Basics of 3D Graphics

Rui Ying

Compute Normals

Since normals are for vertices, we can iterate through vertices to compute their normal.

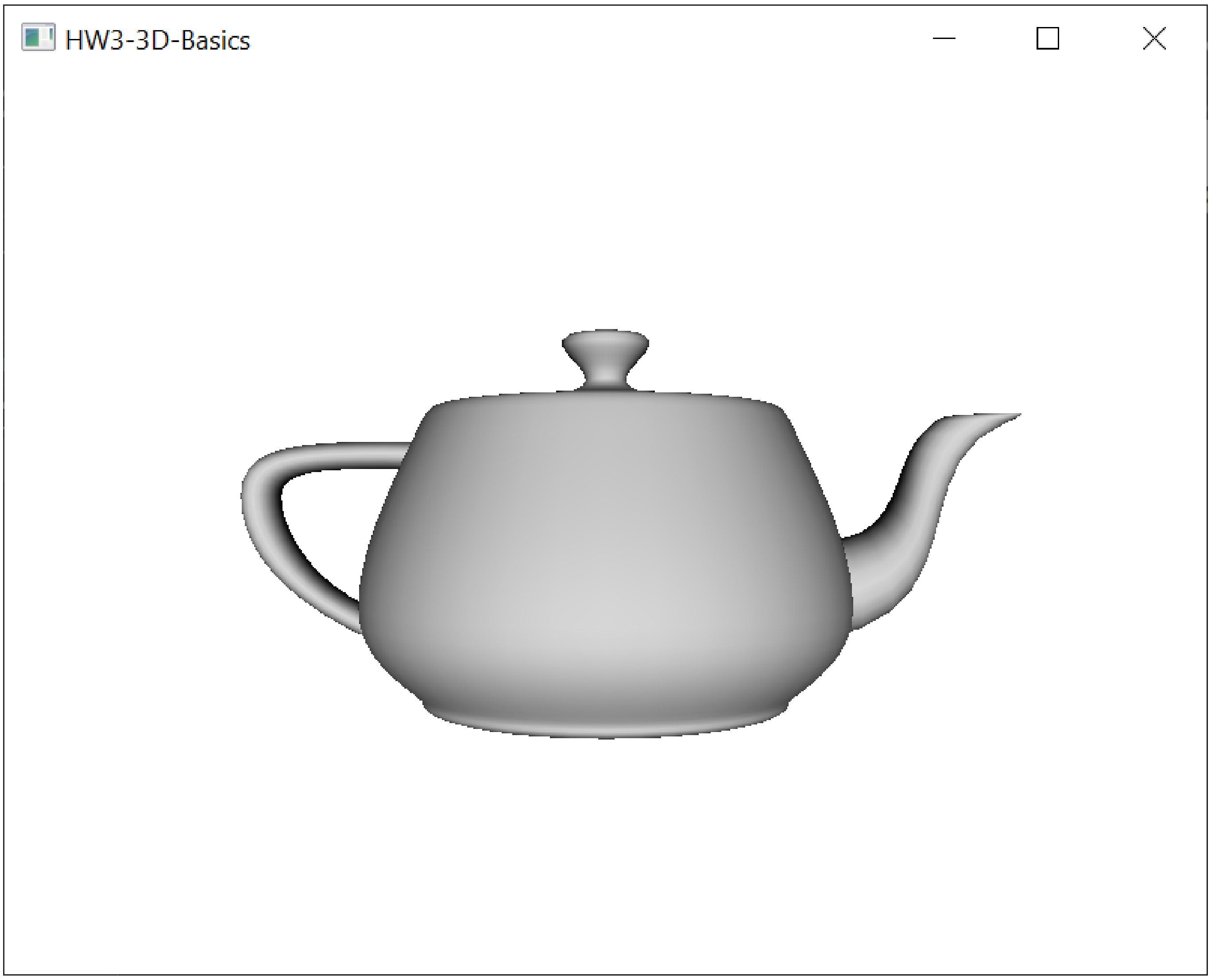
Because the normal of one vertex may be determined by the triangles that share it, we first need to find the triangles.

