COSC 414/519I: Computer Graphics

2023W2

Shan Du

Drawing and Manipulating Objects Composed of Other Objects

- One of the key issues when drawing an object consisting of multiple objects (segments) is that you have to program to avoid conflicts when the segments move.

Drawing and Manipulating Objects Composed of Other Objects

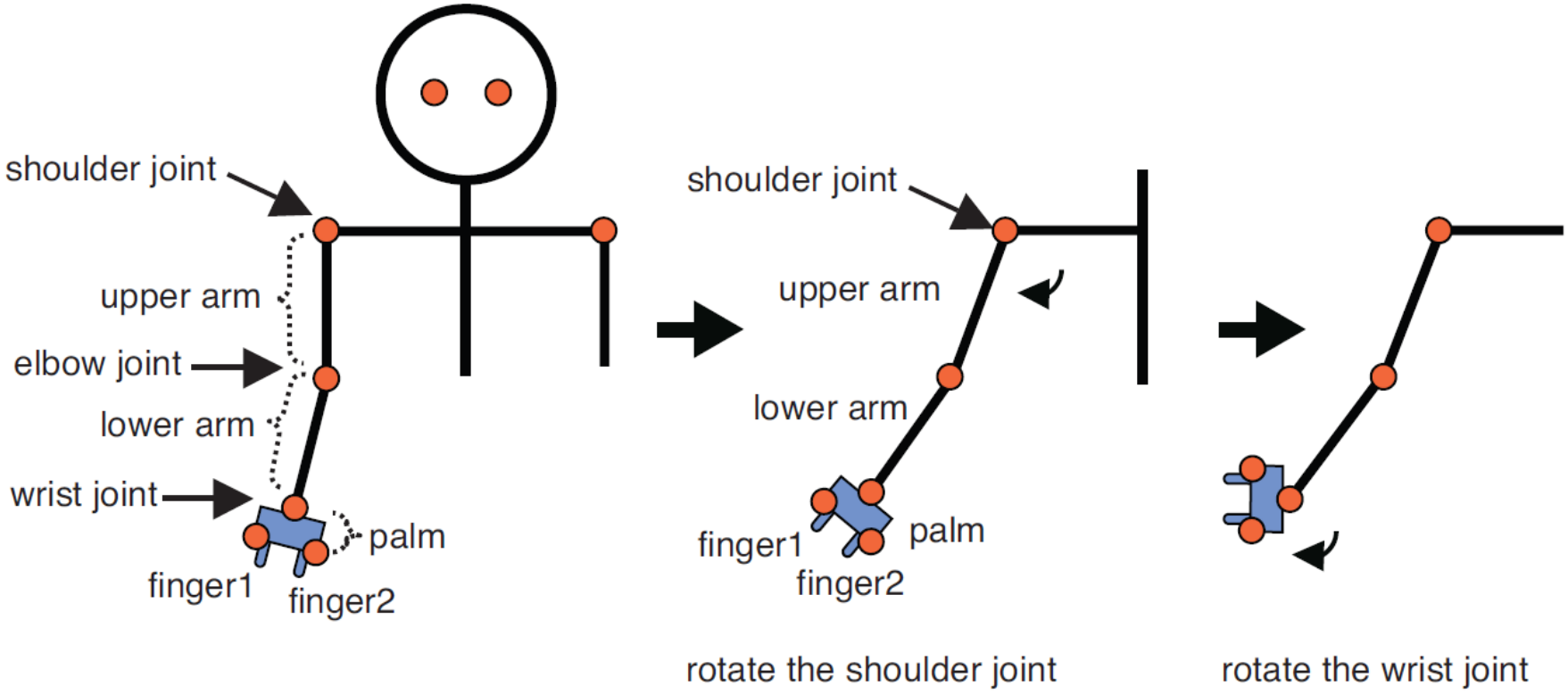


Figure 9.2 The structure and movement from the arm to the fingers

Drawing and Manipulating Objects Composed of Other Objects

- Each segment moves around a joint as follows:
  - When you move the upper arm by rotating around the shoulder joint, depending on the upper arm movement, the lower arm, palm, and fingers accordingly.
  - When you move the lower arm using an elbow joint, the palm and fingers move but the upper arm does not.
  - When you move the palm using the wrist joint, both palm and fingers move but the upper and lower arm do not.
  - When you move fingers, the upper arm, lower arm, and palm do not move.

Drawing and Manipulating Objects Composed of Other Objects

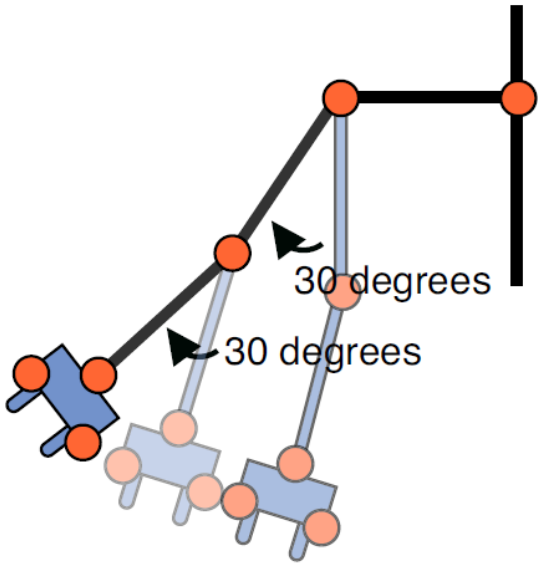
- To summarize, when you move a segment, the segments located below it move, while the segments located above are not affected. In addition, all movement, including twisting, is actually rotation around a joint.

Hierarchical Structure

- The typical method used to draw and manipulate the object with such features is to draw each part object (such as a box) in the order of the object's hierarchical structure from upper to lower, applying each model matrix (rotation matrix) at every joint.
- It is important to note that, unlike humans or robots, segments in 3D graphics are not physically joined. So if you inadvertently rotate the object corresponding to an upper arm at the shoulder joint, the lower parts would be left behind. When you rotate the shoulder joint, you should explicitly make the lower parts follow the movement. To do this, you need to rotate the lower elbow and wrist joints through the same angle that you rotate the shoulder joint.

