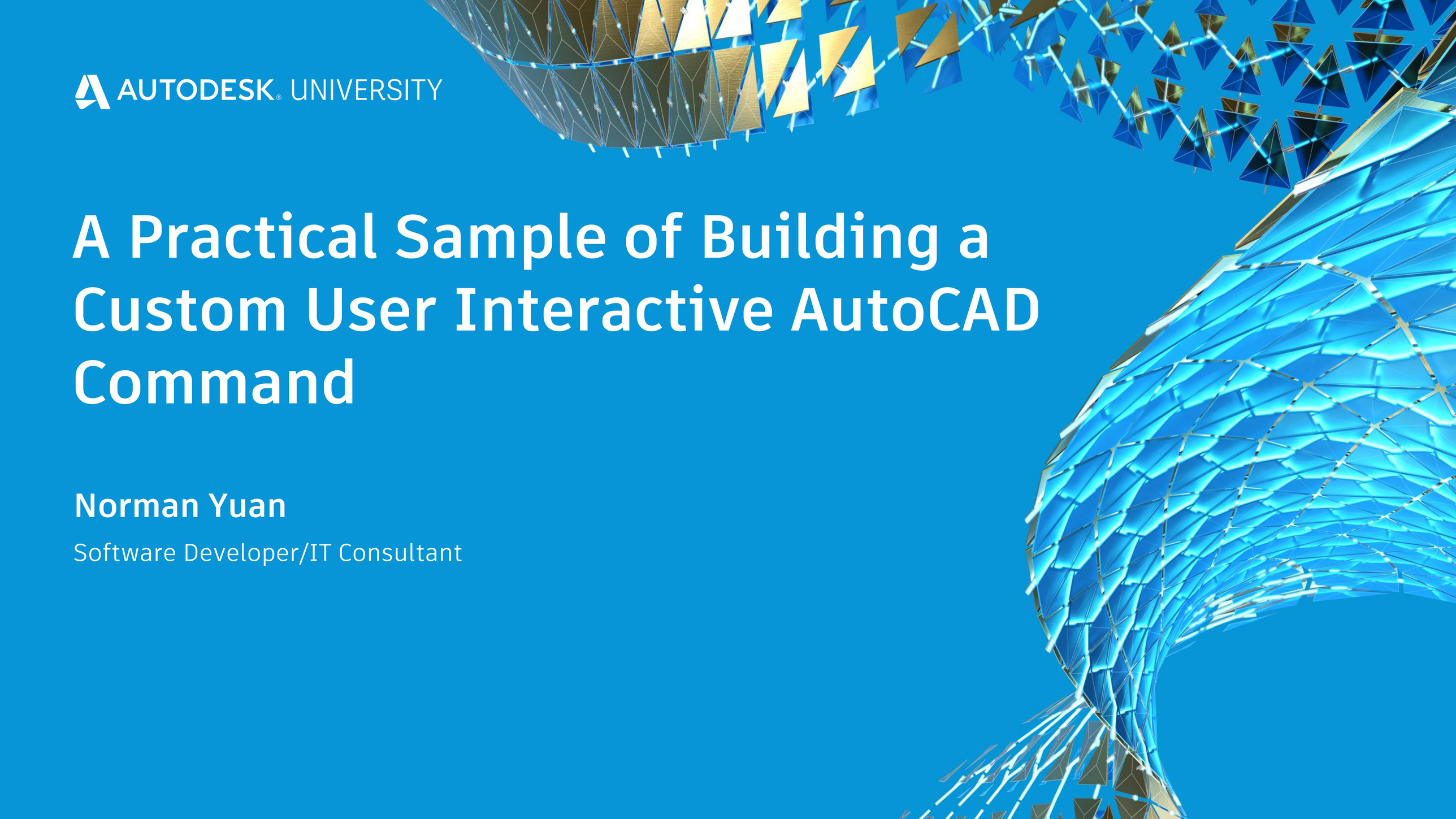
AUTODESK. UNIVERSITY

A Practical Sample of Building a Custom User Interactive AutoCAD Command

Norman Yuan

Software Developer/IT Consultant



About the Speaker

Norman Yuan

Software Developer, Independent IT Consultant

Hobby: Running, seriously

Email: norman.yuan@live.com

Blog: Drive AutoCAD With Code

The Audience of This Class

Who You Are

Someone who does AutoCAD .NET API programming, especially those who have experience of doing LISP/VBA and are interested in .NET API programming or have started transition to .NET API

