CES226095

Civil 3D Data Mining with Dynamo

Andrew Milford
Autodesk

Learning Objectives

- Learn how to create custom Civil 3D nodes for Dynamo using C# and the zero-touch interface
- Interrogate your Civil 3D geometry and models from within Dynamo
- Leverage the flexibility of Dynamo to extract Civil 3D model data.
- Create Civil 3D geometry from within the Dynamo workspace

Description

In this class you will learn how to connect your Autodesk Civil 3D models into the Dynamo environment using C# and the "zero-touch" interface. Learn how to connect to a running Civil 3D application using the Civil 3D COM API and create customised nodes for common Civil 3D object types. We will then import the zero-touch library into Dynamo Studio, where we will apply the new nodes into a Dynamo canvas and extract Civil 3D model data for analysis or construction. Finally, we will investigate how we can transfer data from within Dynamo back into our current Civil 3D project. This session is intended for those wishing to extend their Civil 3D customisation skills, leveraging the power of the Dynamo interface.

Speaker

Andrew Milford is a Senior Civil Infrastructure Implementation Consultant for Autodesk and is responsible for providing technical and business consulting, ensuring customers achieve successful adoption of Autodesk's Infrastructure Solutions across the Asia/Pacific region. Prior to joining Autodesk, Andrew gained over 24 years of design experience in the civil infrastructure industry, working as a geometric road designer for large consulting companies such as SKM/Jacobs, Arcadis Consulting, and AECOM. Andrew’s experience extends from designing large highways, tunnels, and interchanges down to smaller subdivision work using a variety of different design and drafting software packages. He is an AutoCAD Civil 3D Certified Professional and loves diving deep into AutoCAD Civil 3D software to develop and automate processes through scripts, AutoLISP, .NET API C# and VB, and Python.

Introduction

Civil 3D is a civil engineering design tool for producing high-quality 3D models for construction and documentation.
Dynamo is a visual programming and scripting tool that works extremely well with Excel and Revit.

In today's competitive environment, there is now a strong emphasis on leveraging automation to perform more and more design and documentation tasks, maintaining high-quality deliverables delivered within short timeframes. Typically, the development of these automation tools is reserved for the very few individuals within companies and require a mid-to-high degree of programming knowledge to create even the most basic of programs.

By combining the modelling strengths of Civil 3D, the existing visual programming capabilities of Dynamo and adding a touch of our own Civil 3D API knowledge, this class will demonstrate practical workflows in extracting and creating Civil 3D & AutoCAD information through some customised Dynamo nodes and scripts.

Overview

This document has been split into 4 learning objectives, with each exercise building on the previous chapter's outcomes. Additionally, each chapter is split into two sections, the first detailing the require coding concepts to connect Civil objects into Dynamo, and the second half focusing on the application of those Dynamo nodes to extract (or create) model data.

Exercise 1 - Learn how to create custom Civil 3D nodes for Dynamo using C# and the zero-touch interface

The first objective is to understand how to setup a C# Civil 3D / Dynamo Zero-Touch Visual Studio project template, with some tips and tricks to create quick and re-useable code for future projects.
We will look at how to connect to the Civil 3D API using the COM interop using less than ten lines of code, as well as how to retrieve all of the current open documents.
Finally, we will add more information and meaning to our Dynamo nodes through the Override.ToString() function

Exercise 2 - Interrogate your Civil 3D geometry and models from within Dynamo

Following from Objective 1, we will extend our C# application and introduce some basic Dynamo objects representing Civil 3D objects, in particular – Surface, Alignments and Profiles.

We will learn how to extract information from our Civil 3D surfaces by creating a Dynamo Surface node, which will extract and manipulate this data for exporting into Excel, which we will then pass downstream to PowerBI for some powerful visual analysis.

Our Dynamo Alignment node will allow us to interrogate either a single alignment, or a filtered group of alignments, extracting key information (again, for exporting to Excel)

Finally, as a real-world example, we will create a Dynamo node containing a simple algorithm to extract detailed safety barrier setout information, relative to key control strings.

Exercise 3 - Leverage the flexibility of Dynamo to extract Civil 3D model data

In this exercise, we will attempt to leverage the most complex of all Civil 3D objects – Corridors, Baselines and Feature Lines.

We will setup a Dynamo Corridor node, which will give us access to the internal Baselines and Featurelines for uses such as reporting of 3D Feature Lines for construction (or other applications). These nodes will lay the foundation for us to be able to generate additional model information, using a combination of standard Dynamo nodes and Civil 3D information

Exercise 4 – Create Civil 3D / CAD geometry from within the Dynamo workspace

In our final exercise we will build on our learnings from the previous exercise and create nodes to assist in the creation of AutoCAD and/or Civil 3D geometry.

We will use Dynamo to place geometry relative to baselines, and finish by using a populated Excel spreadsheet to control the placement of objects.

Finally, as a bonus, we will use similar techniques to generate 3D solid linemarking from our corridor model, for importing into Navisworks

Software Requirements

For the exercises in this class, we will use the following software

- Civil 3D 2019
- Visual Studio Community 2017

Visual Studio Project Setup

Create a new Visual Studio project and name the project '**AU2018.Civil3D**'. Note that to leverage the Civil 3D 2019 libraries, the project target framework is set to .Net Framework 4.7.1
.

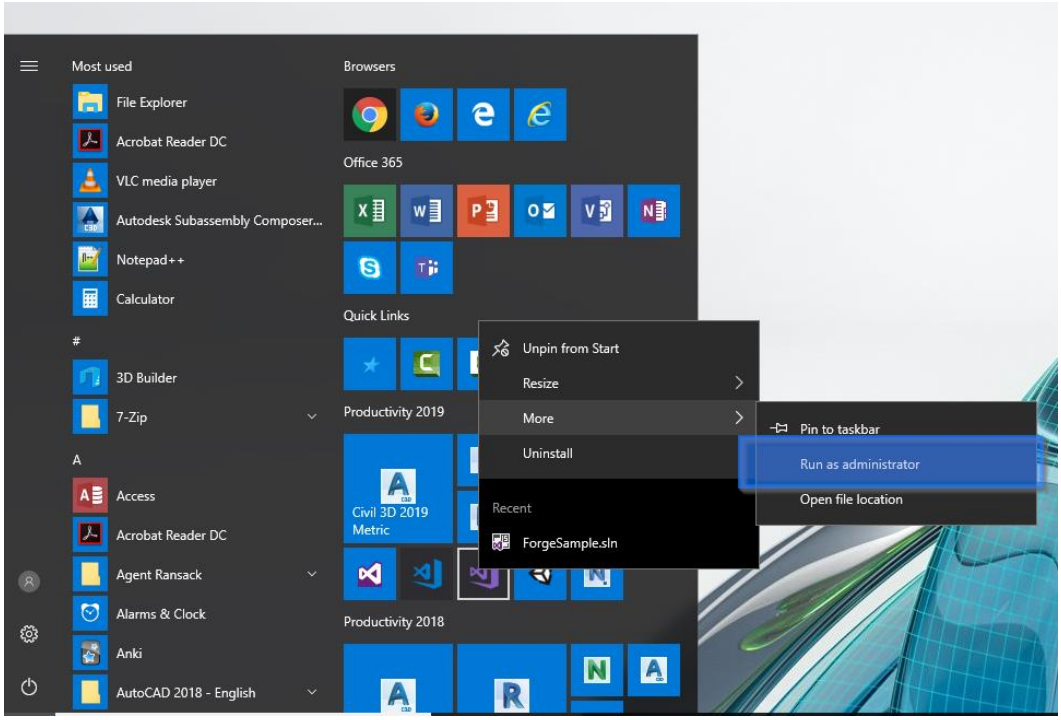


Figure 1: Run Visual Studio as Administrator

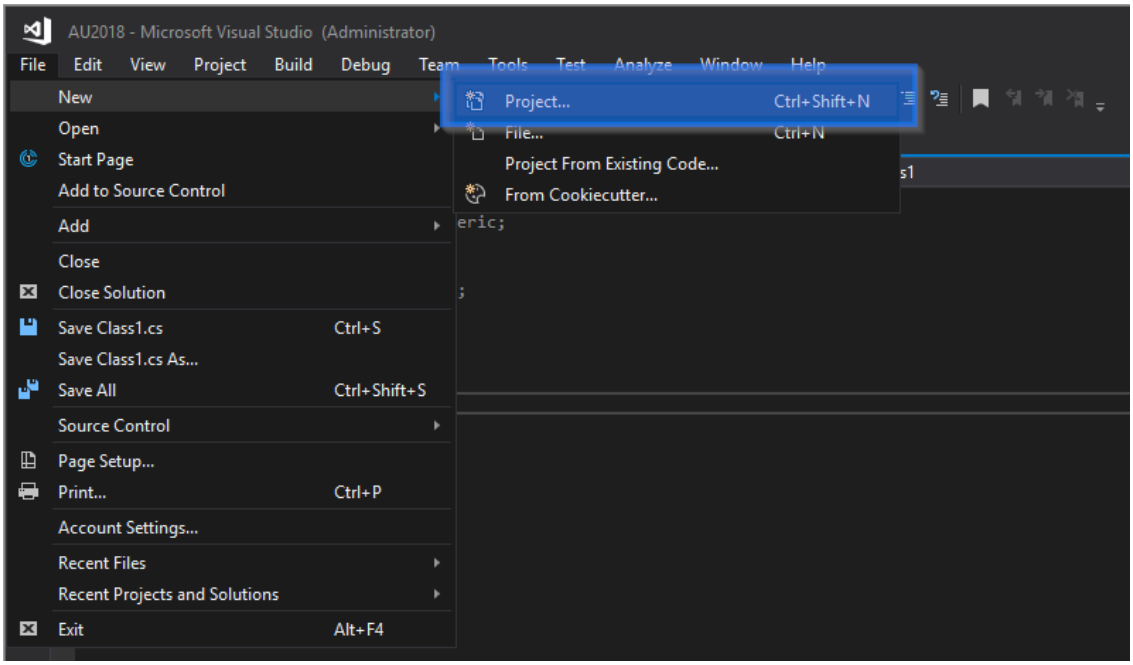


Figure 2: Create a new Visual Studio Project

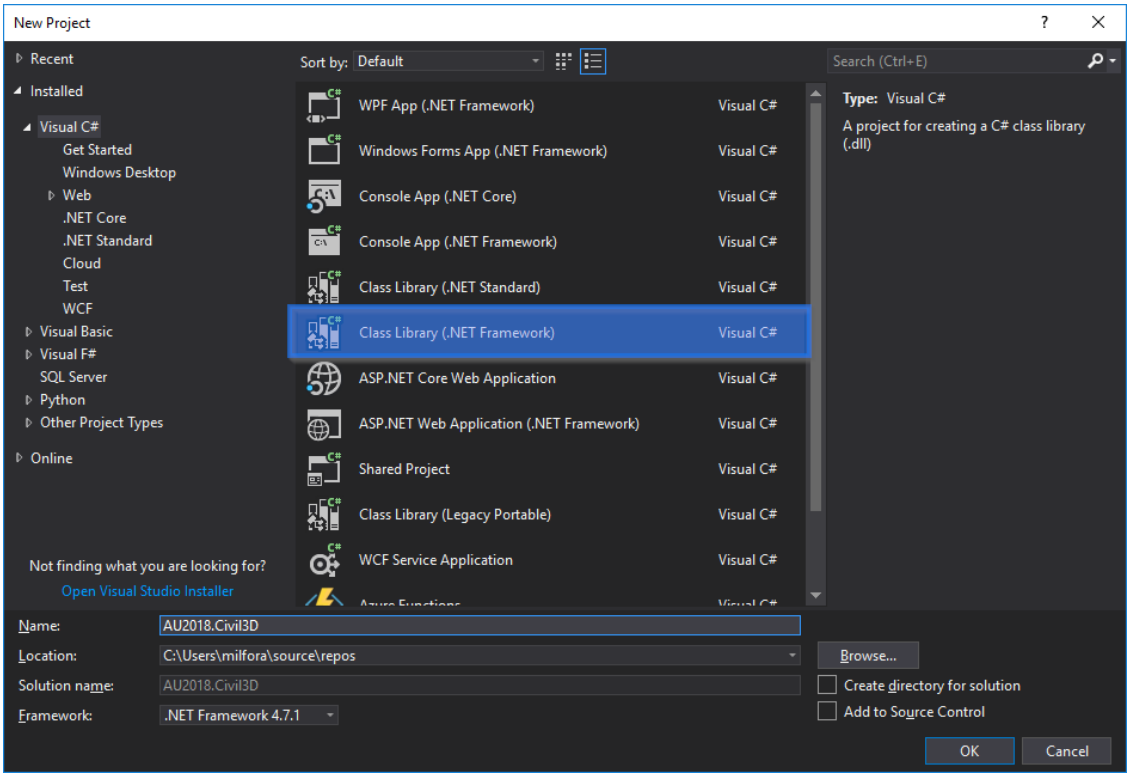


Figure 3: Create a Visual Studio Project - Class Library

Add Project References - Dynamo

Use NuGet to download packages from internet, right-click the ‘References’ in the Solution Explorer, and select ‘Manage Nuget Packages’

Using Nuget makes it easier to install references, as you do not need to have the original software installed

For this example, we need two Nuget packages

- *DynamoVisualProgramming.ZeroTouchLibrary* which depends on
- *DynamoVisualProgramming.DynamoServices* and will be downloaded automatically.

Make sure they match your Dynamo version (which in this example is 1.3.0)

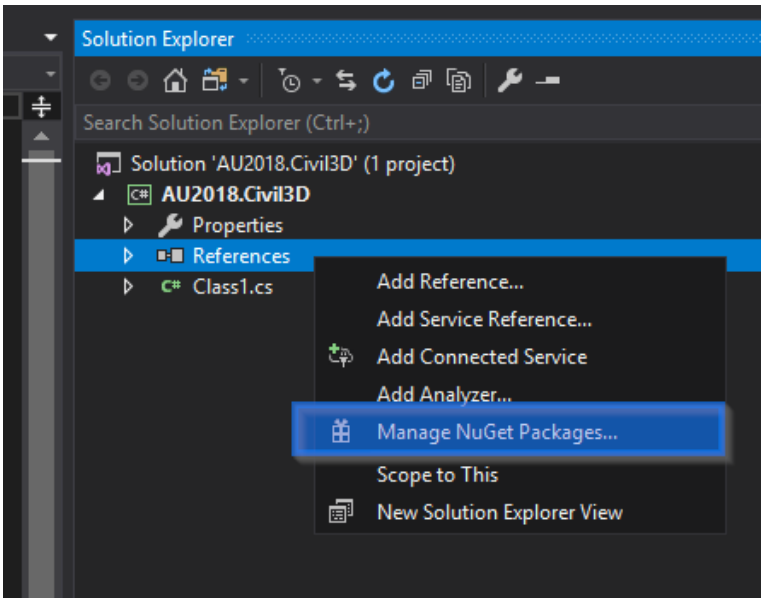


Figure 4: Manage Nuget Packages

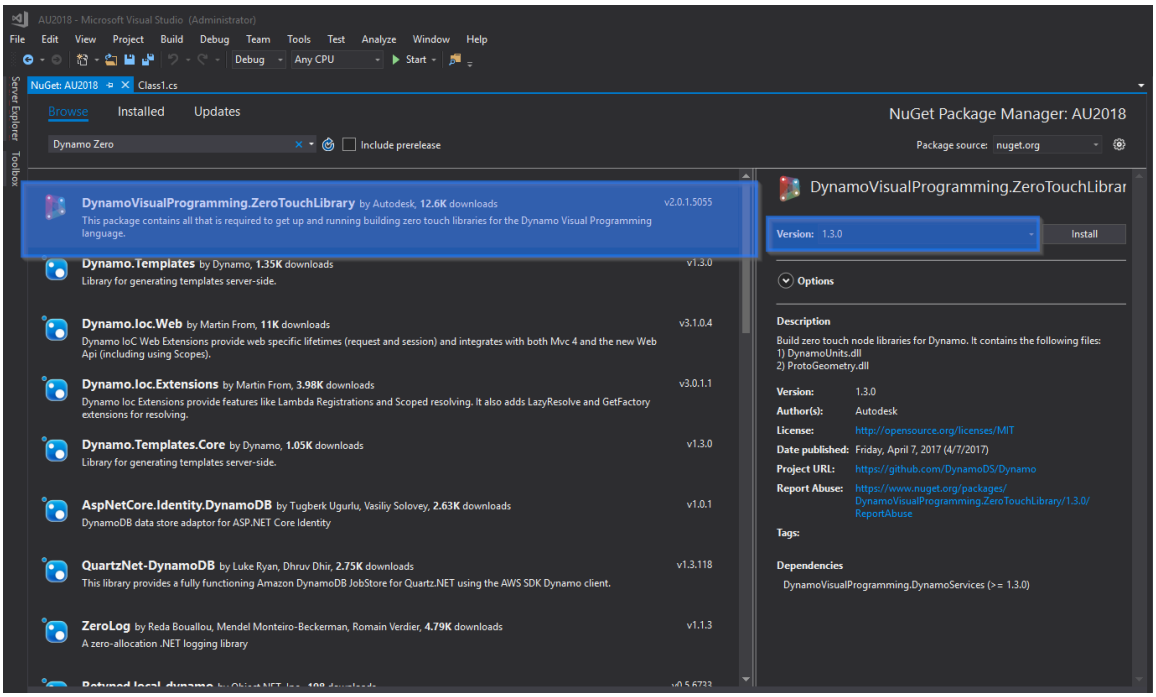


Figure 5: Dynamo Nuget Packages

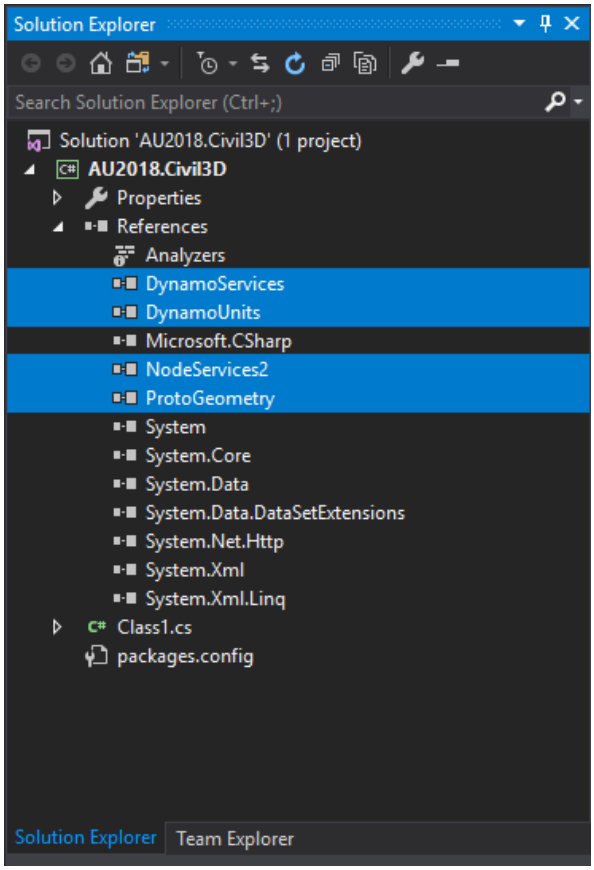


Figure 6: Dynamo References

Once the Dynamo References have been installed, select each item in turn and, in the Properties window, set the ‘Copy Local’ option to ‘False’

The Reference Files are installed to the following location
C:\Program Files\Dynamo\Dynamo Core\1.3

- DynamoServices.dll
- DynamoUnits.dll
- NodeServices2.dll
- ProtoGeometry.dll
- *DynamoCore.dll*

Add Project References – Civil 3D

You can either set a reference to existing Civil 3D libraries located in the Civil 3D installation folder, or you can perform a manual installation.

The typical location for the Civil 3D installation folder is
‘C:\Program Files\Autodesk\AutoCAD 2019\C3D’

For a manual installation of AutoCAD Civil 3D Reference, the download location is located at
https://www.autodesk.com/developer-network/platform-technologies/autocad/objectarx
https://www.autodesk.com/developer-network/platform-technologies

The following locations point to locally installed library files
C:\Program Files\Autodesk\AutoCAD 2019\C3D\Autodesk.AECC.Interop.UiLand.dll

- AECCApplication

C:\Program Files\Autodesk\AutoCAD 2018\C3D\Autodesk.AECC.Interop.Land.dll

- AeccAlignment
- AeccSurface

C:\Program Files\Autodesk\AutoCAD 2018\C3D\Autodesk.AECC.Interop.UIRoadway.dll

- AeccRoadwayApplication
- AeccRoadwayDocument

C:\Program Files\Autodesk\AutoCAD 2018\C3D\Autodesk.AECC.Interop.Roadway.dll
C:\Program Files\Autodesk\AutoCAD 2019\Autodesk.AutoCAD.Interop.dll

- AcadApplication
- AcadDocument

C:\Program Files\Autodesk\AutoCAD 2019\Autodesk.AutoCAD.Interop.Common.dll
C:\Program Files\Autodesk\AutoCAD 2019\ACA\Autodesk.AEC.Interop.UIBase.dll
C:\Program Files\Autodesk\AutoCAD 2019\ACA\Autodesk.AEC.Interop.Base.dll

