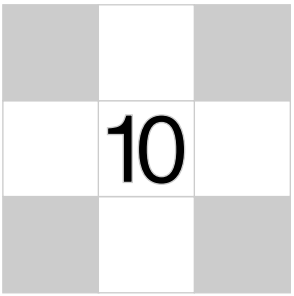

Coordinate Systems

This chapter discusses coordinate systems and the transformation between different coordinate systems. This chapter:

- analyzes the `world2ndc` matrix we have worked with in all of the tutorials;
- explains the need for world and normalized device coordinate systems;
- describes and experiments with the world coordinate window;
- derives all essential coordinate transformation operators and demonstrates how to implement these operators.

After this chapter we should:

- understand the difference between modeling design space and device drawing space;
- understand the need for transforming mouse click positions to the world coordinate space.

In addition, with respect to hands-on programming, we should:

- be able to program $\mathbf{M}_{w2n}$ operators to support any user-specified world coordinate window;
- be able to utilize graphics API matrix processors in supporting necessary coordinate transformation operations;
- be able to transform mouse clicks to the world coordinate space.

10.1 Understanding Tutorial 3.1

Recall that in the first tutorial of Chapter 3, we began by measuring and discussing how to draw two squares with the following measurements:

$$\text{LargeSquare}_{\text{wc}} = \begin{cases} V_a = & (160, \quad 122), \\ V_b = & (80, \quad 122), \\ V_c = & (80, \quad 42), \\ V_d = & (160, \quad 42), \end{cases} \tag{10.1}$$

wc. Abbreviation for world coordinate. We will study the details of this coordinate system in Section 10.3.

and

$$\text{SmallSquare}_{\text{wc}} = \begin{cases} V_a = & (160, \quad 122), \\ V_e = & (210, \quad 122), \\ V_f = & (210, \quad 172), \\ V_g = & (160, \quad 172). \end{cases} \tag{10.2}$$

Tutorial 3.1 displayed these squares where each millimeter was represented with a pixel on the application window.

Tutorial 10.1. The `world2ndc` (`w2n`) Transform

Tutorial 10.1.
Project Name
`D3D_ViewTransform0`

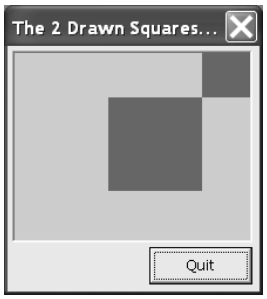

Figure 10.1. Tutorial 10.1: Re-implement Tutorial 3.1 with 200×160 drawing area.

- **Goal.** Understand the `w2n` transformation operator we have encountered in all of the tutorials.
- **Approach.** Analyze the simplest tutorials we have worked with and study the effect of the operator.

Figure 10.1 is a screenshot of running Tutorial 10.1. This tutorial is identical to Tutorial 3.1 except that the UI drawing area is 200 pixels $\times$ 160 pixels. We can verify that vertices V_e, V_f, and V_g are outside of the UI drawing area and are not visible. Recall that before we developed the `UWBGL_D3D_Lib` support, drawing was performed during the `GrfxWindowD3D::OnPaint()` function call, as shown in Listing 10.1. We are now equipped with sufficient knowledge to completely understand this very first graphics API tutorial. In particular, we are interested in understanding the mysterious Step 2 of setting and computing the parameters for the coordinate systems. Based on the knowledge learned from the previous chapters, we understand that in Step 2, we first initialize all three of the D3D matrix processors (`WORLD`, `VIEW`, and `PROJECTION`) to the identity matrix. We then compute the `w2n` matrix by concatenating a translation and a scaling matrix. Finally, we load the `w2n` matrix into the `VIEW` matrix processor. In fact, a similar code

Source file. `GrfxWindowD3D.cpp` file in the *GrfxWindow* folder of the `D3D_ViewTransform0` project.

```
void CGrfxWindowD3D::OnPaint()
    Step 1: select graphics hardware render buffer.
    ⋮
    Step 2: initialize selected hardware and set the coordinate system parameters.
    ⋮
    // Initialize D3D matrix processors setting all three
    // matrix processors to identity matrix
    m_pD3DDevice->SetTransform(D3DTS_WORLD, &identity);
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &identity);
    m_pD3DDevice->SetTransform(D3DTS_PROJECTION, &identity);

    D3DXMATRIX world2ndc;                  // Coordinate Transformation operator
    D3DXVECTOR3 scale(2.0f/width, 2.0f/height, 1.0f); // Parameters for the operator
    D3DXVECTOR3 translate(-1.0f, -1.0f, 0.0);         //
    // compute the coordinate transformation operator
    D3DXMatrixTransformation(&world2ndc, scale, translate);
    // Load the operator into the D3D VIEW matrix
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &world2ndc);

    Step 3: clear drawing buffer and draw two squares.
    ⋮
    v[0].m_point = D3DXVECTOR3(160,122,0);
    v[1].m_point = D3DXVECTOR3( 80,122,0);
    ⋮
    m_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, (CONST void *)v, ...);
    ⋮
```

Listing 10.1. `CGrfxWndD3D::OnPaint()` (Tutorial 10.1)

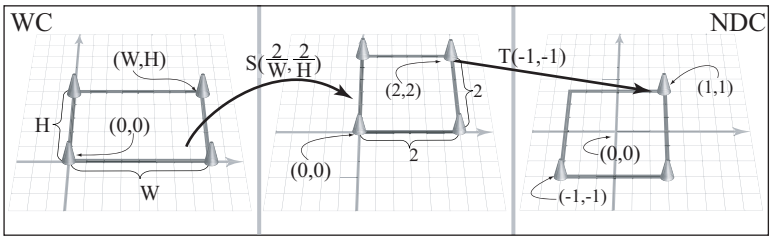

Figure 10.2. The operations of `w2n`.

fragment exists in every tutorial we have worked with. We perform these operations each time we initialize the graphics API for redraw: in the earlier tutorials when servicing the `Redraw/Paint` events (in the `OnPaint()` function); and in the later tutorials, when redrawing the `WindowHandler` UI drawing area (in the `DrawGraphics()` function). Figure 10.2 shows the details of the transformation operation that is implemented by the `w2n` matrix:

- **Scale factor.** Recall that the `width` and `height` parameters are the dimensions of the application window. In Listing 10.1, the scaling factors are defined to be $s_x = \frac{2.0}{\texttt{width}}$ and $s_y = \frac{2.0}{\texttt{height}}$. The diagram on the left of Figure 10.2 shows that this scaling factors will shrink a `width` $\times$ `height` rectangle into a 2×2 rectangle. For example, in this case we have defined the dimension of the application window to be 200 pixels $\times$ 160 pixels.

- **Displacement.** The displacements of the translation operator are constants for all cases: $t_x = -1$ and $t_y = -1$. The diagram on the right of Figure 10.2 shows how this translation operator moves the 2×2 rectangle resulting from the scaling operation.

- `w2n` **matrix.** We will refer to this matrix (or the transformation operator) as $\mathbf{M}_{w2n}$. We see that this is a concatenated operator with a scaling followed by a translation, or

$$\mathbf{M}_{w2n} = \mathbf{ST} = \mathbf{S}\left(\frac{2}{200}, \frac{2}{160}\right)\mathbf{T}(-1,-1). \tag{10.3}$$

Notice that $\mathbf{M}_{w2n}$ is loaded into the `VIEW` matrix processor (or $\mathbf{M}_V$) of the D3D rendering context (RC). In the case of D3D, we know that all vertices $\mathbf{V}_i$ will be transformed by

$$\mathbf{V}_{it} = \mathbf{V}_i\,\mathbf{M}_W\,\mathbf{M}_V\,\mathbf{M}_P.$$

In this case, both the `WORLD` ($\mathbf{M}_W$) and the `PROJECTION` ($\mathbf{M}_P$) matrices are initialized to the identity matrix. For this reason, the vertices of the squares will only be transformed by $\mathbf{M}_{w2n}$ in the `VIEW` matrix processor. For example, $V_a = (160, 122)$ will become V_{at}:

$$\begin{aligned}
\mathbf{V}_{at} &= \mathbf{V}_a\,\mathbf{M}_{w2n} \\
&= \begin{bmatrix}160 & 122\end{bmatrix}\mathbf{S}(\tfrac{2}{200}, \tfrac{2}{160})\mathbf{T}(-1,-1) \\
&= \begin{bmatrix}\frac{160\times 2}{200} - 1 & \frac{160\times 2}{150} - 1\end{bmatrix} \\
&= \begin{bmatrix}0.6 & 0.525\end{bmatrix}.
\end{aligned}$$

When applying $\mathbf{M}_{w2n}$ to all of the vertices, we get

$$\text{LargeSquare}_{\text{ndc}} = \begin{cases} V_{at} = & (0.6, & 0.525), \\ V_{bt} = & (-0.2, & 0.525), \\ V_{ct} = & (-0.2, & -0.475), \\ V_{dt} = & (0.5, & -0.475), \end{cases} \tag{10.4}$$

and

$$\text{SmallSquare}_{\text{ndc}} = \begin{cases} V_{at} = & (0.6, & 0.525), \\ V_{et} = & (1.1, & 0.525), \\ V_{ft} = & (1.1, & 1.15), \\ V_{gt} = & (0.6, & 1.15). \end{cases} \tag{10.5}$$

ndc. Abbreviation for *normalized device coordinate*. We will study the details of this coordinate system in Section 10.2.

Here we see that Direct3D actually transforms the input vertices into much smaller numbers. It is interesting that we defined the vertices of the squares according to $\text{LargeSquare}_{\text{wc}}$ (Equation (10.1)) and $\text{SmallSquare}_{\text{wc}}$ (Equation (10.2)), only to define the transform operator $\mathbf{M}_{n2w}$ (Equation (10.3)) to ensure that the D3D API transforms these input vertices to $\text{LargeSquare}_{\text{ndc}}$ (Equation (10.4)) and $\text{SmallSquare}_{\text{ndc}}$ (Equation (10.5)). Based on these observations, it is logical to conclude the following.

- **The effects of $\mathbf{M}_{n2w}$.** If we define the input vertices according to $\text{LargeSquare}_{\text{ndc}}$ and $\text{SmallSquare}_{\text{ndc}}$, then there would be no need for the $\mathbf{M}_{w2n}$ operator of Equation (10.3).

Tutorial 10.2.
Project Name
`D3D_ViewTransform1`

Tutorial 10.2. Drawing without the `w2n` Transform

- **Goal.** Verify the effects of the `w2n` matrix.
- **Approach.** Draw the two squares defined by the vertices of $\text{LargeSquare}_{\text{ndc}}$ and $\text{SmallSquare}_{\text{ndc}}$ with identity in the D3D `VIEW` matrix processor.

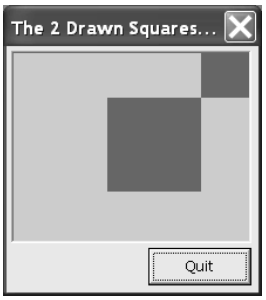

Figure 10.3. Tutorial 10.2.

