Revit API – An Introduction

Some simple code examples using C# and…

A few projects using the API to generate geometry

Ritchie Jackson

16th March 2011

Presentation Scope

Target Audience

- Novice programmers with some C# experience, familiar with Revit but not the API

Interface

- Macro generation using Visual Studio Tools for Applications (VSTA) & C# ->
- Code can be edited, built and run ‘live’ in a Revit session

Focus

- Underlying *geometry* fundamental to creation of 3D objects ->
- Selection of component type at user’s discretion

Examples

- Elementary geometry with workflow
- Comparison of two component types highlighting commonality of code
- More complex project applications showing workflow

Code Appendix

- Module and Macro setup with full code for first example
- Additional code in accompanying file

Why use the API for Form Generation?

Firstly **:** **–**

- Strategy dependent on object and project specifics

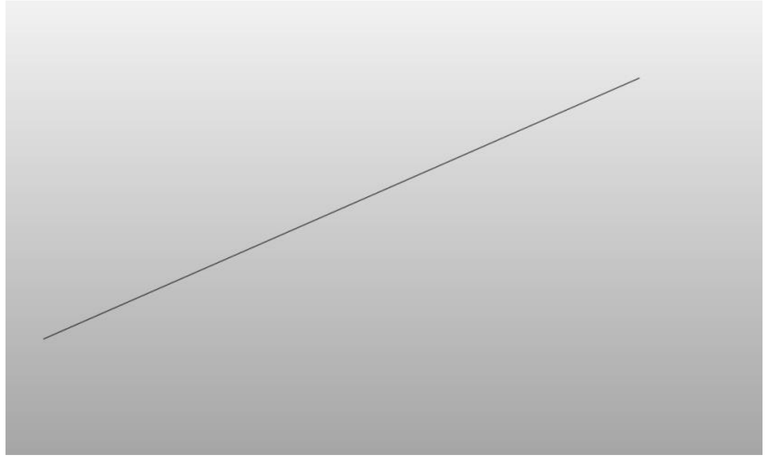
Rough guide : If the object has :-

Features unique to one project?	-> do it manually – in-place or system family
Dimensional variation	-> do it manually – family parameters
Repeated form with variation	-> do it manually – nested families and / or adaptive components
Complex formulae & conditionals	-> grey area – manual might work
Complex object dependencies	-> using the API begins to make sense

But :-

- When starting to use the API, the learning process dictates that the above rules will have to be ignored – *walk then run*

Simple API Macro Examples with C# Code

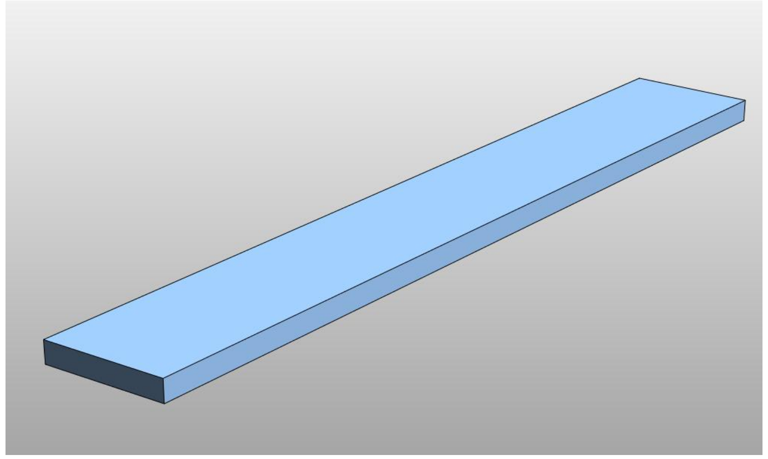


Basic_01_Line : Single line

- About as simple as it gets

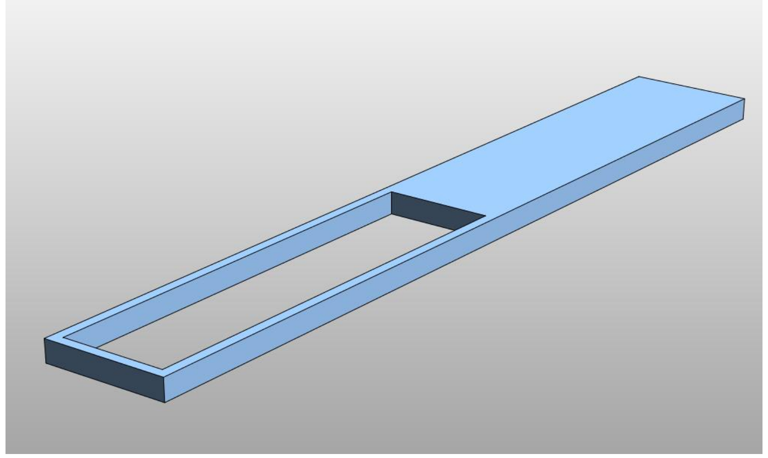
- Historical Note :-
  1980's migration to CAD –
  Ubiquitous outcry –
'I can do it faster on the drawing board'

- Well, you've got to start somewhere…



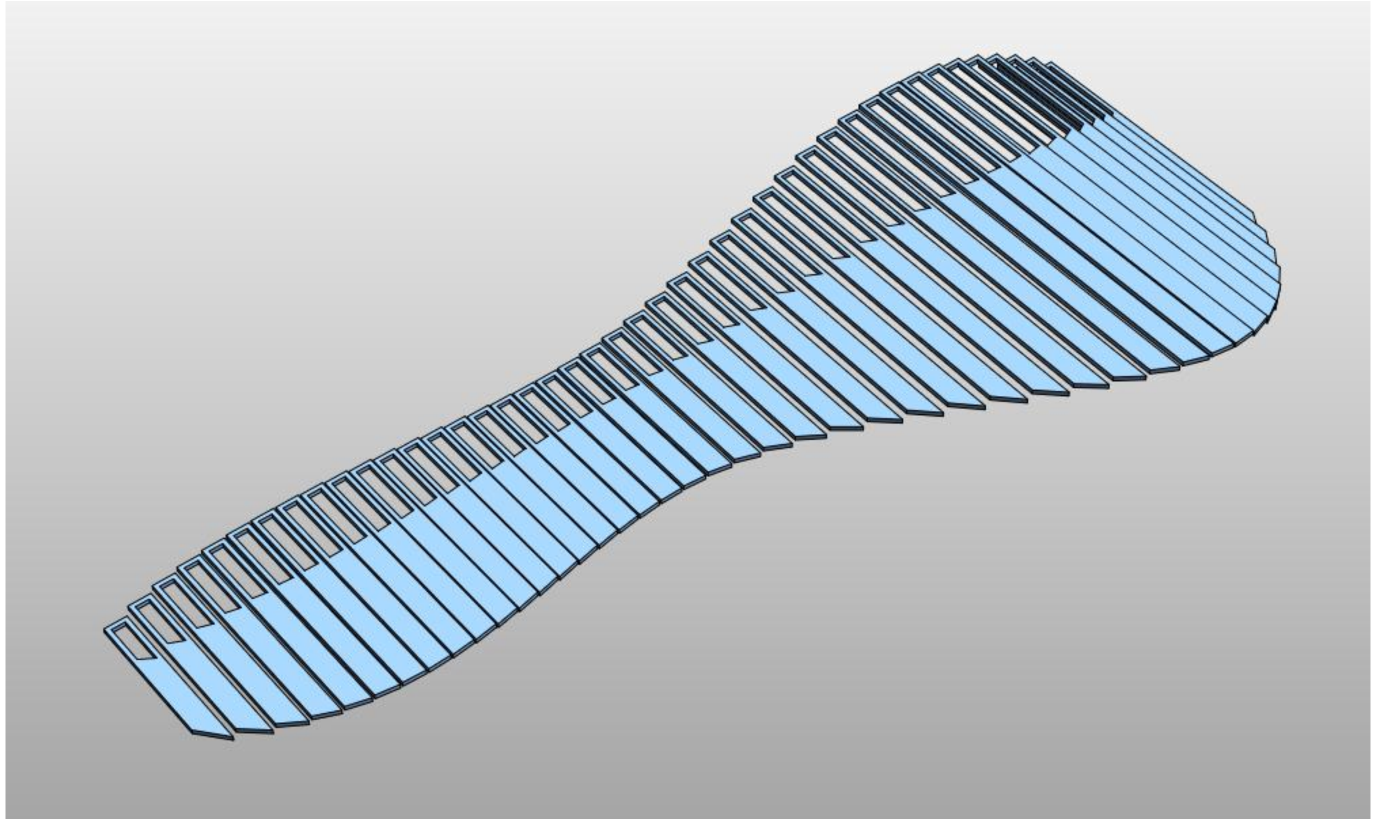
Basic_02_Extrusion : Simple box

- Add three more lines in a closed loop and make a solid



Basic_03_Extrusion : Box+cutout

- A closed loop within a closed loop



Basic_04_BoardWalk : Iteration

- Start to leverage the power of the API
- Control the length and height offset of individual boxes using a spline profile's y-Axis offset and duplicate the boxes along the spline's x-Axis length

Workflow Comparison : Generic Model vs. System Floor

Generic Model Code

```
// Array of Arrays for the Profiles
CurveArrArray extrudeProfile

      // Extrusion Plane
      Plane plane <- floorPerimeter
      SketchPlane <- plane
      // Create the Floor
Extrusion floor <-
(extrudeProfile, planeSK, thickness)
```

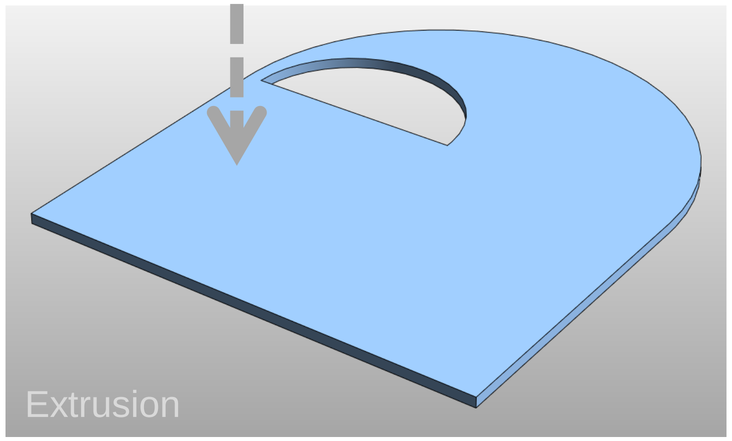
Pseudo-Code Workflow

Common Code

```
// Floor Slab Perimeter Points
XYZ pnt01
XYZ pnt02
XYZ pnt03
XYZ pnt04
XYZ pnt05
// Perimeter Geometry
Line line01 <- (pnt01 -> pnt02)
Line line02 <- (pnt02 -> pnt03)
Line line03 <- (pnt04 -> pnt01)
Arc arc01 <- (pnt03, pnt04, pnt05)
// Place Perimeter elements in an Array
CurveArray floorPerimeter <-
              (line01, line02, arc01, line03)
// Floor Slab Cut-out Points
XYZ pnt06
XYZ pnt07
XYZ pnt08
// Cut-out Geometry
Line line04 <- (pnt06 -> pnt07)
Arc arc02 <- (pnt06, pnt07, pnt08)
// Place Cut-out elements in an Array
CurveArray floorCutout <-
              (line04, arc02)
```

System Floor Code

```
// Create the Floor
Floor floor1 <- (floorPerimeter)



// Constraining Levels
Level levelLo
Level levelHi
// Create the Cut-out
Opening floorCut <-
  (levelLo, levelHi,
   floorCutout)
```

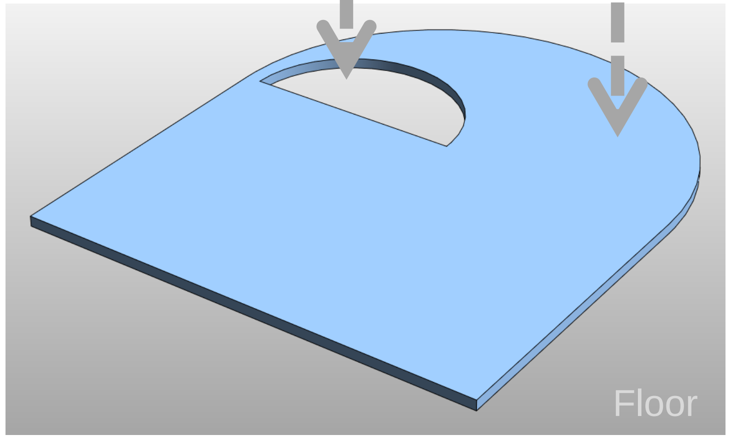


Learning :-

In many cases it’s easier to use a Generic Model

Final Document :-

Imperative to have the correct type – If it’s a floor then use System Floor

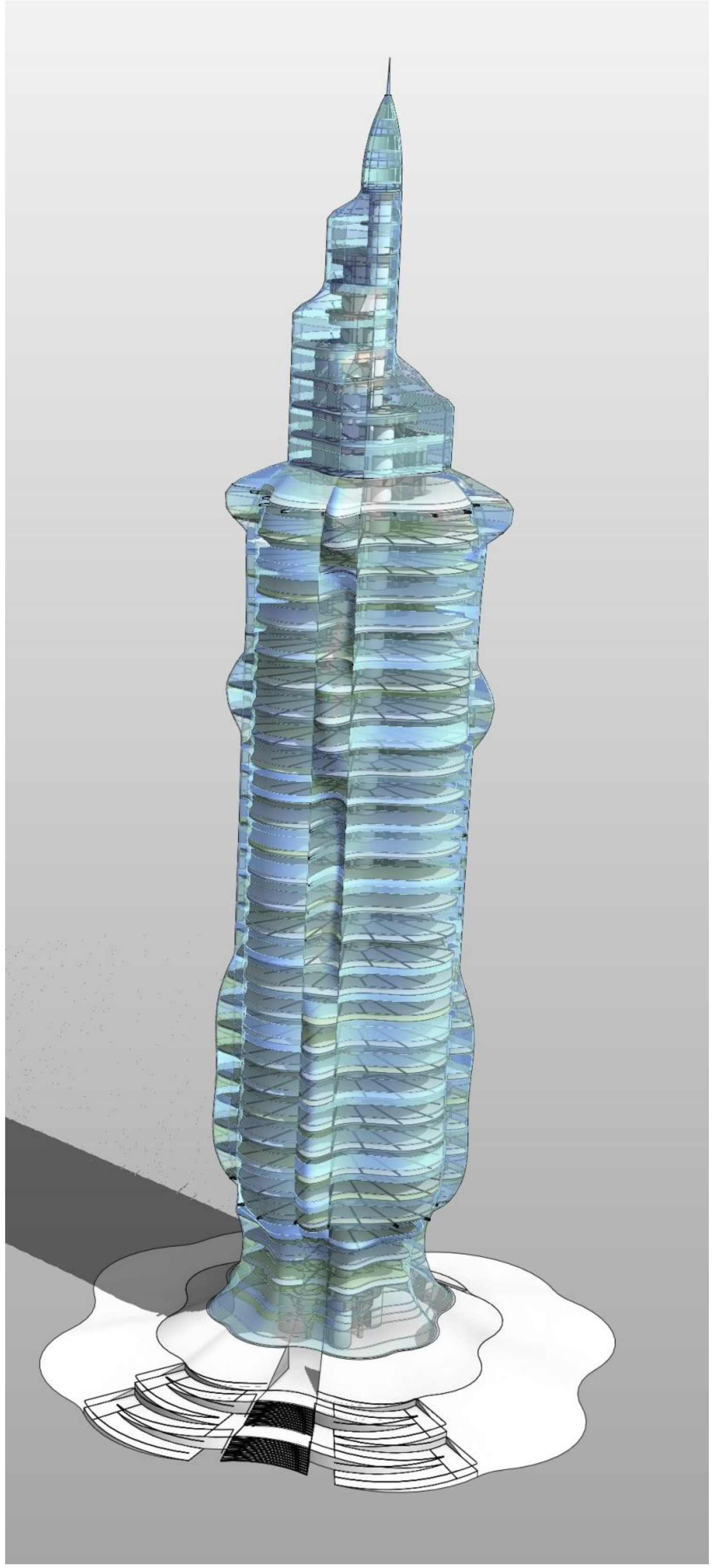


Project API Examples

- Generic Model Family used for all API-constructed elements in order to simplify learning process -
- Regardless of component type, underlying geometry (points, lines, curves, …) is similar for all objects

