Chapter 17

D3DX Helper Objects

"For a man to help another is to be a god."
Pliny the Elder, *Natural History*, II, 77

17.1 Overview

In addition to the objects provided by D3DX as concrete classes, D3DX provides several abstract COM objects for performing tasks associated with rendering. Matrix stack objects aid in the maintenance of coordinate systems while traversing a hierarchy of relative coordinate frames. Font and sprite objects aid in the rendering of text with GDI and textured screen-space quadrilaterals, respectively. The render to surface and render to environment map objects simplify the use of secondary render targets and the construction of environment map textures. These are relatively simple objects.

D3DX also provides an object that implements visual effects. Effects are specified through a text syntax and implement an effect with one or more techniques. Each technique consists of a collection of rendering passes and the required state for each pass. Effects can be parameterized to allow the application to pass in variable data.

17.2 Matrix Stacks

A matrix stack object is helpful in implementing scene graphs containing a hierarchy of coordinate frames described by relative transformation matrices. A matrix stack object is created with `D3DXCreateMatrixStack`. The `DWORD` argument `zero` is for unused flags and is always zero. The `ID3DXMatrixStack` interface is summarized in interface 17.1. A newly created matrix stack is empty and its top element is undefined.

```
HRESULT D3DXCreateMatrixStack(DWORD zero,
```

```
            ID3DXMatrixStack **result);
```

Interface 17.1: Summary of the `ID3DXMatrixStack` interface.

ID3DXMatrixStack

Read-Only Properties	
`GetTop`	The topmost matrix in the stack
Methods	
`LoadIdentity`	Loads an identity matrix
`LoadMatrix`	Loads an arbitrary matrix
`MultMatrix`	Post-multiplies a matrix
`MultMatrixLocal`	Pre-multiplies a matrix
`Pop`	Pops the topmost matrix from the stack
`Push`	Pushes a matrix onto the stack
`RotateAxis`	Post-multiplies a rotation matrix
`RotateAxisLocal`	Pre-multiplies a rotation matrix
`RotateYawPitchRoll`	Post-multiplies a rotation matrix
`RotateYawPitchRollLocal`	Pre-multiplies a rotation matrix
`Scale`	Post-multiplies a scaling matrix
`ScaleLocal`	Pre-multiplies a scaling matrix
`Translate`	Post-multiplies a translation matrix
`TranslateLocal`	Pre-multiplies a translation matrix

```
interface ID3DXMatrixStack : IUnknown
{
    // read-only properties
    D3DXMATRIX *GetTop();

    // methods
    HRESULT LoadIdentity();
    HRESULT LoadMatrix(const D3DXMATRIX *m);
    HRESULT MultMatrix(const D3DXMATRIX *m);
    HRESULT MultMatrixLocal(const D3DXMATRIX *m);
    HRESULT Pop();
    HRESULT Push();
    HRESULT RotateAxis(const D3DXVECTOR3 *v,
                float angle);
    HRESULT RotateAxisLocal(const D3DXVECTOR3 *v,
                float angle);
    HRESULT RotateYawPitchRoll(float Yaw,
                float pitch,
                float roll);
```

```
    HRESULT RotateYawPitchRollLocal(float Yaw,
                float pitch,
                float roll);
    HRESULT Scale(float x, float y, float z);
    HRESULT ScaleLocal(float x, float y, float z);
    HRESULT Translate(float x, float y, float z);
    HRESULT TranslateLocal(float x, float y, float z);
};
```

The topmost matrix on the stack can be accessed with the `GetTop` method. This method returns a pointer to the matrix, which allows the application to read or write values into the matrix. The matrix stack object is rather forgiving of erroneous usage patterns, but makes no complaints to the debug output stream, so you should be careful to correctly use the stack, or you may get unexpected values from the matrix.

You can load an identity matrix, or an arbitrary matrix with the `LoadIdentity` and `LoadMatrix` methods, respectively. Once a matrix is on the stack, you can perform stack manipulations with the `Push` and `Pop` methods. `Push` inserts a copy of the topmost matrix at the top of the stack and pushes all the other elements down in the stack. `Pop` removes the topmost stack element. Although the matrix stack will not return an error when you pop off more elements than are on the stack, the top element of the stack will be undefined.

