylinder, sphere)
Mapping

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

Texture Space

Object Space

(s, t) = (0.2, 0.8)

(0.8, 0.4)

(0.4, 0.2)
Mapping texture onto parametric surface

- **Point on the parametric surface**

  \[ p : x = x(u, v), \quad y = y(u, v), \quad z = z(u, v) \]

- **The map from texture to the parametric coord, invertible**

  \[ u = as + bt + c \]
  \[ v = ds + et + f \]
Mapping texture onto parametric surface

- Example of a linear mapping
- Does not take into account curvature of surface
- Equal size texture patches are stretched to fit various areas
Mapping texture to a surface using an intermediate surface

- **Two-step mapping**
  - Map the texture to a simple intermediate surface (sphere, cylinder, cube)
  - Map the intermediate surface (with the texture) onto the surface being rendered
Two-step mapping example

- **cylinder**: \( x = r \cos(2 \pi u) \)
  \( y = r \sin(2 \pi u) \)
  \( z = v \cdot h \)
  
  first step: \( u = s, \ v = t \)

- **sphere**
- **cube**
Two-step mapping example

- Second-step: map intermediate surface to the surface being rendered
- Various strategies: a, b, c
The texture mapping transformation
Texture mapping transformation

Consider surface visible at current pixel.
Find the patch on the surface that corresponds to it.
- Map screen coord of pixel corners back to object
- Find texels that map to the surface patch
- If multiple texels lie on patch combine them:
  weighted avg; supersampling with postfiltering
Environmental Mapping

- Put texture on a highly reflective object by picking up texture from the environment in which the object is immersed.
- Realized as two-step process
  - Project the environment (excluding the object) onto an intermediate surface.
  - Place object back, and map texture from intermediate surface to object
Bump mapping

- 2D Texture map creates odd looking rough surfaces
- Bump mapping: texture map that alters surface normals.
  - Use texture array to set a function which perturbs surface normals
  - Altered normals match a bumpy surface
  - Applying illumination model to the new normals shades the bumps correctly
Bump mapping

- Bump map is in texture array: \( d(s,t) << 1 \)
- \( p \) point on the surface corresponding to texture coordinates \( s,t \).
- \( N \) the normal at \( p \)
- \( p' \) the bump point for \( p \)
  \[
  p' = p + d(s,t)N
  \]
We actually do not “bump” the surface, just the normal at \( p \).
- \( N' \) the normal at \( p' \). This normal used by the illumination model at \( p \).
Bump mapping

How to get $N'$:

- given two vectors tangent to the bumpy surface, $N'$ is their cross product

- The two vectors follow from the partial derivatives of the $p'$ equation wrt $u,v$

  $$p' = p + d(s,t)N$$

- These partial derivatives expressed in terms of the derivatives of $d(s,t)$ as $s,t$ change