What Is Your Knowledge/Skill Level

- Basic programming knowledge gained from AutoCAD LISP/VBA work
- Basic AutoCAD .NET API programming, beginner or intermediate level
- Basic general MS .NET programming, again, beginner or intermediate level

The Topic

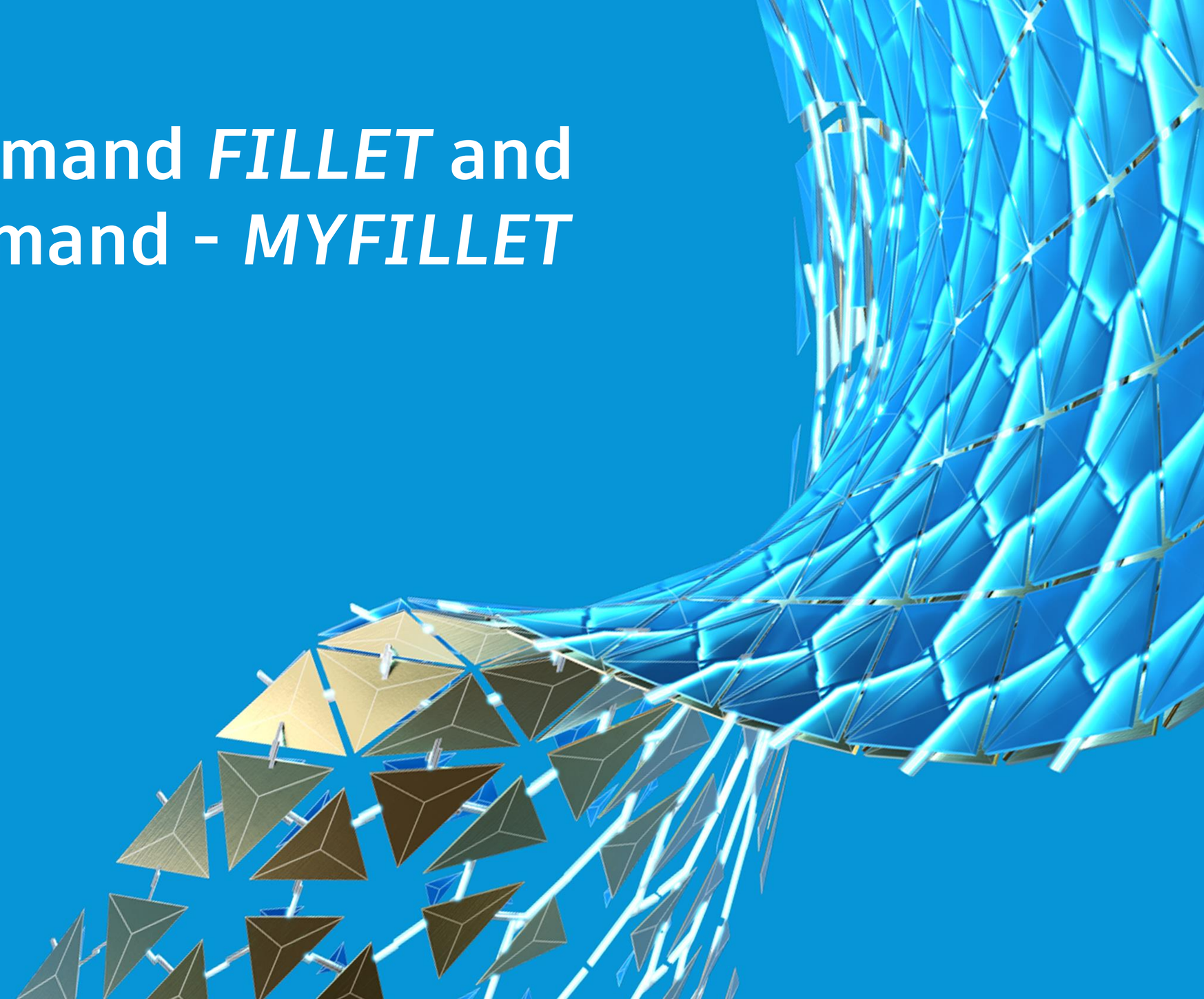
WHY THE TOPIC ON USER-INTERACTIVE CUSTOM COMMAND

Instead of developing a whole set of CAD applications to support entire CAD Office work as a system, many beginner programmers start their endeavor from individual custom commands as immediate enhancement to the CAD operation. I have observed from the readership of my block “Drive AutoCAD with Code” that one of my articles about creating a custom “MOVE” command has highest browsing count month by month over nearly 10 years, since it was published in year 2011, which indicating the interest on building user-interactive command is there.

