Microsoft Direct3D Devices 1. Device Types

This is preliminary documentation and is subject to change.

Hardware Device

The primary device type is the hardware device, which supports hardware-accelerated rasterization and both hardware and software vertex processing. If the computer on which your application is running is equipped with a display adapter that supports Microsoft® Direct3D®, your application should use it for 3-D operations. Direct3D hardware devices implement all or part of the transformation, lighting, and rasterizing modules in hardware.

Applications do not access 3-D cards directly. They call Direct3D functions and methods. Direct3D accesses the hardware through a hardware abstraction layer. If the computer running your application supports the abstraction layer, it will gain the best performance by using a hardware device.

To specify hardware rasterization and shading, create a Device object using the DeviceType.Hardware constant, as shown in Create a Direct3D Hardware Device.

Note Hardware devices cannot render to 8-bit render-target surfaces.

Reference Device

Direct3D supports an additional device type called a reference device or reference rasterizer. Unlike a software device, the reference rasterizer supports every Direct3D feature. Because these features are implemented for accuracy, rather than speed, and are implemented in software, the results are not very fast. The reference rasterizer does make use of special CPU instructions whenever it can, but it is not intended for retail applications. Use the reference rasterizer only for feature testing or demonstration purposes.

To create a reference device, create a Device object using the DeviceType.Reference constant.

Devices 2. Creating a Device

This is preliminary documentation and is subject to change.

Note All rendering devices created by a given Microsoft® Direct3D® object share the same physical resources. Although your application can create multiple rendering devices from a single Direct3D object, because they share the same hardware, extreme performance penalties will be incurred.

To create a Direct3D device, your managed code application should first create a PresentParameters object that contains the presentation parameters for the device. The following code example contains a simple initialization using the Device(Int32,DeviceType,Control,CreateFlags,PresentParameters) constructor that receives a PresentParameters object.

```
using Microsoft.DirectX.Direct3D;
.
.
.
// Global variables for this project
Device device = null;              // Create rendering device
PresentParameters presentParams = new PresentParameters();

device = new Device(0, DeviceType.Hardware, this,
                    CreateFlags.SoftwareVertexProcessing, presentParams);
```

This call also specifies the default adapter, a hardware device, and software vertex processing.

Note that a call to create, dispose, or reset the device should happen only on the same thread as the window procedure of the focus window. After creating the device, set its state.

Related Topics

- Create a Direct3D Device

Devices 3. Selecting a Device

This is preliminary documentation and is subject to change.
Applications can query hardware to detect the supported Microsoft® Direct3D® device types. This section contains information on the primary tasks involved in enumerating display adapters and selecting Direct3D devices.
An application must perform a series of tasks to select an appropriate Direct3D device. Note that the following steps are intended for a full-screen application. In most cases, a windowed application can skip most of these steps.
1. Initially, the application must enumerate the display adapters on the system. An adapter is a physical piece of hardware. Note that the graphics card might contain more than a single adapter, as is the case with a dual-head display. For a detailed example of enumerating adapters, see (SDK root)\Samples\C# \Common\D3DEnumeration.cs. Applications that are not concerned with multiple-monitor support can disregard this step and use the default adapter enumeration.
2. For each adapter, the application enumerates the supported display modes with the Adapters property of the Manager object. The collection of adapters can be queried with the AdapterListEnumerator object.
3. If required, the application checks for the presence of hardware acceleration in each enumerated display mode by checking the return value of the **checkType** parameter of Manager.CheckDeviceType. Other parameters of CheckDeviceType can provide additional information on the current display and back buffer formats of the device.
4. The application checks for the desired level of functionality for the device on this adapter using the Device.DeviceCaps property. This property returns a Caps structure that describes the capabilities of the device. Those devices that do not support the required functionality will not contain a valid structure. The device capability returned by DeviceCaps is guaranteed to be constant for a device across all display modes verified by CheckDeviceType.
5. Devices can always render to surfaces of the format of an enumerated display mode that is supported by the device. If the application is required to render to a surface of a different format, it can call CheckDeviceType. If the device can render to the format, then it is guaranteed that all capabilities returned by DeviceCaps are applicable.
6. Lastly, the application can determine if multisampling techniques, such as full-scene antialiasing, are supported for a render format by using the CheckDeviceMultiSampleType method.