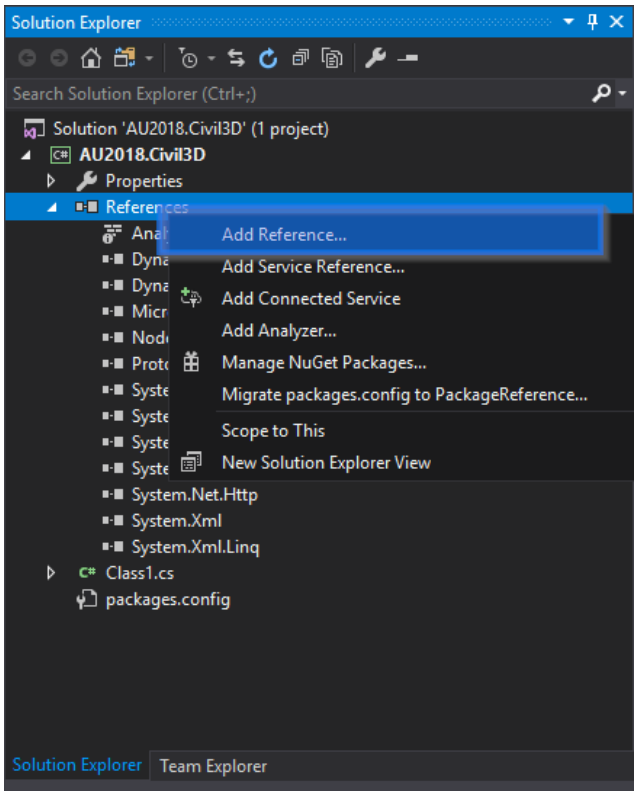


Figure 7: Add a Civil 3D Reference

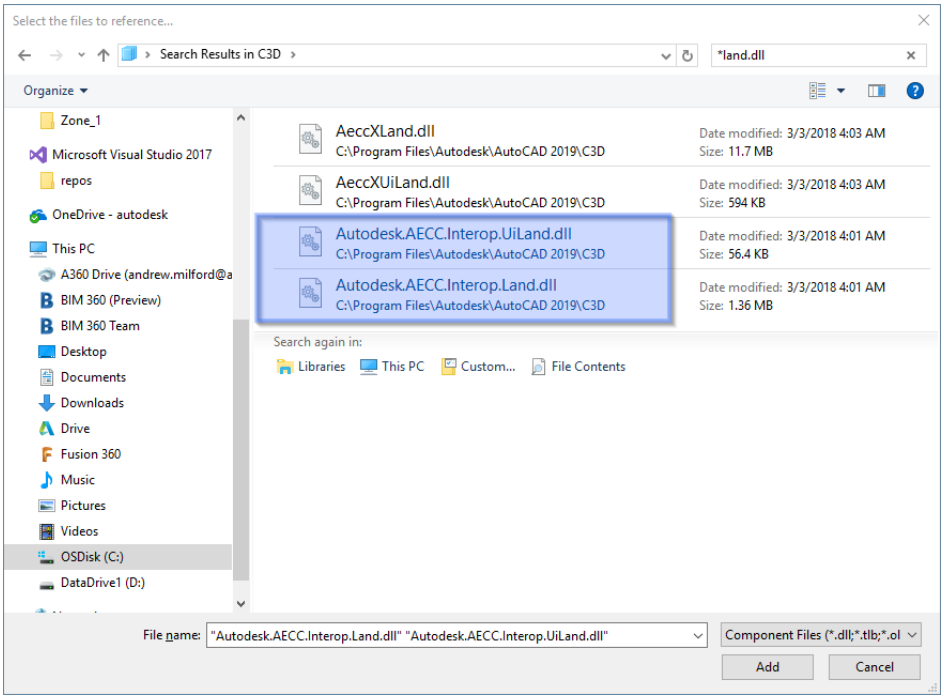


Figure 8: Civil 3D Land Reference installation locations

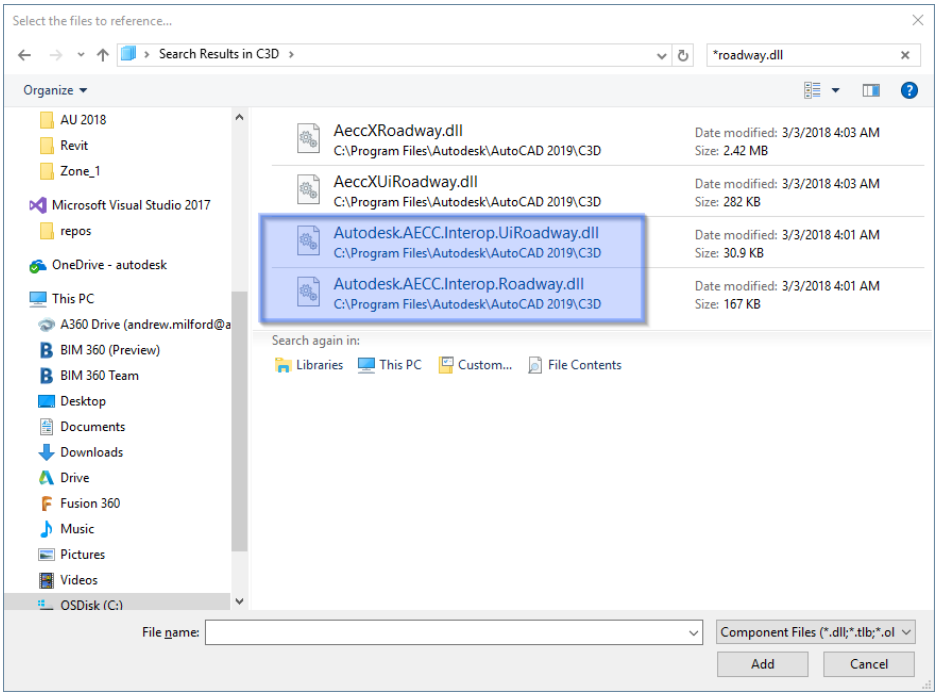


Figure 9: Civil 3D Roadway Reference installation locations

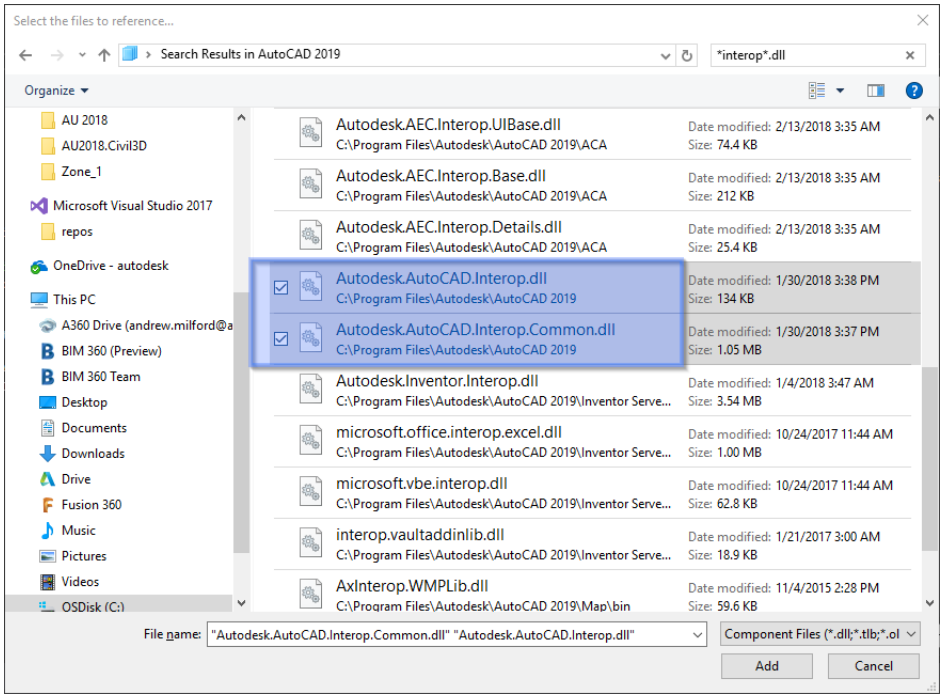


Figure 10: Civil 3D AutoCAD Reference installation locations

Packages

Dynamo Packages are created automatically when creating a local package or can be edited manually. For more information of packages, refer to the link below:
http://primer.dynamobim.org/en/11_Packages/11-4_Publishing.html

To create a package manually, add a '*pkg.json*' file to the project (located in the **Web > JSON** file folder). We will automatically copy this to the Dynamo packages folder during debugging sessions.

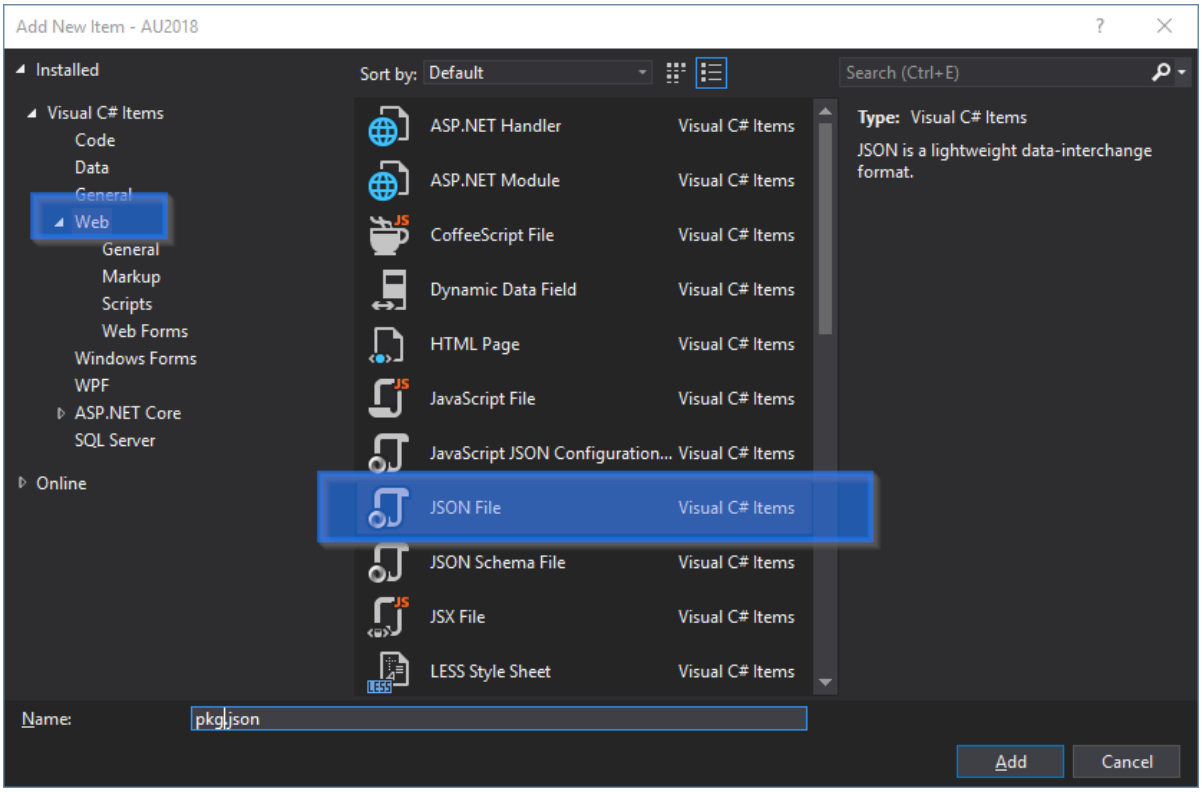


Figure 11: Adding a 'pkg.json' file to the project

Some sample boilerplate code can be found in the following location
https://github.com/DynamoDS/DynamoSamples/blob/master/dynamo_package/Dynamo%20Samples/pkg.json

For this example, we will use the code below inside the JSON file.

```
{
  "license": "",
  "file_hash": null,
  "name": "AU 2018 - Civil 3D",
  "version": "1.0.0",
  "description": "Civil 3D sample nodes for AU 2018",
  "group": "AU 2018",
  "keywords": null,
  "dependencies": [],
  "contents": "",
  "engine_version": "1.3.0.0",
  "engine": "dynamo",
  "engine_metadata": "",
  "site_url": "",
  "repository_url": "",
  "contains_binaries": true,
  "node_libraries": [
      "AU2018.Civil3D, Version=0.1.0.0, Culture=neutral, PublicTokenKey=null"
  ]
}
```

Build Events and Debugging

Build events are setup to automatically copy files when building or debugging projects.

Open the Project Properties panel (Right click the project name in the Solution Explorer, select Properties) and paste the following two lines into the Build Events > Post-build event command line

```
xcopy /Y "$(TargetDir)*.*" "$(AppData)\Dynamo\Dynamo Core\1.3\packages\$(ProjectName)\bin\"
xcopy /Y "$(ProjectDir)pkg.json" "$(AppData)\Dynamo\Dynamo Core\1.3\packages\$(ProjectName)"
```

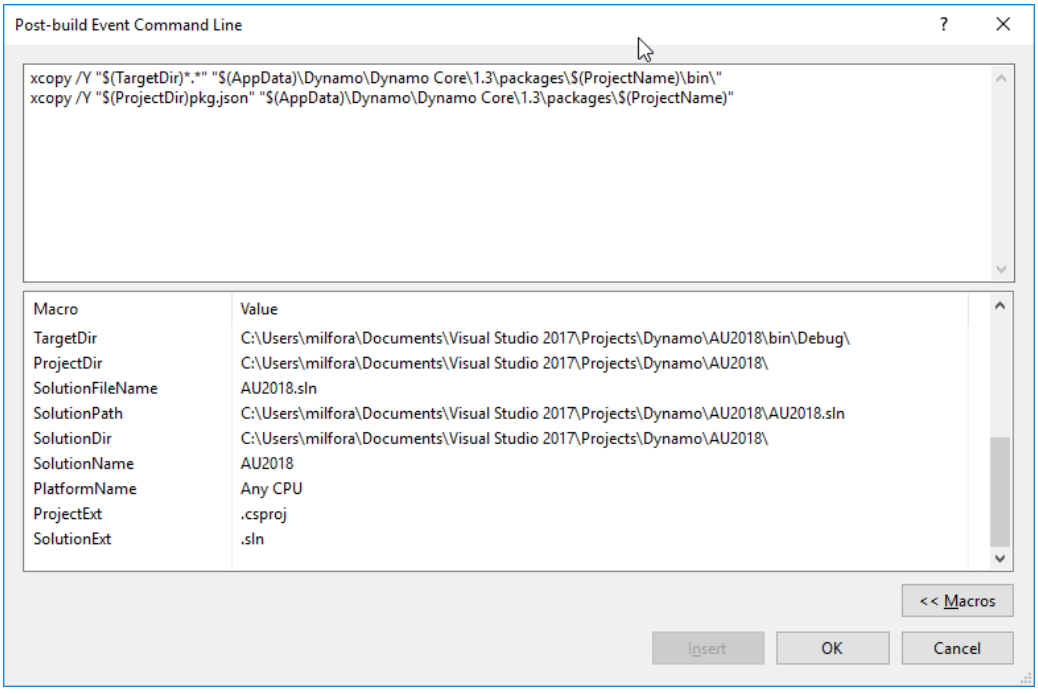


Figure 12: Setup Post-build events

In the debug panel, we will associate the *DynamoSandbox.exe* to run when we are debugging the program.

In the Properties > Debug > Start External Program, click the 'Browse' button and select the Dynamo Sandbox executable from the installation folder:

'C:\Program Files\Dynamo\Dynamo Revit\1.3\DynamoSandbox.exe'

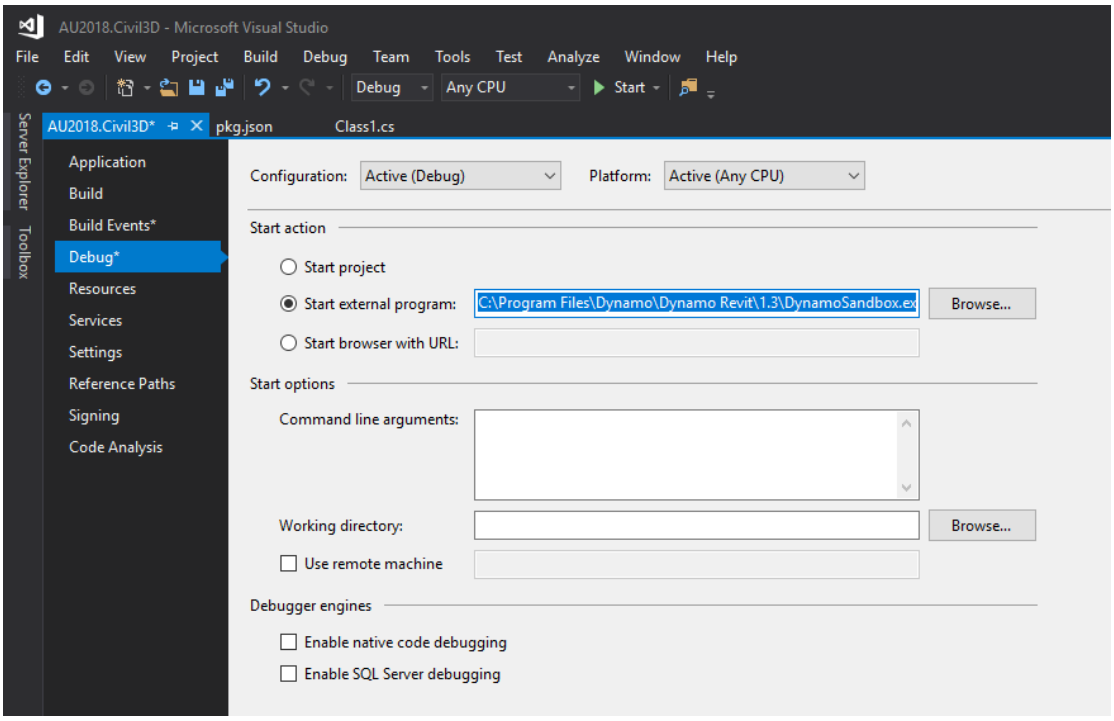


Figure 13: Debugging using DynamoSandbox.exe

If you are setting up a Dynamo project for the first time, it is good practice to set up a small test method to ensure the program is working as expected.

The following code is a test debug setup with a basic sample method from the Dynamo website.

Add a new class to the project, called 'Test', and add the following code:

```
namespace AU2018.Civil3D
{
    public static class Test
    {
        public static double AddNumbers(double num1, double num2)
        {
            return num1 + num2;
        }

    }
}
```

Note that we change the class type to a '`public static`'.

This results in the new node '**AddNumbers'** appear in the AU2018 library, under the long structure 'AU2018 > Civil3D > AU2018 > Civil 3D >

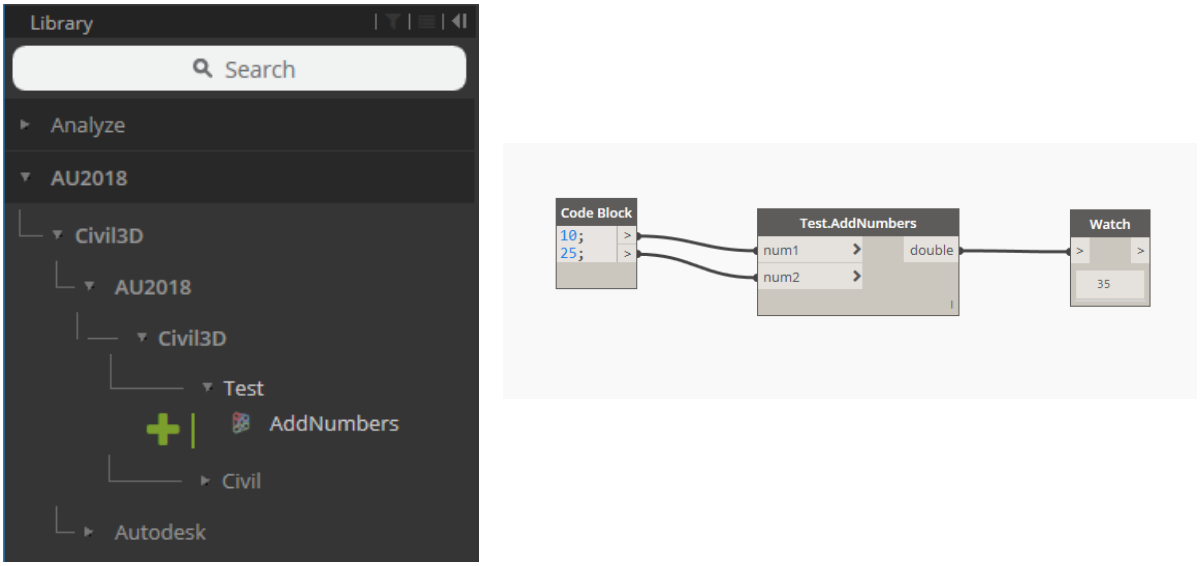


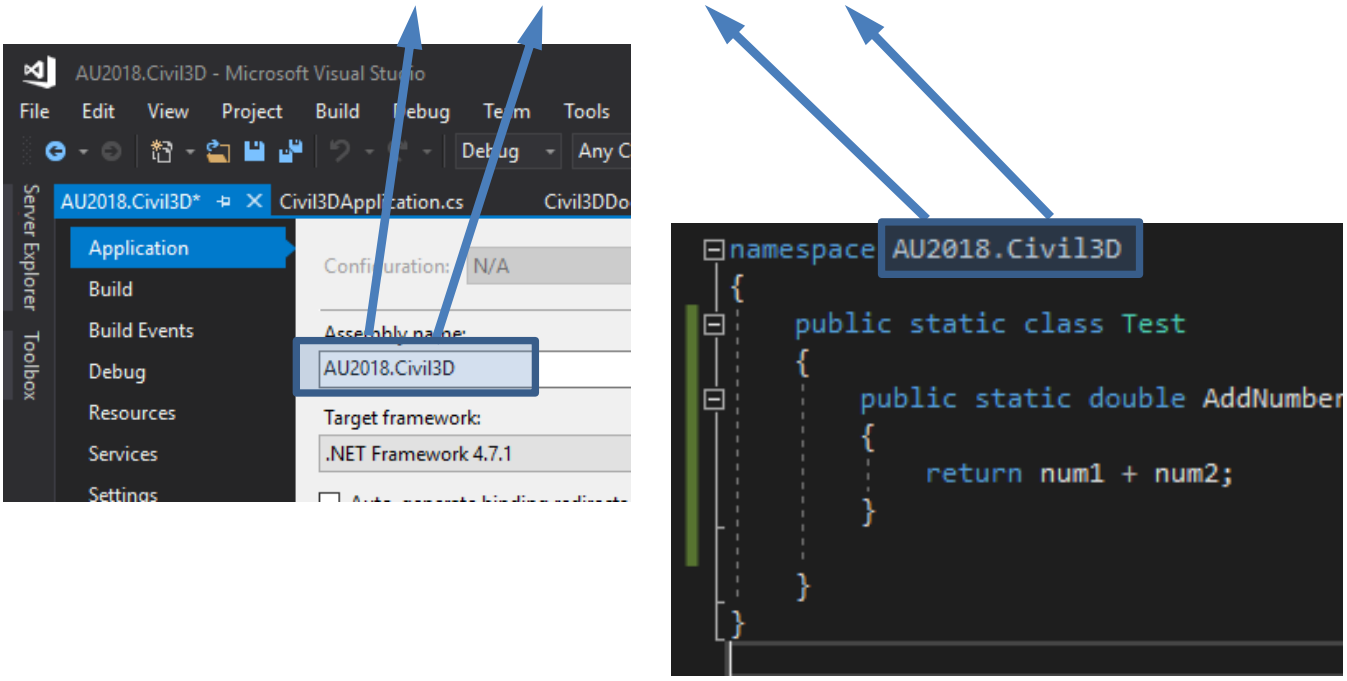
Figure 14: Dynamo test method node

In the next section, we will look at how to customise and shorten the path to the Dynamo custom nodes

Library and Node Naming

Dynamo will automatically format the code names inside of Dynamo for us, in the format:

AU2018 > Civil3D > AU2018 > Civil3D >



The first two levels represent the assembly name (the DLL), the last two are derived from the namespace, which in our case are identical.

To fix this, we can do either of the following things:

1. Change the project assembly name (and change the corresponding `"node_libraries"` parameter value in pkg.json)
2. Add an '*AU2018.Civil3D_DynamoCustomization.xml*' file to the project

It is recommended to use Option 2 (adding the XML file), as this will keep the assembly name and namespaces unchanged.

Create the XML file '*AU2018.Civil3D_DynamoCustomization.xml*' in the project root and copy the contents below into the file.