Hierarchical Structure

- It is straightforward to program so that the rotation of one segment propagates to the lower segments and simply requires that you use the same model matrix for the rotation of the lower segments.
- For example, when you rotate a shoulder joint through 30 degrees using one model matrix, you can draw the lower elbow and wrist joints rotated through 30 degrees using the same model matrix. Thus, by changing only the angle of the shoulder rotation, the lower segments are automatically rotated to follow the movement of the shoulder joint.



rotate the shoulder joint

Single Joint Model

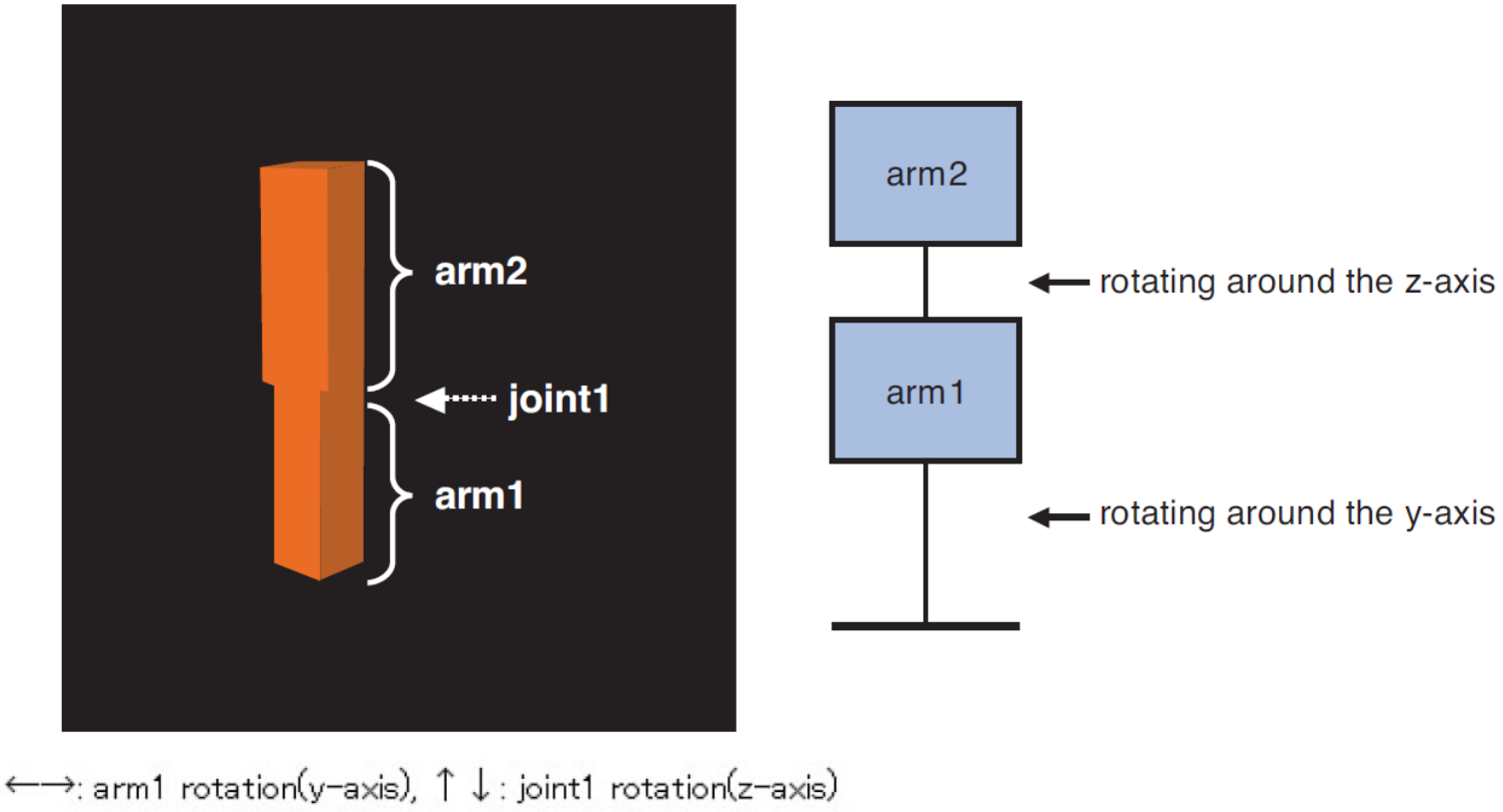
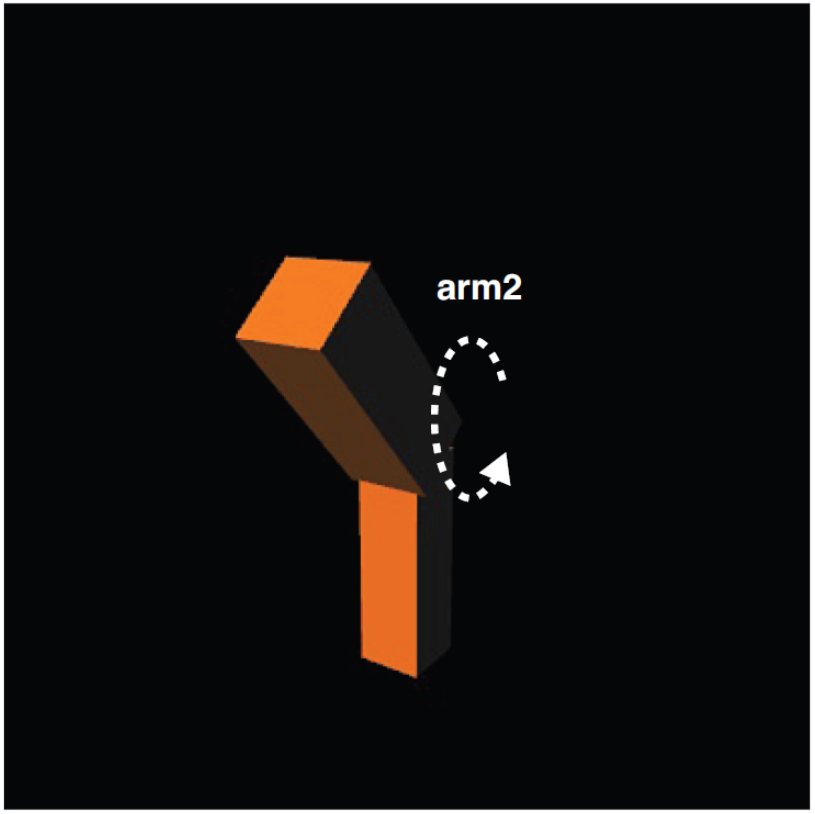


