AUTODESK UNIVERSITY

AS125032

DirectContext3D: API for Displaying External Graphics in Revit

Alex Pytel
Autodesk
Alex.Pytel@Autodesk.com

Learning Objectives

- Recognize how DirectContext3D fits into the graphics pipeline and Revit API.
- Learn how to use DirectContext3D to display simple graphics.
- Understand the current limitations of DirectContext3D.
- Learn how to design substantial applications that use DirectContext3D.

Description

The DirectContext3D API exposes an interface to Revit plug-ins that allows graphics supplied by the external applications to coexist with standard content in Revit. Because external graphics can have diverse uses, external applications that utilize DirectContext3D can have a redefining effect on typical workflows in Revit. For example, displaying of external content can be viewed as an alternative to linking or importing. In Revit 2018, this API capability enables the direct display of Navisworks content in Revit as a part of the Coordination Model feature. This class will provide the necessary foundation for Revit API developers to begin using DirectContext3D to display external graphics directly in Revit views. We will discuss several examples, including Coordination Model.

AUTODESK UNIVERSITY

Contents

Overview

DirectContext3D is an API for displaying graphics in Revit by utilizing an internal graphics pipeline that accepts primitives, such as points, lines, and triangles, as input. The external graphics served by DirectContext3D applications exist separately from the host Revit document and do not need to be converted to an internal format. Using DirectContext3D to draw graphics allows external applications to avoid the performance cost associated with permanently importing content into Revit and allows the applications to have more flexibility with managing the external content. However, the applications are responsible for any optimizations that may be necessary. CoordinationModel, an internal Revit addin, uses DirectContext3D to display complex graphics served by a Navisworks component that performs sophisticated optimization of the graphics.

When to Use DirectContext3D

The main strengths of DirectContext3D are:
1. Flexibility gained from not having to import graphics as a prerequisite for drawing.
2. Simplicity gained from interaction with an internal graphics pipeline that handles many tasks related to drawing.

Due to the flexible nature of the framework, many different applications of DirectContext3D are possible. However, the following scenarios fit the design of the API especially well:
- drawing temporary graphics,
- visualization of data, and
- drawing graphics that are served by an external provider.

Getting Started with DirectContext3D

There are three main steps to getting started with DirectContext3D:
1. Understand how to implement a basic external application using the Revit API. (This is outside the scope of this document.)
2. Understand how DirectContext3D connects to Revit's graphics pipeline using callbacks and the tasks that are the responsibility of an external application that utilizes DirectContext3D.
3. Learn about the various parts of the API and their purpose.

AUTODESK UNIVERSITY

DirectContext3D Callbacks

External applications that use DirectContext3D can carry out drawing operations from callbacks that are made by Revit to the applications. The callbacks structure the use of DirectContext3D in such a way that the applications participate in appropriate stages of Revit's internal drawing process. This constraint makes it possible for geometry submitted by external applications to be drawn in the same way as geometry native to Revit. Support for the callback architecture is provided by a part of Revit API called the External Service Framework (ESF).

Drawing as Delegated Functionality

ESF is a framework that allows Revit to delegate functionality to external applications based on a system of services and servers. In this context, servers are local providers of data or other functionality that constitutes a service. When Revit requests the service to be performed, ESF calls the servers that implement the functionality corresponding to the service. External applications create and register servers for a particular service.

DirectContext3D contains an external service called DirectContext3DService that interacts with servers implementing the IDirectContext3DServer interface. The functionality delegated to the servers consists of two main tasks:

1. Drawing external graphics by making DirectContext3D calls.
2. Computing the bounding box of the external graphics.

The server interface contains two callbacks corresponding to these two tasks: RenderScene() and GetBoundingBox(). Figure 1 summarizes the relationship between DirectContext3D callbacks and ESF.

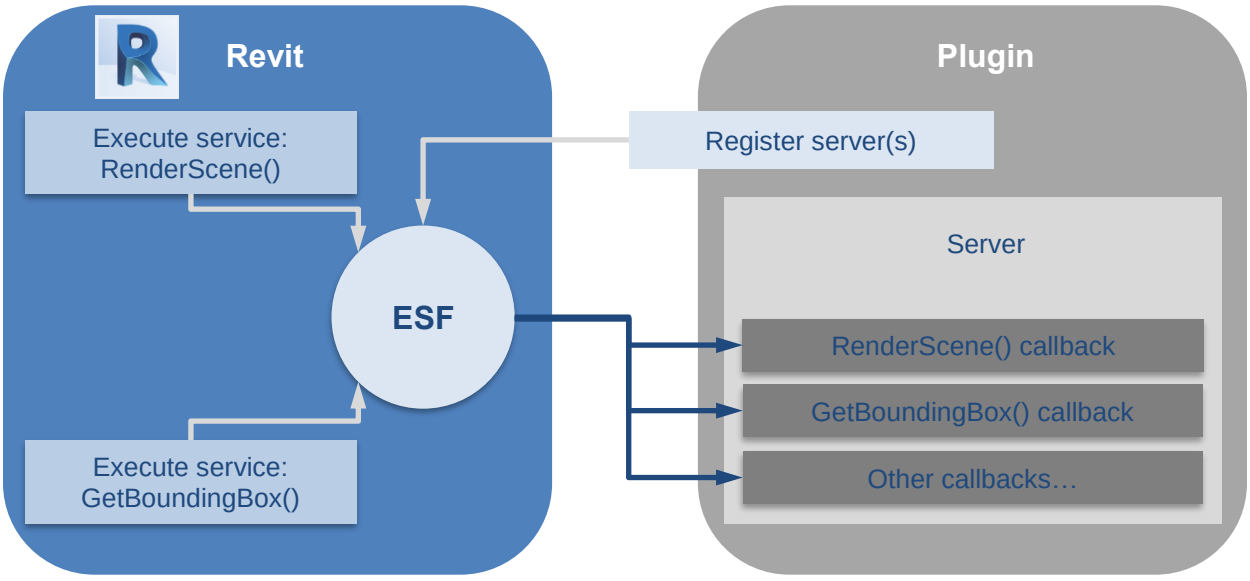

Figure 1. The relationship between DirectContext3D callbacks and ESF.

DirectContext3DService is a MultiServerService, which means that it can have multiple servers that are active simultaneously. There is no restriction on the number of external applications and how many DirectContext3D servers each of them registers. Each active server is called back by Revit to participate in the drawing process.

Implementing RenderScene()

For drawing graphics, the most important callback is RenderScene(). To implement this callback, external applications use the main part of the DirectContext3D API to execute drawing operations. The drawing process is discussed in detail in the following sections.

The graphics drawn by an external application may depend on the current state of Revit, for example the view where drawing is taking place and the display style of the view. For this reason, it is essential that an external application is able to call DirectContext3D (and other Revit API) from within the DirectContext3D callbacks, completing a bi-directional channel of communication with Revit. In particular, one of the roles of the DrawContext object of DirectContext3D is to provide rendering-related information to an application, which is executing inside RenderScene(). Figure 2 illustrates the main tasks of RenderScene() and the role of DrawContext.

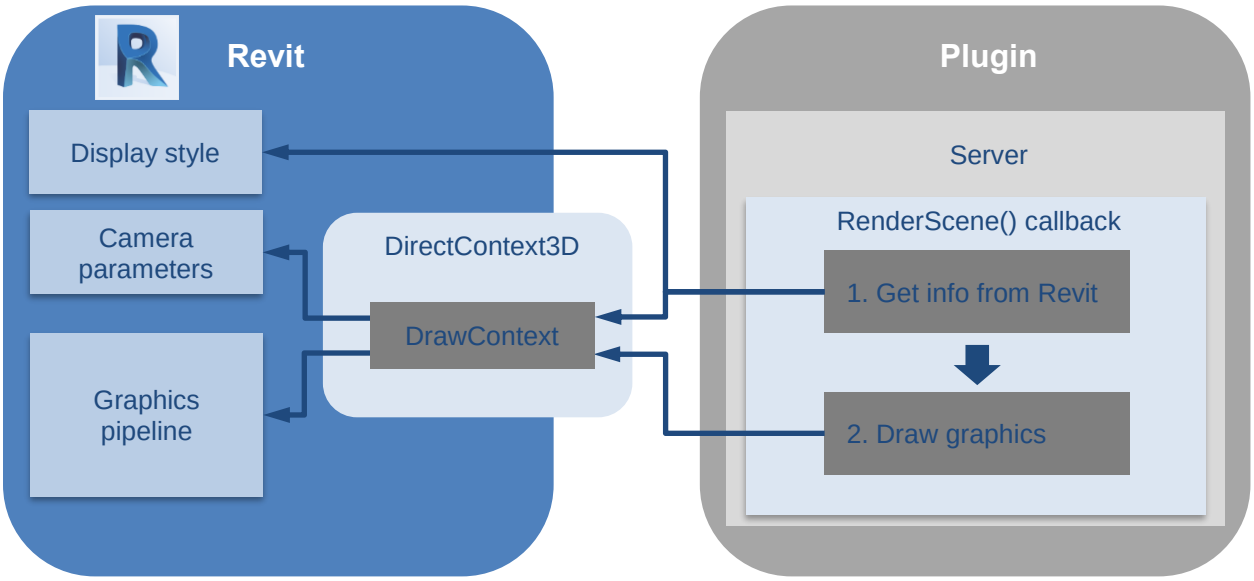

Figure 2. Communication with Revit from inside RenderScene() callback.