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>AU2018.Civil3D</name>
  </assembly>
  <namespaces>
    <namespace name="AU2018.Civil3D">
      <category>AU2018</category>
    </namespace>
    <namespace name="AU2018.Civil3D.Civil">
      <category>AU2018</category>
    </namespace>
  </namespaces>
</doc>
```

Select the file in the Solution Explorer, then set its '*Copy to Output Directory*' property to '*Copy always*'.

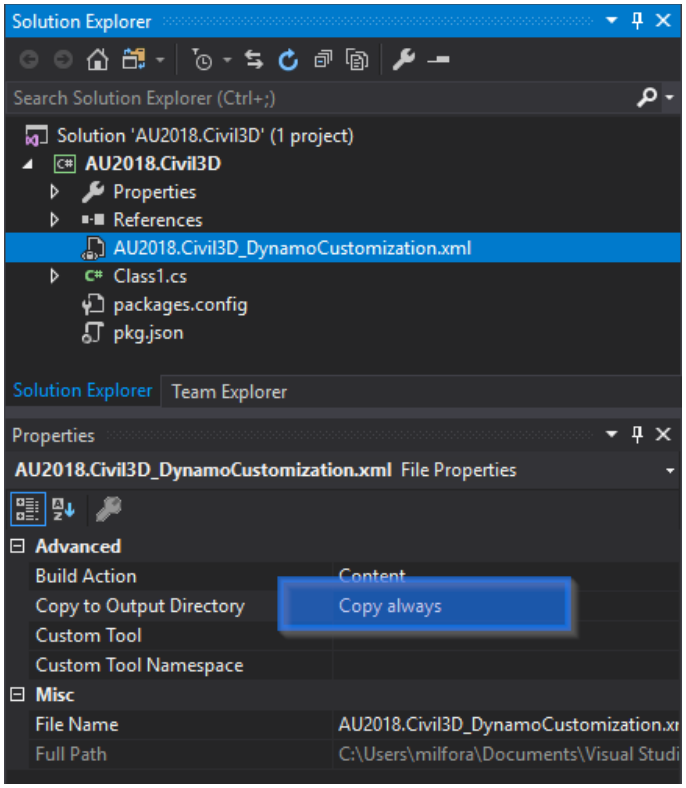


Figure 15: Copy XML file to project output directory

Let's start coding!

Learning Objective 1 - Learn how to create custom Civil 3D nodes for Dynamo using C# and the zero-touch interface

This section contains the most important part of the lesson, the initial project setup. It is desirable to have a neat data structure in order to get the most out of the Dynamo Interop.

Civil 3D Object Hierarchy

The Civil 3D COM API can be found through the help menu from within Civil 3D, navigate to Help – Developers Guide – Legacy COM API

In this section we will focus on two key components of the COM API, the Civil 3D Application object and the Document object.

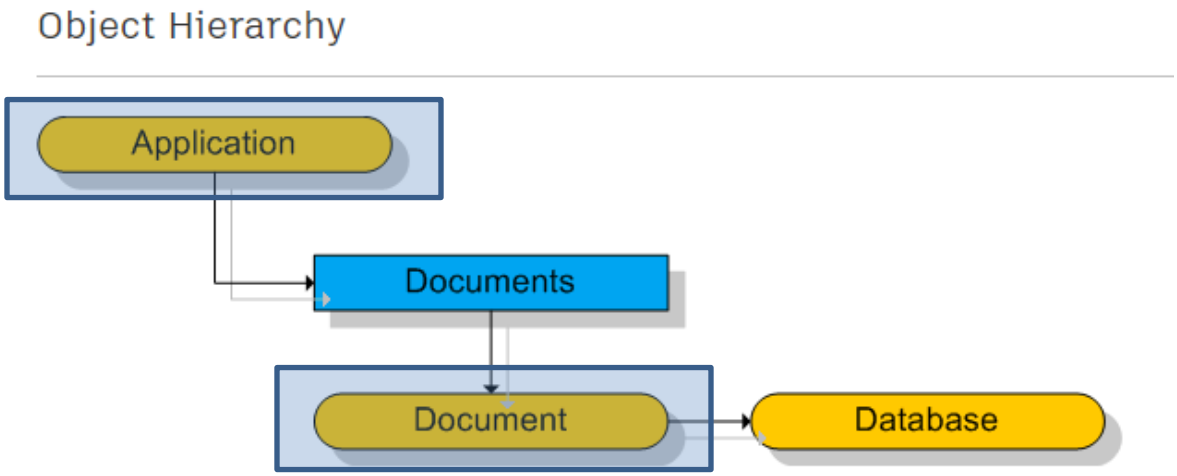


Figure 16: Civil 3D Application and Document Objects

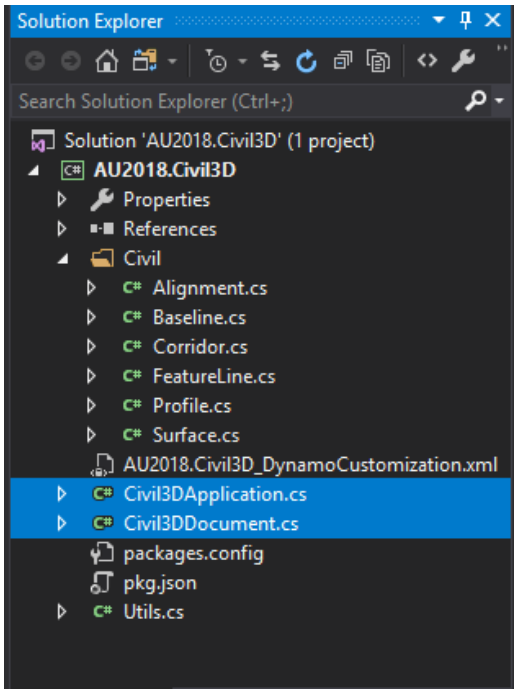
Visual Studio 2017

Every Civil 3D class created will contain, as a minimum:

- Private / Public Properties
  - These properties will be assigned values at runtime through the objects constructor, and will include information such as Object Name, Length, StartStation, EndStation etc (where applicable)
  - Each class that represents a Civil 3D object will contain a property representing the 'AeccXXX' object (for example, the 'Surface.cs' class will contain a property of type AeccSurface called '_surface')
- Constructor
  - The constructor will initialise an object when created in Dynamo
- Methods
  - Each class will include, in addition to its own custom methods, a public override of the ToString() method. This is to provide the end-user of Dynamo useful output information about the node, and what data it currently contains.

New Classes

- Civil3DApplication.cs
- Civil3DDocument.cs



Civil3DApplication.cs

This is the class which will establish the connection between Civil 3D and Dynamo

Declare constants

```
// AutoCAD Application
const string progID = "AutoCAD.Application";

// Get a Civil 3D 2019 Roadway Application (preferred - as this includes corridors)
const string civAppName = "AeccXUiRoadway.AeccRoadwayApplication.13.0";
```

In the above code, the `"AeccXUiRoadway.AeccRoadwayApplication.13.0"` refers to the Civil 3D 2019 version. If you require building the project against older versions, change the 13.0 to the relevant version, such as:

- 12.0 Civil 3D 2018
- 11.0 Civil 3D 2017
- 10.0 Civil 3D 2016 etc.

To determine the version you are running, navigate to the Civil 3D installation folder '*C:\Program Files\Autodesk\AutoCAD 2019\C3D*', and search for the relevant dll (i.e. Autodesk.AECC.Interop.UiRoadway.dll), right-click and select 'Properties'
From the 'Details' tab, the File version or Product version will indicate the correct value.

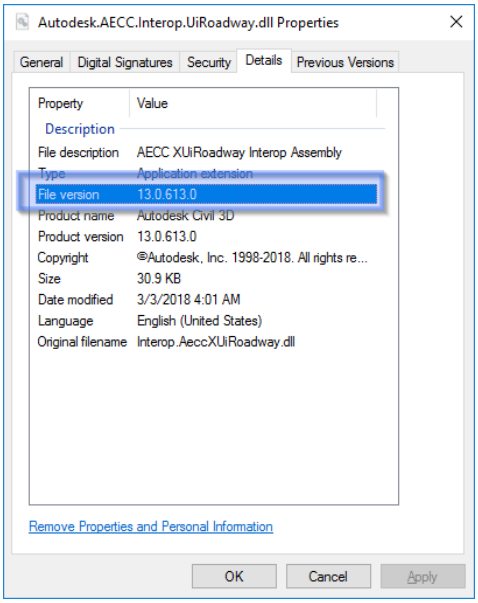


Figure 17: Determine dll file version

```
/// Constructor
public Civil3DApplication()
{
    this._civApp = this.GetApplication();
    this._activeDocument = this._civApp.ActiveDocument;
}


/// Get the active Civil 3D Application
internal AeccRoadwayApplication GetApplication()
{
    AcadApplication acadApplication = null;
    try
    {
        acadApplication = (AcadApplication)Marshal.GetActiveObject(progID);
    }
    catch
    {
    }

    if (acadApplication != null)
    {
        return acadApplication.GetInterfaceObject(civAppName);
    }

    return null;
}


/// Return a list of all the open Documents
public static IList<Civil3DDocument> GetDocuments(Civil3DApplication Civil3DApplication)
{
    IList<Civil3DDocument> docsList = new List<Civil3DDocument>();
    foreach (AeccRoadwayDocument doc in Civil3DApplication._civApp.Documents)
```

```
    {
        docsList.Add(new Civil3DDocument(doc as AeccRoadwayDocument));
    }
    return docsList;
}

/// Override the string output for the CivilApplication object
public static IList<Civil3DDocument
public override string ToString()
{
    return string.Format($"Civil3DApplication (ActiveDocument =
{this._activeDocument.Name})");
}
```

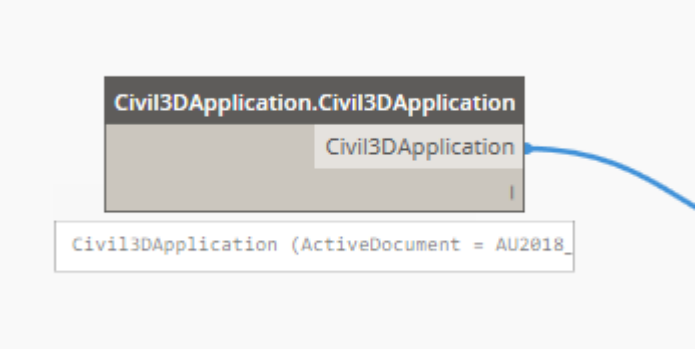


Figure 18: Dynamo node 'Civil3DApplication' with ToString() override

Civil3DDocument.cs

This class represents a single Civil 3D Document.

```
/// Constructor
internal Civil3DDocument(AeccRoadwayDocument civDoc)
{
    Name = civDoc.Name;

    this.CivilDocument = civDoc;
    this._surfaces = civDoc.Surfaces;
    this._alignments = civDoc.AlignmentsSiteless;
    this._corridors = civDoc.Corridors;
}
```

On creation of a new Civil3DDocument, the constructor above will create a list of Surfaces, Alignments and Corridors in the drawing, as well as a reference o the Civil 3D drawing object itself.

```
/// Return a list of all the Civil 3D Surfaces
public static IList<Civil.Surface> GetSurfaces(Civil3DDocument Civil3DDocument)
{
    IList<Civil.Surface> surfacesList = new List<Civil.Surface>();
    foreach (AeccSurface surf in Civil3DDocument._surfaces)
    {
        surfacesList.Add(new Civil.Surface(surf as AeccSurface));
    }

    return surfacesList;
}

/// Return a list of all the Civil 3D Alignments
public static IList<Alignment> GetAlignments(Civil3DDocument Civil3DDocument)
{
    IList<Alignment> alignmentList = new List<Alignment>();
    foreach (AeccAlignment align in Civil3DDocument._alignments)
    {
        alignmentList.Add(new Alignment(align as AeccAlignment));
    }

    return alignmentList;
}

/// Return a list of all the Civil 3D Corridors
public static IList<Corridor> GetCorridors(Civil3DDocument Civil3DDocument)
{
    IList<Corridor> corridorList = new List<Corridor>();
    foreach (AeccCorridor corr in Civil3DDocument._corridors)
    {
        corridorList.Add(new Corridor(corr as AeccCorridor));
    }

    return corridorList;
}
```

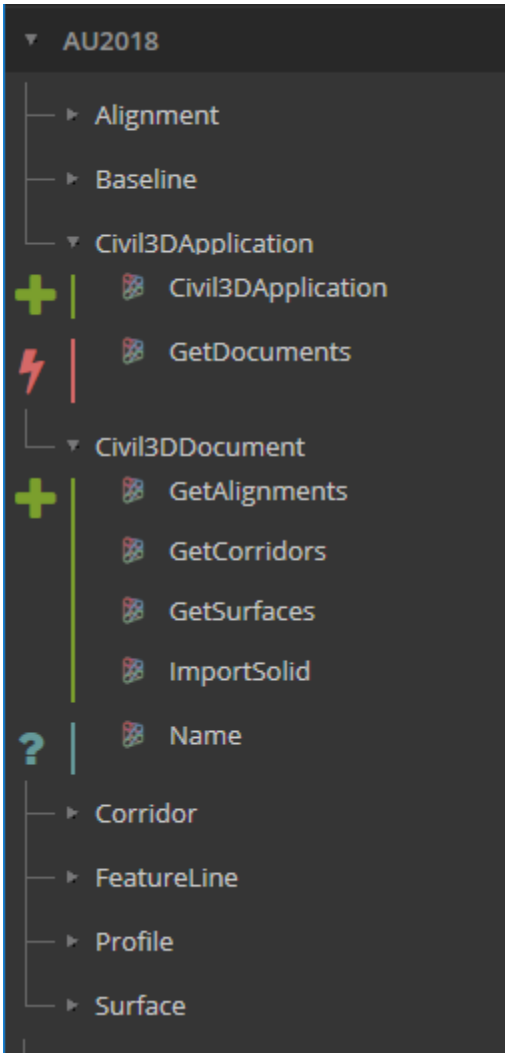
The ‘GetSurfaces’, ‘GetAlignments’ and ‘GetCorridors’ methods are similar in that they all take a Civil3DDocument as its input. The methos then cycles through all of the respective objects within the current drawing, finally returning a list of all the Surfaes, Alignments and Corridors, respectively.

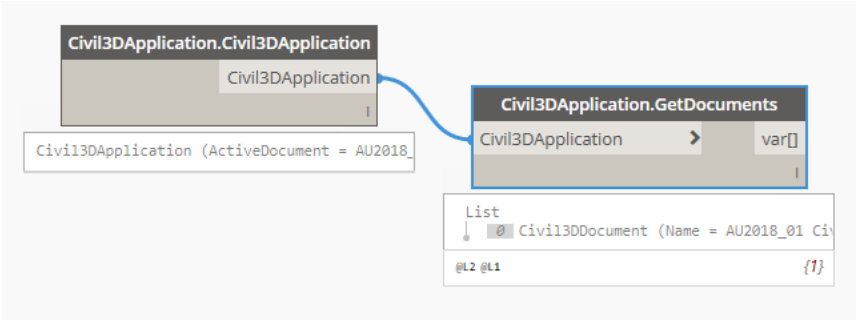
```
/// Override the string output for the CivilDocument object
public override string ToString()
{
    return string.Format($"Civil3DDocument (Name = {this.Name})");
}
```

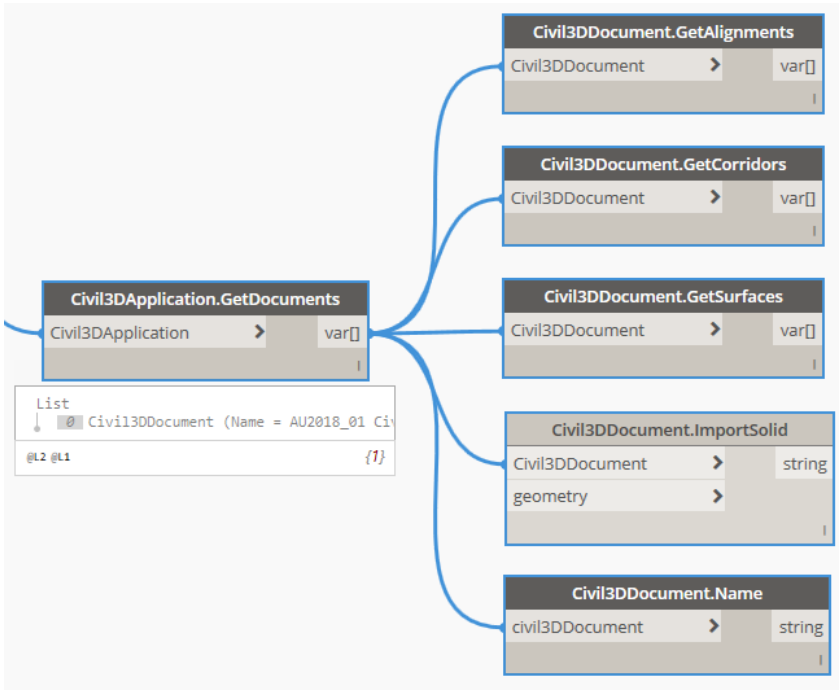
Dynamo Studio

When using Dynamo to connect to Civil 3D, it is important to remember the following:

- Civil 3D must be open to establish a connection
- Only one session of AutoCAD Civil 3D can be open, although you can have multiple drawings open in the single session
- The version of AutoCAD/Civil 3D open must match those of the referenced dll's in the Visual Studio project (2019 in our case)
- The CivilApplication.CivilApplication node must always be present at the start of the Dynamo graph, followed by the CivilApplication.GetDocuments







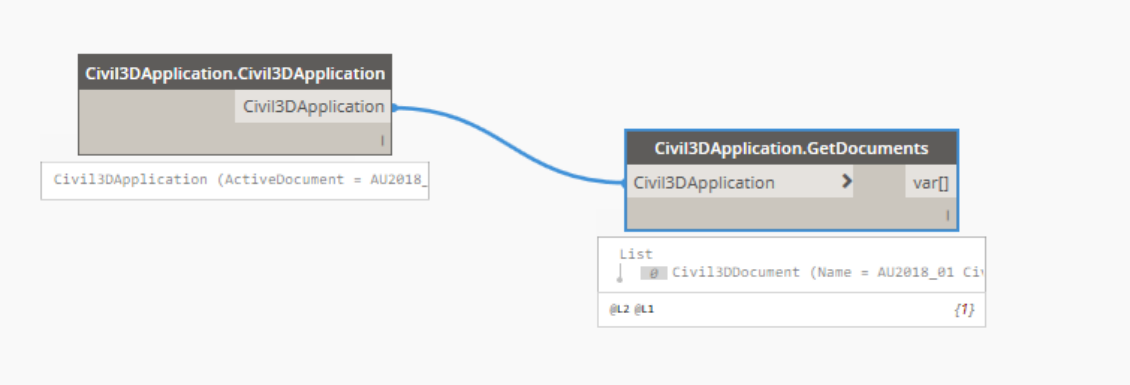


Figure 19: The CivilApplication node and the GetDocuments node

It is considered good practice to automate as much of the document selection process as possible. For example, if there is more than one drawing open in Civil 3D, the Dynamo GetDocuments will return all of the documents that are currently open. Since we can only work on one document inside of Dynamo, we need to setup a snippet to search for the drawing by name.

In the example below, the Civil 3D document name is extracted and compared against another string value (being the name of the drawing). The String.Contains node will allow this comparison and return a Boolean (True/False) value, which we then run through a List.FilterByBoolMask node, which will return two lists, one which contains all of the documents that match the string pattern, and another which does not contain the documents.

In our case, we can simply discard the ‘False’ outputs, and focus on the selected document. This approach to setting up Dynamo scripts allow for more flexible and robust scripts, reducing the room for error.

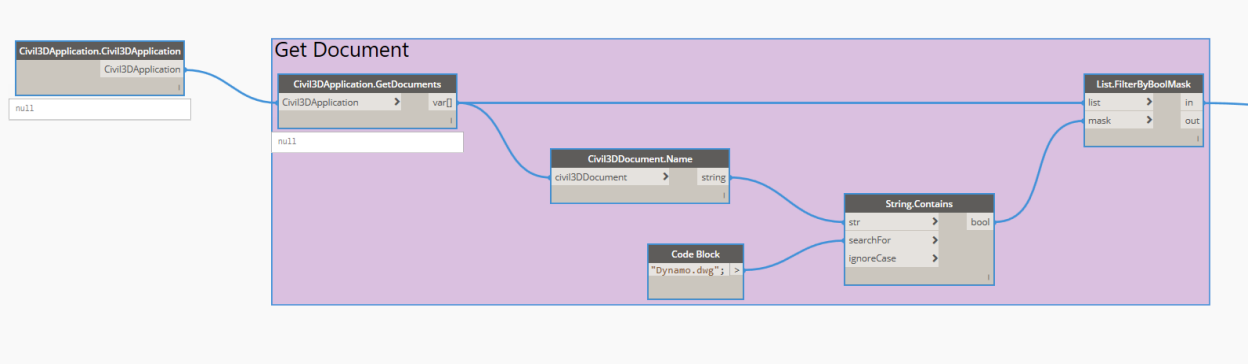


Figure 20: Selecting Documents by string filter

Learning Objective 2 - Interrogate your Civil 3D geometry and models from within Dynamo

Following on from Objective 1, we will now create three more classes, Surfaces, Alignments and Profiles.
The three classes all follow a similar structure to the classes created in Objective 1, and contain Civil 3D Aecc* object properties, a constructor, unique methods and a ToString() override

Civil 3D Object Hierarchy

In this section we will focus on three components of the COM API, Surfaces, Alignments and Profiles.

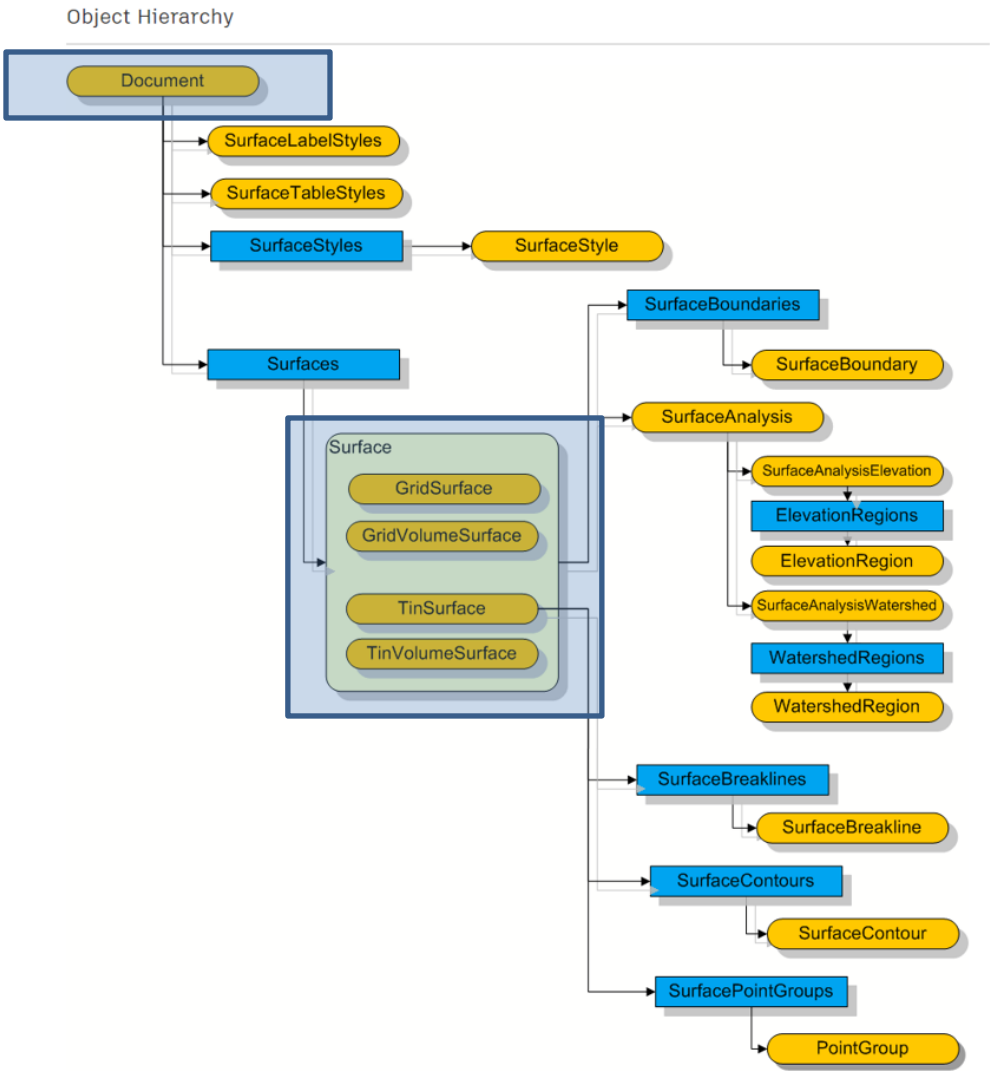


Figure 21: Civil 3D Surface Objects

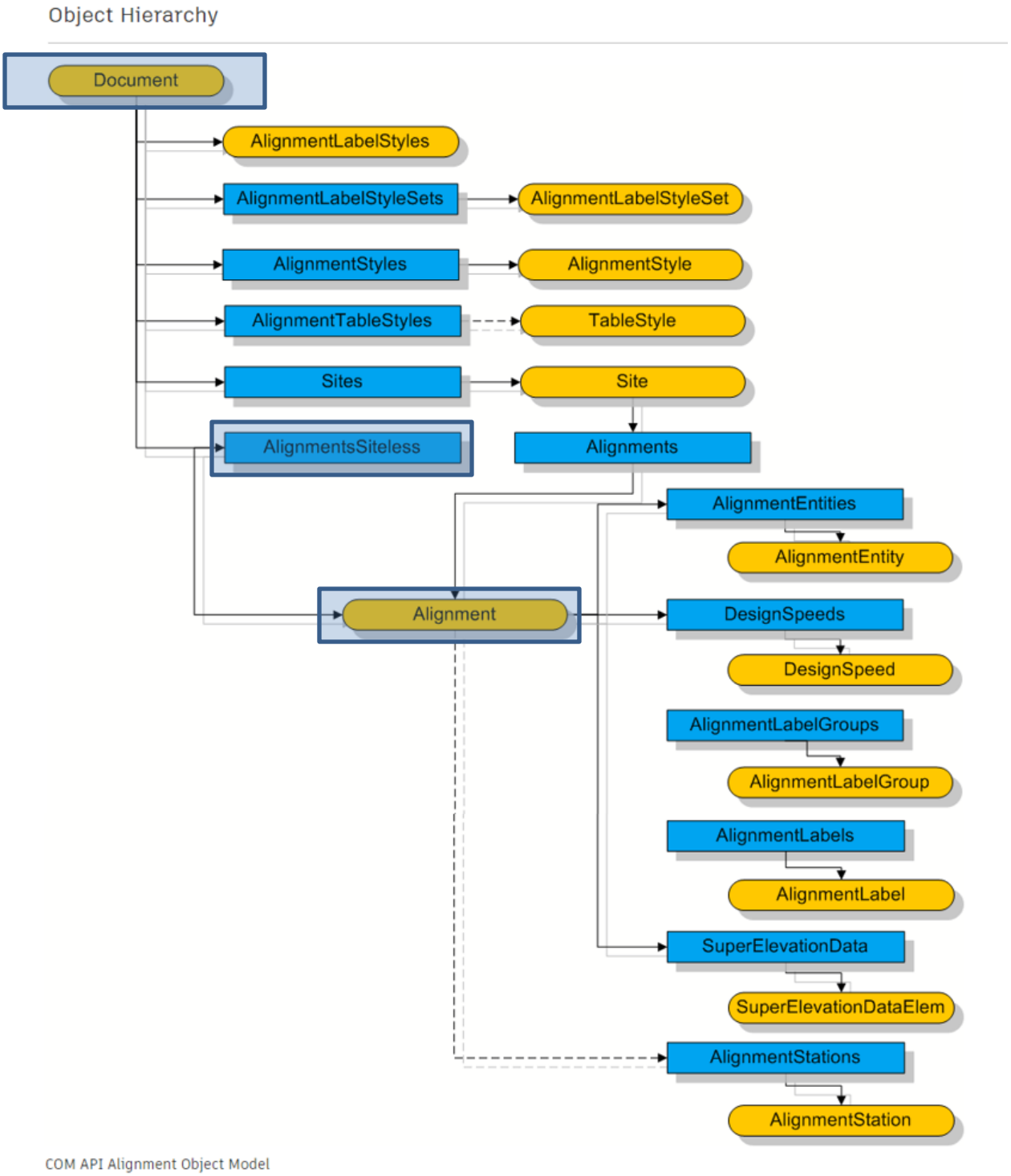


Figure 22: Civil 3D Alignment Objects

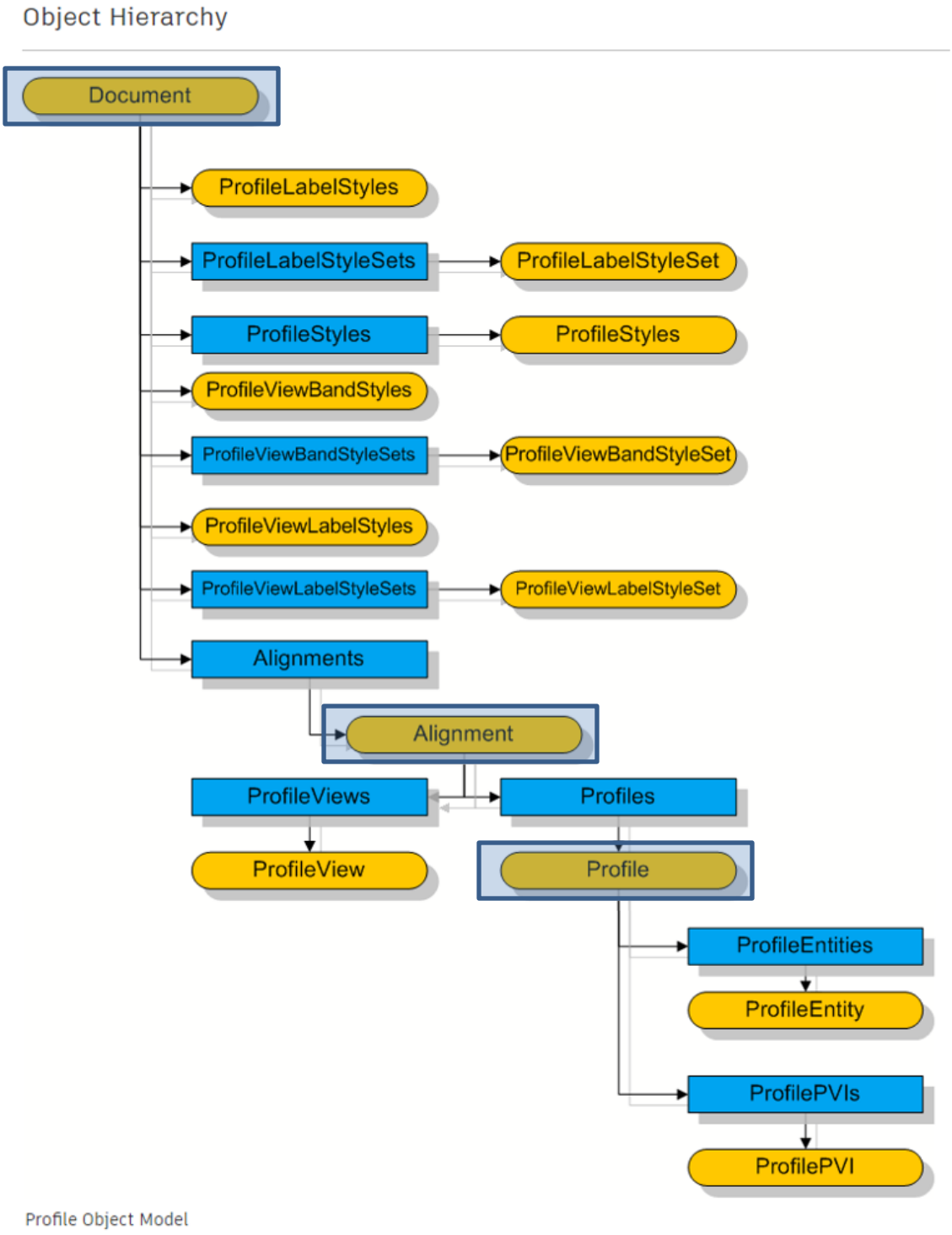
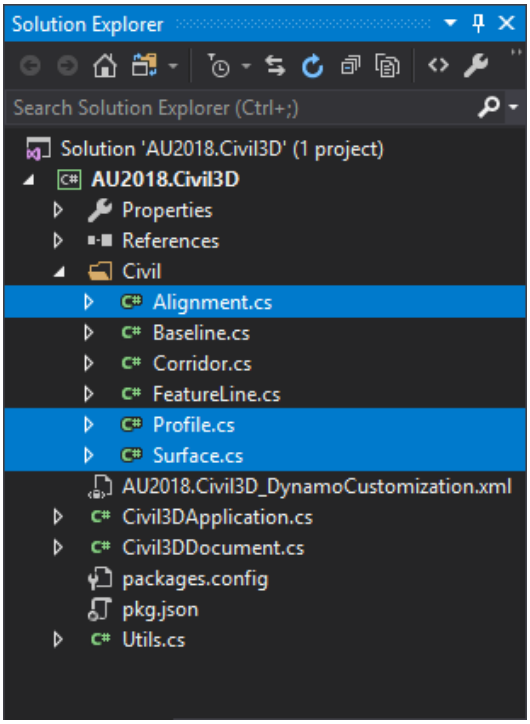


Figure 23: Civil 3D Profile Objects

Visual Studio 2017

New Classes

- Surface.cs
- Alignment.cs
- Profile.cs



Surface.cs

This class represents a single Civil 3D Surface.