Figure 9.4 JointModel and the hierarchy structure used in the program

Single Joint Model



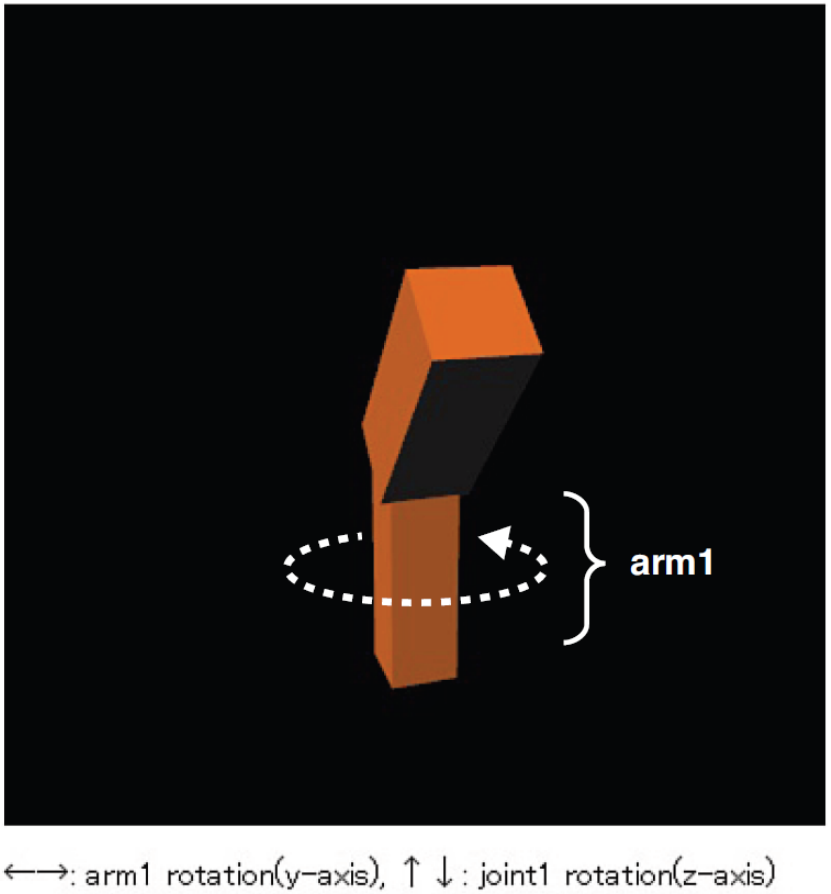


Figure 9.5 The display change when pressing the arrow keys in JointModel

Sample Program (JointModel.js)

- To better model the arm, we will use a cuboid.
- The cuboid has its origin at the center of the bottom surface and is 3.0 by 3.0 and 10.0 units in height.
- By setting the origin at the center of the bottom surface, its rotation around the z-axis is the same as that of joint1 in Figure 9.5, making it convenient to program.

Sample Program (JointModel.js)

