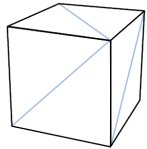
02561 Computer Graphics

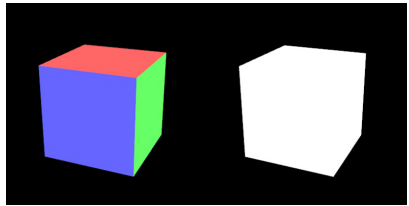
Lighting and shading

Jeppe Revall Frisvad

September 2024

Drawing solid objects

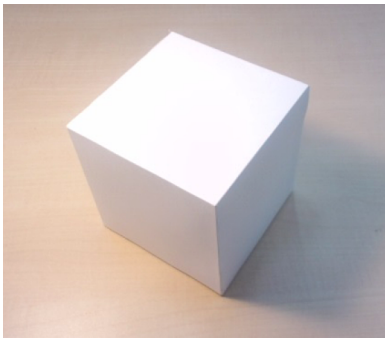




- Wireframe to coloured: switch draw mode from `gl.LINES` to `gl.TRIANGLES`.

- A white cube doesn't look right. It's all flat.
- We would expect something like a real cube →
- We need to shade our objects.
  Flat shading is inadequate.

- Even so, let's do the flat shading to get started.



Replacing an object

- Suppose we can draw a sphere at different levels of subdivision.
- How to replace an object with a new one at a different level?
- Recycle the vertex buffer:

```
window.onload = function init() {
  .
  .
  .
  gl.program = initShaders(gl, "vertex-shader", "fragment-shader");
  gl.useProgram(gl.program);
  gl.vBuffer = gl.createBuffer();
  var numVerts = initSphere(gl, numSubdivs);
  .
  .
  .
}
```

```
function initSphere(gl, numSubdivs) {
  .
  .
  .
  tetrahedron(pointsArray, va, vb, vc, vd, numSubdivs);
  gl.bindBuffer(gl.ARRAY_BUFFER, gl.vBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, flatten(pointsArray), gl.STATIC_DRAW);
  .
  .
  .
}
```

Exercise (WP04P1)

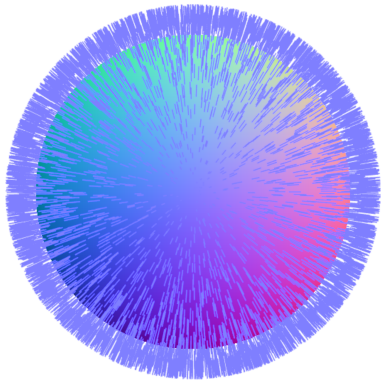
- Draw a solid sphere in perspective view. (**A**: 6.6)
- Insert buttons for subdividing or coarsening the mesh (incrementing or decrementing the level of subdivision). (**A**: 3.6.2)
- Recycle the vertex buffer (see previous slide).
- For a unit sphere at the origin, vertex positions $\boldsymbol{p}$ are also vertex normals $\vec{n}$:

$$\vec{n}_{xyz} = \boldsymbol{p}_{xyz} \,.$$

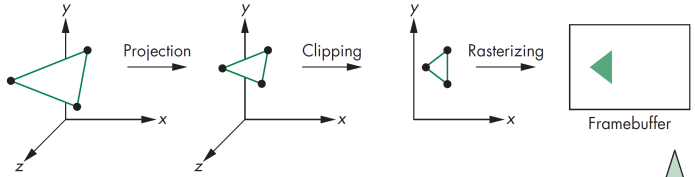
- Get the normal from the position and visualize it using the vertex color $\boldsymbol{c}$:

$$\boldsymbol{c}_{rgb} = 0.5\,\vec{n}_{xyz} + 0.5 = 0.5\,\boldsymbol{p}_{xyz} + 0.5 \,.$$

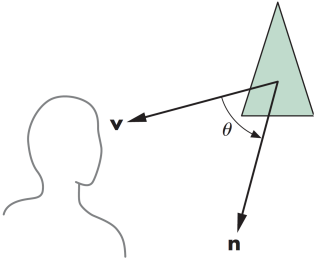
$$\boldsymbol{c} = 0.5\,\boldsymbol{p} + 0.5 \,.$$



Hidden surface removal



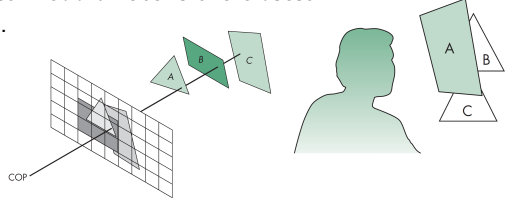
- We usually draw closed 3D objects.
  - Backfacing triangles (usually half the triangles) then do not appear on screen.
  - Let us not draw them: ensure $\boldsymbol{v} \cdot \boldsymbol{n} > 0$.
  - Backface culling: `gl.enable(gl.CULL_FACE);`



- Sometimes triangles are hidden behind other triangles.
  - In NDC, the fragment with the smallest z-coordinate is the closest.
  - Let us discard the hindmost fragments.