Once we find a triangle, we use the cross product to calculate the normal of the triangle surface and normalize it.

According to the defination, the normal of one vertex is the average normals of the surfaces that share it. So we not only have to do average but also normalize the final result so it will be a **normal**.

```
for (int i = 0; i < g_meshNormals.size(); i += 3)
{
    int vertexIndex = i / 3;

    // save to compute average
    float sum[3] = { 0.0f, 0.0f, 0.0f };
    int count = 0;
    for (int j = 0; j < g_meshIndices.size(); j += 3) {
        if (g_meshIndices[j] == vertexIndex ||
            g_meshIndices[j + 1] == vertexIndex ||
            g_meshIndices[j + 2] == vertexIndex) {
            // compute the normal of this found triangle
            float a[3] = { g_meshVertices[g_meshIndices[j + 1] * 3] -
g_meshVertices[g_meshIndices[j] * 3],
                            g_meshVertices[g_meshIndices[j + 1] * 3 + 1] -
g_meshVertices[g_meshIndices[j] * 3 + 1],
                            g_meshVertices[g_meshIndices[j + 1] * 3 + 2] -
g_meshVertices[g_meshIndices[j] * 3 + 2] };
            float b[3] = { g_meshVertices[g_meshIndices[j + 2] * 3] -
g_meshVertices[g_meshIndices[j + 1] * 3],
                            g_meshVertices[g_meshIndices[j + 2] * 3 + 1] -
g_meshVertices[g_meshIndices[j + 1] * 3 + 1],
                            g_meshVertices[g_meshIndices[j + 2] * 3 + 2] -
g_meshVertices[g_meshIndices[j + 1] * 3 + 2] };
            float r[3];
            crossProduct(a, b, r);
            normalize(r);
            sum[0] += r[0];
            sum[1] += r[1];
            sum[2] += r[2];
            count++;
        }
    }
```

```
        // average and normalize
        sum[0] /= count;
        sum[1] /= count;
        sum[2] /= count;
        normalize(sum);
        g_meshNormals[i] = sum[0];
        g_meshNormals[i + 1] = sum[1];
        g_meshNormals[i + 2] = sum[2];
    }
```



Make the teapot rotate

ModelView matrix is a homogeneous one. So it's generally in this form:

```
[R  0]
[0  1]
```

where R is a rotation matrix.

According to Wikipedia, 3D rotation matrix around y-axis would be the second one below:

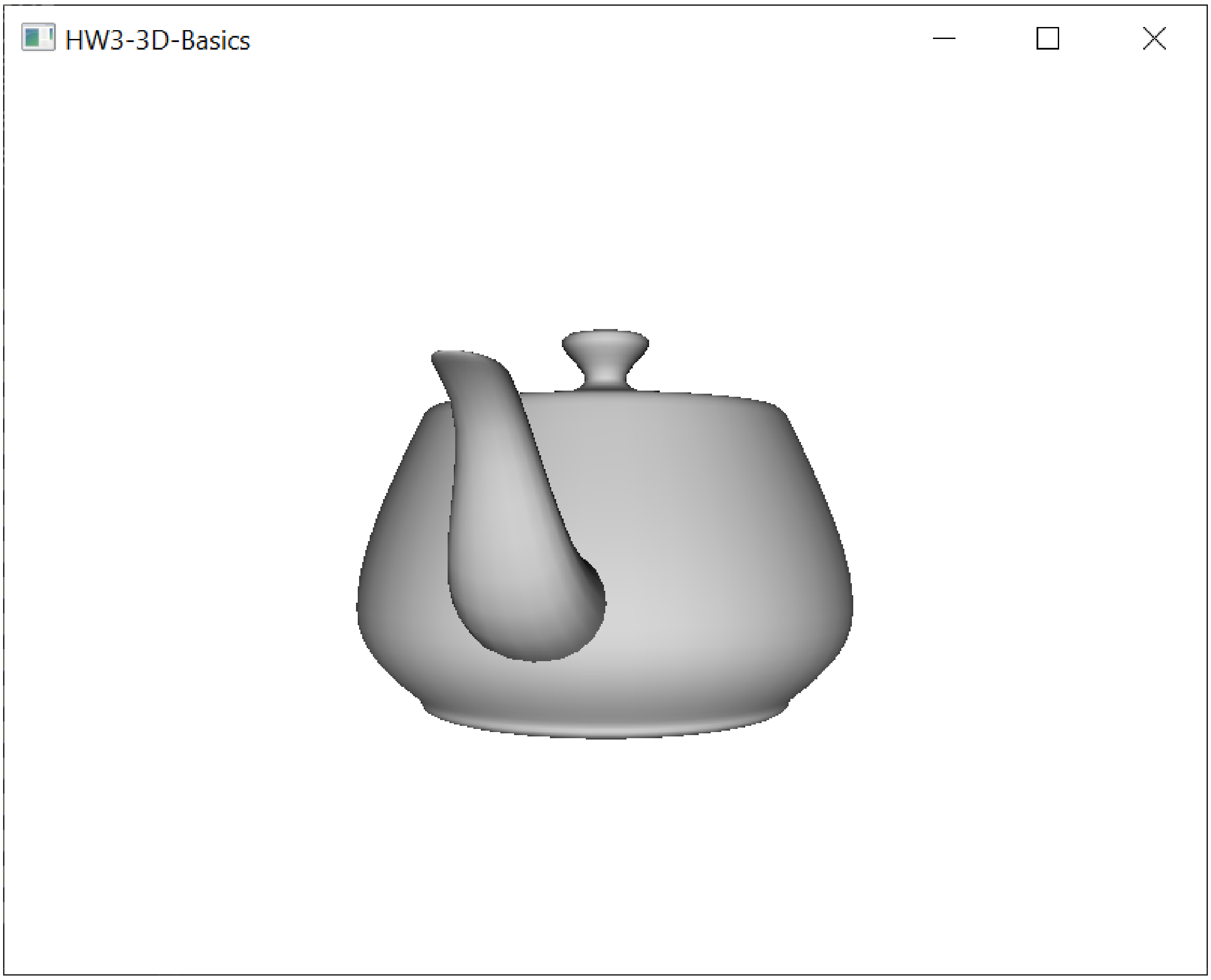
$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Set R and also the distance for z-axis, we can rotate the teapot and see it from a distance.

```
// angle is the rotation angle simulated from system time
// angle = getTime() * 50;
g_modelViewMatrix[0] = std::cos(angle / 180.f * M_PI);
g_modelViewMatrix[2] = std::sin(angle / 180.f * M_PI);
g_modelViewMatrix[5] = 1.0f;
g_modelViewMatrix[8] = -std::sin(angle / 180.f * M_PI);
g_modelViewMatrix[10] = std::cos(angle / 180.f * M_PI);

g_modelViewMatrix[14] = -distance;
g_modelViewMatrix[15] = 1.0f;
```