Figure 10.3 is a screenshot of running Tutorial 10.2. We observe the output to be identical to that of Tutorial 10.1. However, the drawing routines of these two tutorials are different in significant ways. From Listing 10.2, we observe the following differences.

- **Step 2.** We do not compute the `w2n` matrix. Instead, we initialize all three matrix processors to identity and proceed to drawing the squares.

Source file. `GrfxWindowD3D.cpp` file in the *GrfxWindow* folder of the `D3D_ViewTransform1` project.

```
void CGrfxWindowD3D::OnPaint()
{
    ⋮
    Step 2: initialize selected hardware and set the coordinate system parameters.
    ⋮
    // Initialize D3D matrix processors setting all three
    // matrix processors to identity matrix
    m_pD3DDevice->SetTransform(D3DTS_WORLD, &identity);
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &identity);
    m_pD3DDevice->SetTransform(D3DTS_PROJECTION, &identity);

    Step 3: clear drawing buffer and draw two squares.
    ⋮
    // Vertices of the LargeSquare_ndc
    v[0].m_point = D3DXVECTOR3(0.60f,0.525f,0);  // V_at
    v[1].m_point = D3DXVECTOR3(-0.2f,0.525f,0);  // V_bt
    v[2].m_point = D3DXVECTOR3(-0.2f,-0.475f,0); // V_ct
    v[3].m_point = D3DXVECTOR3(0.60f,-0.475f,0); // V_dt
    m_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, (CONST void *)v,...);

    // Vertices of the SmallSquare_ndc
    v[0].m_point = D3DXVECTOR3(0.6f,0.525f,0); // V_at
    v[1].m_point = D3DXVECTOR3(1.1f,0.525f,0); // V_et
    v[2].m_point = D3DXVECTOR3(1.1f,1.15f,0);  // V_ft
    v[3].m_point = D3DXVECTOR3(0.6f,1.15f,0);  // V_gt
    m_pD3DDevice->DrawPrimitiveUP(D3DPT_TRIANGLEFAN, 2, (CONST void *)v,...);
    ⋮
}
```

Listing 10.2. `CGrfxWndD3D::OnPaint()` (Tutorial 10.2)

- **Step 3.** The two squares are specified by vertices defined by $\text{LargeSquare}_{\text{ndc}}$ (Equations (10.4)) and $\text{SmallSquare}_{\text{ndc}}$ (Equations (10.5)).

In this tutorial, we verify our observations that we can reproduce the effects of the $\mathbf{M}_{w2n}$ operator of Equation (10.3). Based on the results of this tutorial, we further observe the following.

- When all three matrix processors in the D3D RC are initialized to be the identity matrix, only vertices between the range of ± 1.0 are displayed.

Tutorial 10.3. Verify the ± 1.0 Drawing Area

Tutorial 10.3.
Project Name
`D3D_ViewTransform2`

- **Goal.** Verify that the application window displays all vertices inside the range of ± 1.0.
- **Approach.** With all matrix processors set to the identity matrix, draw a circle with center located at the origin ($(0,0)$) and radius of 1.0.

Figure 10.4 is a screenshot of running Tutorial 10.3. In this case, the output UI drawing area is defined to be 200 pixels $\times$ 200 pixels. Once again, we initialize all the matrix processors of the D3D API to identity and proceed to draw a unit circle located at the origin. Recall that we approximate a circle with a triangle fan where vertices of the triangles are located on the circumference of the circle. We observe that the circle perfectly fits within the application drawing area. This tutorial verifies that the reason we need the $\mathbf{M}_{w2n}$ transform is that the D3D graphics API automatically transforms all vertices from within the range of

$$\begin{cases} -1.0 \leq x \leq 1.0, \\ -1.0 \leq y \leq 1.0, \end{cases}$$

to the entire application drawing area. In computer graphics, we refer to this square area covered by ± 1 as the normalized space, or normalized device coordinate (NDC).

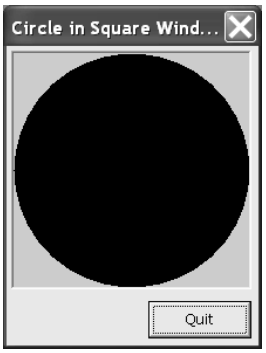

Figure 10.4. Tutorial 10.3: Drawing a circle of radius 1.0 and center at $(0,0)$ with D3D

Tutorials 10.4 and 10.5. Experimenting with the NDC

Tutorial 10.4.
Project Name
`D3D_ViewTransform3`

Tutorial 10.5.
Project Name
`D3D_ViewTransform4`

- **Goal.** Understand that the entire NDC is mapped onto the application drawing area, regardless of the dimensions of the application window.
- **Approach.** Draw the unit circle onto application draw areas with drastically different dimensions and observe the results.

To further understand the transformation performed internally (and automatically) by D3D, in Tutorials 10.4 and 10.5 we define the UI drawing areas to be 100 pixels $\times$ 200 pixels and 200 pixels $\times$ 100 pixels, respectively. In both tutorials, the drawing routines are identical to that of Tutorial 10.3, where the same unit circle with center located at the origin is drawn in each case. Figures 10.5 and 10.6 are screenshots of running Tutorials 10.4 and 10.5. It is interesting that in both cases, just as in the case of Tutorial 10.3, the unit circles fit perfectly within the bounds of the application windows. Of course, in this case, because the application windows are rectangular, the circles are squashed into corresponding ellipses.

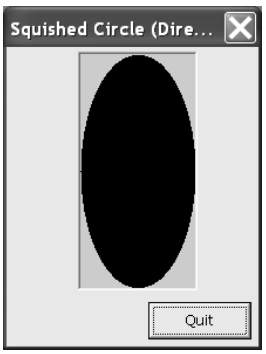

Figure 10.5. Tutorial 10.4: Drawing the same circle onto a 100×200 window.

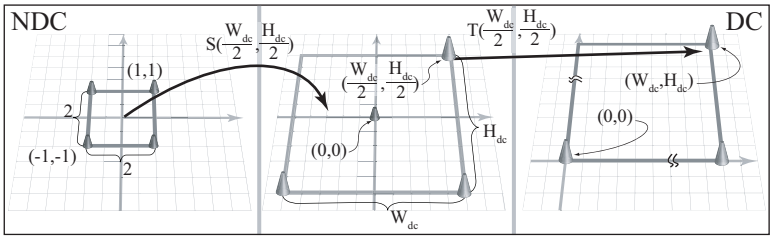

Figure 10.7. D3D's $\mathbf{M}_{n2d}$ operator.

The $\mathbf{M}_{n2d}$ transform. From our discussions, we observe that the D3D API must be performing

$$\mathbf{M}_{n2d} = \mathbf{S}\left(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}\right)\mathbf{T}\left(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}\right) \tag{10.6}$$

on all vertices, where

$$\begin{aligned} W_{dc} &= \text{Width of drawing area (on device),} \\ H_{dc} &= \text{Height of drawing area (on device).} \end{aligned}$$

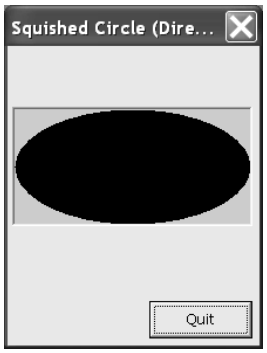

Figure 10.6. Tutorial 10.5: Drawing the circle onto a 200×100 window.

Figure 10.7 illustrates the transformation described by Equation (10.6). On the left diagram we see that the $\mathbf{S}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2})$ operator scales the 2×2 NDC space into an $H_{dc} \times W_{dc}$ region. The center and right diagrams of Figure 10.7 show that the translation operator moves the region to the proper device location. In general, any vertex V_i we specify to the D3D graphics API undergoes the transform

$$\mathbf{V}_{dc} = \mathbf{V}_i\,\mathbf{M}_W\,\mathbf{M}_V\,\mathbf{M}_P\,\mathbf{M}_{n2d},$$

where $\mathbf{M}_W$, $\mathbf{M}_V$, and $\mathbf{M}_P$ are the `WORLD`, `VIEW`, and `PROJECTION` matrix processors of the D3D RC and $\mathbf{V}_{dc}$ is the vertex on the UI drawing area. A very important lesson we have learned so far is that whereas the matrix processors ($\mathbf{M}_W$, $\mathbf{M}_V$, and $\mathbf{M}_P$) are under our program's control, $\mathbf{M}_{n2d}$ will be applied internally by the D3D graphics API automatically and is not under our program's control. Another important observation is that the graphics API (D3D) knows what the underlying display device resolution is (width/height) and computes $\mathbf{M}_{n2d}$ accordingly.

We see the that the $\mathbf{M}_{w2n}$ operator we construct in Step 2 of Listing 10.1 for Tutorial 10.1 (and for every single tutorial we have worked with so far) is to complement the $\mathbf{M}_{n2d}$ transform (Equation (10.6)) that D3D performs automatically. An obvious question is, "Why would D3D automatically perform the $\mathbf{M}_{n2d}$ operation?" To answer this question, we must first understand coordinate systems.

Linearity of affine transformation. Before we leave this section, notice that we analyzed the $\mathbf{M}_{w2n}$ operator from Equation (10.3) based on transforming the

$(0,0)$ to $(200,160)$ rectangular area to the area within the ± 1 range (NDC) (Figure 10.2). This same operator also proportionally transforms the two squares inside the rectangular area where the transformed squares are proportionally located inside the NDC space. This is an example of the linear property of affine transformation: if the transform operator works for a rectangle, then all geometric contents inside the rectangle will also transform appropriately. For this reason, when deriving coordinate transformations, we only need to consider the operator that transforms the enclosing rectangular region of interest.

10.2 Device and Normalized Coordinate Systems

In Section 3.1 when we wanted to describe vertex positions of the squares, we borrowed concepts from the Cartesian coordinate system with the horizontal and vertical axes and units on the axis. From the discussion in the previous section, we see two examples of applying the concepts associated with the Cartesian coordinate system.

Coordinate system and space. Coordinate *systems* and coordinate *spaces* are used interchangeably in this book. For example, device coordinate *system* and device coordinate *space* are both referred to as the DC.

1. **Device coordinate (DC).** When we draw and refer to positions on the application window, implicitly, we assume a coordinate system. We assume that the origin is located at the lower-left corner of the window, with units being pixels. Note that the DC is a variable coordinate system, where it can be changed even during the lifetime of an application (e.g., by resizing the application window size). The DC has dimension width (W_{dc}) by height (H_{dc}). The application's drawing area is the DC space.