You can post-multiply a matrix onto the top of the stack with the `MultMatrix` method. You can pre-multiply a matrix onto the top of the stack with the `MultMatrixLocal` method. The `Scale`, `RotateAxis`, `RotateYawPitchRoll`, and `Translate` methods create scale, axis rotation, Euler angle rotation and translation matrices and post-multiply them onto the top of the stack. The `Local` versions of these methods pre-multiply the constructed matrix onto the top of the stack.

You can also use the C++ Standard Library classes `std::stack` or `std::vector` to implement your own matrix stack. The stack class has `push`, `pop` and `top` methods that function similarly to those in `ID3DXMatrixStack`. The vector class has `push_back`, `pop_back` and `back` methods that serve the same purposes.

```
typedef std::stack<D3DXMATRIX> t_matrix_stack;
typedef std::vector<D3DXMATRIX> t_matrix_vector;
```

Neither of these classes have methods for constructing matrices and multiplying them onto the top of the stack. However, you can perform these operations directly using the C++ extensions to `D3DXMATRIX` and matrix construction helper classes. For example, the following snippet uses a stack implemented as a `std::vector`. First, a rotation matrix is post-multiplied onto the top of the stack and then a translation matrix is pre-multiplied onto the top of the stack.

```
std::vector<D3DXMATRIX> stack;
stack.push_back(D3DXMATRIX(1, 0, 0, 0,
```

```
                              0, 1, 0, 0,
                              0, 0, 1, 0,
                              0, 0, 0, 1));
stack.back() *= rt::mat_rot_x(D3DX_PI*0.5f);
stack.back() = rt::mat_trans(1,0,0)*stack.back();
```

17.3 Fonts

D3DX provides the `ID3DXFont` interface for GDI-based text rendering in Direct-3D. Text rendered with this interface will first be rasterized by GDI into a memory DC. The contents of the memory DC will then be copied to a texture and a textured quadrilateral will be drawn to get the text into the render target. This involves two potentially slow operations each time text is rendered: GDI text rendering is performed, and the new texture contents must be copied into device memory. However, the quality of the rendered text is high, properly handling kerning and ligature issues, as well as text containing multibyte character sequences and UNICODE text. An instance of the interface is created by the `D3DXCreateFont` function.

```
HRESULT D3DXCreateFont(IDirect3DDevice9 *device,
        HFONT font,
        ID3DXFont **result);
```

The `font` argument specifies the handle to an existing GDI font. You can create a font from a `LOGFONT` structure for use with this function and delete the font after calling `D3DXCreateFont`, as in this example from the `rt_Text` sample.

```
{
    HFONT font = ::CreateFontIndirect(&m_d3dx_lf);
    THR(::D3DXCreateFont(m_pd3dDevice, font, &m_d3dx_font));
    ::DeleteObject(font);
}
```

The function `D3DXCreateFontIndirect` performs these steps for you as a convenience.

```
HRESULT D3DXCreateFontIndirect(IDirect3DDevice9 *device,
        const LOGFONT *description,
        ID3DXFont **result);
```

The `ID3DXFont` interface is summarized in interface 17.2. The font interface has only two properties: the associated device, and the `LOGFONT` description of the associated GDI font.

Interface 17.2: Summary of the `ID3DXFont` interface.

ID3DXFont

Read-Only Properties	
GetDC	GetDC
GetDescA	GetDescA
GetDescW	GetDescW
GetDevice	The associated device
GetGlyphData	GetGlyphData
GetTextMetricsA	GetTextMetricsA
GetTextMetricsW	GetTextMetricsW

Methods	
DrawTextA	Renders ASCII text
DrawTextW	Renders UNICODE text
OnLostDevice	Releases device resources
OnResetDevice	Restores device resources
PreloadCharacters	PreloadCharacters
PreloadGlyphs	PreloadGlyphs
PreloadTextA	PreloadTextA
PreloadTextW	PreloadTextW

```
interface ID3DXFont : IUnknown
{
    // read-only properties
    HRESULT GetDevice(IDirect3DDevice9 **value);
    HRESULT GetLogFont(LOGFONT *value);

    // methods
    HRESULT Begin();
    INT DrawTextA(LPCSTR text,
                  INT count,
                  RECT *region,
                  DWORD flags,
                  D3DCOLOR color);
    INT DrawTextW(LPCWSTR text,
                  INT count,
                  RECT *region,
                  DWORD flags,
                  D3DCOLOR color);
    HRESULT End();
    HRESULT OnLostDevice();
    HRESULT OnResetDevice();
};
```

The font object uses a state block to preserve the state of the device during drawing and a texture for use in drawing the result of GDI's rendering. The `OnLostDevice` and `OnResetDevice` methods are used to manage the lifetime of the font's device resources. They should be called when the device is lost or reset, respectively.

