GRASSHOPPER for BEGINNERS

A Quickstart Guide to Parametric Design

by Thomas Jeremy Tait

2025 Edition

hopific

Introduction

Grasshopper has been a game-changer in the design world. In fact, all leading architecture offices rely on it for top-level designs. It has saved my colleagues and me so much time, I can't imagine life without it now.
Need to create a stunning facade pattern? Not a problem! Do you want to evaluate a design for both sustainability and efficiency? Consider it done. All thanks to the flexibility of parametric design.

Gone are the days of late-night modeling and repetitive commands; with Grasshopper, you can build a parametric model in a fraction of the time. While Grasshopper won't create designs by itself (your unique ideas are, and always will be, at the heart of any project), it does equip designers to rapidly transform their vision into tangible and adaptable designs.

But where should you start? From outside, Grasshopper looks like a complex machinery, which requires some very specific knowledge, and figuring it out on your own won't help much here. Just like any other tool, Grasshopper has its own learning curve. That's why I decided to create this e-book for Grasshopper beginners. It's your first step toward unlocking the potential of parametric design and gaining greater control over your architectural creations.

I hope that you will find this e-book useful and if you need any help in that journey, I'm just an email away at thomas@hopific.com.

Thomas Jeremy Tait, Founder of hopific.com

Table of Contents

1. Introduction to Visual Programming

Understanding Grasshopper: More Than Just Software

Before we open Grasshopper and build our first script, it's essential to understand the foundational concepts Grasshopper is built on.

Due to its well-designed interface, its easy to mistake Grasshopper for just another software. But Grasshopper is not just a program, it's a visual programming environment.

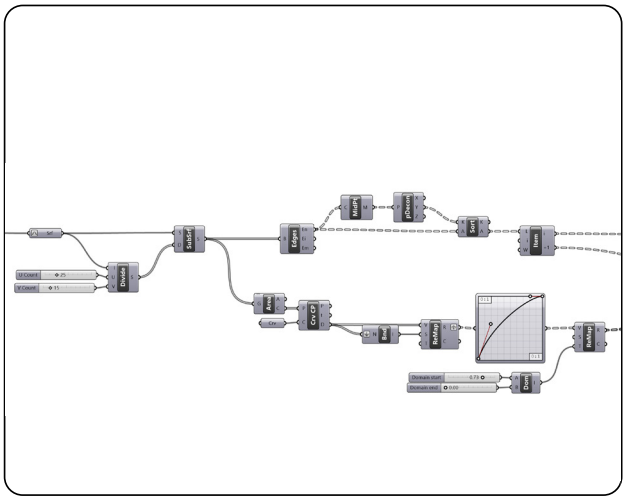
Compared to conventional programming, visual programming allows us to create programs by manipulating program elements graphically rather than by specifying them with code.

But the end result is the same. By connecting and creating sequences of components, we are building a program or algorithm with graphical building blocks. In doing so, Grasshopper allows us to combine Rhino's powerful modelling commands with the scalability and control of programming techniques.

Programming

```
def ChangeBuildingProperties():

    filter = Rhino.DocObjects.ObjectType.Hatch
    rc, objrefs = Rhino.Input.RhinoGet.GetMultipleObjects("Select Hatches:", False, filter)
    if rc != Rhino.Commands.Result.Success: return

    sc.doc.Objects.UnselectAll()
    sc.doc.Views.Redraw()

    hatches = []
    objects = []
    for objref in objrefs:
        hatch = objref.Hatch()
        object = objref.Object()
        if hatch:
            hatches.append(hatch)
            objects.append(object)

    print "Number of hatches :", len(hatches)

    for i in range(len(hatches)):

        hatchCrv = geHatchCrvs(hatches[i])

        print "length of hatchCrv: ", len(hatchCrv)

        if len(hatchCrv) == 2:
            print "Closed block"

            #show which object is selected
            #print "ID: ", objects[i].Id
            rs.SelectObjects(objects[i].Id)

            #copy the object attributes
            attributesTmp = objects[i].Attributes.Duplicate()

            #pick longer curve
            curvesLength = []
```