If you are interested in my article on custom commands, here are a few of them:

https://drive-cad-with-code.blogspot.com/2011/01/creating-move-command-with-net-api.html
https://drive-cad-with-code.blogspot.com/2011/01/creating-move-command-with-net-api-take.html
https://drive-cad-with-code.blogspot.com/2011/01/mimicking-autocads-area-command-with.html
https://drive-cad-with-code.blogspot.com/2011/01/mimicking-autocads-area-command-with_20.html
https://drive-cad-with-code.blogspot.com/2011/01/mimicking-autocads-area-command-with_21.html
https://drive-cad-with-code.blogspot.com/2011/12/creating-my-own-scale-command.html

Built-In Command *FILLET* and Custom Command - *MYFILLET*



AutoCAD Built-in Command *FILLET* Versus Custom Command *MYFILLET*

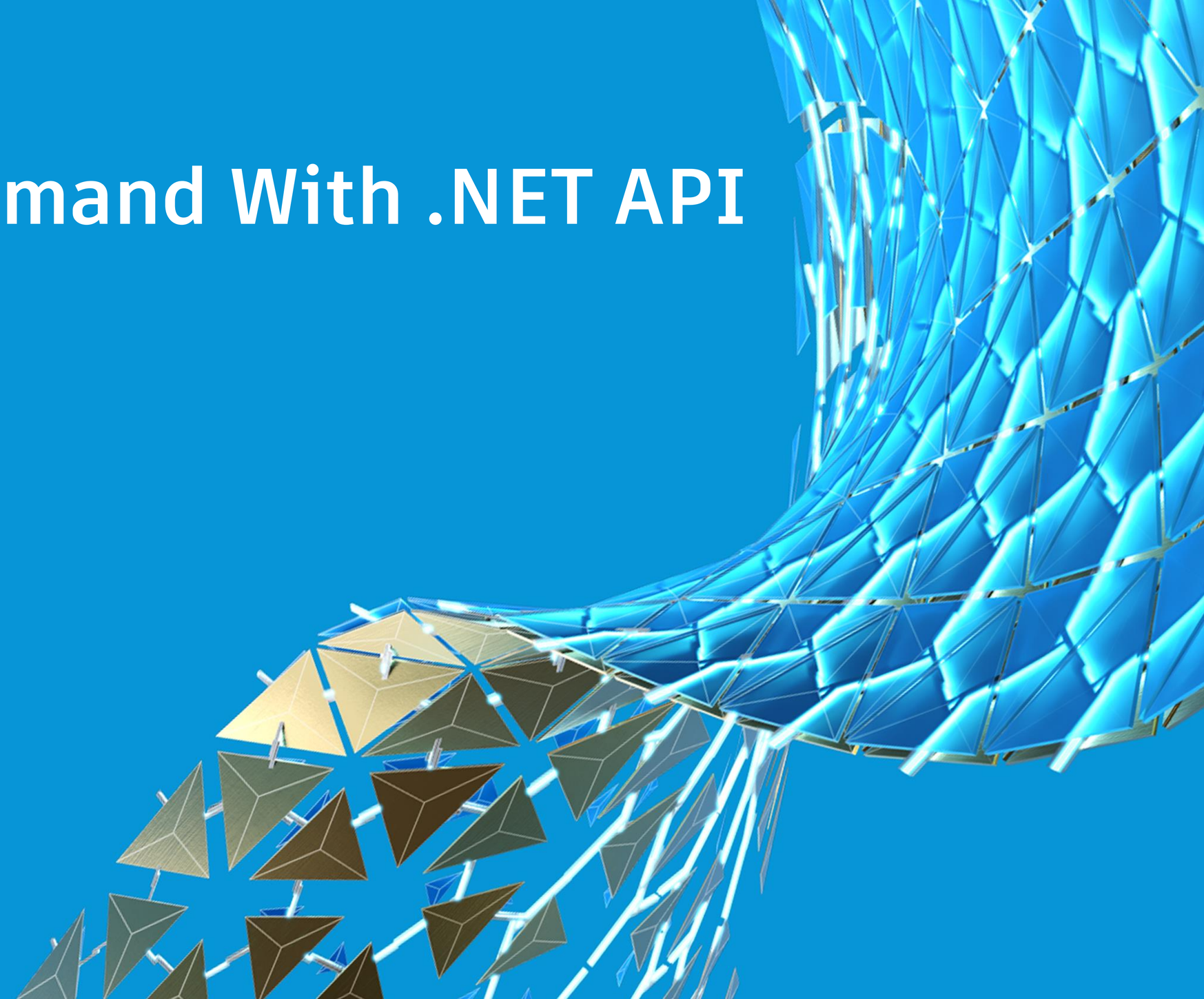
This video clip shows the execution of built-in *FILLET* command:

