Learning Objectives

- Learn how to build a library for multiple applications
- Learn how different applications define geometry
- Learn how to use a custom geometry library to aid in the communication between applications
- Learn how to set up a file-free communication pipeline between applications

Description

It's becoming increasingly common for architectural projects to be developed using more than once piece of software, and plug-in developers are pressured to make sure their tools support these multiple platforms. Whether you're making a plug-in that's implemented in Revit software that you also want to use within Dynamo software, or you want to communicate with any other CAD- or vector-based application, one of the biggest hurdles you have to overcome is that different applications—even Revit software and Dynamo software—use different geometry libraries. This class will cover ways we have dealt with porting plug-ins to different formats using a custom geometry library and how that has also lent itself to enabling a more direct communication between applications.

Your AU Experts

Timothy Logan has been working in the architecture industry for 9 years, working as a BIM Specialist with HKS, Inc. out of Dallas, Design Technology Manager for LMN Architects in Seattle, and now as a Computation Designer with HKS again in the LINE studio. He has worked everywhere from a lowly BIM jockey to managing the technology at a large architecture firm to his current position with HKS LINE where he’s developing custom software to push design technology forward to do more today that it did yesterday and enhance how we practice and deliver architectural projects.
Separating Custom Processes from Host Application

Types of Plugins
All of the people at our firm that develop plugins and software on the design technology side of things come from an architectural background and are largely self-trained. While this likely limits our exposure to best practices, we have traditionally seen three types of software that we develop. One is an attempt to replicate a process our outcome from a previous generation of technology, a second tries to automate a process, and another is to create a tool that enhances our capabilities. The two former project types tend to be very specific to an application, while the latter we find ourselves rebuilding for software after software.

Replicating Behavior
The vast majority of development our company has done over the past 30 plus years has been an attempt to replicate what was usually a procedure or outcome that we had in a previous technology generation, whether that’s trying to replicate an outcome from some manual drafting procedure in AutoCAD, or replicating some outcome from our AutoCAD days in Revit. This tends to lend itself to very specific implementations that don’t need to work on multiple software platforms since they’re just filling what is seen as a gap in capabilities.

As an example, one of the problems we’ve always had with Revit has been trying to replicate our company standard drawing index within Revit. It’s certainly been possible from the beginning, but it’s been a fairly complicated process of making sure we set our sheets up with the correct parameters/values and making sure our consultants set their sheets up with the same parameters. And then there are the consultants that do not use Revit so we’re left with trying to create place holder sheets for them. To try an alleviate this, early in our Revit use we paid to have a tool developed that would take an Excel file and create a drafting view for it, replicating the Excel as best it could. It worked, but the user experience of the tool was less than desired and we never deployed it or used it in a project. Much more recently I set about making a tool to get Excel files into a Revit schedule, taking advantage of newer functionality. It wasn’t specifically for a drawing index, but that’s one of the typical use cases and this was much more successful from a UX standpoint and has been put to use in several projects around the office.
Sometimes it’s hard to get people to change an entrenched idea of how a deliverable should look, so we still have a lot of projects where people choose to manage their drawing index in AutoCAD, but our new tool is helping to minimize that. Regardless of which tool or method our project teams wind up using, they are trying to solve a problem that’s very much in the confines of getting Revit to look or function much like our traditional expectations so it doesn’t make sense to try and separate too much of the plugin from the host application (Revit).

**Automating Procedure**

Not always, but typically when we automate a procedure, we are fairly locked to the host application as well. We have developed several plugin’s in Revit that just try and help automate tedious procedures, things like renumbering doors to match rooms, or duplicating views, upgrading files, and the like. Most of these types of plugins are so specific to the Revit and how Revit organizes and manages data that it doesn’t necessarily make sense to try and detach what’s happening from the host application.
That doesn’t mean there are no gains to separating or planning out different aspects of the plugins. There are many functions that we’re having to write (or copy) over and over again that may lend themselves to being separated to a separate library. For instance, all of our internally deployed plugins are written to be included in the Revit ribbon on a custom toolbar, but they’re all separate plugins (different projects, solutions, etc.) so there is a need to search for the tab and panel, and create them if necessary or just add to them if not. This is a lot of code that could be set aside in a separate library.

Our typical button creation occurs in the OnStartup method of the ExternalApplication. Because we’re frequently adding and removing things, especially for beta testers, we’ve having to evaluate if the tab exists and create it if necessary, then determine if the panel exists and create it if necessary. This way we can be fluid with apps we add to the UI and not have to worry about compiling all commands into a single ExternalApplication, something we had done previously.

```csharp
public Result OnStartup(UIControlledApplication application)
{
    try
    {
        // Get the path to the command
        string path = typeof(RevitScaleApp).Assembly.Location;

        // Verify if the tab exists, create it if necessary
        adWin.RibbonControl ribbon = adWin.ComponentManager.Ribbon;
        string tabName = "HKS";
        string panelName = "Tools";
        adWin.RibbonTab tab = null;
        foreach (adWin.RibbonTab t in ribbon.Tabs)
        {
            if (t.Id == tabName)
            {
                tab = t;
                break;
            }
        }
        if (tab == null)
            application.CreateRibbonTab(tabName);

        // Verify if the panel exists
        List<RibbonPanel> panels = application.GetRibbonPanels(tabName);
        RibbonPanel panel = null;
        foreach (RibbonPanel rp in panels)
        {
            if (rp.Name == panelName)
            {
                panel = rp;
                break;
            }
        }
        if (panel == null)
            panel = application.CreateRibbonPanel(tabName, panelName);

        // Construct the PushButtonData
    }
    catch
    {
        // Handle the exception
    }
    return Result.None;
}
```
Cross Platform Plugins for Revit, Dynamo, and Beyond

```csharp
PushButtonData scalePBD = new PushButtonData(
    "Revit Scale", "Revit\nScale", path, "RevitLibTest.RevitScaleCmd")
{
    LargeImage = Imaging.CreateBitmapSourceFromHBitmap(
    ToolTip = "Get a number value compare the document scale to 1m.",
};

// Add the button to the panel
panel.AddItem(scalePBD);
return Result.Succeeded;
} catch
{
    return Result.Failed;
}
```