After completing the above steps, the application should have a list of display modes in which it can operate. The final step is to verify that enough device-accessible memory is available to accommodate the required number of buffers and antialiasing. This test is necessary because the memory consumption for the mode and multisample combination is impossible to predict without verifying it. Moreover, some display adapter architectures might not have a constant amount of device-accessible memory. This means that an application should be able to report out-of-video-memory failures when going into full-screen mode. Typically, an application should remove full-screen mode from the list of modes it offers to a user, or it should attempt to consume less memory by reducing the number of back buffers or using a simpler multisampling technique.
A windowed application performs a similar set of tasks, as follows.
1. It determines the desktop rectangle covered by the client area of the window.
2. It enumerates adapters, looking for the adapter for which the monitor covers the client area. If the client area is owned by more than one adapter, then the application can choose to drive each adapter independently, or to drive a single adapter and have Direct3D transfer pixels from one device to another at presentation. The application can also disregard the above two steps and use the default adapter, although this might result in slower operation when the window is placed on a secondary monitor. The default adapter is the first ordinal identified by the DeviceCreationParameters.AdapterOrdinal. The adapter ordinal value can be queried with the GetDeviceCaps method or returned by the AdapterInformation.Adapter property.
3. The application should call CheckDeviceType to determine if the device can support rendering to a back buffer of the specified format while in desktop mode. The AdapterInformation.CurrentDisplayMode property can be used to determine the desktop display format.
4.

Devices 5. Determining Hardware Support

This is preliminary documentation and is subject to change.
Microsoft® Direct3D® provides the following methods to determine hardware support.

- CheckDeviceFormat

Used to verify whether a surface format can be used as a texture, whether a surface format can be used as a texture and a render target, or whether a surface format can be used as a depth-stencil buffer. In addition, this method is used to verify depth buffer format support and depth-stencil buffer format support.

- CheckDeviceType

Used to verify a device's ability to perform hardware acceleration, a device's ability to build a swap chain for presentation, or a device's ability to render to the current display format.

- CheckDepthStencilMatch

Used to verify whether a depth-stencil buffer format is compatible with a render-target format. Note that before calling this method, the application should call CheckDeviceFormat on both the depth-stencil and render target formats.

Devices 6. Processing Vertex Data

This is preliminary documentation and is subject to change.
The Device class supports vertex processing in both software and hardware. In general, the device capabilities for software and hardware vertex processing are not identical. Hardware capabilities are variable, depending on the display adapter and driver, while software capabilities are fixed.
The following constants of the CreateFlags enumeration control vertex processing behavior for the hardware and reference devices.

- SoftwareVertexProcessing
- HardwareVertexProcessing
- MixedVertexProcessing

The BehaviorFlags structure can also be used to control behavior using properties that access these same constant values. Specify one of these flags when calling the Device constructor. The MixedVertexProcessing mixed-mode flag enables the device to perform both software and hardware vertex processing. Only one of these three vertex processing flags may be set for a device at any one time; they are mutually exclusive. Note that the HardwareVertexProcessing flag is required to be set when creating a pure device (PureDevice).
To avoid dual vertex processing capabilities on a single device, only the hardware vertex processing capabilities can be queried at run time. Software vertex processing capabilities are fixed and cannot be queried at run time.
You can consult the properties of the VertexProcessingCaps structure to determine the hardware vertex processing capabilities of the device. For software vertex processing the following VertexProcessingCaps capabilities are supported (which are all but two of the available capabilities).

- SupportsDirectionAllLights
- SupportsLocalViewer
- SupportsMaterialSource
- SupportsPositionAllLights
- SupportsTextureGeneration
- SupportsTweening

In addition, the following table lists the values that are set for device capabilities (Caps) for the software vertex processing mode.