Dolly Zoom Effect and Orthogonal Projection

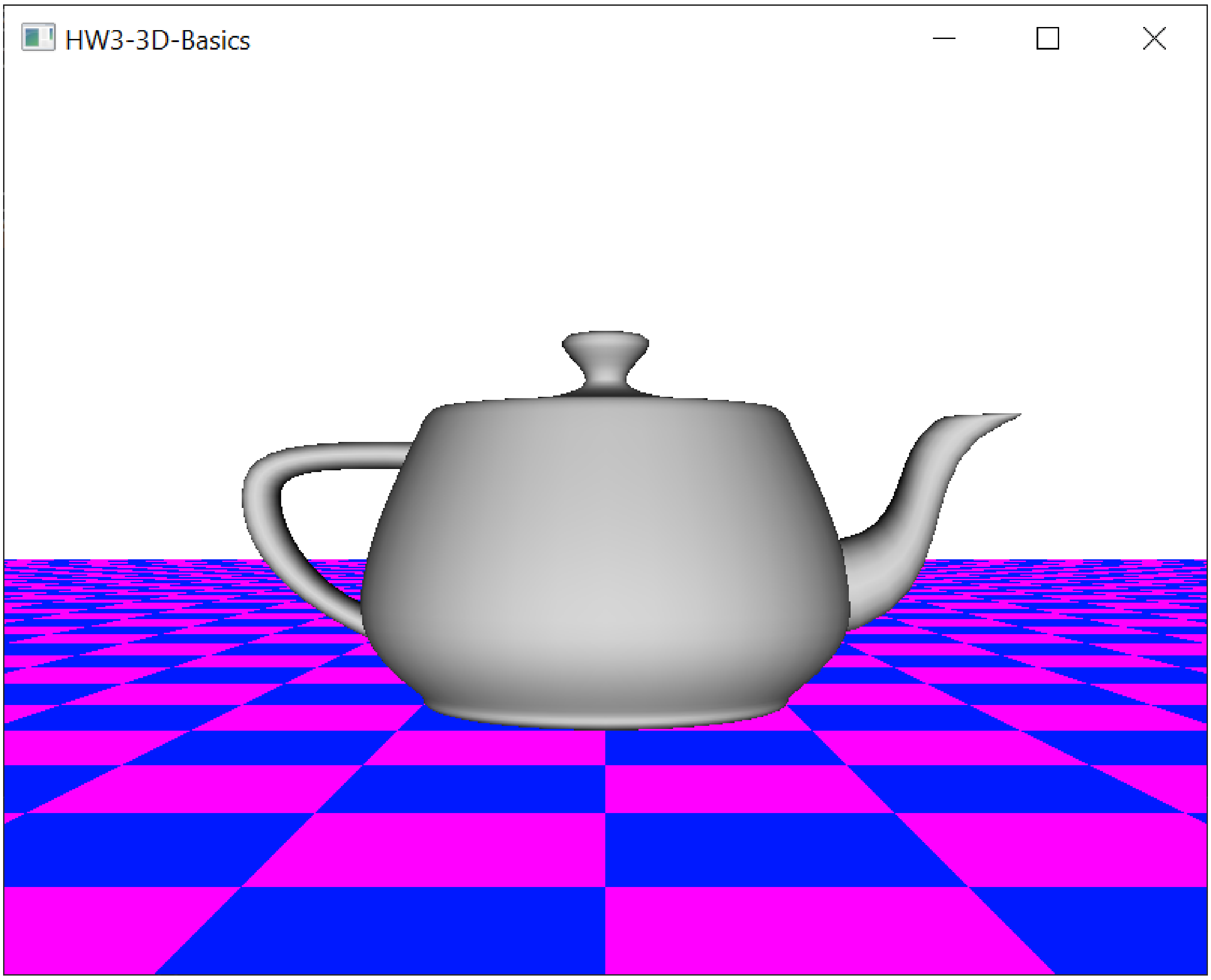
Dolly Zoom

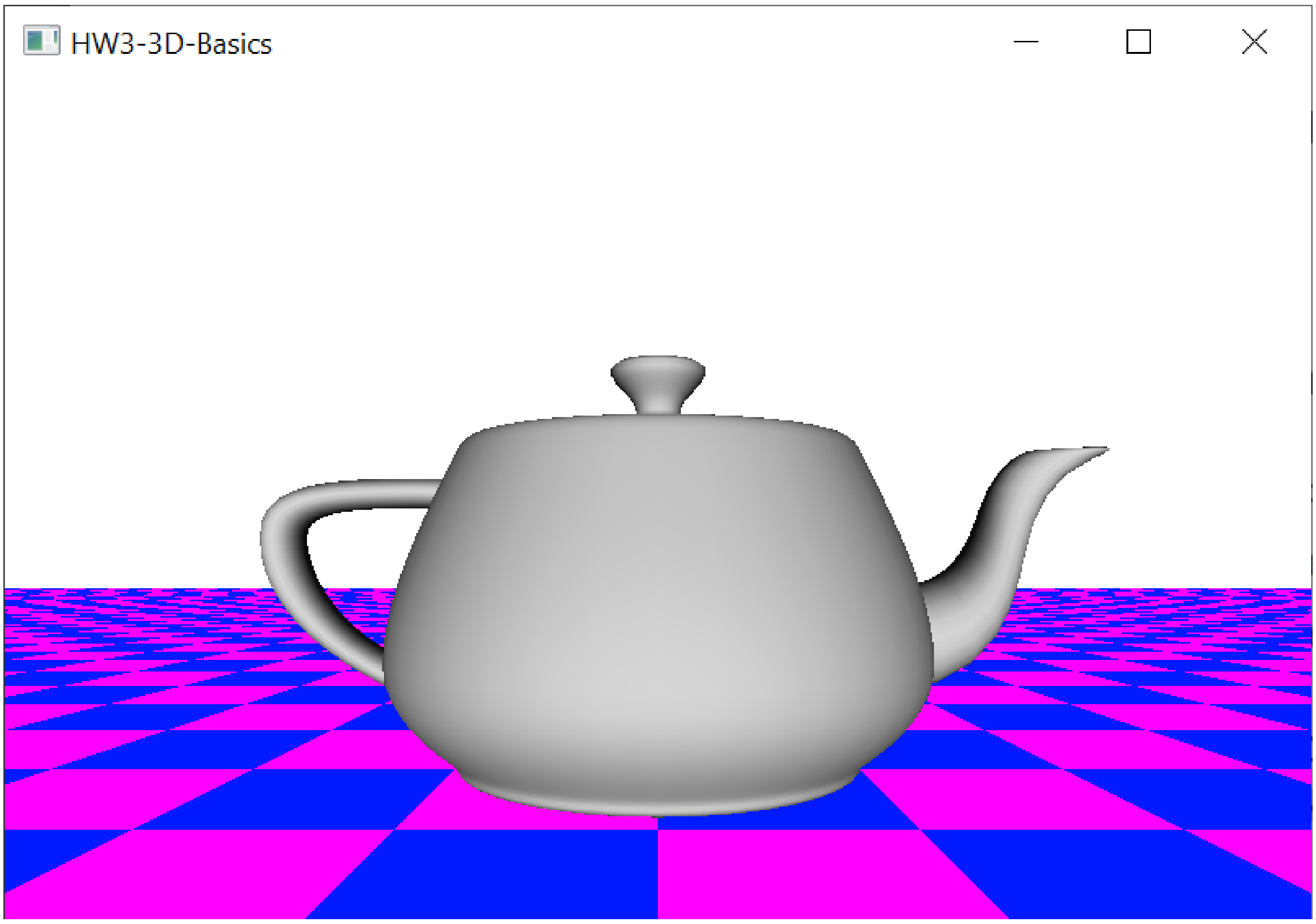
According to the slides, we calculate the `halfWidth` first and then by changing `fov` we get different `distance`. They keep a relationship and produce the dolly zoom effect.

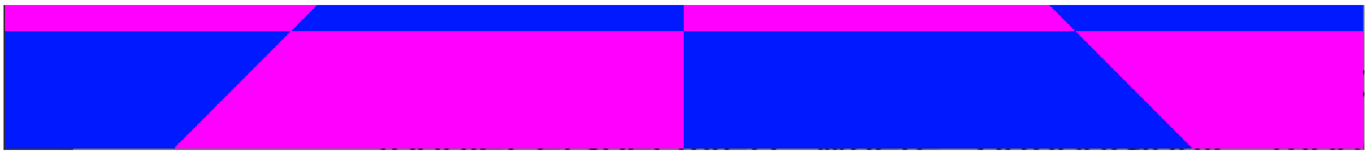
```
// fov is simulated from system time
// fov = getTime() * 30 / 180.f * M_PI;
distance = halfWidth / std::tan(fov / 2);
float fovInDegree = radianToDegree(fov);
gluPerspective(fovInDegree, (GLfloat)g_windowWidth / (GLfloat)g_windowHeight,
1.0f, 40.f);
```

Use the following code to prevent the camera inside the teapot:

```
if (closer) {
    fov = pauseFov + (getTime() - pauseDollyTime) / 5;
}
else {
    fov = pauseFov - (getTime() - pauseDollyTime) / 5;
}
if (fov > 60.0 / 180 * M_PI) {
    closer = false;
    pauseFov = fov;
    pauseDollyTime = getTime();
}
if (fov < 30.0 / 180 * M_PI) {
    closer = true;
    pauseFov = fov;
    pauseDollyTime = getTime();
}
```