2. **Normalized (device) coordinate (NDC).** With center at the origin and *x*/*y* ranges between -1 and $+1$, the NDC defines a 2×2 square area. This is the internal coordinate system of the D3D graphics API. We have seen that, as programmers working with D3D, we are responsible for programming the matrix processors such that all vertices are transformed into the NDC (i.e., the $\mathbf{M}_{w2n}$ operator). In turn, D3D will automatically transform vertices from NDC to DC when processing the vertices (i.e., the $\mathbf{M}_{n2d}$). The NDC never changes.

NDC and OpenGL. For similar reasons as discussed here, the OpenGL API also defines the NDC as its internal coordinate system. The OpenGL API also performs the exact same $\mathbf{M}_{n2d}$ operator (as defined by Equation (10.6)) on every input vertices.

Although the internal NDC representation causes extra complexity and processing, the NDC representation is also very important for the following reasons.

- **Consistency and flexibility.** A well-defined constant coordinate system is important for the internal implementation of the D3D API. As programmers of the API, such a well-known coordinate system provides a fixed reference

as the rest of the application changes: as the DC window size changes, we can continue to communicate to the D3D API based on the NDC. In this way, our solution can be completely independent of the size of the UI drawing area. Because our solution is designed with reference to the NDC, our program can run in the same way with a 200×200 or a 500×500 UI window.

- **Convenience.** With the strategically chosen center (origin) and the coordinate ranges (of ± 1), it is straightforward to transform the NDC square to other rectangular regions. For example, we have already seen in Equation (10.6) that it takes simple scaling and translation operations to transform the NDC to DC. In general, it is convenient to transform NDC to any coordinate space, for example, to the coordinate space defined for paper on printouts.

However, for humans, the ± 1 range of NDC is not always intuitive and often inconvenient to work with. For example, it is not straightforward to design a geometric face (e.g., Figure 9.21) where all vertex information must be constraint to between -1 and 1. To compensate for this rigid constraint, we introduce the *world coordinate* space for our programs to work in.

10.3 The World Coordinate System

When designing the geometric face of Figure 9.21, the implicit unit of measurement was the pixel. The implementation of Tutorial 9.9 conformed to this assumption where, for example, the drawing area is exactly 300 pixels $\times$ 300 pixels. It would seem that if we want to display this face design on an application window with a different dimension, we would have to re-measure and re-define all vertex positions. However, if we examine the implementation of Tutorial 9.9 more closely, in the `CDrawOnlyHandler::DrawGraphics()` function, the `w2n` matrix is the $\mathbf{M}_{w2n}$ operator:

$$\mathbf{M}_{w2n} = \mathbf{S}\left(\frac{2}{width}, \frac{2}{height}\right)\mathbf{T}(-1,-1),$$

where

$$\begin{cases} \text{width} = & 300 \text{ pixels}, \\ \text{height} = & 300 \text{ pixels}. \end{cases}$$

```
void CDrawOnlyHandler::DrawGraphics()
    float width = 300.0f, height=300.0f;
    ⋮
    // Construct the matrix that will transform from the world bounds
    // to NDC
    D3DXVECTOR3 scale(2.0f/width, 2.0f/height, 1.0f);
    D3DXVECTOR3 translate(-1.0f, -1.0f, 0.0);
    D3DXMatrixTransformation(&world2ndc, ... &scale, ... &translate);
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &world2ndc);
    ⋮
```

Source file. `DrawOnlyHandler.cpp` file in the *WindowHandler* folder of the `D3D_XformList` project.

Listing 10.3. The `DrawGraphics()` function of Tutorial 9.9.

Figure 10.8 illustrates the transformation that takes place. Note the following.

- **Step 1.** The $\mathbf{M}_{w2n}$ operator transforms all vertices from our *design space* into the NDC. In this case, the design and measurements are based on a drawing area of width = 300 and height = 300. In the implementation of Tutorial 9.9, the values for the width and height are fixed based on the width and height of the application window. However, notice that the input of the $\mathbf{M}_{w2n}$ transformation is our design space and that the output is the NDC space. In fact, neither the input nor the output is related to the DC dimension!

- **Step 2.** The $\mathbf{M}_{n2d}$ operator is internal to the D3D graphics API. This operator transforms all vertices from the NDC to the final application drawing area, or the device coordinate (DC). Notice that this transformation, $\mathbf{M}_{n2d}$,

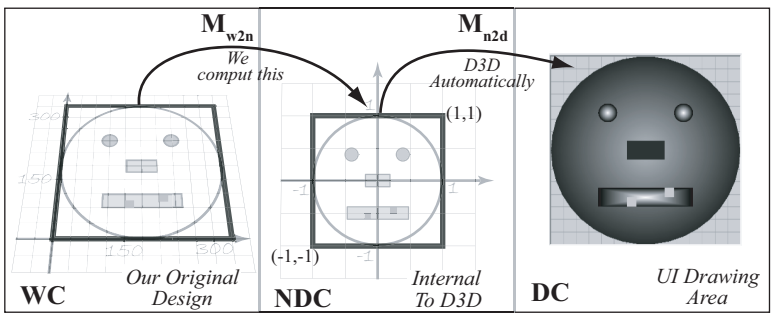

Figure 10.8. Transformation for drawing of the face in Figure 9.21.

is governed by the dimensions of the application window and is independent of the values we used for the $\mathbf{M}_{w2n}$ operator.

These observations indicate that, in fact, our 300×300 design space need not be related to the application window size. Indeed, the D3D internal NDC space helped separate $\mathbf{M}_{w2n}$ and $\mathbf{M}_{n2d}$ into independent operators, where

$$\text{Design space} \xrightarrow{\mathbf{M}_{w2n}} \text{NDC} \xrightarrow{\mathbf{M}_{n2d}} \text{DC}.$$

As application programmers, we only need to be concerned with the $\mathbf{M}_{w2n}$ operator, which is independent of the DC!

10.3.1 Design Space versus Drawing Area

Tutorial 10.6. DC-Independent Design Space

Tutorial 10.6.
Project Name:

`D3D_XformListCoordSpace`
Library Support:
`UWB_MFC_Lib1`
`UWB_D3D_Lib9`

- **Goal.** Verify that the 300×300 design space of Figure 9.21 is indeed independent of the dimensions of the UI drawing area.
- **Approach.** Change the dimension of the UI drawing area to 500×500 and observe the output.

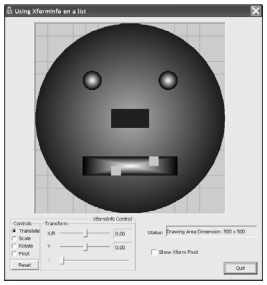

Figure 10.9. Running Tutorial 10.6.

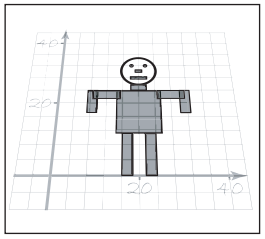

Figure 10.10. A full-figure geometric person.

Figure 10.9 is a screenshot of running Tutorial 10.6. Although the UI front end of this tutorial appears to be significantly different from that of Tutorial 9.9, these two tutorials have identical back-end implementations! The only significant difference is that in this case, the UI drawing area is 500×500 pixels. From the output of this tutorial, we see a larger but the same geometric face as the one observed in Tutorial 9.9. In this tutorial, we display a 300×300 design space in a 500×500 drawing area; we have verified that our design space is indeed independent of the device dimensions.

In computer graphics, we refer to the 300×300 coordinate space where we designed the original face the world coordinate space or the world coordinate system (WC). The world refers to the fact that the geometric objects within this space are the world that we would like to draw onto the output display area. We observe that as long as we correctly construct the $\mathbf{M}_{w2n}$ operator (or the WC-to-NDC operator), we can select to work in any convenient WC space.

10.3.2 Working with a Convenient WC Space

It is important to select a design space or the WC such that it is convenient to specify our geometric objects. For example, in anticipation of specifying the full-

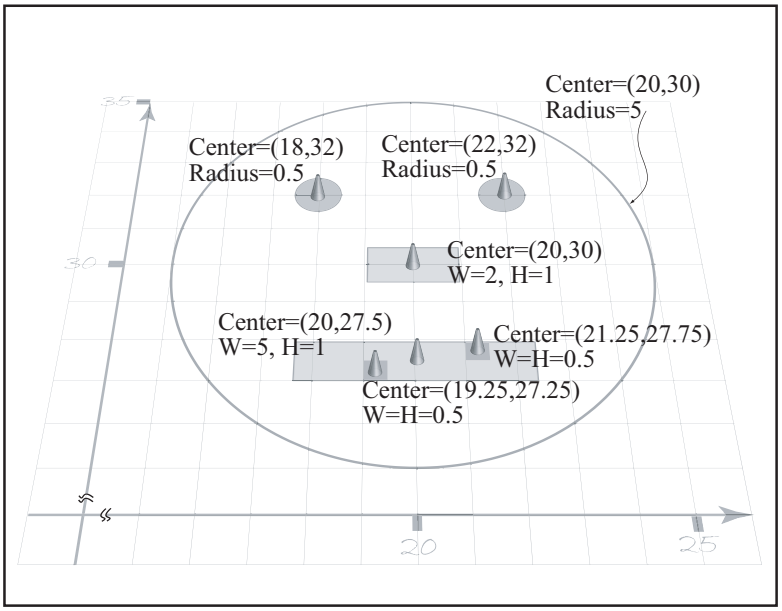

Figure 10.11. WC window of the face from Figure 10.10.

figure geometric person of Figure 10.10, we can choose a more convenient WC space to specify the face of Figure 9.21. Figure 10.11 shows a Window into the WC to illustrate the details of the new geometric face. In this case, the rectangular WC window is bounded by

$$\text{WC window} = \begin{cases} 15 \leq x \leq 25 \\ 25 \leq y \leq 35 \end{cases} = \begin{cases} \text{center} = (cx_{wc}, cy_{wc}) = (20, 30), \\ \text{width} = W_{wc} = 10, \\ \text{height} = H_{wc} = 10. \end{cases}$$

If we want to display the content of this WC window, we must construct an appropriate $\mathbf{M}_{w2n}$ operator. As we saw in Figure 10.8, as programmers of the D3D graphics API, our goal is to construct the $\mathbf{M}_{w2n}$ operator to transform the WC window into D3D's internal coordinate system, i.e., the NDC. In turn, D3D will automatically transform the content of the NDC to the drawing area on the application window.

Linearity of affine transformation. Recall that to transform geometric contents between rectangular regions, we can concentrate on constructing the operator that transforms between the rectangles that surround the regions. The contents inside the rectangles will transform proportionally.