    `gl.enable(gl.DEPTH_TEST);`
  - When rendering

    ```
    gl.clear(gl.COLOR_BUFFER_BIT
             | gl.DEPTH_BUFFER_BIT);
    ```



Enabling the depth buffer in WebGPU

- Set up the pipeline ↓ to include depth and create a canvas-filling depth texture ↙

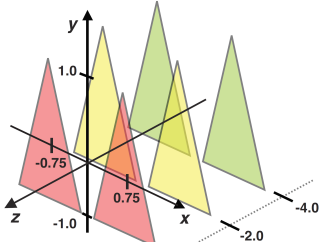
```
const pipeline = device.createRenderPipeline({
  layout: "auto",
  vertex: {
    module: wgsl,
    entryPoint: "main_vs",
    buffers: [vPositionBufferLayout]
  },
  fragment: {
    module: wgsl,
    entryPoint: "main_fs",
    targets: [{ format: canvasFormat }]
  },
  primitive: {
    topology: "triangle-list",
  },
  multisample: {
    count: msaaCount,
  },
  depthStencil: {
    depthWriteEnabled: true,
    depthCompare: 'less',
    format: 'depth24plus'
  },
});
```

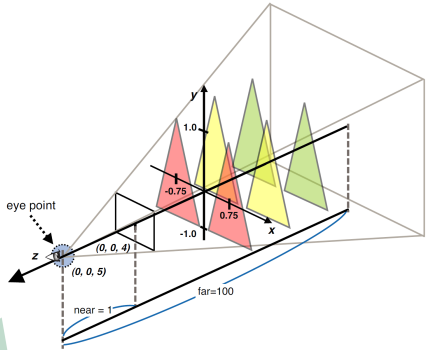
```
const msaaTexture = device.createTexture({
  size: { width: canvas.width, height: canvas.height },
  format: canvasFormat,
  sampleCount: msaaCount,
  usage: GPUTextureUsage.RENDER_ATTACHMENT,
});

const depthTexture = device.createTexture({
  size: { width: canvas.width, height: canvas.height },
  format: 'depth24plus',
  sampleCount: msaaCount,
  usage: GPUTextureUsage.RENDER_ATTACHMENT,
});


const encoder = device.createCommandEncoder();
const pass = encoder.beginRenderPass({
  colorAttachments: [{
     view: msaaTexture.createView(),
     resolveTarget: context.getCurrentTexture().createView(),
     loadOp: "clear",
     clearValue: { r: 0.3921, g: 0.5843, b: 0.9294, a: 1.0 },
     storeOp: "store",
  }],
  depthStencilAttachment: {
    view: depthTexture.createView(),
    depthLoadOp: "clear",
    depthClearValue: 1.0,
    depthStoreOp: "store",
  }
});
```

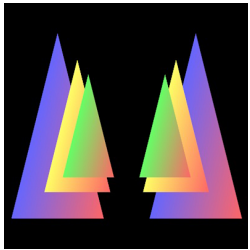
- And update the render pass accordingly. ↑

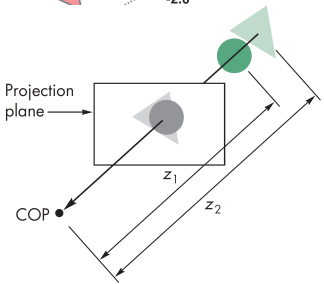
Depth buffering (z-buffering)

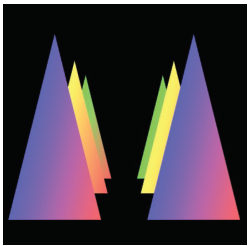
- Example:











↑

without depth test

←with depth test

Exercise (WP04P2)

- When visualizing the surface normals as colors, the result may seem strange.
- Now, enable the depth test.
- Then enable backface culling.
- There is no visible effect of the back face culling.
- Try changing the convention to having clockwise wound triangles being front facing:

  ```
  gl.frontFace(gl.CW);
  ```