Visual Programming



Grasshopper: The Perfect Blend of Rhino and Programming

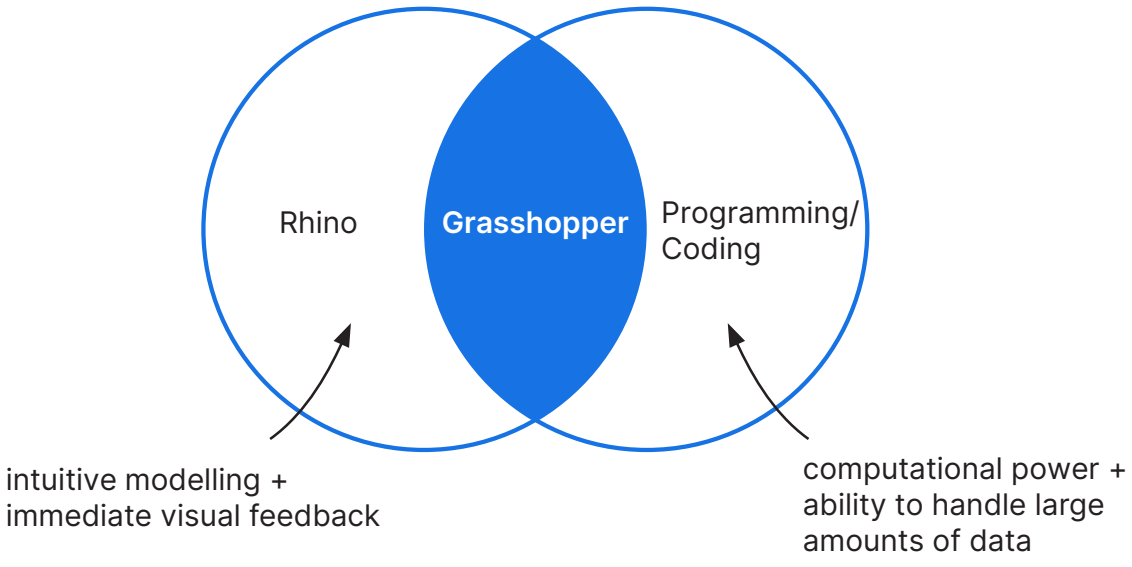
Grasshopper lives at the intersection between Rhino modelling and programming. It builds on Rhino's core command library and extends its possibilities by allowing us to use programming techniques to control and process large amounts of data at once.

Bridging this divide, Grasshopper inherits features from both Rhino as well as from programming.

From Rhino, Grasshopper inherits almost all commands, and, with a few exceptions, the components have the same name.

This means that we can leverage all the powerful modeling techniques Rhino has to offer: We can use NURBS Curves and Surfaces, we can Loft, Extrude, Split, and much more.
Grasshopper also takes advantage of the Rhino display pipeline and gives us a live preview of our script in the Rhino viewport itself.

But if Grasshopper just mirrored Rhino's commands, it wouldn't be so difficult to learn! As a visual programming interface, it also relies on coding logic and programming techniques. While far from actual coding, there are some essential programming concepts to understand when learning Grasshopper.



Essential Programming Concepts for Grasshopper

Thanks to Grasshopper's graphical interface, we don't need to write actual code when we write our Grasshopper scripts. Still, there are four essential programming concepts that are important to understand for working with Grasshopper effectively.

- Variables and Functions
- Algorithms
- Data Management
- Debugging

Variables and Functions: The Building Blocks of Visual Programming

Any script, visual or not, has two key ingredients: **variables and functions**. As the name suggests, variables are values that we can alter at any time. In Grasshopper, they represent the geometry we aim to transform.

However, variables alone aren't enough. They merely represent a static piece of information with no direction. This is where functions come into play.

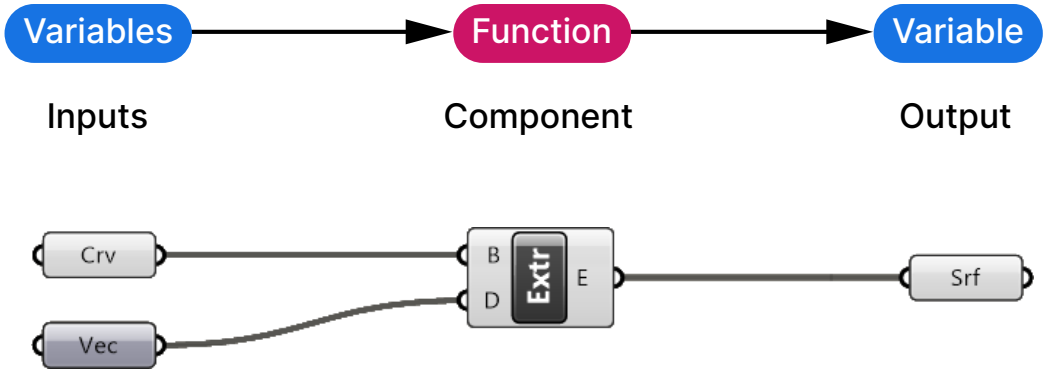
Functions take one or more variables, perform a specific operation on them, and then output the transformed variables. In Grasshopper, they act as active modifiers of geometry: they enable us to create, scale, rotate, twist, and move objects, among other operations.