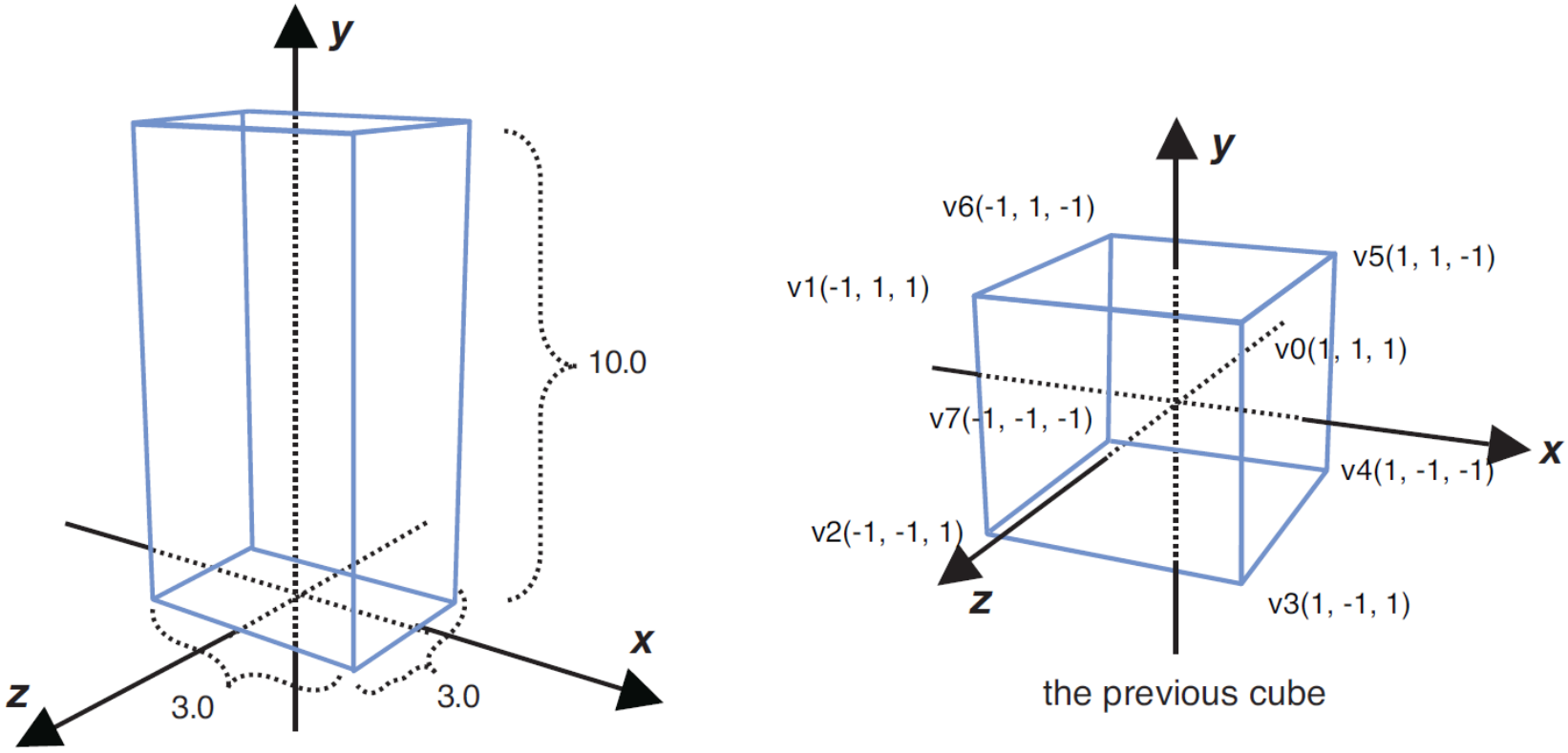


Figure 9.6 A cuboid for drawing the robot arm

Sample Program (JointModel.js)

```
function initVertexBuffers(gl) {
  // Vertex coordinates(a cuboid 3.0 in width, 10.0 in height, and 3.0 in length
  // with its origin at the center of its bottom)
  var vertices = new Float32Array([
    1.5, 10.0, 1.5, -1.5, 10.0, 1.5, -1.5,  0.0, 1.5,  1.5,  0.0, 1.5, // v0-v1-v2-v3 front
    1.5, 10.0, 1.5,  1.5,  0.0, 1.5,  1.5,  0.0,-1.5,  1.5, 10.0,-1.5, // v0-v3-v4-v5 right
    1.5, 10.0, 1.5,  1.5, 10.0,-1.5, -1.5, 10.0,-1.5, -1.5, 10.0, 1.5, // v0-v5-v6-v1 up
   -1.5, 10.0, 1.5, -1.5, 10.0,-1.5, -1.5,  0.0,-1.5, -1.5,  0.0, 1.5, // v1-v6-v7-v2 left
   -1.5,  0.0,-1.5,  1.5,  0.0,-1.5,  1.5,  0.0, 1.5, -1.5,  0.0, 1.5, // v7-v4-v3-v2 down
    1.5,  0.0,-1.5, -1.5,  0.0,-1.5, -1.5, 10.0,-1.5,  1.5, 10.0,-1.5  // v4-v7-v6-v5 back
  ]);

  // Normal
  var normals = new Float32Array([
    0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0, 0.0, 1.0,  0.0, 0.0, 1.0, // v0-v1-v2-v3 front
    1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0, 0.0, 0.0,  1.0, 0.0, 0.0, // v0-v3-v4-v5 right
    0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0, 1.0, 0.0,  0.0, 1.0, 0.0, // v0-v5-v6-v1 up
   -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, -1.0, 0.0, 0.0, // v1-v6-v7-v2 left
    0.0,-1.0, 0.0,  0.0,-1.0, 0.0,  0.0,-1.0, 0.0,  0.0,-1.0, 0.0, // v7-v4-v3-v2 down
    0.0, 0.0,-1.0,  0.0, 0.0,-1.0,  0.0, 0.0,-1.0,  0.0, 0.0,-1.0  // v4-v7-v6-v5 back
  ]);

  // Indices of the vertices
  var indices = new Uint8Array([
     0, 1, 2,   0, 2, 3,    // front
     4, 5, 6,   4, 6, 7,    // right
     8, 9,10,   8,10,11,    // up
    12,13,14,  12,14,15,    // left
    16,17,18,  16,18,19,    // down
    20,21,22,  20,22,23     // back
  ]);
```

Sample Program (JointModel.js)

- Because the robot arm in this program is moved by using the arrow keys, the event handler *keydown()* is registered at line 71:

```
70   // Register the event handler to be called when keys are pressed
71   document.onkeydown = function(ev){ keydown(ev, gl, n, viewProjMatrix,
                                                ➥u_MvpMatrix, u_NormalMatrix); };
```

Sample Program (JointModel.js)

```
var ANGLE_STEP = 3.0;     // The increments of rotation angle (degrees)
var g_arm1Angle = -90.0; // The rotation angle of arm1 (degrees)
var g_joint1Angle = 0.0; // The rotation angle of joint1 (degrees)

function keydown(ev, gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
  switch (ev.keyCode) {
    case 38: // Up arrow key -> the positive rotation of joint1 around the z-axis
      if (g_joint1Angle < 135.0) g_joint1Angle += ANGLE_STEP;
      break;
    case 40: // Down arrow key -> the negative rotation of joint1 around the z-axis
      if (g_joint1Angle > -135.0) g_joint1Angle -= ANGLE_STEP;
      break;
    case 39: // Right arrow key -> the positive rotation of arm1 around the y-axis
      g_arm1Angle = (g_arm1Angle + ANGLE_STEP) % 360;
      break;
    case 37: // Left arrow key -> the negative rotation of arm1 around the y-axis
      g_arm1Angle = (g_arm1Angle - ANGLE_STEP) % 360;
      break;
    default: return; // Skip drawing at no effective action
  }
  // Draw the robot arm
  draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix);
}
```

Sample Program (JointModel.js)

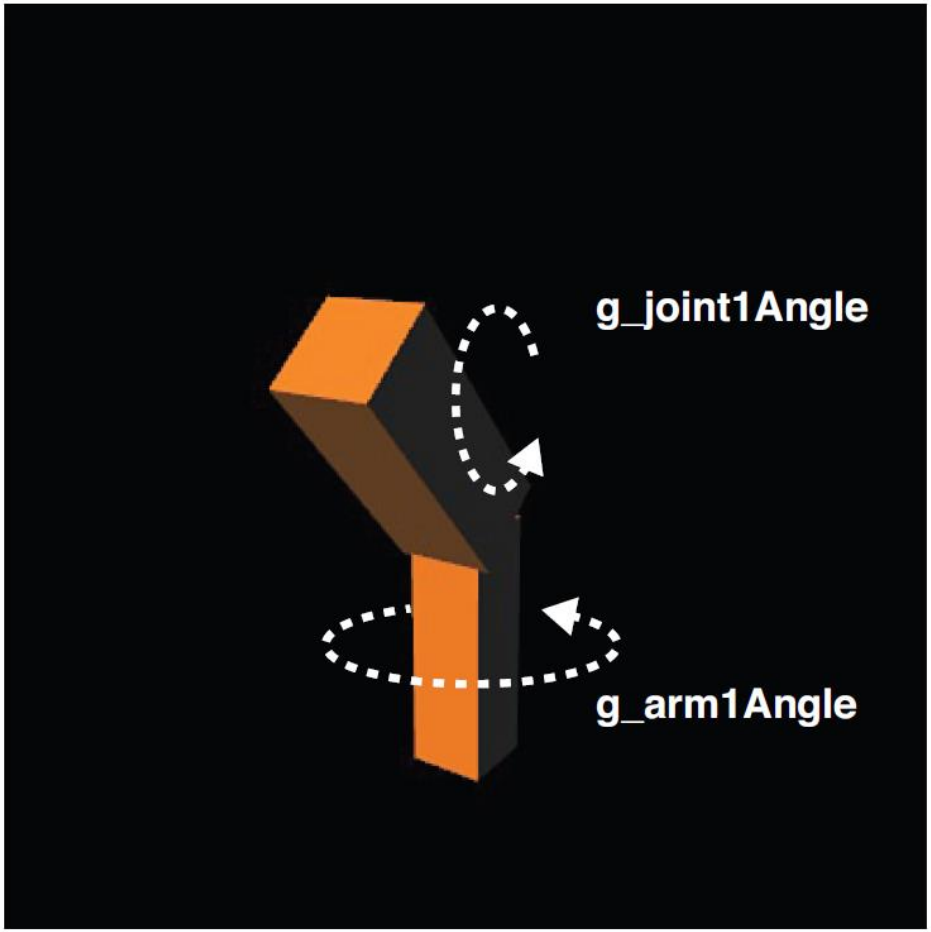


Figure 9.7 g_joint1Angle and g_arm1Angle

Sample Program (JointModel.js)

- Draw the Hierarchical Structure (draw())

  The draw() function draws the robotic arm according to its hierarchical structure.