https://autode.sk/3dllUcq

This video clip shows the custom *FILLET* command:

https://autode.sk/2GYILhV

Custom Command With .NET API



Create Custom Command with AutoCAD .NET API

About CommandMethod

These steps will create a custom command:

- Create a public method in a class;
- This method does not have any argument, nor it returns anything. That is, its signature should look like

  public void MyCommandMethod(){…}
- Decorate this method with Autodesk.AutoCAD.Runtime.CommandMethodAttribute.
- This method can be static or non-static. However, being static or not will cause AutoCAD to instantiate the class instance, in which this method is defined, in different scope: in application context or document context. This is a important concept to know for AutoCAD .NET API programming.

  Note: for more information about command method being static or not, read this article from my block:
  CommandMethod: Static Or Not Static

Create Custom Command with AutoCAD .NET API

About CommandClass

A class can have one or more methods in it being decorated as CommandMethod. The class can also be specified as CommandClass with this line of code somewhere with the DLL project, such as AssemblyInfo.cs, or any code file:

```
[assembly: CommandClass(typeof(ClassNameSpace.ClassName))]
```

The purpose of specifying a class as CommandClass is to optimize how AutoCAD search for custom commands (methods decorated by CommandMethodAttribute) when the assembly is loaded into AutoCAD. This is not mandatory, but good practice to use, especially if the loaded assembly has tons of classes.

Note: there is a very good article to read on this topic:
AutoCAD .NET CommandClass and Some Clarification

Create Custom Command with AutoCAD .NET API

About ExtensionApplication

A class can implement IExtensionApplication interface to have some code run when the assembly is loaded into AutoCAD. Like how CommandClass is specified, this line of code can be in any code file of the DLL project, or in AssemblyInfo.cs:

```
[assembly: ExtensionApplication(typeof(ClassNameSpace.ClassName))]
```

Similarly to CommandClass, it is not mandatory. Differently from CommandClass, each assembly (Dll project) only allow ExtensionApplicationAttribute to be used once. If it is not used, AutoCAD will search all classes for class that implements IExtensionApplication interface. If found, the class' Initialize() method is called when the DLL is loaded. It the assembly has multiple classes implement IExtensionApplication interface and no ExtensionApplicationAttribute is applied, AutoCAD runs the Initialize() method of the class it finds first, and ignore the others.