Member	Software vertex processing capabilities
MaxActiveLights	Unlimited
MaxUserClipPlanes	6
MaxVertexBlendMatrices	4
MaxStreams	16
MaxVertexIndex	int(0xffffffff)

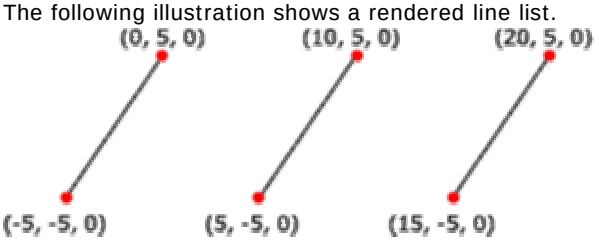
Software vertex processing provides a guaranteed set of vertex processing capabilities, including an unbounded number of lights and full support for programmable vertex shaders. You can toggle between software and hardware vertex processing at any time when using the hardware device, the only device type that supports both hardware and software vertex processing. The only requirement is that vertex buffers used for software vertex processing must be allocated in system memory, typically with the SystemMemory constant of the Pool enumeration.
Note The performance of hardware vertex processing is comparable to that of software vertex processing. For this reason, it is a good idea to provide, within a single device type, both hardware- and software-emulation functionality for vertex processing. This is not the case for rasterization, for which host processors are much slower than specialized graphics hardware. Thus both hardware- and software-emulated rasterization are not provided within a single device type. Software vertex processing is the only instance of functionality duplicated between the run time and the hardware (driver) within a single device. All other device capabilities represent potentially variable functionality provided by the driver.

Devices 7. Device-Supported Primitive Types

You can render primitive types from a managed code application with any of the Device rendering methods.

Line Lists

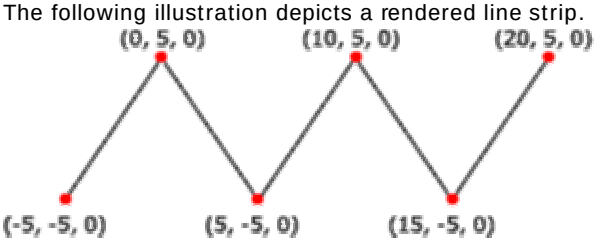
This is preliminary documentation and is subject to change.
A line list is a list of isolated, straight line segments. Line lists are useful for such tasks as adding sleet or heavy rain to a 3-D scene. Applications create a line list by filling an array of vertices, and the number of vertices in a line list must be an even number greater than or equal to two.
The following illustration shows a rendered line list.



You can apply materials and textures to a line list. The colors in the material or texture appear only along the lines drawn, not at any point in between the lines.

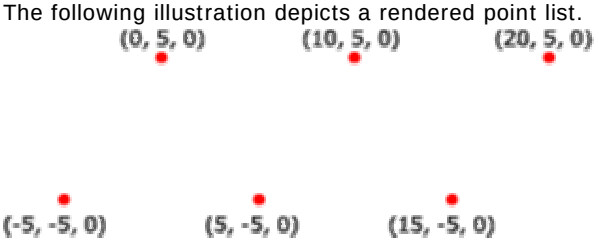
Line Strips

This is preliminary documentation and is subject to change.
A line strip is a primitive that is composed of connected line segments. Your application can use line strips for creating polygons that are not closed. A closed polygon is a polygon whose last vertex is connected to its first vertex by a line segment. If your application makes polygons based on line strips, the vertices are not guaranteed to be coplanar.
The following illustration depicts a rendered line strip.



Point Lists

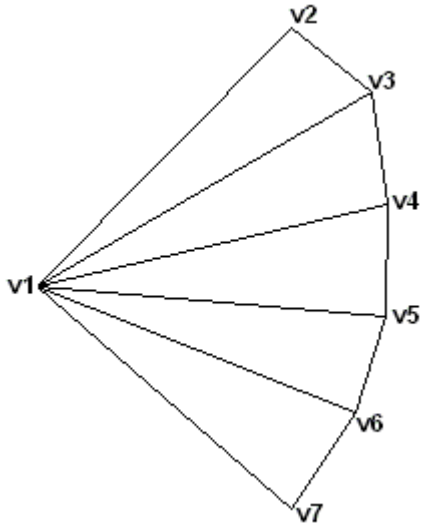
This is preliminary documentation and is subject to change.
A point list is a collection of vertices that are rendered as isolated points. Your application can use them in 3-D scenes for star fields, or dotted lines on the surface of a polygon.
The following illustration depicts a rendered point list.