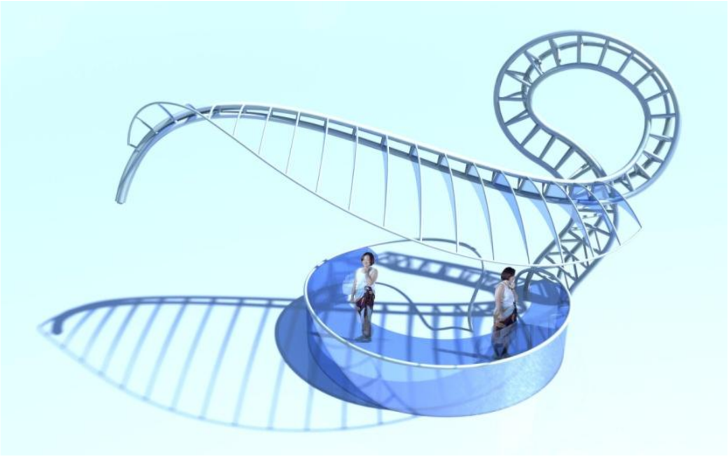
High Rise

- *API Scope:-*
- Façade Set out
- Façade Panels
- Façade Materials
- Floor plates
- Beams
- Beam Materials



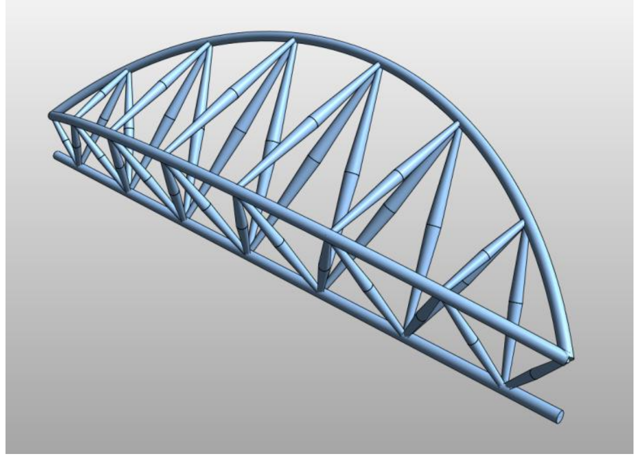
Roller-Coaster Reception

- *API Scope:-*
- All elements
- Materials



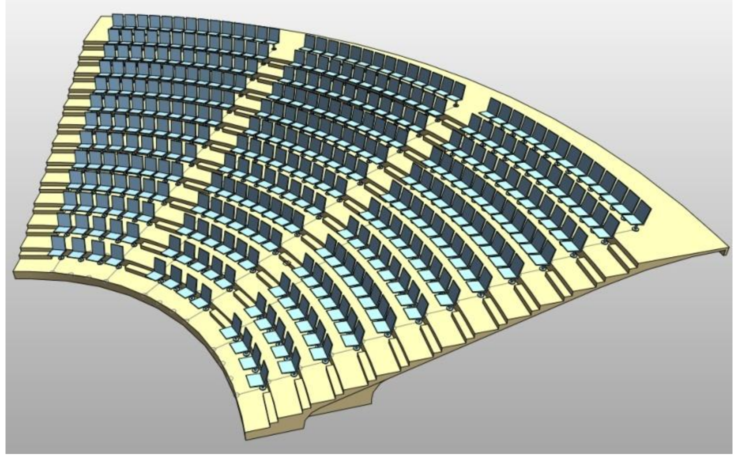
Truss

- *API Scope:-*
- All elements

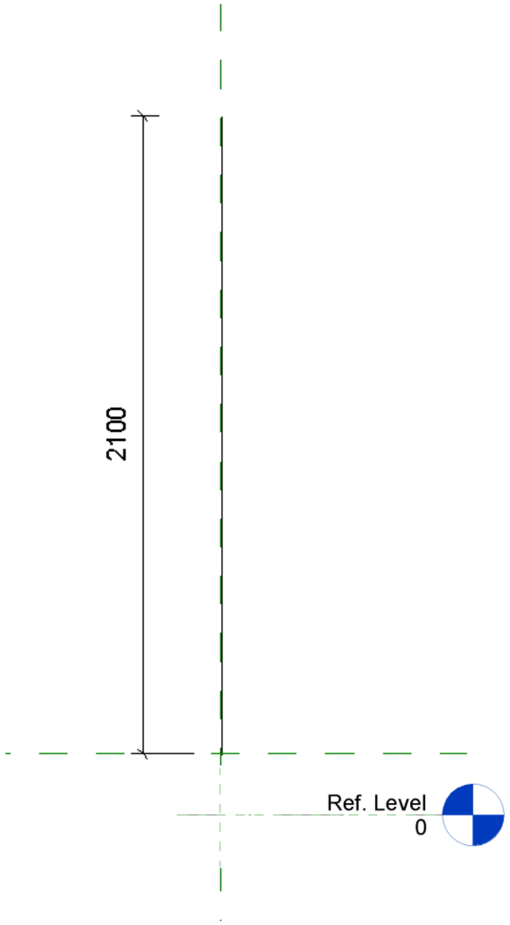


Amphitheatre

- *API Scope:-*
- Set out only



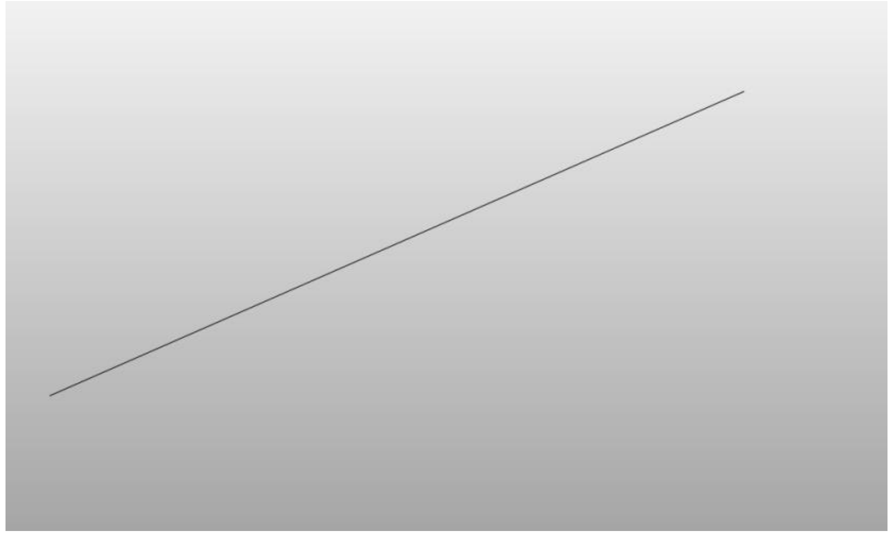
Macro : Basic_01_Line : A single line



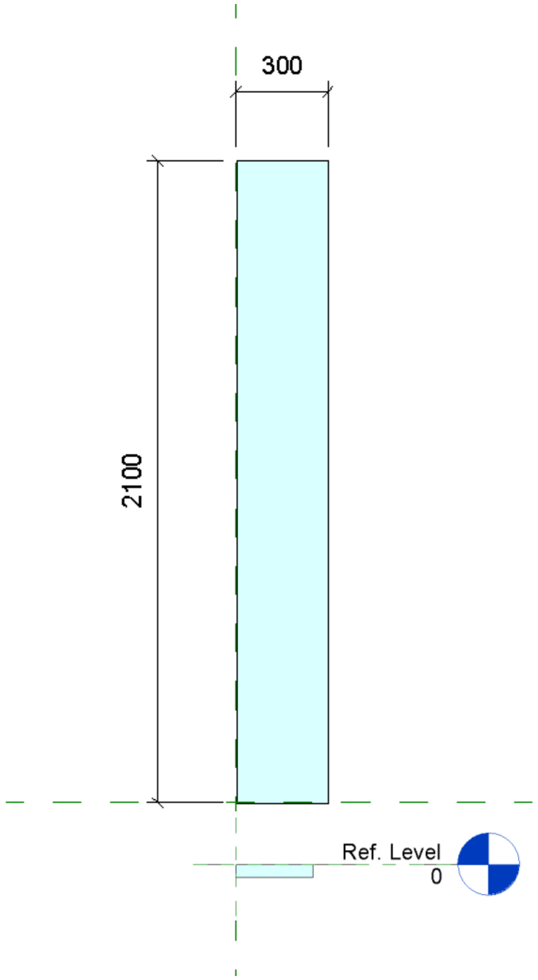
- 7 lines of code – not very promising

Pseudo-Code Workflow : Full Code in Appendix

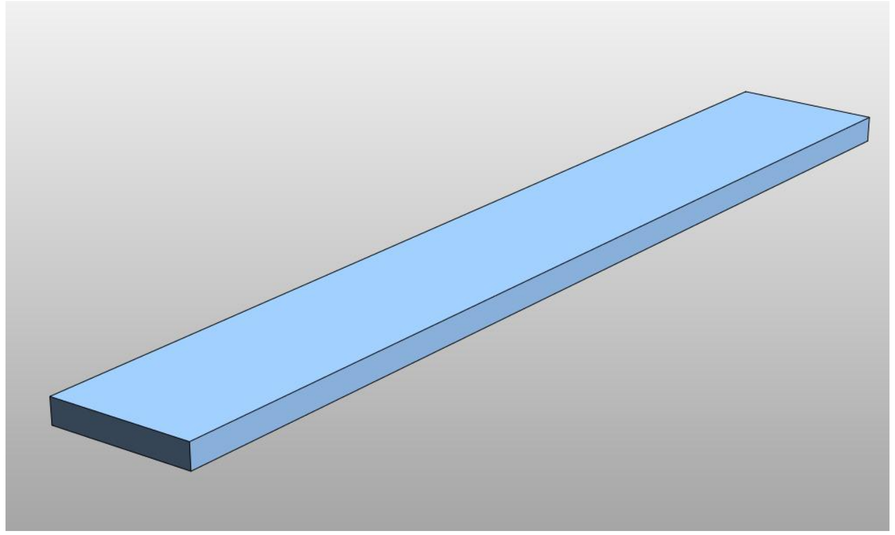
```
// Convert Revit's internal 'Imperial Feet' to Millimeters
double ftMM <- 1 / 304.8

// Define two points on the Line in millimeters
XYZ point01 * ftMM
XYZ point02 * ftMM

// Create the underlying Line geometry
Line line01 <- (point01 -> point02)

// Define the Plane for the Line placement
// using the X-Axis, Y-Axis, Origin
Plane plane <- XYZ.BasisX, XYZ.BasisY, point01

// Create a Sketch Plane from this Plane
// in order to display the Line
SketchPlane planeSK <- plane

// Display the Line on the Sketch Plane
ModelLine line01M <- line01, planeSK
```



Macro : Basic_02_Extrusion : A simple box



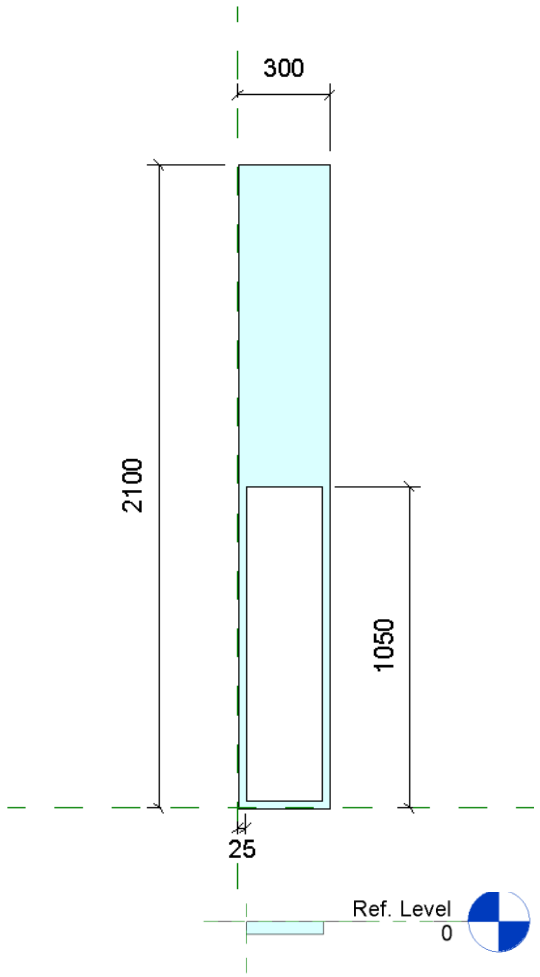
- 19 lines of code – slight improvement



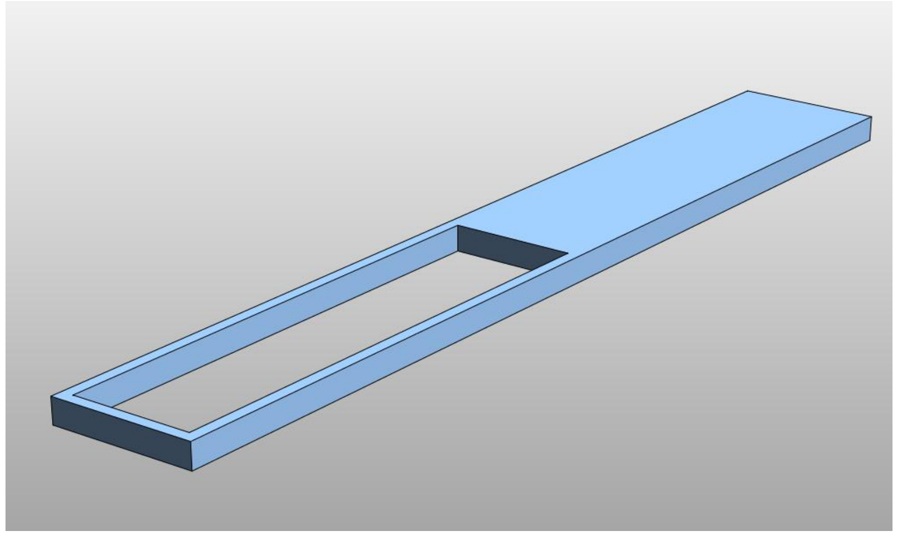
Pseudo-Code Workflow : Full Code in accompanying file

```
// Convert Revit’s internal ‘Imperial Feet’ to Millimeters
double ftMM <- 1 / 304.8

// Define four points for the Profile
XYZ point01 * ftMM
XYZ point02 * ftMM
XYZ point03 * ftMM
XYZ point04 * ftMM

// Create the underlying Line geometry
Line line01 <- (point01 -> point02)
Line line02 <- (point02 -> point03)
Line line03 <- (point03 -> point04)
Line line04 <- (point04 -> point01)

// Create an Array to hold the closed-loop of Profile Edges
CurveArray curveAr1

// Place the single closed-loop of Profile Edges in the Array
line01 -> curveAr1
line02 -> curveAr1
line03 -> curveAr1
line04 -> curveAr1

// Create an Array of Arrays - allows for multiple profiles
CurveArrArray curveArAr
// Place the closed-loop in the Array of Arrays
curveAr1 -> curveArAr

// Define the Plane for the Extrusion placement
// using the X-Axis, Y-Axis, Origin
Plane plane <- XYZ.BasisX, XYZ.BasisY, point01

// Create a Sketch Plane from this Plane
// in order to display the Line
SketchPlane planeSK <- plane

// Create the Extrusion
Extrusion extrude <- curveArAr, planeSK, thickness
```