FIGURE 3: CODE SAMPLE – CREATING REVIT UI BUTTONS

Instead of having to replicate a large portion of this code every time we want to add a command to the Ribbon UI of Revit, we could create a separate library containing frequently used functions. The next code snippet is a static method that we can run from a hypothetical RevitLib class that just adds buttons to the Revit UI by specifying the tab name, panel name, and providing a list of PushButtonData objects.

```csharp
/// <summary>
/// Add buttons to the Revit UI
/// </summary>
/// <param name="revApp">Revit application for button generation</param>
/// <param name="tabName">Name of tab to place buttons</param>
/// <param name="panelName">Name of panel to place buttons</param>
/// <param name="buttons">List of PushButtonData to generate the buttons</param>
/// <returns></returns>
public static bool AddToRibbon(UIControlledApplication revApp, string tabName, string panelName, List<PushButtonData> buttons)
{
    try
    {
        // Verify if the tab exists, create it if necessary
        adWin.RibbonControl ribbon = adWin.ComponentManager.Ribbon;
        adWin.RibbonTab tab = null;
        foreach (adWin.RibbonTab t in ribbon.Tabs)
        {
            if (t.Id == tabName)
            {
                tab = t;
                break;
            }
        }
        if (tab == null)
            revApp.CreateRibbonTab(tabName);

        // Verify if the panel exists
        List<RibbonPanel> panels = revApp.GetRibbonPanels(tabName);
        RibbonPanel panel = null;
        foreach (RibbonPanel rp in panels)
        {
            if (rp.Name == panelName)
            {
                panel = rp;
                break;
            }
        }
    }
```

AUTODESK UNIVERSITY 2015
} if (panel == null)
    panel = revApp.CreateRibbonPanel(tabName, panelName);

    // Add the button(s) to the panel
    foreach (PushButtonData pbd in buttons)
    {
        panel.AddItem(pbd);
    }
    return true;
} catch
{
    return false;
}

FIGURE 4: CODE SAMPLE – STATIC METHOD FOR ADDING UI BUTTONS IN REVIT

By doing this, each time we create a new ExternalApplication we can use our static method in this separate library for generating the buttons in the UI. This way it saves us from having to remember as much in how to implement the buttons as well as saving about 30 lines of code for each implementation.

```csharp
public Result OnStartup(UIControlledApplication application)
{
    try
    {
        // Get the path to the command
        string path = typeof(RevitScaleApp).Assembly.Location;

        // Construct the PushButtonData
        PushButtonData scalePBD = new PushButtonData(
            "Revit Scale", "Revit\nScale", path, "RevitLibTest.RevitScaleCmd"
        )
        {
            ToolTip = "Get a number value compare the document scale to 1m."
        };

        // Add the scalePBD to a list. There could be an override option for the AddToRibbon method of
        RevitLib
        // manage creating a single versus a list of buttons.
        List<PushButtonData> buttons = new List<PushButtonData> { scalePBD };

        // Create the button(s)
        LINE.RevitLib2016.AddToRibbon(application, "HKS", "Tools", buttons);
        return Result.Succeeded;
    } catch
    {
        return Result.Failed;
    }
}
```

FIGURE 5: CODE SAMPLE – IMPLEMENTING UI BUTTON CREATION THROUGH A SEPARATE LIBRARY

You can see the complete example projects for the common library and sample Revit plugin using it on github:

https://github.com/logant/RevitLibExample and https://github.com/logant/RevitLibTest
Creating New Tools
I would argue that historically most of our software development would fall into the previous types of applications or plugins, something to fill a very specific hole in a given application. Even so, even early on we started using software development to make new tools to augment our workflow and allow us to do more. While some of this may be particular to an application, there’s quite a bit that we’re seeing the need to port over to many of the new applications in our toolkit.

As an example, back in the mid 1990’s HKS created an AutoCAD command to create a seating bowl profile. This uses a fairly standard algorithm to compute the parabolic bowl profile, generating the line work of the bowl section within AutoCAD.

Then around four years ago, we decided to try and port this to Revit as another option for creating bowl sweeps within Revit. This was mostly a direct replication of the AutoCAD command, just using it to build Revit elements (profile families) rather than just drawing lines in model space like AutoCAD.
And of course after this in more recent times as computational design tools, in this case Grasshopper, started to become more involved in our work, we started to build tools to do the same within there. This gave us the ability to create a bowl profile, as well as easily modify and stack multiple profile definitions together to make a complete bowl profile where each profile is dependent on the one that came before it going from the field out to the exterior.
Each of these variations on a bowl profile generator work on the same basic algorithm, but each one is built independently from each other and one cannot be updated without updating all of the others. With tools like this, the logic is the same for each version of the command, but the problem has always been that writing each to date has been entirely dependent on the geometry library of the host application.

Planning What to Separate
We’re taking advantage of an effort to rebuild our Revit plugins to think about how we can not only make the plugins better, but to see how we can better implement creating plugins. This has pushed us to start looking at how we can minimize the copy/pasting we do between applications, as in the Revit button instantiation shown above, but we’re also looking at how we can better implement custom tools like our bowl generator example.

Application Specific Library
As stated, one of the places we’re looking at segregating from any given plugin has been host application specific functions that we find ourselves using over and over again. This can be something like the UI implementation that we’ve shown, or something getting a scale factor for when we need to deal specifically with content in a specific units (usually meter based metric).

Geometry Library
One of the big things that we’ve determined from this evaluation of our code is the need of a common geometry language to separate out applications like our bowl generator. There are different levels for how this geometry library may need to be implemented, depending on if we need primarily just a data structure to pass information, say a Point object that has three coordinate parameters, or if we need to perform geometric operations like projecting a point onto a curve.

Investigating How Applications Define Geometry
A Case for Making a Geometry Library
Most of what we need to know about geometry is well known, at least to mathematicians who study it, which makes geometry is a convenient language for dialog between applications. However, we have found that even though the content is very similar the dialect can be very different between how each application talks geometry. So to better understand what’s happening it takes time and investigation, and this investigation typically needs to take place between every application that may use it. There is definitely some overlap, particularly when different applications are provided from one source, but generally different 3d or geometry related applications have different geometry cores that underlie them. So its safe to assume that Revit, Dynamo, Inventor, AutoCAD, 3ds Max, Maya, Rhino, SketchUp, Vectorworks, Catia, Bently, Archicad, Solidworks, and on and on, may have different geometry libraries, or even different implementations of the same geometry libraries.

So far, we’ve only worked up to coordinate based elements – points, vectors, planes – and curve based elements – lines, arcs, circles, polylines, and NURBS curves. Even within these two specific groups there are gaps in both properties/functions, as well as types of objects (I’ve just now thought of ellipses). This
is a project that we plan to continue for our own better understanding more so than it being vital (more on that later).

