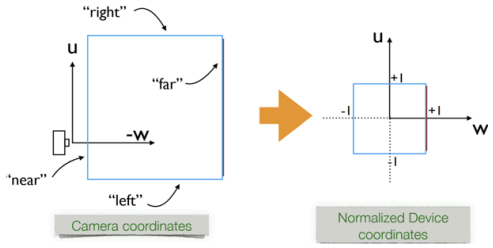
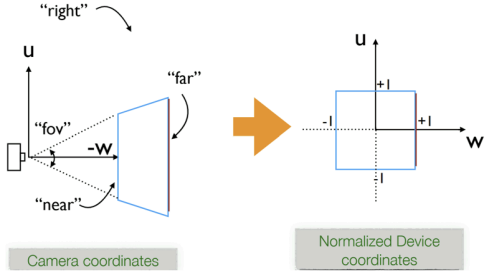
VIEWING PIPELINE

(1) model matrix (rotate, scale, translate, etc.)
world coords is absolute to the origin coordinate
(2) lookAt(out, eye, center, up) → mat4
world (x,y,z) to camera (u,v,w)
camera coords: (x-width, y-height, z-depth)
(3) projection transform view frustum to **[-1,1] cube**
ortho(out, l, r, b, t, near, far) → mat4



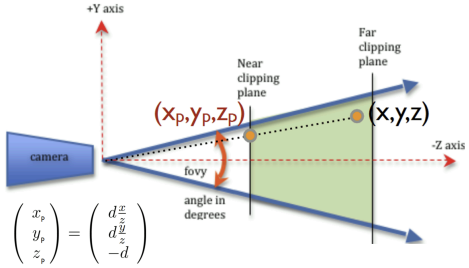
perspective(out, fovy, aspect, near, far) → mat4



(4) viewpoint transforms [-1,1] cube to screen (x,y)
NDC to canvas is 2D scaling and transform

(3) PROJECTION

ortho projection = closer objects are same size
– **left(l) right(r) top(t) bottom(b)** width-height bounds
– **near - far** depth bounds
perspective projection = closer objects are smaller
– **fovy** vertical FoV in radians
– **aspect** viewport width/height
– **near - far** depth bounds



$$\begin{pmatrix} x_p \\ y_p \\ z_p \end{pmatrix} = \begin{pmatrix} d\frac{x}{z} \\ d\frac{y}{z} \\ -d \end{pmatrix}$$

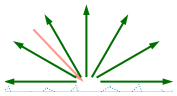
focal length d distance from eye to projection plane
(x,y,z) in view frustum projects to (x_p,y_p,z_p) = (dx/z, dy/z, -d) in image plane
division by z → coordinates not linear anymore

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 1/d & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ -z \\ z/d \end{pmatrix} \equiv \begin{pmatrix} dx/z \\ dy/z \\ -d \\ 1 \end{pmatrix} \Leftrightarrow \begin{pmatrix} x_p = dx/z \\ y_p = dy/z \\ z_p = -d \end{pmatrix}$$

homo representation of perspective projection
f(u) = uBP returns a vec3, P matrix is 4x3
fromRotation(out, rad, axis) → mat4 rotates by rad around axis, much faster than plain rotate

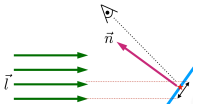
LIGHTING

Lambertian Reflectance: any ray shot to the surface reflects uniformly in any direction



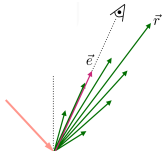
Diffuse: results from lambertian matte appearance

$$I_d = k_d(\vec{l} \cdot \vec{n})$$

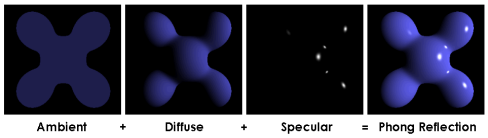


brighter pixels → perpendicular to light source, independent of view direction
Specular: light reflects more symmetric

$$I_s = k_s(\vec{r} \cdot \vec{e})^s$$



concentrating the light → shiny
r → dir of most light (reflect l over n)
e → view direction, has to be in the same space as r
s → specular exponent (shininess, more local specular)
Ambient: general, uniform light present in a scene, independent of any specific light source (assuming the object itself is the light source)
constant regardless of eye/light/object placement