In Grasshopper, these functions are embodied by **components**. Quite literally, we can plug variables into components on the left and we get the transformed variables in the outputs on the right.

In traditional programming, variables are saved in memory and assigned a unique name. The way these variables are accessed is by their name. By simply referring to "variable A" in code, the variable will be retrieved from memory.

In Visual Programming in Grasshopper, variables take on a more tangible form. Instead of remaining an abstract in memory, they live in the outputs of components. This means that at any time we can see all the variables in our script directly on the Grasshopper canvas.

This way, we can match variables with functions by creating a wire connection between them.



Mastering Algorithms: From Basic Building Blocks to Complex Designs

By creating wire connections between components, we can create complex chains of commands. We can define a custom set of rules that leads to a specific output. Instead of modelling the actual geometry, we are describing the logic that results in the desired geometry. We outline the procedure that leads to the result.

And this is why that's so powerful: When we adjust a variable or input in Grasshopper, the change ripples down the entire command chain leading to an updated final geometry. This flexibility allows us to make fundamental adjustments to a complex design swiftly and efficiently.

But there's more to Grasshopper than simply recreating the same chain of commands we would use in Rhino. We need to first carefully craft the logic of our design, and then translate it into component sequences. By doing so, we are building an algorithm.

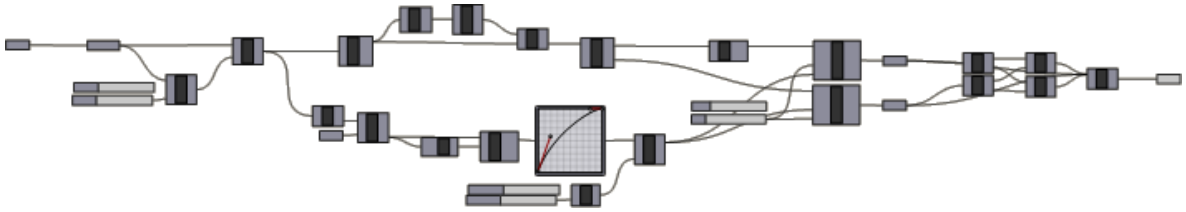
In order to build our own algorithms, we need to learn to describe a design in terms of the logical steps needed to generate it. The logic of our design depends on the individual design goal and on which key parameters we want our design to be driven by. We need to, for example, define a hierarchy of variables: which variables will determine other variables?

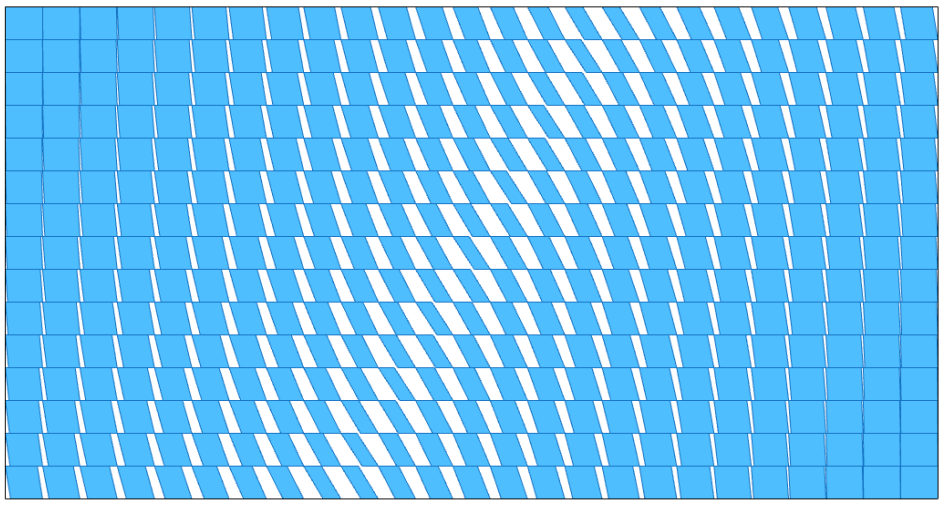
And since we can't select geometry directly in the Rhino viewport, we also need to define which objects we want to modify in relative terms. This ensures that our script works for a wide variety of cases, without our manual intervention.

