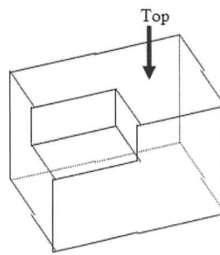
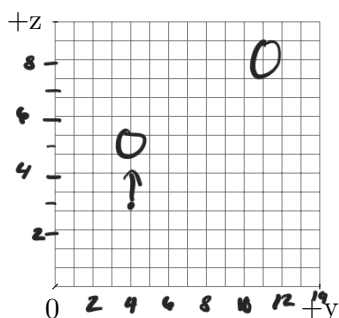
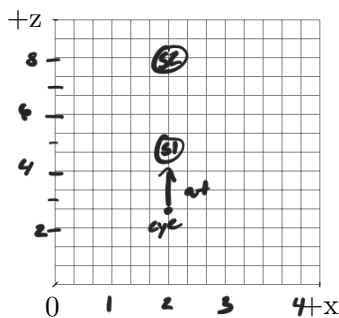


Homework 4 - Viewing & Projection

1. What matrix is created by $\text{LookAt}(\text{eye}=[0,0,2], \text{at}=[0,0,0], \text{up}=[0,1,0])$? Is there a way to get this matrix using translate and/or rotate and/or scale? Explain.
2. Illustrate the difference between orthographic (parallel) and perspective projection.
3. Consider the 3D solid cube, with a cube-shaped notch removed from one corner.



- (a) Sketch a wire frame of the top view of the object, using orthogonal projection.
 - (b) Sketch a wire frame of the top view of the object, using perspective projection.
4. Suppose we set up a camera using $\text{LookAt}(\text{eye}=[2,4,3], \text{at}=[2,4,4], \text{up}=[0,1,0])$ and render a scene with two spheres of radius 1, centered at $[2,4,5]$ and $[2,10,8]$.
 - (a) Draw this scene in 2D both on the x-z plane (left) and on the y-z plane (right). Label the eye and the at points as well as the two spheres.



- (b) What is the distance to the near and far plane if they are set to bound the objects as closely as possible? Describe the distance in numerical values and explain it.
- (c) If we render this scene with a projection matrix, how many spheres will be seen in each of the following cases? Why?
- i. perspective(fovy=1°, aspect=1, zNear=1, zFar=100)
 - ii. perspective(fovy=175°, aspect=1, zNear=4, zFar=10)
 - iii. perspective(fovy=60°, aspect=1, zNear=1, zFar=10)

$$1. \begin{bmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

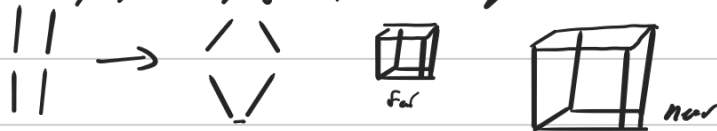
Yes, this matrix can be obtained using a translation of $(0, 0, -2)$. The Look At setup in this case does not involve any rotation or scaling, so a simple translation by $-eye$ gives the same result.

2. Orthographic projection preserves parallel lines and object sizes regardless of depth, while perspective projection makes objects appear smaller as they get farther from the camera.

In orthographic projection, lines remain parallel:



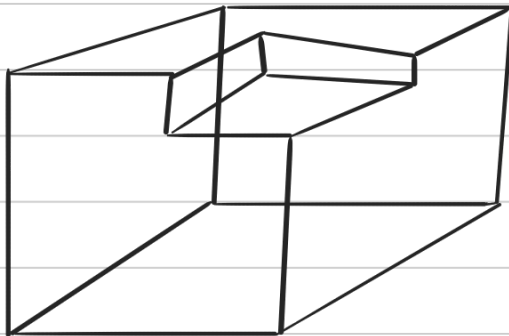
In perspective projection, lines converge toward a vanishing point:



3. a)



b)