HW3-3D-Basics


HW3-3D-Basics



Orthogonal Projection

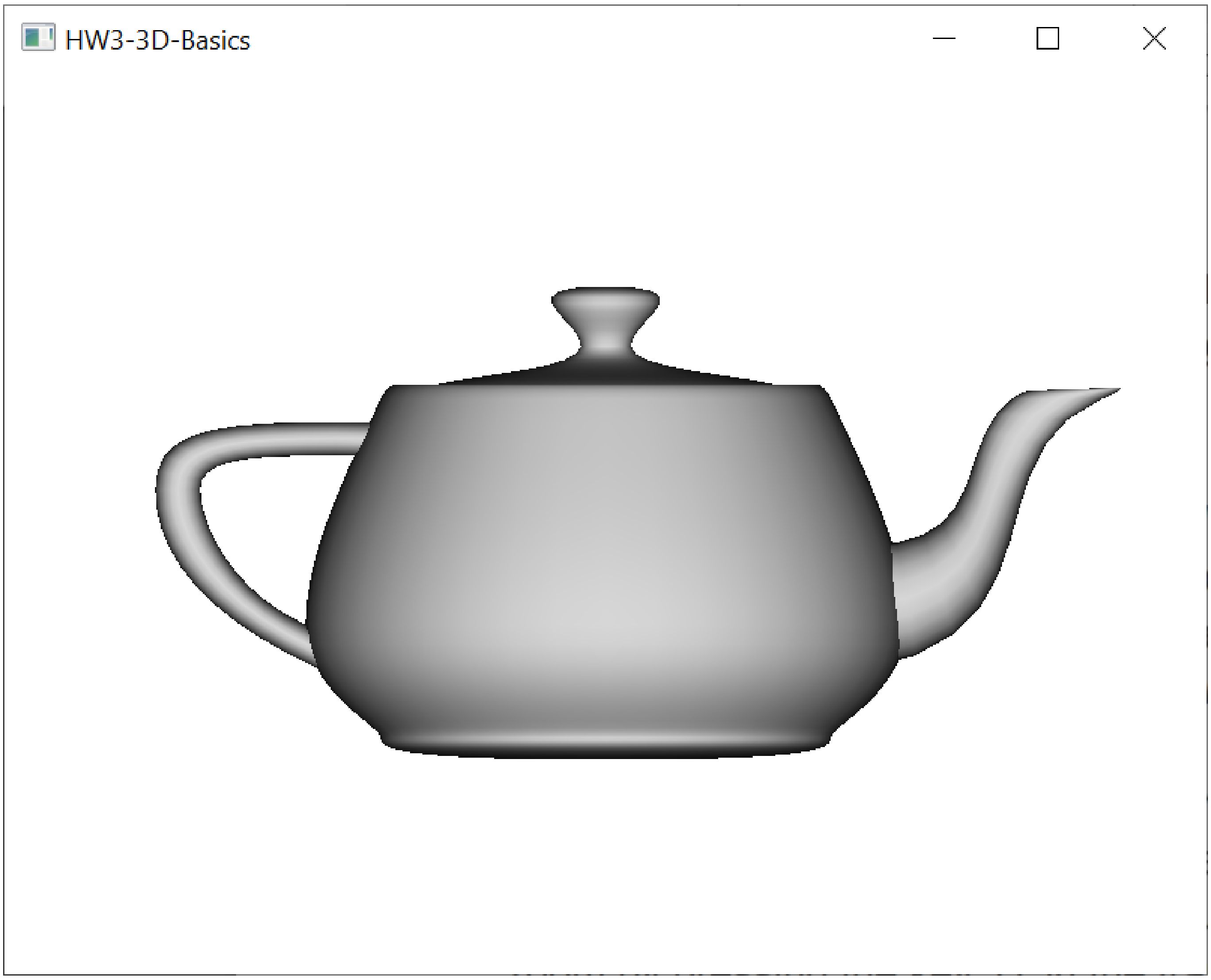
`glOrtho()` takes

```
void WINAPI glOrtho(
   GLdouble left,
   GLdouble right,
   GLdouble bottom,
   GLdouble top,
   GLdouble zNear,
   GLdouble zFar
);
```

The question is to find suitable values for `bottom` and `top`.

