Local/model space (modelling coordinates)=>World/global space (world coordinates)=>View/camera space (viewing coordinates)=>Clip/projection space (projection coordinates)=>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.