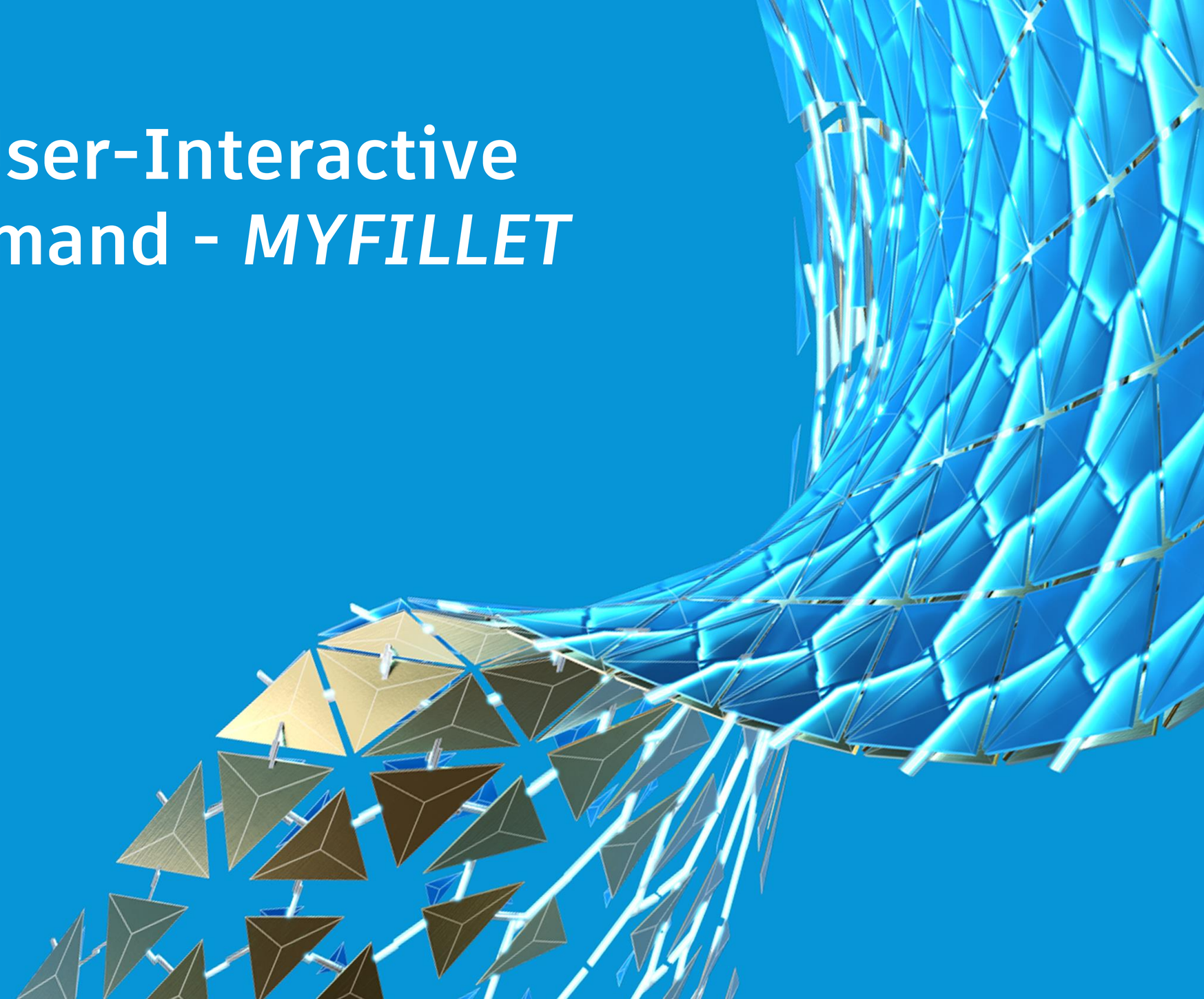
Create Custom Command with AutoCAD .NET API

Organizing Custom Command Code

- Place all code, command classes, helper classes… in the same DLL project. This approach is used by most beginners, and/or when the CAD application development scope is small or quite limited

- Place CommandClass code (i.e. all the custom commands), plus possible ExtensionApplication class code in a DLL project (command project) and the concrete code of CAD business process/logic functionality in other separated projects. The command project sets references to those business process/logic projects. This way, only the command project is to be explicitly loaded into AutoCAD, which makes configuring DLL loading on AutoCAD startup much easier.