DirectContext3D and Graphics Pipeline in Revit

DirectContext3D is a graphics API that exposes a low-level part of the graphics pipeline used in Revit internally. To draw graphics using this pipeline external applications submit lists of graphics primitives encoded in pairs of vertex and index buffers. The callbacks of DirectContext3D constrain any interaction with the pipeline to be in lockstep with internal rendering.

Relevant Computer Graphics Concepts

This section provides some computer graphics background concerning the key aspects of the rendering pipeline that external applications can access using DirectContext3D.

Overview of the Drawing Process

To support rendering of diverse 3D scenes, a graphics API and the drawing process that it represents must be general-purpose. For this reason, graphics packages share a lot of similarities and it is possible to identify certain typical elements of an abstract drawing process in a given API. DirectContext3D belongs to a class of APIs that produce graphics on the screen from geometric primitives, such as triangles, that are specified in terms of the positions of their vertices. These geometric shapes are converted into corresponding pixel-based representations in a process called rasterization. The output of rasterization is stored in 2D buffers, which are similar to bitmap images, and is later presented on the screen.

There can be many parameters that determine the outcome of the drawing process. The simplest parameters are the positions and colors of the primitive shapes. However, graphics APIs typically implement many drawing operations that can be enabled optionally and whose parameters can be controlled. For example, the rasterized shapes can be blended into the output buffer according to their transparency, instead of overwriting the destination values. There can also be additional drawing operations that are concerned with manipulating the output buffers, rather than the initial geometric shapes. Some drawing operations can be programmable, i.e., specified using a standalone program that is executed at the time of drawing.

The complexity of a graphics API can be managed by organizing the supported drawing operations into a graphics pipeline, in which each stage can be controlled independently. Optional stages, such as blending for transparent objects, can be enabled only when appropriate objects are submitted into the pipeline for drawing. The act of drawing an object in the output buffer can be considered to begin with setting of the parameters that control the state of the entire pipeline. Although it may be possible for certain state to persist from previous drawing operations, it is important to understand that a given object will be drawn in the intended way only if the total state of the pipeline is correct.

DirectContext3D is designed along the same lines as many other APIs for drawing graphics to the screen. The API represents a simple pipeline that accepts an encoding of geometric shapes as input and rasterizes them according to the state of the pipeline. However, there is one fundamental difference that complicates the design of DirectContext3D and how it operates: the graphics pipeline is implemented inside of Revit and DirectContext3D is an additional interface level that allows external applications (i.e., Revit plugins) to execute drawing operations using the internal pipeline. The complexity of drawing in Revit necessitates that many pieces of the internal pipeline, for example operations on the output image buffers and the programmable stages, are

not exposed through DirectContext3D. On the other hand, these limitations make DirectContext3D simpler to use.

Transformations

To control the position and orientation of a geometric primitive during drawing, it is necessary to apply a transformation to its vertices. For example, adding a fixed offset to the vertices will translate a shape in a specified direction. Typically, there are several levels of transformations applied to a given object (see Figure 3).

- **Modeling transformations.** These transformations serve to place a given shape in relationship to others inside the virtual world space. The sizes and positions of shapes in the world space can be given meaning in terms of real world units.
- **View transformations.** The virtual scene can be transformed so that it can be presented from a specific viewpoint, which is referred to as an eye or camera. Movements of the camera correspond to transformations applied to the world. For example, moving the camera to the left means that the world moves to the right. (The transformation of the camera is the inverse of the associated transformation of the world.)
- **Projection transformations.** To display a 3D scene on a 2D screen requires a projection transformation to be applied. Often two types of projection transformations are implemented: orthographic and perspective.

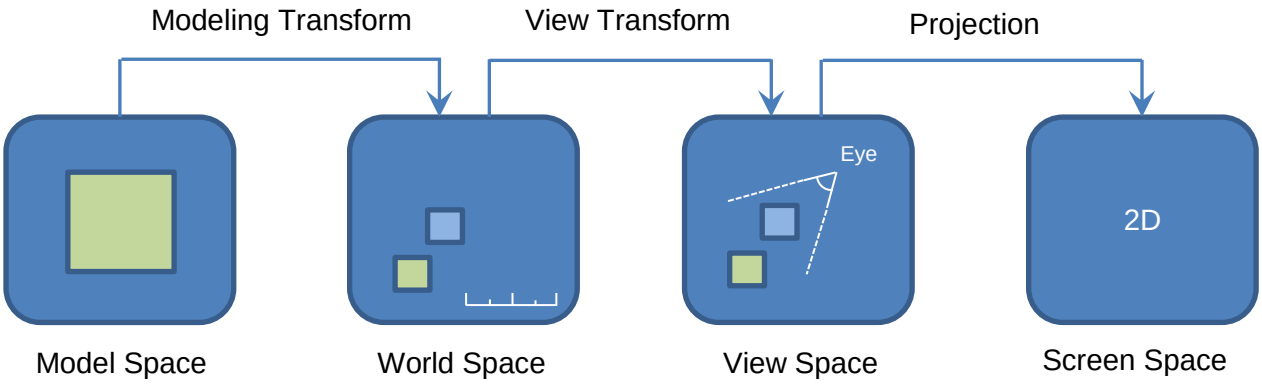

Figure 3. Transformations and Spaces.

DirectContext3D exposes only a single type of transformation, which is referred to as the World transformation (DrawContext.SetWorldTransform()). This transformation is applied to all geometric primitives that are submitted for rendering (before any camera transformations) and should be updated as each object is submitted through the API. The World transformation of DirectContext3D belongs with modeling transformations in the above classification. The rest of the transformations are controlled by Revit. In particular, whether the Revit view is an orthographic or perspective view determines the projection transformation.

Depth Buffer

To render a 3D scene means to produce a 2D image in which only the visible parts of the objects in the scene are displayed. In a graphics pipeline that is based on rasterization this effect, called hidden surface elimination, is typically achieved using a depth buffer. The depth buffer is an additional 2D image that is produced by the pipeline during rasterization and contains the depth value of each pixel. Comparing depth values during rasterization of each objects allows the pipeline to discard occluded pixels and determine the visible parts of geometry with pixel-level precision.

The depth buffer hidden surface elimination algorithm can process objects in any order and can correctly update the 2D pixel representation of the scene when additional objects are submitted. DirectContext3D benefits from the depth buffer algorithm by having the algorithm combine rasterized objects drawn by Revit in a given view with the additional objects submitted through the API. The result is a composite image containing objects drawn by Revit and objects drawn by external applications using DirectContext3D.

DirectContext3D as an Interface to A Graphics Pipeline

The main component of DirectContext3D is an interface to a graphics pipeline that accepts an encoding of geometric shapes as input and rasterizes them according to the state of the pipeline (e.g. the world transformation that is set). Compared to a typical graphics API, there are many actions that do not need to be taken or configured using DirectContext3D, because the target of rendering is a Revit view and the work is done inside of Revit. First, it is not necessary to perform any setting up of output image buffers. Second, view navigation in Revit determines most of the transformations that need to be applied to the geometry submitted through DirectContext3D. Third, complex multi-pass rendering operations on submitted geometry and screen-space post-processing effects operating on the output image buffers are performed internally. Similarly, control over advanced materials, shading effects, and texturing is not exposed through DirectContext3D.

The following sequence of actions lists the basic tasks for which DirectContext3D applications remain to be responsible when using the graphics pipeline:

1. Encode geometry in a set of vertex and index buffers.
2. Set the transformation that the pipeline will use to transform the geometry.
3. Control the color and shading of the geometry using an EffectInstance object.
4. Submit the geometry into the pipeline.

External applications should also be aware of the following provisions of DirectContext3D that are related to submitting graphics into the pipeline:

- External applications can draw transparent objects by submitting them separately into a dedicated rendering pass.
- DirectContext3D provides a callback, GetBoundingBox(), that allows external applications to report the outline of the graphics that they submit. This provision allows Revit views that contain external graphics to be navigated consistently with other Revit views.
- Camera parameters and other rendering-related information can be queried with the help of the DrawContext object. This information is useful for an external application that needs to optimize the graphics that it submits into the pipeline.

The following subsection provides additional details regarding the geometry encoding, so that the rest of the discussion is grounded with a definition of what constitutes input to the graphics pipeline used with DirectContext3D.

Vertex and Index Buffers

DirectContext3D supports rendering of three types of primitives: triangles, lines, and points. Each type of primitive is defined in terms of vertices. A geometric object that can be submitted using one drawing operation consists of several primitives of the same type. An object consisting of triangles is referred to as a (triangle) mesh.

A given vertex in a mesh can be shared among several triangles. The number of vertices submitted for processing can be substantially reduced if a vertex is listed once in a vertex buffer and referenced as many times as required using an index buffer. So, the vertex buffer lists the vertices of an object in any order, while the index buffer contains indices into the vertex buffer. In the case of the triangle primitive, the index buffer contains a set of three indices for each triangle.

In DirectContext3D a vertex is defined by its position (three floating point values) and one or both of two optional attributes: normal vector (three values) and color (one 32-bit value containing an integer that encodes four color components). Note that the attributes increase the storage requirements for a vertex and increase the benefit of shared vertex reuse due to the index buffer. However, the number of unique vertices may increase for an object if vertices may no longer be shared, because a vertex with the same position has different attribute values in the primitives that are incident on it.