**Easy/Hard**

Even though this is a well know subject, there are many parts of what we’ve started doing with our geometry library that are really easy and well understood, and some that is quite a bit more difficult. Sometimes I just get it, sometimes I get it enough to implement if not fully understand, and sometimes I don’t understand enough yet to even try to implement it. Depending on what you’re trying to do, sometimes this is enough. While entirely subjective to my understanding at this point in time, these are what I generally found easy and difficult in our investigations so far:

<table>
<thead>
<tr>
<th>Easy</th>
<th>Difficult</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points</td>
<td>Vectors (Slightly)</td>
</tr>
<tr>
<td>Vectors (Mostly)</td>
<td>Planes (Slightly)</td>
</tr>
<tr>
<td>Planes (Mostly)</td>
<td>Transforms</td>
</tr>
<tr>
<td>Lines</td>
<td>Arcs (Slightly)</td>
</tr>
<tr>
<td>Arcs (Mostly)</td>
<td>Circles (Slightly)</td>
</tr>
<tr>
<td>Circles (Mostly)</td>
<td>Nurbs Curves</td>
</tr>
<tr>
<td>Polylines</td>
<td></td>
</tr>
</tbody>
</table>

**A Case For Not Making a Geometry Library**

I work within a studio at HKS (a large firm) that is charged with, and motivated by, research within architecture. This takes on different forms depending on who you talk to. For some in my studio, research for them is doing competitions where they are free to stretch their design skills to further extremes. For others, like myself, research time is used to better understand the tools we use and investigate how we interact and use them. This time is admittedly not a luxury that is provided to all who practice architecture, even within my firm. That being said, there are plenty of open source projects out there that have their own geometry library. There are projects like Satoru Sugihara’s iGeo computational design library, or Peter Boyer’s verb library that could probably be adapted to use (assuming their licensing accommodates this).

With Computational Design becoming more prevalent in the industry, geometry is something that we have begun deal with intimately every day and it is our hope is that better understanding of geometry will lead to efficiency and novelty for how we implement ideas. Our self-interest in creating a geometry library at this point lies more within the realm of better educating ourselves in the vocabulary of geometry not necessarily as an absolute necessity.

**How Our Investigation Developed**

Investigating geometry and how we would apply it to the cause of separating out process from our current slate of design software took a few different, mostly sequential stages. First, there was an informal investigation based primarily on our experience in software development and computational design. Then we had a little bit more formal investigation where we started investigating different software packages to better understand the geometry we would need and the properties and functions
that go along with it. And finally we had to start comparing how the different geometry libraries created and stored information. The former two stages are mostly preoccupied with developing our own library, where the latter stage would be relevant to both using a custom library or a found/open source library.

**The Informal**

This initial part of the investigation was mostly taken up by cataloging the types of geometric elements and related properties and processes that were relevant to our current needs and expectations for the tools. This primarily was us making a list of things like Points, Vectors, Lines, etc. and all of the construction options and properties we wanted access to. A lot of this list making was just based on our experience, particularly with a more computational design perspective where we’re constantly creating and manipulating geometry.

As an example, we knew we were going to need a point object. Points are arguably the basic starting point for geometry. Points make lines, which make surfaces, which make solids. They’re a basic element that appears at some level in everything we do geometrically.

Points are easy to summarize: they’re made up of 3 numbers that represents the X, Y, and Z of our dominant understanding of 3d space. Creating a point is just a matter of supplying these three numbers. After that, you have to decide what you want to do with a point once you have it. Sometimes just having it is enough, you’re locating something in space, but you also may want to add two points together, subtract them, move them by adding a vector to them, rotate around some arbitrary (or not) axis with a transform, convert them to text, or convert text to a point.

**The Formal**

This middle stage was probably the most difficult, and is still ongoing. This involved both investigating how different software platforms implement geometry and its functions, as well as the mathematical functions behind these functions so we could effectively implement them. Tools like Dynamo and Grasshopper were invaluable here. These computational design tools allowed us to quickly see how geometry was implemented from the point of constructing it, seeing what properties were available, and what kind of actions you could do to it. Dynamo in particular was useful in how its node library is organized, with a clear delineation from Create, Query, and Action subsections for the nodes of a given geometry object.
Just looking through the API documentation for Revit or other geometry focused applications is also quite helpful in seeing the possibilities of their points, like the XYZ object in Revit. Just looking at it you start to see that the XYZ is more than just a point, but also acts as a vector. Vectors are very similar to points in that they’re both made from 3 coordinate values, but they don’t necessarily function identically. For this case it’s probably just important to mention that points locate an object in 3d space, but vectors only provide direction and length so are used for very different things. It’s also worth noting that Revit isn’t necessarily unique in this respect as it’s not at all uncommon to use a ‘vector’ to represent a mathematical vector and point.
Cross Platform Plugins for Revit, Dynamo, and Beyond

## Constructors

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>XYZ()</td>
<td>Creates a default XYZ with the values (0, 0, 0).</td>
</tr>
<tr>
<td>XYZ(Double, Double, Double)</td>
<td>Creates an XYZ with the supplied coordinates.</td>
</tr>
</tbody>
</table>