TODO: fix font

The `xfontBegin` and `xfontEnd` methods set and restore device state in preparation for text rendering. When drawing more than one text string, call `xfontBegin` to set the appropriate state once for all the text. Similarly, call `xfontEnd` after the text is rendered to restore the device state. If these methods are not called explicitly, then the methods `DrawTextA` and `DrawTextW` will call them implicitly each time they are called, to properly save and restore device state.

The `DrawTextA` and `DrawTextW` methods render ANSI or UNICODE text, respectively.[1] These methods function similarly to the GDI function `::DrawText`. If the `count` argument is -1, then the `text` argument is assumed to be `NULL` terminated. The `region` parameter points to a rectangle in render target pixel coordinates where the text should be rendered. The `color` argument gives the color of the rendered text. The texture map created internally by `ID3DXFont` is modulated by this color.

The `flags` argument specifies a set of formatting options that are identical to those described in MSDN for `::DrawText`. The flags are summarized as follows.

`DT_BOTTOM` The text is aligned to the bottom of the rectangle. This value must be used with `DT_SINGLELINE`.

`DT_CALCRECT` The width and height of the required rectangle are computed and returned to the caller. No rendering is performed. If there are multiple lines of text, `DrawText` uses the width of the rectangle given in the `region` parameter and extends the base of the rectangle to bound the last line of text. If there is only one line of text, `DrawText` modifies the right side of the rectangle so that it bounds the last character in the line. In either case, `DrawText` returns the bounds of the formatted text but does not draw the text.

`DT_CENTER` The text is centered horizontally within the rectangle.

`DT_EDITCONTROL` Duplicates the text-displaying characteristics of a multiline edit control. Specifically, the average character width is calculated in the same manner as for an edit control, and the function does not display a partially visible last line.

`DT_END_ELLIPSIS` **or** `DT_PATH_ELLIPSIS` Truncates the string without adding ellipses so that the result fits in the specified rectangle. The string is not modified unless the `DT_MODIFYSTRING` flag is specified. Specify `DT_END_ELLIPSIS` to truncate characters at the end of the string, or `DT_PATH_ELLIPSIS` to truncate characters in the middle of the string. If the string

[1] Collectively, they will be referred to as `DrawText`.

contains backslash characters, DT_PATH_ELLIPSIS preserves as much of the text as possible after the last backslash.

DT_EXPANDTABS Expands tab characters. The default number of characters per tab is eight. The DT_WORD_ELLIPSIS, DT_PATH_ELLIPSIS, and DT_END_ELLIPSIS values cannot be used with the DT_EXPANDTABS value.

DT_EXTERNALLEADING Includes the font external leading in line height. Normally, external leading is not included in the height of a line of text.

DT_HIDEPREFIX Windows 2000: Ignores the ampersand (&) prefix character in the text. The letter that follows is not underlined, but other mnemonic-prefix characters are still processed.

DT_INTERNAL Uses the system font to calculate text metrics.

DT_LEFT Aligns text to the left.

DT_MODIFYSTRING Modifies the string to match the displayed text. This flag has no effect unless the DT_END_ELLIPSIS or DT_PATH_ELLIPSIS flag is specified.

DT_NOCLIP Draws without clipping. DrawText is somewhat faster when DT_NOCLIP is used.

DT_NOFULLWIDTHCHARBREAK Windows 98, Windows 2000: Prevents a line break at a double-wide character string (DBCS), so that the line-breaking rule is equivalent to a single-byte character string (SBCS). For example, this can be used in Korean windows for more readability of icon labels. This is effective only if DT_WORDBREAK is specified.

DT_NOPREFIX Turns off processing of prefix characters. Normally, DrawText interprets the mnemonic-prefix character & as a directive to underscore the character that follows, and the mnemonic-prefix characters && as a directive to print a single &. By specifying DT_NOPREFIX, this processing is turned off. Compare with DT_HIDEPREFIX and DT_PREFIXONLY.