To give a concrete example, the vertex buffer may look like: 0, 0, 0, 1, 0, 0, 0, 1, 0 (each value is a floating point number). If the vertex format is VertexFormatBits.Position and the primitive is triangle, then each triple of values defines one vertex and there are three vertices. If the vertex format is VertexFormatBits.PositionNormal, there will be an additional set of three values following each position. In other words, the vertex buffer will look like: p0, n0, p1, n1, p2, n2 (p = position, n = normal). The index buffer for one triangle may look like: 0, 1, 2. Each additional triangle requires an additional triple of indices.

Accessing the Graphics Pipeline Using DirectContext3D

Revit calls external applications twice to contribute their geometry to the graphics pipeline. Execution enters into the external applications when it is time to submit opaque model geometry and the process is repeated a second time for transparent model geometry. A DirectContext3D application submits both opaque and transparent geometry from the RenderScene() callback. The two calls are distinguished by the value of DrawContext.IsTransparentPass(). Other than submitting the transparent geometry no further steps, such as management of blending, need to be taken by the external application. The results of the two rendering passes are composited by Revit internally. Figure 4 illustrates the two rendering passes and the actions that external applications need to take from the RenderScene() callback.

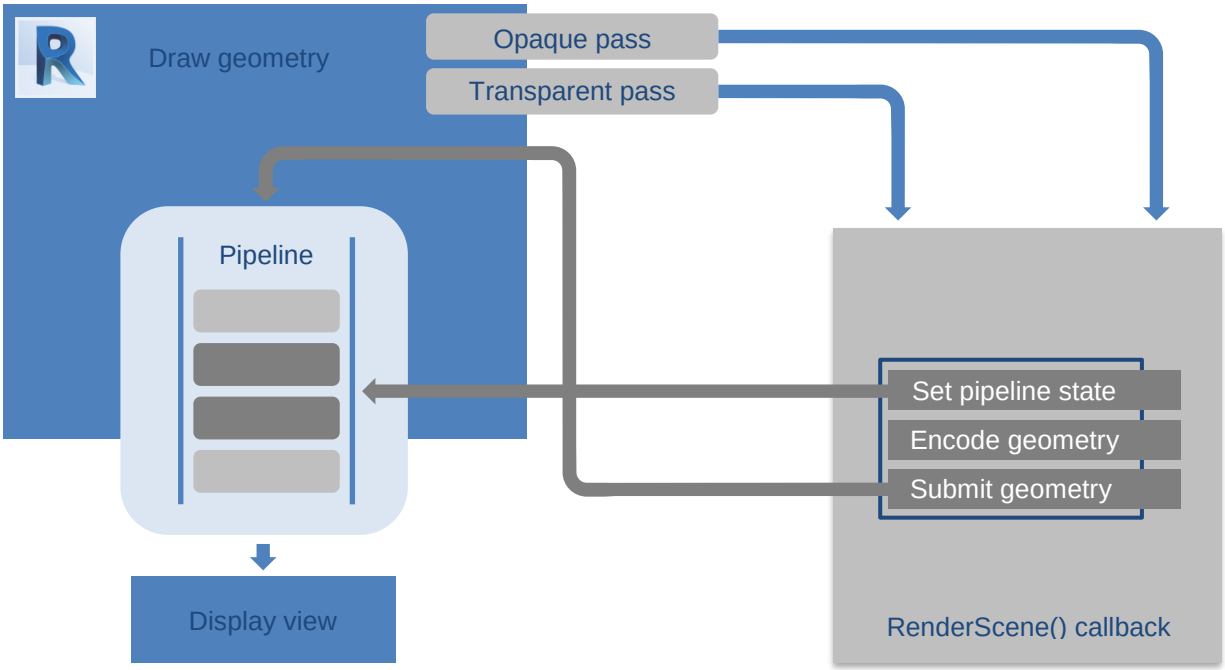

Figure 4. Access to the graphics pipeline from RenderScene().

External applications can take actions that access the graphics pipeline only when Revit is in the middle of its internal rendering pipeline and calls RenderScene(). The restriction extends to creation of objects such as VertexBuffer that are used in drawing operations. Applications can test that they are executing in the scope that permits access to the graphics pipeline by checking the value of DrawContext.IsAvailable(). DrawContext ties together a lot of the DirectContext3D functionality related to drawing and acts as an interface for collecting certain rendering-related information from Revit, such as the camera parameters. Rendering-related parameters also have meaning only when Revit is in the appropriate state internally, so that execution of calls such as DrawContext.GetCamera() is subject to the constraint represented by IsAvailable().

Revit also maintains outlines, or bounding boxes, of the geometry contained in a view for the purposes of view navigation and internal optimizations. Therefore, DirectContext3D applications have an additional responsibility to compute the outline of the geometry that they submit. Revit obtains the value using the GetBoundingBox() callback. For performance reasons, it is recommended that external applications cache the value of the outline after computing it (and report the cached value thereafter). Also note that GetBoundingBox() can be called on a different thread than RenderScene(). Figure 5 illustrates the GetBoundingBox() callback.

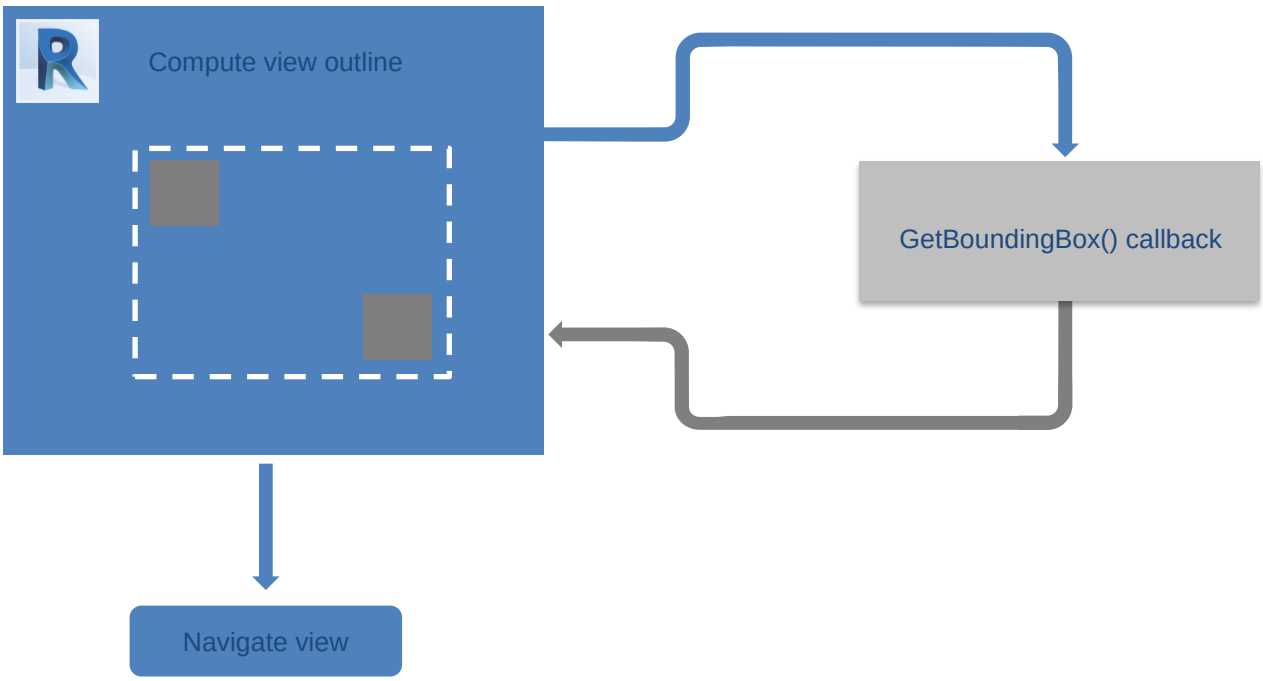

Figure 5. Execution of the GetBoundingBox() callback.

AUTODESK UNIVERSITY

Utilizing DirectContext3D

DirectContext3D allows an external application to participate in Revit's internal drawing process by acting as a provider of additional objects for rendering. The format of the external geometry is a list of geometric primitives, such as triangles, encoded in pairs of vertex and index buffers. A server implementing the IDirectContext3DServer interface provides the communication channel, which is necessary for the application to request display-related information from Revit and execute appropriate drawing operations. Revit calls certain methods of the server interface when it is in the correct state for communicating with the external application.

Main Steps for Drawing

Before drawing requests can be issued, an external application must instantiate and register a server derived from IDirectContext3DServer. This can be done as early as inside OnStartup() of IExternalApplication.

```
ExternalServiceId serviceId =
ExternalServices.BuiltInExternalServices.DirectContext3DService;
m_serviceDC3D = ExternalServiceRegistry.GetService(serviceId) as MultiServerService;

m_testServer = new DC3DTestServer();
IList<Guid> activeList = m_serviceDC3D.GetActiveServerIds();
activeList.Add(m_testServer.GetServerId());

m_serviceDC3D.AddServer(m_testServer);
m_serviceDC3D.SetActiveServers(activeList);
```

The server can begin to interact with Revit's graphics pipeline as soon as a document is opened and a view is displayed. Without extra work from the application and server, the graphics submitted by the server will be drawn in most types of open Revit views.