- You should now see the back side of the sphere.
- The default winding is counterclockwise (`gl.CCW`).
- In WebGPU, control face culling in the pipeline setup:

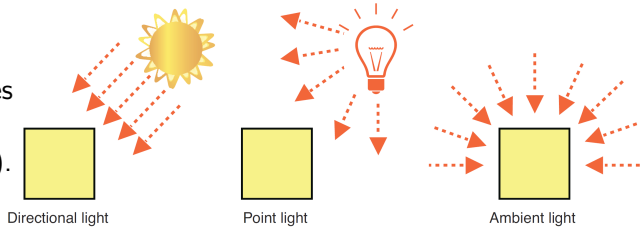
```
primitive: {
  topology: "triangle-list",
  frontFace: "ccw", // options { "ccw", "cw" }
  cullMode: "back", // options { "none", "front", "back" }
},
```

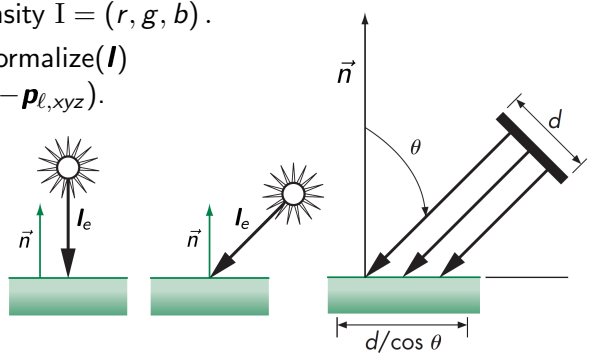
Light sources

- Use homogeneous coordinates to set a point light ($w = 1$) or a directional light ($w = 0$).
- Light source specification
  Position: $\boldsymbol{p}_\ell = (x, y, z, w)$.
  Emission: $L_e = (r, g, b)$. Or intensity $\mathrm{I} = (r, g, b)$.
- Direction toward the light $\vec{\omega}_i = \text{normalize}(\boldsymbol{l})$
  (for a directional light $\boldsymbol{l} = -\boldsymbol{l}_e = -\boldsymbol{p}_{\ell,xyz}$).
- For normalized $\vec{\omega}_i$ and surface normal $\vec{n}$:

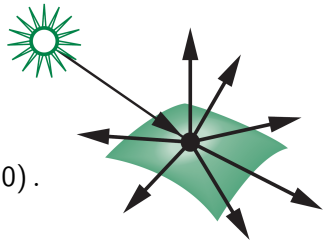
$$\cos\theta = \vec{n} \cdot \vec{\omega}_i \,.$$

- An ambient source has no $\boldsymbol{l}_e$ only a background colour L_a.





Lighting of Lambertian objects



- The light reflected from a perfectly diffuse object is

$$L_{r,d} = k_d L_i \max(\cos\theta, 0) = k_d L_i \max(\vec{n} \cdot \vec{\omega}_i, 0)\,.$$

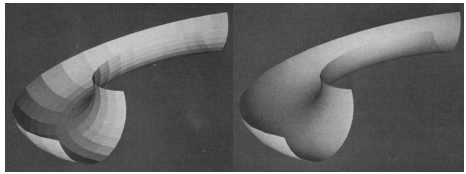
- k_d is a diffuse reflection coefficient (material colour).
- L_i is incident light: $L_i = V L_e$, where V is visibility (assume $V = 1$).
- A point light has no area and thus has intensity I instead of emission L_e.
- For a point of incidence $\boldsymbol{x}$, a point light has $\vec{\omega}_i = \text{normalize}(\boldsymbol{p}_{\ell,xyz} - \boldsymbol{x}_{xyz})$ and

$$L_i = V \frac{\mathrm{I}}{\|\boldsymbol{p}_\ell - \boldsymbol{x}\|^2}\,.$$

- To add ambient light L_a, we use an ambient reflection coefficient k_a $(= k_d)$ and have

$$L_o = L_{r,d} + L_{r,a} = k_d L_i \max(\cos\theta, 0) + k_a L_a\,.$$

Gouraud shading



- Gouraud came up with the idea of linear interpolation between triangle vertices.
- This is hardwired in the varying variables of modern GPUs.
- Let us then do shading calculations in the vertex shader.
- We first need the direction toward the light $\vec{\omega}_i$:

```
vec3 w_i = lightPos.w == 0.0 ? normalize(-lightPos.xyz) : normalize(lightPos.xyz - pos.xyz);
```

  - `lightPos` is a `uniform vec4` uploaded from JavaScript.
  - pos is obtained from the vertex position attribute: `vec4 pos = M*aPosition;`
- Given other parameters k_d, k_a, L_e, L_a as uniforms and the surface normal $\vec{n}$ as an attribute, we can calculate L_o and store it in a varying color variable.