  Two global variables, *g_modelMatrix* and *g_mvpMatrix*, are created and will be used in both *draw()* and *drawBox().*

Sample Program (JointModel.js)

```
// Coordinate transformation matrix
var g_modelMatrix = new Matrix4(), g_mvpMatrix = new Matrix4();

function draw(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
  // Clear color and depth buffer
  gl.clear(gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);

  // Arm1
  var arm1Length = 10.0; // Length of arm1
  g_modelMatrix.setTranslate(0.0, -12.0, 0.0);
  g_modelMatrix.rotate(g_arm1Angle, 0.0, 1.0, 0.0);    // Rotate around the y-axis
  drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix); // Draw

  // Arm2
  g_modelMatrix.translate(0.0, arm1Length, 0.0);    // Move to joint1
  g_modelMatrix.rotate(g_joint1Angle, 0.0, 0.0, 1.0);  // Rotate around the z-axis
  g_modelMatrix.scale(1.3, 1.0, 1.3); // Make it a little thicker
  drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix); // Draw
}
```

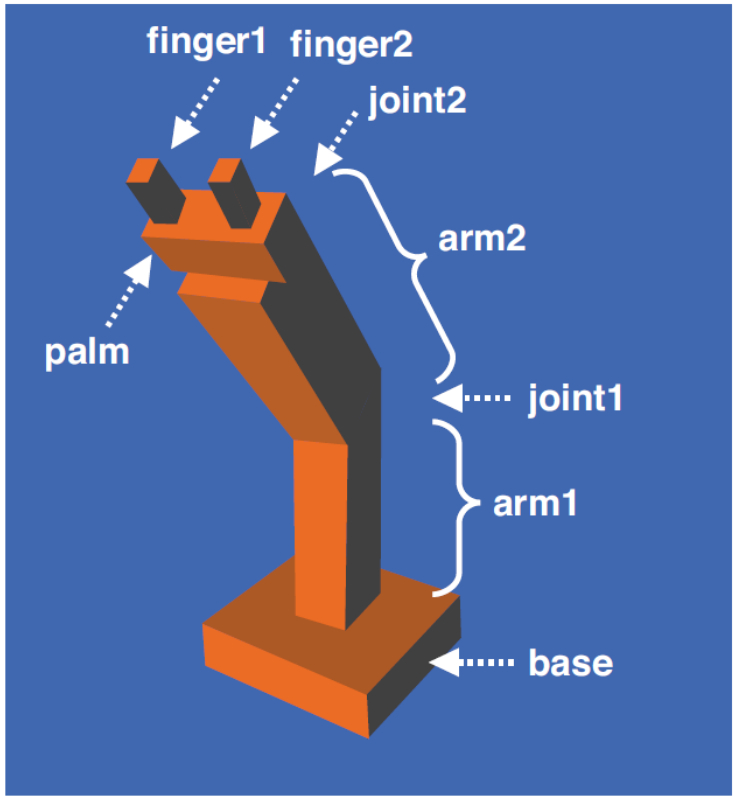
Sample Program (JointModel.js)

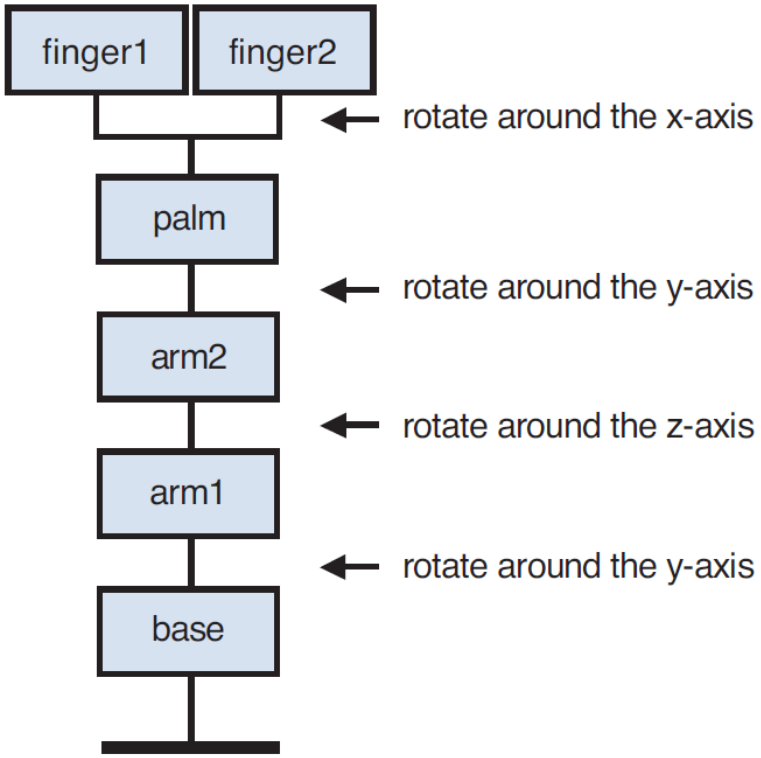
```
var g_normalMatrix = new Matrix4(); // Coordinate transformation matrix for normals

// Draw the cube
function drawBox(gl, n, viewProjMatrix, u_MvpMatrix, u_NormalMatrix) {
  // Calculate the model view project matrix and pass it to u_MvpMatrix
  g_mvpMatrix.set(viewProjMatrix);
  g_mvpMatrix.multiply(g_modelMatrix);
  gl.uniformMatrix4fv(u_MvpMatrix, false, g_mvpMatrix.elements);
  // Calculate the normal transformation matrix and pass it to u_NormalMatrix
  g_normalMatrix.setInverseOf(g_modelMatrix);
  g_normalMatrix.transpose();
  gl.uniformMatrix4fv(u_NormalMatrix, false, g_normalMatrix.elements);
  // Draw
  gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
}
```

Sample Program (JointModel.js)

- When drawing each part, the same process is repeated: (1) translation (*setTranslate()*, *translate()*), (2) rotation (*rotate()*), and (3) drawing the part (*drawBox()*).
- When drawing a hierarchical model performing a rotation, typically you will process from upper to lower in the order of (1) translation, (2) rotation, and (3) drawing segments.