There are two sets of methods that a server implementation must override: those inherited from IExternalServer and those inherited from IDirectContext3DServer. One of the key points is identifying the server via a GUID.

```
public DC3DTestServer()
{
    m_serverId = Guid.NewGuid();
}

virtual public Guid GetServerId() // inherited from IExternalServer
{
    return m_serverId;
}
```

The main work performed by the server is inside RenderScene() and GetBoundingBox(). A detailed example is provided further below. A possible sequence of calls inside RenderScene() may look like this in pseudocode:

```
Use View and DisplayStyle arguments to discover properties of the view.
Use DrawContext methods such as GetCamera() to get additional information.
If allocating new buffers
```

```
    Calculate buffer sizes and construct: VertexBuffer, IndexBuffer, VertexFormat.
    Given vertex format, construct a list of vertices for the geometry.
    Fill vertex and index buffers.
    While iterating over vertices, update the outline of the geometry.
    Unmap buffers.
If a new EffectInstance is needed, construct it.
Call DrawContext.FlushBuffer().
```

A server can re-use objects such as VertexBuffer and EffectInstance from one invocation of RenderScene() to another, which is better for performance. However, the objects can become invalid due to a change in low-level graphics state internal to Revit. A DirectContext3D server must check the validity of objects using their IsValid() methods and be prepared to have to re-create them. In a minimal application, the code can be simpler if DirectContext3D objects are not allowed to persist between the calls.

At the time when a server needs to generate the contents of vertex and index buffers, it is also convenient to compute the outline of the geometry, which is needed for GetBoundingBox(). When the outline is already computed, GetBoundingBox() can simply return it. However, there is a complication: a server should expect that GetBoundingBox() and RenderScene() can be called in any order and on different threads. So, a server should be able to compute the outline of its geometry independently of drawing it. Synchronization may be needed to avoid a race condition when shared state is accessed simultaneously from the two callbacks.

DirectContext3D Server Interface Methods

A DirectContext3D server inherits the following methods from the IExternalServer interface. These methods characterize the server as a component of the External Service Framework (ESF).

`Guid GetServerId()`: The GUID that identifies the server.
`String GetVendorId()`: The vendor of the server.
`ExternalServiceId GetServiceId()`: See below.
`String GetName()`: The name of the server.
`String GetDescription()`: The description of the server.

GetServiceId() should be implemented like this:

```
virtual public ExternalServiceId GetServiceId()
{
    return ExternalServices.BuiltInExternalServices.DirectContext3DService;
}
```

The second set of interface methods comes from the IDirectContext3DServer interface and contains callbacks that the server should use to submit geometry for drawing.

`bool CanExecute(View view)`: allows the server to prevent its callbacks (UseInTransparentPass(), GetBoundingBox(), and RenderScene()) from being executed for the specified view.
`bool UseInTransparentPass(View view)`: allows the server's RenderScene() to be called an additional time, so that the server can draw transparent geometry.

`Outline GetBoundingBox(View view)`: allows the server to specify the outline of the geometry that it submits for the view. Note that the view argument can be null, in which case the server is expected to return a view-independent outline. The outline should be transformed in the same way as the corresponding geometry if the server uses DrawContext.SetWorldTransform().
`void RenderScene(View view, DisplayStyle displayStyle)`: allows the server to submit geometry for rendering.
`String GetApplicationId()`: used for internal addins only. The server should return the empty string.
`String GetSourceId()`: used for internal addins only. The server should return the empty string.
`bool UsesHandles()`: used for internal addins only. The server should return false.

DirectContext3D Objects

VertexFormatBits. Numerical representation of the format of a vertex. Vertices always have position and can optionally have a normal vector, a color, or both a normal vector and a color. VertexFormatBits values are: Position, PositionNormal, PositionColored, and PositionNormalColored.

VertexFormat. Specifies the format of vertex data in a vertex buffer. Used when calling FlushBuffer(). Can become invalid in which case the external application must recreate the object.

Vertex. The base class for VertexPosition, VertexPositionNormal, VertexPositionColored, and VertexPositionNormalColored. Each derived class contains the data necessary to represent one vertex of the corresponding format in 3D space. Objects of these classes can be used with DirectContext3D vertex streams to fill vertex buffers.

VertexStream. Has the following subclasses: VertexStreamPosition, VertexStreamPositionNormal, VertexStreamPositionColored, and VertexStreamPositionNormalColored. See below for an explanation of the functionality provided by DirectContext3D vertex streams.

VertexBuffer. A buffer that stores vertex data for rendering. Must be mapped before it can be written to and unmapped before it can be submitted using FlushBuffer(). Can become invalid in which case the external application must recreate the object and its contents.

PrimitiveType. Numerical representation of the type of primitive. PrimitiveType values are: TriangleList, LineList, PointList.

IndexPrimitive. The base class for IndexPoint, IndexLine, and IndexTriangle. Similar to the vertex classes, but each derived class represents one primitive (point, line, or triangle) in the index buffer. Objects of these classes can be used with DirectContext3D index streams to fill index buffers.

IndexStream. Similar to VertexStream. Has subclasses: IndexStreamPoint, IndexStreamLine, and IndexStreamTriangle.

IndexBuffer. Similar to VertexBuffer. Can become invalid.

EffectInstance. Determines the appearance of geometry when it is drawn. Used when FlushBuffer() is called. Can become invalid in which case the external application must recreate the object and its contents.

DrawContext. The central component of the DirectContext3D API, consisting of static methods that can be used to submit geometry for drawing and obtain information about the state of the view where rendering is occurring. DrawContext enforces the constraint that drawing-related operations can occur only when Revit is in the appropriate state internally.

Using Streams to Fill Buffers

The functionality of DirectContext3D streams is an optional feature that is intended to make it easier to get started with using the API. The streams are adapters for VertexBuffer and IndexBuffer objects that insert vertices and indices into the buffers, respectively. When inserting data into the buffers, which are flat arrays, the streams minimize the opportunity for errors by keeping track of the number of data needed for each complete object being written. Additionally, a stream keeps track of the current state of a buffer, such as the current position for writing to the buffer, as well as its capacity and mapped status.

For example, a VertexBuffer is an array of floats that can contain vertices of one format, such as PositionNormal. To write one vertex of this format into the vertex buffer it is necessary to write six floats. (The size of the vertex can be determined by calling the static function VertexPositionNormal.GetSizeInFloats().) This write operation can be performed by using a stream object as follows:

1. Obtain an instance of VertexStreamPositionNormal from the VertexBuffer instance.
2. Create an object of type VertexPositionNormal, which contains the data corresponding to the vertex.
3. Call addVertex() of the VertexStreamPositionNormal stream object, which will only accept an object of type VertexPositionNormal as an argument.
4. Repeat steps 2 and 3 for additional vertices.

The effect of this process is to write data into the buffer associated with the stream one float at a time. However, static type checking ensures that vertices of the intended format are written. Even more, using streams ensures that vertices are written without overlap and that the correct number of data is written for each vertex.

The alternative to using streams is to obtain the address of the mapped buffer using GetMappedHandle() and to write data into the memory, while keeping track of the remaining capacity of the buffer. This unstructured way of filling a buffer may be more appropriate than using streams in some situations. For example, the Coordination Model plugin relies on an external library to supply the contents of vertex and index buffers. Providing the library with a memory address for writing the data (as well as the size limit of each buffer) keeps the interface between the plugin and the library simple.

Using VertexFormat and EffectInstance

Solid geometry rendered using DirectContext3D can be shaded in several ways depending on its vertex format and the effect instance employed. As a server prepares geometry for rendering, it must determine the following:

1. The source of the color that the geometry can use.
2. The type of shaded appearance applicable to the geometry.
3. The combination of VertexFormat and EffectInstance that will determine the final appearance of the geometry.

For the first choice, the color (and transparency) can be specified in the effect instance for a whole piece of geometry submitted in one invocation of FlushBuffer(). Alternatively, the color can be specified for each vertex individually, in which case the color in the interior of each triangle will be linearly interpolated during rasterization. For the latter option to be available, vertex data must include the color attribute, so that the format of the vertices should be either PositionColored or PositionNormalColored.

For the second choice, there are two possible types of shaded appearance for the geometry, corresponding to either the shaded display style or the consistent colors display style. In the shaded display style, surfaces are lit according to the direction of their normals, while in the consistent colors style surfaces have a flat appearance. For the shaded appearance to be applicable to a piece of geometry, its vertices must provide the normal attribute, so that the vertex format should be either PositionNormal or PositionNormalColored. The different types of shading are illustrated in Figure 6.