## Methods

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Add</td>
<td>Adds the specified vector to this vector and returns the result.</td>
</tr>
<tr>
<td>AngleOnPlaneTo</td>
<td>Returns the angle between this vector and the specified vector projected to the specified plane.</td>
</tr>
<tr>
<td>AngleTo</td>
<td>Returns the angle between this vector and the specified vector.</td>
</tr>
<tr>
<td>CrossProduct</td>
<td>The cross product of this vector and the specified vector.</td>
</tr>
<tr>
<td>DistanceTo</td>
<td>Returns the distance from this point to the specified point.</td>
</tr>
<tr>
<td>Divide</td>
<td>Divides this vector by the specified value and returns the result.</td>
</tr>
<tr>
<td>DotProduct</td>
<td>The dot product of this vector and the specified vector.</td>
</tr>
<tr>
<td>Equals</td>
<td>Determines whether the specified Object is equal to the current Object.</td>
</tr>
<tr>
<td>GetHashCode</td>
<td>Serves as a hash function for a particular type.</td>
</tr>
<tr>
<td>GetLength</td>
<td>Gets the length of this vector.</td>
</tr>
<tr>
<td>GetType</td>
<td>Gets the Type of the current instance.</td>
</tr>
<tr>
<td>IsAlmostEqualToXYZ</td>
<td>Determines whether this vector and the specified vector are the same within the tolerance (1.0e-09).</td>
</tr>
<tr>
<td>IsAlmostEqualToXYZ(Double)</td>
<td>Determines whether 2 vectors are the same within the given tolerance.</td>
</tr>
<tr>
<td>IsUnitLength</td>
<td>The boolean value that indicates whether this vector is of unit length.</td>
</tr>
<tr>
<td>IsZeroLength</td>
<td>The boolean value that indicates whether this vector is a zero vector.</td>
</tr>
<tr>
<td>Multiply</td>
<td>Multiplies this vector by the specified value and returns the result.</td>
</tr>
<tr>
<td>Negate</td>
<td>Negates this vector.</td>
</tr>
<tr>
<td>Normalize</td>
<td>Returns a new XYZ whose coordinates are the normalized values from this vector.</td>
</tr>
<tr>
<td>Subtract</td>
<td>Subtracts the specified vector from this vector and returns the result.</td>
</tr>
<tr>
<td>ToString</td>
<td>Gets formatted string showing (X, Y, Z) with values formatted to 9 decimal places. (Overloads Object.ToString.)</td>
</tr>
<tr>
<td>TripleProduct</td>
<td>The triple product of this vector and the two specified vectors.</td>
</tr>
</tbody>
</table>

## Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>Adds the two specified vectors and returns the result.</td>
</tr>
<tr>
<td>Division</td>
<td>Divides the specified vector by the specified value.</td>
</tr>
<tr>
<td>Multiply(Double, XYZ)</td>
<td>Multiplies the specified number and the specified vector.</td>
</tr>
<tr>
<td>Multiply(XYZ, Double)</td>
<td>Multiplies the specified number and the specified vector.</td>
</tr>
<tr>
<td>Subtraction</td>
<td>Subtracts the two specified vectors and returns the result.</td>
</tr>
</tbody>
</table>

## Properties

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasisX</td>
<td>The basis of the X axis.</td>
</tr>
<tr>
<td>BasisY</td>
<td>The basis of the Y axis.</td>
</tr>
<tr>
<td>BasisZ</td>
<td>The basis of the Z axis.</td>
</tr>
<tr>
<td>Item</td>
<td>Gets the first coordinate.</td>
</tr>
<tr>
<td>Y</td>
<td>Gets the second coordinate.</td>
</tr>
<tr>
<td>Z</td>
<td>Gets the third coordinate.</td>
</tr>
<tr>
<td>Zero</td>
<td>The coordinate origin or zero vector.</td>
</tr>
</tbody>
</table>

*Figure 10: Revit XYZ Element Documentation*
**Interoperability Comparisons**

The final step before calling any feature in our geometry library finished is to compare the results of using it with other geometry based software platforms. We made sure to check through several applications and how they implement the areas of geometry we’re currently focused on because not all applications are built to handle creating, querying, or manipulating geometry in the same way. Take Revit and Dynamo as our primary example. Both software are parametric modeling environments, but the method in which the parametricism is implemented in them is quite different. On a fundamental level, Revit doesn’t expose points as an object to a user except as a Reference Point within the conceptual mass environment. Because of this, the way you use and manipulate points is different than if they were treated just as another piece of geometry as it is in Dynamo.

What this means for us is that we need to look at several sources to determine what we may need to include with our geometry library. If our main source of information on how geometry is used was Revit, we would be missing out on a number of useful functions that we see once we start delving into tools like Dynamo.

**Potential Problem Areas**

This section on geometry is going to be focused on some of the areas that we’ve seen the greatest difference with the different software platforms and how they deal with geometry, or understand what could or should be done with it, just to point out some gotcha moments that you may want to look for.

**Imaginary Geometry**

The first problem area to keep an eye out for is geometry types that do not exist in one platform versus another. Dynamo, AutoCAD, and Rhino all have polylines, but Revit does not. If you’re in a project or typical family file, when you select a curve of some type it will only select the one segment. When you’re in the conceptual mass family and you select a curve, you may see what appears to be a polyline or polycurve based on selection, but when you look at your selection count you will see multiple curves, not one.

*Figure 11: There is No Polyline*
This just means when you translate the geometry, you have to know to explode it and get the pieces before you try and create it within Revit. For a polyline this is relatively easy as you can just use the vertices to reconstruct multiple lines, but other element types may not be quite so easy to reconstruct.

**Non-Functions**

The second area goes back to the idea of looking at several sources before you start planning out the features you need. If you’re only looking at one implementation of a geometry library, then you may be lead down a path where you start to make poor assumptions about what’s possible and what should be included. Two things that stand out are obtaining the closest point between a point and a curve, and splitting a curve. I’m still fairly certain splitting a curve is not possible via Revit’s API (though I’ve been wrong before), but when I started this process I was similarly certain that Revit didn’t have either of these functions. After investigating a little deeper, it turns out it does have a closest point function, it just calls the function Project rather than something like Dynamo’s ClosestPointTo function. The name Project in conjunction with a curve doesn’t initially make me thing of project a point onto the curve, but I can understand the naming convention even though it’s just a different way of looking at it than I am used to seeing. An example of this can be found by reviewing the Spline_ClosestPoint.dyn file provided with this class.
The important take-away is that as with most things you research, you want more than one source. Checking the properties and functions you’re trying to develop against multiple sources will help make sure you know what things you need to consider, but also when you’re using this for translation and interoperability then you need to understand how one geometry library looks compared to another.
An Example: NURBS Curves
As an example of how we evaluate geometry, I’d like to discuss a NURBS curve. These are very common types of freeform curves that are found in most modern CAD or BIM software and are also one of the areas where you’re likely to see some deviation in implementation. In discussing them, it would be beneficial to define some terms and basic information about the history of freeform curves.

Freeform Curves
Freeform curves start, at least for our history, from Bezier curves. Bezier curves were developed independently by several people, most famously Pierre Bezier from whom the curves are named (he was the first to publish) and Paul de Casteljau, who’s algorithms for calculating Bezier curves are still in use. Bezier curves have $n + 1$ control points, with a degree of $n$, meaning that if you add a control point you are also changing the degree. This makes it so that every control point affects global change across the entire curve, i.e. changing just one will affect the shape across the length of the curve. Because of this you see piecewise Bezier curves, Bezier curves that are strung together end to end to make what appears to be a large curve of a specific degree.

The degree of a curve affects the influence of control points to the curve. In the following figures, you can see that a lower degree curve will maintain a closer proximity to the control points with less ability to affect change along the length of the curve.

