Recall: Step 1: Create Texture Object

- OpenGL has **texture objects** (multiple objects possible)
  - 1 object stores 1 texture image + texture parameters
- First set up texture object

```c
GLuint mytex[1];
glGenTextures(1, mytex);               // Get texture identifier
glBindTexture(GL_TEXTURE_2D, mytex[0]); // Form new texture object
```

- Subsequent texture functions use this object
- Another call to `glBindTexture` with new name starts new texture object
**Recall: Step 2: Specifying a Texture Image**

- Define input picture to paste onto geometry
- Define texture image as array of *texels* in CPU memory
  ```c
  Glubyte my_texels[512][512][3];
  ```
- Read in scanned images (jpeg, png, bmp, etc files)
  - If uncompressed (e.g. bitmap): read into array from disk
  - If compressed (e.g. jpeg), use third party libraries (e.g. Qt, devil) to uncompress + load
### Recall: Specify Image as a Texture

**Tell OpenGL:** this image is 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 (0: highest resolution. More later)
- **components:** elements per texel
- **w, h:** width and height of `texels` in pixels
- **border:** used for smoothing (discussed later)
- **format, type:** describe texels
- **texels:** pointer to texel array

**Example:**

```c
glTexImage2D(GL_TEXTURE_2D, 0, 3, 512, 512, 0, GL_RGB, 
             GL_UNSIGNED_BYTE, my_texels);
```
Recall: Step 3: Assign Object Corners to Texture Corners

- Each object corner \((x,y,z)\) \(\Rightarrow\) image corner \((s, t)\)
  - E.g. object \((200,348,100)\) \(\Rightarrow\) \((1,1)\) in image
- Programmer establishes this mapping
- Target polygon can be any size/shape
Recall: Step 5: Passing Texture to Shader

- Pass vertex, texture coordinate data as vertex array
- Set texture unit

```c
offset = 0;
GLuint vPosition = glGetUniformLocation(program, "vPosition");
enableVertexAttribArray(vPosition);
glVertexAttribPointer(vPosition, 4, GL_FLOAT, GL_FALSE, 0,BUFFER_OFFSET(offset));
offset += sizeof(points);
GLuint vTexCoord = glGetUniformLocation(program, "vTexCoord");
enableVertexAttribArray(vTexCoord);
glVertexAttribPointer(vTexCoord, 2, GL_FLOAT, GL_FALSE, 0, BUFFER_OFFSET(offset));

// Set the value of the fragment shader texture sampler variable
// ("texture") to the the appropriate texture unit.

glUniform1i(glGetUniformLocation(program, "texture"), 0);
```
Recall: Step 6: Apply Texture in Shader (Fragment Shader)

- Textures applied in fragment shader
- Samplers return a texture color from a texture object

```glsl
in vec4 color; //color from rasterizer
in vec2 texCoord; //texture coordinate from rasterizer
uniform sampler2D texture; //texture object from application

void main() {
    gl_FragColor = color * texture2D(texture, texCoord);
}
```
6 Main Steps to Apply Texture

1. Create texture object
2. Specify the texture
   - Read or generate image
   - assign to texture (hardware) unit
   - enable texturing (turn on)
3. Assign texture (corners) to Object corners
4. Specify texture parameters
   - wrapping, filtering
5. Pass textures to shaders
6. Apply textures in shaders

still haven’t talked about setting texture parameters
Step 4: Specify Texture Parameters

• Texture parameters control how texture is applied
  • **Wrapping parameters** used if s,t outside (0,1) range
    - **Clamping:** if s,t > 1 use 1, if s,t <0 use 0
    - **Wrapping:** use s,t modulo 1