Macro : Basic_03_Extrusion : A box with cutout



- 33 lines of code – getting there



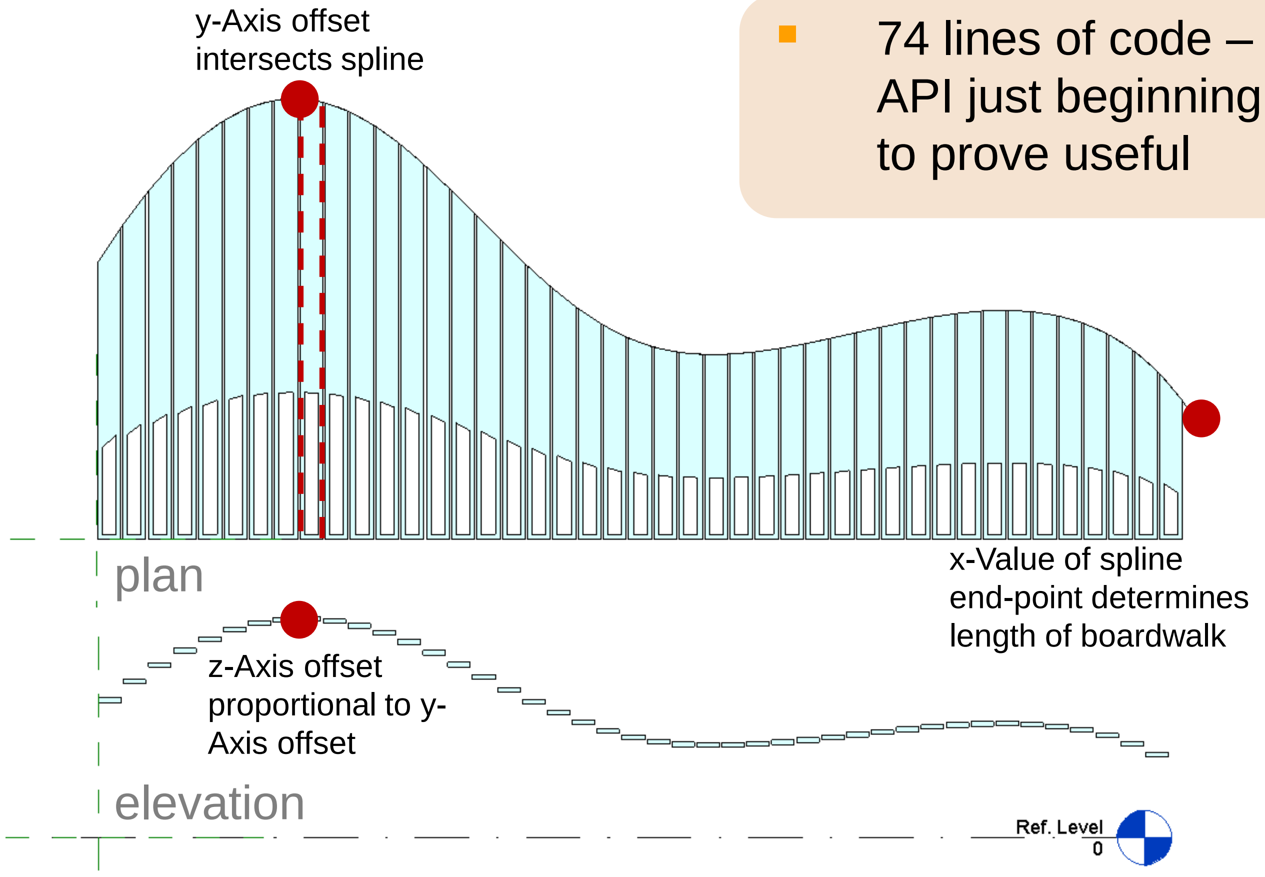
Pseudo-Code Workflow : Full Code in accompanying file

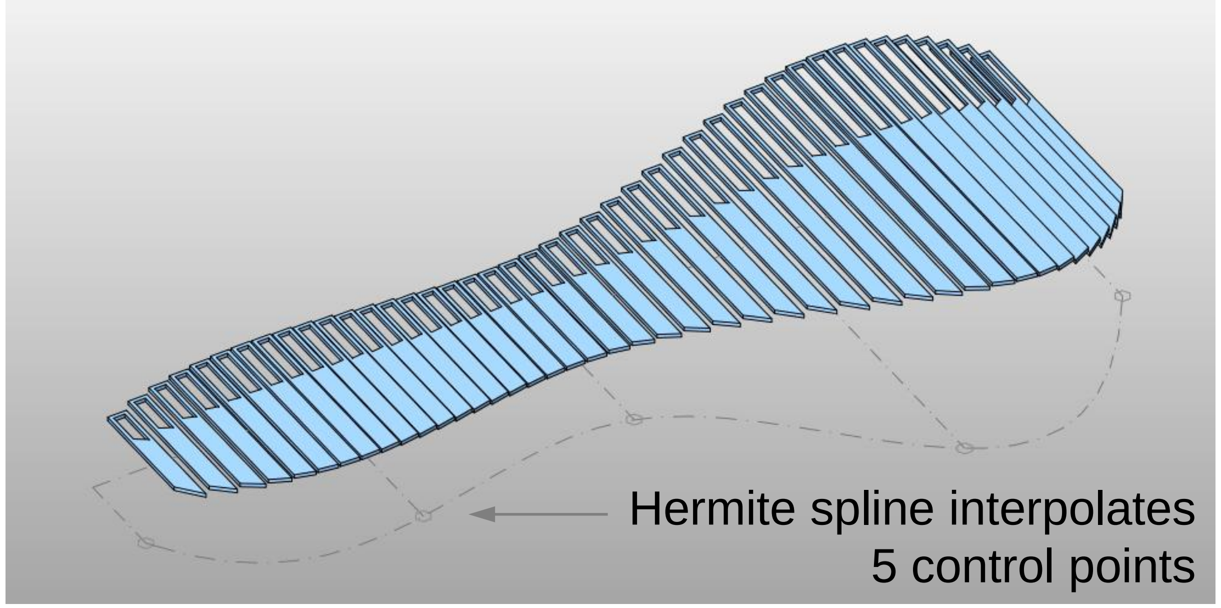
```
// Add the following to the previous example

// Define four more points for the second profile
XYZ point05 * ftMM
XYZ point06 * ftMM
XYZ point07 * ftMM
XYZ point08 * ftMM

// Create the underlying Line geometry
Line line05 <- (point05 -> point06)
Line line06 <- (point06 -> point07)
Line line07 <- (point07 -> point08)
Line line08 <- (point08 -> point05)

// Create an Array for the 2nd closed-loop of Profile Edges
CurveArray curveAr2

// Place the 2nd closed-loop of Profile Edges in the 2nd Array
line05 -> curveAr2
line06 -> curveAr2
line07 -> curveAr2
line08 -> curveAr2

// Place all the closed-loops in the Array of Arrays
curveAr2 -> curveArAr

// Re-use the Extrusion command –
// the Array of Arrays now holds both inner and outer Profiles
Extrusion extrude <- curveArAr, planeSK, thickness
```

Macro : Basic_04_BoardWalk : Iteration

Stringing the boxes together…

- 74 lines of code – API just beginning to prove useful





Hermite spline interpolates 5 control points

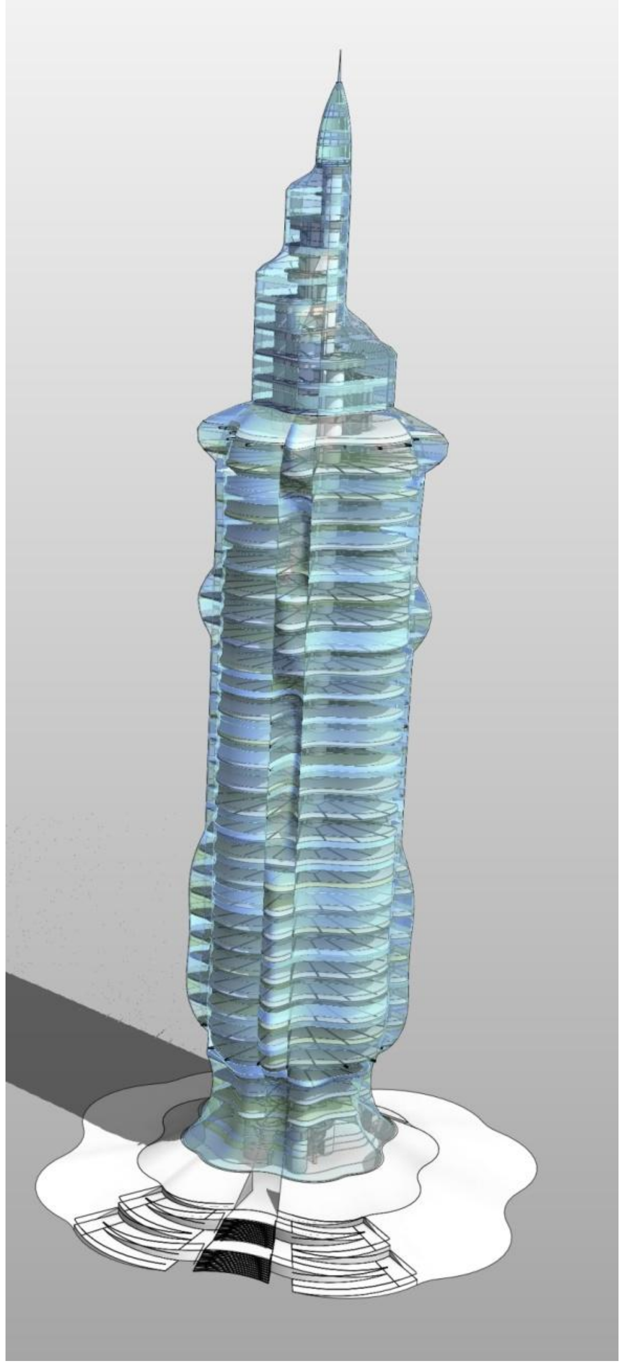
Pseudo-Code Workflow : Full Code in accompanying file

```
// Define the Interpolation Points for a Spline
// controlling one edge of the Boardwalk
XYZ pntS01 * ftMM    XYZ pntS02 * ftMM    XYZ pntS03 * ftMM
XYZ pntS04 * ftMM    XYZ pntS05 * ftMM

// Define a List to hold the Spline Points
IList<XYZ> splinePnts = new List<XYZ>();
// Add the Points to this List
pntS01 -> splinePnts    pntS02 -> splinePnts    pntS03 -> splinePnts
pntS04 -> splinePnts    pntS05 -> splinePnts
// Create the Spline
HermiteSpline spline1 <- (spline1Pnts)

// Define the Plank spacing
double plankSpace <- plankWidth + gap * ftMM
// Determine the number of required Planks ->
// Overall length is the x-Value of the last point on the Spline
int numberOfPlanks <- pntS05.X / plankSpace

// Create the BoardWalk
for (int i = 0; i < numberOfPlanks; i++)
{
    // x-Offset for the bottom-left corner of the current Plank
    double start <- i * plankSpace
    // Project a line parallel to the y-Axis through this point
    // to find the intersection with the Spline and hence the length
    double leftSideLength
    // Do the same for the bottom-right plank corner
    double rightSideLength
    // Send this information to the ‘drawPlank’ function
    drawPlank(start, leftSideLength, rightSideLength)

}
// Encapsulate the amended code for ‘box-with-cutout’ in a function
// but allow for size and placement variations
drawPlank(start, leftSideLength, rightSideLength)
{
    // 'length1' -> Box left side, 'length2' -> Box right side
    // 'edgeOffSet' -> cut-out offset, 'xPosition' -> x-Axis offset
}
```

High-Rise : API Scope

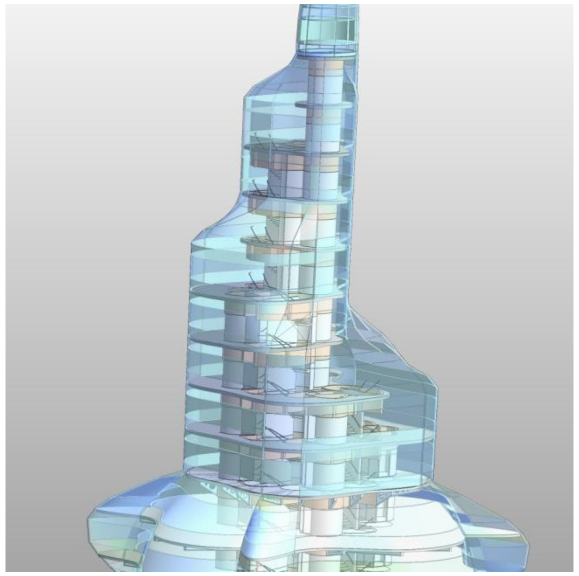
Input Parameters :-

- Control floor profiles
- Control floor level location
- Beam springing points
- Number of Floors
- Floor-to-Floor height
- Transom heights



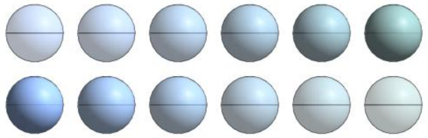
Apex :-
API used for glazing –

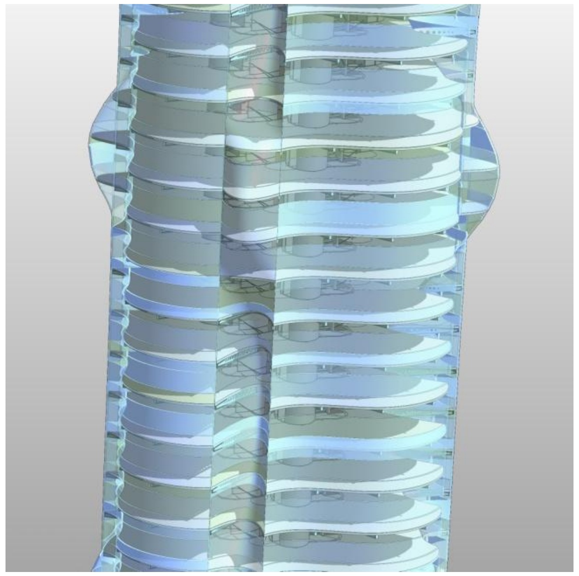
Code from Tower adapted



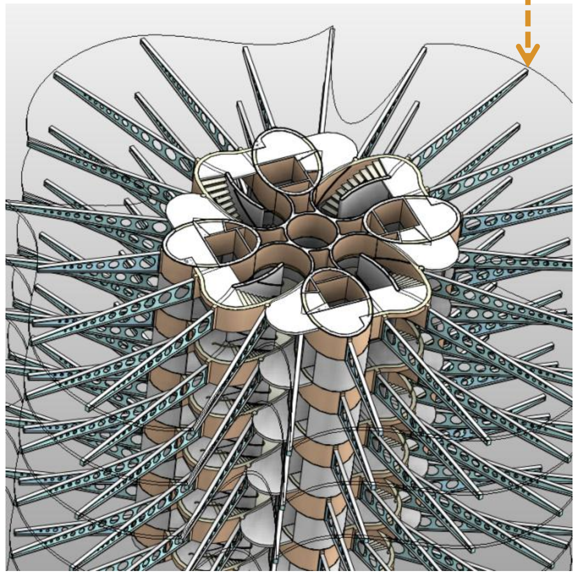
Tower :-
API used for glazing, floor plates and beams.