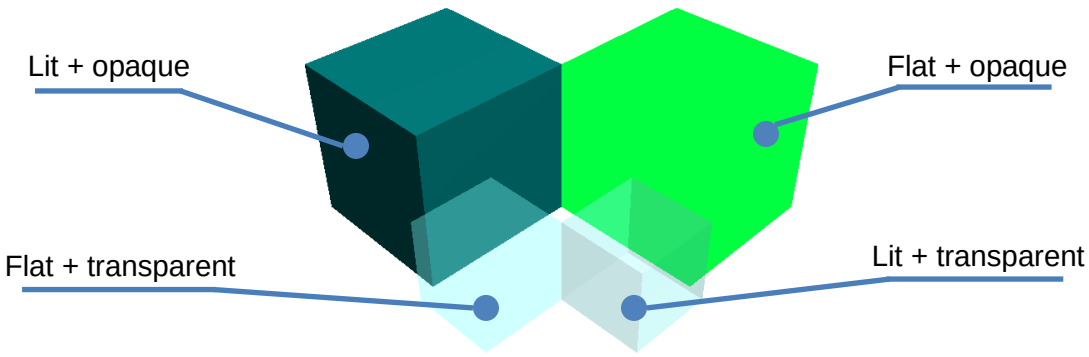

Figure 6. Different types of shaded appearance.

The final appearance of the geometry is determined when it is submitted using FlushBuffer() and a combination of VertexFormat and EffectInstance. VertexFormat must exactly describe the format of the vertex data in the vertex buffer and determines which display options are available for the piece of geometry. A specific display option is selected according to the vertex format setting that has been provided to the constructor of EffectInstance. Therefore, it is possible to store vertex data in a certain format and to draw it with different shading options by flushing the vertex buffer with an appropriate effect instance. For a combination of VertexFormat and EffectInstance to be valid, EffectInstance.MatchesFormat(vertexFormat) must be true. The

following table lists the results of different VertexFormat and EffectInstance pairings (P = Position, N = Normal, C = Colored). The last row of the table is also visualized in Figure 7.

		Vertex format specified for EffectInstance			
		P	PN	PC	PNC
V. format in buffer	P	flat	invalid	invalid	invalid
	PN	flat	shaded	invalid	invalid
	PC	flat	invalid	flat, vert. color	invalid
	PNC	flat	shaded	flat, vert. color	shaded, vert. color

There are two different sets of color-related properties that apply to an effect instance depending on the vertex format provided in the object's constructor. (However the properties may have no effect if the colors come from the vertices.) The flat shaded appearance has only one color parameter. The lit appearance has the following parameters: ambient color, diffuse color, specular color, and glossiness. In both cases, the transparency value can also be set. Transparency is further discussed in the following section, which also includes an example of setting the other parameters of an effect instance.

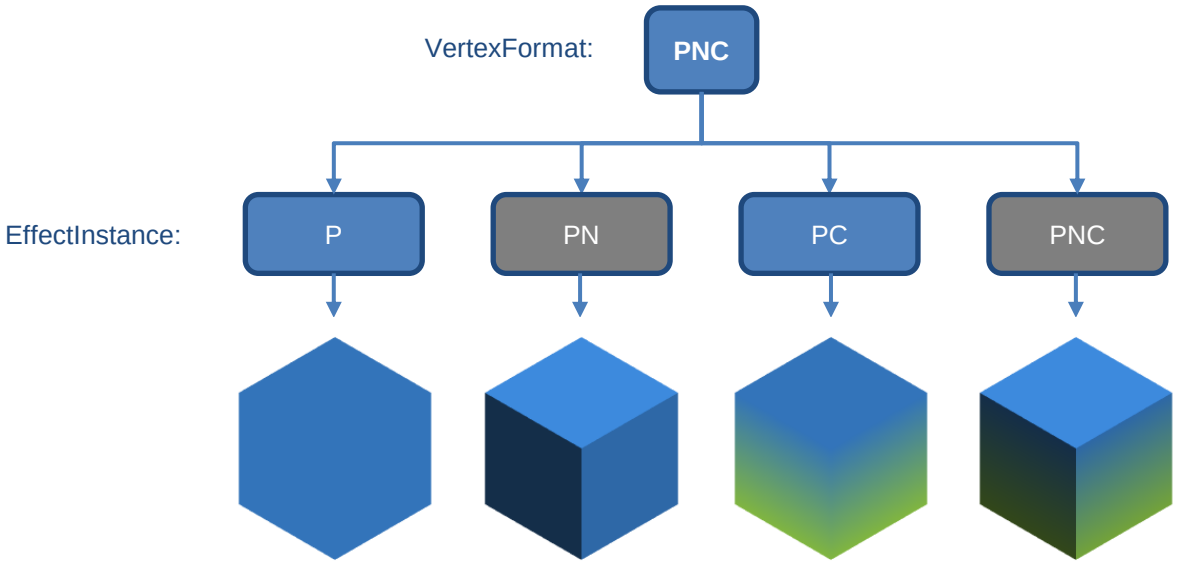

Figure 7. Using VertexFormat and EffectInstance.

There are two minor limitations that reduce the number of options that a server has when selecting a VertexFormat and EffectInstance for its geometry. First, shaded objects can not be correctly displayed in a view whose display style is consistent colors, or hidden line. To avoid this situation a server should check the display style argument to its RenderScene() method and fall back to using an EffectInstance that is set up for a flat appearance. Second, the ambient shadows graphics display option requires that normals are provided even for geometry that should be displayed using flat shading. Therefore, it may be simpler for a server to always limit itself to using the PositionNormal and PositionNormalColored formats only.

Using Transparency and Color

Each time a transparent object is drawn, pixels in the output area covered by the transparent objects are adjusted according to an operation called blending. To perform blending, Revit draws transparent objects in a separate pass after opaque objects. For external applications using DirectContext3D management of these two passes is simplified, so that an external application can draw transparent objects by submitting them in the same way as opaque objects. In other words, drawing transparent objects is equivalent to submitting an additional scene with only transparent objects in it.

To be able to submit the transparent objects, the external application should have an IDirectContext3DServer that returns true from UseInTransparentPass(View view). The server's RenderScene() call back will be executed an additional time by Revit. The two calls of RenderScene() for opaque and transparent objects can be distinguished by testing DrawContext.IsTransparentPass().

The application can control the degree of transparency of an object using its color. In graphics, it is common to specify color using four components: red, green, blue, and alpha. Each color component is an 8-bit unsigned integer ranging in value from 0 to 255. Alpha determines transparency with the value 255 being fully transparent. With the addition of DirectContext3D to Revit API a new color class is provided, ColorWithTransparency, that allows manipulation of all four of its components. The components are stored in a single 32-bit integer in the order ABGR, with A being the highest byte. However, the meaning of the components can be interpreted differently in certain usage scenarios.

Scenario	Order of components	Transparency
(1) Using ColorWithTransparency	ABGR	255 is fully transparent
(2) Specifying color attribute of vertices stored in vertex buffers	ARGB	0 is fully transparent
(3) Setting effect instance properties	ABGR	255 is fully transparent

When buffer contents are filled out using DirectContext3D streams the conversion needed by scenario 2 is carried out automatically by DirectContext3D.

Scenario 3 is the situation when an effect instance is used to specify the color and transparency of geometry submitted in a single invocation of FlushBuffer(). The color-related properties of an effect instance can be updated as shown below. Note that the range of the transparency value is adjusted for the call to SetTransparency().

```
VertexFormatBits format;
ColorWithTransparency color;
...
switch (format)
{
   case VertexFormatBits.Position:
        effect.SetColor(color.GetColor());
        effect.SetTransparency(color.GetTransparency() / 255.0);
        break;
   case VertexFormatBits.PositionColored: break;
   case VertexFormatBits.PositionNormal:
        effect.SetAmbientColor(color.GetColor());
```

```
            effect.SetDiffuseColor(color.GetColor());
            effect.SetTransparency(color.GetTransparency() / 255.0);
            break;
        case VertexFormatBits.PositionNormalColored: break;
            default: break;
}
```

Example: Drawing a Cube

The low-level nature of DirectContext3D provides for many options for defining the format and primitives of the geometry that is submitted for drawing. In the following example, a cube is defined as a mesh containing 12 triangles (6 faces with 2 triangles each). Additionally, the vertices of the cube are paired with normal vectors corresponding to the faces. Thus 8 vertices of the geometric cube appear in 3 faces each and have a different normal in each face. So, the total number of distinct vertices in the mesh is 24.

```
using Autodesk.Revit.DB.DirectContext3D;
public const int NumVertices = 24; // 8 vertices with 3 choices of normal vector
public const PrimitiveType Primitive = PrimitiveType.TriangleList;
public const int NumPrimitives = 12; // 6 faces, 2 triangles each
```

The vertex and normal data can be stored in arrays for convenience:

```
XYZ[] vertexNormals = new XYZ[]
{
   new XYZ(0.0, 0.0, 1.0), //+Z normal
   // . . . 6 entries total
};

XYZ[] vertexPositions = new XYZ[NumVertices]
{
   new XYZ( // . . .
   // . . . 24 entries total
};
```