```
/// Constructor for a Civil 3D surface
internal Surface(AeccSurface surface)
{
    this._surface = surface;
    this.Name = surface.DisplayName;
}

/// Static Method to extract surface data for analysis
[MultiReturn(new[] { "NumPoints", "NumTris", "Area2D", "Area3D", "MaxElev", "MinElev",
"MaxGrade", "MinGrade" })] // DesignScript.Runtime
public static Dictionary<string, object> GetTerrainStatistics(Surface Civil3DSurface)
{
    AeccSurface surf = Civil3DSurface._surface;
    AeccTinSurface tinSurf = null;
    AeccGridSurface gridSurf = null;
    AeccTinVolumeSurface tinVolSurf = null;
    AeccGridVolumeSurface gridVolSurf = null;

    switch (surf.Type)
    {
        case AeccSurfaceType.aecckGridSurface:
```

```
            //surf1 = surf as AeccGridSurface;
            break;
        case AeccSurfaceType.aecckTinSurface:
            tinSurf = surf as AeccTinSurface;
            break;
        case AeccSurfaceType.aecckGridVolumeSurface:
            //surf1 = surf as AeccGridVolumeSurface;
            break;
        case AeccSurfaceType.aecckTinVolumeSurface:
            //surf1 = surf as AeccTinVolumeSurface;
            break;
        default:
            break;
    }

    AeccTinSurfaceStatistics tinStats = tinSurf.Statistics;
    int numPoints = tinStats.NumberOfPoints;
    int numTri = tinStats.NumberOfTriangles;
    double area2D = tinStats.Area2d;
    double area3D = tinStats.Area3d;
    double maxElev = tinStats.MaxElevation;
    double minElev = tinStats.MinElevation;
    double maxGrade = tinStats.MaxGrade;
    double minGrade = tinStats.MinGrade;

    return new Dictionary<string, object>
        {
            { "NumPoints", numPoints },
            { "NumTris", numTri },
            { "Area2D", area2D },
            { "Area3D", area3D },
            { "MaxElev", maxElev },
            { "MinElev", minElev },
            { "MaxGrade", maxGrade  },
            { "MinGrade", minGrade  }
        };
}
```

.

The method above receives a single Civil3DDocument as its input, and returns a Dictionary containing data for each Civil 3D Surface within the passed-in Document (Number of points, Number of Tris etc.). The output node in Dynamo contains multiple ports, from which we can extract just the specific data we wish to interrogate

The ‘MultiReturn’ attribute (located before the method signature) is a Dynamo-specific attribute that allows the output of multiple lists, in the form of a dictionary.

For more information on the MultiReturn attribute, refer to the documentation at the Dynamo Zero-Touch Development Github page
https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development

```
/// Static method to drape a list of Dynamo points onto a surface
public static IList<Point> DrapePoints(Surface Civil3DSurface, IList<Point> Points)
{
    IList<Point> ptsList = new List<Point>();

    foreach (Point pt in Points)
    {
        double z = Civil3DSurface._surface.FindElevationAtXY(pt.X, pt.Y);
        ptsList.Add(Point.ByCoordinates(pt.X, pt.Y, z));
    }

    var ptsListPruned = Point.PruneDuplicates(ptsList, 0.001).ToList();
    return ptsListPruned;
}



/// Override the string output for the Civil 3D Surface object
public override string ToString()
{
    return string.Format($"Surface (\"{this.Name}\")");
}
```

Alignment.cs

This class represents a single Civil 3D Alignment

```
/// Constructor for a Civil 3D Alignment
internal Alignment(AeccAlignment alignment)
{
    this._alignment = alignment;
    this._entities = alignment.Entities;
    this.Name = alignment.DisplayName;
    this.Length = alignment.Length;
    this.StartStation = alignment.StartingStation;
    this.EndStation= alignment.EndingStation;
}



/// Extract Civil 3D Alignment Station, Offset and Elevation data
[MultiReturn (new[] { "Station", "Easting", "Northing"})]
public static Dictionary<string, object> GetAlignmentStations(Alignment Alignment, double
Interval, bool Geometry)
{
    AeccAlignmentStations stations =
Alignment._alignment.GetStations(AeccStationType.aeccMajor, Interval, 1.0);

    IList<AeccAlignmentStation> stationsList = new List<AeccAlignmentStation>();
    IList<double> s = new List<double>();
    IList<double> e = new List<double>();
    IList<double> n = new List<double>();

    foreach (AeccAlignmentStation station in stations)
    {
        stationsList.Add(station);
```

```
    }

    // if geometry points are included
    if (Geometry == true)
    {
        AeccAlignmentStations stationsGeom =
Alignment._alignment.GetStations(AeccStationType.aeccGeometryPoint, Interval, 1.0);
        foreach (AeccAlignmentStation station in stationsGeom)
        {
            stationsList.Add(station);
        }
    }

    // Sort and remove duplicates (if geometry selected)
    var uniqueList = stationsList.GroupBy(i => i.Station).Select(j =>
j.FirstOrDefault());
    var sortedList = uniqueList.OrderBy(i => i.Station);

    foreach (AeccAlignmentStation stat in sortedList)
    {
        s.Add(stat.Station);
        e.Add(stat.Easting);
        n.Add(stat.Northing);
    }

    return new Dictionary<string, object>
    {
        {"Station", s},
        {"Easting", e},
        {"Northing", n}
    };

}
```

The method above receives an Alignment as its input, as well as a chainage interval and a Boolean value determining whether to include geometry points (TS, SC, CS etc), and returns a Dictionary containing data for each Station, Offset and Elevation. Like the previously mentioned Surface method ‘GetTerrainStatistics’, the output node in Dynamo contains multiple ports, from which we can extract just the specific data we wish to interrogate for alignment reports.

```
/// Get the Civil 3D Profiles associated with an Alignment
public IList<Profile> GetProfiles()
{
    IList<Profile> profiles = new List<Profile>();

    foreach (AeccProfile profile in this._alignment.Profiles)
    {
        profiles.Add(new Profile(profile));
    }

    return profiles;
}
```

This method returns all Profiles associated to the parent Alignment. We will use this to provide elevation values to the output reports from Dynamo

```
/// Override the string output for the Civil 3D Alignment object
public override string ToString()
{
    return string.Format($"Alignment (Name = {this.Name}, Length =
{this.Length.ToString("#.###")} )");
}
```

Profile.cs

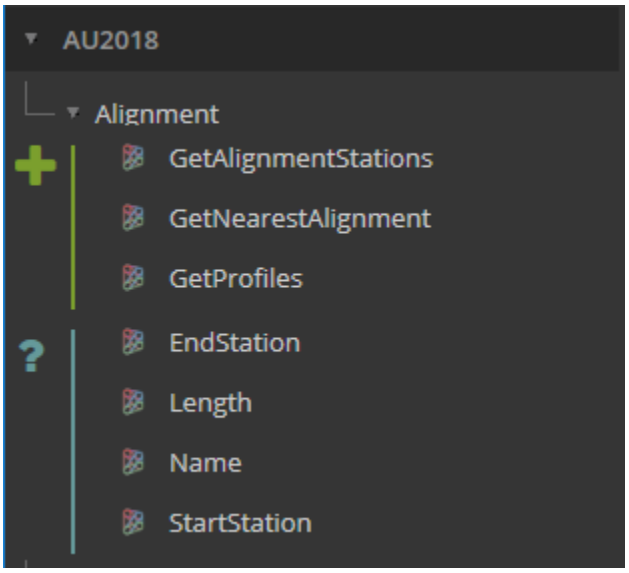
This class represents a Civil 3D Profile

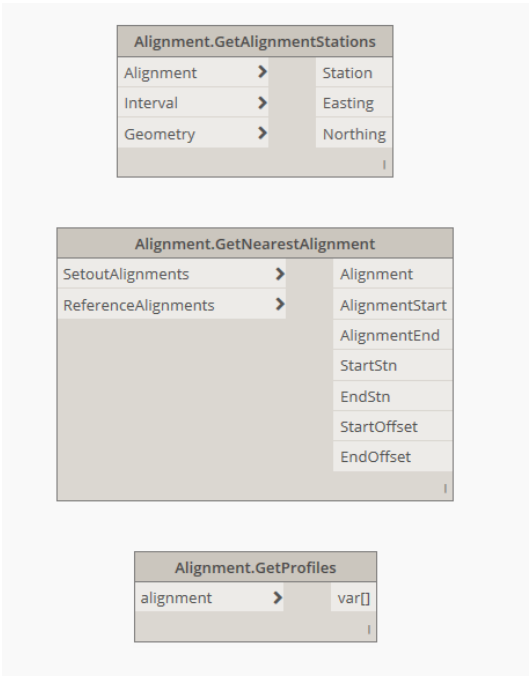
```
/// Constructor for a Civil 3D Profile
internal Profile(AeccProfile profile)
{
    this._profile = profile;
    this.Name = profile.DisplayName;
}

public IList<double> GetProfileElevationByStation(Alignment Alignment, IList<double>
Stations)
{
    IList<double> elevations = new List<double>();

    foreach (double station in Stations)
    {
        try
        {
            elevations.Add(this._profile.ElevationAt(station));
        }
        catch (Exception)
        {
            elevations.Add(-999.000);
        }