DT_PREFIXONLY Windows 2000: Draws only an underline at the position of the character following the ampersand (&) prefix character. Does not draw any character in the string.

DT_RIGHT Aligns text to the right.

DT_RTLREADING Displays text in right-to-left reading order for bi-directional text when a Hebrew or Arabic font is selected. The default reading order for all text is left-to-right.

DT_SINGLELINE Displays text on a single line only. Carriage returns and line feeds do not break the line.

`DT_TABSTOP` Sets tab stops. Bits 15-8 (high-order byte of the low-order word) of the Format parameter specify the number of characters for each tab. The default number of characters per tab is eight. The `DT_CALCRECT`, `DT_EXTERNALLEADING`, `DT_INTERNAL`, `DT_NOCLIP`, and `DT_NOPREFIX` values cannot be used with the `DT_TABSTOP` value.

`DT_TOP` Top-justifies text (single line only).

`DT_VCENTER` Centers text vertically (single line only).

`DT_WORDBREAK` Breaks words. Lines are automatically broken between words if a word would extend past the edge of the rectangle specified by the `region` parameter. A carriage return/line feed sequence also breaks the line.

`DT_WORD_ELLIPSIS` Truncates text that does not fit in the rectangle and adds ellipses.

For special effects with fonts, you can modify the state used to render

The SDK sample framework includes a `CD3DFont` class that implements text rendering in a different manner. It rasterizes character glyphs into a texture map and draws textured quads for each character when a string is drawn. This method uses GDI only for the construction of the texture map; all drawing is performed entirely with Direct3D. This improves performance, but introduces some limitations on the text that can be drawn, see section B.6.

17.4 Sprites

A sprite is a textured quadrilateral that is always drawn parallel to the screen. D3DX implements a sprite object with the `ID3DXSprite` interface. An instance of a sprite object is created with `D3DXCreateSprite`.

```
HRESULT D3DXCreateSprite(IDirect3DDevice9 *device,
            ID3DXSprite **result);
```

The interface is summarized in interface 17.3. The only property exposed by a sprite is the device property, which returns the device associated with the sprite. The texture used for the sprite does not belong to the sprite and is passed into the sprite drawing methods. A sprite uses a state block to save and restore state on the device, with the `Begin` and `End` methods, respectively. Calls to `Begin` and `End` should surround code that draws multiple sprites so that the appropriate device state is set and restored only once for all the drawn sprites. The `OnLostDevice` and `OnResetDevice` methods should be called when a device is lost or reset, respectively, to manage the device resources associated with the sprite.

Interface 17.3: Summary of the `ID3DXSprite` interface.

ID3DXSprite

Read-Only Properties	
`GetDevice`	The associated device

Write-Only Properties	
`SetWorldViewLH`	SetWorldViewLH
`SetWorldViewRH`	SetWorldViewRH

Properties
`GetTransform`
`SetTransform`

Methods	
`Begin`	Begin
`Draw`	Draw
`End`	End
`Flush`	Flush
`OnLostDevice`	Releases device resources
`OnResetDevice`	Restores device resources

```
interface ID3DXSprite : IUnknown
{
    // read-only properties
    HRESULT GetDevice(IDirect3DDevice9 **value);

    // methods
    HRESULT Begin();
    HRESULT Draw(IDirect3DTexture9 *texture,
                 const RECT *region,
                 const D3DXVECTOR2 *scaling,
                 const D3DXVECTOR2 *rotation_center,
                 float angle,
                 const D3DXVECTOR2 *translation,
                 D3DCOLOR color);
    HRESULT DrawTransform(IDirect3DTexture9 *texture,
                 const RECT *region,
                 const D3DXMATRIX *matrix,
                 D3DCOLOR color);
    HRESULT End();
    HRESULT OnLostDevice();
    HRESULT OnResetDevice();
};
```

The `Draw` method renders a sprite using the given texture. The `region` parameter defines what portion of the texture will be used to render the sprite; if `NULL` is passed, then the entire texture is used for the sprite. The `color` argument is used to modulate the given texture. If no modulation is required, then pass opaque white as the color.[2]

The `scaling`, `rotation_center`, `angle`, and `translation` arguments are used to transform the sprite in screen space. The sprite is initially located in screen space with its upper-left corner at $(0, 0)$. The order of transformations is $S(s)T(-C_r)T_z(\theta)T(C_r)T(P)$, where s is the `scaling` parameter, C_r is the `rotation_center` parameter, θ is the `angle` parameter, and P is the `translation` parameter.