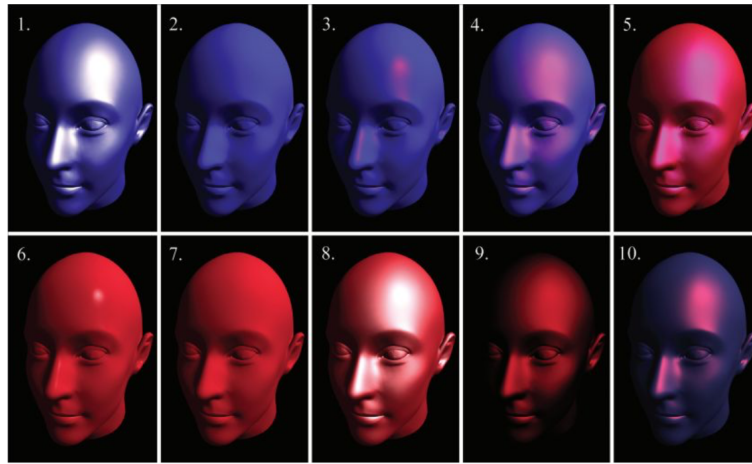
4 a) Completed above!

- b)
- Closest point (surface of sphere 1): $z = 5 - 1 = 4$
 - Furthest point (surface of sphere 2): $z = 8 + 1 = 9$
- so, near = 4, far = 9

- c)
- I. \rightarrow 2 spheres seen. Narrow field of view but includes full z range 4-9.
 - II. \rightarrow 2 spheres seen. Wide field of view and includes both spheres.
 - III. \rightarrow 2 spheres seen. Normal field of view and includes full z range.

Homework 4 - Lighting

1. Define the three components of light used in the Phong lighting model: diffuse, specular, and ambient.
2. Match the images below to the properties defined in the table. Write the image number in the correspondent table row.



	Diffuse Color	Specular Color	Specular Exponent (giving the size of the highlight)	Image Number
(a)	Red	Black	Any	<u>5</u>
(b)	Blue	Red	Small	<u>3</u>
(c)	Blue	White	Big	<u>1</u>
(d)	Darker Blue	Red	Medium-Big	<u>4</u>
(e)	Blue	Red	Big	<u>6</u>
(f)	Blue	Black	Any	<u>2</u>
(g)	Black	Red	Big Red	<u>9</u>
(h)	Red	White	Small	<u>7</u>
(i)	Red	White	Big	<u>8</u>
(j)	Red	Blue	Big	<u>10</u>

3. Consider a scene with a single sphere of radius 1 centered at the origin $(0,0,0)$. The sphere's ambient coefficient is $k_a = \text{RGB}(0.1,0.1,0.1)$, diffuse coefficient $k_d = \text{RGB}(0.5,0.5,0.5)$, and specular coefficient $k_s = \text{RGB}(0.5,0.5,0.5)$, with specular exponent $s = 2$. There is a light at position $p_1 = (0,2,0)$ with color $c_1 = \text{RGB}(1.0, 0.0, 0.0)$. The camera is located at position $p_2 = (3,1,0)$. Calculate the Phong Illumination for the sphere fragment located at position $p_3 = (0,1,0)$.

- (a) Calculate the ambient portion of the color of this fragment.
- (b) Calculate the diffuse portion of the color of this fragment.

(c) Calculate the specular portion of the color of this fragment.

(d) Calculate the total color of this fragment.

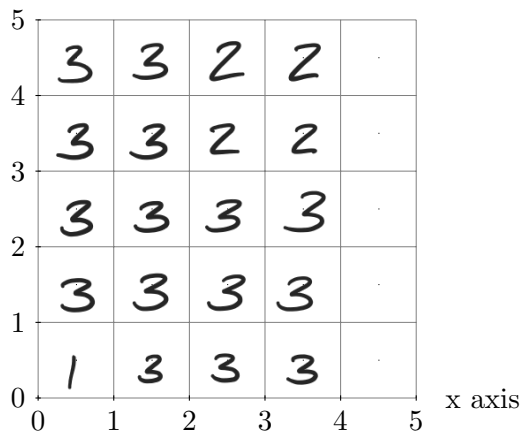
(e) Now suppose the camera moves to $p_4 = (0,3,0)$. What is the total color of the fragment?