A Multijoint Model





←→: arm1 rotation, ↑ ↓ : joint1 rotation, xz: joint2(wrist) rotation, cv: finger rotation

Figure 9.8 The hierarchical structure of MultiJointModel

A Multijoint Model

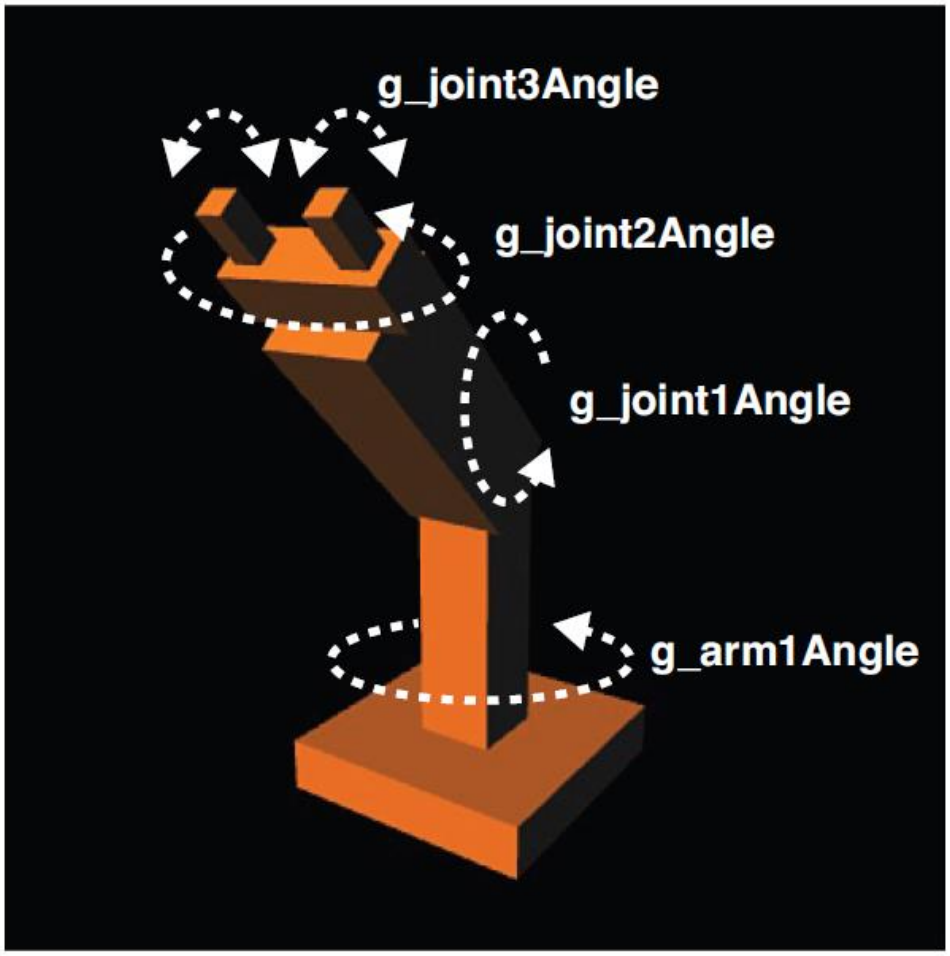


Figure 9.9 The variables controlling the rotation of segments

Sample Program (MultiJointModel.js)

- Because finger1 and finger2 do not have a parent-child relationship, a little more care is needed. In particular, we have to pay attention to the contents of the model matrix.
- We need to store the model matrix before drawing finger1 and retrieving it after drawing finger1.

Sample Program (MultiJointModel.js)

- This approach can be used to draw an arbitrarily long robot arm. It will scale when new segments are added to the hierarchy. You only need to use *pushMatrix()* and *popMatrix()* when the hierarchy structure is a sibling relation, not a parent-child relation.

```
234 var g_matrixStack = [];  // Array for storing matrices
235 function pushMatrix(m) { // Store the specified matrix
236   var m2 = new Matrix4(m);
237   g_matrixStack.push(m2);
238 }
239
240 function popMatrix() { // Retrieve a matrix from the array
241   return g_matrixStack.pop();
242 }
```

Sample Program (MultiJointModel.js)

- Draw Segments (drawBox())
  - The three-dimensional object used here, unlike JointModel, is a cube whose side is 1.0 unit long. Its origin is located at the center of the bottom surface so that you can easily rotate the arms, the palm, and the fingers.
  - The third to fifth arguments, *width* , *height*, and *depth*, specify the width, height, and depth of the cuboid being drawn.

Sample Program (MultiJointModel.js)

- Draw Segments (drawBox())
  - Since the model matrix is multiplied by a scaling matrix at *line 250* so that the cube will be drawn with the size specified by width , height, and depth.
  - Note that we store the model matrix at *line 248* and retrieve it at *line 261* using *pushMatrix()* and *popMatrix()* .

```
248   pushMatrix(g_modelMatrix);   // Save the model matrix
249     // Scale a cube and draw
250     g_modelMatrix.scale(width, height, depth);
```

Display Shadows

- Shadows are important components of realistic images and give many visual cues to the spatial relationships among the objects in a scene.
- Shadows require a light source to be present.
- A point is in shadow if it is not illuminated by any light source, or, equivalently, if a viewer at that point cannot see any light sources.

Display Shadows

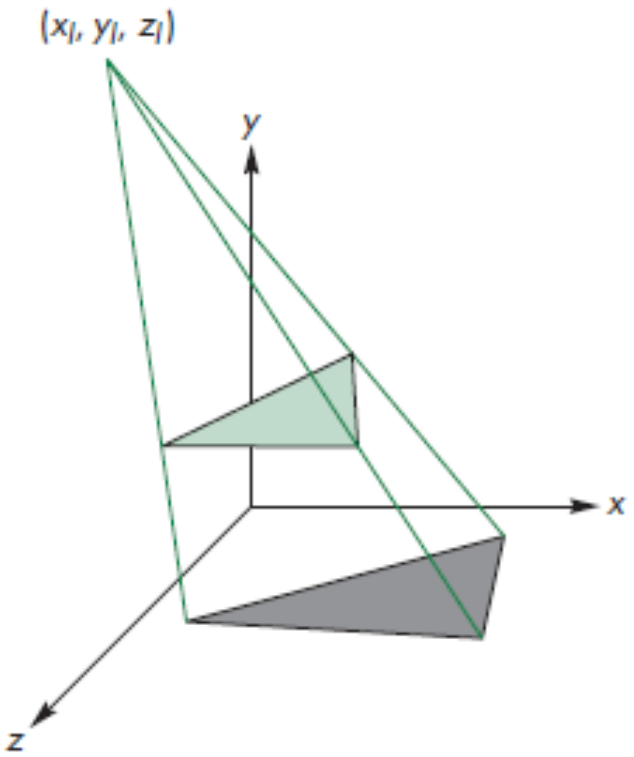
- There are several methods to realize shadowing, but we will explain a method that uses a shadow map (depth map).
- This method is quite expressive and used in a variety of computer graphics situations and even in special effects in movies.