    }

    return elevations;
}
```

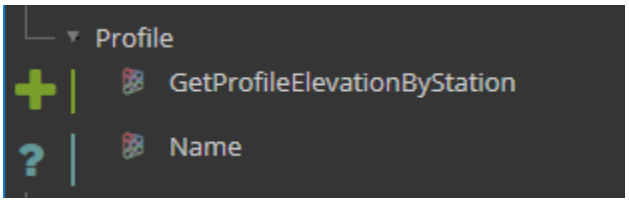
This method takes an Alignment object and a list of Stations (as doubles). The method cycles through each passed station on the alignment and extracts the elevation at that station. The output value is a list of doubles that return the elevation of all passed stations..

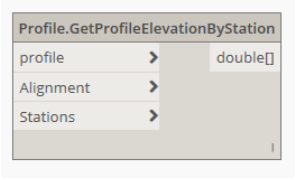
```
/// <summary>
/// Override the string output for the Civil 3D Profile object
/// </summary>
/// <returns></returns>
public override string ToString()
{
    return string.Format($"Profile (Name = {this.Name} )");
}
```

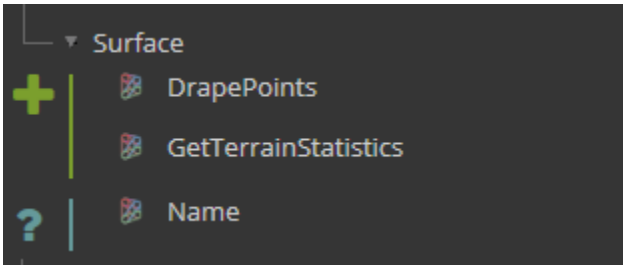
Dynamo Studio

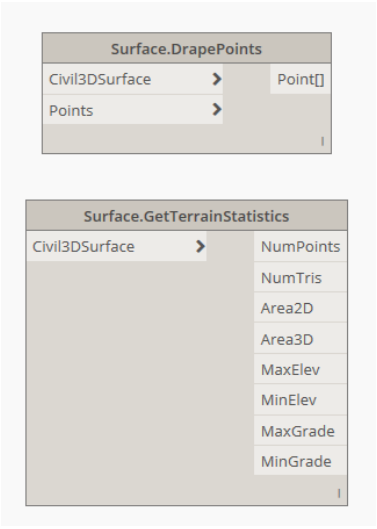




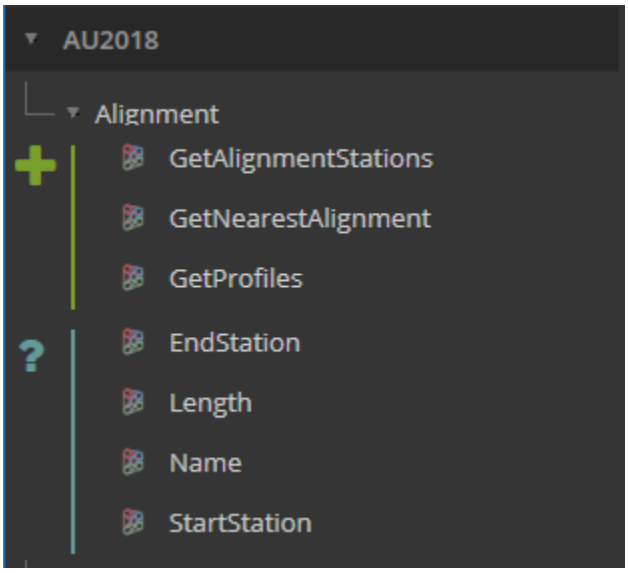






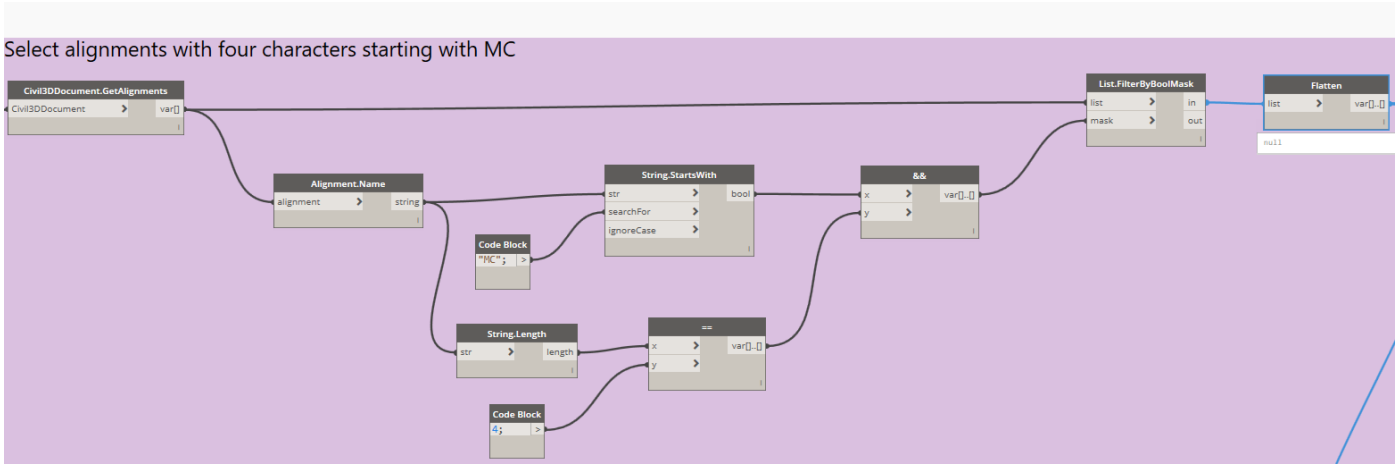


Alignment Nodes



Similar to the process described in the previous section (selecting documents by filter), we sort through alignment names, searching for strings matching a specific pattern (“MC” in the example below).

Again, the List.FilterByBoolMask assists in creating the alignment filter



The snippet below will simply take in an alignment (or list of alignments), and either:

1. Alignment.GetAlignmentStations
    - Get the Alignment station information (Station, Easting and Northing) The output from this node is split into three separate ports to simplify downstream operations.
2. Alignment.GetProfiles
    - Get the Profiles associated with the passed-in Alignment The output of this node is a list of profiles.

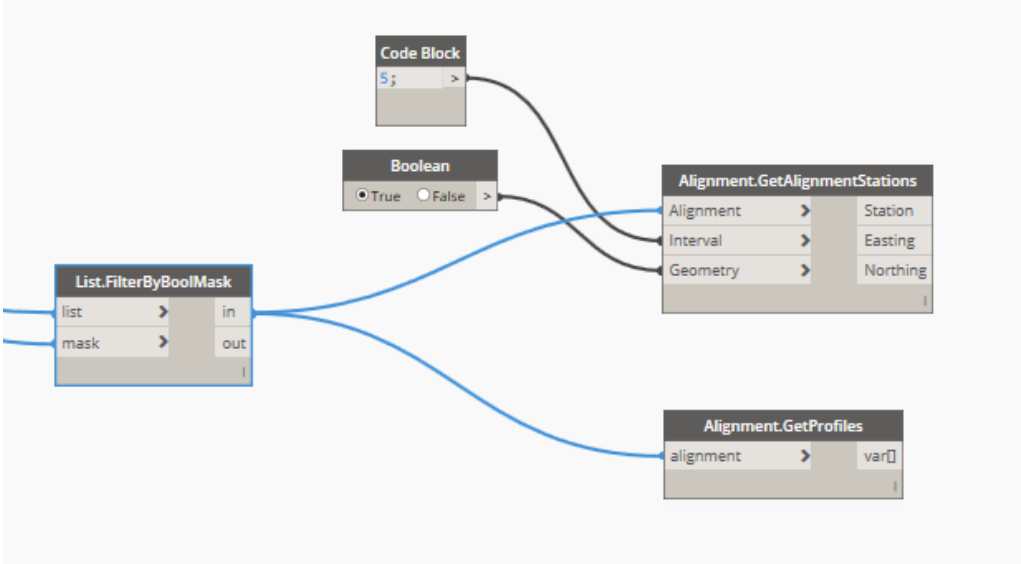


Figure 24: Alignment Create Nodes

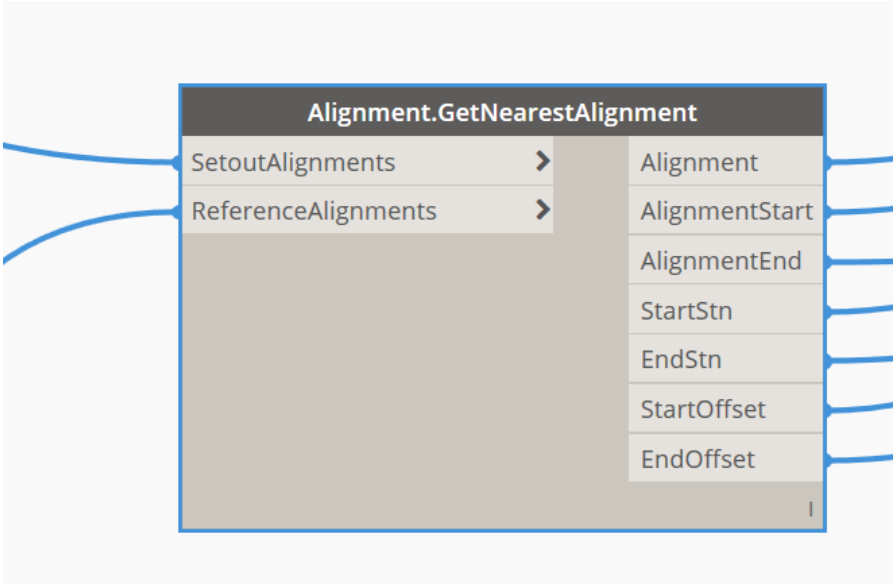
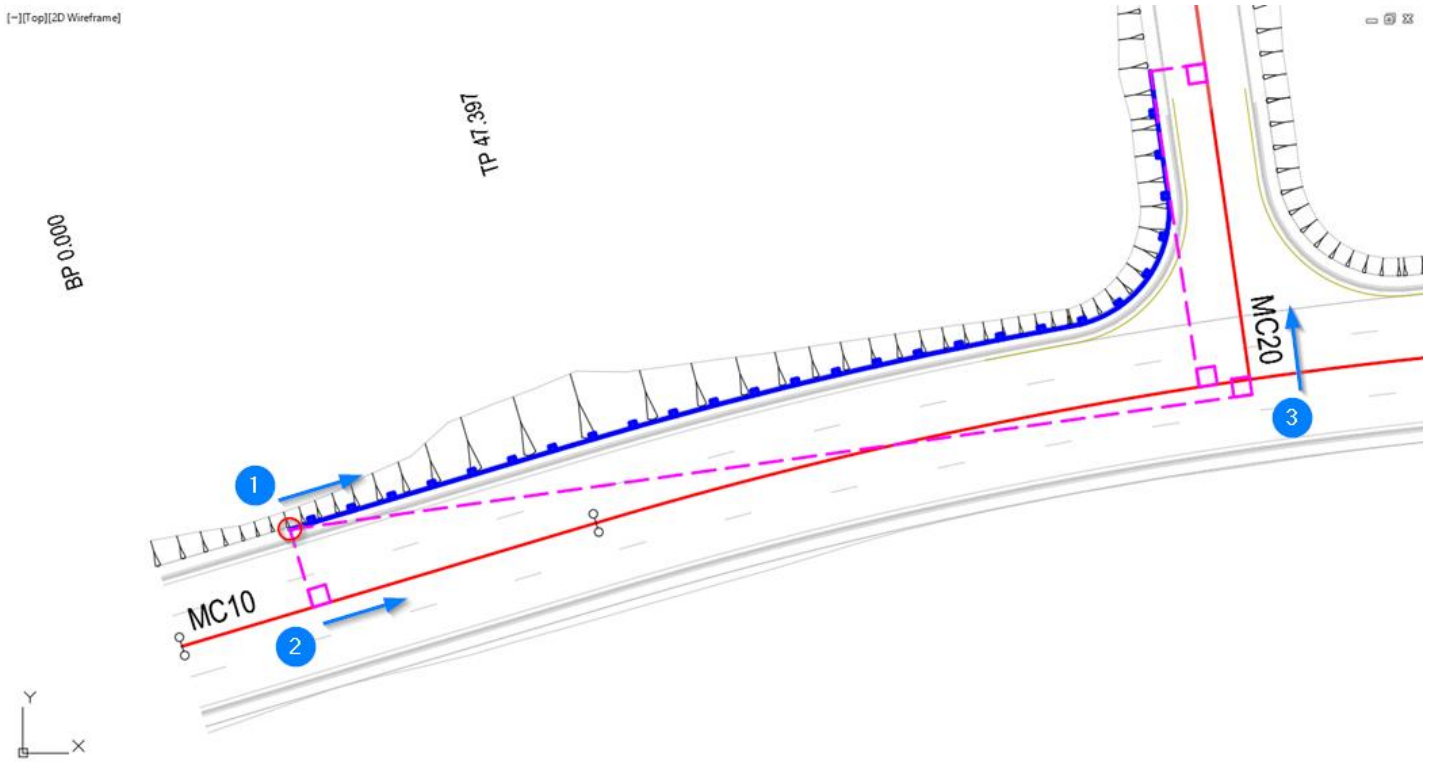


Figure 25: Alignment.GetNearestAlignment node

The Alignment.GetNearestAlignment is a more complex node that takes two alignments (or lists of alignments) and will cycle through each setout alignment and retrieve the nearest reference alignment (and its chainage / offset). The node determines which reference alignment is suitable by comparing the bearing of the setout alignment point and comparing it against the similar bearing on all reference alignments.

For example, using the image below as a reference

- From Point 1, extract the start point coordinates of the **BLUE** barrier string
  - Additionally, calculate the bearing at the start point (Point 1)
- Cycle through the **RED** alignment strings, calculate the normal from the Barrier string point (Point 1) onto each alignment string to calculate Points 2 and 3. (Normals are shown in **MAGENTA**)
  - Extract the Chainage and Offset from Point 1 at each of the alignment strings.
  - Measure the distance from Point 1 to each calculated normal point on each alignment string.
  - Calculate the bearing at each alignment normal point (Points 2 and 3)
- To determine which alignment is the 'closest' for the data table extraction, we need to consider two criteria:
  - If the bearing at the Start Point (Point 1) is within a certain tolerance of an alignment normal point's bearing.
    In the example below, the bearing at Point 1 and the bearing at Point 2 (along MC10) are almost identical. Oppositely, the bearing at Point 1 and the bearing at Point 3 (along MC20) are well outside tolerance. In this case, MC10 is the closest reference alignment, and MC20 is discarded.
  - The offset distance from the start point (Point 1) to each of the alignment points (Points 2 and 3).
    In the example below, the distance between Points 1 and 2 (MC10) is less than the distance from Points 1 and 3 (MC20)
  - By combining these two criteria we can safely extract the correct reference string for the start and end points of all setout objects in our collection



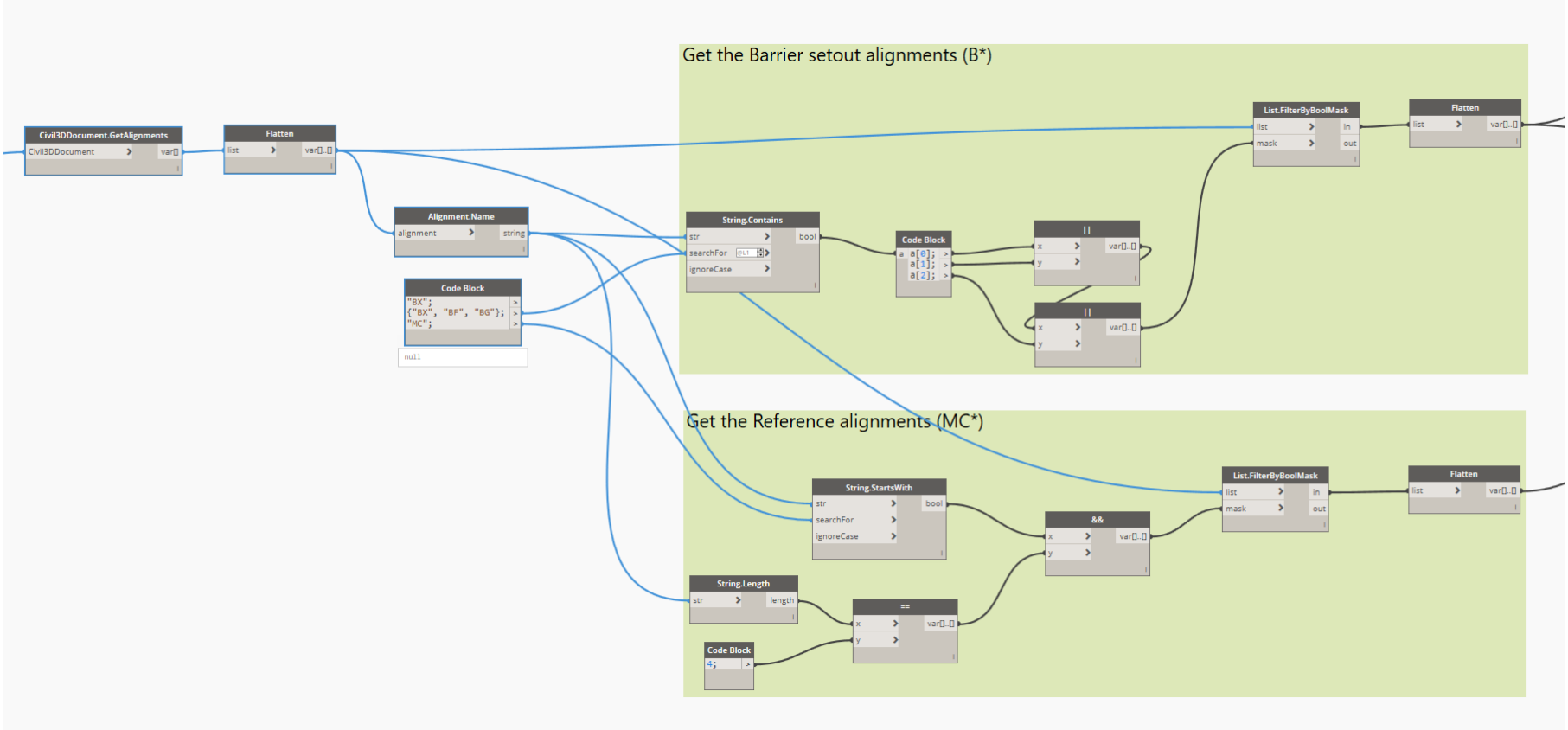


Figure 26: Sample AU 2018 - 02.3 - Barrier Schedule.dyn (Part 1)

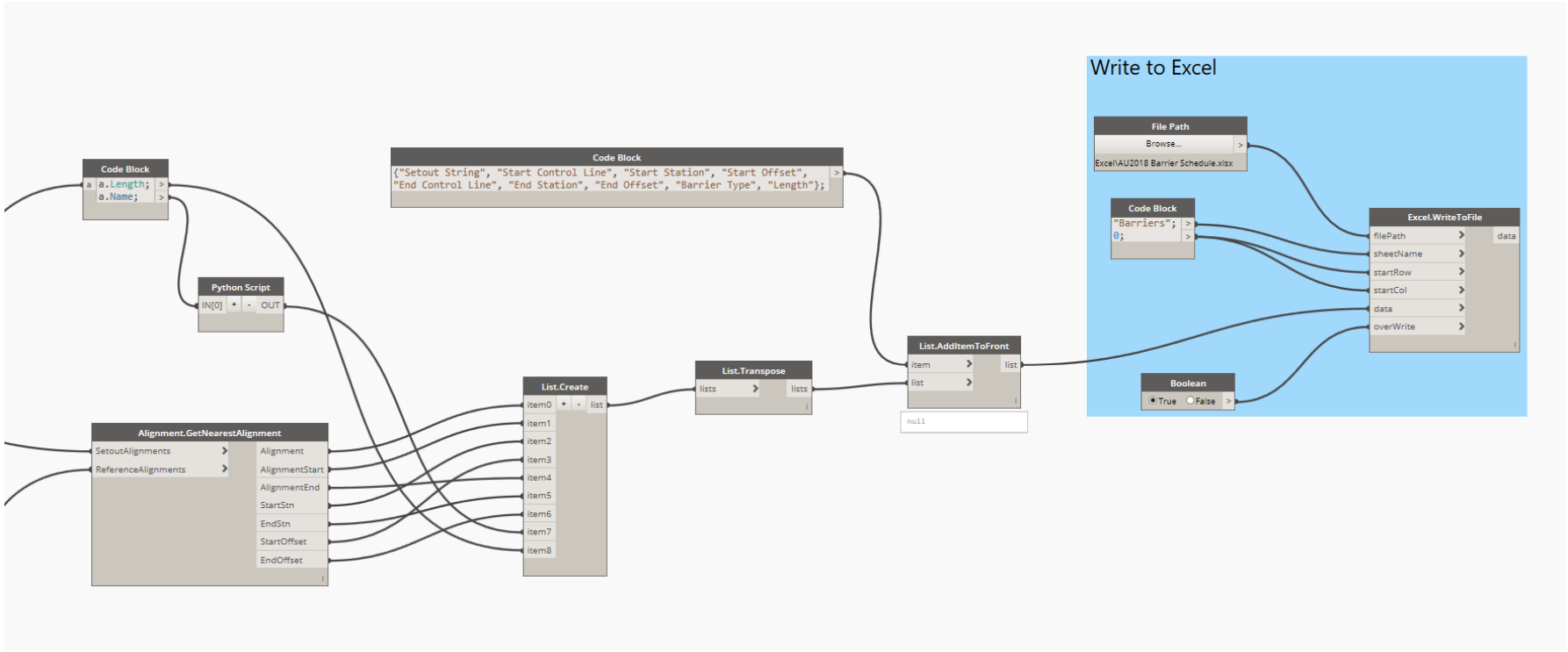


Figure 27: Sample AU 2018 - 02.3 - Barrier Schedule.dyn (Part 2)

The remaining nodes in the Alignment category are Query nodes, which simply return information about the Alignment object., including

- Name
- Start Station
- End Station
- Length

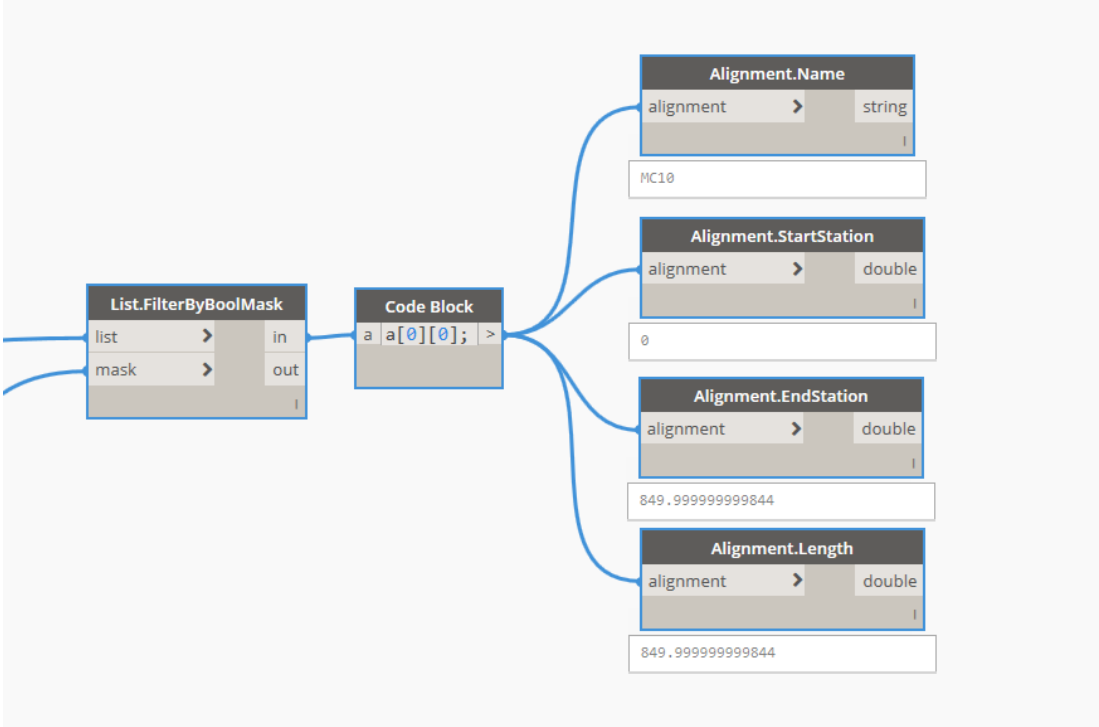
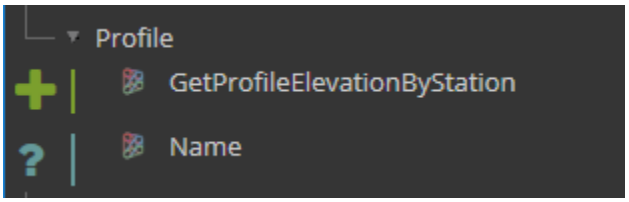


Figure 28: Alignment Query Nodes

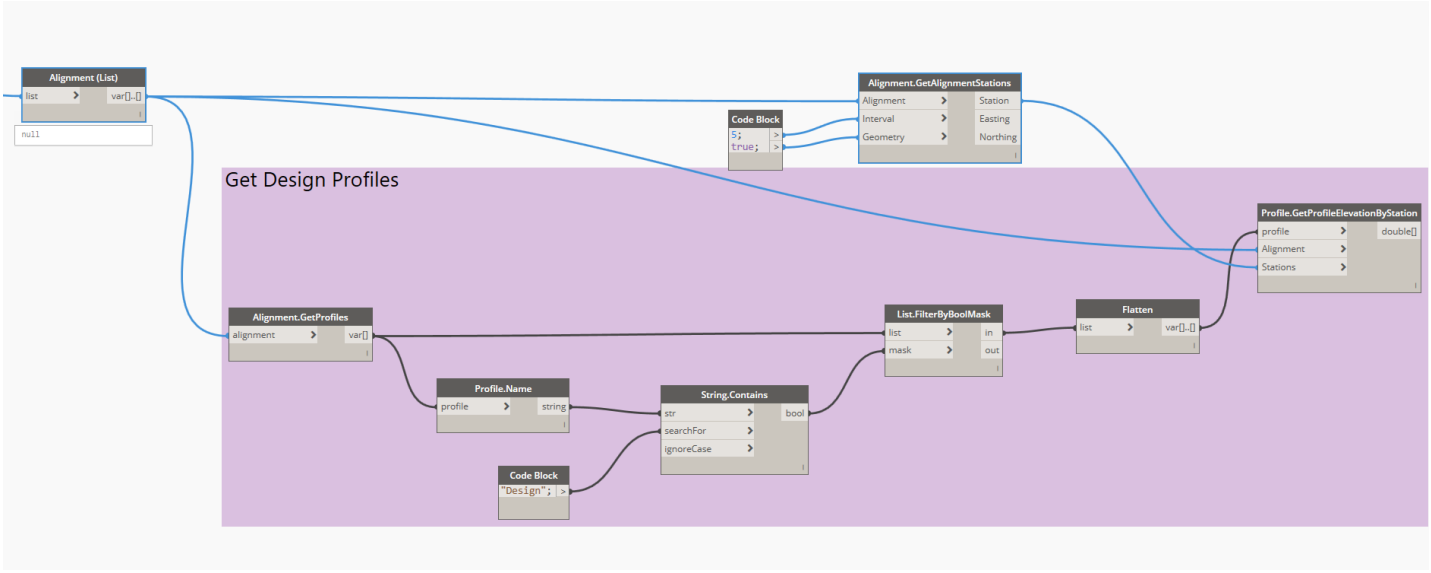
Profile Nodes



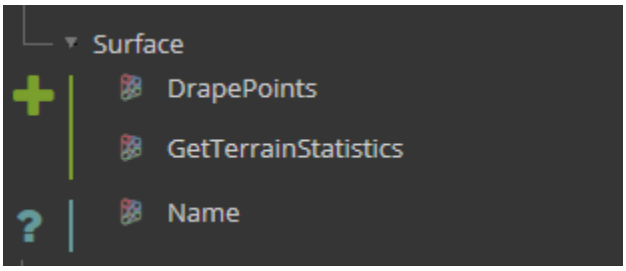
Profile nodes, in this example, have been setup to complement the alignment nodes and to provide additional elevation information as part the reporting process.

In the example below, the alignment (or list of alignments) are extracted and passed into the Alignment.GetProfiles, Alignment.GetAlignmentStations and Profile.GetProfileStationByElevation nodes.

The aim of this group is to, once again, provide a robust and repeatable workflow to extract only the profiles that contain the word ‘Design”. The List.FilterByBooleanMask ensures only the correct profile is used for the passed in alignment(s).



Surface Nodes



The surface nodes in this example, specifically ‘GetTerrainStatistics’, provide some basic functionality to extract surface information, such as:

- Number of Points
- Number of Triangles
- Area (2D and 3D)
- Max / Min Elevation
- Max / Min Grade

As this node was written with multiple output paths, you can pick and choose which surface information to export. Additionally, this node will accept either a single surface, or a list of surfaces. This makes analysing in external packages (such as Power BI) much simpler.

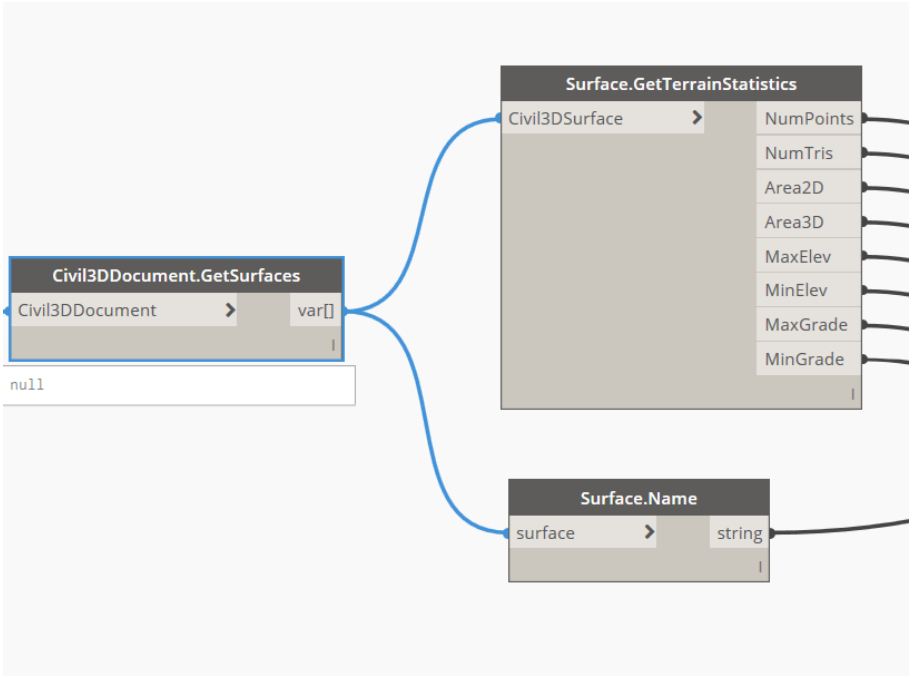


Figure 29: Surface.GetTerrainStatistics Nodes

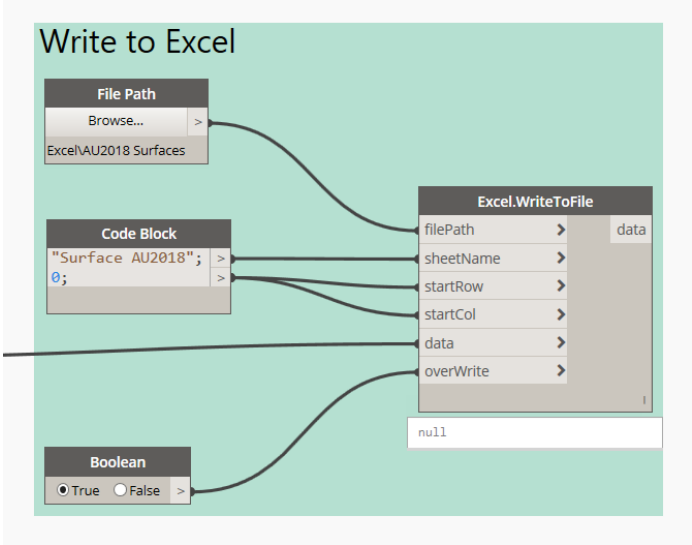


Figure 30: Write Surface data to Excel

Once surface information has been extracted, the outputs can be written to Excel using the Excel nodes built into Dynamo and Dynamo Studio.

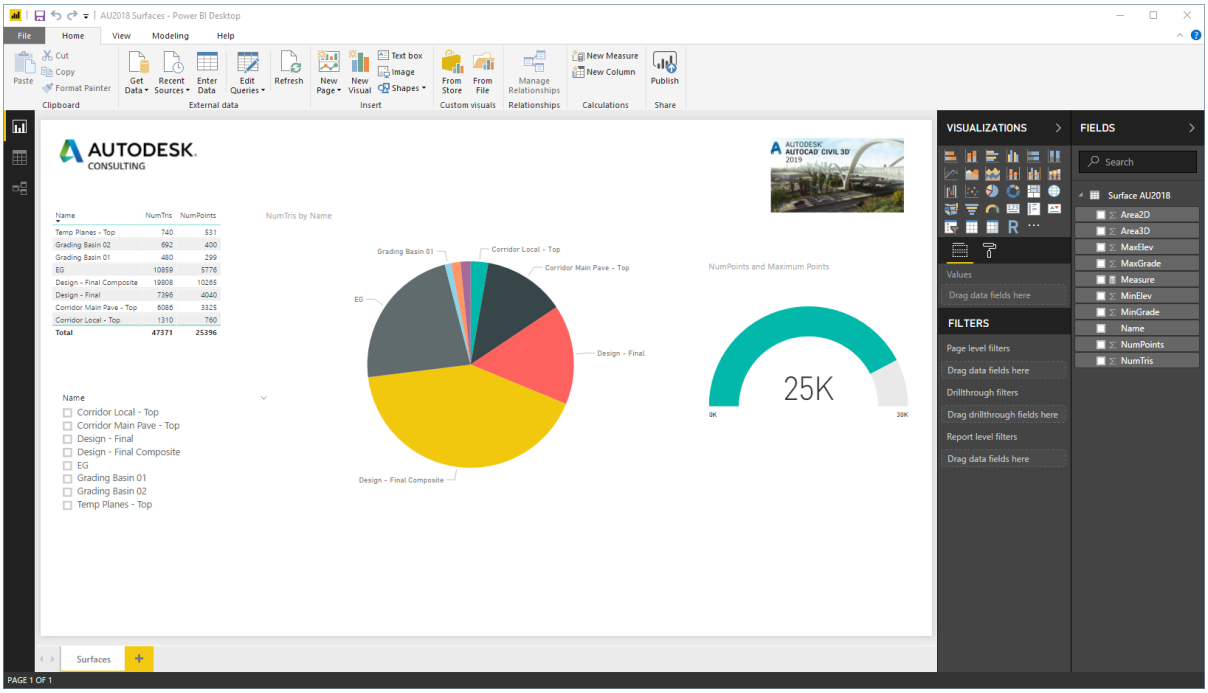


Figure 31: Surface statistics depicted in Power BI Desktop

Finally, the ‘Surface.DrapePoints’ node will simply take a list of 2D points and a surface, returning a list of 3D points where the input 2D point intersects the provided surface.

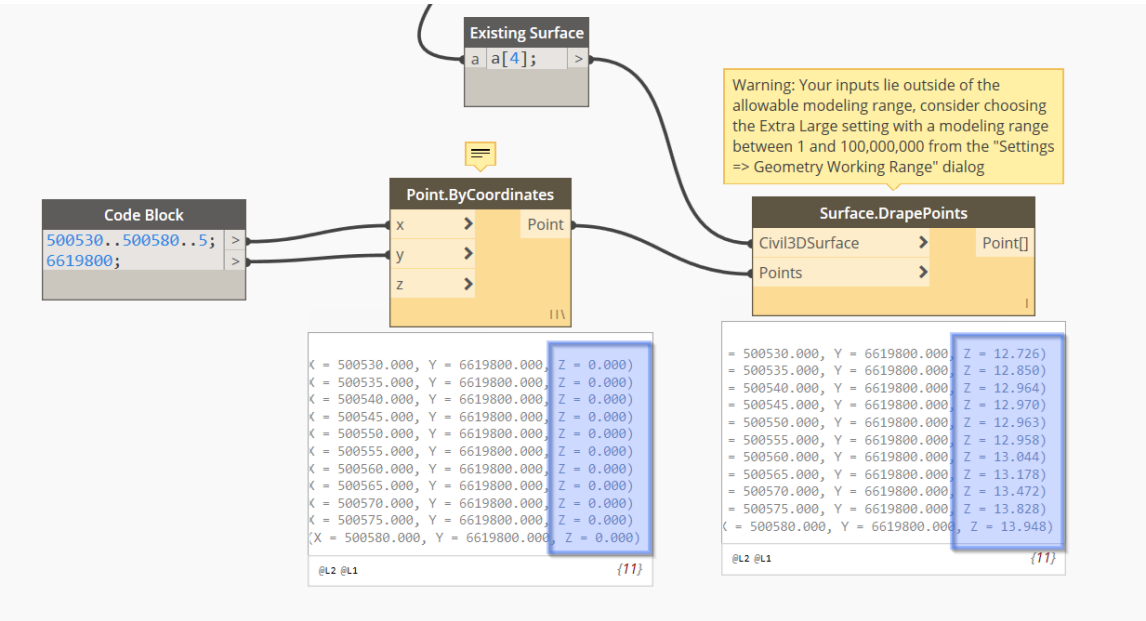


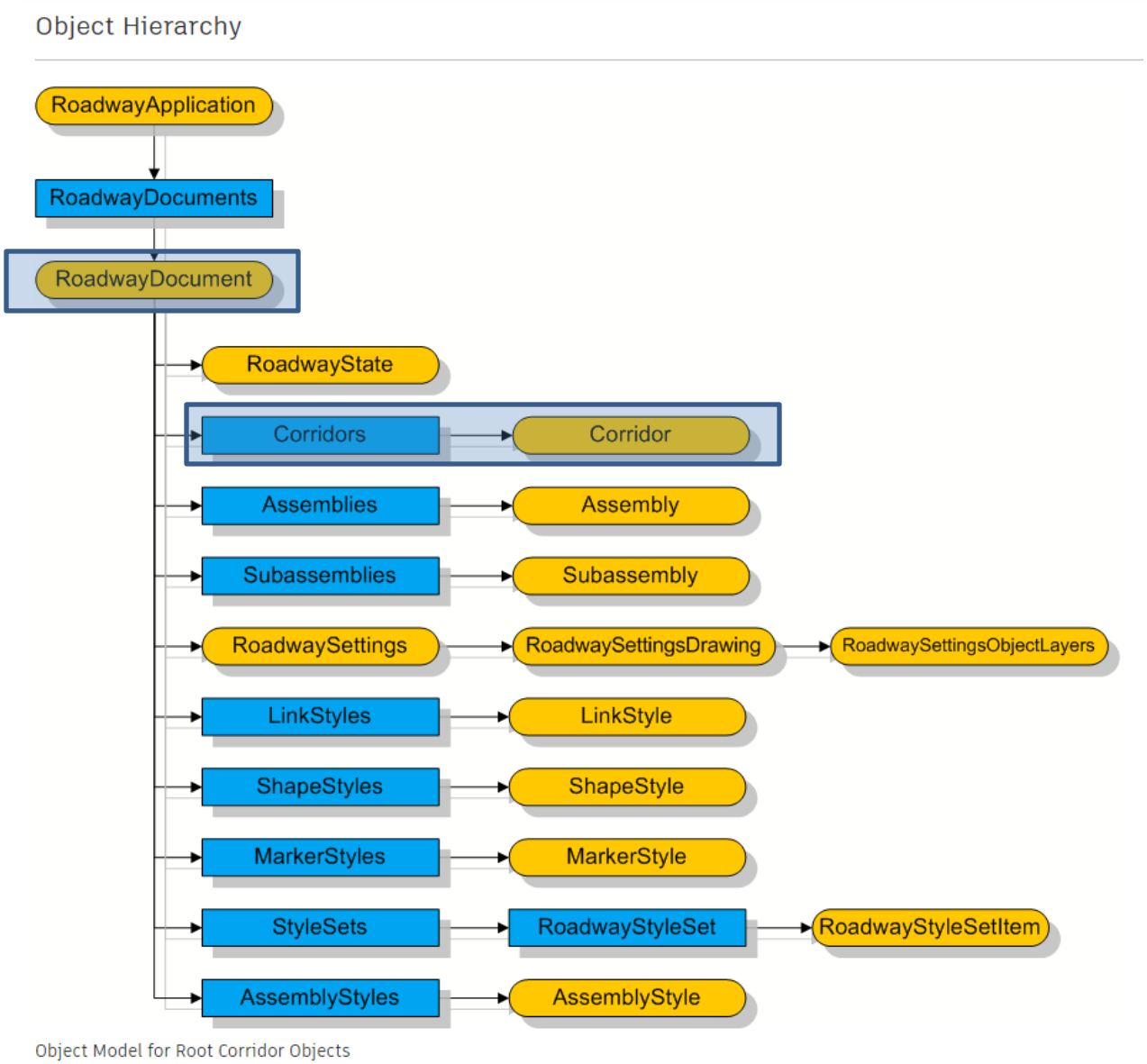
Figure 32: Surface.DrapePoints node

Learning Objective 3 - Leverage the flexibility of Dynamo to extract Civil 3D model data

In this exercise, we will create an additional three classes, representing Corridors, Baselines and Corridor FeatureLines.
The three classes all follow a similar structure to the classes created in Objective 1, and contain Civil 3D Aecc* object properties, a constructor, unique methods and a ToString() override

Civil 3D Object Hierarchy

In this section we will focus on three components of the COM API, Corridors, Baselines and FeatureLines.



Object Model for Root Corridor Objects

Figure 33: Civil 3D Corridor Objects

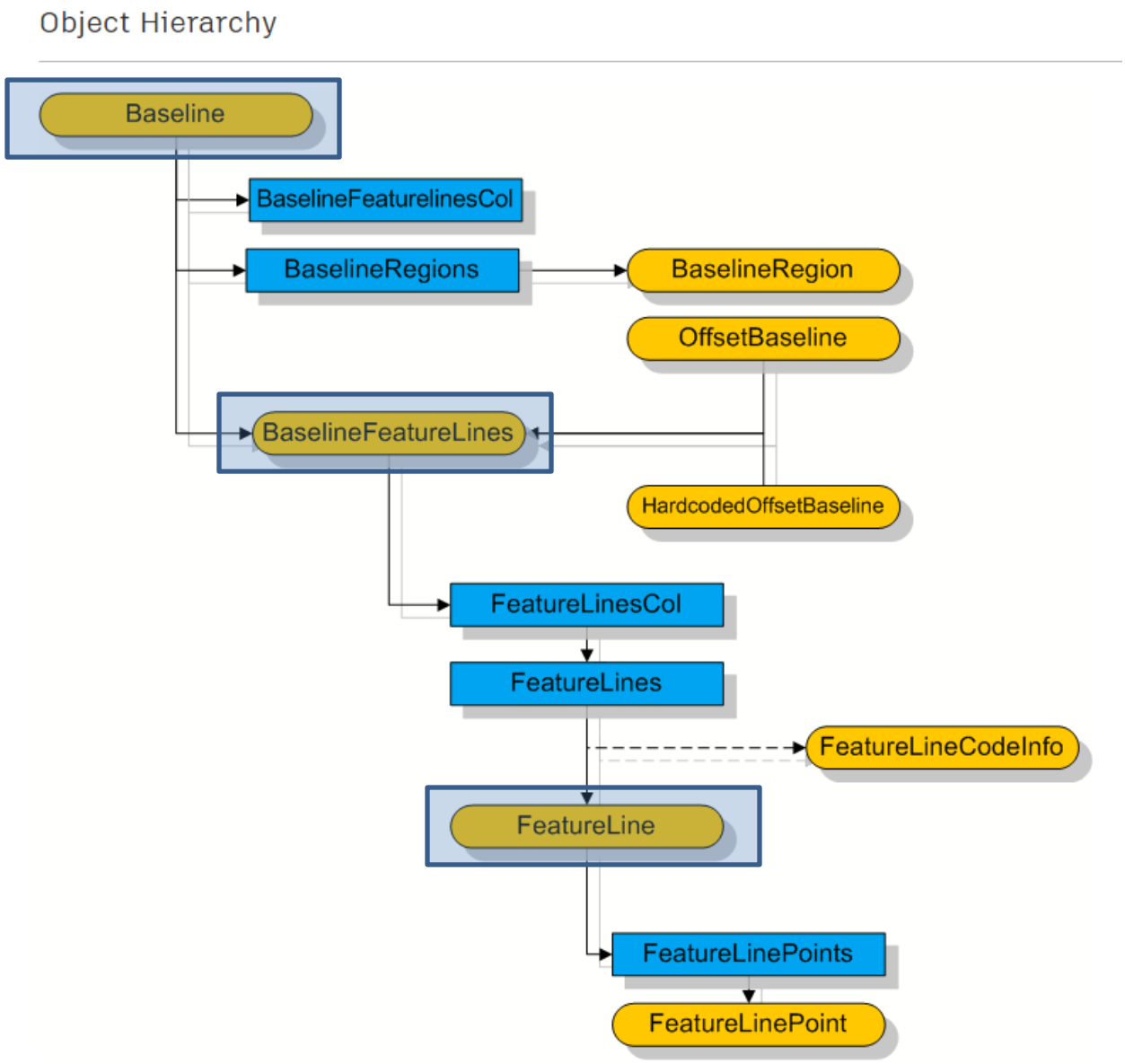
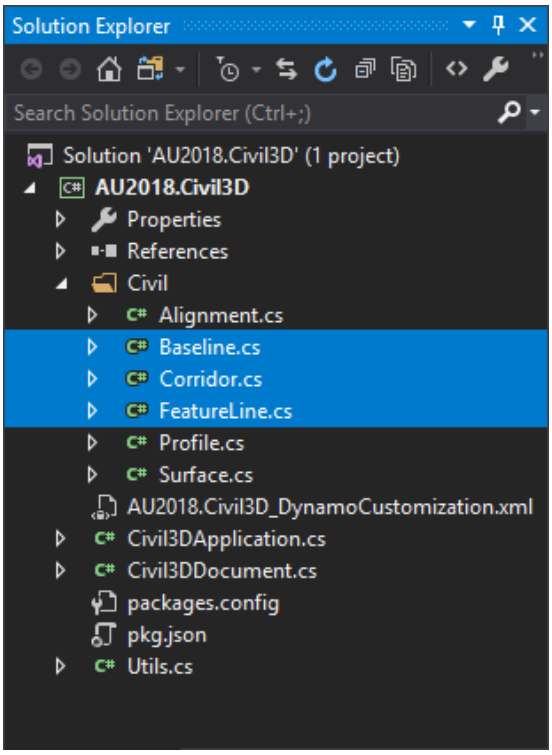


Figure 34: Civil 3D Baseline and FeatureLine Objects

Visual Studio 2017

New Classes

- Corridor.cs
- Baseline.cs
- FeatureLine.cs



Corridor.cs

This class represents a single Civil 3D Corridor object.