Figure 10.12 illustrates one way to construct the $\mathbf{M}_{w2n}$ operator, where we first move the center of the region to the origin and then scale the region into a 2×2 area, or

$$\mathbf{M}_{w2n} = \mathbf{T}(-20, -30)\mathbf{S}\left(\frac{2}{10}, \frac{2}{10}\right). \tag{10.7}$$

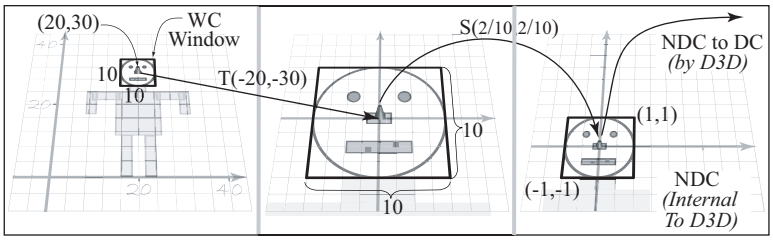

Figure 10.12. $\mathbf{M}_{w2n}$: world-to-NDC transform for the face of Figure 10.11.

Tutoral 10.7. Working with a Convenient WC Space (Figure 10.12)

Tutorial 10.7.
Project Name:
`D3D_ChangedCoordinate`
Library Support:
UWB_MFC_Lib1
UWB_D3D_Lib9

- **Goal.** Understand and work with a world coordinate (WC) window that is *not* centered on the origin.
- **Approach.** Program the transformation of Figure 10.12 and observe the output.

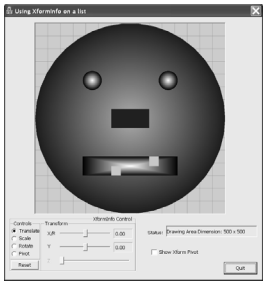

Figure 10.13. Tutorial 10.7.

Figure 10.13 is a screenshot of running Tutorial 10.7. We observe that the output of this tutorial is identical to that of Tutorial 10.6. However, this tutorial implements the geometric face as defined by Figure 10.11 (in `CModel.cpp`). The `DrawGraphics()` function shown in Listing 10.4 defines the $\mathbf{M}_{w2n}$ transform operator of Equation 10.7. We see that although the world coordinate spaces and the actual geometric values specified in this tutorial are very different from that of Tutorial 10.6, the output from the two tutorials are identical to each other. This tutorial verifies that we can define the WC window to support a convenient WC space for designing our world and that the WC space is independent of the dimensions of the UI drawing device.

Source file.
`DrawOnlyHandler.cpp` file in the *WindowHandler* folder of the `D3D_ChangedCoordinates` project.

```
void CDrawOnlyHandler::DrawGraphics()
    ⋮
    // Translate the center of WC to the origin
    D3DXMatrixTranslation(&toPlace, -20.0f, -30.0f, 0.0);
    // Scale to 2x2
    D3DXMatrixScaling(&toSize, 2.0f/10.0f, 2.0f/10.0f, 1.0);
    // build Mw2n operator of Equation 10.7
    D3DXMatrixMultiply(&world2ndc, &toPlace, &toSize);
    // Load to the VIEW matrix (MV)
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &world2ndc);
```

Listing 10.4. The `DrawGraphics()` function of Tutorial 10.7.

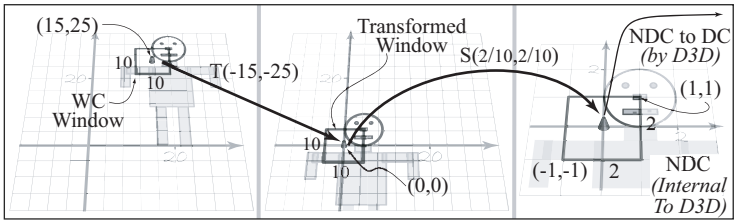

Figure 10.14. Location of WC window as defined by Equations (10.8) and (10.9).

Notice that there are no physical boundaries around the WC window. As programmers, we choose the WC window and program the $\mathbf{M}_{w2n}$ operator accordingly. For example, in Tutorial 10.7, we could easily choose to select a different WC window:

$$\text{WC window} \begin{cases} \text{center} = (15,25), \\ \text{width} = \textit{height} = 10, \end{cases} \tag{10.8}$$

and define the $\mathbf{M}_{w2n}$ to be

$$\mathbf{M}_{w2n} = \mathbf{T}(-15,-25)\mathbf{S}\left(\frac{2}{10},\frac{2}{10}\right). \tag{10.9}$$

Tutorial 10.8.
Project Name:
`D3D_TranslateWC`
Library Support:
UWB_MFC_Lib1
UWB_D3D_Lib9

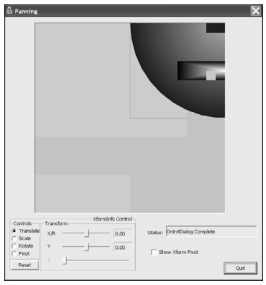

Figure 10.15. Running Tutorial 10.8.

Tutorial 10.8. Translating the WC Window (Figure 10.14)

- **Goal.** Verify our understanding that we can move the WC window in the WC system to show a different part of the WC system.
- **Approach.** Program Equation (10.9) to verify our understanding.

Figure 10.15 is a screenshot of running Tutorial 10.8. The implementation of this tutorial is identical to that of Tutorial 10.7, except we program the $\mathbf{M}_{w2n}$ according Equation (10.9) as shown in Listing 10.5. The image output of the the program does reflect our predictions from Figure 10.14. In this case, we can see the greyish outline of the rest of the geometric person of Figure 10.10. If we examine the class definition in Listing 10.6, at label A we observe that the `CModel` class defines the geometries for the entire human of Figure 10.10 in the `m_figure PrimitiveList` object. At label B, we see that the `CModel::Draw()` function draws all the geometries that are defined in the `m_figure` object. However, from the output of this tutorial in Figure 10.15, we see that only the geometries inside the WC window are displayed in the UI drawing area. From this tutorial, we see that the graphics API clips away all the geometries outside of the NDC ± 1 range.

Clipping. Graphics APIs only process and display geometries inside the ± 1 range of the NDC space. Geometries outside of this range are clipped and/or otherwise guaranteed to not show up in the UI drawing area.

Source file. `DrawOnlyHandler.cpp` file in the *WindowHandler* folder of the `D3D_TranslateWC` project.

```
void CDrawOnlyHandler::DrawGraphics()
    ⋮
    // Translate the center of WC to the origin
    D3DXMatrixTranslation(&toPlace, -15.0f, -25.0f, 0.0);
    // Scale to 2x2
    D3DXMatrixScaling(&toSize, 2.0f/10.0f, 2.0f/10.0f, 1.0);
    // build M_w2n operator of Equation 10.9
    D3DXMatrixMultiply(&world2ndc, &toPlace, &toSize);
    // Load to the VIEW matrix (M_V)
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &world2ndc);
```

Listing 10.5. The `DrawGraphics()` function of Tutorial 10.8.

Source file. `Model.cpp` file in the *Model* folder of the `D3D_TranslateWC` project.

A:
```
CModel::CModel()  { // CModel class constructor
    ⋮
    // geometry for the torso
    UWB_PrimitiveRectangle* pTorso = new UWB_PrimitiveRectangle();
    pTorso->SetCorners( vec3(15,11,0), vec3(25,24,0) );
    ⋮
    // geometry for the left leg
    UWB_PrimitiveRectangle* pLeftLeg = new UWB_PrimitiveRectangle();
    ⋮
    // defines the entire geometry for the simple person of Figure 10.10
    ⋮
    // m_figure is a PrimitiveList
    m_figure.Append(pTorso);  m_figure.Append(pLeftLeg);
    ⋮
    // append all defined goemetry into the m_figure PrimitiveList
}
```

B:
```
void CModel::DrawModel()  // CModel draw function
    ⋮
    // Set up the WORLD matrix
    m_xform.SetUpModelStack( m_DrawHelper );
    // draws the entire goemetric human of Figure 10.10
    m_figure.Draw( lod, m_DrawHelper );
    ⋮
```

Listing 10.6. The `CModel` class of Tutorial 10.8.

10.4 The World Coordinate Window

In order to properly support WC design space, in our programs we must identify a rectangular region of interest, or the *window* inside the WC:

$$\text{WC window} = \begin{cases} \text{center} & = (cx_{wc}, cy_{wc}), \\ \text{width} & = W_{wc}, \\ \text{height} & = H_{wc}. \end{cases}$$

We must then construct a corresponding $\mathbf{M}_{w2n}$ to transform this region to the NDC space. In general, we transform the center of the region to the origin ($T(-cx_{wc}, -cy_{wc})$) and then scale the width and height of the region to the NDC 2×2 square ($S(\frac{2}{W_{wc}}, \frac{2}{H_{wc}})$):

$$\mathbf{M}_{w2n} = \mathbf{T}(-cx_{wc}, -cy_{wc})\mathbf{S}(\frac{2}{W_{wc}}, \frac{2}{H_{wc}}). \tag{10.10}$$

10.4.1 Vertices in Different Coordinate Spaces

As programmers working with graphics APIs, we specify vertices in the WC, or $\mathbf{V}_{wc}$. As illustrated in Figure 10.16, this vertex undergoes different transforms until it reaches the DC, or $\mathbf{V}_{dc}$, before being displayed on the output drawing area:

$$\mathbf{V}_{wc} \xrightarrow{\mathbf{M}_{w2n}} \mathbf{V}_{ndc} \xrightarrow{\mathbf{M}_{n2d}} \mathbf{V}_{dc},$$

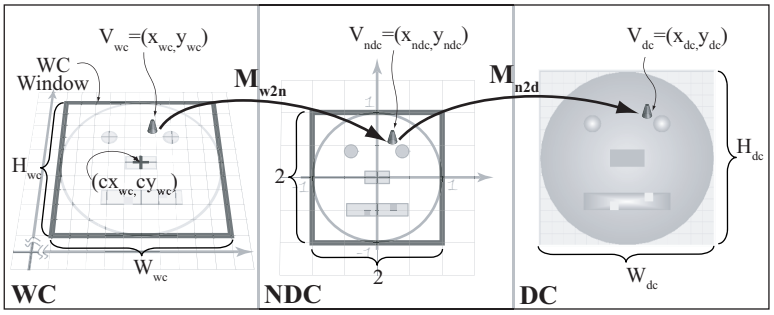

Figure 10.16. Transformation of a vertex between different coordinate systems.

where

$$\begin{aligned}\mathbf{V}_{wc} &= \begin{bmatrix} x_{wc} & y_{wc} \end{bmatrix}, \\ \mathbf{V}_{ndc} &= \mathbf{V}_{wc}\,\mathbf{M}_{w2n}, \\ \mathbf{V}_{dc} &= \mathbf{V}_{ndc}\,\mathbf{M}_{n2d},\end{aligned}$$

or

$$\begin{aligned}\mathbf{V}_{dc} &= \mathbf{V}_{ndc}\,\mathbf{M}_{n2d} \\ &= \mathbf{V}_{wc}\,\mathbf{M}_{w2n}\,\mathbf{M}_{n2d}.\end{aligned}$$