Display Shadows

- The shadow map method is based on the idea that the sun cannot see the shadow of objects.
- Essentially, it works by considering the viewer's eye point to be at the same position as the light source and determining what can be seen from that point. All the objects you can see would appear to be in the light. Anything behind those objects would be in shadow.
- With this method, you can use the distance to the objects from the light source to judge whether the objects are visible.

Projections and Shadows

- Projected Shadows
  - The shadow is a flat polygon, called a shadow polygon.
  - It is a projection of the original polygon onto the surface.
  - The center of projection is at the light source.

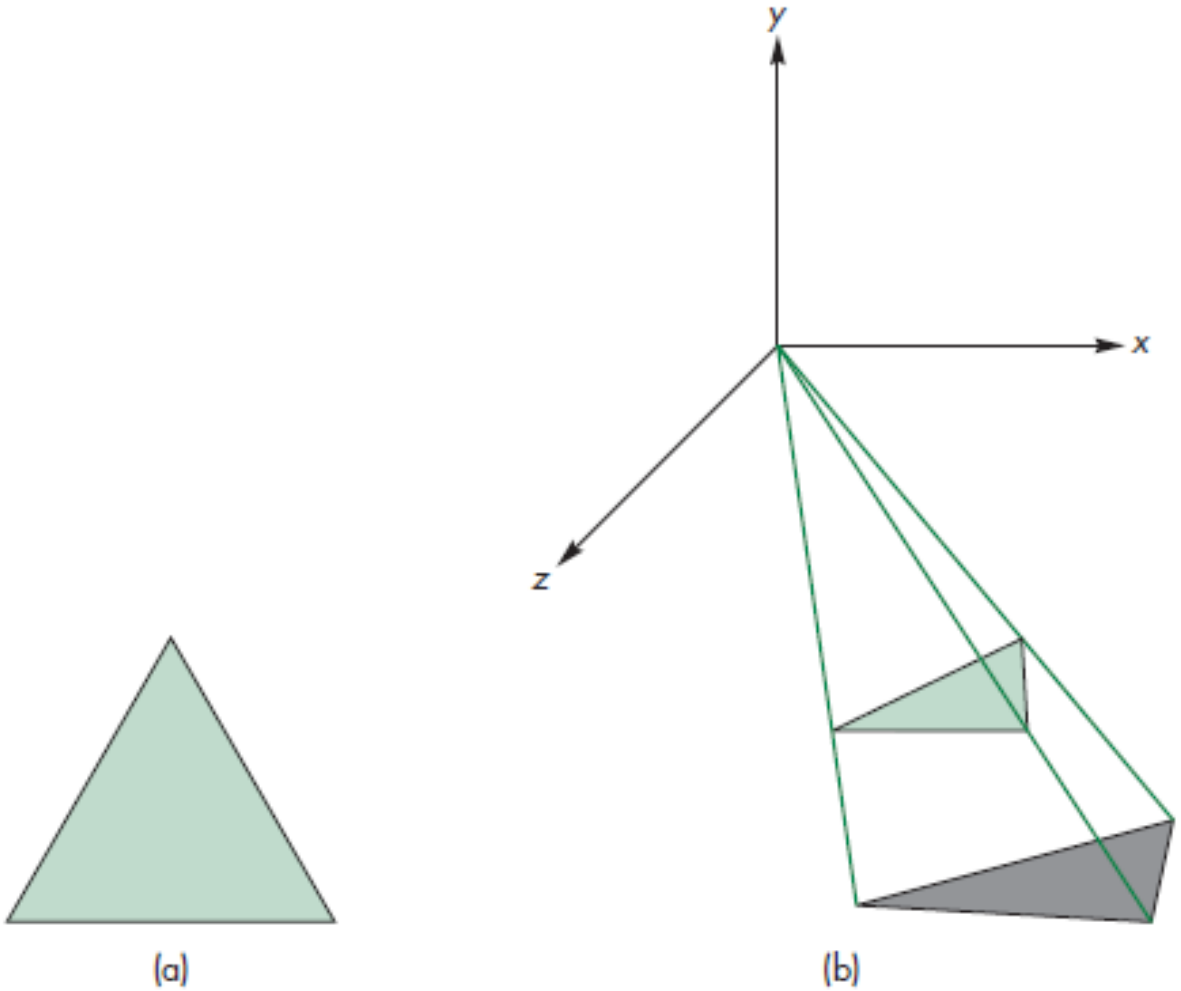


Shadow from a single polygon.

Projections and Shadows

- Projected Shadows
  - If we do a projection onto the plane of a surface in a frame in which the light source is at the origin, we obtain the vertices of the shadow polygon.
  - These vertices must then be converted back to a representation in the object frame.

Projections and Shadows



Shadow polygon projection. (a) From a light source. (b) With source moved to the origin.

Projections and Shadows

- Projected Shadows
  - Suppose we start a light source at (x_l, y_l, z_l).
  - We put the light source at the origin by using a translation matrix $\mathrm{T}(-x_l, -y_l, -z_l)$, then we have a simple perspective projection through the origin. The projection matrix is

$$\mathbf{M} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{-y_l} & 0 & 0 \end{bmatrix}.$$

Projections and Shadows

- Projected Shadows
  - Finally, we translate everything back with $\mathrm{T}(x_l, y_l, z_l)$. The concatenation of this matrix and the two translation matrices projects the vertex (x, y, z) to

$$x_p = x_l - \frac{x - x_l}{(y - y_l)/y_l}$$

$$y_p = 0$$

$$z_p = z_l - \frac{z - z_l}{(y - y_l)/y_l}.$$

Projections and Shadows

- Although we are performing a projection with respect to the light source, the matrix that we use is the model-view matrix.