- Henri Gouraud. Continuous shading of curved surfaces. *IEEE Transactions on Computers C-20*(6), pp. 623–629. June 1971.

Exercise (WP04P3)

- Get the surface normal (use an attribute [**A**: 6.9, but see Wiki] or $\vec{n} = \boldsymbol{p}_{xyz}$).
- Upload parameters (k_d, L_e, and $\boldsymbol{p}_\ell$ or $\boldsymbol{l}_e$) as uniform vectors.
- Add a tick function for animation and a button for toggling orbiting of the camera around the sphere on/off.

- Orbiting is spherical motion of the camera around a centre point:
  - Make radius r and angle α variables (radius is constant) and use sine and cosine to move the camera in a circle in the xz-plane.
  - Move the camera by recalculating the view matrix with a new eye position:
    $\boldsymbol{e} = (r \sin\alpha, 0, r \cos\alpha)$.
  - Use the `lookAt` function for recalculating the view matrix.
  - Upload the view matrix $\boldsymbol{V}$ as a `uniform mat4` for each update so that

    ```
    gl_Position = P*V*pos;
    ```

Phong reflection model

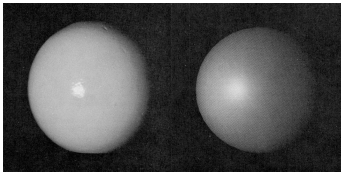
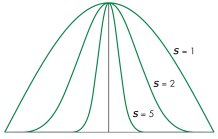
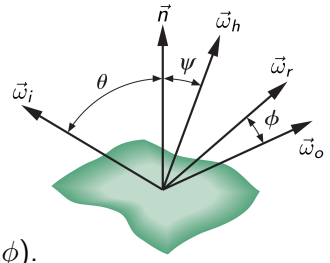


photo Phong

- Most real materials exhibit highlights.
- How to include a highlight in the reflection model?
- The Phong reflection model:

$$L_o = L_{r,d} + L^P_{r,s} + L_{r,a}\,.$$
$$L^P_{r,s} = k_s L_i(\max(\vec{\omega}_r \cdot \vec{\omega}_o, 0))^s$$
$$\vec{\omega}_r = 2(\vec{\omega}_i \cdot \vec{n})\vec{n} - \vec{\omega}_i\,.$$





- The Phong exponent s is called the shininess.
- $\vec{\omega}_r$ is the direction of perfect reflection.
- $\vec{\omega}_o$ is the direction toward the observer ($\vec{\omega}_r \cdot \vec{\omega}_o = \cos\phi$).
- To have $\vec{\omega}_o$ in the shader, we then either need the eye point in world coordinates or all vectors in eye space (where the eye point is at the origin).

- Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM 18*(6), pp. 311–317. June 1975.

Blinn-Phong reflection model

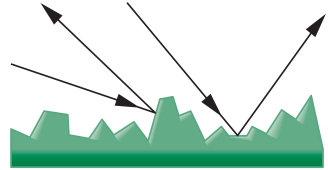
- If we think of a diffuse surface as being very rough with scattering due to many little microfacets, a more correct version of the Phong highlight is [Blinn 1977]

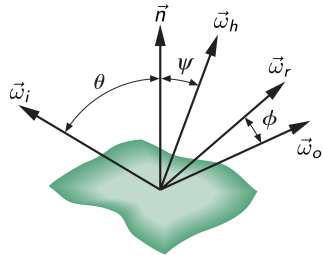
$$L_{r,s}^{BP} = k_s L_i (\max(\vec{n} \cdot \vec{\omega}_h, 0))^s .$$

$$\vec{\omega}_h = \frac{\vec{\omega}_i + \vec{\omega}_o}{\|\vec{\omega}_i + \vec{\omega}_o\|} = \text{normalize}(\vec{\omega}_i + \vec{\omega}_o) .$$

- $\vec{\omega}_h$ is called the halfway (or half) vector.
- ψ is called the halfway angle ($\vec{\omega}_h \cdot \vec{n} = \cos\psi$).
- With this specular term, we have the Blinn-Phong reflection model:

$$L_o = L_{r,d} + L_{r,s}^{BP} + L_{r,a} .$$





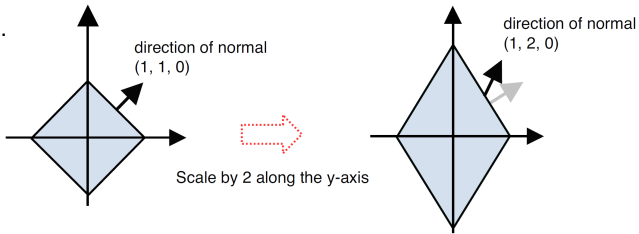
- James F. Blinn. Models of light reflection for computer synthesized pictures. *Computer Graphics (SIGGRAPH '77) 11*(2), pp. 192–198. Summer 1977.