If we let

$$\mathbf{M}_{w2d} = \mathbf{M}_{w2n}\,\mathbf{M}_{n2d}, \tag{10.11}$$

then

$$\mathbf{V}_{dc} = \mathbf{V}_{wc}\,\mathbf{M}_{w2d}. \tag{10.12}$$

Recall that $\mathbf{M}_{n2d}$ is defined by Equation (10.6) and that $\mathbf{M}_{w2n}$ is defined by Equation (10.10) (re-listing the two equations):

$$\mathbf{M}_{n2d} = \mathbf{S}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2})\mathbf{T}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}), \tag{10.6}$$

$$\mathbf{M}_{w2n} = \mathbf{T}(-cx_{wc}, -cy_{wc})\mathbf{S}(\frac{2}{W_{wc}}, \frac{2}{H_{wc}}). \tag{10.10}$$

Consecutive scaling operators. Two consecutive scaling operators:

$$S(s_{x1}, s_{y1})S(s_{x2}, s_{y2})$$

have the same effect as scaling once by the combined effect of the scaling factors:

$$S(s_{x1}s_{x2}, s_{y1}s_{y2}).$$

In this way,

$$\begin{aligned}\mathbf{V}_{dc} &= \mathbf{V}_{wc}\,\mathbf{M}_{w2d} \\ &= \mathbf{V}_{wc}\,\mathbf{M}_{w2n}\,\mathbf{M}_{n2d} \\ &= \mathbf{V}_{wc}\,\mathbf{T}(-cx_{wc}, -cy_{wc})\,\mathbf{S}(\frac{2}{W_{wc}}, \frac{2}{H_{wc}})\,\mathbf{S}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2})\,\mathbf{T}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}),\end{aligned}$$

or

$$\mathbf{V}_{dc} = \mathbf{V}_{wc}\,\mathbf{T}(-cx_{wc}, -cy_{wc})\,\mathbf{S}(\frac{W_{dc}}{W_{wc}}, \frac{H_{dc}}{H_{wc}})\,\mathbf{T}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}). \tag{10.13}$$

Comparing Equations (10.12) and (10.13), we see that

$$\mathbf{M}_{w2d} = \mathbf{T}(-cx_{wc}, -cy_{wc})\mathbf{S}(\frac{W_{dc}}{W_{wc}}, \frac{H_{dc}}{H_{wc}})\mathbf{T}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}). \tag{10.14}$$

Equation (10.14) says that the operator $\mathbf{M}_{w2d}$, which transforms from the WC (x_{wc}, y_{wc}) to the DC (x_{dc}, y_{dc}), does the following.

- **Move.** The center of the WC window to the origin with $(\mathbf{T}(-cx_{wc}, -cy_{wc}))$. The result of this transform is a $W_{wc} \times H_{wc}$ rectangle centered at the origin.
- **Scale.** With the WC window width of W_{wc}, the scaling factor $\frac{W_{dc}}{W_{wc}}$ changes the width to W_{dc}. In a similar fashion, the height becomes H_{dc}. After the scaling operator, we have a $W_{dc} \times H_{dc}$ rectangle centered at the origin.

- **Move.** The rectangle centered at the origin has half its width/height on either side of the *y*/*x* axis. The translation of $\mathbf{T}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2})$ moves the lower-left corner of the rectangle to the origin, with the upper-right corner located at (W_{dc}, H_{dc}). This is the definition of the DC space.

If we expand the operators in Equation (10.13), then, to transform a point (x_{wc}, y_{wc}) from our design space (WC) to a point (x_{dc}, y_{dc}) on the device drawing area (DC),

$$\begin{aligned} x_{dc} &= ((x_{wc} - cx_{wc})\frac{W_{dc}}{W_{wc}}) + \frac{W_{dc}}{2}, \\ y_{dc} &= ((y_{wc} - cy_{wc})\frac{H_{dc}}{H_{wc}}) + \frac{H_{dc}}{2}, \end{aligned} \tag{10.15}$$

where

$$\text{Device drawing area} \begin{cases} \text{width} & = W_{dc}, \\ \text{height} & = H_{dc}, \end{cases}$$

and

$$\text{WC window} = \begin{cases} \text{center} & = (cx_{wc}, cy_{wc}), \\ \text{width} & = W_{wc}, \\ \text{height} & = H_{wc}. \end{cases}$$

From Equation (10.15), we see that when the size of the device drawing remains constant (i.e., W_{dc} and H_{dc} do not change), then the transformation from WC to DC is governed by the parameters of the WC window as follows.

1. **Center (cx_{wc}, cy_{wc}).** Defines the location of the WC window. Intuitively, by changing the center we are moving the WC window and thus should observe different rectangular regions in the WC system, or panning of the view. Tutorial 10.10 will examine panning in detail.

2. **Dimension (W_{wc} H_{wc}).** Defines the size of the WC window. Intuitively, by changing the dimension, we increase/decrease the rectangular region to be displayed. With a fixed-size UI drawing device, increasing the size of the WC window means showing a larger amount of the WC system in the fixed-size DC drawing area, or a zooming-out effect. With the same logic, decreasing the size of the WC window creates a zooming-in effect. Tutorial 10.11 will examine zooming in detail.

3. **Ratio of scaling factors ($\frac{W_{dc}}{W_{wc}}$ versus $\frac{H_{dc}}{H_{wc}}$).** We scale the width of the WC window by $\frac{W_{dc}}{W_{wc}}$ and the height by $\frac{H_{dc}}{H_{wc}}$. When these two scaling factors are different, the proportion of the results in DC space will also be different from that of the original WC space. For example, a square will be transformed into a rectangle. Section 10.4.4 will examine this effect in detail.

We remind ourselves that Equation (10.15), or the $\mathbf{M}_{w2d}$ operator, is the *net* transformation that would be applied to vertices specified in WC space. From Equation (10.11), we observe that, as programmers, we are only responsible for $\mathbf{M}_{w2n}$, or half of the $\mathbf{M}_{w2d}$ operator. The other half of the operator, $\mathbf{M}_{n2d}$, is computed automatically by the graphics API. In this case, we analyze Equation (10.15) and the $\mathbf{M}_{w2d}$ operator to understand the details of the images generated in the UI drawing device. We will examine the effects of changing the WC window based on tutorial implementations. Let us first extend our library to support working with the WC window.

UWBGL_D3D_Lib10

Change summary. See p. 522 for a summary of changes to the library.

This library extends from `Lib9` by extending the `UWB_WindowHandler` class to support the definition of the WC window and programming of the $\mathbf{M}_{w2n}$ operator. Recall that the `WindowHandler` object is defined to abstract a view/controller pair. For the `WindowHandler` object to properly display different regions of the

Source file. `uwbgl_WindowHandler3.h` file in the *Common Files/WindowHandler* subfolder of the `UWBGL_D3D_Lib10` project.

```
class UWB_WindowHandler : public UWB_IWindowHandler {
    ⋮
    // Set/Get methods for accessing the m_WCWindow object
A:  virtual void SetWCWindow(const UWB_BoundingBox &window);
    virtual const UWB_BoundingBox* GetWCWindow();
    // Set/Get methods for the drawing device (via m_hAttachedWindow)
B:  virtual void SetDeviceSize(int width, int height);
    virtual void GetDeviceSize(int &width, int &height);
    // wcPt -> dcPt (Equation 10.13)
C:  virtual void WorldToDevice(float wcX wcY, int &dcX &dcY);
    // dcPt -> wcPt (Equation 10.19)
    virtual void DeviceToWorld(int dcX dcY, float &wcX &wcY);
    // Drawing the WCWindow
D:  virtual void DrawWCWindow(UWB_DrawHelper&);
    // Compute M_w2n and load to API matrix processor
E:  virtual void LoadW2NDCXform() = 0;
  protected:
    ⋮
    HWND m_hAttachedWindow; // The UI drawing device (a MFC Window)
F:  UWB_BoundingBox m_WCWindow; // Window for the World Coordinate
};
```

Listing 10.7. The `WindowHandler` class of `UWBGL_D3D_Lib10`.

```
void UWB_WindowHandler::WorldToDevice(float wcx, float wcy, int &dcx, int &dcy) const {
    ⋮
    GetDeviceSize(dcW, dcH); // dcW/dcH are the width/height of the drawing device
    center = m_WCWindow.GetCenter(); // center of the WCWindow
    // Equation 10.13
    dcx = (m_WCWindow.Width() / 2.0f) + ((dcW/m_WCWindow.Width())  * (-center.x + wcx));
    dcy = (m_WCWindow.Height()/ 2.0f) + ((dcH/m_WCWindow.Height()) * (-center.y + wcy));

void UWB_WindowHandler::DeviceToWorld(int dcx, int dcy, float &wcx, float &wcy) const
    ⋮
    GetDeviceSize(dcW, dcH); // dcW/dcH are the width/height of the drawing device
    vec3 center = m_WCWindow.GetCenter(); // center of the WCWindow
    // Equation 10.19
    wcx = center.x + ((m_WCWindow.Width()/dcW)  * (dcx - (dcW/2.0f)));
    wcy = center.y + ((m_WCWindow.Height()/dcH) * (dcy - (dcH/2.0f)));
```

Source file. `uwbgl_WindowHandler3.cpp` file in the *Common Files/WindowHandler* subfolder of the `UWBGL_D3D_Lib10` project.

Listing 10.8. The `WindowHandler` transform and draw functions.

```
class UWBD3D_WindowHandler : public UWB_WindowHandler {
    ⋮
    virtual void LoadW2NDCXform() const; // Computes and loads VIEW matrix with M_w2n
    ⋮
```

```
void UWBD3D_WindowHandler::LoadW2NDCXform() const
    // center of the m_WCWindow (cx_wc, cy_wc)
    vec3 center = m_WCWindow.GetCenter();
    // T(-cx_wc, -cy_wc)
    D3DXMatrixTranslation(&toPlace, -center.x, -center.y);
    // S(2/W_wc, 2/H_wc)
    D3DXMatrixScaling(&toSize, 2.0f/m_WCWindow.Width(), 2.0f/m_WCWindow.Height());
    // M_w2n = T(-cx_wc, -cy_wc)S(2/W_wc, 2/H_wc)
    D3DXMatrixMultiply(&world2ndc, &toPlace, &toSize);
    // M_V ← M_w2n
    m_pD3DDevice->SetTransform(D3DTS_VIEW, &world2ndc);
    ⋮
```

Source file. `uwbgl_D3DWindowHandler4.h/cpp` files in the *D3D Files/ WindowHandler* folder of the `UWBGL_D3D_Lib10` project.