```
/// Constructor for a Civil 3D Corridor
internal Corridor(AeccCorridor corridor)
{
    this._corridor = corridor;
    this.Name = corridor.DisplayName;

    // Add baselines to Corridor model
    IList<Baseline> baselines = new List<Baseline>();
    foreach (AeccBaseline baseline in corridor.Baselines)
    {
        baselines.Add(new Baseline(baseline));
    }

    this.Baselines = baselines;
}
```

The constructor above ensures that all Baseline objects associated to the corridor are added to the Corridor object.

```
/// Get the Feature Lines from the Civil 3D Corridor
public IList<FeatureLine> GetFeatureLines(string Code)
{
    IList<FeatureLine> fLinesColl = new List<FeatureLine>();

    foreach (AeccBaseline baseline in this._corridor.Baselines)
    {
        //AeccFeatureLinesCol flColl = baseline.MainBaselineFeatureLines.FeatureLinesCol;

        foreach (AeccBaselineRegion baselineRegion in baseline.BaselineRegions)
        {
            foreach (AeccFeatureLines flines in
baseline.MainBaselineFeatureLines.FeatureLinesCol)
            {
                foreach (AeccFeatureLine fline in flines)
                {
                    if (fline.CodeName == Code)
                    {
                        IList<Point> points = new List<Point>();
                        IList<double> chainages = new List<double>();

                        foreach (AeccFeatureLinePoint pt in fline.FeatureLinePoints)
                        {
                            Point ptDyn = Point.ByCoordinates(pt.XYZ[0], pt.XYZ[1],
pt.XYZ[2]);

                            if (pt.Station >= baselineRegion.StartStation && pt.Station <=
baselineRegion.EndStation)
                            {
                                points.Add(ptDyn);
                                chainages.Add(pt.Station);
                            }
                        }
                        PolyCurve curve = PolyCurve.ByPoints(points);
                        fLinesColl.Add(new FeatureLine(baseline, baselineRegion, fline, curve,
points, chainages));
                    }
                }
            }
        }

    }
    return fLinesColl;
}
```

This method contains multiple nested loops. The looping follows the following structure:

- Corridor
  - Baselines
    - BaselineRegoins
      - FeatureLineCollection
        - FeatureLines
          - FeatureLinePoints

We need to check that the station of the point falls within the current region, in order to stop the spanning of featurelines across regions (i.e. across intersections).

The final output is a list of FeatureLines,

```
/// Override the string output for the Civil 3D Corridor object
public override string ToString()
{
    return string.Format($"Corridor (Name = {this.Name})");
}
```

Baseline.cs

This class represents a single Civil 3D Baseline.
A Civil 3D Corridor can contain multiple baselines, and even identical multiple baselines.
Consideration should be given to naming baselines for easy identification in Dynamo.

```
/// Constructor for a Civil 3D Baseline
internal Baseline(AeccBaseline baseline)
{
    this._baseline = baseline;
    this.Name = baseline.Alignment.Name;
    this.StartStation = baseline.StartStation;
    this.EndStation = baseline.EndStation;
}


////////////////////////////////////////////////
// Methods
////////////////////////////////////////////////

/// Create a Dynamo Point at a specific Station/Offset/Elevation
public Point PointByStationOffsetElevation(double Station, double Offset, double
Elevation)
{
    Point pt = null;

    double[] soe = new double[] { Station, Offset, Elevation };
    var xyz = this._baseline.StationOffsetElevationToXYZ(soe);
    pt = Point.ByCoordinates(xyz[0], xyz[1], xyz[2]);

    return pt;
}
```

This method returns a Dynamo Point from the corridor baseline, which is located by Station, Offset and Elevation. We can use these Dynamo points to setout other geometry, or extract additional information.

```
/// Add an AutoCAD block at a specific Station/Offset/Elevation
public string AddBlockByStationOffsetElevation(string Name, double Station, double
Offset, double Elevation, double Rotation)
{

    AcadDocument doc = this._baseline.Corridor.Document as AcadDocument;
    double[] soe = { Station, Offset, Elevation };

    Point ptDyn = null;

    if (Station >= this.StartStation && Station <= this.EndStation)
    {
        // Vector at chainage on baseline
        var vec = this._baseline.GetDirectionAtStation(Station);
        Vector vecY = Vector.ByCoordinates(vec[0], vec[1], vec[2]); // Forward vector
        Vector vecX = vecY.Cross(Vector.ZAxis());                   // Perp Vector

        double[] pt = this._baseline.StationOffsetElevationToXYZ(soe);
        ptDyn = Point.ByCoordinates(pt[0], pt[1], pt[2]);

        double rot = Vector.YAxis().AngleAboutAxis(vecY, Vector.ZAxis());
        rot = Utils.DegreesToRadians(rot += Rotation);

        string name = Utils.AddBlock(doc, ptDyn, Name, 1.0, rot);

        ptDyn.Dispose();
    }
    else
    {
        return null;
    }

    return Name;
}
```

This method will place a block reference into the current AutoCAD document, based on its Station, Offset and Elevation from a baseline. The addition of a Rotation input allows the user to control the rotation of the block insertion relative to the Y vector of the baseline at the provided station. The ‘Utils.Addblock’ method is described later in this document.

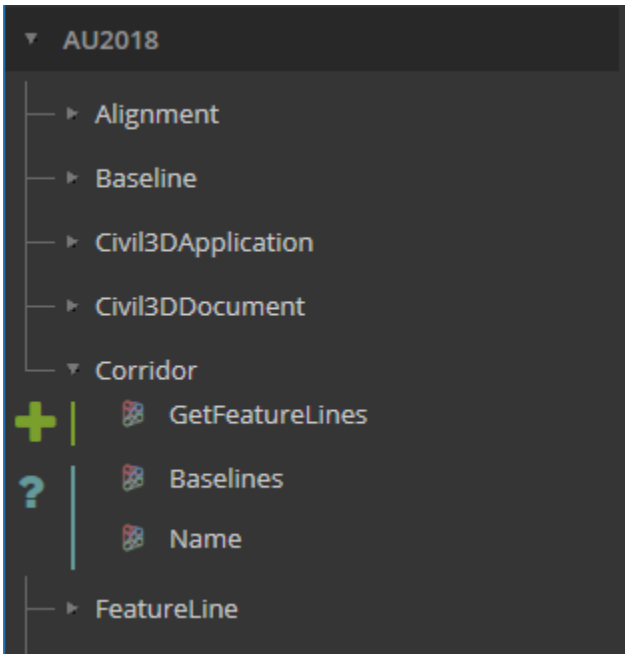
```
/// Override the string output for the Civil 3D Baseline object
public override string ToString()
{
    return string.Format($"Baseline (Name = {this.Name})");
}
```

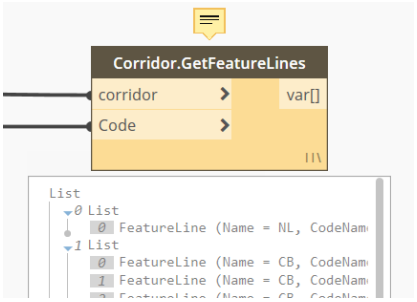
FeatureLine.cs

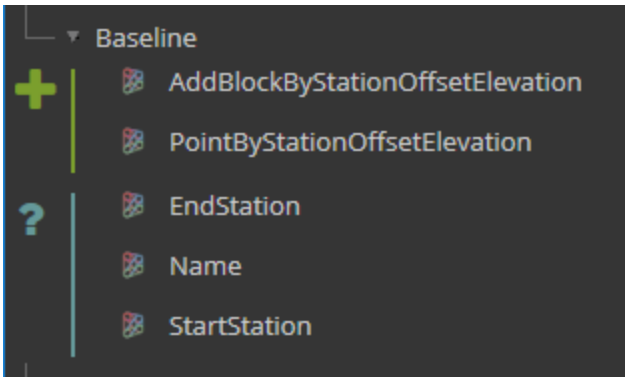
This class represents a single Civil 3D Corridor FeatureLine

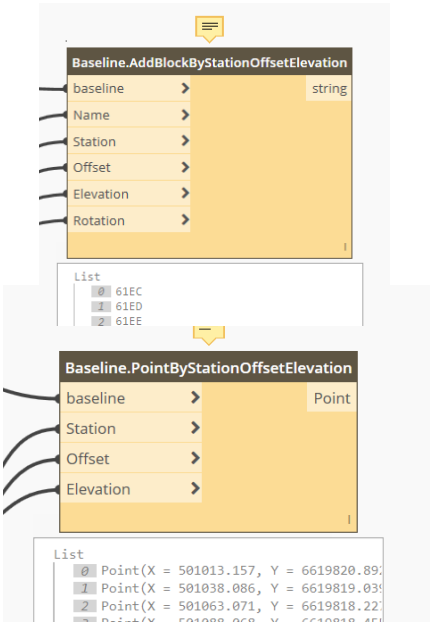
```
/// Constructor for a Civil 3D Corridor Feature Line
internal FeatureLine(AeccBaseline baseline, AeccBaselineRegion region, AeccFeatureLine
featureLine, PolyCurve curve, IList<Point> points, IList<double> chainages)
{
    this._featureLine = featureLine;
    this._baseline = baseline;
    this.Curve = curve;
    this.CodeName = featureLine.CodeName;
    this.StartStation = region.StartStation;
    this.EndStation = region.EndStation;
    this.Points = points;
    this.Chainages = chainages;
}


public CoordinateSystem CoordinateSystemByStationOffsetElevation(double Station, double
Offset, double Elevation)
{
    double[] soe = { Station, Offset, Elevation };