API assigned finish to façade panels – randomly chosen from 12 materials (6 colours x 2 reflectance values)





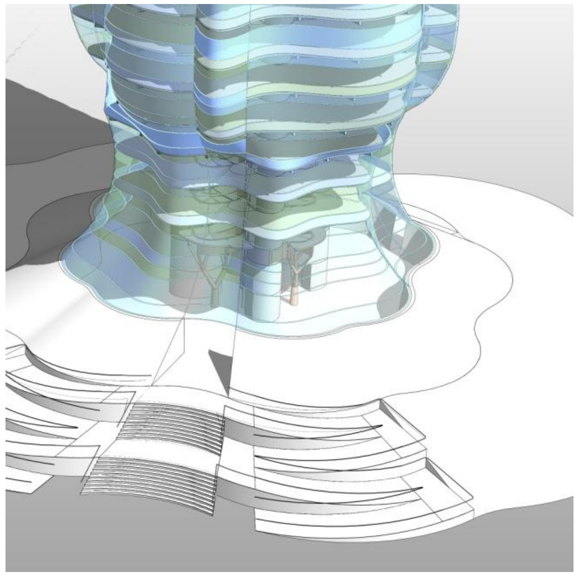
API adapts beam end-points to slab profile



Core :-
Fixed, with static springing points for floor plate support structure

Podium :-
API used for glazing –

all other elements generated conventionally via UI

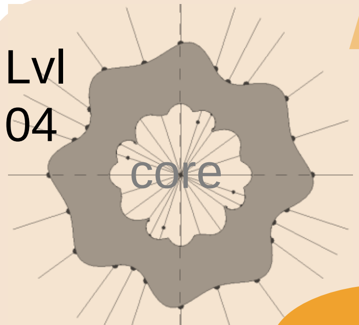


High-Rise : Workflow

1

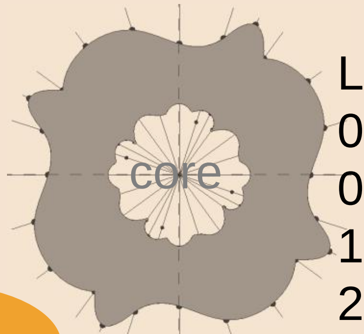
start

Control Floor Plates



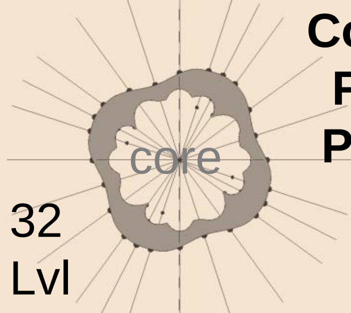
Lvl 04

Type-01



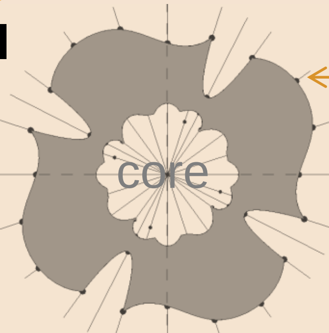
Type-02

Lvl 06 09 12 24 30



32 Lvl

Type-04



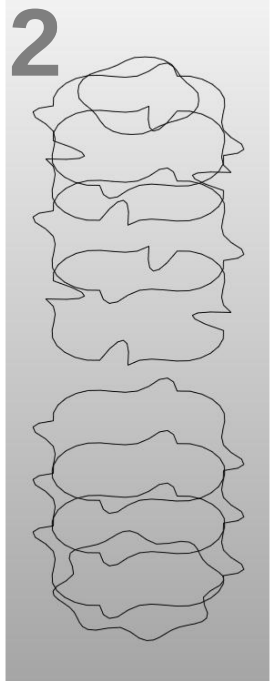
Type-03

20 28 Lvl

24 Control Points and 8 Splines per Floor

Assign Control Floor Plates to Levels

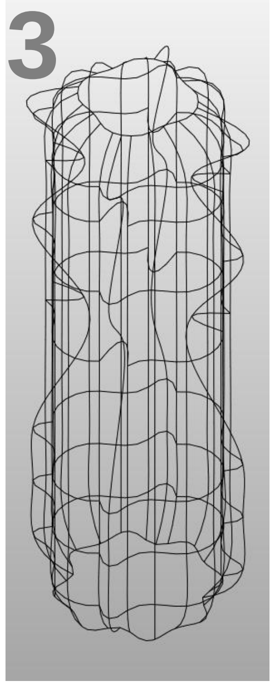
2



Lvl - Type
32 - 04
30 - 02
28 - 03
24 - 02
20 - 03
12 - 02
09 - 02
06 - 02
04 - 01

Generate Spline Mullion Controls

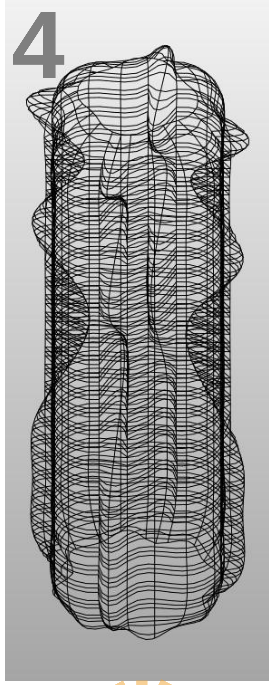
3



24 Mullion Splines interpolating Floor Plate Control Points

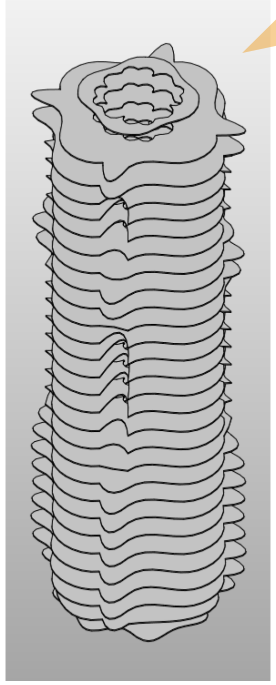
Generate intermediate Floor and Transom set outs