The next step is to create and map a vertex buffer and an index buffer. The arithmetic for computing the capacity of the buffers is structured using some helper functions.

```
public static int GetPrimitiveSize(PrimitiveType primitive)
{
    switch (primitive)
    {
       case PrimitiveType.LineList: return IndexLine.GetSizeInShortInts();
       case PrimitiveType.PointList: return IndexPoint.GetSizeInShortInts();
       case PrimitiveType.TriangleList: return IndexTriangle.GetSizeInShortInts();
       default: break;
    }
    return IndexTriangle.GetSizeInShortInts();
}
```

```
public static int GetVertexSize(VertexFormatBits format)
{
    switch (format)
    {
       case VertexFormatBits.Position: return VertexPosition.GetSizeInFloats();
       case VertexFormatBits.PositionColored: return
VertexPositionColored.GetSizeInFloats();
       case VertexFormatBits.PositionNormal: return
VertexPositionNormal.GetSizeInFloats();
       case VertexFormatBits.PositionNormalColored: return
VertexPositionNormalColored.GetSizeInFloats();
       default: break;
    }
    return VertexPosition.GetSizeInFloats();
}

// . . .

int NumIndices = GetPrimitiveSize(Primitive) * NumPrimitives;
VertexBuffer vertexBuffer = new VertexBuffer( GetVertexSize(
VertexFormatBits.PositionNormal) * NumVertices);
IndexBuffer indexBuffer = new IndexBuffer(NumIndices);

vertexBuffer.Map(GetVertexSize(VertexFormatBits.PositionNormal) * NumVertices);
indexBuffer.Map(NumIndices);
```

Next the pair of buffers can be filled using vertex and index streams. Although only the PositionNormal format is used in this example, the following code is set up to handle the other formats. The handling of streams associated with different formats is only a suggestion that shows how to avoid type casting, which can clutter the code in a larger application that uses multiple formats.

```
switch (format)
{
    case VertexFormatBits.Position: VertexStreamP =
vertexBuffer.GetVertexStreamPosition(); break;
    case VertexFormatBits.PositionColored: VertexStreamPC =
vertexBuffer.GetVertexStreamPositionColored(); break;
    case VertexFormatBits.PositionNormal: VertexStreamPN =
vertexBuffer.GetVertexStreamPositionNormal(); break;
    case VertexFormatBits.PositionNormalColored: VertexStreamPNC =
vertexBuffer.GetVertexStreamPositionNormalColored(); break;
    default: break;
}

switch (primitive)
{
    case PrimitiveType.LineList: IndexStreamL = m_indexBuffer.GetIndexStreamLine();
break;
    case PrimitiveType.PointList: IndexStreamP = m_indexBuffer.GetIndexStreamPoint();
break;
    case PrimitiveType.TriangleList: IndexStreamT =
m_indexBuffer.GetIndexStreamTriangle(); break;
    default: break;
}
```

```
// . . .

for (int vertex = 0; vertex < NumVertices; ++vertex)
{
   XYZ vertexPosition = vertexPositions[vertex];
   XYZ vertexNormal = vertexNormals[vertex / 4];

   VertexStreamPN.AddVertex(new VertexPositionNormal(vertexPosition, vertexNormal));
}

// Stitch the vertices in the vertex buffer into triangles.

for (int indexSet = 0; indexSet < 3; ++indexSet)
{
   int indexBase = 8 * indexSet;

   IndexStreamT.AddTriangle(new IndexTriangle(0 + indexBase, 2 + indexBase, 3 +
indexBase));
   IndexStreamT.AddTriangle(new IndexTriangle(0 + indexBase, 3 + indexBase, 1 +
indexBase));

   indexBase += 4;

   IndexStreamT.AddTriangle(new IndexTriangle(0 + indexBase, 3 + indexBase, 2 +
indexBase));
   IndexStreamT.AddTriangle(new IndexTriangle(0 + indexBase, 1 + indexBase, 3 +
indexBase));
}
```

When the buffers are unmapped they are ready to be submitted for drawing. However, FlushBuffer() needs one additional piece of information, which is the EffectInstance.

```
vertexBuffer.Unmap();
indexBuffer.Unmap();

EffectInstance effect = DC3DGraphics.GetEffectInstance(VertexFormatBits.PositionNormal);
ColorWithTransparency color = new ColorWithTransparency(0, 0, 100, 0);

effect.SetAmbientColor(color.GetColor());
effect.SetDiffuseColor(color.GetColor());
```

The final step is to call FlushBuffer().

```
DrawContext.FlushBuffer(vertexBuffer, NumVertices, indexBuffer, NumIndices, new
VertexFormat(VertexFormatBits.PositionNormal), effect, Primitive, 0, NumPrimitives);
```

Management of DirectContext3D objects can depend on the application. For example, an application that expects to support all of the vertex formats can allocate a VertexFormat of each type and use the four objects in a centralized manner.

A DirectContext3D server also needs to compute the outline of the geometry that it submits. In the case of a cube, the outline can be determined from knowing that the geometry is a cube of a pre-determined size. More generally, the outline of a triangle mesh can be determined by adding each vertex of the mesh to a growing outline. Note that if SetWorldTransform() is used, the server must return the transformed outline from GetBoundingBox().

Current Features and Limitations

Revit and DirectContext3D share a fundamental connection in terms of the graphics pipeline that makes it possible for DirectContext3D graphics to be combined with graphics generated internally by Revit. This relationship allows many of the display-related features of Revit to work seamlessly with DirectContext3D graphics. However, DirectContext3D is limited to a subset of Revit's functionality and there are features of Revit that would be too costly in terms of performance to be used with DirectContext3D. Additionally, some features of Revit that are fundamentally compatible with DirectContext3D are not currently exposed through the API, because the features require a lot of configuration and management of external sources of data (e.g., textures).

DirectContext3D-compatible Display Features of Revit

Composition of graphics in the same view. DirectContext3D and Revit graphics are composited using the same depth buffer. This allows DirectContext3D and Revit graphics to occlude each other correctly.

Anti-aliasing. Anti-aliasing improves the visual quality of boundaries between areas of different color resulting from rasterization of primitives such as lines and polygons. Visual artifacts occur, because rasterization determines the location of boundaries with pixel-level precision. DirectContext3D and Revit graphics use the same anti-aliasing scheme to mitigate the artifacts.

Transparency. DirectContext3D applications can draw transparent objects by submitting them into the transparent rendering pass of Revit.

Orthographic and perspective views. Graphics submitted using DirectContext3D are transformed using orthographic or perspective projection according to the setting of each view.

Discipline-specific views. DirectContext3D graphics are drawn in Architecture, MEP, and Structural views.

Section box. Revit's section box functionality clips away parts of the model. The clipping applies to DirectContext3D graphics in the same way as to graphics generated by Revit.

Image export. Graphics currently displayed in a Revit view, including DirectContext3D graphics, can be exported as an image with a specified format and resolution.

Ambient shadows. Ambient shadows effect is a screen-space post-processing pass that simulates variations in shading that result from nearby objects blocking the light. When the effect is activated for a Revit view, both Revit and DirectContext3D graphics are processed. For the effect to work correctly for DirectContext3D graphics, the submitted geometry must include the normal vector attribute.

Depth Cueing. Depth Cueing is a post-processing effect, which helps to visualize the depth of objects in a view by making objects that are farther away appear as if they are receding into a fog. Graphics submitted using DirectContext3D are subject to this effect in the same way as graphics generated by Revit.

Progressive Drawing. Revit draws graphics incrementally by dividing drawing operations into batches. After each batch is processed, Revit checks whether the remaining batches do not need to be finished, because graphics need to be redrawn again due to UI events. The ability to interrupt drawing early allows Revit to be more responsive to user input. When DirectContext3D applications provide graphics to Revit, they can call DrawContext.IsInterrupted() to find out whether they are wasting time on drawing operations that do not need to be completed.

Limitations of DirectContext3D

Printing only as a raster image. DirectContext3D is based on the part of Revit's graphics pipeline that is used for rendering graphics to the screen via rasterization. Therefore, having DirectContext3D graphics in a view forces that view to be printed in a raster format.

No advanced materials. Graphics displayed in Revit's views can use advanced shading effects and textures. However, DirectContext3D does not provide a way to manage and communicate the necessary information (such as a shader program or a texture) to Revit's graphics pipeline.

No shadows. Revit does not process DirectContext3D graphics in a way that allows the graphics to cast shadows.

2D graphics and annotations. DirectContext3D provides no special support for 2D drawing operations and graphics primitives defined in a 2D space (i.e., all DirectContext3D vertices have three components). Additionally, some 2D graphics in Revit have special meaning as annotations and DirectContext3D does not provide a way to draw such objects. All graphics drawn using DirectContext3D are classified as model objects.

Selection and snapping. DirectContext3D does not provide support for picking and does not provide a way for external applications to use any internal capabilities of Revit related to selection and snapping.

Saving of DirectContext3D graphics with the document. The state of DirectContext3D graphics that are displayed in a view depends on the external applications that are repeatedly requested to resubmit the graphics for drawing. Permanent storage of any necessary state is left as a responsibility of each individual application.

Animation. DirectContext3D is primarily designed for drawing static scenes.

Designing Applications that Use DirectContext3D

External applications that use DirectContext3D have considerable freedom to draw complex graphics in Revit views. However, the flexible nature of DirectContext3D leaves the responsibility of managing the external content and especially of implementing any optimizations that may be necessary up to each specific application. This section discusses both simple patterns of using DirectContext3D and the lessons learned during the development of a large application, the Coordination Model plugin. Coordination Model exemplifies the use of DirectContext3D by displaying Navisworks models streamed by a Navisworks component, which performs heavy optimization.

Design Patterns

Some of the main building blocks of a DirectContext3D application are servers and buffers. The following are some suggestions on how to use these objects.

Using Buffers

There are three ways to use buffers:
1. Make the buffers, fill, and flush them on every call to RenderScene().
2. Re-use existing buffers; fill and flush the buffers on every call.
3. Re-use existing buffers and their contents.