    CoordinateSystem cs;

    if (Station >= this._baseline.StartStation && Station <= this._baseline.EndStation)
    {
        var vec = this._baseline.GetDirectionAtStation(Station);
        Vector vecY = Vector.ByCoordinates(vec[0], vec[1], vec[2]);
        Vector vecX = vecY.Cross(Vector.ZAxis());

        double[] pt = this._baseline.StationOffsetElevationToXYZ(soe);
        Point ptDyn = Point.ByCoordinates(pt[0], pt[1], pt[2]);

        cs = CoordinateSystem.ByOriginVectors(ptDyn, vecX, vecY, Vector.ZAxis());
        ptDyn.Dispose();
    }
    else
    {
        return null;
    }

    return cs;
}



/// Override the string output for the Civil 3D FeatureLine object
public override string ToString()
{
    return string.Format($"FeatureLine (Name = {this.CodeName}, CodeName =
{this.CodeName}, StartStation = {this.StartStation}, EndStation = {this.EndStation}");
}
```

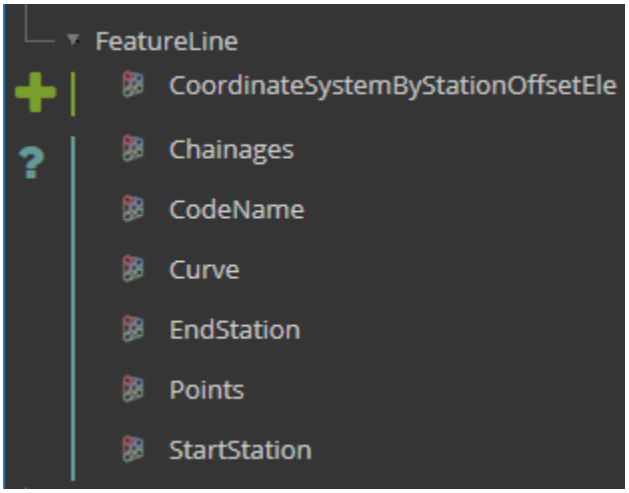
Dynamo Studio

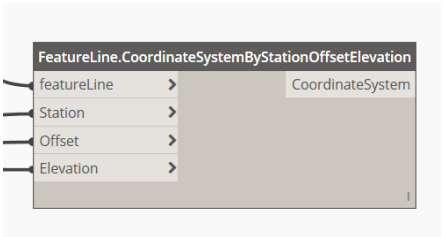












It should be noted that when running certain Civil 3D nodes, a warning message will appear stating that in ‘Warning: Your inputs lie outside the allowable modelling range…”
This is only a warning and not an error and will appear if your model is located too far from the origin (0,0).
You can setup Dynamo to remove this message (Settings => Geometry Working Range) by changing the settings to ‘Large’ or ‘Extra Large’.
It is advised NOT to do this, as setting this to a setting higher than ‘Medium’ has a negative effect on the Dynamo visual display and tends to show linear elements with gaps where none actually exist.
It is recommended to leave the Geometry Working Range set to ‘Medium’

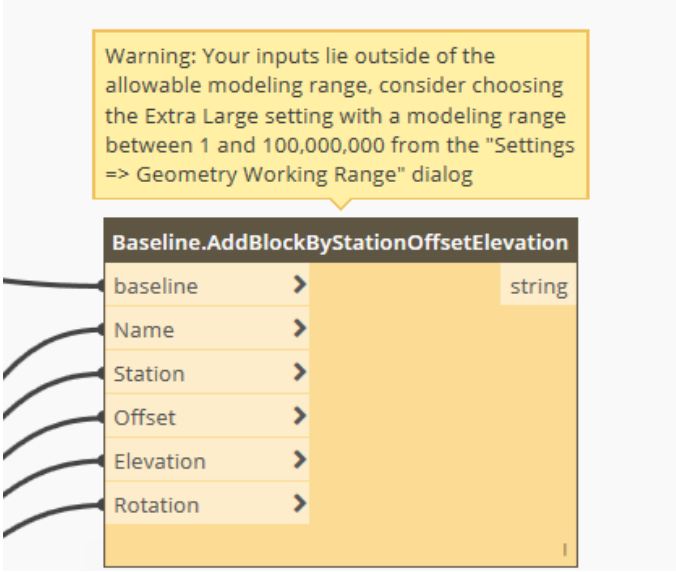


Figure 35: 'Geometry Working Range' warning

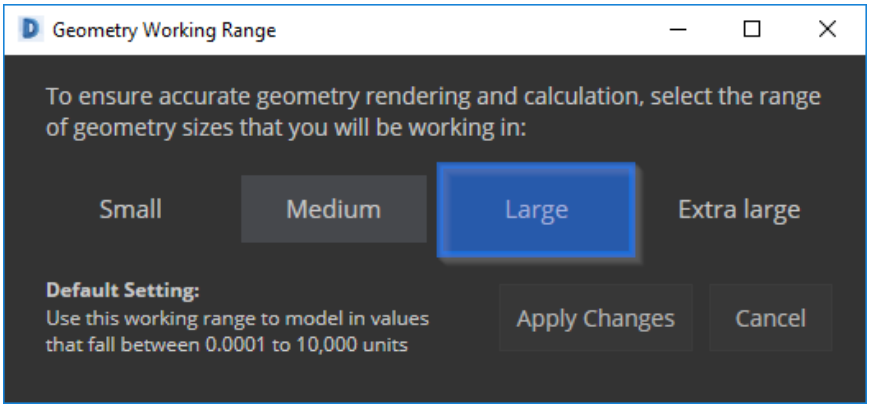
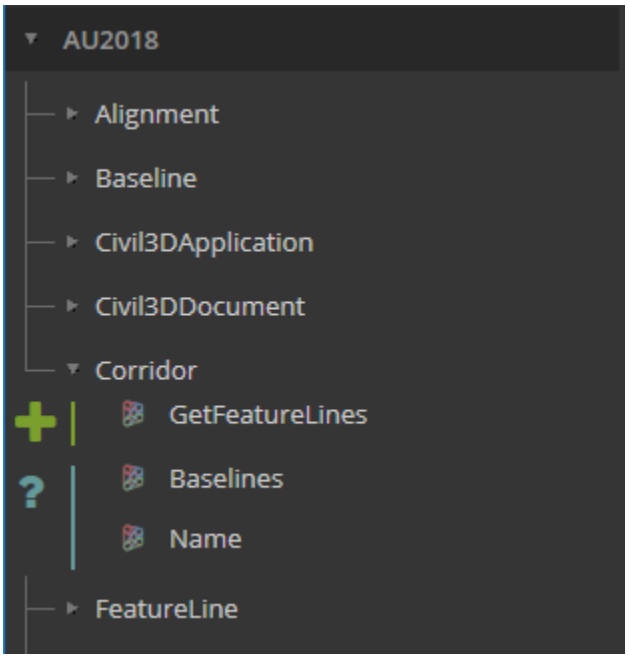


Figure 36: Geometry Working Range dialog

Corridor Nodes



The Corridor.GetFeatureLines node takes in a Civil 3D Corridor and a text string containing the required FeatureLine string code to extract. The output is a list of Civil 3D FeatureLines, which are used downstream to create Dynamo geometry.

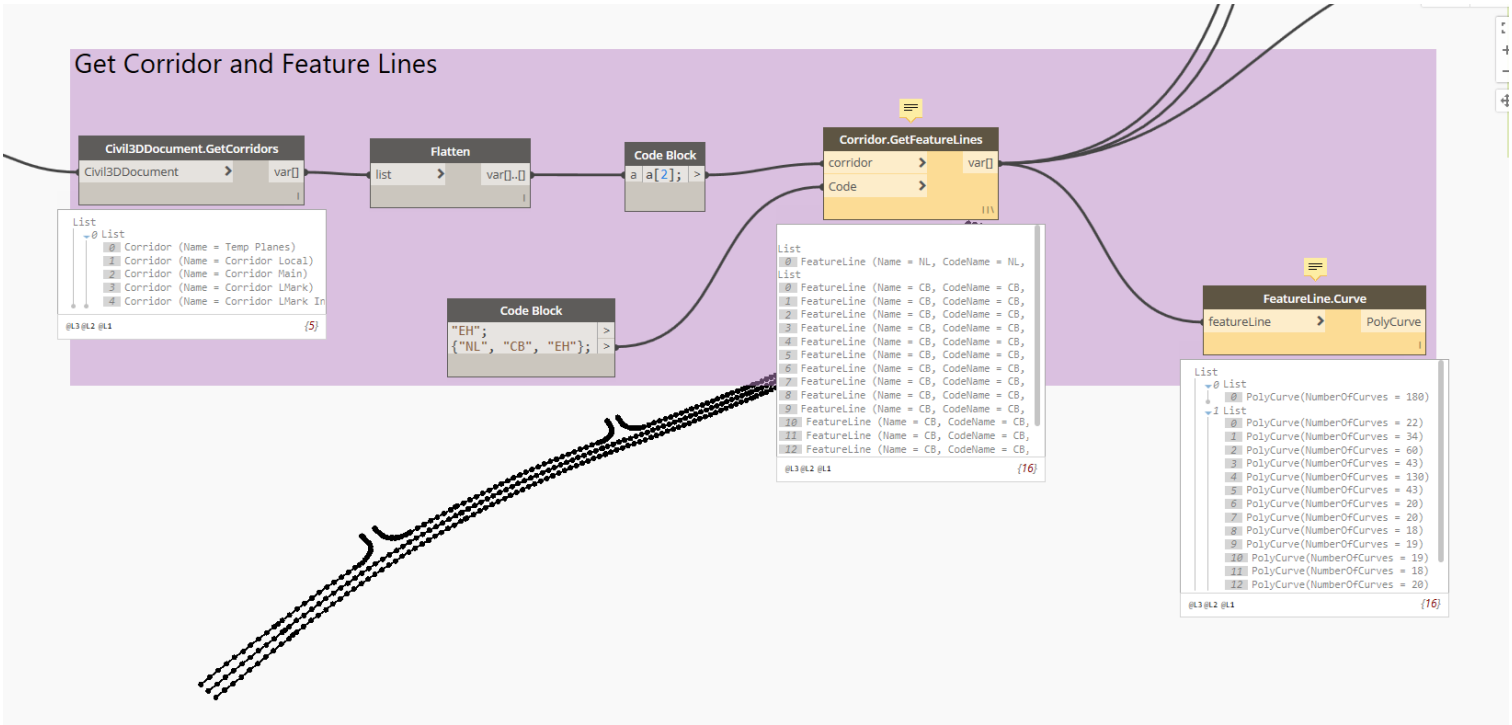
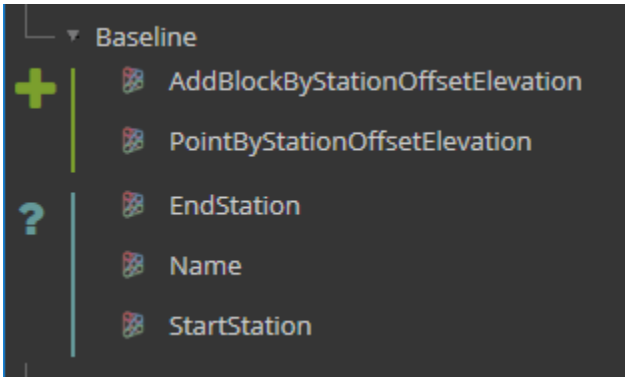


Figure 37: Corridor.GetFeatureLines node with background preview

Baseline Nodes



The Baseline.AddBlockByStationOffsetElevation node will created AutoCAD Block Reference along a provided Baseline, at a specific Station, Offset and Elevation. The output is a new AutoCAD Block Reference handle.

Similarly, the Baseline.PointByStationOffsetElevation will create a list Dynamo points along a Baseline at a specific Station, Offset and Elevation.

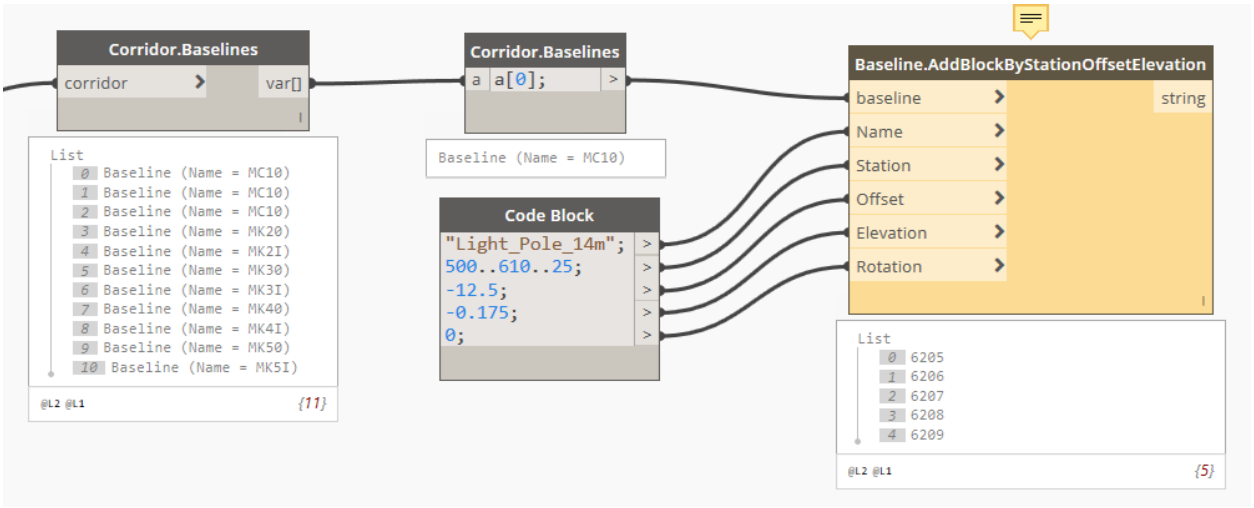


Figure 38: Baseline.AddBlockByStationOffsetElevation node

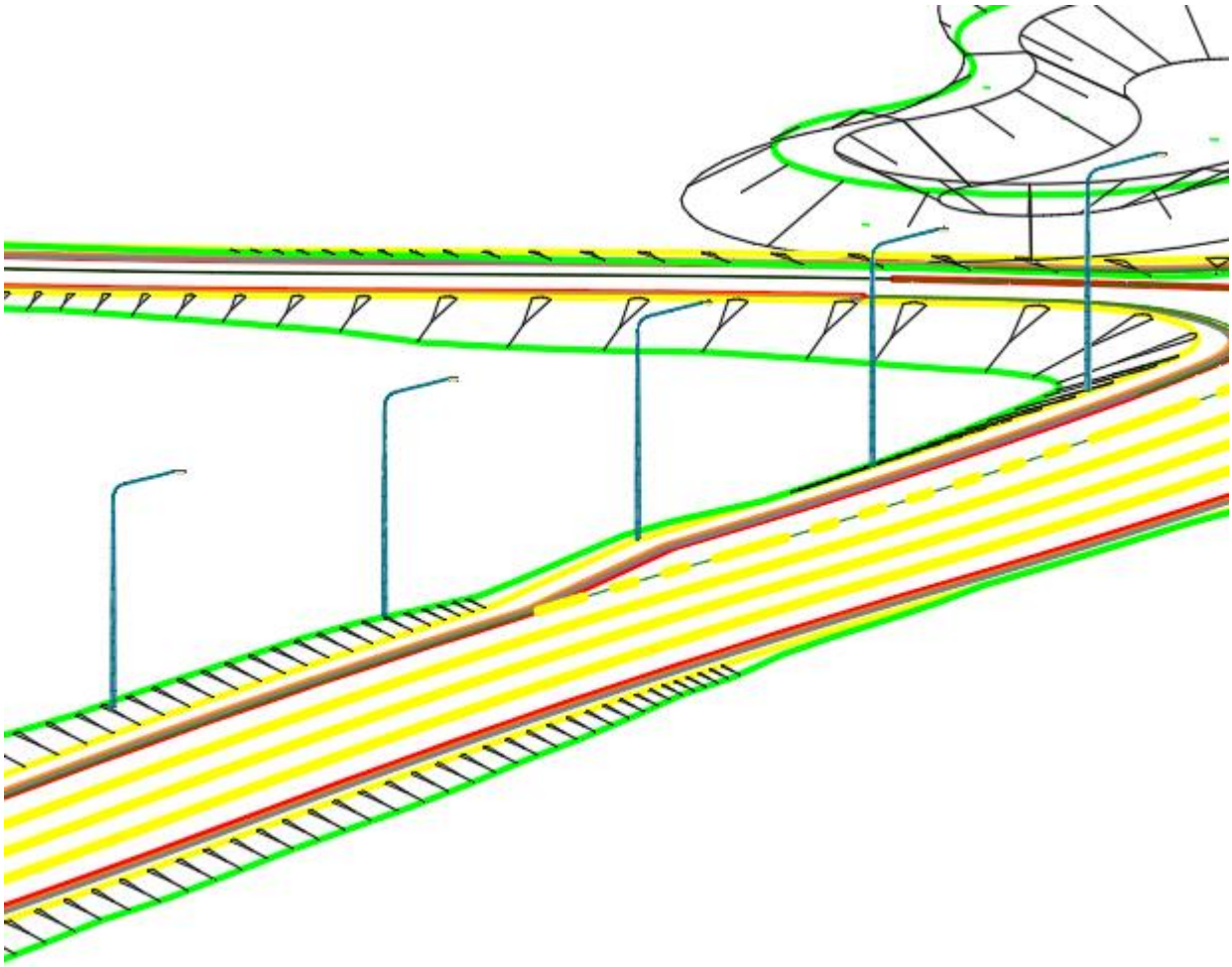
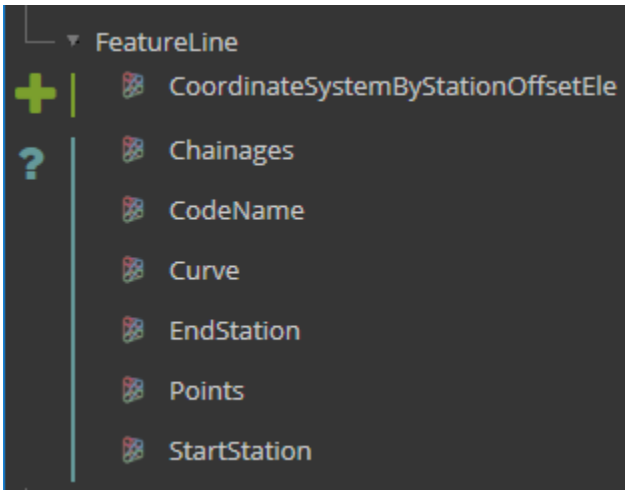


Figure 39: 3D Blocks inserted via Dynamo relative to a Corridor Baseline

FeatureLine Nodes



The FeatureLine.CoordinateSystemByStationOffsetElevation node creates a Dynamo coordinate system relative to a Civil 3D FeatureLine. It requires a FeatureLine, Station, Offset and Elevation to create, and outputs a list of Coordinate Systems relative to the FeatureLine (i.e. The 'Y' vector follows the bearing of the FeatureLine). This can then be used downstream with native Dynamo geometry functions to extract information such as Vectors.

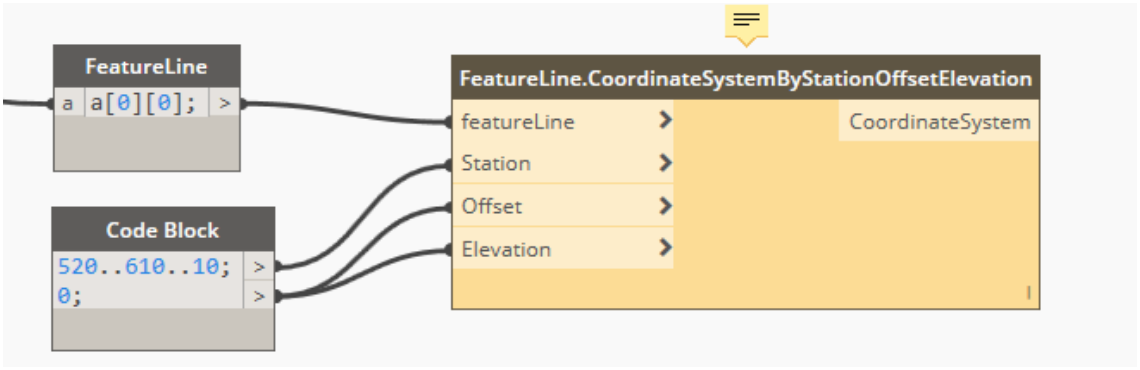


Figure 40: FeatureLine.CoordinateSystemByStationOffsetElevation

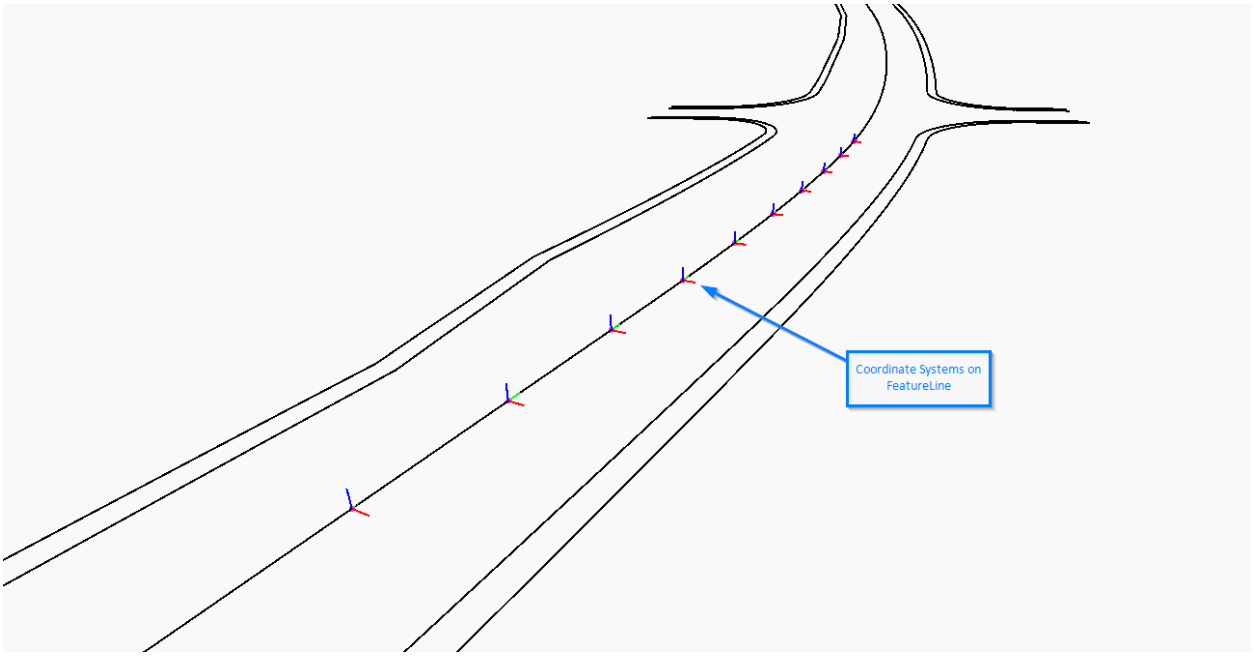


Figure 41: Coordinate System on FeatureLine

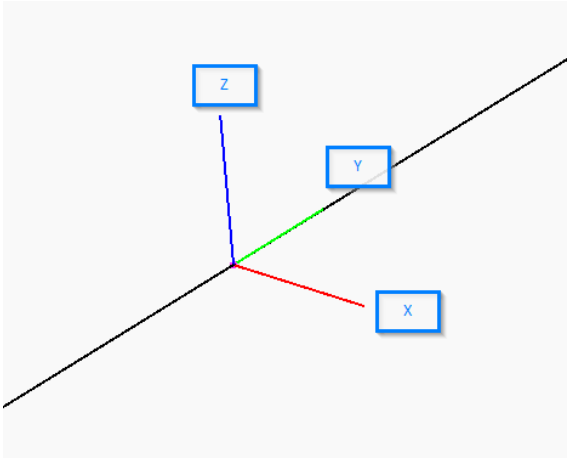


Figure 42: Coordinate System oriented to a Featureline's bearing

The remaining nodes in the FeatureLine category are Query nodes, which return information about the FeatureLine object., including:

- Chainages
- Code Name
- Start Station
- End Station

Additionally, there are two special query nodes that will generate Dynamo Geometry, these are

- Curve Will generate a Dynamo curve of the Civil 3D FeatureLine
- Points Create a list of Dynamo Points from the Civil 3D FeatureLine

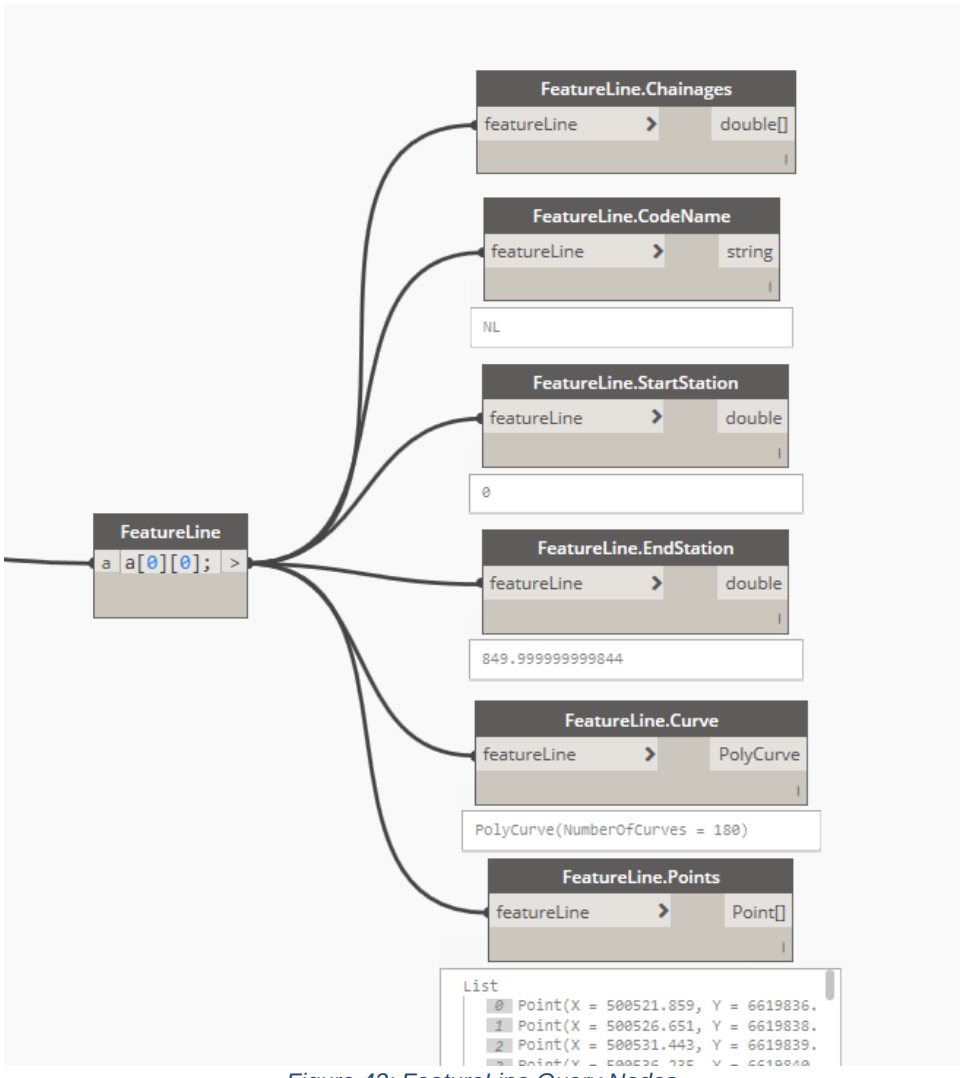


Figure 43: FeatureLine Query Nodes

Learning Objective 4 – Create Civil 3D / CAD geometry from within the Dynamo workspace

In our final exercise we will build on our learnings from the previous exercise and create nodes to assist in the creation of AutoCAD and/or Civil 3D geometry.

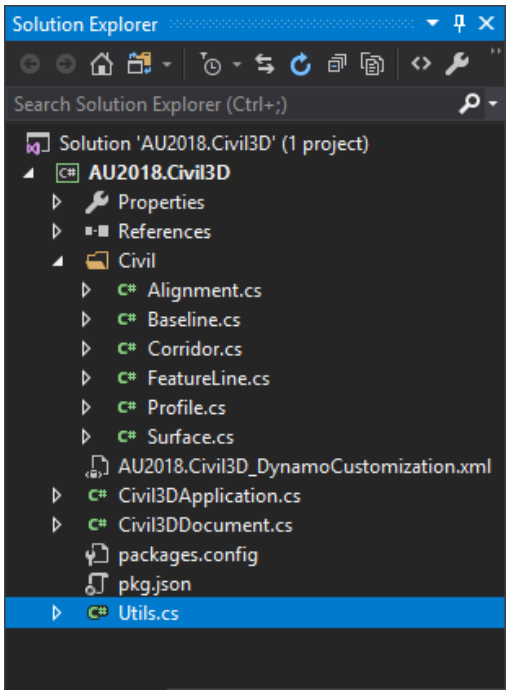
We will use Dynamo to place geometry relative to baselines, and finish by using a populated Excel spreadsheet to control the placement of objects.

Finally, as a bonus, we will use similar techniques to generate 3D solid linemarking from our corridor model, for importing into Navisworks

Visual Studio 2017

New Classes

- **Utils.cs**



.

Utils.cs

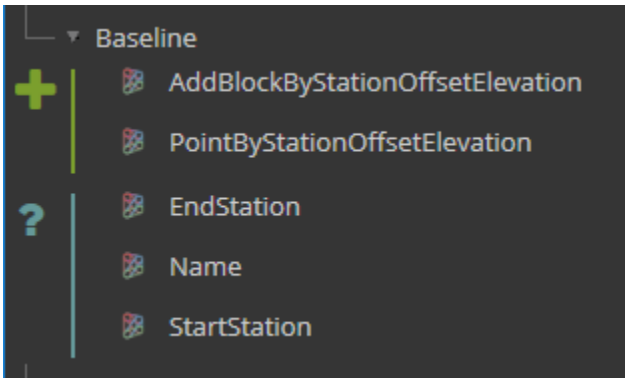
This class is a class made of basic utilities that do not fall into any specific Civil 3D category.