Blinn-Phong → ambient + diffuse + specular

GPU PIPELINE (IN ORDER)

Command Buffer: Triangle of a mesh are dispatched to GPU for processing
Vertex Queue: broken up into vertices, fed into queue
Vertex Processing (TCL): Execute computations on vertices (example: transformations to NDC), then encoded in the vertex shader
Assembly: Assembles vertices back to triangles
Rasterize: Convert each triangle into a collection of pixels that covers spatial extent. Uses barycentric coords

$$\vec{x}_{\text{frag}} = w_1\vec{x}_1 + w_2\vec{x}_2 + w_3\vec{x}_3$$

Pixel is inside the triangle and rasterized into fragments if all 3 weights are positive. Weights always add up to 1
Varying variables uses the same weights to interpolate
Pixel Processing: Determine color of pixels, sent to fragment shader
Pixel Tests: Determine which pixels are rejected and which written to framebuffer (Z Testing) uses Z buffer

GLSL

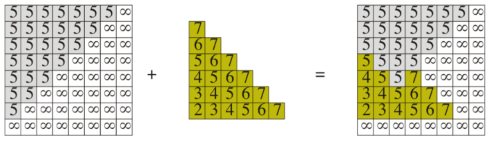
Type qualifiers: v/s, f/s means appear in which shader
Const: compile time constants (v/s, f/s). hardcoded
Attribute: vertex properties from host program (v/s)
Uniform: Constants for both shaders (v/s, f/s)

Varying: how we communicate from v/s to f/s, only used within shaders
Assigned on per-vertex basis by v/s, interpolated on per-fragment by f/s
Value in f/s interpolated from v/s using same weights from the barycentric weights (from rasterization)
Computing lighting or shading (*gl_fragColor*) **on v/s doesn't make sense** because interpolation inaccuracy

Vertex shader: Prepare *varying* variables, set *gl_Position* to the NDC of the vertex being processed.
Fragment shader: Inputs *varying* variables and compute the color of each fragment (*gl_fragColor*)

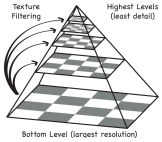
WEBGL SHADERS

create shader: gl.createShader(?), **shader source:** gl.shaderSource(?), **compile:** gl.compileShader(?)
linking: attach shaders and checks for varying pairs enable vertex attribute array before providing model positions and colors from flat arrays
buffer: how data is transferred from CPU to GPU
ARRAY_BUFFER and ELEMENT_ARRAY_BUFFER
create buffer, bind to a type, and transfer data from array associate buffer with pre-loaded data with attribute
triangle mesh: separate buffer from vertex attributes
Z-buffer algorithm: gl.enable(gl.DEPTH_TEST)
record color and z-value (depth) of every drawn pixel
pixel is drawn if depth of new pixel < depth old pixel



TEXTURE MAPPING

Tex coords (UVs): in the range $[0,1]^2$. Determines pixel of the texture to sample from.
Samplers: supports lookup operations with u/v coordinates that retrieve color from texture image. Reside in GPU texture memory and in shaders.
Texture image: determine color of pixels, sent to fragment shader. not vertex because interpolation will have been done AFTER mapping (inaccurate)
Aliasing: use interpolation or **MipMaps** to prevent jagging aliasing artifacts. Each MipMap is the same image at a lower resolution to be mapped to a smaller surface (better to be blurry than jagged)

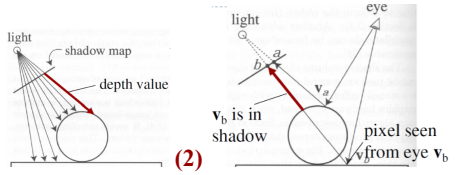


Interpolation: nearest (sharper) vs linear (smoother)
Wrapping: deal with UVs outside of [0,1] range