![Figure 15: Freeform Curves of Different Degrees. From Top: Degree 3, Degree 5, and Degree 11](image)

![Figure 16: Movement of Control Point 6 (Figure 15) does not affect beyond Control Point 4 with the Degree 3 Curve (Right), But does for the Degree 5 Curve (Left).](image)
While they maintain a specific degree and are quite flexible, piecewise Bezier curves are still problematic to work with because you have to manage the control points for each segment individually to ensure the tangency between the last two control points in one section matches the tangency of the first two control points in the next segment. To get around this, B-Splines were created, with its main contribution being the knotting of splines together automatically, without specific user intervention and keeping the curves at the segment transition smooth. The knot vector of a curve is just an indication of where the Bezier curve spans are within a B-Spline. The ratios of a knot vector are more important than the whole, such that a knot vector of \([0,0,0,1,2,3,3,3]\) is identical to \([0,0,0,10,20,30,30,30]\).

Then after B-Splines we get to NURBS, or Non-Uniform Rational B-Splines. The important new word is *Rational*, which in this case indicates that the control points have a weight, otherwise they are very similar to B-Splines. With Bezier curves and B-Splines, the control points essentially have the same weight and affect the curve to the same extent, but NURBS you can modify the weight of a control point to affect an area of a curve less or more. It is this property that has led to the popularity of the curve in many CAD packages and 3d modeling programs since the addition of a weight means that a NURBS curve can accurately describe a freeform curve, a line, or even a circle or arc, something that Bezier and B-Splines cannot do because every control point affects the curve the same.

**Continuity**

Curve continuity is a means of describing the continuity of a curve. It’s often described in terms of geometric continuity or parametric continuity, both of which cover different aspects of continuity. The continuity of a curve has traditionally be much more important to aviation and automotive design than architecture, but with more freeform surfaces appearing the continuity has become an import factor in design. Geometric continuity is usually the easier of the two to understand and is written as either \(G_0\), \(G_1\), or \(G_2\). \(G_0\) is when the end points match, though not
necessarily smoothly. G1 is when there is parallel tangency between the end of one segment and the beginning of the next, i.e. the vector from the last two control points of one segment is parallel to the first two control points of the next segment, and G2 is when there is tangency among the sequential end segment vectors and the length of the vectors are identical.

So NURBS curves have an interesting amount of complexity to them just by their nature, and after investigating them between several applications, it becomes apparent that there are small deviations in how they’re implemented that affect how you create them in one application versus any other.

Specifications
There are typically seven properties that go into making a NURBS curve: control points, weights, knots, degree, periodic, closed, and order. There’s a chance that NURBS creation methods in most applications will utilize some combination of these. Control points, weights, knots, and degree have been discussed to some extent, the others less so. Closed and periodic are sometimes used interchangeably, but mean slightly different things. Closed just refers to the curve being closed, the start point and end point occupy the same space. Periodic also refers to a closed curve, but specifically closed so that there’s a smooth transition the end back to the start and not a kink of any kind. The order is equal to the degree + 1, so a 3 degree curve will have an order of 4, which is to indicate the number of nearby points that influence a curve.

Creating a NURBS Curve
When creating a NURBS curve, there will be some combination of maybe half of the aforementioned properties. Because of this there will be some assumptions being made on the rest of the properties. Generally any creation method you will have to specify the control points, but almost everything else could be generated or assumed.

- Degree - Without specifying a degree, most applications will default to a degree of 3.
- Order - Once you have a degree, you can generate your order as degree + 1.
- Weights – Each control point has an associated weight, and without a weight being defined a value of 1.0 is assumed for each.
Closed – If the first and last control points occupy the same space, the curve will be closed.

Periodic – Typically if a curve is determined to be closed it will be generated as periodic and give you an entirely smooth closed curve.

Knots – For knots, some assumption is also made, but the assumption can affect the end result. The number of knots is typically defined as control points + degree + 1, so if you have 7 control points, a degree of 3, there should be 11 knots. There are some exceptions to this, for instance Rhino requires control points + degree – 1, so 2 less knots. There is often a sequence of the same knot value starting or ending a knot vector called the knots multiplicity and the number of them should not exceed the order of the curve. Between the start and end multiplicity you should have a non-decreasing sequence of numbers. So if we take our previous example of 11 knots, an auto-generated knot sequence will likely be \([0,0,0,1,2,3,4,4,4,4]\).

Actual Comparison

So to make the actual comparison of our NURBS curve, I’ll discuss Revit, Dynamo, and Rhino and their NURBS creation methods. As we look through here we can start to see how the shape of a NURBS curve in one application may be slightly different than one created with the exact same information in another. Typically this is because of the assumptions that the creation methods make when filling in the left out information.

Starting with the control polygon for simplicity, I drew a polyline in CAD and imported it into Revit. Then using Revit’s Model Line command (manual drafting) I drew a Spline type curve clicking on the vertices of the imported control polygon.
The result shows a very similar spline to the one shown above, but we can verify how closely it lines up by importing the same curve created in Rhino. In doing so, we see that the Revit and Rhino generated splines have a slightly different shape.
Since Dynamo doesn’t have a manual method of drawing, the next step is to look at how the programs create NURBS curves programmatically. To make this test, I’ve started by drawing a series of NURBS curves in Rhino using this same control polygon. I’ve created the test with 4 different curves: a 3 degree open curve, a 3 degree closed curve, a 5 degree open curve, and a 5 degree closed curve. You can see this file in the class files by reviewing the file *Rhino_Splines.3dm*. If you do not have Rhino then you can review them by looking at the four exported DWG files named *3deg_Closed_NurbSpline.dwg, 5deg_Open_Nurbspline.dwg*, etc.

![Figure 22: 3 and 5 Degree NURBS Curves, Both Closed and Open](image)

After manually drawing the splines, I went into Grasshopper to export out information related to them. This was done by referencing the curves into Grasshopper and using a C# component two extract the information and format it into a text file. This Grasshopper file is available in the class files and is named *Rhino_Splines.gh*. This information is then used to recreate the curves in Dynamo and Revit.
First within Revit, I imported the curves one at a time, starting with the 3 degree open curve. Then I wrote a document based macro that will read the exported text file (XML format) and use Revit’s two NURBS curve creation methods to generate the curve. When you run the macro (SplineImport) you will first be prompted to select a file. Select an XML file exported from the above Grasshopper script, provided in the class files as Rhino_Splines.xml.
After selecting an appropriate XML file, a form will ask you to specify which curve you want to import (based on the four curves exported from Grasshopper) and which line style you would like to assign to the generated curves. I created two line styles named “Spline Test 01” and “Spline Test 02” for this comparison. If you simply click OK and try to create the 3_degree_Open curve, you will most likely get a failure. This goes back to the knot structure that Rhino uses (control points + degree - 1) and the one that Revit and many other applications uses (control points + degree + 1). Checking the “Allow modification of data” checkbox will insert a duplicate knot value to the beginning and end of the knots list and should create the curve successfully.