```
/// Create a single AutoCAD / Civil 3D block
[IsVisibleInDynamoLibrary(false)]
public static string AddBlock(AcadDocument doc, Point Point, string Name, double Scale,
double Rotation)
{
    double[] pt = new double[] { Point.X, Point.Y, Point.Z };
    dynamic blk = doc.ModelSpace.GetType().InvokeMember("InsertBlock",
System.Reflection.BindingFlags.InvokeMethod, null, doc.ModelSpace, new object[] {pt,
Name, Scale, Scale, Scale, Rotation });
    return Name;
}
```

This method works with the ‘`AddBlockByStationOffsetElevation`’ method in ‘Baseline.cs’, which takes an AcadDocument, a Dynamo Point, a Block name (as a string), a Scale and Rotation in order to correctly place the block reference in AutoCAD

The `[IsVisibleInDynamoLibrary(false)]` attribute above the method declaration is a Dynamo attribute designed to hide the method from exposure in Dynamo.

Dynamo Studio



Following from the lesson in Learning Objective 3, an extension of the Baseline.AddBlockByStationOffsetElevation node allows us to use an external Excel file to create Civil 3D / AutoCAD objects directory from an Excel file. It should be noted that the script provided (‘*Sample AU 2018 - 03.3 - Place Objects From Excel.dyn*’) does not consider deleting elements prior to their construction, deleting is still a manual process.

The script will read an Excel file and extract information such as Baseline, Block Name, Station, Offset, Elevation, Rotation and Handle. The Handle column is included to allow for automatic updating (delete/create) of AutoCAD Block References (not in icluded in this example)
Once the handle is written back to the Excel file, it opens the possibility of updating the object when the script is re-run.

This approach to discreet element placement from Excel allows more flexibility than setting up the parameters and dimensions directly into a Dynamo script.

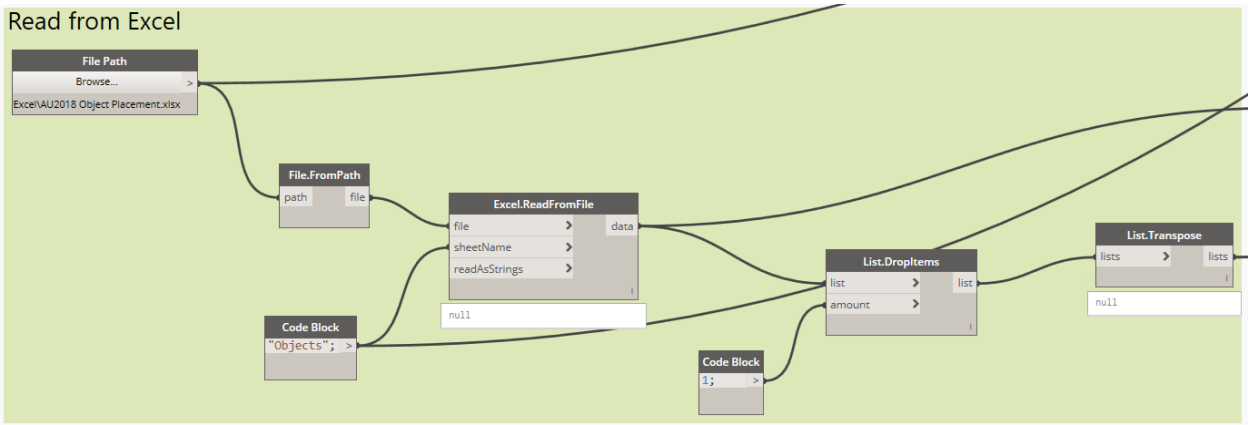


Figure 44: Read parameters from Excel

	A	B	C	D	E	F	G
1	Baseline	Name	Station	Offset	Elevation	Rotation	Handle
2	MC10	Light_Pole_14m	655	9	-0.175	180	
3	MC10	Light_Pole_14m	675	9	-0.175	180	
4	MC10	Light_Pole_14m	695	9	-0.175	180	
5	MK50	Light_Pole_14m	25	-1.8	-0.1	0	
6	MC10	TrafficSignal	616	-12.3	-0.175	0	

Figure 45: Excel Parameters

The final script ("*Sample AU 2018 - 04.1 - Linemarking.dyn*") uses the Corridor.GetFeatureLines node to extract linemarking strings created in the Civil 3D model and, based on the linetype, creates 3D linemarking strings using standard Dynamo nodes.
The ***Civil3DDocument.ImportSolid*** node imports the segmented Dynamo linemarking solid geometry into the Civil 3D Document.
A code block for each linetype specifies the paint and gap lengths for that particular linetype, which is then used to split the FeatureLine, based on measuring points along the curve.
Once the FeatureLIne has been split, the Dynamo Curves are offset according to the linemarking width and a Dynamo Surface is created. The Surface is thickened and, finally, a Dynamo Solid is created.
This Solid is then passed back into the Civil 3D Document, resulting in 3D Solid geometry that is suitable for importing into Navisworks.

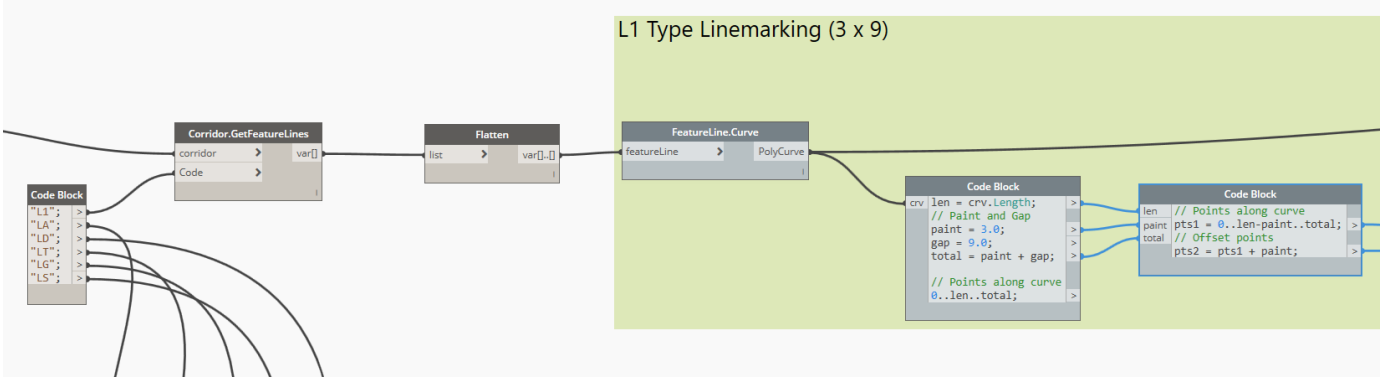


Figure 46: Creating linemarking from FeatureLines

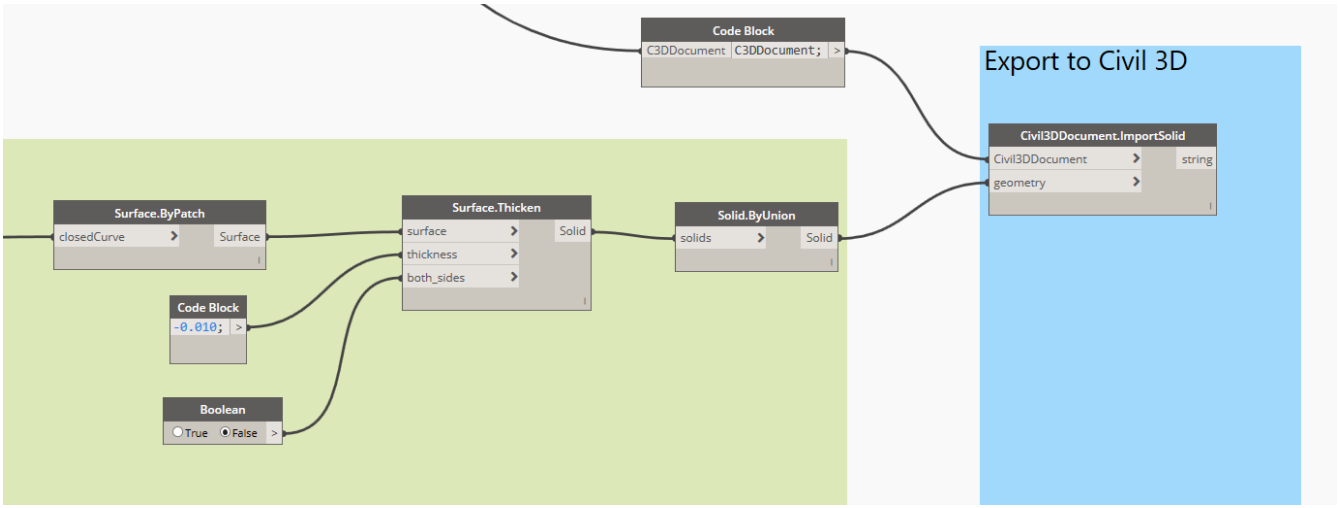


Figure 47: Creating Solids from Dynamo geometry

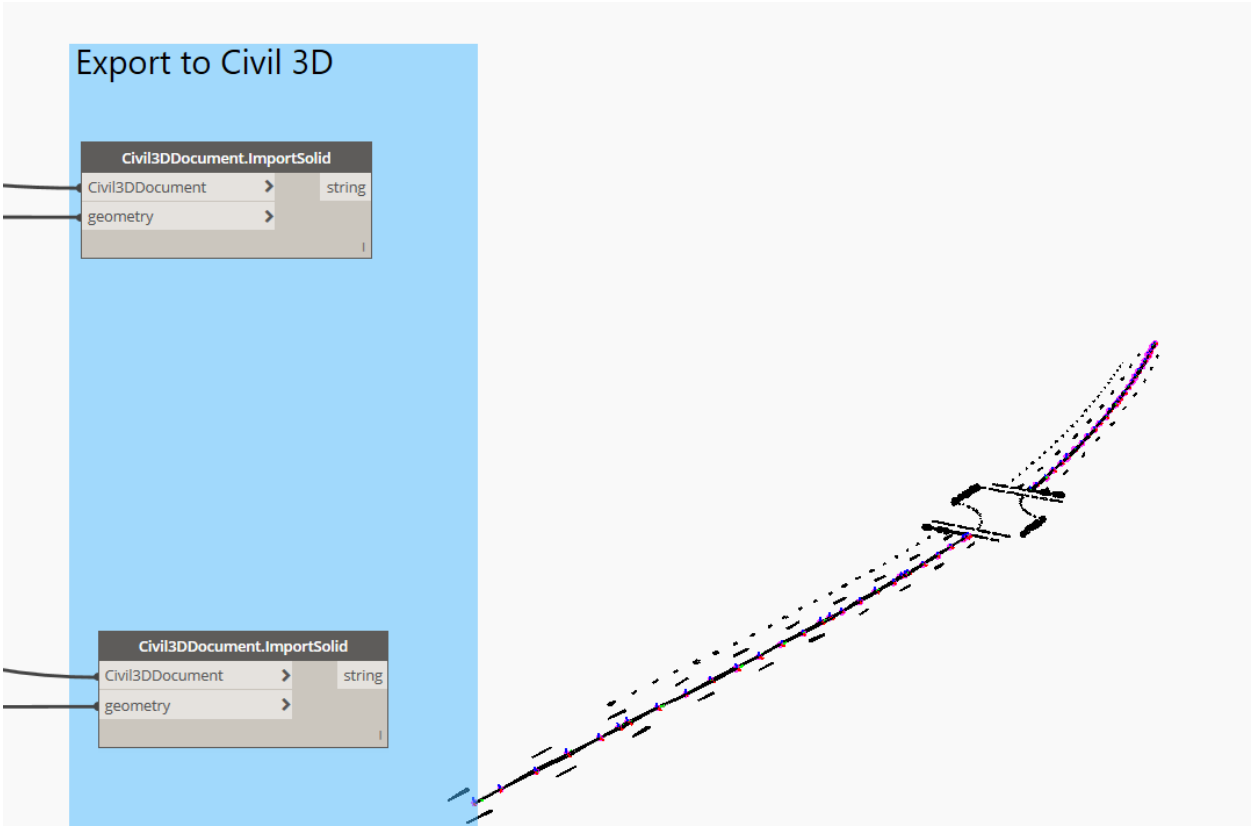


Figure 48: Dynamo Linemarking Solids

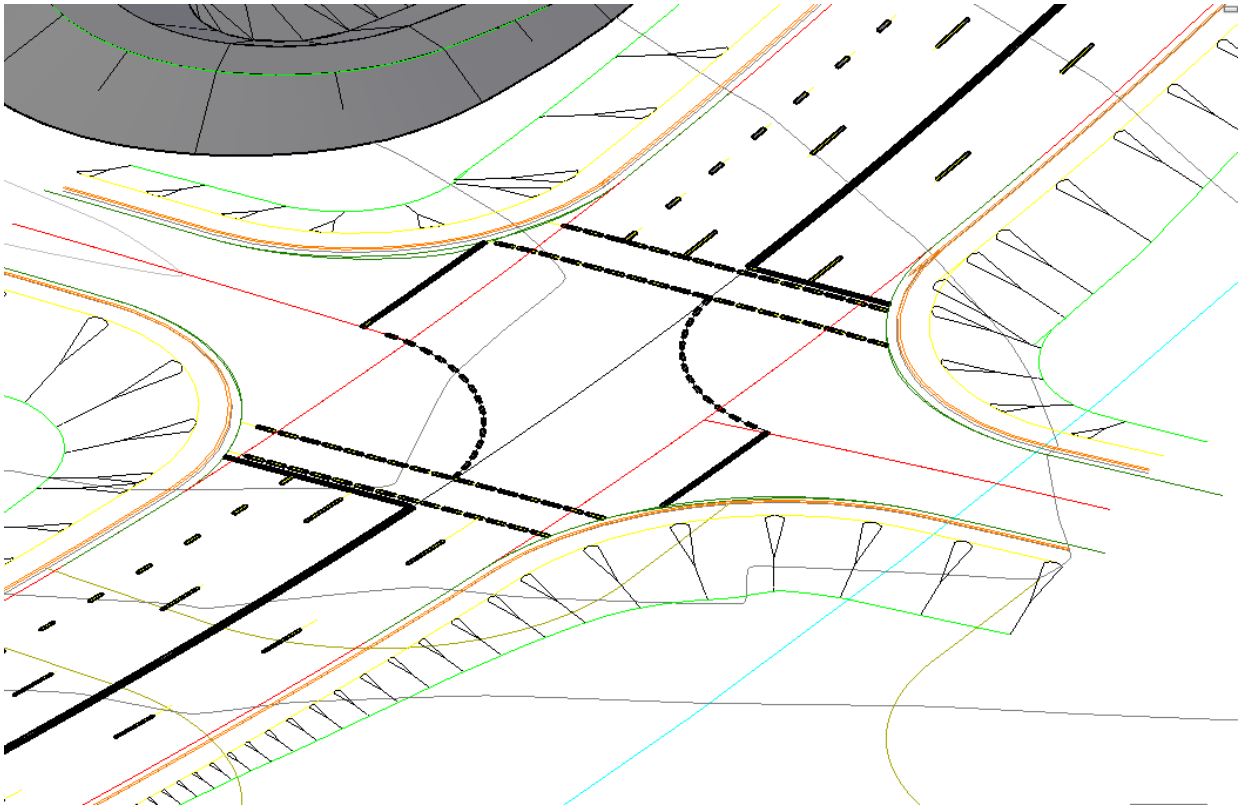


Figure 49: Dynamo Solids imported to Civil 3D

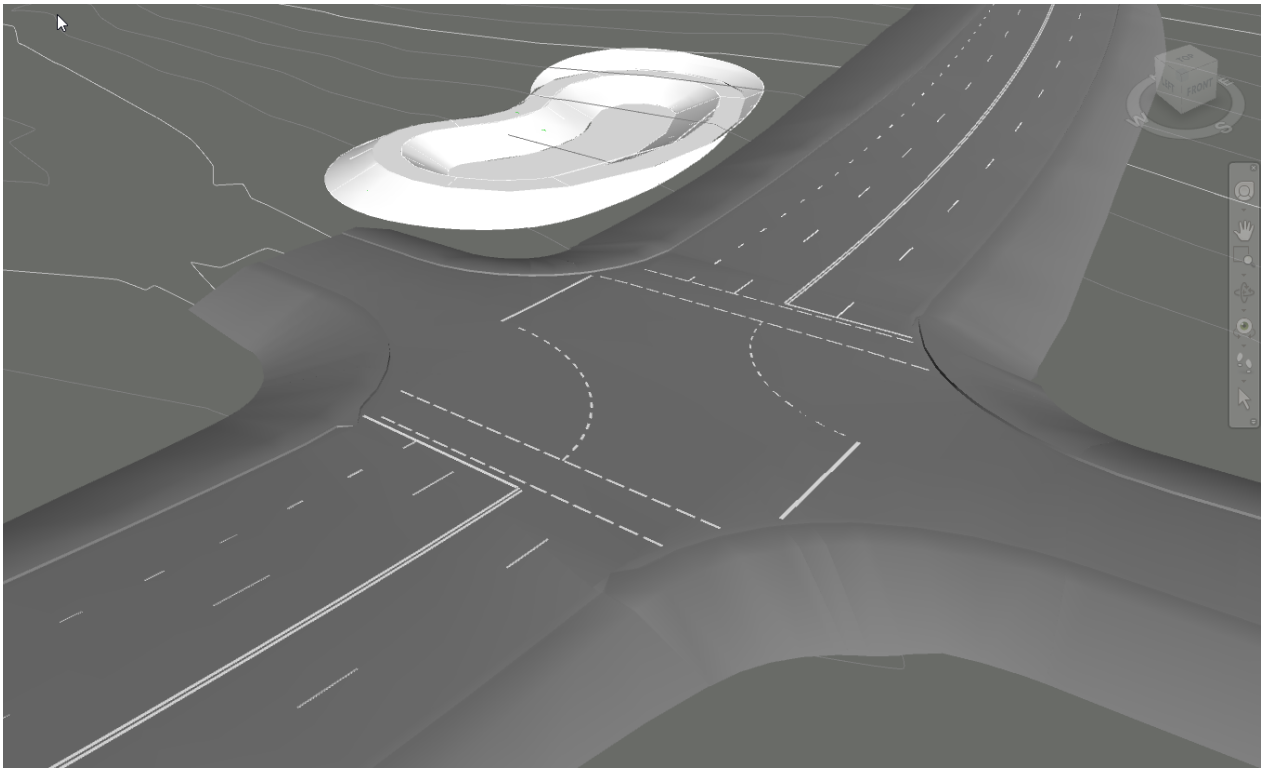


Figure 50: Civil 3D Solids in Navisworks.

Conclusion

Using a combination of Civil 3D's model information and Dynamo's visual scripting capabilities, we have been able to create a set of re-useable tools which have not taken long to setup but allow us great flexibility in extending Civil 3D's native capabilities.

The flexibility of Dynamo, and its ability to read and write Excel files with ease, allows us to extract information from almost any Civil 3D object with ease.

This presentation has just been the tip of the Civil 3D & Dynamo iceberg, and hopefully will inspire you to create some amazing tools.

All the material available in the dataset is provided as open source so feel free to modify it and use it in your own applications. Feel free to contact me on LinkedIn to share workflows and ideas.

Good luck

Andrew.

References

Civil 3D Resources

Civil 3D .Net API Reference
http://docs.autodesk.com/CIV3D/2019/ENU/API_Reference_Guide/index.html

Civil 3D COM API Reference
http://docs.autodesk.com/CIV3D/2012/ENU/API_Reference_Guide/com/getting_started.html

Blog references - Programming with AutoCAD and Civil 3D
https://adndevblog.typepad.com/

Dynamo Resources

Dynamo Zero Touch Plugin Development
https://github.com/DynamoDS/Dynamo/wiki/Zero-Touch-Plugin-Development

Publishing a Dynamo Package
http://primer.dynamobim.org/en/11_Packages/11-4_Publishing.html

Blogs:

http://dynamobim.org/blog/

https://github.com/teocomi/dug-dynamo-unchained

Dynamo Primer
http://dynamoprimer.com/en/