4. Consider an orthographic scene with objects listed below being rendered at 5 x 5 pixels with the view plane at $z=0$ (near=0, far=99). Show the state of the z-buffer (with numbers) once these squares have been rendered (without using anti-aliasing or sub-sampling). Note that the pixels are drawn such that the z-buffer values are stored halfway between integers (look at the labels below the z-buffer).

Red square with vertices: (0,0,1), (1,0,1), (1,1,1), (0,1,1)

Green square with vertices: (2,2,2), (4,2,2), (4,4,2), (2,4,2)

Blue square with vertices: (0,0,3), (4,0,3), (4,4,3), (0,4,3)



5. Consider the following vertex and fragment shaders:

```
// Vertex Shader
uniform mat4 u_NormalMatrix;
uniform mat4 u_ModelMatrix;
uniform mat4 u_ViewMatrix;
uniform mat4 u_ProjectionMatrix;

attribute vec4 a_Position;
varying vec3 v_Position;
attribute vec4 a_Normal;
varying vec3 v_Normal;

void main() {
    v_Normal = normalize(vec3(u_NormalMatrix * a_Normal));
    v_Position = vec3(u_ModelMatrix * a_Position);
    gl_Position = u_ProjectionMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;
}';

// Fragment Shader
varying vec3 v_Position;
varying vec3 v_Normal;
uniform vec3 u_LightPos;
```

```
uniform vec3 u_LightColor;

void main() {
    vec3 l = normalize(u_LightPos - v_Position);
    float nDotL = max(dot(l, v_Normal), 0.0);
    gl_FragColor = vec4(u_LightColor * nDotL);
}
```

- (a) In the vertex shader, what is the normal matrix and why do we need to multiply it by the normal vector?
- (b) In the vertex shader, why do we multiply the vertex position by the model matrix?
- (c) What is the fragment shader computing?

/

Diffuse: Light that scatters equally in all directions from a surface, depending on the angle between the light and the surface normal. It gives the object its basic color under light.

Specular: Light that reflects in a specific direction, creating shiny highlights. It depends on the viewer's position and simulates glossiness.

Ambient: Light that is scattered in the environment and hits all surfaces equally, regardless of light or viewer direction. It prevents objects from being completely dark in shadows.

3. a) $\text{Ambient} = k_a \times c_1 = (0.1, 0.1, 0.1) \times (1.0, 0.0, 0.0) = (0.1, 0.0, 0.0)$

b)

- Light direction = $\text{normalize}(p_1 - p_3) = \text{normalize}((0,1,0)) = (0,1,0)$
- Surface normal at $p_3 = (0,1,0)$
- $n \cdot l = 1$
- Diffuse = $k_d \times c_1 \times \max(n \cdot l, 0) = (0.5, 0.5, 0.5) \times (1.0, 0.0, 0.0) \times 1 = (0.5, 0.0, 0.0)$

c)

- View direction = $\text{normalize}(p_2 - p_3) = \text{normalize}((3,0,0)) = (1,0,0)$
- Reflection vector = $2(n \cdot l)n - l = 2(1)(0,1,0) - (0,1,0) = (0,1,0)$
- $r \cdot v = (0,1,0) \cdot (1,0,0) = 0$
- Specular = $k_s \times c_1 \times \max(r \cdot v, 0)^2 = (0.5, 0.5, 0.5) \times (1.0, 0.0, 0.0) \times 0^2 = (0.0, 0.0, 0.0)$


d)

$$\text{Total} = \text{Ambient} + \text{Diffuse} + \text{Specular}$$

$$= (0.1, 0.0, 0.0) + (0.5, 0.0, 0.0) + (0.0, 0.0, 0.0) = (0.6, 0.0, 0.0)$$

e)

- New view direction = $\text{normalize}(p_4 - p_3) = \text{normalize}((0,2,0)) = (0,1,0)$
- $r \cdot v = (0,1,0) \cdot (0,1,0) = 1$
- Specular = $(0.5, 0.5, 0.5) \times (1.0, 0.0, 0.0) \times 1^2 = (0.5, 0.0, 0.0)$
- Total = $(0.1, 0.0, 0.0) + (0.5, 0.0, 0.0) + (0.5, 0.0, 0.0) = (1.1, 0.0, 0.0)$

- 5.
- a) The normal matrix is used to correctly transform normal vectors into eye space. We multiply the normal by the normal matrix to maintain correct lighting under non-uniform scaling or rotation of the model.
 - b) We multiply the vertex position by the model matrix to convert the position from object space to world space, so it can be used for lighting and further transformations.
 - c) The fragment shader computes the diffuse lighting at each fragment using the dot product between the normalized light direction and the surface normal. It outputs the color scaled by how directly the surface faces the light.
- 

Chapter 7: Toward the 3D World — Notes & Outline

Triangles and Cubes

- Cubes are just collections of triangles. What works for rendering triangles extends to more complex shapes like cubes.