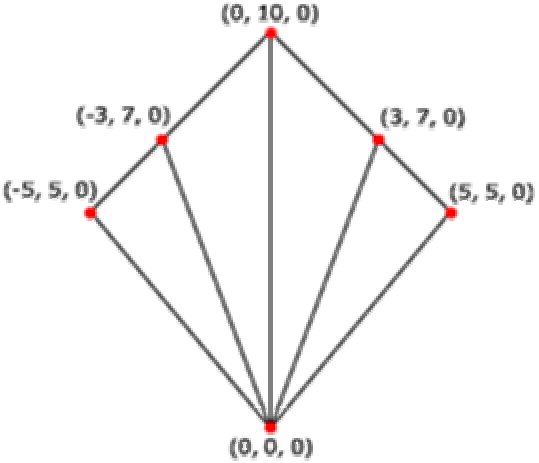
Your application can apply materials and textures to a point list. The colors in the material or texture appear only at the points drawn, and not anywhere between the points.

Triangle Fans

This is preliminary documentation and is subject to change.
A triangle fan is similar to a triangle strip, except that all the triangles share one vertex. This is shown in the following illustration.

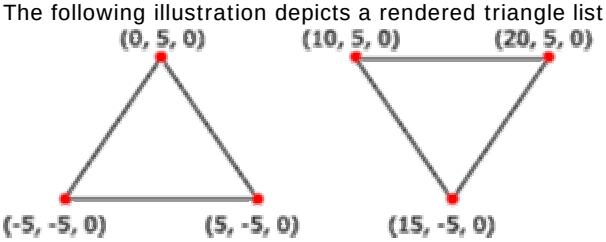


The system uses vertices v2, v3, and v1 to draw the first triangle, v3, v4, and v1 to draw the second triangle, v4, v5, and v1 to draw the third triangle, and so on. When flat shading is enabled, the system shades the triangle with the color from its first vertex.
The following illustration depicts a rendered triangle fan.



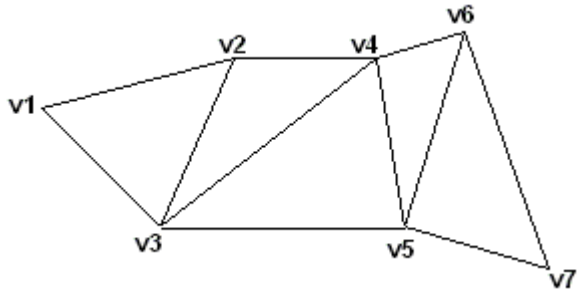
Triangle Lists

This is preliminary documentation and is subject to change.
A triangle list is a list of isolated triangles. They might or might not be near each other. A triangle list must have at least three vertices. The total number of vertices must be divisible by three.
Use triangle lists to create an object that is composed of disjoint pieces. For instance, one way to create a force-field wall in a 3-D game is to specify a large list of small, unconnected triangles. Then apply a material and texture that appears to emit light to the triangle list. Each triangle in the wall appears to glow. The scene behind the wall becomes partially visible through the gaps between the triangles, as a player might expect when looking at a force field.
Triangle lists are also useful for creating primitives that have sharp edges and are shaded with Gouraud shading. See Face and Vertex Normal Vectors.
The following illustration depicts a rendered triangle list.



Triangle Strips

This is preliminary documentation and is subject to change.
A triangle strip is a series of connected triangles. Because the triangles are connected, the application does not need to repeatedly specify all three vertices for each triangle. For example, you need only seven vertices to define the following triangle strip.



The system uses vertices v1, v2, and v3 to draw the first triangle, v2, v4, and v3 to draw the second triangle, v3, v4, and v5 to draw the third, v4, v6, and v5 to draw the fourth, and so on. Notice that the vertices of the second and fourth triangles are out of order; this is required to make sure that all the triangles are drawn in a clockwise orientation.
Most objects in 3-D scenes are composed of triangle strips. This is because triangle strips can be used to specify complex objects in a way that makes efficient use of memory and processing time.
The following illustration depicts a rendered triangle strip.