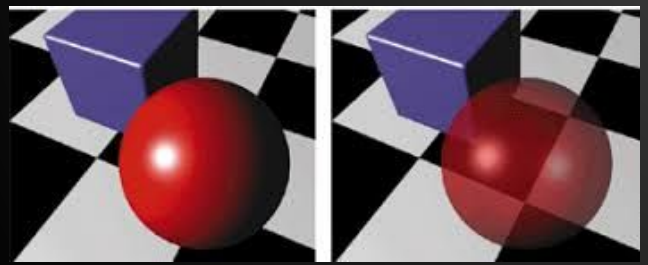
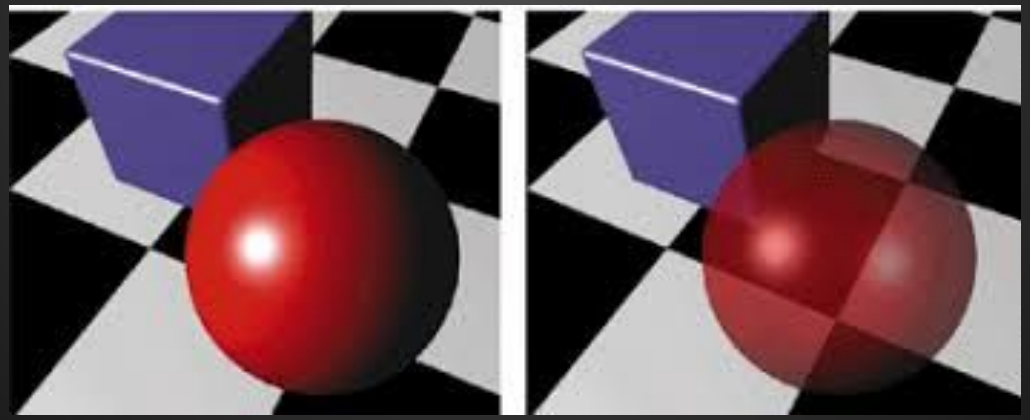
Alpha Blending and Transparency

CS 4204
Shuban Sridhar



Goal

Objective was to produce transparency using alpha blending. Essentially, each object would have an additional property to represent its level of transparency, and additional methods for shading, depth ordering, and blending would have to be implemented to achieve the desired output.



Implementation

Step 1: Alpha Channel Support

Originally, we only stored RGB colors (red, green, blue). We extended this to RGBA by adding an alpha channel, where 1.0 represents fully opaque and 0.0 represents fully transparent. This required changes to the Mesh class to store alpha values and pass them through the rendering pipeline.

```
class Mesh:
    def __init__(self, diffuse_color, specular_color, ka, kd, ks, ke, alpha=1.0):
        # Empty lists for verts, faces, and normals
        self.verts = []
        self.faces = []
        self.normals = []
        self.transform = Transform()
        # Only set these once with alpha included
        self.diffuse_color = np.array([*diffuse_color, alpha])  # Make it RGBA
        self.specular_color = np.array([*specular_color, alpha])
        self.ka = ka
        self.kd = kd
        self.ks = ks
        self.ke = ke
        self.transform = Transform()
```

Step 2: Depth Sort

For transparency to work correctly, we need to render objects in the correct order - from back to front (painter's algorithm). We implemented this by:

- Converting mesh vertices to camera space
- Calculating the average depth of each mesh
- Sorting meshes based on their depth
- Rendering them in order from farthest to nearest

This ensures that transparent objects properly show what's behind them.

```
def sort_meshes_by_depth(self, meshes):
    mesh_depths = []
    for mesh in meshes:
        vertices_world = [mesh.transform.apply_to_point(np.array(v)) for v in mesh.verts]
        vertices_camera = [self.camera.transform.apply_inverse_to_point(v) for v in vertices_world]
        avg_depth = np.mean([v[2] for v in vertices_camera])
        mesh_depths.append((mesh, avg_depth))
    return [m for m, d in sorted(mesh_depths, key=lambda x: x[1], reverse=False)]
```