4

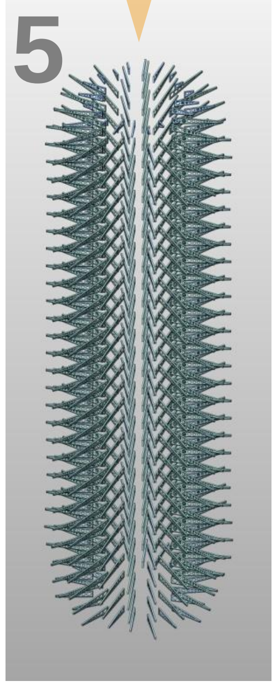


Transoms at +900 and + 2700 from Floor Level

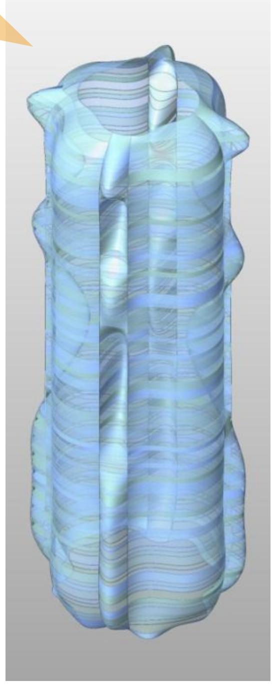
5



Floor Plates

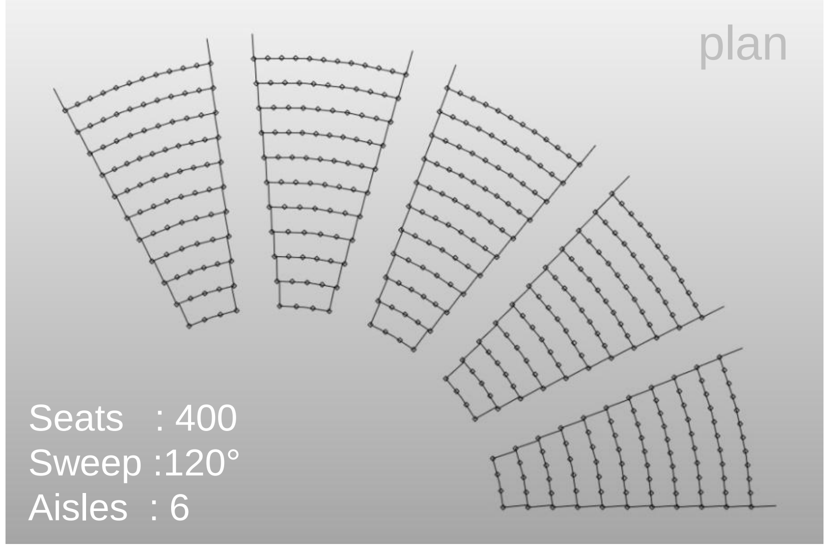


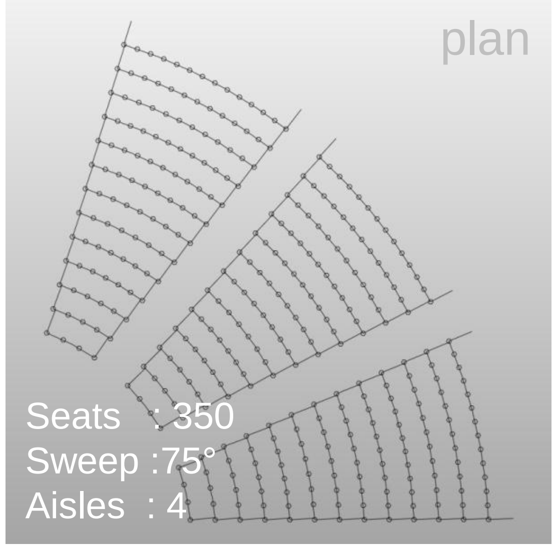
Beams

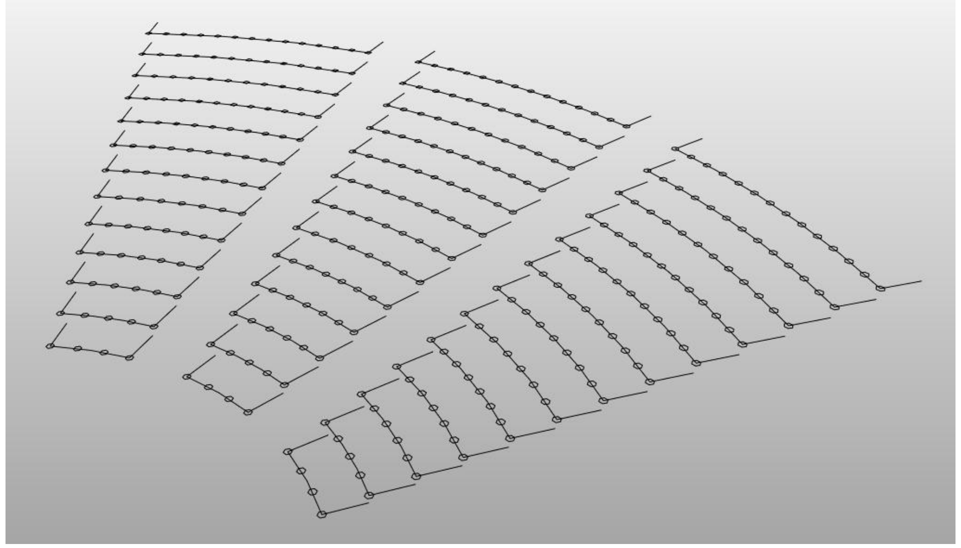


Façade

Amphitheatre



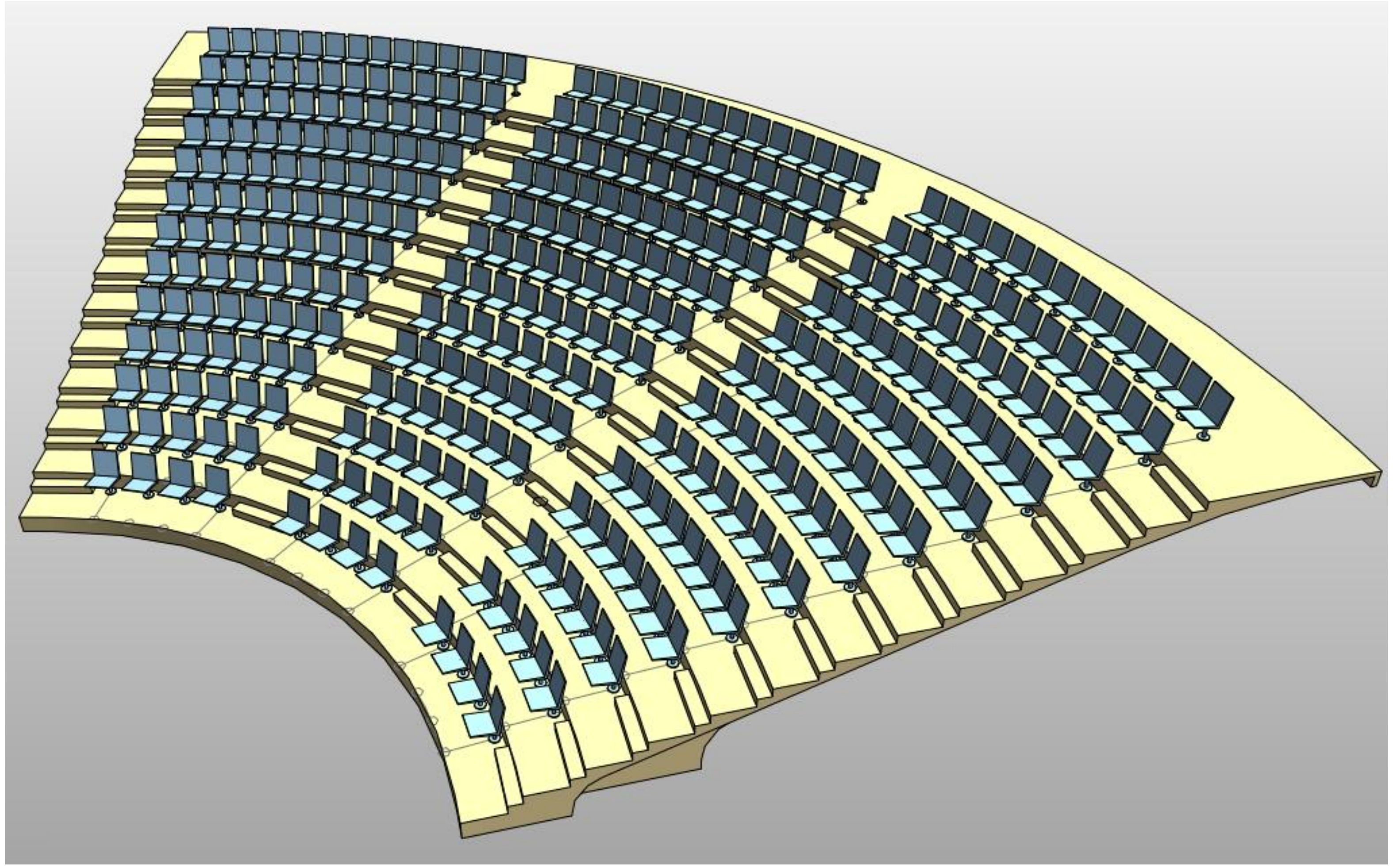




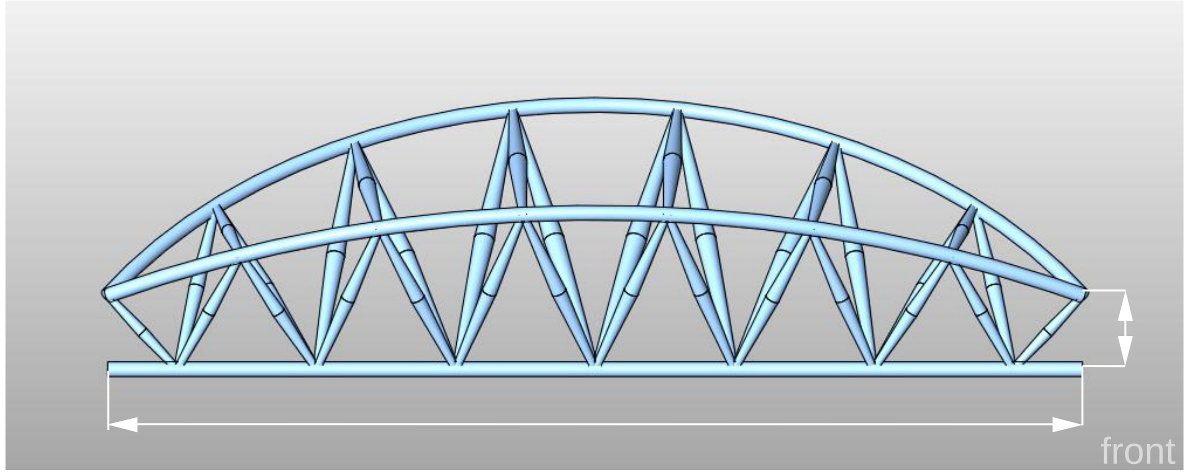
Input Parameters :-

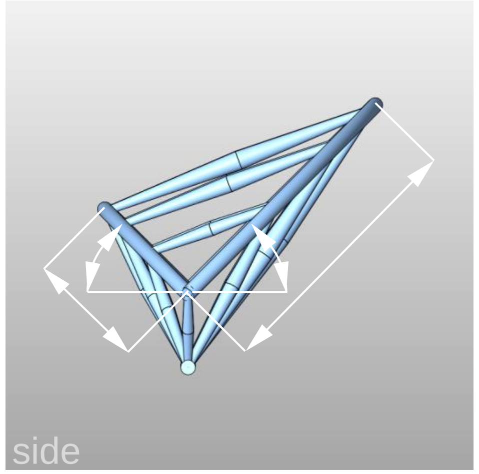
- Required number of Seats
- Seat spacing minimum
- Row spacing
- Sweep angle in plan
- Rake angle
- Number of Aisles –
  - side Aisles included
- Aisle Width
- Start Radius – Front Row

Only the set outs were generated in the API – probably not efficient to code the complete structure – faster to flesh it out manually



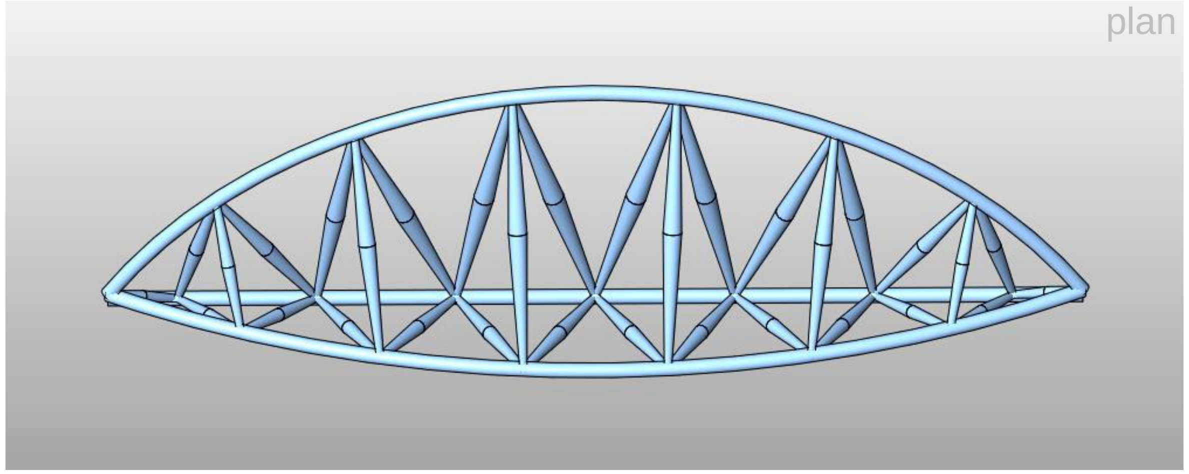
Truss

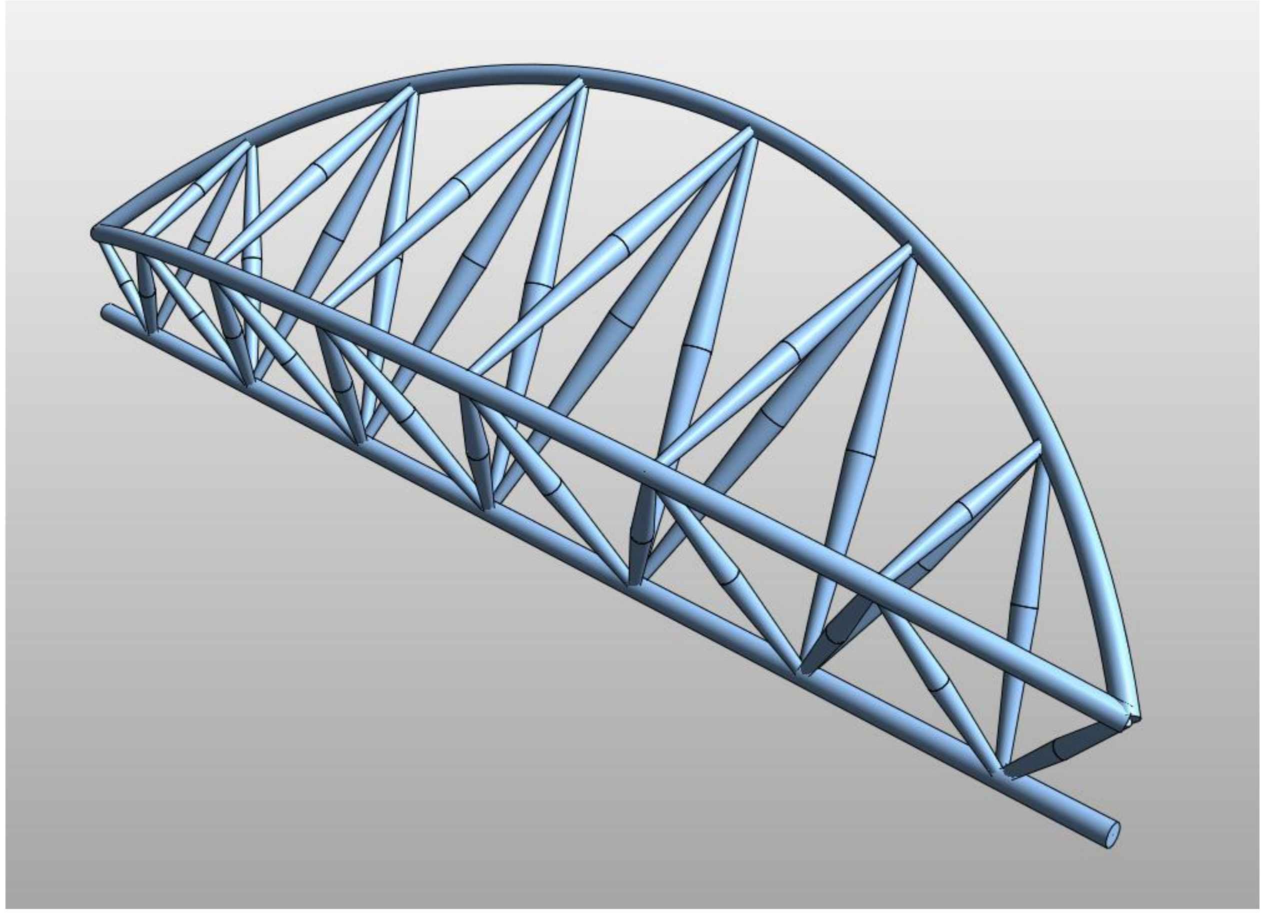




Input Parameters :-

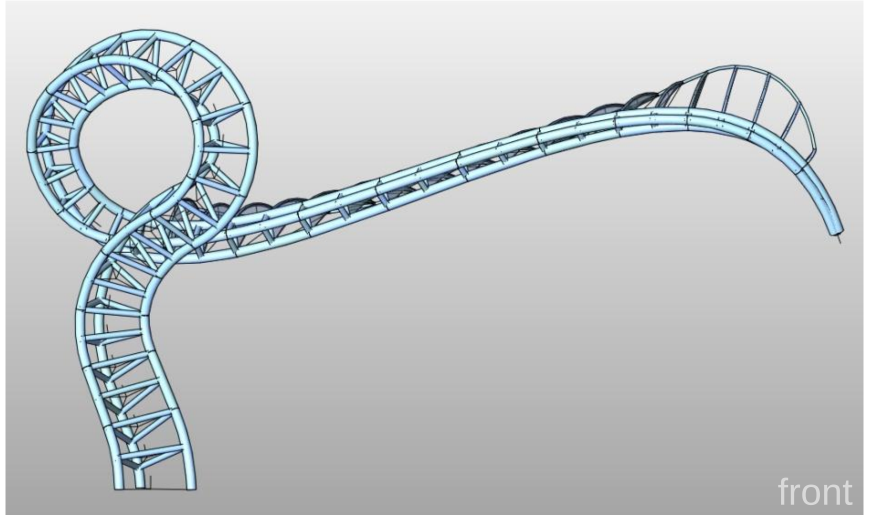
- Span
- Number of Bracing Bays
- Component Radii
- Top Chord offsets at Apex
- Top Chord Angles
- Depth to lower Chord

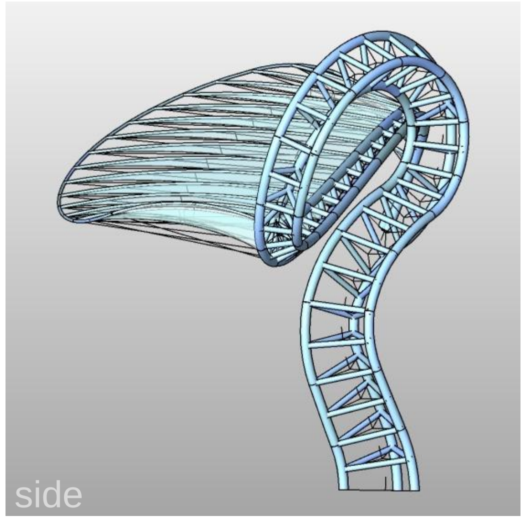




This could easily be done without the API – but adding complexity can often cause conventional methods to 'break'

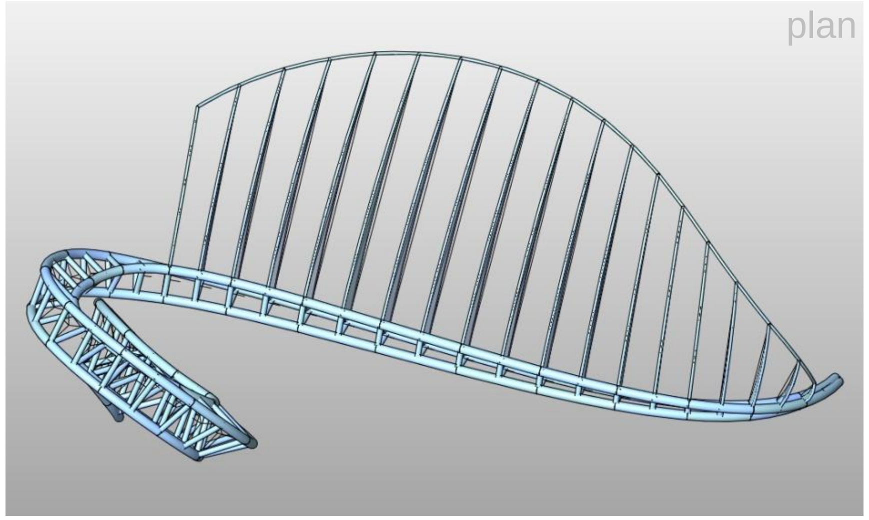
Roller-Coaster Reception

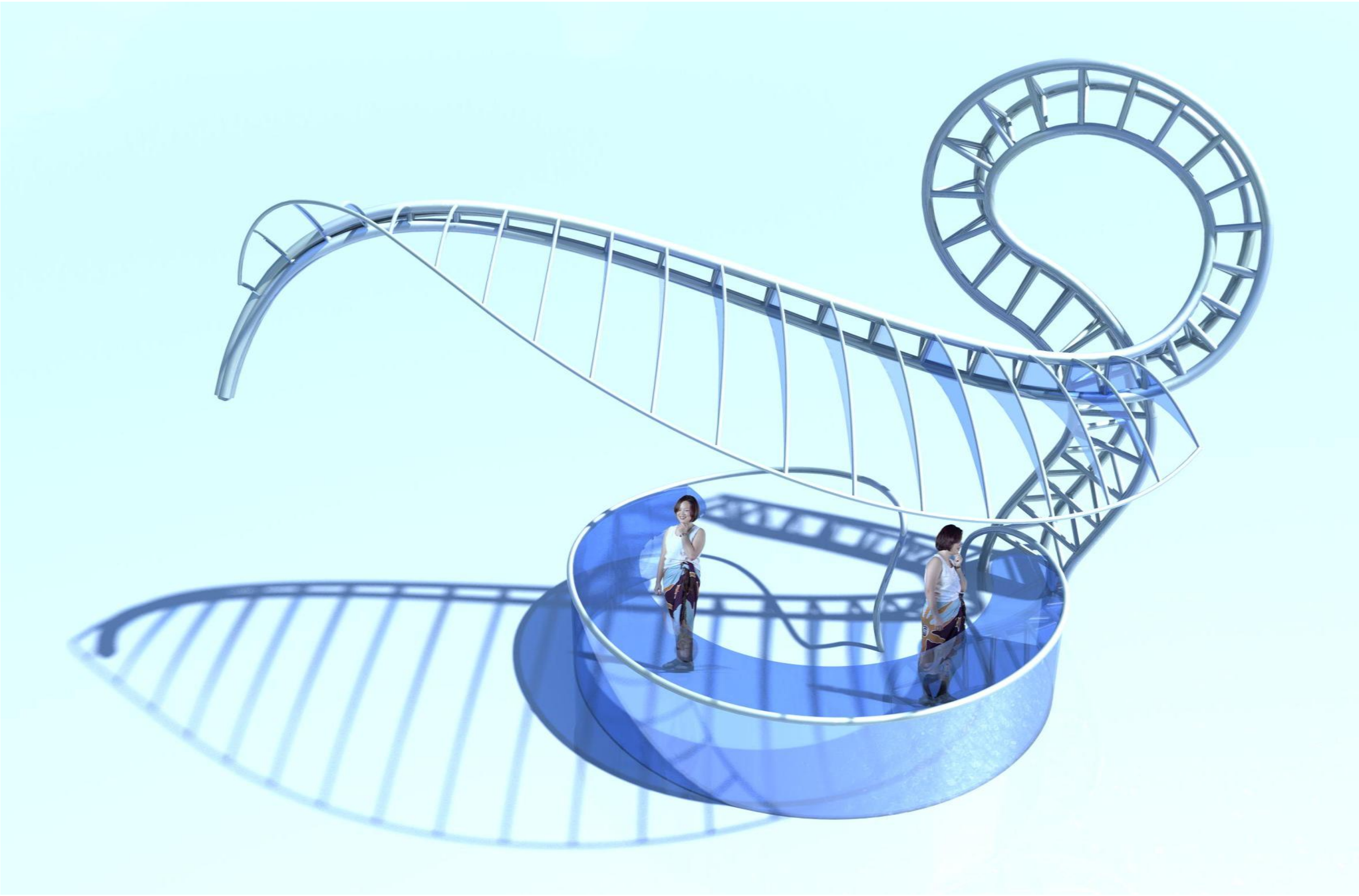




Input Parameters :-

- Spline interpolation points
- Number of Sectors
- Component Radii
- Frame Base Offset from Spline
- Frame Apex Offset from Spline
- Rafter Start Sector
- Rafter End Sector
- Rafter Length