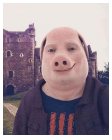
NON COLOR TEXTURES

Decal texturing: decals are textures used as selectors to determine where each texture goes.
Bump mapping: encodes "height offset" of a non-flat surface to of an underlying flat model. normal are approx by comparing values of neighbor texture pixels
Normal mapping: stores the 2 components of normals in the R and B channels. Alters the object normals.
Parallax mapping: simulate perspective of different view angle via texture offsets, degree of embossing
Specular mapping: control specularity (exponent term) via texture. Works well with bump maps.
Skybox/cubemaps: large textured cube around the camera to emulate appearance of faraway surroundings
Environment mapping: use cubemaps to simulate reflections, giving appearance that the object is interacting with the surrounding environment
Projective texture: textured point light source casts image onto object (compute NDCs of projected lights)

SHADOW MAPPING



(1) Place camera at light location, render and store depth image (**shadow map**)
(2) Render from a normal camera viewpoint.
at every fragment, compare distance to light with shadow map value
shadow map is larger → shadowed fragment
distant to light is larger/equal → lit fragment
Issues: if FoV goes past shadow map, solution is to use a cubemap.
sampling/resolution of shadow map, thin objects, multiple light sources, soft shadows

すいちゃんは今日も可愛い

Render-to-texture: render to a texture image, then sample it in subsequent passes as a texture for reflections, environment cubemaps, shadow mapping
Multi-pass rendering: first pass: render scene into 6 faces of environment cube map, second pass: normal environment mapping with rendering texture

LIGHT RENDERING

Rendering: to map **3D scenes onto 2D screens**.
Requires 3D geometry, camera & lights specifications, texture & material types. Usually concerns photorealism

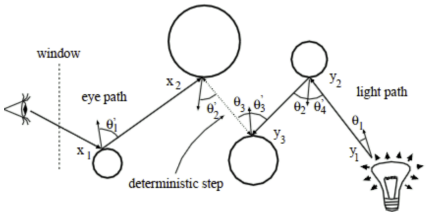
Model-centric rendering: centered around primitives that make up objects. Object's appearance is independent of other object's existence.
(+) Fast. Great standardization. Can fake effects (i.e. shsdfows mapping, reflective surfaces)
(–) Cannot natively do advanced lighting. Several effects are difficult to fake. Not photorealistic

Light-centric rendering: simulates rays of light scattering in a scene. There is a way to render anything but too expensive and don't fully understand materials
Requires light sources, dispersion scheme, attenuation by media, amount reaching the camera.
(+) Backed up by physics, can be photorealistic, many effects are possible, potential of parallelism
(–) Slow and expensive (parallel processing of rays)
Enables complex reflections and soft shadows, transparency and refractions, color bleeding, subsurface scattering, caustics, dispersion, depth of field, defocus

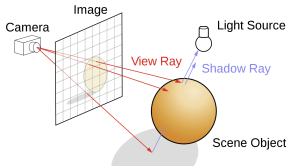
Direct Illumination traces light only to a limited extent. Uses feed forward algorithm. Minimal dependence between objects. Ideal for parallel processing
Global Illumination computes a *steady state* of light in a scene. Uses an iterative algorithm. Objects' appearance are interdependent. Less ideal for parallel.

RAY TRACING

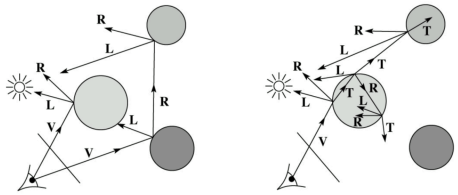
Forward ray tracing traces light from source to camera. More physically accurate but wastes rays L(D|S)*E in reality, L(D|S)E in OpenGL



Backward ray tracing traces light from camera, hopefully reaching a light source.



Recursive ray tracing each ray allowed max of N bounces. First N-1 are specular, last can be specular or diffuse. Spawns one of the three each bounce.



Reflection ray reflects off a surface, incrementing age by 1 (maxes at age=N as per usual). specular
Transmission ray spawns at each hit to a transparent object (uses snell's law to compute refraction)
Shadow ray guarantees at the last bounce (but attempts every bounce). Points to light and kill rays intersecting any object. Otherwise use a local lighting model.
(+) Less rays wasted. More photorealistic.
(–) Assumes first N-1 reflections are PERFECTLY reflective.Difficult to make soft shadows. Ignore a lot of potential diffuse lighting.

PRACTICE EXAM