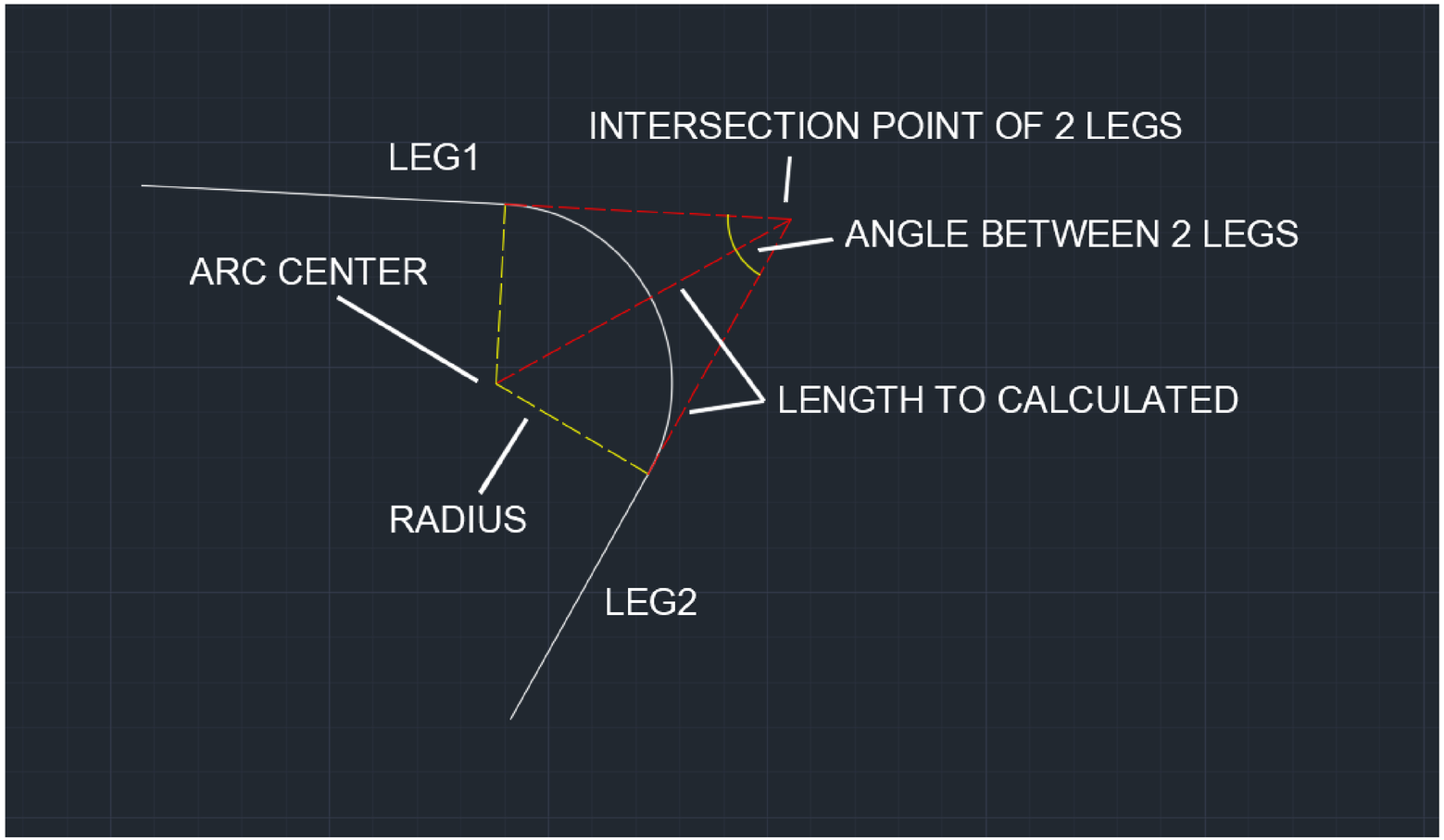
Analysis of User-Interactive Custom Command - *MYFILLET*



Analysis of User-Interactive Custom Command - *MYFILLET*

Information/Data Needed for Command *MYFILLET* to Work

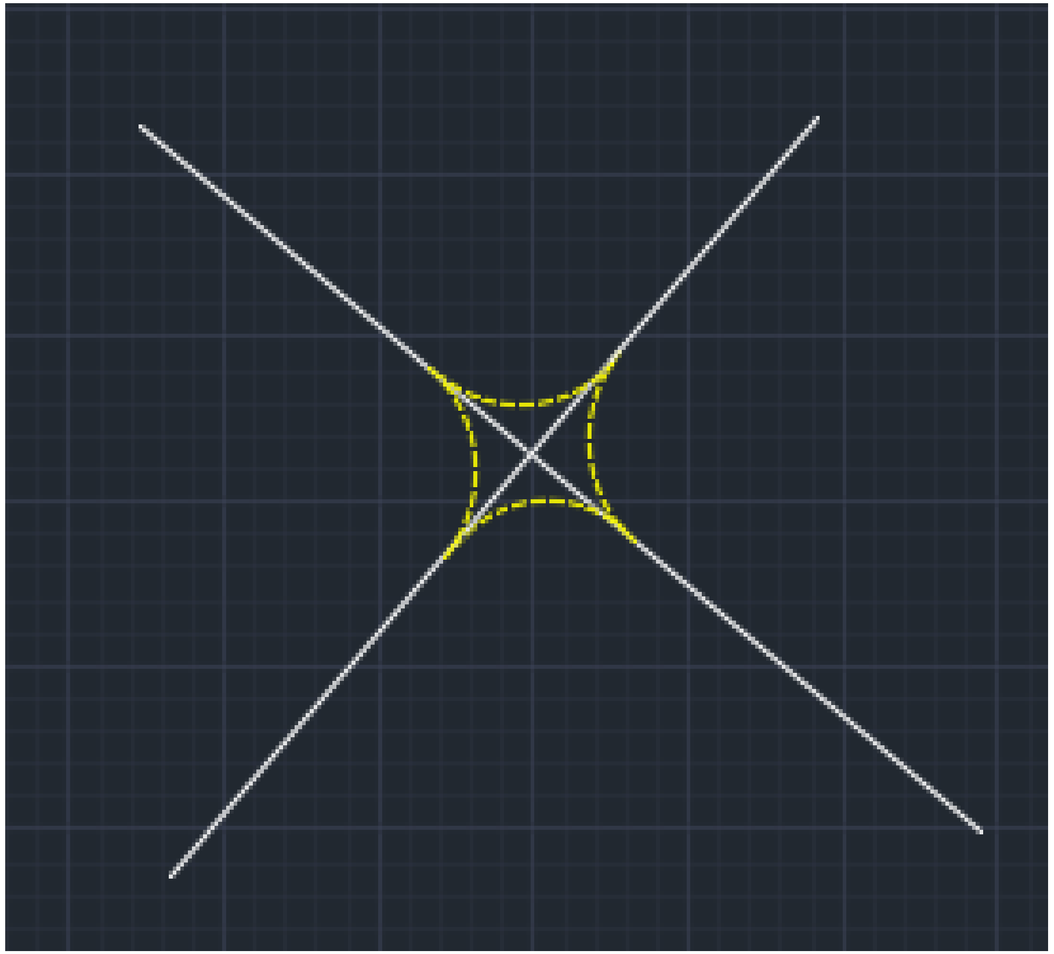
This picture show what information is needed for FILLET operation:



- 2 Legs to be connected with arc. The 2 legs can be 2 Lines, 1 Line and 1 Polyline, 2 Polylines, or 2 segments of the same Polyline

- Which end of the leg segment is to be connected to the fillet arc. That means the point user click while selecting legs is important: the closest end point of the segment should be the end to connect to fillet arc

- Radius of the fillet arc



Analysis of User-Interactive Custom Command - *MYFILLET*

Functionality Scope of the Custom Command

In order to simplify the development work, and/or limit the development work effort, the command’s functionality is scoped with following limits:

- If a leg is Polyline, only straight segment is allowed to be selected as fillet leg
- If 2 legs are 2 different entities, and one of them or both of them is/are Polyline/Polylines, then only the start segment or end segment is allowed to be selected as fillet leg
- If 2 legs are selected from the same Polyline, the 2 legs must be connected segments

Analysis of User-Interactive Custom Command - *MYFILLET*

User Interaction – Collecting User Inputs

From the required information/Data for the command, we know 3 user inputs are needed:

- Selecting first leg
- Selecting second leg
- Entering fillet radius. When doing static fillet, radius can have a default value, so user does not have to enter; once user enters a value for the radius, the command remembers it as default value for next execution; when doing dynamic fillet, the value of radius changes dynamically when mouse cursor is dragged
- Validating user input (see later description of “Workflow of the Custom Command MyFillet”)

Code Development of Custom Command - *MYFILLET*

Code Development of Custom Command - *MYFILLET*

Workflow of the Custom Command - *MyFillet*

The command’s workflow should be:

1. User starts the command;

2. User selects the first leg. Upon selection, necessary validations are performed:

- Selected entity must be either Line or Polyline
- Pick point is remembered for later calculation to determine which end of the leg is to connect to the arc
- If it is Polyline, the pick point is also used to calculate which segment of the polyline is selected

3. User select the second leg. Upon the selection, following validations/calculations are performed

- Selected entity must be either Line or Polyline
- If one of the legs is Polyline, the selected segment of the Polyline must be either start or end segment
- If the 2 legs are from the same Polyline, the 2 selected segment must be connected
- The selected segment from a Polyline must be straight segment

4. Temporary entities as TransientGraphics Drawables (ghost images) are generated, using default arc radius

5. The visual hints of the legs and arc are drawn as temporary graphics as static fillet

6. User is given option to accept the fillet, or change the arc’s radius, or draw dynamic fillet graphics;

7. If user chooses option “Apply”, the command is done.

8. If user chooses option of “Radius”, user enters new radius and back to step 4;

9. If user chooses to show dynamic fillet, the temporary graphics change dynamically according to the mouse cursor’s move;

10. In dynamic fillet mode, user can choose to go back to static fillet (step 4), or select the radius by clicking moved mouse cursor, the dynamic radius is applied;

11. The command is completed by permanently drawing the fillet arc, and possibly join the 2 legs into a single Polyline, if one of the leg is a Polyline.