Viewing Direction

- **Eye Point, Look-At Point, Up Vector** define the camera.
- The `lookAt()` function generates a view matrix that positions the scene relative to the camera.
- **Sample Program:** `LookAtTriangles.js` – Shows basic camera movement.

Comparing Camera Views

- Rotated triangles behave differently when using a static vs. moving viewpoint.
- **Sample Program:** `LookAtRotatedTriangles.js` – Demonstrates this comparison.

Changing View with Keyboard

- You can dynamically adjust the view matrix with keyboard inputs.
 - **Sample Program:** `LookAtTrianglesWithKeys.js` – Moves the camera left/right using arrow keys.
-

Visible Range (Orthographic Projection)

- Defined using a **box-shaped view volume**: left, right, top, bottom, near, far.
- Orthographic keeps parallel lines parallel (no perspective distortion).
- **Sample Programs:** `OrthoView.html` & `OrthoView.js`

View Clipping and Vertex Shader Role

- The vertex shader transforms geometry and clips it to the defined volume.
 - Changing near/far values clips objects too close or too far.
 - **Sample Program:** `LookAtTrianglesWithKeys_ViewVolume.js`
-

Perspective Projection (Frustum)

- Defined using a **pyramid-shaped volume**.
- Simulates real-world depth — objects further away appear smaller.

- **Sample Program:** PerspectiveView.js

Projection Matrix Role

- Transforms 3D coordinates into the canonical view volume.
 - Combined with **model** and **view** matrices for full transformation.
 - **Sample Program:** PerspectiveView_mvp.js
-

Depth and Z-Buffering

- **Hidden surface removal** ensures closer objects block farther ones.
 - **Z-fighting** occurs when two surfaces are at similar depth.
 - **Sample Program:** DepthBuffer.js – Shows depth handling.
-

Drawing Cubes

- Use **vertices and indices** for efficiency.
 - Colors can be assigned per face.
 - **Sample Program:** HelloCube.js, ColoredCube.js
-

Chapter 8: Lighting Objects — Notes & Outline

Lighting Basics

- WebGL lighting simulates light bouncing off surfaces.
- Light adds realism by highlighting depth and surface orientation.

Types of Light

- **Directional Light:** Like sunlight – parallel rays.
- **Point Light:** Emits light in all directions from a point.

Types of Reflected Light

- **Ambient:** Soft light present everywhere.
 - **Diffuse:** Based on angle between surface and light.
 - **Specular (not yet in this chapter):** Reflects in a specific direction (like a mirror).
-

Directional Light and Diffuse Lighting

- $\text{Diffuse} = \text{lightColor} * \text{kd} * \max(\text{dot}(\text{normal}, \text{lightDir}), 0)$
- Surface appears brightest when facing the light directly.
- **Sample Program:** LightedCube.js

Ambient Lighting

- Adds a constant light to all surfaces.
 - Helps prevent completely black shadows.
 - **Sample Program:** LightedCube_ambient.js
-

Normal Matrix and Lighting Transforms

- When objects rotate or scale, normals must also be transformed correctly.
 - Use the **inverse transpose of the model matrix**.
 - **Sample Program:** LightedTranslatedRotatedCube.js
-

Point Light Sources

- Light comes from a point in space, intensity decreases with distance.
- Per-fragment lighting improves realism over per-vertex.
- **Sample Programs:** PointLightedCube.js, PointLightedCube_perFragment.js

Summary

- Combining ambient + diffuse + point lighting = more realistic 3D scenes.
- Proper use of projection/view/model matrices and lighting makes 3D graphics immersive.