Doing so will show you two different curves. The curve created with Spline Test 01 line style (the blue line) will look very much like a spline created through Revit’s manual drafting methods. The curve created with the Spline Test 02 line style (magenta) will look very much like how Rhino defines the curve. Looking at the macro’s source you will see the two different creation methods. The first creates a NurbSpline using control points and weights so degree, knots and all the rest are generated by assumptions. For the second option the creation method uses control points, weights, knots, degree, closed, and rational options so much less is assumed and you get a faithful representation of the curve.

```csharp
// Build the Curves
NurbSpline ns1 = NurbSpline.Create(controlPoints, weights);
NurbSpline ns2 = NurbSpline.Create(controlPoints, weights, knots, degree, closed, rational);
```

Figure 25: Select the Curve to Import and the Line Style to Assign to Each Creation Method

Figure 26: Revit API NurbSpline.Create Methods
At this point, I also felt it was worth-while to look at how Revit is generating the missing information. We are already looking at the data from Rhino via the Grasshopper export, but seeing the differences in the data can be very interesting. For this investigation there is another macro named SplineInformation within the same project (**Spline_Test2016.rvt**). This shows that the major difference between how Revit generates the NURBS curve compared to Rhino is how it generates the knots.
Figure 27: Comparison of Spline Properties Between NurbSpline.Create(Control Points, Weights) [Top] and using NurbSpline.Create(Control Points, Weights, Knots, Degree, Closed, Rational) [Bottom]
Knowing how to do things manually in Revit generally provides a good idea of what will be possible through its API. So while drawing a spline you will notice that you cannot create a closed spline, at least not one with smooth continuity. While the macros presented here will create a NURBS curve in Revit for both the 3 degree open and 5 degree open curves, they will fail when attempting to create one of the closed curves.

Dynamo is next up on the list of programs to check, so first I wrote a Dynamo script to parse through the Text file and organize the data into a Dynamo friendly data. This is then used to start creating NurbsCurves from the core Dynamo library. When we look at the creation methods for NurbsCurve has 4 different creation options, but we’re only interested in the first four as the rest are more relevant to an interpolated curve (spline through points or Hermite spline in Revit) and more of a traditional Bezier curve with the ByPointsTangents option. The first four all have to do with control points, so we’ll focus on those.

![Dynamo Spline Creation Options]

Dynamo is next up on the list of programs to check, so first I wrote a Dynamo script to parse through the Text file and organize the data into a Dynamo friendly data. This is then used to start creating NurbsCurves from the core Dynamo library. When we look at the creation methods for NurbsCurve has 4 different creation options, but we’re only interested in the first four as the rest are more relevant to an interpolated curve (spline through points or Hermite spline in Revit) and more of a traditional Bezier curve with the ByPointsTangents option. The first four all have to do with control points, so we’ll focus on those. Just looking at the first option, just supplying the control points (akin to manually drawing in Rhino) you can see that the Dynamo and imported Rhino curves line up nicely. You can verify this using the SplineTest.dyn file in the class example files.
Only having the visual check is probably enough to assume that Dynamo and Rhino make similar assumptions when generating the missing information for creating a NURBS curve. Looking at the Knots, the one thing we identified that was drastically different between a Revit generated NURBS curve and a Rhino generated NURBS curve, we can see that the knots align very well with how Rhino generates the knots.
What is shown here is only a single specific geometry type, but indicates the kind of validation that may need to go into developing a geometry library that needs to be interoperable with accuracy between separate software platforms. What’s been showcased is the minimum amount of information for a library for translation and interoperability, and more needs to go into it regarding the functions that you want to perform on or with the geometry objects. Closest point to a curve, point at a given parameter, tangency at a given parameter, length, area (if closed), will need to start being input into the class for it to be useful as a stand-alone resource.

**Figure 30: Dynamo NURBS Curve Information – Compare the Knots Here Against Knots From Figure 27**
Using Geometry for Interoperability

Lyrebird as Case Study
Lyrebird is a plugin developed to allow interoperability between Grasshopper and Revit. The intention was to create something that allowed us to develop a plugin that could allow us to create multiple types of elements with different element creation styles as intelligently as possible. The instantiation within Revit stores information via Extensible Storage so that updates are possible if changes are made in the Grasshopper file.

This project was started and largely developed while I was with LMN Architects in Seattle. LMN had (and likely still has) a prominent use of Rhino in early design and used Grasshopper for a lot of sophisticated computational design spearheaded by their Tech Studio. Dynamo had been around for a while, but it hadn’t really developed to the same level as Grasshopper and we were often in Rhino at the beginning phase of a project anyways. The original idea was first implemented as a tool that would let Grasshopper send data out to multiple software platforms. This initial set of plugins let it export and instantiate elements in Revit 2014 and Catia V5. Some of the functionality, such as the general communication strategy between software was kept, but other parts, such as Catia support and the general UI/UX were dropped or replace.

Another Tool in a Crowded Toolbox
At the time I started working on this project, there were already several other options out there. There was Case Inc.’s OpenNURBS plugins that would create Rhino elements in Revit. This was foundational to their later release of Rhynamo, a plugin for importing Rhino geometry in Dynamo that could be transferred to Revit. Case’s OpenNURBS plugin always felt more massing centric, it was about getting geometry more than data and directly creating BIM elements.

There was also Chameleon, which could both pull Revit geometry into Grasshopper and push data to create Revit objects. This plugin had a sophisticated communication mechanism and was file free, but if memory serves it was mostly applicable to creating curtain walls and adaptive components.

Hummingbird was another project to create Revit elements from Grasshopper. Hummingbird gave the greatest flexibility in the types of elements it could create but at the time at least it didn’t update based on changes in Grasshopper.

There was GeometryGYM which generates IFC files based on what you’ve built and organized in Rhino and Grasshopper. This is probably the oldest interoperability plugin I can think of for Grasshopper. It also had the benefit of being IFC so it could work with any other BIM platform.