Listing 10.9. The `LoadW2NDCXform()` functions.

model, it must support the WC window and the associated transformations. From Listing 10.7, we see that, at label F, we define a `UWB_BoundingBox` object to represent the WC window. The get/set access functions for the `m_WCWindow` are defined at label A. Because the UI drawing area (drawing device) is supported by the MFC GUI API (`m_hAttachedWindow`), the device get/set functions at label B are implemented by interacting with the `m_hAttachedWindow` object. The two functions at label C implements the WC-to-DC transformations (more details to follow). The drawing function at label D allows us to visualize the WC window as a wire-framed rectangle. The function at label E should compute the $\mathbf{M}_{w2n}$ matrix and load the graphics API with this matrix. Note that this function is a pure virtual function; since `WindowHandler` is a graphics API–independent class, we do not know how to implement this function. This function will be implemented by the `D3DWindowHandler` class. Listing 10.8 shows the implementations of the transformation functions. We see that in both cases, the functions are faithful implementations of equations we have derived. We will derive Equation (10.19) in Section 10.5 and discuss how to work with these functions to handle mouse inputs. Listing 10.9 shows the implementation of the `LoadW2NDCXform()` function in the `D3D_WindowHandler` class. From the listing, we observe step-by-step implementation of Equation (10.10). The computed $\mathbf{M}_{w2n}$ operator is loaded into the `VIEW` ($\mathbf{M}_V$) matrix processor of the D3D API.

Tutorial 10.9. Working with `UWBGL_D3D_Lib10`

Tutorial 10.9.
Project Name:
`D3D_WCSupport`
Library Support:
`UWB_MFC_Lib1`
`UWB_D3D_Lib10`

- **Goal.** Demonstrate how to work with the new `UWBGL_D3D_Lib10` library.
- **Approach.** Re-implement Tutorial 10.8 based on the new library to understand how to work with the new functions.

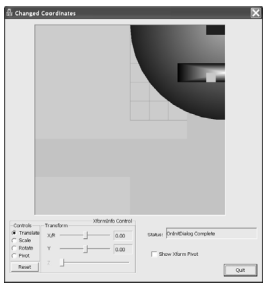

Figure 10.17. Running Tutorial 10.9.

Figure 10.17 is a screenshot of running Tutorial 10.9. This tutorial is identical to Tutorial 10.8 except that the new implementation takes advantage of the new WC window functionality provided by the `UWBGL_D3D_Lib10` library. Listing 10.10 shows that, in the `CTutorialDlg::OnInitDialog()` function, at label A we initialize the view/controller pair WC window (`m_view`) as part of the application state initialization process. When redrawing the view/controller pair in the `DrawGraphics()` function, at label B we call the `LoadW2NDCXform()` function to compute and load the $\mathbf{M}_{w2n}$ operator into the D3D `VIEW` matrix processor.

```
BOOL CTutorialDlg::OnInitDialog()
    ⋮
    if(!m_view.Initialize(*this, IDC_PLACEHOLDER)) // m_view is a DrawOnlyHandler (D3DWindowHandler)
        return FALSE;
    UWB_BoundingBox wcWindow(vec3(10, 20),vec3(20, 30)); // initialize the wcWindow
    // set the m_WCWindow of WindowHandler object
A:  m_view.SetWCWindow( wcWindow );
    ⋮
```

Source file. `TutorialDlg.cpp` file in the *Source Files* folder of the `D3D_WCSupport` project.

```
void CDrawOnlyHandler::DrawGraphics()
    ⋮
    BeginDraw();
B:  LoadW2NDCXform();   // compute and load the M_w2n to the VIEW matrix processor
    m_pD3DDevice->Clear( ⋮ ); // Clears the device for drawing
    theApp.GetModel().DrawModel(); // Tells the Model to draw itself
    EndDrawAndShow();
    ⋮
```

Listing 10.10. Working with `UWBGL_D3D_Lib10` (Tutorial 10.9).

10.4.2 WC Window Position: Panning

Recall that Equation (10.15) defines the transformation of a point from WC space (x_{wc}, y_{wc}) to DC space (x_{dc}, y_{dc}):

$$\begin{aligned} x_{dc} &= ((x_{wc} - cx_{wc})\tfrac{W_{dc}}{W_{wc}}) + \tfrac{W_{dc}}{2}, \\ y_{dc} &= ((y_{wc} - cy_{wc})\tfrac{H_{dc}}{H_{wc}}) + \tfrac{H_{dc}}{2}, \end{aligned} \quad (10.15)$$

where $W_{dc} \times H_{dc}$ is the drawing device dimension and

$$\text{WC window} = \begin{cases} \text{center} &= (cx_{wc}, cy_{wc}), \\ \text{width} &= W_{wc}, \\ \text{height} &= H_{wc}. \end{cases} .$$

In Tutorial 10.8, we observed that by changing the WC center position (cx_{wc}, cy_{wc}), we can show different regions of the model defined in the WC system. Based on our discussions, we can predict that a continuous changing of WC window position would create an effect of panning through the WC system.

Tutorial 10.10.
Project Name:
`D3D_Panning`
Library Support:
`UWB_MFC_Lib1`
`UWB_D3D_Lib10`

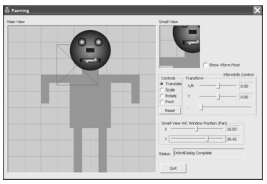

Figure 10.18. Running Tutorial 10.10.

Tutorial 10.10. Moving the WC Window Center ((cx_{wc}, cy_{wc}))

- **Goal.** Verify that the effect of changing the center position of a WC window does indeed correspond to continuous displaying of different regions in the WC system.
- **Approach.** Allow the user to interactively change the center position of a WC window and examine the results.

Figure 10.18 is a screenshot of running Tutorial 10.10. In this tutorial, the large (main) view displays a larger region of the WC system (the entire geometric person), while the small view only displays part of the WC system visible in the main view. The red wire-framed rectangle in the main view represents the WC window of the small view. The two slider bars on the lower-right of the application window control the center position ((cx_{wc}, cy_{wc})) of the WC window for the small view. By changing these two slider bars, we can observe the red rectangle in the main view and the image showing in the small view pan across the main view. This tutorial verifies that changing the WC window position creates the panning effect. Figure 10.19 illustrates that the implementation of Tutorial 10.10 involves two different types of `CWindowHandler` objects. The top-center rectangle represents the `CDrawOnlyHandler` of the small view, whereas the bottom-center rectangle is the `CMainHandler` of the main view. As we can see, the `CMainHandler` maintains a reference to the small view. This reference provides the small view WC window information for the main view to draw the wire-framed red rectangle. With the `UWBGL_D3D_Lib10` support, both of the `Handler` objects have an instance of `UWB_BoundingBox` representing their corresponding WC window. As in the case of Listing 10.10, during `DrawGraphics()`, each

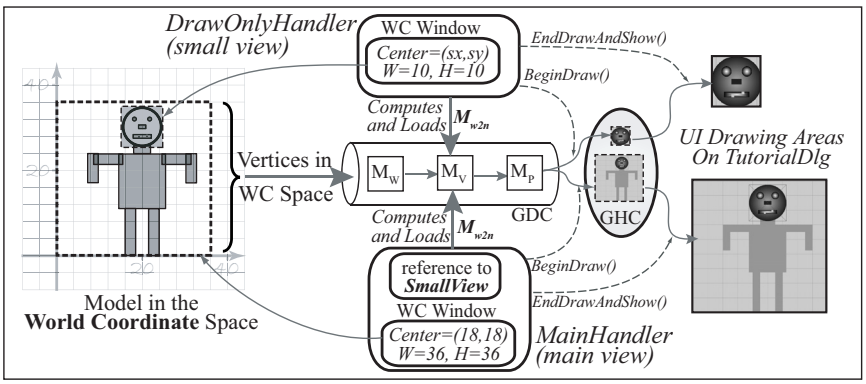

Figure 10.19. Implementation of Tutorial 10.10.

```
class CDrawOnlyHandler : public UWBD3D_WindowHandler {
    ⋮
    UWBMFC_UIWindow m_window; // the UI drawing area (device)
};

class CMainHandler : public CDrawOnlyHandler {
    ⋮
    void SetLinkedHandler(CDrawOnlyHandler* pHandler) { m_pLinkedHandler = pHandler; }
    ⋮
  protected:
    CDrawOnlyHandler* m_pLinkedHandler; // reference to the small view
};
```

Source file. `DrawOnlyHandler.h` file in the *WindowHandler* folder of the `D3D_Panning` project.

Listing 10.11. The `WindowHandler` classes of Tutorial 10.10.

```
 class CTutorialDlg : public CDialog {
     ⋮
A:   CMainHandler     m_main_view; // the main view
     CDrawOnlyHandler  m_small_view;  // the small view
};
```

Source file. `TutorialDlg.h/cpp` file in the *Source Files* folder of the `D3D_Panning` project.

```
BOOL CTutorialDlg::OnInitDialog() {
     ⋮
     if( !m_main_view.Initialize(*this, IDC_PLACEHOLDER) ) return FALSE;
     if( !m_small_view.Initialize(*this, IDC_PLACEHOLDER3) ) return FALSE;
     ⋮
B:   m_small_view.SetWCWindow(theApp.GetModel().GetWorldBounds());
     m_main_view.SetWCWindow( UWB_BoundingBox( vec3(0,0,0), vec3(36,36,0) ) );
C:   m_main_view.SetLinkedHandler( &m_small_view );
     ⋮
```

Listing 10.12. The `CTutorialDlg` classes of Tutorial 10.10.

`Handler` object calls the `LoadW2NDCXform()` function to compute and load the D3D RC $\mathbf{M_V}$ matrix with their corresponding $\mathbf{M}_{w2n}$ operator before drawing. The left side of Figure 10.19 shows that the model is defined in the WC space. When the model draws, it sends *all* the vertices of all the geometries to the D3D RC.

Source file.
`DrawAndMouseHandler.cpp` file in the *WindowHandler* folder of the `D3D_Panning` project.

```
void CMainHandler::DrawGraphics()
    ⋮
    BeginDraw();
    ⋮
A:  LoadW2NDCXform(); // compute M_w2n and load to M_V
    ⋮
    m_pD3DDevice->Clear( ⋮ ); // clears the UI drawing area (device)
    theApp.GetModel().DrawModel();   // model sends all geometries
    ⋮
    // draws the red wire-frame rectangle
B:  m_pLinkedHandler->DrawWCWindow(helper);
    EndDrawAndShow();
```

Listing 10.13. The `CMainHandler::DrawGraphics()` function of Tutorial 10.10.

As we have learned, the $\mathbf{M}_{w2n}$ operator ensures that only the region defined by the corresponding WC window is transformed into the ± 1 NDC space where the rest of the geometries will be clipped away by the graphics API. The right side of Figure 10.19 reminds us that that each `Handler` object is responsible for initializing the `swapchain` links with `BeginDraw()` and flushing the hardware buffer to the UI draw area with the `EndDrawAndShow()` function calls. Listing 10.11 shows the definition of the `CDrawOnlyHandler` and `CMainHandler` classes of Tutorial 10.10. We see that indeed both of the classes subclass from the `UWBD3D_WindowHandler` class and that the `CMainHandler` has a reference to the small view (m_pLinkedHandler). Referring to Listing 10.12, at label A we see that the main application window (`CTutorialDlg`) has an instance of each of the `Handler` objects representing the main and the small views. At label B, the WC window for each of the views is initialized. At label C, the main view gets a reference to the small view. Listing 10.13 shows the `DrawGraphics()` function of `MainHandler` calling `LoadW2NDCXForm()` to compute and load the corresponding $\mathbf{M}_{w2n}$ operator at label A (before drawing the model). At label B, the WC window of the `m_pLinkedHandler` (small view) is drawn (as the red wire-frame rectangle) in the main view.