In general, re-using existing objects should allow for higher performance. Additionally, it is better to use fewer buffers by combining similar geometry (in terms of vertex format and effect instance properties). However, there are several considerations that can make this difficult to achieve in practice. First, application logic may require that certain objects are split into different buffers. Second, even a static scene does not need to be drawn as a fixed set of primitives (for example if occluded objects are culled and the view point changes). Third, editable objects can change sufficiently that they can no longer stay merged with other objects. For these reasons, a combination of patterns 2 and 3 listed above is appropriate.

Similar considerations may be applied to managing the parameters of effect instance objects. It is possible to identify a set of values that are needed frequently during drawing and apply these values via effect instances that are re-used between drawing calls.

If buffers and other DirectContext3D objects are re-used between calls, the external application must provide a way for the objects and their contents to be re-created (just before they are used in RenderScene()). This is needed when internal changes in Revit invalidate the objects. The status of the objects can be checked using their IsValid() methods. One case when this situation arises is during image export.

Structure of RenderScene()

RenderScene() is the entry point into an external application's workflow for drawing graphics. Inside this method, a DirectContext3D server manages its DirectContext3D objects by updating and rebuilding them if necessary (see the previous section). Dependencies among the objects may require that changes are made in a certain order, for example:

1. Remake geometry of an object. Create and fill the pair of vertex and index buffers.
2. Update the untransformed outline of the object.
3. Update the transformation of an object. (In a hierarchical scene, several transformations may need to be multiplied together, and it can make sense to cache the result.)
4. Update the transformed outline of the object. (The server must return the transformed outline from GetBoundingBox().)
5. Remake and/or update the effect instance.

The strategy described above may lead to an implementation of RenderScene() similar to the following (in pseudocode):

```
check DrawContext.IsTransparentPass() and determine which geometry to draw

if (m_cube.DirtyGeometry || !m_cube.IsValid())
{
    m_cube.CreateBuffers(VertexFormatBits.PositionNormal, . . .);
    m_cube.MapBuffers();

    fill the buffers

    if (m_cube.Outlines.DirtyUntransformedOutline)
    {
        m_cube.Outlines.UpdateUntransformedOutline(. . .);
        m_cube.Outlines.DirtyUntransformedOutline = false;
        m_cube.Outlines.DirtyTransformedOutline = true;
    }

    m_cube.UnmapBuffers();
    m_cube.DirtyGeometry = false;
}

if (m_cube.Outlines.DirtyTransform)
{
    m_cube.Outlines.CachedTransform = m_parameters.GetTransform();
    m_cube.Outlines.DirtyTransform = false;
    m_cube.Outlines.DirtyTransformedOutline = true;
}

if (m_cube.Outlines.DirtyTransformedOutline)
{
    m_cube.Outlines.UpdateTransformedOutline();
    m_outline = new Outline(m_cube.Outlines.TransformedOutline);
    m_cube.Outlines.DirtyTransformedOutline = false;
}

if (!m_colorEffect.IsValid())
{
    m_colorEffect.CreateEffects();
}

if (m_colorEffect.DirtyColor)
{
    m_colorEffect.UpdateColor(m_parameters.Color);
    m_colorEffect.DirtyColor = false;
}
```

```
bool flat = displayStyle == DisplayStyle.FlatColors || displayStyle == DisplayStyle.HLR
|| displayStyle == DisplayStyle.Wireframe;
m_cube.Flush(VertexFormatBits.PositionNormal, m_colorEffect.GetEffect(flat ||
m_parameters.FlatShading));
```

Using Servers

DirectContext3D provides for multiple external applications that can be active at the same time and can use at least one server each. However, the framework does not stipulate how graphics are divided among the servers belonging to a particular application. There may be minor tradeoffs associated with the choice to use fewer or more servers. Using fewer servers can reduce the API overhead, because a smaller number of both API objects and API calls will be needed. Using more servers may have a hidden benefit in allowing Revit to combine the servers' bounding boxes more efficiently internally than if the operation was performed by an external application. However, the latter effect is likely very minor.

One essential task of an external application in maintaining multiple servers is to respect the active status of DirectContext3D servers owned by other applications. An inactive server is not executed by the framework. When a DirectContext3D application wants to register (or unregister) a server, it can do so in a way similar to the code below.

```
ActiveServers = new HashSet<Guid>(m_DC3D.GetActiveServerIds());

. . .
m_DC3D.AddServer(server);
ActiveServers.Add(id);
. . .
m_DC3D.RemoveServer(id);
ActiveServers.Remove(id);
. . .

m_DC3D.SetActiveServers(ActiveServers.ToList<Guid>());
```

Coordination Model

CoordinationModel is a Revit addin that uses DirectContext3D to display Navisworks models in Revit, allowing designs made in different software packages to be coordinated. The interoperability is possible because of the Navisworks graphics format, which can be processed using a Navisworks component and the DirectContext3D interface for drawing graphics. The ability to direct the output of the Navisworks component to DirectContext3D makes the implementation of CoordinationModel simple from a graphics perspective. However, CoordinationModel is complex when it comes to managing the state of the external graphics, which the addin carries out according to the linked file paradigm. Some aspects of the implementation may provide insights into possible uses of DirectContext3D.

Overview

CoordinationModel treats each Navisworks document as one unit of information that should be maintained as a linked file in the host Revit document. Each link has one type and several instances. The type is associated with a unique DirectContext3D server, which submits its graphics once for each instance. The server's responsibilities include:

- Keeping track of instances.
- Loading, unloading, and reloading the content.
- File management.
- Interacting with the Navisworks component to obtain the graphics for drawing.
- Interacting with handle elements.

The Navisworks component used in CoordinationModel is a C++ library, while the main part of CoordinationModel is written in C#. For this reason, a server interacts with Navisworks graphics using a wrapper written in CPP/CLI. Some of the DirectContext3D calls are also made from the wrapper object. The flow of graphics content is illustrated in Figure 8.

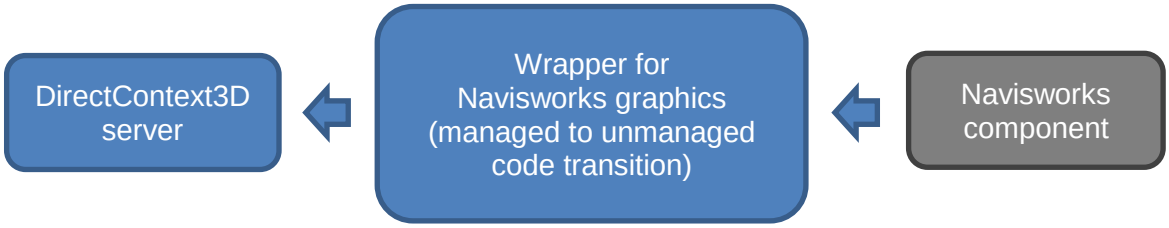

Figure 8. CoordinationModel server obtains graphics from a Navisworks file.

CoordinationModel links are represented in the host Revit document using special elements that act as handles for the external graphics that they contain. These handle elements facilitate repositioning of the external model instances and allow the content to be reloaded when the document is reopened. CoordinationModel servers must synchronize their state with the corresponding handles using a part of DirectContext3D API which is only exposed to internal addins.

Use of DirectContext3D

Navisworks technology makes use of sophisticated optimizations when producing graphics for rendering. As a user of this functionality, a CoordinationModel server only needs to supply the Navisworks library with certain information, such as the current camera parameters. Therefore, from the point of view of drawing graphics, CoordinationModel server implementation is not significantly different from the examples given earlier in this document.

In particular, the Navisworks library serves graphics in a way that matches well with the possible usage patterns of DirectContext3D buffers. The library requests CoordinationModel code to create and map the buffers and then uses GetMappedHandle() to access the allocated memory. In this way, there is a fixed set of pre-allocated buffers available to the Navisworks library that it can fill and flush multiples times on each draw call.

On the other hand, the implementation of a CoordinationModel server is complicated considerably by the state management that is required for the server to manage its content in a manner consistent with the link paradigm. This issue is further discussed in the next section.

Lessons Learned

The main theme in CoordinationModel code is to keep the state of internal objects associated with each linked file consistent. There are two types of user actions that can require the state to be updated:

1. Commands given through CoordinationModel UI. For example: add, remove, or unload a link. Links in the unloaded state are present in the document, but the corresponding external graphics are not displayed.
2. Indirect changes to the links. For example: re-assignment of an instance of one link type to a different type, copy-paste, and synchronization of changes made to the document via worksharing.

The second type of state update is considerably more difficult to execute, because the CoordinationModel application has to monitor the Revit document for changes. However, the main responsibility to update the state in both cases lies with each CoordinationModel server, which must synchronize the state of the following objects: the external file, the Navisworks graphics wrapper instance, and the handle elements. There is also some work to maintain DirectContext3D objects that can persist between drawing calls, such as the buffers, but the effort is minimal in comparison.

It is likely that other DirectContext3D applications will face a similar challenge when attempting to match their graphics with the state of an element in the host Revit document. The following suggestions may be helpful.

- Try to keep the state of a DirectContext3D server as simple as possible.
- Minimize the number of states and avoid partial state updates that would allow a server to be re-purposed for serving a different set of graphics. The implementation will likely be more robust if servers are created and destroyed according to all-or-nothing updates.