The `DrawTransform` method takes a transformation matrix directly. The matrix maps screen space coordinates to screen space. Remember that screen space has an increasing y value from top to bottom, unlike cartesian coordinates, when constructing this matrix.

One drawback of `ID3DXSprite` is that there is no batching of the rendered sprites, effectively resulting in a call to `DrawPrimitive` for each time one of the draw methods is called on the sprite object. This reduces performance if many sprites must be drawn.

17.5 Surface Rendering

D3DX provides the `ID3DXRenderToSurface` interface to simplify rendering to an arbitrary surface. An instance of this object is obtained with the `D3DXCreateRenderToSurface` function.

```
HRESULT D3DXCreateRenderToSurface(IDirect3DDevice9 *device,
                UINT width,
                UINT height,
                D3DFORMAT format,
                BOOL depth_stencil,
                D3DFORMAT ds_format,
                ID3DXRenderToSurface **result);
```

A summary of the interface is given in interface 17.4.

Interface 17.4: Summary of the `ID3DXRenderToSurface` interface.

ID3DXRenderToSurface

Read-Only Properties

`GetDesc`	A description of the render target surface
`GetDevice`	The associated device

Methods

[2] Most easily written in C++ as `~0`.

BeginScene	Start rendering to a surface
EndScene	Finish rendering to a surface
OnLostDevice	Releases device resources
OnResetDevice	Restores device resources

```
interface ID3DXRenderToSurface : IUnknown
{
    // read-only properties
    HRESULT GetDesc(D3DXRTS_DESC *value);
    HRESULT GetDevice(IDirect3DDevice9 **value);

    // methods
    HRESULT BeginScene(IDirect3DSurface9 *surface,
                const D3DVIEWPORT9 *viewport);
    HRESULT EndScene();
    HRESULT OnLostDevice();
    HRESULT OnResetDevice();
};
```

GetDevice returns the device associated with the render to surface object. The GetDesc method returns a description of the created render to surface object in the D3DXRTSDESC structure. The structure gives the dimensions and format of the render target and any associated depth buffer, if any. A render to surface object must be destroyed and recreated to change any of these values.

```
typedef struct _D3DXRTS_DESC
{
    UINT      Width;
    UINT      Height;
    D3DFORMAT Format;
    BOOL      DepthStencil;
    D3DFORMAT DepthStencilFormat;
} D3DXRTS_DESC;
```

The OnLostDevice and OnResetDevice manage any device resources associated with the render to surface object. They should be called when the device is lost or reset, respectively.

The BeginScene and EndScene methods are the meat of the object. The surface parameter specifies the destination of the rendering. The destination is not required to be a surface created with the render target usage. If the destination isn't a valid render target, then ID3DXRenderToSurface will create a render target surface which will be copied into the destination surface after rendering is complete.

If you call BeginScene while the associated device is inside a scene, then the call will fail. The following code snippet shows how to render to a texture level

surface and use the texture to render another scene on the back buffer of the default swap chain.

```
CComPtr<IDirect3DSurface9> level;
THR(m_texture->GetSurfaceLevel(0, &level));
THR(m_rts->BeginScene(level, NULL));
render_texture_scene();
THR(m_rts->EndScene());
THR(::D3DXFilterTexture(m_texture, NULL, 0, D3DX_DEFAULT));

THR(m_pd3dDevice->BeginScene());
THR(m_pd3dDevice->SetTexture(0, m_texture);
render_scene();
THR(m_pd3dDevice->EndScene());
```

The most detailed level of a texture is obtained to be the destination of the rendering. The call to `render_texture_scene` results in the setting of device state and the rendering of primitives for the scene rendered onto the texture. After the scene is rendered, D3DX is used to filter the top level to other mipmap levels in the texture. Finally, the rendered and filtered texture is used as the source for rendering and `render_scene` is called.