With this structure, the two slider bars are connected to the small view's WC window center position. When the user adjusts the slider bars, the changes are immediately updated to the `m_small_view`'s `m_WCWindow`. In the subsequent redraw, the updated WC window computes an appropriate $\mathbf{M}_{w2n}$ operator that corresponds to the user input.

10.4.3 WC Window Dimension: Zooming

If we double the WC window size of the small view from Tutorial 10.10 such that

$$\text{WC window} \begin{cases} \text{center} = (15,25), \\ W_{wc} = H_{wc} = 20, \end{cases}$$

then the WC window covers a larger region in the WC system. From Equation (10.15):

$$\begin{aligned} x_{dc} &= ((x_{wc} - cx_{wc})\tfrac{W_{dc}}{W_{wc}}) + \tfrac{W_{dc}}{2}, \\ y_{dc} &= ((y_{wc} - cy_{wc})\tfrac{H_{dc}}{H_{wc}}) + \tfrac{H_{dc}}{2}, \end{aligned} \qquad (10.15)$$

we see that as W_{wc} and H_{wc} are the denominators in the scale factors, doubling these effectively halves the scaling factors. This implies that we should expect the elements on the DC display to decrease in size. As illustrated in Figure 10.20, we see that if we increase the WC window size, more elements in the WC are displayed in the output display area. Correspondingly, each individual element does indeed appear to be smaller on the output display. We see that increasing the WC window size results in the zooming-out sensation. For the same reasons, we can expect a zooming-in effect when we decrease the WC window size.

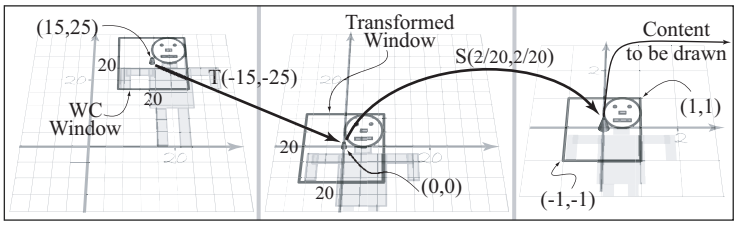

Figure 10.20. Double the WC window size.

Tutorial 10.11. Scaling the WC Window (W_{wc}, H_{wc})

Tutorial 10.11.
Project Name:
`D3D_Zooming`
Library Support:
`UWB_MFC_Lib1`
`UWB_D3D_Lib10`

- **Goal.** Verify zooming effect in relation to the WC window dimensions.
- **Approach.** Extend Tutorial 10.10 and allow user to adjust the width and height of the small view WC window.

Figure 10.21 is a screenshot of running Tutorial 10.11. This tutorial extends Tutorial 10.10 with two more slider bars at the lower right of the application window. The two new sliders are connected directly to the width (W_{wc}) and height (H_{wc}) of

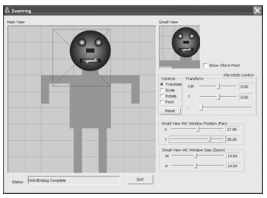

Figure 10.21. Running Tutorial 10.11.

the `m_small_view`'s `m_WCWindow`. Based on the previous discussions, we understand that as the user adjust the slider bars, the resulting changes will be reflected in the subsequent $\mathbf{M}_{w2n}$ (Equation (10.10)) operator that is loaded into the $\mathbf{M}_V$ matrix. In this tutorial, we can interactively adjust and observe the expected zooming effects. In addition, we observe the following.

- Changing W_{wc} without corresponding changes to H_{wc} (or changing H_{wc} without corresponding changes to W_{wc}) creates an annoying squeezing effect. We will examine this closely in the next section.
- The zooming effect appears to be defined with respect to the center of the WC window. That is, as we zooming in, it appears we are getting closer to the center of the WC window. This is a direct result of our scaling the width and height of the WC window with respect to the center of the WC window. It is left as an exercise for the reader to derive implementations to support zooming with respect to some other position in the WC window.

10.4.4 WC Window Width-to-Height Ratio: Aspect Ratio

Recall that the WC-to-DC transformation is governed by Equation (10.14), the $\mathbf{M}_{w2d}$ operator, or

$$\mathbf{M}_{w2d} = \mathbf{T}(-cx_{wc}, -cy_{wc})\, \mathbf{S}(\frac{W_{dc}}{W_{wc}}, \frac{H_{dc}}{H_{wc}})\, \mathbf{T}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}). \quad (10.14)$$

We observe that the middle scaling operator controls the zoom factor. When the scaling factors in the x- and y-directions are different, then the transformation from WC to DC will involve resizing objects in x- and y-directions by different amounts. In the cases of Tutorials10.4 and 10.5, we worked in the NDC space directly with a WC window size of

$$\begin{cases} W_{wc} = 2 \text{ with} & -1 \leq x \leq 1, \\ H_{wc} = 2 \text{ with} & -1 \leq y \leq 1. \end{cases}$$

For both tutorials,

$$W_{dc} \neq H_{dc}.$$

This means that the factors of the scaling operator in Equation (10.14) are different in the x- and y-directions. For this reason, in both cases we observed the squashed circles. In order to maintain proportional shapes from WC to DC, the x and y scaling factors in the $\mathbf{M}_{w2d}$ operator must be the same, or

$$\frac{W_{dc}}{W_{wc}} = \frac{H_{dc}}{H_{wc}}.$$

Collecting WC and DC terms on either sides of the equation,

$$\frac{W_{dc}}{H_{dc}} = \frac{W_{wc}}{H_{wc}}. \tag{10.16}$$

We define aspect ratio to be:

$$\text{aspect ratio } = \frac{\text{width}}{\text{height}}.$$

Equation (10.16) says that to maintain proportional shapes when transforming from WC to DC, the aspect ratio of the WC window must be the same as that of the device drawing area.

Tutorial 10.12. Experimenting with Aspect Ratios

Tutorial 10.12.
Project Name:
`AspectRatio`
Library Support:
`UWB_MFC_Lib1`
`UWB_D3D_Lib10`

- **Goal.** Understand WC window and DC aspect ratios and verify the artifacts when the ratios do not match.
- **Approach.** Extend Tutorial 10.11 and allow interactive changing of UI drawing dimension.

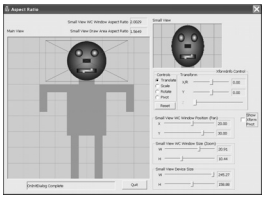

Figure 10.22. Running Tutorial 10.12.

Figure 10.22 is a screenshot of running Tutorial 10.12. This tutorial extends Tutorial 10.11 by including two more slider bars at the lower right of the application window. These two new slider bars control the device dimension of the small-view UI drawing area, or W_{dc} and H_{dc}. Recall that the other four slider bars control the small-view WC window location (cx_{wc} and cy_{wc}) and dimension (W_{wc} and H_{wc} sliders). From Equation (10.16) we understand that the image displayed in the small view will be distorted if/when we adjust the sliders bars such that

$$\frac{W_{dc}}{W_{wc}} \neq \frac{H_{dc}}{H_{wc}}.$$

We note that in general these four numbers can be controlled by the following.

- **Our application.** As the programmer of the application, we can design our application to allow user control of these values. For example, our application could allow the user to zoom in the world by allowing the WC window to change size, or our application could allow the user to increase/descrease the UI drawing area. In these cases, in order to maintain object proportions, it is important that our application presents a coherent and meaningful user interface. For example, we should implement zooming functionality by providing the user with one single zoom factor. The single zoom factor would ensure that the ratio $\frac{W_{dc}}{W_{wc}}$ remains constant and thus avoid the situation where WC and DC aspect ratios are different.

- **Window manager.** Because our application runs in a shared windowing environment, it is always possible that the environment may change the size of the UI drawing area and inform our application about the changes. In this case, if the DC aspect ratio is changed, then in order to maintain proper proportions, we must change the WC window dimension accordingly. For example, initially our application may have a WC window size of 10×10 and a DC displaying area of 200×200. If for some reason, the window manager decides to change the DC drawing area to 200×100, we must update the WC window to match the 0.5 DC aspect ratio. In this case, we can:

  - *increase the WC window size* to 10×20 and show more of the WC space; or
  - *decrease the WC window size* to 5×10 and show less of the WC space.

  The first option guarantees that we will show at least the original WC window, whereas the second option guarantees that we will show at most the original WC window. The choice between the above two options is a policy decision: neither choice is more correct. We should program our application to support the behavior specified by the user.

10.4.5 Summary

The world coordinate (WC) system is introduced to dissociate our model design space from the dimensions of the UI drawing area. As we have seen, this is advantageous for the following reasons.

- With the WC, we can define any coordinate system that is convenient for designing the geometric models and not be concerned with the dimensions of the eventual output drawing area.
- With the WC *window*, we can control the parameters (center, width, and height) to select the exact regions of the WC space to be displayed in the UI drawing area.

To properly support the WC system, the graphics API introduces the normalized device coordinate (NDC) system, a coordinate system bounded by ± 1. Our job as the graphics API programmer is to compute the $\mathbf{M}_{w2n}$ (Equation (10.10)) operator for each of the WC window regions we wish to display. The graphics API will

automatically clip away all geometries outside of the ± 1 range and will automatically compute the $\mathbf{M}_{n2d}$ (Equation (10.6)) operator for drawing the geometries to the output device. In our library design, the $\mathbf{M}_{w2n}$ operator is computed by the `WindowHandler` class and loaded into the `VIEW` matrix processor (M_V) of the D3D RC. Finally, we have seen that when defining the WC window, we must ensure that the WC window aspect ratio is the same as that of the UI drawing device. Otherwise, disproportional scaling will occur, resulting in images that appear squeezed.

10.5 Inverse Transformation

So far, we have concentrated on learning the output of geometric elements. To build interactive applications, we must interact with the user. In particular, we must understand how the added window coordinate system affects a user's picking or selecting an on-screen object by clicking the mouse buttons (in DC space).

In the Tutorial 5.6 implementation of the ball-shooting program, we compared mouse click positions with circles in the `AllWorldBalls` collection to locate the selected one. Notice that this simple operation involves a comparison across two distinct coordinate systems.

- **Mouse-click position (pt_{dc}).** This is a point on the display device (that the user's mouse clicked on). In the case of MFC, this position is returned to us in the hardware coordinate space. As described in Chapter 2 (Tutorial 2.4), we *flipped* the y-axis by subtracting the y-value from the height of the device and thereby converting the point into the device coordinate space with the origin at the lower-left corner. In all of the following discussions, we assume that the hardware-to-device transform has already been performed and we will work with points in DC space, pt_{dc}.

