**Graphics Pipeline**: vertex processing ==vertex stream=>triangle processing==triangle stream=>rasterization==fragment stream=>fragment processing==shaded fragments=>framebuffer operations=>display. **Space transform**: Local/model space (modelling coordinates)==model transform==>World/global space (world coordinates)==viewing transform==>View/camera space (viewing coordinates)==projection transform==>Clip/projection space (projection coordinates)==perspective division==>Normalized Device Coordinates (NDC)== viewport transform==>Screen space (clip space after rasterization). **Clip to Screen**: (x,y)->(u,v). , , , . **Cohen-Sutherland**: 4 digit binary value, 1=outside of border, 0=inside. 1 in same bit position for both end-points is completely outside. Use region code to test line crosses (inverted bits). **Intersection point calculations**: , = or . **Half-plane test**: (p-q)\*n<0, p inside P; (p-q)\*n=0, p on P; (p-q)\*n>0, p outside P. **Orthographic proj**: focal point at infinity; rays parallel and orthogonal to image plane. **Perspective proj**: focal point behind image plane. **Lec4-Shader** **uniform**=specify surface color. **attributes**=specify how to pull data from buffers and provide to vertex shader. **varying**=interpolate between vertex shader and fragment shader. **Two steps**: 1. Create and load buffers with data 2. Bind each buffer to a shader attribute. **GLSL**=hardware-independence, no double-precision float. **Use shader in OpenGL**: 1. Create empty shader objects with glCreateShader. 2. Load source code into shader with glShaderSource. 3. Compile shader with glCompileShader 4. Create empty program object with glCreateProgram. 5. Bind shaders to program with glAttachShader. 6. Link program with glLinkProgram 7. Register program for use with glUseProgram. **WebGL Data Flow**: 1. Take vertex data and place into VBOs. 2. Stream VBO data to vertex shader. 3. Send vertex index info using drawArrays()/drawElements() 4. VS set screen position of each vertex, which are then passed to FS. 5. Primitive assembly: GPU produce primitives using vertices and indices. 6. Clipping: rasterizer discards primitive part outside viewport. 7. Parts inside viewport are broken down into pixel fragments. 8. Rasterization: vertex values interpolated across each fragment. 9. Fragments with interpolated values are passed into FS. 10. FS sets color values, add texture and lighting. 11. Fragments be discarded/passed into frame buffer. 12. FS optionally use stencil buffer/depth buffer to choose which fragments to write onto final image. 13. Image is passed into drawing buffer for later usage as texture data. **Backward Texture map**: screen coordinates -> object coordinates -> texture coordinates. **3D Texture map**: a stack of bitmaps, memory intensive. Applications=medical 3D images, solid materials. **Procedural Texture Map**: define function that returns color dependent on value of s, (s,t)/(s,t,r). **Two-part mapping**: 1) find an intermediate object closest to 3D object that we want to map texture onto, map texture on this intermediate object 2) map texture from inter to actual object.**Common inter objects**: cylinder=convert rectangular coordinates (x,y,z) to cylindrical (r,phi,h), sphere (used in environment mapping)=convert rectangular coordinates (x,y,z) to spherical (theta,phi), cube. **Parametric Surfaces**: If geometric object is defined parametrically, an additional mapping function involving (u,v) space is needed. E.g., x=rcos(2piu),y=rsin(2piu),z=v/h. **Sampling Texture solutions**: 1) nearest neighbour sampling=take nearest known value. 2) Bilinear Interpolation=calculate direct distances to 4 neighbours and take weighted average depend on distance to those values. 3) Mipmaps: use if we downscale > 2 times with bilinear. Exactly 2 times smaller versions of the texture.

**NPOT**: Not Power-of-Two, solution={{best:reduce image into square image as 2^n}{second best:make texture images same as native resolution of monitor}}. **gl.texParameteri()**: can disable Mipmapping and UV repeat. **FS Note**: fragment color is computed by fetching texel (pixel within texture) that sampler says best maps to fragment position. **Lec7-Light** **Multiple Textures**: use constant values as array indices, adjust LoadTexture and SetupTexture functions to handle multiple textures. **Lighting model**: technique that defines way to calc final color given all components required for it. **Diffuse surface**: not matter at which angle we look at the material because diffusion of light. On which angle light reaches material matters. **Lambert Lighting Model**: models diffuse reflection of light; surface normal & light source direction are normalized=>calc cos with algebraic dot product, not geometric. Only use positive cosine=>subtract light if light source is on other side of material. **illumination percentage**: cosine of angle between surface normal and direction towards light. E.g., cos=1=>0-degree angle, surface unit receive 100% light. 30-degree angle, 87%=cos(30 deg). **Diffuse model**=I\_d=I\_p\*k\_d\*cos(theta)=…\*cos(L\*N), I\_p=incident light intensity, k\_d=diffuse reflectivity constant. **Ambient Light**: to approx indirect illumination, add an ambient term (constant term I\_a\*k\_a) => non-illuminated areas not be totally black. Disad=other parts more illuminated. **Total Light**=diffuse+ambient=I\_a\*k\_a+I\_p\*k\_d\*cos(L\*N). **Attenuation**: introduces factor modifies light intensity depending on distance from viewer. **Specular Reflection (镜面)**: reflected light not same in all directions. Mirror=perfect specular reflection. **Phong Lighting Model**: calc specular highlight=cos^2 of viewer direction and reflected light. **total intensity**=diffuse+ambient+k\_s\*cos^n(phi), k\_s=specular constant. **Lec8-Shading** **Shading model**: technique that defines when and where do we calculate components needed for light calc. **Steps to shading**: 1. Model (model of real/abstract object) 2. Surface mesh (discretize surface into mesh of polygons, mostly triangles) 3. Normal vectors 4. Materials (to add material spectral properties). 5. Lights 6. Final shading. **Coarse mesh**: objects with low uniform curvature. **Dense mesh**: 1. high non-uniform curvature. 2. Deformable objects or rough surfaces. **Store and preserve mesh**: in ASCII format (.obj). **Materials**: 1. model as triplets of RGB: constant k\_d\*k\_s. 2. texture can be added for realism. **Light sources**: 1. Directional 2. Positional 3. Areal **Normals**: 1) critical to model light-object interaction. 2) represent orientation of surface. **Light reflection models**: color is calculated based on normals. 1. Flat shading: use one surface normal per polygon, uniform color on this polygon 2. Gouraud shading: per vertex, different normal per vertex, color interpolated over polygon=> handles badly materials that have a specular reflection. 3. Phong shading: normal from vertices is interpolated, color is calc per fragment, account for specular reflection. **Lec9-Transformation**: **Vector Cross Product**: U cross V=(|U||V| sin(theta))W, W=normal vector. **Windows and Viewports**: viewport mapping=world coordinates must be mapped into screen coordinates, This mapping is homogeneous transformation.