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP )
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT )
```
Step 4: Specify Texture Parameters
Mipmapped Textures

- **Mipmapping** pre-generates prefiltered (averaged) texture maps of decreasing resolutions
- Declare mipmap level during texture definition
  
  ```
  glTexImage2D( GL_TEXTURE_*D, level, … )
  ```
Magnification and Minification

**Magnification:** Stretch small texture to fill many pixels

**Minification:** Shrink large texture to fit few pixels
Step 4: Specify Texture Parameters
Texture Value Lookup

How about coordinates that are not exactly at the intersection (pixel) positions?

A) Nearest neighbor
B) Linear Interpolation
C) Other filters
Example: Texture Magnification

- 48 x 48 image projected (stretched) onto 320 x 320 pixels

Nearest neighbor filter

Bilinear filter (avg 4 nearest texels)

Cubic filter (weighted avg. 5 nearest texels)
Texture mapping parameters

1) Nearest Neighbor (lower image quality)
   \[ \text{glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);} \]
   Or \( \text{GL_TEXTURE_MAX_FILTER} \)

2) Linear interpolate the neighbors (better quality, slower)
   \[ \text{glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);} \]
Dealing with Aliasing

- Point sampling of texture can lead to aliasing errors.
Area Averaging

Better but slower option is *area averaging*
Other Stuff

- Wrapping texture onto curved surfaces. E.g. cylinder, can, etc

\[
s = \frac{\theta - \theta_a}{\theta_b - \theta_a} \quad t = \frac{z - z_a}{z_b - z_a}
\]

- Wrapping texture onto sphere

\[
s = \frac{\theta - \theta_a}{\theta_b - \theta_a} \quad s = \frac{\phi - \phi_a}{\phi_b - \phi_a}
\]

- Bump mapping: perturb surface normal by a quantity proportional to texture
Environment Mapping

- Environmental mapping is a way to create the appearance of highly reflective surfaces.
Reflecting the Environment

Sphere of environment around object

Cube of environment around object
Cube mapping

- Need to compute reflection vector, \( r \)
- Use \( r \) by for lookup
- OpenGL: hardware support for cube maps
Environment Map

Use reflection vector to locate texture in cube map
Cube Environment Map Example

- Six textures: one for each face cube surrounding object
- Load 6 textures separately into 1 OpenGL cubemap
Cube Maps

- Loaded cube map texture can be accessed in GLSL through cubemap sampler

  \[
  \text{vec4 texColor} = \text{textureCube(mycube, texcoord)};
  \]

- Texture coordinates must be 3D
OpenGL Stages

- After projection, several stages before objects drawn to screen
- These stages are non-programmable

Vertex shader: programmable

In hardware: NOT programmable
Hardware Stage: Primitive Assembly

- Up till now: Transformations and projections applied to vertices individually
- **Primitive assembly**: After transforms, projections, individual vertices grouped back into primitives
- E.g. **v6, v7 and v8** grouped back into triangle
Hardware Stage: Clipping

- After primitive assembly, subsequent operations are per-primitive
- **Clipping**: Remove primitives (lines, polygons, text, curves) outside view frustum (canonical view volume)
Rasterization

- Determine which pixels that primitives map to
  - Fragment generation
  - Rasterization or scan conversion
Fragment Processing

- Some tasks deferred until fragment processing

Hidden Surface Removal

Transformation
Projection

Hidden surface Removal
Antialiasing

Modeling → Geometric processing → Rasterization → (Fragment processing) → Frame buffer
Clipping

- 2D and 3D clipping algorithms
  - 2D against clipping window
  - 3D against clipping volume

- 2D clipping
  - Lines (e.g. dino.dat)
  - Polygons
  - Curves
  - Text
Clipping 2D Line Segments

- **Brute force approach**: compute intersections with all sides of clipping window
  - Inefficient: one division per intersection
2D Clipping

- **Better Idea:** eliminate as many cases as possible without computing intersections
- Cohen-Sutherland Clipping algorithm
Clipping Points

Determine whether a point \((x,y)\) is inside or outside of the world window?

If \((xmin <= x <= xmax)\) and \((ymin <= y <= ymax)\)

then the point \((x,y)\) is inside
else the point is outside
Clipping Lines

3 cases:

Case 1: All of line in
Case 2: All of line out
Case 3: Part in, part out
Clipping Lines: Trivial Accept

Case 1: All of line in
Test line endpoints:

\[
X_{\text{min}} \leq P1.x, \ P2.x \leq X_{\text{max}} \quad \text{and} \quad Y_{\text{min}} \leq P1.y, \ P2.y \leq Y_{\text{max}}
\]

Note: simply comparing x,y values of endpoints to x,y values of rectangle

Result: trivially accept.
Draw line in completely
Clipping Lines: Trivial Reject

**Case 2:** All of line out
Test line endpoints:

- \( p1.x, p2.x \leq X_{min} \) OR
- \( p1.x, p2.x \geq X_{max} \) OR
- \( p1.y, p2.y \leq y_{min} \) OR
- \( p1.y, p2.y \geq y_{max} \)

**Note:** simply comparing x,y values of endpoints to x,y values of rectangle

**Result:** trivially reject.
Don’t draw line in
Clipping Lines: Non-Trivial Cases

**Case 3:** Part in, part out

Two variations:
- One point in, other out
- Both points out, but part of line cuts through viewport

Need to find inside segments

Use similar triangles to figure out length of inside segments

\[
\frac{d}{dely} = \frac{e}{delx}
\]
Clipping Lines: Calculation example

If chopping window has (left, right, bottom, top) = (30, 220, 50, 240), what happens when the following lines are chopped?

(a) p1 = (40,140), p2 = (100, 200)

(b) p1 = (20,10), p2 = (20, 200)

(c) p1 = (100,180), p2 = (200, 250)
int clipSegment(Point2& p1, Point2& p2, RealRect W)
{
    do{
        if(trivial accept) return 1; // whole line survives
        if(trivial reject) return 0;  // no portion survives
        // now chop
        if(p1 is outside)
            // find surviving segment
            {
                if(p1 is to the left) chop against left edge
                else if(p1 is to the right) chop against right edge
                else if(p1 is below) chop against the bottom edge
                else if(p1 is above) chop against the top edge
            }
else // p2 is outside
    // find surviving segment
{
    if(p2 is to the left) chop against left edge
    else if(p2 is to right) chop against right edge
    else if(p2 is below) chop against the bottom edge
    else if(p2 is above) chop against the top edge
}
}while(1);
Using Outcodes to Speed Up Comparisons

- Encode each endpoint into outcode (what quadrant)

\[ b_0 b_1 b_2 b_3 \]

\begin{align*}
  b_0 &= 1 \text{ if } y > y_{\text{max}}, \ 0 \text{ otherwise} \\
  b_1 &= 1 \text{ if } y < y_{\text{min}}, \ 0 \text{ otherwise} \\
  b_2 &= 1 \text{ if } x > x_{\text{max}}, \ 0 \text{ otherwise} \\
  b_3 &= 1 \text{ if } x < x_{\text{min}}, \ 0 \text{ otherwise}
\end{align*}

- Outcodes divide space into 9 regions
- Trivial accept/reject becomes bit-wise comparison
References