Code Development of Custom Command - *MYFILLET*

Simplifying Required Calculation

Once user has selected 2 legs, in the case of static fillet, the code needs to calculate the fillet arc's center according to the arc's radius; for dynamic fillet, the code needs to calculate the arc's radius while the arc's center is moved along the center line of the intersecting angle of the 2 legs.

Following rules are set in order to simplify the calculation:

1. When counting angle, 0° is set at the direction of east, and is counted counter-clockwise;
2. The 2 legs are presented visually as 2 Lines, started at their intersection point;
3. The intersecting angle of the 2 legs always starts from leg1 and rotates counter-clockwise to leg2.

Code Development of Custom Command - *MYFILLET*

Implementing Code

The code for this fillet operation is divided into 3 portions to provide different functions in the process:

1. The code to guide user to select 2 fillet legs. Upon the leg selection, the code will do necessary validation to make sure the leg selection is valid
2. The code to generate temporary graphics based on selected legs and radius as visual hint of the fillet legs and arc
3. The code to complete the fillet operation, that is to draw fillet arc, update selected leg entities that connect to the arc; when necessary, join the leg entities and fillet arc into a single Polyline

Naturally, 3 classes are used for 3 functions in the fillet operation: *FilletTargetSelector, FilletGhosts, FilletTool*.

Code Development of Custom Command - *MYFILLET*

Class *FilletTargetSelector*

This class guides user to pick the 2 legs. It will validate the picked entities. Besides the very basic requirement of the legs being either Line or Polyline, the other validation rules occur when the second leg is selected, including:

- If the 2 legs are in parallel, exception will be raised and will be caught in calling class (*FilletTool*);
- If the leg is polyline, the picked segment must be straight one;
- If one leg is a polyline, or both legs are polylines, the picked point must be on the start or end segment of the polyline;
- If the 2 legs on the same polyline, the 2 legs must be connected to each other.

Code Development of Custom Command - *MYFILLET*

Class *FilletTagetSelector* (continued)

After 2 valid legs are selected, an instance of *FilletGhosts* class is generated. Because the main purpose of this class is to generate required data for actually doing the fillet work, and the data involves creating non-database-residing entities (ghost graphics), it implements *IDisposable* interface, so its instance can be wrapped in *using(…){…}* block to make it easier to dispose those ghost entities.

This class also creates an instance of class *AngleSettings*, which saves current drawing's "ANGBASE" and "ANGDIR" system variables and then set them to required value, described previously (0˚ starts in X direction and angle is counted counter-clockwise). The saved angle settings are restored back when this class is disposed (at the end of *using(…){…}* block).

Code Development of Custom Command - *MYFILLET*

Class *FilletGhosts*

This class contains 4 non-database-residing drawables (2 Lines as legs, 1 Arc and 1 Line as center line dividing the intersecting angle of the 2 legs, on which the arc's center point locates). These drawables are used to generate TransientGraphics to show temporary graphics of the fillet arc/legs/center line for user to preview the command's outcome. Obviously, as long as the 2 legs are selected properly, the fillet arc's center point will be located on the calculated center line according to the radius of the arc. Because our custom Fillet command expect to allow user to drag mouse to dynamically change the radius of fillet arc during preview, this class also handles Editor.PointMonitor event to draw the changed fillet arc during preview. Finally, since the temporary ghost drawables are non-database-residing, this class implements *IDisposable* interface, to make disposing them easier by calling its Dispose() method.

Code Development of Custom Command - *MYFILLET*

Class *FilletTool*

This class does the actual work:

- Create instance of *FilletTargetSelector* to let user select fillet legs;
- Draw fillet arc and update the 2 legs that connect to the arc;

This class is used in the separated custom command aggregation project.

Class Summary

This class demonstrated a moderately complicated custom command MYFILLET to show a typical way to develop custom commands as AutoCAD application. We discussed:

- How to create a custom command with AutoCAD .NET API
- How to analyze the workflow of the execution of custom command
- How to collect data inputs through user interaction
- How to organize the data and the action upon it into different classes

Closing

Resources:

https://drive-cad-with-code.blogspot.com/

https://adndevblog.typepad.com/autocad/

https://through-the-interface.typepad.com/

Contact:

Email: norman.yuan@live.com

Blog: Drive AutoCAD with Code