After I started my project, others started appearing too. There was a plugin named Grevit that would transfer data and elements from Grasshopper to Revit and AutoCAD Architecture.

While never released, there was a plugin developed at LMN by Dan Belcher called Cricket that was used to transfer the façade of a project from Grasshopper to Revit. And if you look far enough back in videos I’ve posted to Vimeo you’ll see something I made to transfer Rhino geometry to Revit in a very unsophisticated way.

So including Lyrebird that’s six released plugins, and two unreleased, that are all trying to do the same thing in often very different ways. I kept at developing Lyrebird in spite of this because I felt, then and now, that it works better for how I want to use that type of tool. Others may find
one of the others to better suit their workflow and need, but it did teach me about translating
go geometry between different applications and how to communicate between these separate
applications.

**Relevant Parts**
The two main things that are relevant to this topic are how the geometry was handled and how
the communication occurs between the applications. This marks the first exploration I did to
really understand the difference in geometry between different applications, and it was
enlightening in that respect. The communication pipeline was set up to be more convenient
than a file based interoperability and to try and minimize confusion over what text or
spreadsheet file represented the correct set of data.

**Lyrebird Geometry**
At LMN we had typically been using Chameleon when transferring elements from Rhino to Grasshopper,
but it was limited to adaptive components, which are great, but not always the most appropriate type of
element to use. Lyrebird started out wanting to be able to place adaptive components and other family
instances, then other use cases (feature creep) started to grow out from there. As it currently stands,
Lyrebird has a very simple geometry library containing two classes: LyrebirdPoint and LyrebirdCurve.

**LyrebirdPoint**
LyrebirdPoint is a simple class that stores three number variables named X, Y and Z. It was
created for and really only works to store the very basic information about a point (it’s
coordinates) and really just acts as a convenient structure to rebuild a Revit XYZ object from a
Rhino Point3d object.
**LyrebirdPoint**

LyrebirdPoint is more complex than the LyrebirdPoint, but it’s still fairly simple. It understands Lines, Arcs, Circles, and Splines, and like the LyrebirdPoint, it’s only used to store information in a structured format so a proper Revit curve can be created on the other end.
Translation Process

The process of translating the geometry occurs on both sides of the pipeline, but most of the heavy lifting is done in Grasshopper, using Rhino geometry for this particular plugin. Points are pretty easy and are just a matter of translating the coordinates. Curves, as we’ve discussed previously can be a little more difficult. Since Lyrebird is more limited in its scope, only working between Rhino and Revit geometry, developing a strategy is mostly a mapping exercise between Rhino geometry to Revit geometry.
Rhino Side (via Grasshopper)

Through Grasshopper, all of the Rhino geometry is first translated to an equivalent Lyrebird geometry type. Point3d objects get translated to LyrebirdPoint objects via matching the X, Y, and Z coordinates. Curves take some processing depending on the type of curve. First, the curves are checked against a known LyrebirdCurve type, so they’re checked for being Lines, Arcs, Circles, and Splines.

- **Lines** – If the curve is determined to be a line, then the start and end points are added into a list (start then end) and that list is assigned to the LyrebirdCurve’s ControlPoints property and the CurveType property is set to “Line”. Every other property is left as null.
- **Arcs** – If the curve is determined to be an arc then the start, midpoint, and endpoint are added to a list, in that order, and that is assigned to the LyrebirdCurve’s ControlPoints property and the CurveType is set to “Arc”. All other properties are left as null.
- **Circles** – If the curve is a circle, then five points are collected: start/end point, the points at parameters 0.25, 0.5, and 0.75, and then the start/end point one more time. This list is assigned to the ControlPoints property and the CurveType is set to “Circle”.
- **Splines** – For splines (NURBS curves) the curve’s degree is used to determine whether it tries to create a NURBS curve or a Hermite spline (interpolated curve). For a degree 3 curve, which was seen as more accurate when translated, the control points, weights, degree, and knots are all extracted from Rhino and stored in the LyrebirdCurve’s ControlPoints, Weights, Degree, and Knots and the CurveType set to “Spline.” If the degree was something other than 3, the spline was first interpolated by getting the point at a set number of divisions (100) along the curve and using those as the ControlPoints. Then the Weights and Degree is also defined, leaving out the Knots.
- **Polycurves** – Since Revit doesn’t understand polycurves or polylines, the polycurve is exploded first and then each segment put into one of the above bins (Line, Arc, Circle, or Spline).

See the GetLBCurve method from the GHClient.cs file in the Lyrebird Github repository.

Revit Side

Once the Lyrebird geometry and any other associated information is packaged and sent to Revit, there needs to be another geometry translation from the Lyrebird objects to Revit geometry. Because the geometry was formatted to be Revit friendly before it was sent, this is the easier side of the application. The Revit UnitUtils.ConvertToInternalUnits is used to make sure the correct units are used, but it is assumed the units between both Revit and Rhino match.

- **Lines** – For a Line, the two control points stored in the LyrebirdCurve are translated to a Revit XYZ object. These two XYZ objects are used to create a line using Line.CreateBound(pt1, pt2).
- **Arcs** – For an Arc, the three control points, stored as start point (pt1), mid-point (pt2), and end point (pt3) are converted to Revit XYZ objects. Then Arc.Create(pt1, pt3, pt2) is used to generate the arc. Note the sequence for Revit arc creation is start, end and interior point.
• Circles – A circle is created using two arcs, hence the 5 control points that are stored from Grasshopper. The first of the two arcs are created as Arc.Create(pt1, pt3, p2) just as the arc above this is created, and then the second arc is created as Arc.Create(pt3, pt5, pt4).

• Spline – Splines are created depending on the type of curve (Hermite/interpolated curve or NURBS curve) that is being pushed from Rhino to Revit. Because as we found in the geometry translation above (page?????) the NurbSpline.Create(ControlPoints, Weights) isn’t as reliable as NurbSpline.Create(controlPoints, weights, knots, degree, false, true) so the latter is used. For a Hermite spline, or an interpolated curve as it’s more frequently called, the HermiteSpline.Create(ControlPoints, Periodic) from the LyrebirdCurve is used.

See the CreateObjects method in the Revit2016Server.cs file in the Lyrebird Github repository.