- `AllWorldBalls` **collection.** This is the set of all geometries in the world. By definition, these geometries are defined in the world coordinate (WC) space.

Hardware coordinate. Recall that the MFC API returns mouse click positions in the hardware coordinate system where the top-left is the origin with y-axis incrementing downward and x-axis increase rightward.

Hardware-to-device transform. This transformation is performed by the `HardwareToDevice()` function defined in the `WindowHandler` class.

In the case of Tutorial 5.6, although we did not distinguish between these two coordinate systems, the selection operation functioned correctly because we have carefully defined the WC window to coincide exactly with the DC space. In general, we must transform input mouse positions to the WC space before working with them. For example, in the Tutorial 10.7 implementation of displaying the

geometric face of Figure 10.11, we know

$$\text{DC coordinate:} \begin{cases} 0 \le x_{dc} \le 300, \\ 0 \le y_{dc} \le 300, \end{cases}$$

whereas

$$\text{WC window:} \begin{cases} 15 \le x_{wc} \le 25, \\ 25 \le y_{wc} \le 35. \end{cases}$$

In this case, the user's mouse clicks (pt_{dc}) will return points in DC space with range $[0,300]$. Clearly, this is very different from where the features of the face are defined in the WC space. In this case, we must perform the inverse of the $\mathbf{M}_{w2d}$ operator to transform pt_{dc} into a point in WC space. If we express

$$pt_{dc} = (x_{dc}, y_{dc})$$

as a vector

$$\mathbf{V}_{dc} = \begin{bmatrix} x_{dc} & y_{dc} \end{bmatrix},$$

then we must compute $\mathbf{V}_{wc}$, where

$$\mathbf{V}_{wc} = \mathbf{V}_{dc}\, \mathbf{M}_{w2d}^{-1}.$$

From Equation (10.14), recall that $\mathbf{M}_{w2d}$ is

$$\mathbf{M}_{w2d} = \mathbf{T}(-cx_{wc}, -cy_{wc})\mathbf{S}(\frac{W_{dc}}{W_{wc}}, \frac{H_{dc}}{H_{wc}})\mathbf{T}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2}). \quad (10.14)$$

From the discussion in Section 9.2, we know that the inverse of a concatenated operator is simply the inverse of each element concatenated in the reverse order:

Inverse transforms.

$\mathbf{T}^{-1}(t_x, t_y) = \mathbf{T}(-t_x, -t_y)$
$\mathbf{S}^{-1}(s_x, s_y) = \mathbf{S}(\frac{1}{s_x}, \frac{1}{s_y})$

$$\mathbf{M}_{w2d}^{-1} = \mathbf{M}_{d2w} = \mathbf{T}^{-1}(\frac{W_{dc}}{2}, \frac{H_{dc}}{2})\mathbf{S}^{-1}(\frac{W_{dc}}{W_{wc}}, \frac{H_{dc}}{H_{wc}})\mathbf{T}^{-1}(-cx_{wc}, -cy_{wc}),$$

which is

$$\mathbf{M}_{d2w} = \mathbf{T}(-\frac{W_{dc}}{2}, -\frac{H_{dc}}{2})\mathbf{S}(\frac{W_{wc}}{W_{dc}}, \frac{H_{wc}}{H_{dc}})\mathbf{T}(cx_{wc}, cy_{wc}) \quad (10.17)$$

or

$$\mathbf{V}_{wc} = \mathbf{V}_{dc}\mathbf{T}(-\frac{W_{dc}}{2}, -\frac{H_{dc}}{2})\mathbf{S}(\frac{W_{wc}}{W_{dc}}, \frac{H_{wc}}{H_{dc}})\mathbf{T}(cx_{wc}, cy_{wc}). \quad (10.18)$$

Equation (10.18) says that to transform a point (x_{dc}, y_{dc}) from the device drawing area (DC) to a point (x_{wc}, y_{wc}) in our design space (WC):

$$\begin{aligned} x_{wc} &= ((x_{dc} - \tfrac{W_{dc}}{2})\tfrac{W_{wc}}{W_{dc}}) + cx_{wc}, \\ y_{wc} &= ((y_{dc} - \tfrac{H_{dc}}{2})\tfrac{H_{wc}}{H_{dc}}) + cy_{wc}, \end{aligned} \quad (10.19)$$

where

$$\text{Device drawing area} \begin{cases} \text{width} &= W_{dc}, \\ \text{height} &= H_{dc}, \end{cases}$$

```
void UWB_WindowHandler::DeviceToWorld(int dcx, int dcy, float &wcx, float &wcy) const
    ⋮
A:  GetDeviceSize(dcW, dcH);              // Get the UI drawing device dimension
    vec3 center = m_WCWindow.GetCenter(); // Center position of the WC Window
    float wcW = m_WCWindow.Width();       // Width of the WC Window
    float wcH = m_WCWindow.Height();      // Height of the WC Window
B:  wcx = center.x + ((wcW/dcW) * (dcx - (dcW/2.0f))); // implement Equation 10.17
    wcy = center.y + ((wcH/dcH) * (dcy - (dcH/2.0f))); // by computing Equation 10.19
```

Listing 10.14. The `WindowHandler::DeviceToWorld()` implementation.

Source file. `uwbgl_WindowHandler3.cpp` file in the *Common Files/ WindowHandler* subfolder of the `UWBGL_D3D_Lib10` project.

and

$$\text{WC window} = \begin{cases} \text{center} & = (cx_{wc}, cy_{wc}), \\ \text{width} & = W_{wc}, \\ \text{height} & = H_{wc}. \end{cases}$$

Listing 10.14 shows the implementation of Equation (10.19) in the `DeviceToWorld()` transform function of the `UWB_WindowHandler` class in `D3D_UWB_Lib10`. At label A, we obtain the dimension of the drawing device and the parameters for the WC window. Label B is a direct implementation of Equation (10.19). In general, we must transform all user mouse-click positions by Equation (10.17) before comparing with points in the WC space.

Tutorial 10.13.
Project Name:
`D3D_BadMousePan`
Library Support:
`UWB_MFC_Lib1`
`UWB_D3D_Lib10`

Tutorial 10.14.
Project Name:
`D3D_MousePan`
Library Support:
`UWB_MFC_Lib1`
`UWB_D3D_Lib10`

Tutorials 10.13 and 10.14. Mouse Positions and Inverse Transforms

- **Goal.** Verify mouse positions and the need for device-to-world transformations.
- **Approach.** Extend Tutorial 10.12 to support moving of the small-view WC window by dragging with the left mouse button in the main view.

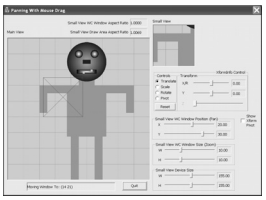

Figure 10.23. Running Tutorial 10.13 and 10.14.

Figure 10.23 is a screenshot of running Tutorials 10.13 and 10.14. In these tutorials, left mouse button drag in the main view defines the center position for the WC window of the small view. For example, from Figure 10.11 we know that the nose position of the geometric person is $(20, 30)$ in WC space. Now, if the user left mouse button clicks at the nose position in the main view, our application will react by moving the center of the small-view WC window to $(20, 30)$. If the user left mouse button drags towards the right-eye position, $(22, 32)$, our application will track the position by moving the center of the small-view WC window. Notice

```
// Mouse button services
void CMainHandler::OnMouseButton(bool down, unsigned int nFlags, int hwX, int hwY)
    ⋮
A:  HardwareToDevice(hwX, hwY, deviceX, deviceY); // hwXY -> deviceXY
    ⋮
B:  if(nFlags & MK_LBUTTON) // Left Mouse Button down
        // Compute new center for small view WC Window
        ComputeBoundPosition(deviceX, deviceY);

// Mouse move services
void CMainHandler::OnMouseMove(unsigned int nFlags, int hwX, int hwY)
    ⋮
A:  HardwareToDevice(hwX, hwY, deviceX, deviceY);
    ⋮
B:  if(nFlags & MK_LBUTTON) // Left Mouse Button drag
        // Compute new center for small view WC Window
        ComputeBoundPosition(deviceX, deviceY);
```

Source file. `DrawAndMouseHandler.cpp` file in the *WindowHandler* folder of the `D3D_BadMousePan` project.

Listing 10.15. The `MainHandler` mouse event service routines of Tutorial 10.13.

that in our discussion, these are in WC coordinate units, while we know that the mouse positions are in device coordinates. Tutorial 10.13 shows us the results of not transforming points to the WC space and using the mouse positions in the DC space directly. Listing 10.15 shows the mouse button click (`OnMouseButton`) and mouse move (`OnMouseMove`) service routines of Tutorial 10.13. In both service routines, at label A we transform the mouse click position from hardware to device coordinate, and then use the DC position to compute the center position for the small-view WC window. For this reason, in Tutorial 10.13, if we click around the top region of the main view, the corresponding DC points will have values around 300, and thus the small-view WC window will be moved to corresponding positions. Because nothing is defined in the 300 range in the WC space, nothing will show up in the small-view drawing area. Recall from Figure 10.10 that the geometric person is defined within the range of $[0, 40]$. This means if we left mouse button click/drag around the lower-left region of the main view, limiting our DC position to within the range of $[0, 40]$, we will see the small-view WC window panning around the geometric person. Clearly we must transform the DC positions to WC before computing the WC window position. Tutorial 10.14 extends from Tutorial 10.13 with the simple inclusion of the `DeviceToWorld()` function

calls. Listing 10.16 shows the mouse event source routines of Tutorial 10.14. As we can see at label A, the `DeviceToWorld()` function is called (for both mouse button and mouse move event service routines). By running Tutorial 10.14, we can verify that this tutorial functions as expected.

```
// Mouse button services
void CMainHandler::OnMouseButton(bool down, unsigned int nFlags, int hwX, int hwY)
    ⋮
    HardwareToDevice(hwX, hwY, deviceX, deviceY); // hwXY -> deviceXY
A:  DeviceToWorld(deviceX, deviceY, wcX, wcY); // deviceXY -> wcXY
    ⋮
    if(nFlags & MK_LBUTTON) // Left Mouse Button down
        // Compute new center of small view WC Window based on wcXY
        ComputeBoundPosition(wcX, wcY);

// Mouse move services
void CMainHandler::OnMouseMove(unsigned int nFlags, int hwX, int hwY)
    ⋮
    HardwareToDevice(hwX, hwY, deviceX, deviceY); // hwXY -> deviceXY
A:  DeviceToWorld(deviceX, deviceY, wcX, wcY); // deviceXY -> wcXY
    ⋮
    if(nFlags & MK_LBUTTON) // Left Mouse Button down
        // Compute new center of small view WC Window based on wcXY
        ComputeBoundPosition(wcX, wcY);
```

Source file. `DrawAndMouseHandler.cpp` file in the *WindowHandler* folder of the `D3D_MousePan` project.

Listing 10.16. The `MainHandler` mouse event service routines of Tutorial 10.14.