For example, instead of selecting a curve directly, we instruct Grasshopper to select the curve closest to a point, or the longest curve in a given group of curves.

Learning how to break down complex geometries into a series of simple, logical operations is an essential skill to master in order to excel at Grasshopper and parametric design.

Component Sequence and its Output





But knowing the sequence of components is not enough. A component can process a single geometry, or many hundreds of objects at once. This is where Data Management comes in.

Data Management: The Key to Efficient Design in Grasshopper

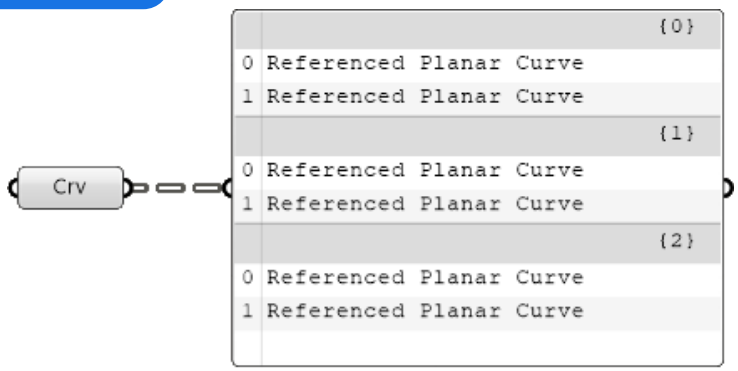
Because we can't select geometry in Grasshopper the way we do in Rhino, our selection and object manipulation occurs on a more abstract level: with Lists and Data Trees.

The benefit of this abstraction is that we can access and manipulate data at scale: We can define and select singular items, manage items within a list, and deal with databases containing thousands of items. This allows us to stay in control when processing and updating a lot of geometry at once (and save hours of work!).

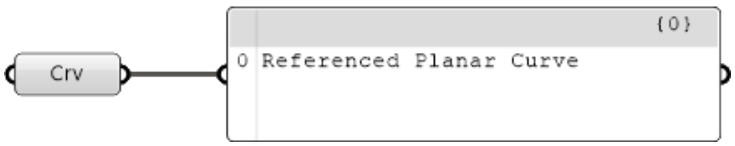
For instance, we can rotate hundreds of façade elements at once, each with an individual rotation angle. In Rhino, we would need to rotate each element individually, one by one.

Examples of different Data Structures

Component containing three sets of two curves each



Component containing a single curve



Data Management is at the core of everything we do in Grasshopper. Every single component outputs data. One of our tasks is to manage and control the flow of data through the component sequences we create. Data management goes beyond simple wire connections, it's about the structure of the data that's running through the wires.

We need it to tell Grasshopper precisely which geometry to modify with which parameter - a practice called "data matching".

Debugging in Grasshopper: Navigating Visual Error Detection

As the number of components in a script grows, so does the likelihood of encountering unexpected results. A script may fail due to an error in the logic or an unexpected variable. While this is a normal part of the process of writing a script, we need a quick way to find the errors and fix them.

Grasshopper has built-in **debugging utilities** that help us to find the problem quickly and get on with our design.

Debugging is a term from the programming world and describes the process of finding the location in a script where something went wrong and resolving it.

There are a few intuitive ways Grasshopper helps us with debugging.

One of them is the **live preview** in the viewport: we don't need to write the whole script before we see if it works as desired, we get visual feedback every step of the way directly in the Rhino viewport.

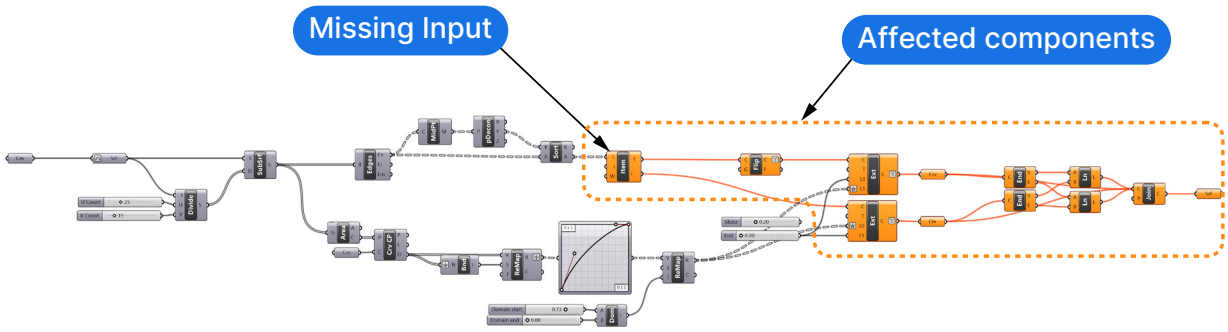
We also get visual cues on the "health" of individual components: when the component is gray - everything's fine, when it's orange one or more inputs are missing and when a component turns red - it means something is wrong with our input data, and the component failed.