File Free Communication

Communication
This is as far as we took the geometry in Lyrebird for two reasons. One, Lyrebird had a limited scope in that it just needed to get Revit and Grasshopper to talk. I present this example first because it showed me how valuable geometry could be to interoperability, but in the end that’s all this project was, interoperability. Two, the geometry itself was a byproduct of trying to get the two software platforms communicate without another file to keep track of. We had one project where some of the design parameters were delivered as a CSV file and there was some confusion as to what the most up to date CSV file was that isn’t necessarily the same as with a model. This confusion wasn’t revealed until late in the process after construction had begun and could have been a costly fix if it wasn’t discovered before that part of the project was being constructed. It’s often be easier to look at a model and see if it’s up to date than it would be to look at a collection of comma separated values and see if it is up to date. Otherwise the same interoperability could have been done using a structured text file like XML or even a simple CSV file, but the immediacy of communication together with eliminating the extra file was desirable.

Foundation
The communication mechanism to get the two pieces of software to talk to each other was Windows Communication Foundation (WCF). I don’t know if I could have pulled this off without the excellent example provided by Victor Chekalin through Jeremey Tammik’s blog.

As a brief synopsis, WCF allows you to create a service, and then send data messages from one end point to another via this service. This service can be hosted via IIS, or through an application as I did. It allows two-way communication through the service, so even though Lyrebird the service is created through Revit, data can stream from Revit to Grasshopper or from Grasshopper to Revit.

Lyrebird is contained in three parts: LyrebirdCommon where the WCF service interface is defined, GHClient where the data is organized and prepared for transferring to Revit, and RevitServer (no relation to Revit’s actual Revit Server) that implements the LyrebirdService
(where the bulk of the work is done) and maintains the task manager for tasks pushed from Grasshopper to Revit.

**Figure 10: Lyrebird’s WCF Organization**

**LyrebirdCommon**
LyrebirdCommon is the class library that connects Revit and Grasshopper by providing a communication language (LyrebirdPoint and LyrebirdCurve) and defining the communications that can occur through this WCF application. LyrebirdCommon has three parts, starting with *LyrebirdService Interface*, where the ServiceContract and OperationContracts are defined for the WCF service. This is where all of the types of messages are defined, from getting all Family and Types to select from in Grasshopper to formatting the information in Grasshopper and sending it to be created in Revit. Next, *LyrebirdObjects* is where the geometry objects LyrebirdPoint and LyrebirdCurve exist, as well as other classes to store and organize data like RevitObject, the class that represents an element to be created in Revit. And the last part of LyrebirdCommon is the LyrebirdChannel where the WCF service host and its address are defined per the selected Revit version. The Lyrebird channel is set up to match the service created in Revit, and is what Grasshopper uses to define the communication pipeline.

LyrebirdCommon just exists to make sure there’s a common language that happens between the applications that connect through it. Looking back on the project today I can see there are probably areas that are part of the Revit side of the application that should be moved to LyrebirdCommon. In particular I’m thinking that it doesn’t make sense to have the address defined in two places, once in the Revit where the service is actually created, and once in LyrebirdCommon where the channel through which Grasshopper communicates is defined.

**Lyrebird RevitServer**
This part of the project, named not because it bares any resemblance to Revit Server, but because it acts as the Server in the WCF setup as compared to the Client that is the Grasshopper side. Perhaps RevitService would have been a better option, but as it’s only really visible to someone looking at the code it will probably remain as it is. This is where all of the heavy lifting for Lyrebird takes place. It implements the LyrebirdService (from LyrebirdCommon’s interface) and is where all of the functions for reading or writing to the Revit document occur.
This is also where the actual WCF service is created and toggled on and off. The major differences in how the WCF is implemented here versus what’s shown in the Building Coder blog is the service model binding. The snippet of code that shows the service endpoint creation shows it using a WSHttpBinding and he specifies that you have to start Revit with the Run As Administrator option in order for it to work. While I have admin rights on my computer, not everyone in a work environment does, so this was a major stumbling block to having Lyrebird act as a viable solution. While I won’t profess to understand everything about how or why, after I spent a couple of weeks of toying around with different WCF options, I finally settled on the NetNamedPipeBinding instead of WSHttpBinding or any of the others because it allowed me to run the service without needing to Run As Administrator.

Lyrebird uses Revit’s OnIdling event, and I’ve set the Lyrebird Service to be turned off by default because it will keep using one of your CPU threads at a constant 100%. While my computer may have 12 threads and I’m rarely using them all, I still felt it was good practice to only turn it on when necessary.

The other major aspect of this side of the pipeline was setting up the TaskContainer to manage the communication queue from Grasshopper into Revit. I believe the one part of the Victor’s example that I used verbatim. It works wonderfully and I’ve never needed to make any changes to it.

All other aspects of the Revit side of things have to deal with the User Experience / User Interface (UX/UI). This project contains the various forms and their associated content that is the primary way we interact with Lyrebird. Stripping data from the instantiated objects (to break the update link), specifying the Settings, or even just modifying the data is possible through the Revit API side of things. While outside the scope of this topic, there may be some interesting things to see as far as how the buttons were implemented into the ribbon UI and how the button icon is toggled when the service is off versus on.
**Lyrebid GHClient**
The Grasshopper side of the plugin is mainly concerned with structuring the information coming in from the inputs along with the properties defined through the forms for selecting a Family Type and specified parameters. The bulk of the files in the GHClient project are forms that are there for bettering the UI/UX of using the plugin more than what it’s doing internally.

There are some interesting things happening here as Lyrebird starts converting the geometry from Rhino to Lyrebird, with the expectation that it’s going to Revit. Part of this involves understanding how Revit’s curves work compared to Rhino, things like knowing there is no polycurve in Revit or that you can’t close a spline with C2 continuity. When the GHClient is organizing curve data in Revit, it will explode polylines into its constituent parts and will split closed splines before pushing it to the RevitObject that gets sent to Revit. Because of this setup, Lyrebird is very dependent on Rhino’s geometry library to manage all of these operations since some of them just don’t exist in Revit (like splitting the spline).

*You can find the project here:* [https://github.com/logant/Lyrebird](https://github.com/logant/Lyrebird)

**Final Thoughts**

**On Necessity**
While we have set out to build our own geometry library, knowing full well that there are other’s out there that may do the job, we are interested in the exploration as much as the outcome. The project that I present (Lyrebird) doesn’t need a legit geometry library as much as a structured way to transfer the data from one application to another. However, in laying the groundwork for this now, we enable ourselves to move more quickly whenever a new application comes to the forefront. As great a tool as Revit and Dynamo are, they’ll inevitably be replaced at some point in the future with something better, and we’ll be prepared when that happens.