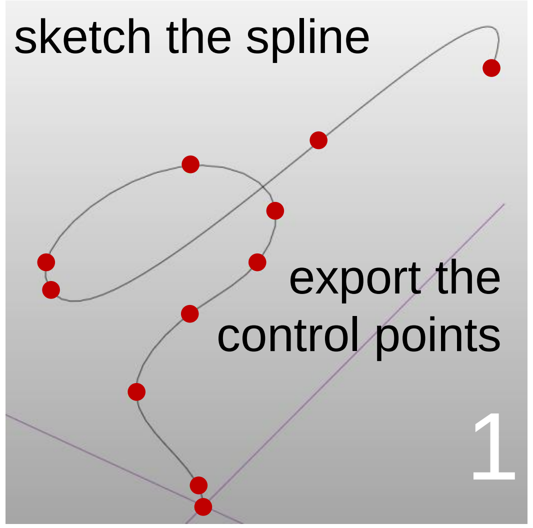




Set out Armature :-

- Single Hermite Spline interpolating control points and created in Revit conceptual mass or external package

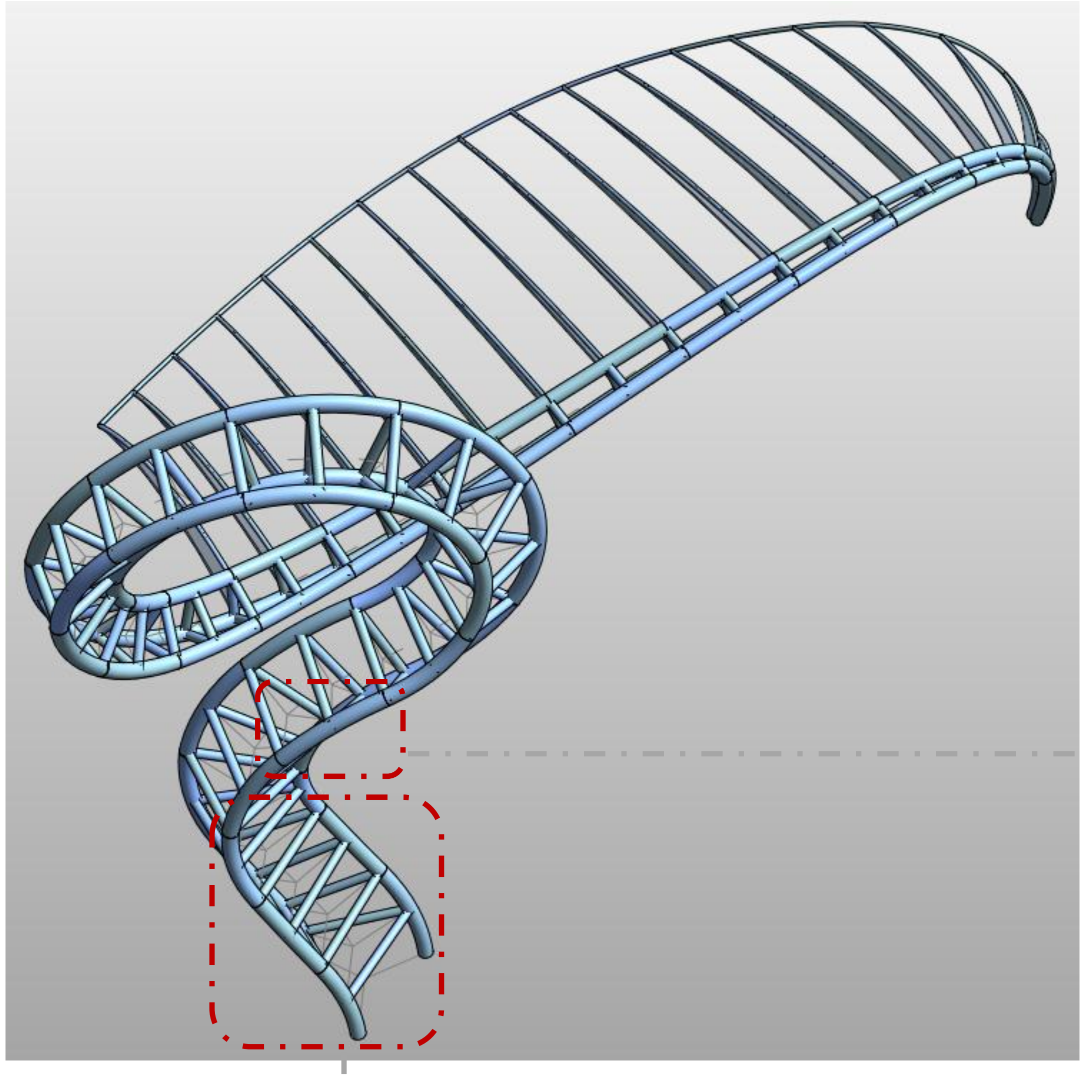
Roller-Coaster Reception : Workflow



sketch the spline

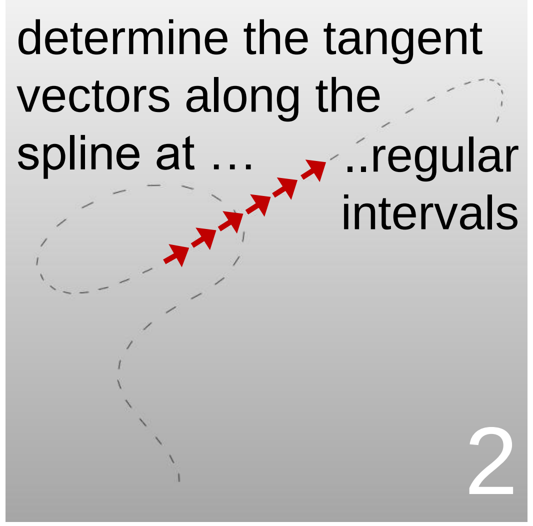
export the control points

1



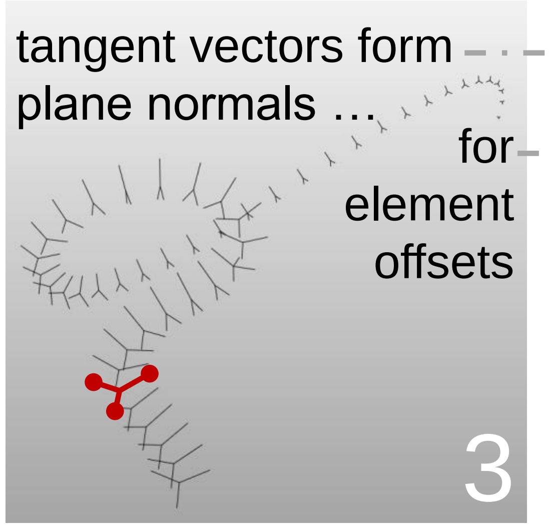
Component rational

- Planar set outs for all elements in order to facilitate fabrication –>
- All curved elements are true arcs
- Number of sectors chosen to ensure *visual* continuity at arc joints –> adjacent arcs not exactly co-tangent



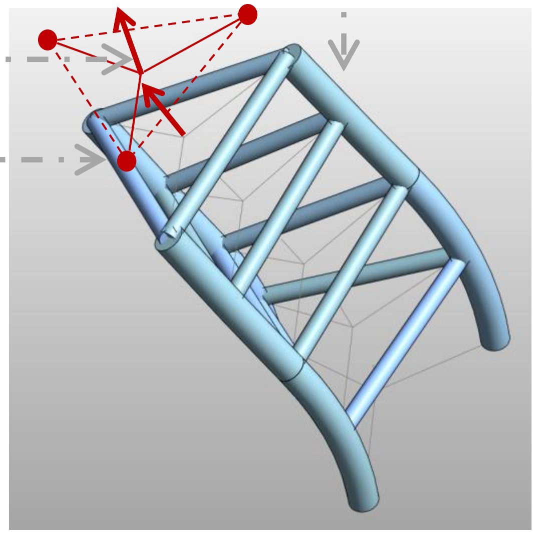
determine the tangent vectors along the spline at … ..regular intervals

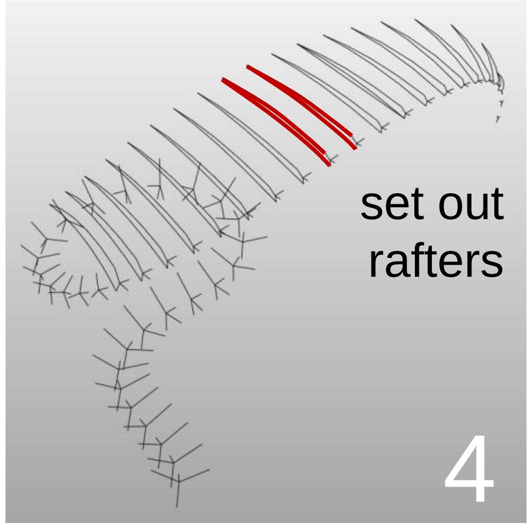
2



tangent vectors form plane normals … for element offsets

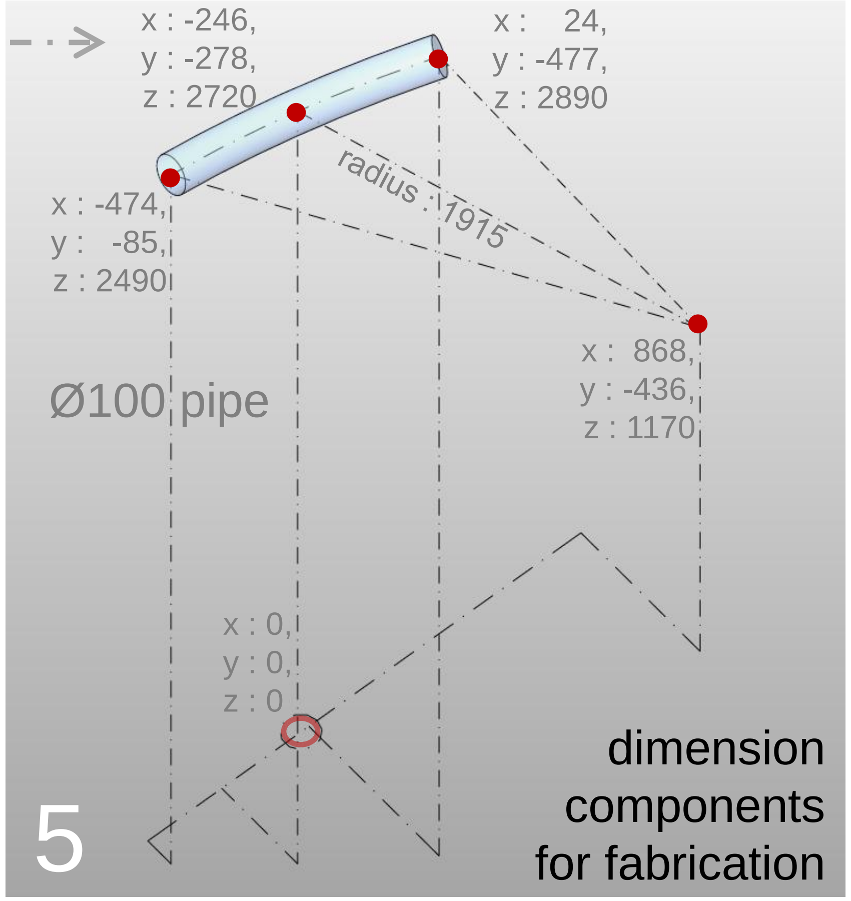
3





set out rafters

4



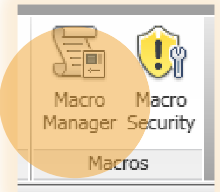
5

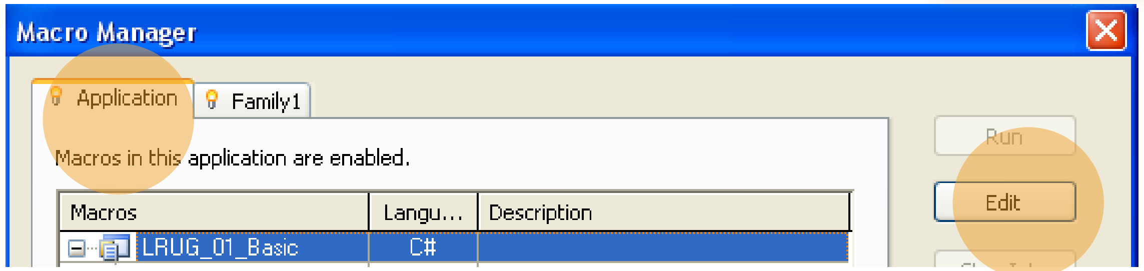
dimension components for fabrication

Appendix : Module and Macro : Quick Start

(‘Starting from Scratch’ on following pages)

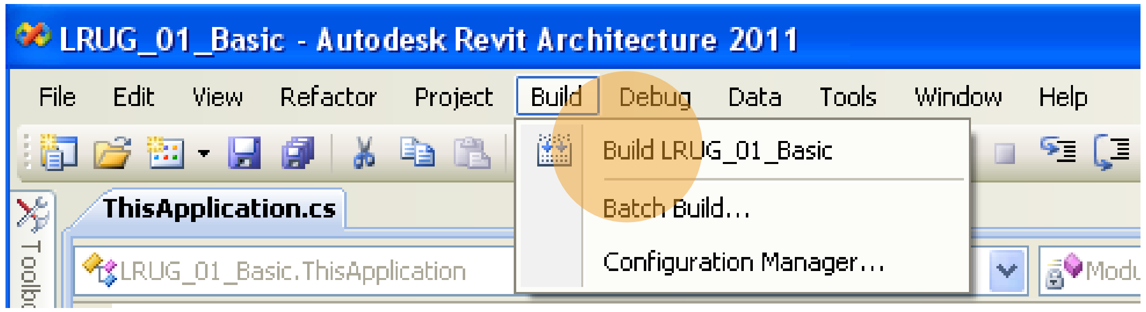
- Download code from LRUG website
- Place ‘LRUG_01_Basic’ folder in
- C:\Program Files\Autodesk\Revit Architecture 2011\Program\VstaMacros\AppHookup\
- Open a new Generic Model family document
- Open the Macro Manager from the ‘Manage’ tab
  - Select ‘Application’ tab
  - ‘Edit’ : LRUG_01_Basic
  - ‘Build’ : LRUG_01_Basic
  - ‘Run’ : Basic_01_Line
    Basic_02_Extrusion
    Basic_03_Extrusion
    Basic_04_Boardwalk
- Play with the code :-
- ‘Edit’ – change some of the variables
  - ‘Build’ again
  - ‘Run’ again

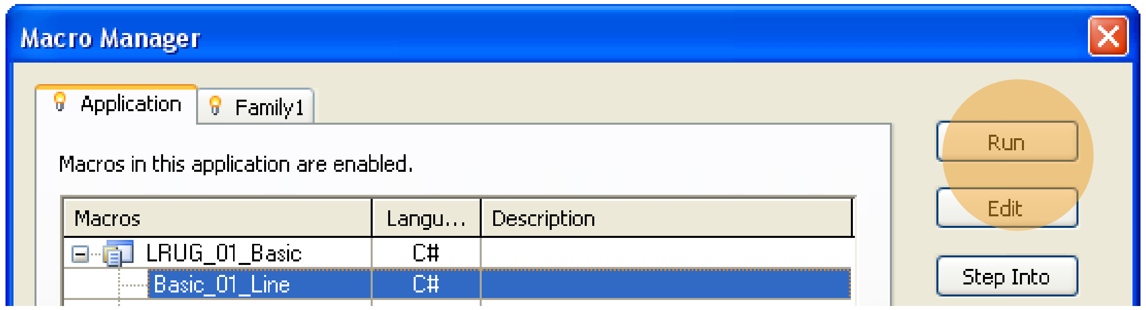




Macros in this application are enabled.