Implementation Part 2

Step 3: Color Blending

We implemented alpha compositing, which is the mathematical process of combining transparent colors. The formula takes into account both the color and alpha values of the new color and the existing background color. This creates the illusion of transparency by properly mixing colors based on their alpha values

```
def blend_colors(self, new_color, existing_color):
    """
    Blend colors using alpha compositing
    Both colors should be in range [0,1]
    """
    alpha_new = new_color[3]
    alpha_existing = existing_color[3]

    # Pre-multiply alpha
    alpha_out = alpha_new + alpha_existing * (1 - alpha_new)
    if alpha_out < 1e-8:  # Avoid division by zero
        return np.zeros(4)

    color_out = (
        new_color[:3] * alpha_new +
        existing_color[:3] * alpha_existing * (1 - alpha_new)
    ) / alpha_out
    ⌘L to chat, ⌘K to generate
    return np.append(color_out, alpha_out)
```

Step 4: Buffer Management

The rendering pipeline needed significant modifications to handle transparency:

- The image buffer was extended to include an alpha channel
- We maintained the buffer in floating-point format (0-1) during calculations for precision
- Only at the final display step did we convert back to 8-bit RGB
- The z-buffer remained crucial for handling depth testing within individual meshes

```
def render(self, shading, bg_color, ambient_light):
    # Initialize with checkered background
    image_buffer = self.create_checkered_background()
    image_buffer = image_buffer.astype(np.float32) / 255.0
    image_buffer = np.dstack((image_buffer, np.ones((self.screen.height, self.screen.width))))
```

Testing

- Set up viewport and camera
- Suzanne is placed furthest back (z=-1) and set as fully opaque (alpha=1.0)
- The cube is positioned closest to the camera (z=1.5) and made semi-transparent (alpha=0.5) to demonstrate the transparency effect
- The sphere is placed at a middle depth (z=0) with partial transparency (alpha=0.7)

Confirmation of depth-ordering:

```
mesh_1.transform.set_position(1,0,-1)      # Suzanne (furthest back)
mesh_3.transform.set_position(-0.4,0.75,0)  # Sphere (middle)
mesh_2.transform.set_position(-0.25, -0.4,1.5)  # Cube (front)
```

```
if __name__ == '__main__':
    screen = Screen(500,500)

    camera = PerspectiveCamera(-1.0, 1.0, -1.0, 1.0, -1.0, -20)
    camera.transform.set_position(0, 0, 4)


    mesh_1 = Mesh.from_stl("suzanne.stl", np.array([1.0, 0.0, 1.0]),\
        np.array([1.0, 1.0, 1.0]),0.05,1.0,0.2,100, alpha=1.0)
    mesh_1.transform.set_rotation(15, 35, 0)
    mesh_1.transform.set_position(1,0,-1)

    mesh_2 = Mesh.from_stl("unit_cube.stl", np.array([0.6, 0.0, 1.0]),\
        np.array([1.0, 1.0, 1.0]),0.05,1.0,0.2,100, alpha=0.5)
    mesh_2.transform.set_position(-0.25, -0.4,1.5)
    mesh_2.transform.set_rotation(0, 0, 10)

    mesh_3 = Mesh.from_stl("unit_sphere.stl", np.array([1.0, 0.6, 0.0]),\
        np.array([1.0, 1.0, 1.0]),0.05,0.8,0.2,100, alpha=0.7)
    mesh_3.transform.set_position(-0.4,0.75,0)

    light = PointLight(50.0, np.array([1, 1, 1]))
    light.transform.set_position(4, -3, 4)

    renderer = Renderer(screen, camera, [mesh_1,mesh_2,mesh_3], light)
    renderer.render("flat",[80,80,80], [0.2, 0.2, 0.2])

    screen.show()
```

Result

Here we see different levels of transparency and depths, along with how those attributes interact with the background.

Expected areas of performance impact:

1. Mesh Sorting: O(n log n) complexity where n is number of meshes
2. Alpha Blending: Additional calculations per pixel for transparent objects
3. Memory Usage: Extra channel for alpha values
4. Z-Buffer Processing: More complex due to transparency handling