17.6 Environment Map Rendering

Rendering to an environment with `ID3DXRenderToEnvMap` is similar to rendering to a surface with `ID3DXRenderToSurface`, except that viewing and projection into the target are suitably adjusted for the appropriate environment map. An instance of this object is created with `D3DXCreateRenderToEnvMap`.

```
HRESULT D3DXCreateRenderToEnvMap(IDirect3DDevice9 *device,
                UINT size,
                D3DFORMAT format,
                BOOL depth_stencil,
                D3DFORMAT ds_format,
                ID3DXRenderToEnvMap **result);
```

The destination of the rendering is not required to be a texture created with the render target usage. If the destination isn't a valid render target, then `ID3DXRenderToEnvMap` will create a render target texture which will be copied into the destination texture after rendering is complete. A summary of the interface is given in interface 17.5.

Interface 17.5: Summary of the `ID3DXRenderToEnvMap` interface.

ID3DXRenderToEnvMap

Read-Only Properties	
GetDesc	A description of the render target used
GetDevice	The associated device

Methods	
BeginCube	Begin rendering to a cubic environment map
BeginHemisphere	Begin rendering to a hemispherical environment map
BeginParabolic	Begin rendering to a parabolic environment map
BeginSphere	Begin rendering to a spherical environment map
End	End rendering to an environment map
Face	Select a cubic environment map face for rendering
OnLostDevice	Releases device resources
OnResetDevice	Restores device resources

```
interface ID3DXRenderToEnvMap : IUnknown
{
    // read-only properties
    HRESULT GetDesc(D3DXRTE_DESC *value);
    HRESULT GetDevice(IDirect3DDevice9  **value);

    // methods
    HRESULT BeginCube(IDirect3DCubeTexture9 *texture);
    HRESULT BeginHemisphere(IDirect3DTexture9 *pos_z,
                IDirect3DTexture9 *neg_z);
    HRESULT BeginParabolic(IDirect3DTexture9 *pos_z,
                IDirect3DTexture9 *neg_z);
    HRESULT BeginSphere(IDirect3DTexture9 *texture);
    HRESULT End();
    HRESULT Face(D3DCUBEMAP_FACES face);
    HRESULT OnLostDevice();
    HRESULT OnResetDevice();
};
```

`GetDevice` returns the device associated with the render to environment map object. `GetDesc` method returns a description of the created render to environment map object in the `D3DXRTEDESC` structure. The structure gives the dimensions and format of the render target texture and any associated depth buffer, if any. The render targets are assumed to be square in all cases. A render to environment map object must be destroyed and recreated to change any of these values.

```
typedef struct _D3DXRTE_DESC
{
    UINT      Size;
```

```
    D3DFORMAT Format;
    BOOL      DepthStencil;
    D3DFORMAT DepthStencilFormat;
} D3DXRTE_DESC;
```

The `OnLostDevice` and `OnResetDevice` manage any device resources associated with the render to surface object. They should be called when the device is lost or reset, respectively.

The `Begin` methods are the meat of the object. They begin rendering a scene for a cubic, spherical, hemispherical or parabolic environment map. The `Face` method should then be called and the environment scene for each of the six cube faces. All environment maps need all six faces of the cube rendered in order to construct a complete environment map. The `End` method completes scene rendering and performs any copying necessary for non-render target destinations. If you call one of the begin methods while the associated device is inside a scene, then the call will fail.

The following code snippet shows how to render to all six surfaces of a cubemap texture and use the texture to render another scene on the back buffer of the default swap chain.

```
THR(m_rtem->BeginCube(m_texture));
for (UINT i = 0; i < 6; i++)
{
    THR(m_rtem->Face(D3DCUBEMAP_FACES(i)));
    THR(m_pd3dDevice->SetTransform(D3DTS_VIEW,
        &::D3DUtil_GetCubeMapViewMatrix(D3DCUBEMAP_FACES(i)));
    render_texture_scene();
}
THR(m_rtem->End());

THR(m_pd3dDevice->BeginScene());
THR(m_pd3dDevice->SetTexture(0, m_texture));
render_scene();
THR(m_pd3dDevice->EndScene());
```

The rendering of the cubemap texture is started and then a loop is made over each face, setting the appropriate viewing transformation The most detailed level of a texture is obtained to be the destination of the rendering. The call to `render_texture_scene` results in the setting of device state and the rendering of primitives for the scene rendered onto the texture. After the scene is rendered on all six faces of the cubemap, the texture is used as the source for rendering and `render_scene` is called.