Macros	Langu...	Description
LRUG_01_Basic	C#	





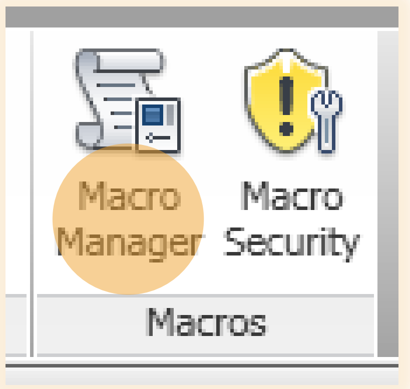
Macros in this application are enabled.

Macros	Langu...	Description
LRUG_01_Basic	C#	
Basic_01_Line	C#	

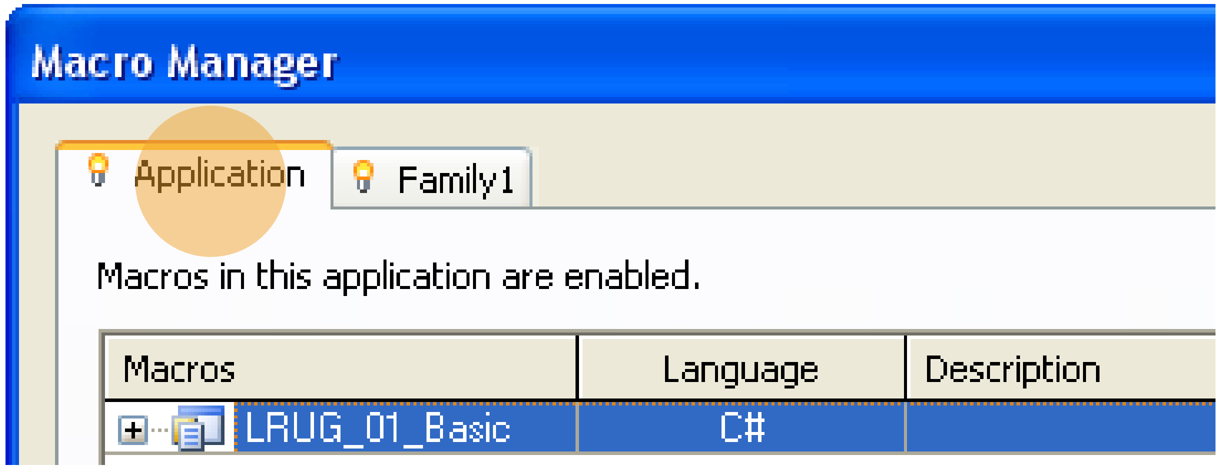
- **NOTE**
  - Code structure is kept to the bare minimum for simplicity, so –>
  - 1 : Little or no error-checking provided
  - 2 : Object-oriented technology is ignored –> users should investigate Classes and encapsulation
  - 3 : Examples highlight re-usability of code –> cut and paste to bootstrap projects

Appendix : Module and Macro in C# - Workflow

- From the ‘Manage’ tab select ‘Macro Manager’ under ‘Macros’



- Select the ‘Application’ tab so that the macros will be visible in newly created documents and …

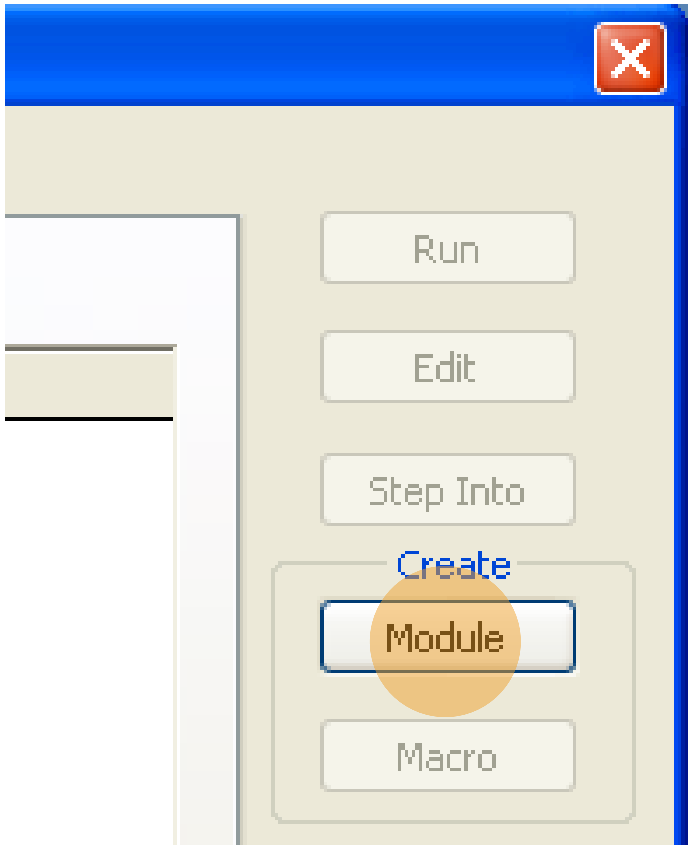


Macros in this application are enabled.

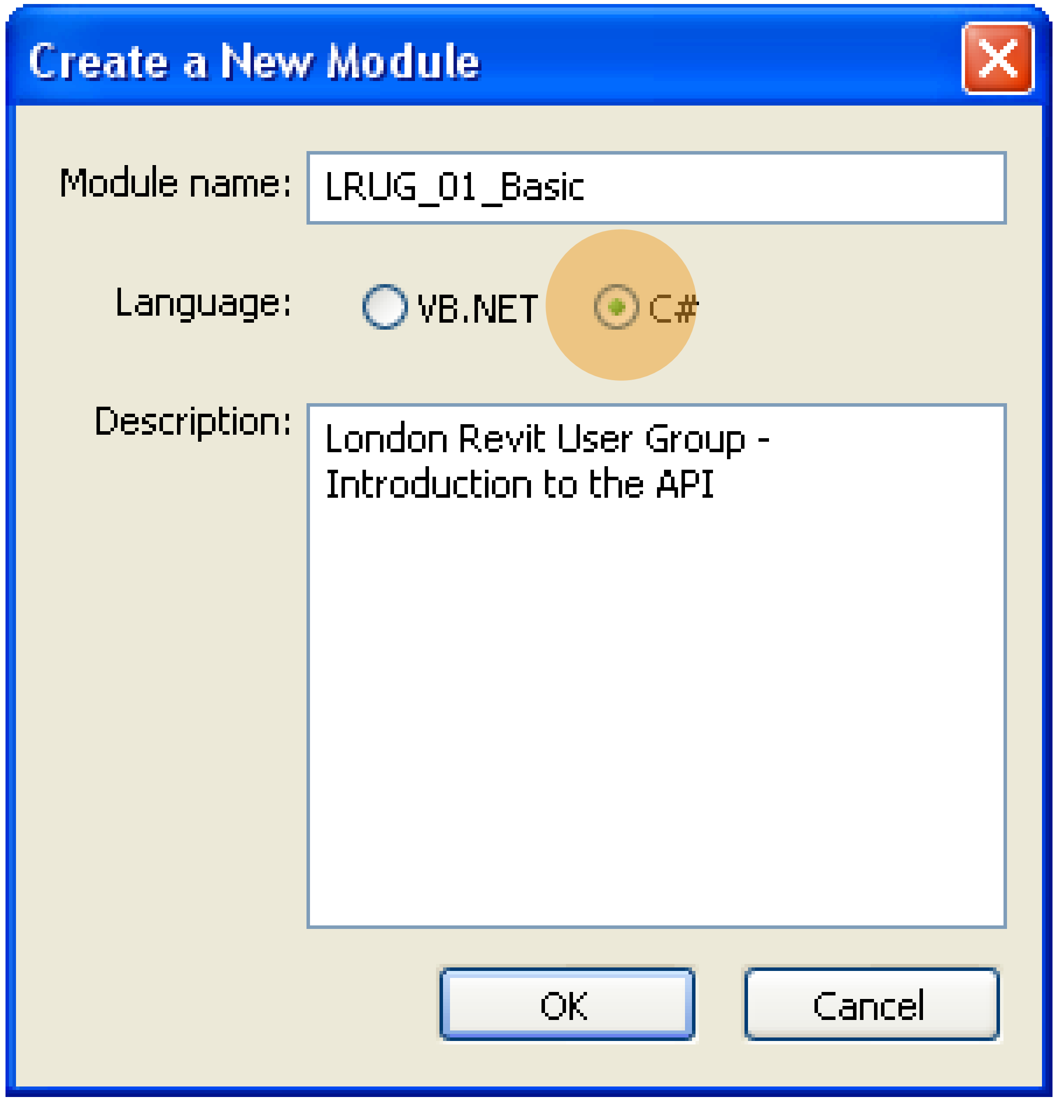
Macros	Language	Description
LRUG_01_Basic	C#	

Appendix : Creating the Module in C#

- 'Create' a new 'Module' in the Macro Manager



- Give the Module a logical name and description and check the C# radio button



Module name: LRUG_01_Basic

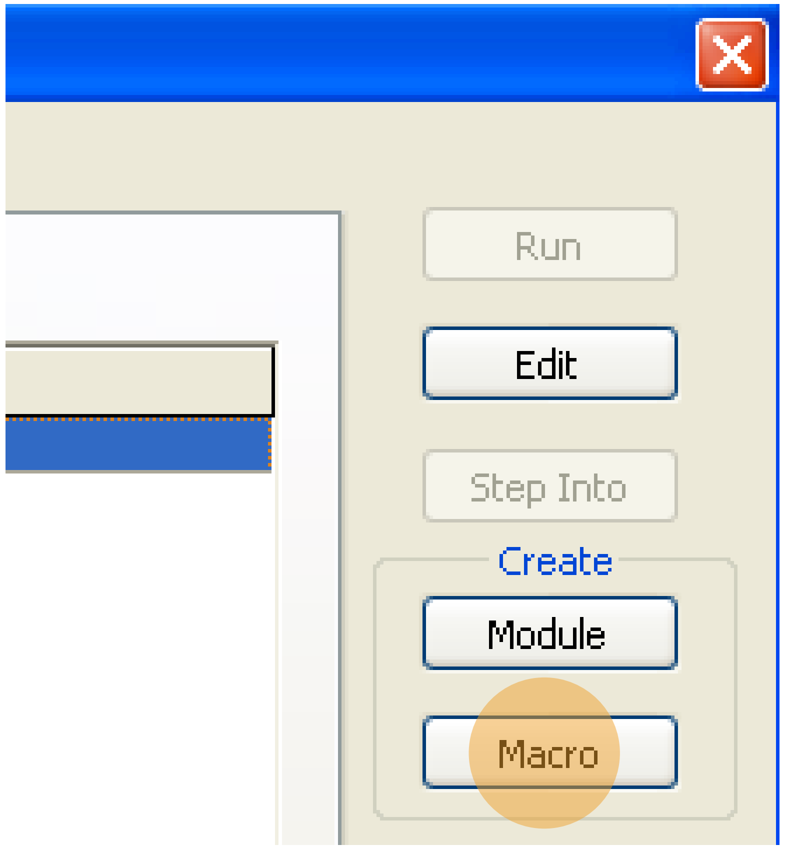
Language: ○ VB.NET ● C#

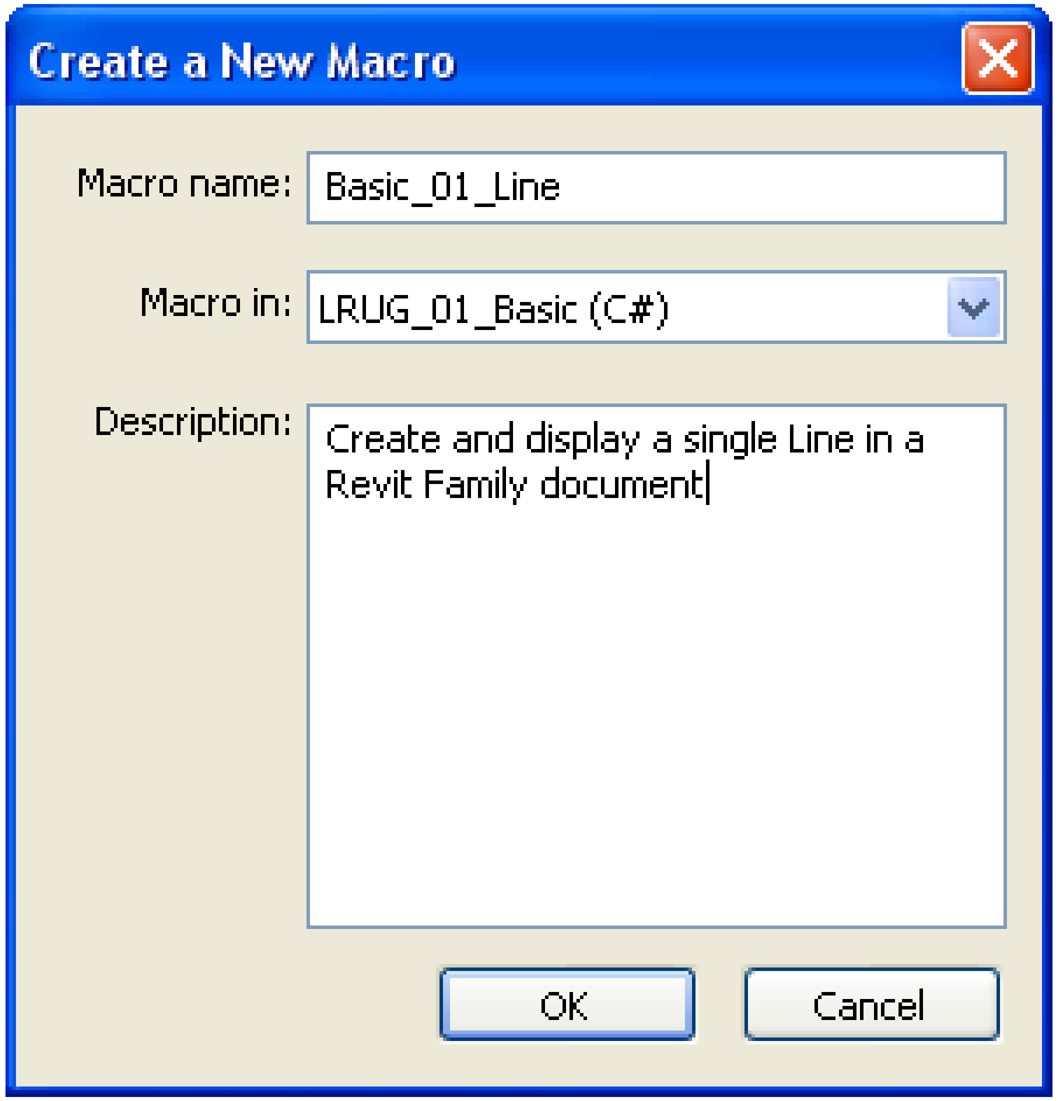
Description: London Revit User Group - Introduction to the API