How to Implement Shadows

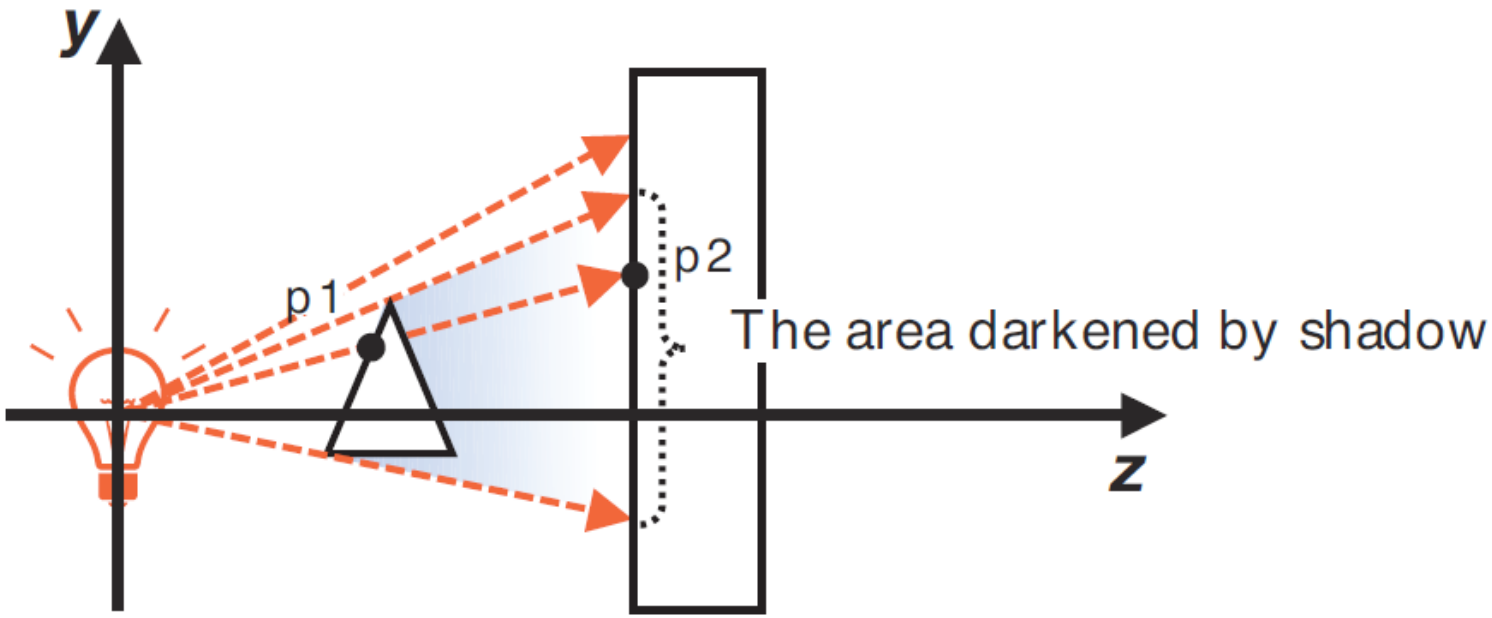


Figure 10.21 Theory of shadow map

P2 is in the shadow because the distance from the light source to P2 is longer than P1.

How to Implement Shadows

- You need two pairs of shaders for this process: (1) a pair of shaders that calculate the distance from the light source to the objects, and (2) a pair of shaders that draws the shadow using the calculated distance.
- Then you need a method to pass the distance data from the light source calculated in the first pair of shaders to the second pair of shaders.
- You can use a texture image for this purpose. This texture image is called the shadow map, so this method is called shadow mapping .

How to Implement Shadows

- The shadow mapping technique consists of the following two processes:
  1. Move the eye point to the position of the light source and draw objects from there. Because the fragments drawn from the position are hit by the light, you write the distances from the light source to each fragment in the texture image (shadow map).
  2. Move the eye point back to the position from which you want to view the objects and draw them from there. Compare the distance from the light source to the fragments drawn in this step and the distance recorded in the shadow map from step 1. If the former distance is greater, you can draw the fragment as in shadow (in the darker color).

How to Implement Shadows

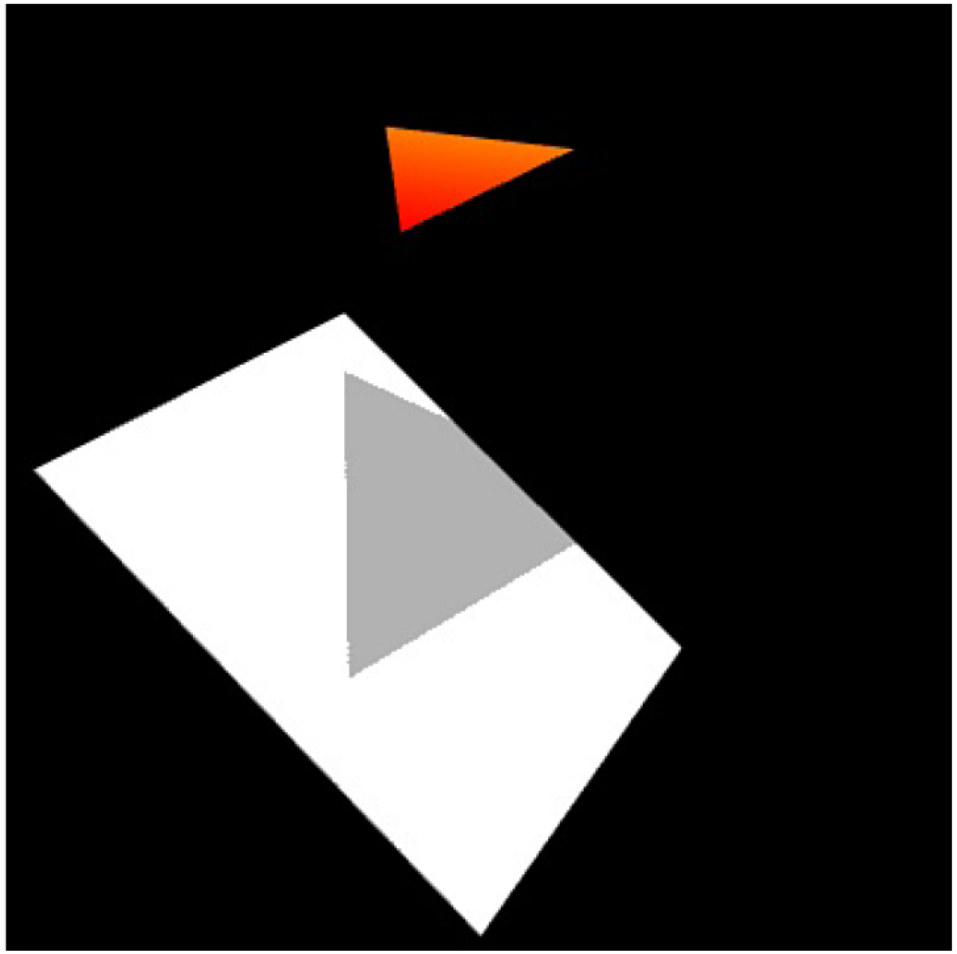


Figure 10.22 Shadow