To keep the teapot original scale, we can let `bottom` and `top` be at the ratio of `windowHeight` and `windowWidth` for `left` and `right`.

```
glScalef(0.5, 0.5, 0.5);
float ratio = float(g_windowHeight) / g_windowWidth;
glOrtho(-1.f, 1.f, -1.f*ratio, 1.f*ratio, 1.f, 40.f);
```



Extra

Rotation about the center of the body

The teapot currently rotates around y-axis, so if we want it to rotate about its body center, we may need to move the object.

So in `loadObj` when the teapot is being given coordinates, we can add a offset to `x` so the the body center is exactly on y-axis. Rotating looks better this way.

```
float x, y, z;
nfile >> x >> y >> z;
g_meshVertices.push_back(x + 1.1f);
g_meshVertices.push_back(y);
g_meshVertices.push_back(z);
```

Physically-realistic torque

Just use some global value to simulate the rotating velocity. And take care of some corner cases to make the animation transition smooth.

```
if (velocity > 200) {
    faster = false;
}
if (velocity < -150) {
    faster = true;
}

if (faster) {
    velocity++;
    angle = int(pauseAngle + std::log2(velocity < 1 ? 1 : velocity)) % 360;
}
else {
    velocity--;
    angle = int(pauseAngle + std::log2(velocity < 1 ? 1 : velocity)) % 360;
}
if (angle == -128) {
    std::cout << velocity << std::endl;
}

pauseSpinTime = getTime();
pauseAngle = angle;
```

Pause / resume

In order to resume after pausing, we need to memorize some values for the future resuming.

For spinning, we can use:

```
// Spin
double angle = 0.0;
double pauseAngle = 0.0;
double pauseSpinTime = 0.0;
int velocity = 0;
bool faster = true;

//
if (teapotSpin) {
    if (velocity > 200) {
        faster = false;
    }
    if (velocity < -150) {
        faster = true;
    }

    if (faster) {
        velocity++;
        angle = int(pauseAngle + std::log2(velocity < 1 ? 1 : velocity)) % 360;
    }
    else {
        velocity--;
        angle = int(pauseAngle + std::log2(velocity < 1 ? 1 : velocity)) % 360;
    }
    if (angle == -128) {
        std::cout << velocity << std::endl;
    }

    pauseSpinTime = getTime();
    pauseAngle = angle;
}
else {
    pauseSpinTime = getTime();
    pauseAngle = angle;
}
```

where `pauseAngle` memorize the angle for the last time and `pauseSpinTime` to re-calculate time intervals when resumed, otherwise the time interval can accumulate and we cannot go back where we were after resuming.

Almost the same for dolly zoom:

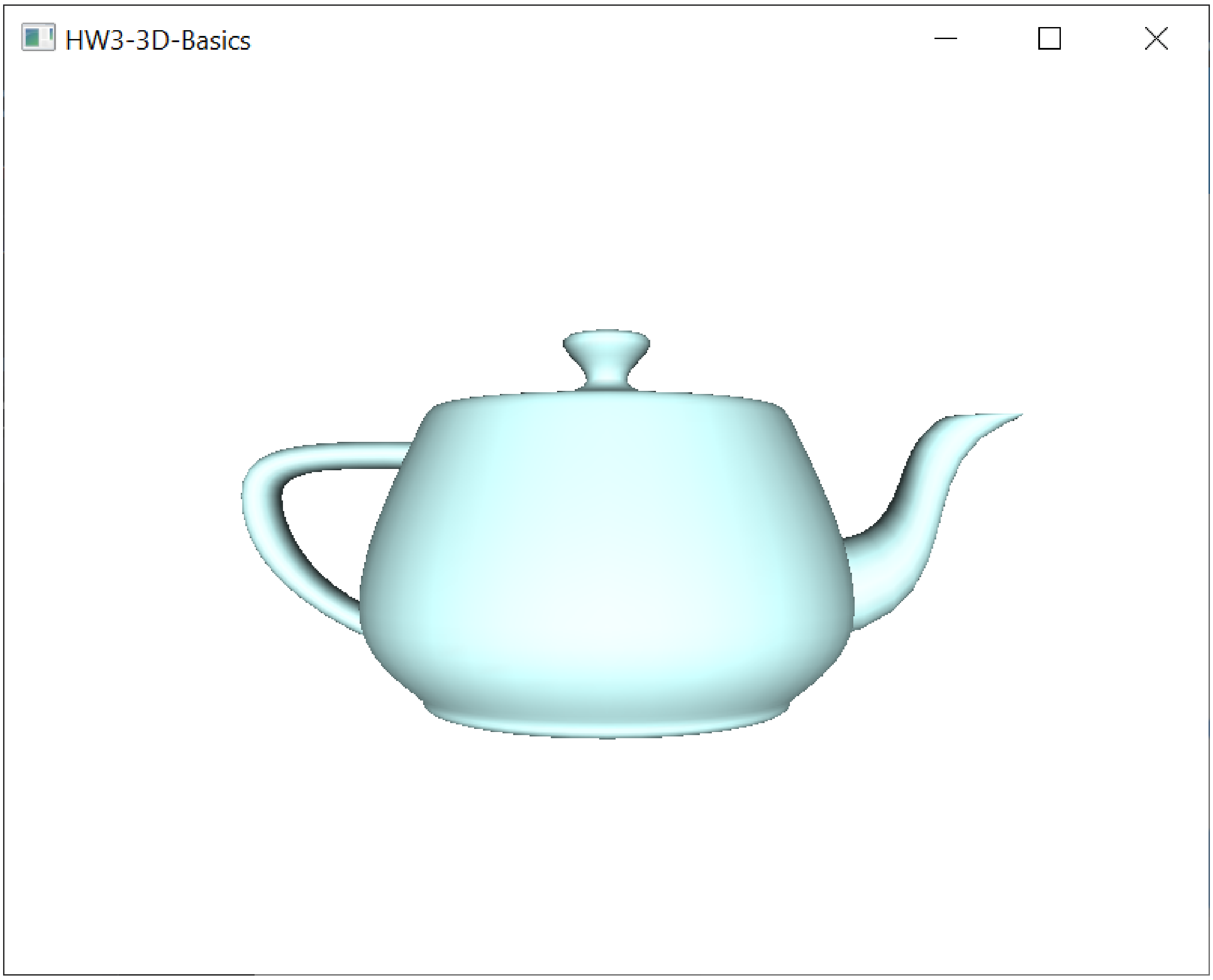
```
// Dolly zoom options
float fov = M_PI / 4.f;
float distance = 4.5f;
float halfWidth = distance * std::tan(fov / 2);
double pauseDollyTime = 0.0;
double pauseFov = M_PI / 4.f;
bool closer = true;

//
if (enableDolly) {
    if (closer) {
        fov = pauseFov + (getTime() - pauseDollyTime) / 5;
    }
    else {
        fov = pauseFov - (getTime() - pauseDollyTime) / 5;
    }
    if (fov > 60.0 / 180 * M_PI) {
        closer = false;
        pauseFov = fov;
        pauseDollyTime = getTime();
    }
    if (fov < 30.0 / 180 * M_PI) {
        closer = true;
        pauseFov = fov;
        pauseDollyTime = getTime();
    }
    distance = halfWidth / std::tan(fov / 2);
}
else {
    pauseDollyTime = getTime();
    pauseFov = fov;
}
```

where `pauseDollyTime` share the same function as `pauseSpinTime` and `pauseFov` memorizes last-time `fov`.

Change the color

By looking up some information, we can simply change the teapot color by putting the following code before drawing the teapot.

```
glEnable(GL_COLOR_MATERIAL);
glColor3f(204.f / 255, 1.f, 1.f);
```



Overall presentation / YouTube video

Utah teapot - HW3 3D Basics - Computer Graphics

References

1. Rotation matrix - Wikipedia
2. glOrtho function - Microsoft Docs

3. glColor3f not working on gluSphere - Stack Overflow