Appendix : Creating the first Macro in C#

- Create a Macro template within this Module

- Give the Macro a logical name and description





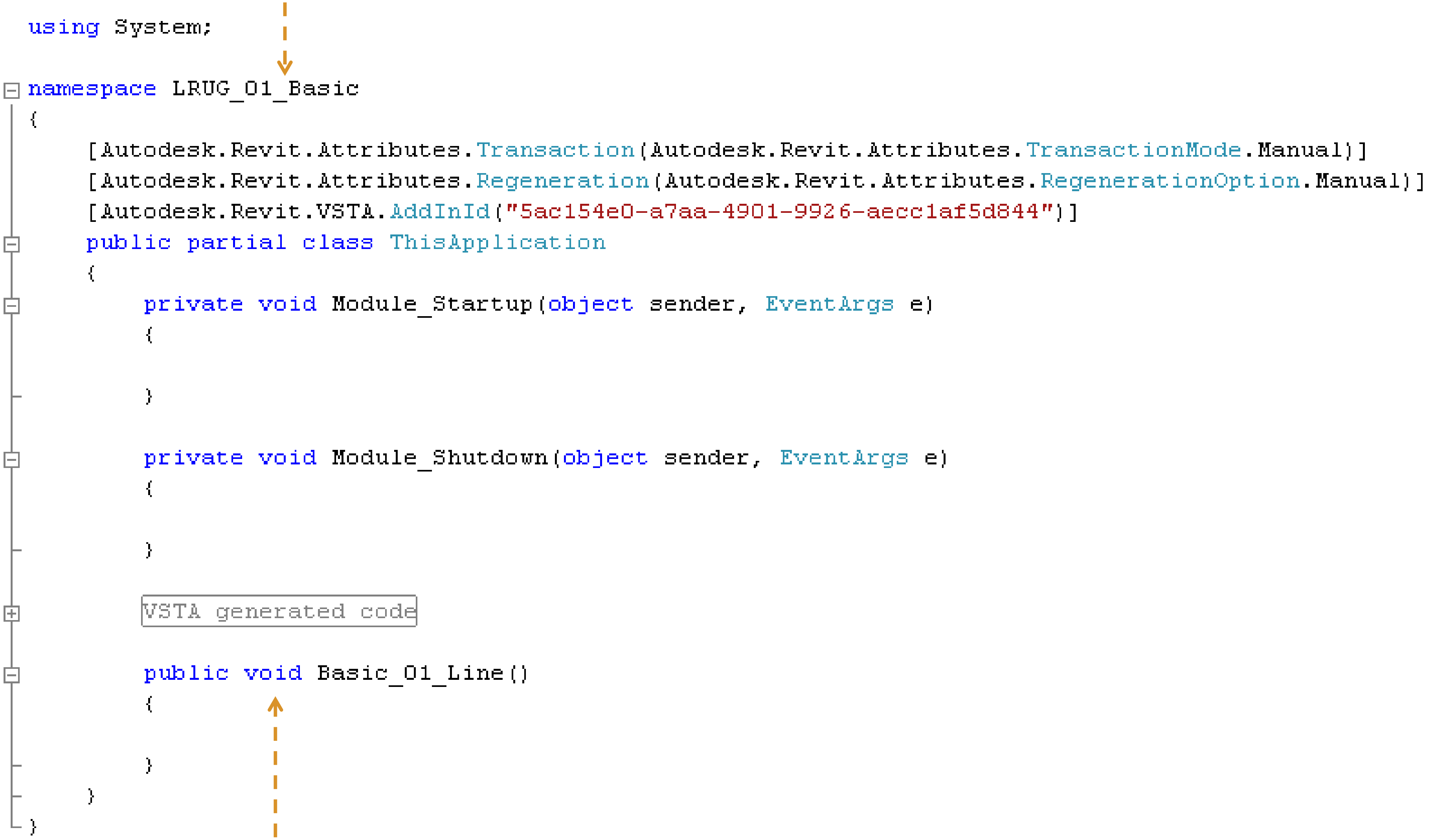
Macro name: Basic_01_Line

Macro in: LRUG_01_Basic (C#)

Description: Create and display a single Line in a Revit Family document

Appendix : Creating the Macro in C#

- The code window opens with its ‘namespace’ given the Module name :-



```
using System;

namespace LRUG_01_Basic
{
    [Autodesk.Revit.Attributes.Transaction(Autodesk.Revit.Attributes.TransactionMode.Manual)]
    [Autodesk.Revit.Attributes.Regeneration(Autodesk.Revit.Attributes.RegenerationOption.Manual)]
    [Autodesk.Revit.VSTA.AddInId("5ac154e0-a7aa-4901-9926-aecc1af5d844")]
    public partial class ThisApplication
    {
        private void Module_Startup(object sender, EventArgs e)
        {

        }

        private void Module_Shutdown(object sender, EventArgs e)
        {

        }

        VSTA generated code

        public void Basic_01_Line()
        {

        }
    }
}
```

- A method with the same name as the Macro has been automatically created

Appendix : Creating the Macro in C#

- At the top under

```
using System;
```

add :-

```
using Autodesk.Revit.DB;
using Autodesk.Revit.UI;
```

to gain access to the Revit functions and to avoid having to prefix each function call with the same

- User code will be inserted in the method corresponding to the Macro name

```
public void Basic_01_Line()
{
 // Your code goes here...
}
```

Appendix : Creating the Macro in C#

- The summary below highlights the basic work-flow – the full definition follows and is the code to be placed in the Macro
- Assume we will be working in millimetres.
- Create a point :-

```
    XYZ point01 = new XYZ();
```

- No values were provided so the systems assumes it's at the origin - `(0.0, 0.0, 0.0)`
- Create a second point – first attempt :-

```
XYZ point02 = new XYZ(0.0, 2100.0, 0.0);
```

- There is a problem – Revit's internal unit of length is Imperial Feet –

  So provide a conversion variable in order to work in Millimetres

```
    double ftMM = 1 / 304.8;
```

- and re-write the point definition, multiplying by the conversion factor :-

```
XYZ point02 = new XYZ(0.0, 2100.0, 0.0) * ftMM;
```

Appendix : Creating the Macro in C#

- Create a Line using the given points :-

```
Line line01 = doc.Application.Create.NewLine(point01, point02, true);
```

- The ‘true’ parameter indicates the Line is ‘bound’ and terminates at the points.
- Establish a Plane for the Line to be drawn on :-

```
Plane plane = doc.Application.Create.NewPlane(XYZ.BasisX, XYZ.BasisY, point01);
```

- The X-Axis, Y-Axis and Origin Point are given.
- Form a Sketch Plane from the Plane :-

```
SketchPlane planeSK = doc.FamilyCreate.NewSketchPlane(plane);
```

- Finally, display the Line :-

```
ModelLine line01F = doc.FamilyCreate.NewModelCurve(line01, planeSF) as ModelLine;
```

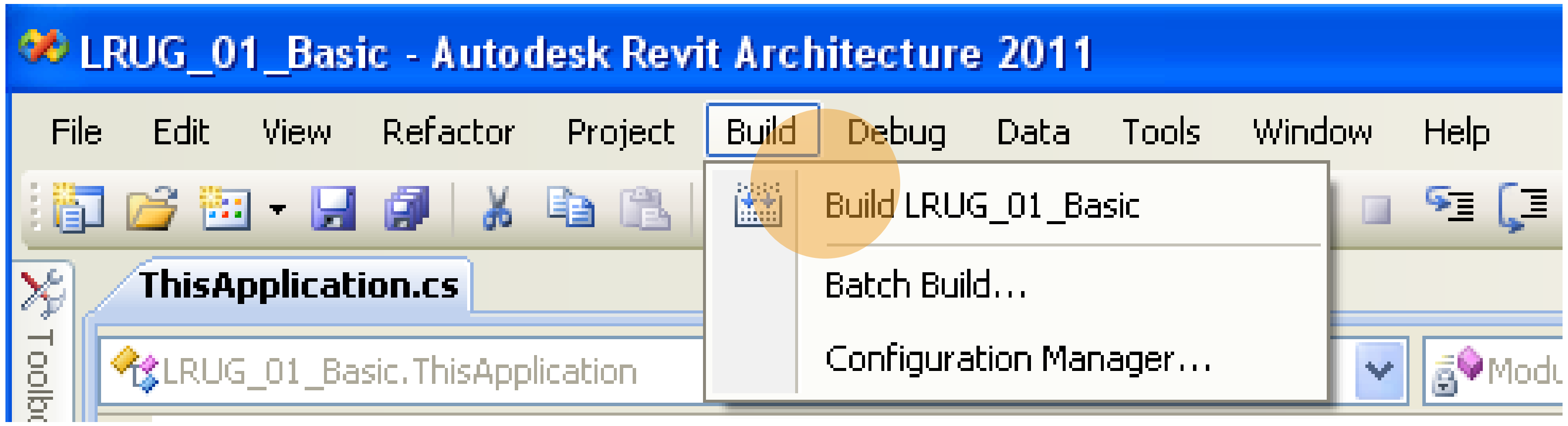
Appendix : Creating the Macro in C#

- This is the full definition – comments, prefixed with //, are not required but aid in development :-

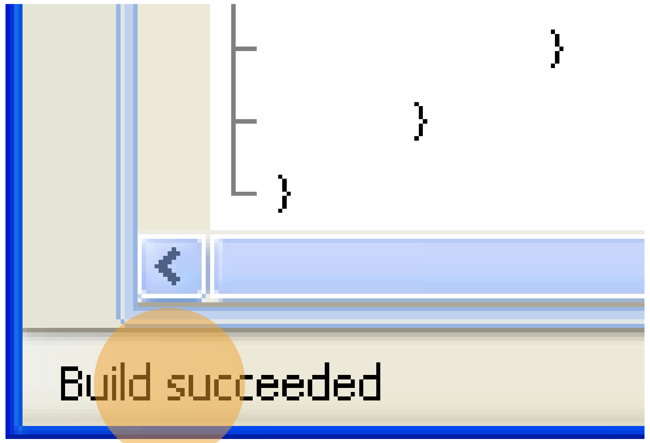
```
public void Basic_01_Line()
{
    // Begin a series of instructions to Revit
    Transaction trans1 = new Transaction(ActiveUIDocument.Document, "Line");
    trans1.Start();

    // Define the document into which the geometry will be placed
    Document doc = ActiveUIDocument.Document;

    // Revit uses Imperial Feet as its internal definition for length -
    // So provide a conversion variable in order to work in Millimetres:-
    double ftMM = 1 / 304.8;

    // Define two points on the Line - multiply by the conversion variable
    XYZ point01 = new XYZ(); // Defaults to (0.0, 0.0, 0.0) if no values are provided
    XYZ point02 = new XYZ(3000.0, 2400.0, 0.0) * ftMM;

    // Create the Line geometry - 'true' indicates it terminates at the setout points
    Line line01 = doc.Application.Create.NewLine(point01, point02, true);

    // Define the Plane for the Line placement - X-Axis, Y-Axis, Origin
    Plane plane = doc.Application.Create.NewPlane(XYZ.BasisX, XYZ.BasisY, point01);

    // Create a Sketch Plane from this Plane in order to display the Line
    SketchPlane planeSF = doc.FamilyCreate.NewSketchPlane(plane);

    // Display the Line
    ModelLine line01F = doc.FamilyCreate.NewModelCurve(line01, planeSF) as ModelLine;

    // End the series of instructions
    trans1.Commit();
}
```

Appendix : Creating the Macro in C#

- Build it : before the Macro can be ‘Run’ it needs to be ‘Built’ in the Editor

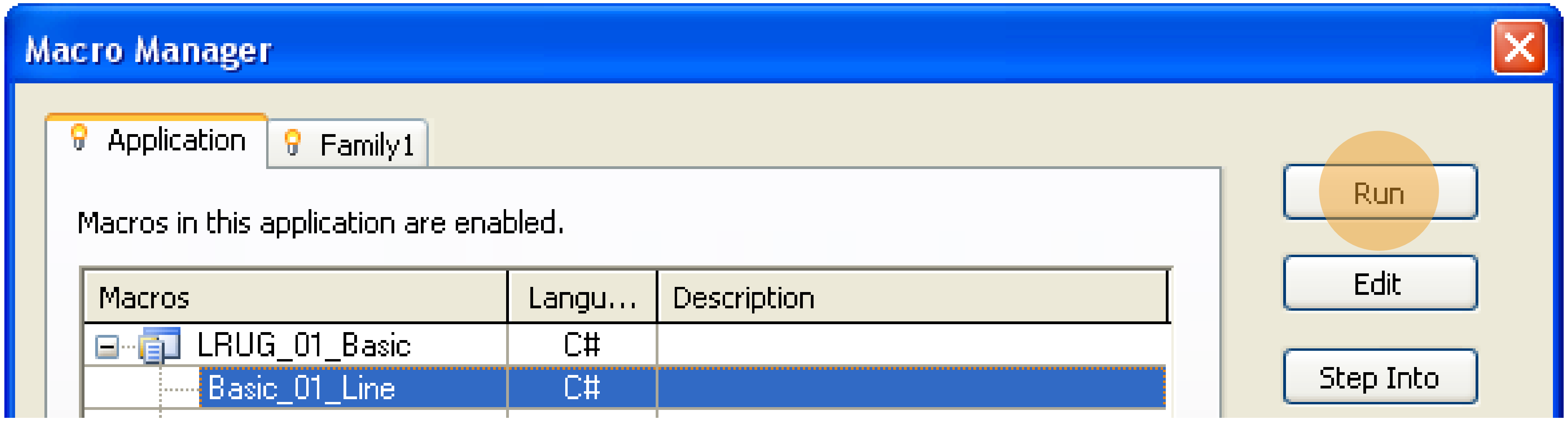


- Hopefully, this message appears at bottom-left



Build succeeded

- Run it : finally, the code produces some output in the Revit document



Macros in this application are enabled.

Macros	Langu…	Description
LRUG_01_Basic	C#	
Basic_01_Line	C#	