In the implementation of CoordinationModel the difficulty with state updates stems mostly from attempting to replicate the behavior of native Revit elements, especially in respect to the type and instance hierarchy. Similar challenges can be expected when mimicking other kinds of Revit functionality using DirectContext3D graphics. It is up to the design of each application to balance the extent to which external graphics behave as native Revit elements. However, the strength of DirectContext3D lies in keeping external graphics exempt from internal rules of Revit and not having to convert the graphics into internal geometry.

Conclusion

DirectContext3D enables external applications to draw graphics in Revit views by participating in the internal rendering pipeline of Revit. The main responsibilities of the external applications are: encoding the external geometry in vertex buffers and submitting the buffers when called back by Revit. Additionally, external applications can adjust the state of the rendering pipeline at the time when they submit graphics for rendering.

The CoordinationModel addin demonstrates that DirectContext3D can handle complex external graphics and exemplifies one possible usage pattern of external graphics as linked files. Since DirectContext3D graphics are not converted to an internal representation in Revit, the implementation of CoordinationModel is fundamentally different from similar functionality in Revit.

DirectContext3D and Workflows Related to External Content

DirectContext3D leaves it up to each application to manage the external content. The CoordinationModel addin follows the linked file paradigm and supports operations such as load, unload, and reload-from that can be applied to each external model. Additionally, CoordinationModel links respect a type and instance hierarchy, which is expected of similar objects in Revit. From the perspective of workflows for managing external content, the similarity of CoordinationModel links and other kinds of linkable objects in Revit is completed by the CoordinationModel user interface (shown in Figure 9), which is intentionally similar to the Manage Links dialog in Revit.

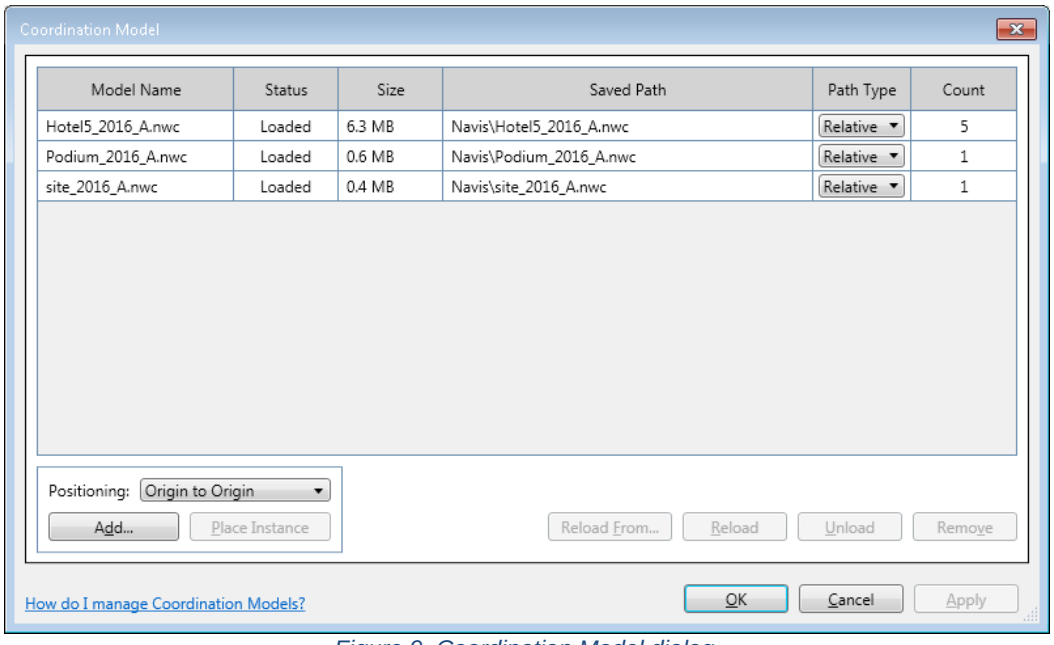

Figure 9. Coordination Model dialog.

However, the source of the graphics displayed for each CoordinationModel link is fundamentally different from other external content that can be displayed in Revit, because CoordinationModel graphics have no internal representation in the document. This suggests a

new way of thinking about external content as a temporary stream of data served by a provider through an API, such as DirectContext3D. This concept can lead to a generalization of the relationship between providers and consumers of data, which can be especially beneficial for emerging workflows that rely on remote data providers.

Additional Resources

Revit API Developers Guide
http://help.autodesk.com/view/RVT/2018/ENU/ (Select **Developers > Revit API Developers Guide**)

Revit SDK
http://www.autodesk.com/revit-sdk

DuplicateGraphics SDK Sample
Summary: The external application creates DirectContext3D servers that extract geometry from selected Revit elements, encode it in pairs of vertex and index buffers, and submit it for rendering using DirectContext3D. The effect of the process, which is triggered using an ExternalCommand, is to display the geometry content of selected Revit elements at an offset, so that the graphics appear to be duplicated.

AUTODESK UNIVERSITY

Appendix

Frequently Asked Questions

1. Can DirectContext3D and CoordinationModel graphics be drawn with visible edges similar to the Hidden Lines Removed display style? DirectContext3D can draw edges using line primitives. It is up to the external application to generate the edge geometry. The DuplicateGraphics SDK sample provides an example. CoordinationModel graphics are in the form of triangle meshes. It is possible to display the edges of the triangles, but that would be different from the Hidden Lines Removed style that shows edges of faces that have not been triangulated yet.
2. Is it possible to pick DirectContext3D graphics? There is no special support for picking in DirectContext3D. An external application can implement picking internally by finding intersections with the graphics that it submits for drawing.
3. Is it possible to snap to DirectContext3D graphics? Currently, there is no support for snapping in DirectContext3D.

Miscellaneous Usage Tips

1. "Global" VertexFormat and EffectInstance objects. It is sufficient to construct a single VertexFormat object of each type (for a total of four). Similarly, EffectInstance objects that use vertex color as the color source can be re-used. Note that these objects that can be used in a global manner in a DirectContext3D application still need to be checked for validity and re-created when they become invalid.
2. Progressive Drawing. To benefit from progressive drawing a DirectContext3D application should return early from the RenderScene() callback when DrawContext.IsInterrupted() is true.
3. Vertex normals. Vertex normals are needed when the Ambient Shadows display option is active. For simplicity, a DirectContext3D application can limit itself to just the PositionNormal and PositionNormalColored vertex formats.
4. DirectContext3D camera parameters. Revit does not typically redraw the whole screen. For this reason DirectContext3D camera parameters reflect the area being redrawn and may not always correspond to a whole view. This makes it possible for DirectContext3D servers to cull the graphics that they submit for rendering.

Code Fragments

GetVertexSize() and GetPrimitiveSize()

```
public static int GetVertexSize(VertexFormatBits format)
{
  switch (format)
  {
    case VertexFormatBits.Position: return VertexPosition.GetSizeInFloats();
    case VertexFormatBits.PositionColored: return
VertexPositionColored.GetSizeInFloats();
```

```
    case VertexFormatBits.PositionNormal: return VertexPositionNormal.GetSizeInFloats();
    case VertexFormatBits.PositionNormalColored: return
VertexPositionNormalColored.GetSizeInFloats();
    default: break;
  }
  return VertexPosition.GetSizeInFloats();
}

public static int GetPrimitiveSize(PrimitiveType primitive)
{
  switch(primitive)
  {
    case PrimitiveType.LineList: return IndexLine.GetSizeInShortInts();
    case PrimitiveType.PointList: return IndexPoint.GetSizeInShortInts();
    case PrimitiveType.TriangleList: return IndexTriangle.GetSizeInShortInts();
    default: break;
  }
  return IndexTriangle.GetSizeInShortInts();
}

// later can map buffers like this:
m_vertexBuffer.Map(GetVertexSize(Format) * NumVertices);
m_indexBuffer.Map(GetPrimitiveSize(Primitive) * NumPrimitives);
```

UpdateTransformedOutline()

```
public void UpdateTransformedOutline()
{
  XYZ outlineMin = UntransformedOutline.MinimumPoint;
  XYZ outlineMax = UntransformedOutline.MaximumPoint;

  XYZ[] corners = new XYZ[8]
  {
    new XYZ(outlineMax.X, outlineMax.Y, outlineMax.Z),
    new XYZ(outlineMax.X, outlineMax.Y, outlineMin.Z),
    new XYZ(outlineMax.X, outlineMin.Y, outlineMax.Z),
    new XYZ(outlineMax.X, outlineMin.Y, outlineMin.Z),
    new XYZ(outlineMin.X, outlineMax.Y, outlineMax.Z),
    new XYZ(outlineMin.X, outlineMax.Y, outlineMin.Z),
    new XYZ(outlineMin.X, outlineMin.Y, outlineMax.Z),
    new XYZ(outlineMin.X, outlineMin.Y, outlineMin.Z),
  };

  for (int pt = 0; pt < 8; ++pt)
    corners[pt] = CachedTransform.OfPoint(corners[pt]);

  TransformedOutline = new Outline(corners[0], corners[1]);
  for (int pt = 2; pt < 8; ++pt)
    TransformedOutline.AddPoint(corners[pt]);
}
```