Lighting of glossy objects

- For glossy objects, we need the direction toward the observer $\vec{\omega}_o$.
- One way is uploading the eye position in world coordinates as a uniform vector.
- The standard approach is moving all calculations to eye space.
- Transform to eye space using the view matrix $\boldsymbol{V}$. Then

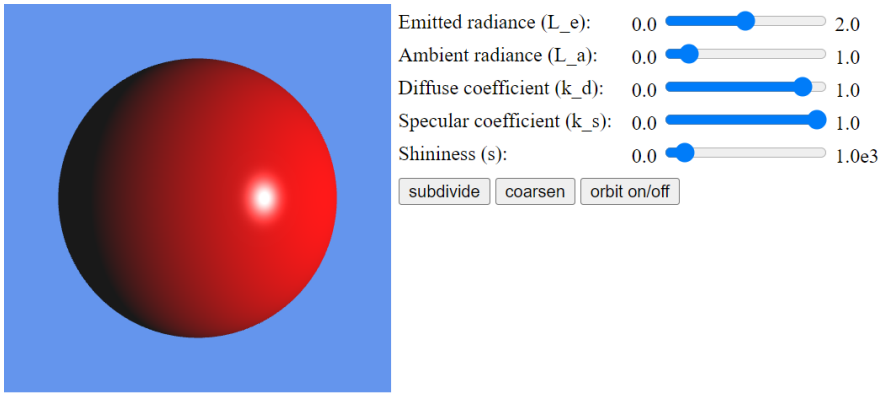
```
vec3 w_o = -normalize((V*pos).xyz);
```

- Note that both $\vec{\omega}_i$, $\vec{\omega}_o$, $\vec{n}$, and derived quantities must be in eye space.
- In cases of non-uniform scaling, a special transformation $\boldsymbol{N}$ is required for $\vec{n}$.
- We need $\vec{v} \cdot \vec{n} = (\boldsymbol{M}\vec{v}) \cdot (\boldsymbol{N}\vec{n})$.
- Then $\vec{v} \cdot \vec{n} = (\boldsymbol{M}\vec{v})^T(\boldsymbol{N}\vec{n})$ and $\vec{v} \cdot \vec{n} = \vec{v}^T(\boldsymbol{M}^T\boldsymbol{N})\vec{n}$.
- The normal matrix is then $\boldsymbol{N} = (\boldsymbol{M}^T)^{-1}$



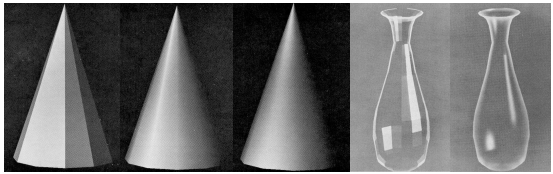
Exercise (WP04P4)

- Upload remaining parameters for the Phong model (k_s, s, L_a) as uniform vectors.
- Decide whether you will do calculations in eye space or world space.
- Update your reflection model calculations to include ambient light and highlights.
- Add sliders to the webpage for modifying the different parameters.
- Use the input event for the sliders to get immediate response while sliding.



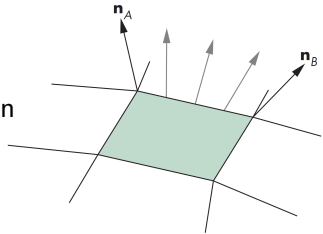
Phong shading

- Gouraud shading requires a very large number of triangles to look good, this is true for highlights in particular.



flat Gouraud Phong flat Phong

- Phong came up with the idea of re-normalizing the vectors for every pixel.
- This is needed because linear interpolation shortens a direction vector.
- We implement it by moving our evaluation of the Phong reflection model to the fragment shader.
- Most importantly, we must normalize the direction vectors in the fragment shader.
- Uniform variables can be declared and accessed in both vertex and fragment shader.



- Bui Tuong Phong. Illumination for computer generated pictures. *Communications of the ACM 18*(6), pp. 311–317, June 1975.