Component Health in Grasshopper:

In addition, **tooltip popups** when hovering over the component inputs and outputs give us additional information.

There are even special components like the **Panel** in Grasshopper, that allow us to read the output of components to understand if the data is being processed correctly.

When a component fails, it will affect all the components “downstream”, meaning all the components that depend on that information to run. Thanks to the component health visualization, we are able to easily pinpoint where the script “broke” and fix it.



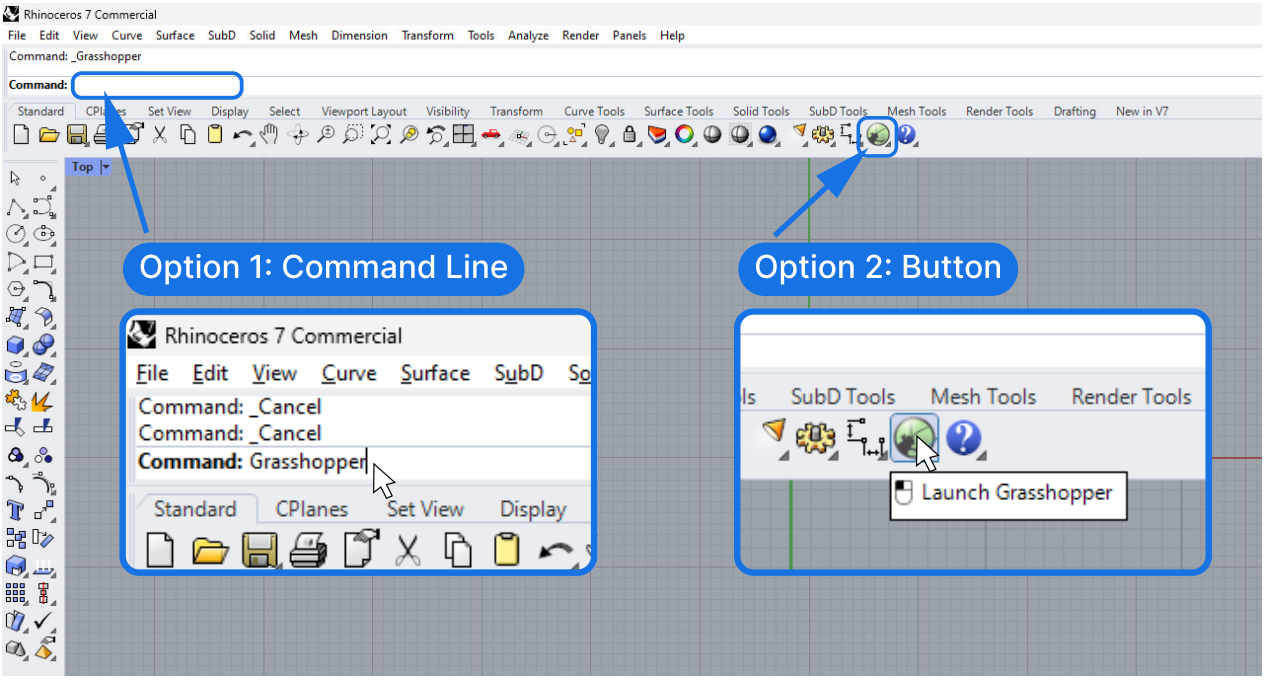
And there you have it! You’ve taken your first step into the world of visual programming with Grasshopper for Rhino. Next, let’s learn about the interface!

2. The Grasshopper Interface: A Beginner's Guide

In this chapter, we'll be exploring how to find and add components, understand their inputs, connect components, and utilize the live preview feature. So, let's launch Rhino and get started!

Starting Grasshopper

First things first: let's open Grasshopper. There are two ways to start up Grasshopper. The first, is to type "Grasshopper" into the Rhino command bar and hitting Enter.
The second option is to click the Grasshopper button on the far right of the Standard Rhino toolbar.



You'll see the Grasshopper loading screen and after a few seconds a new window will open. This is where the magic happens!

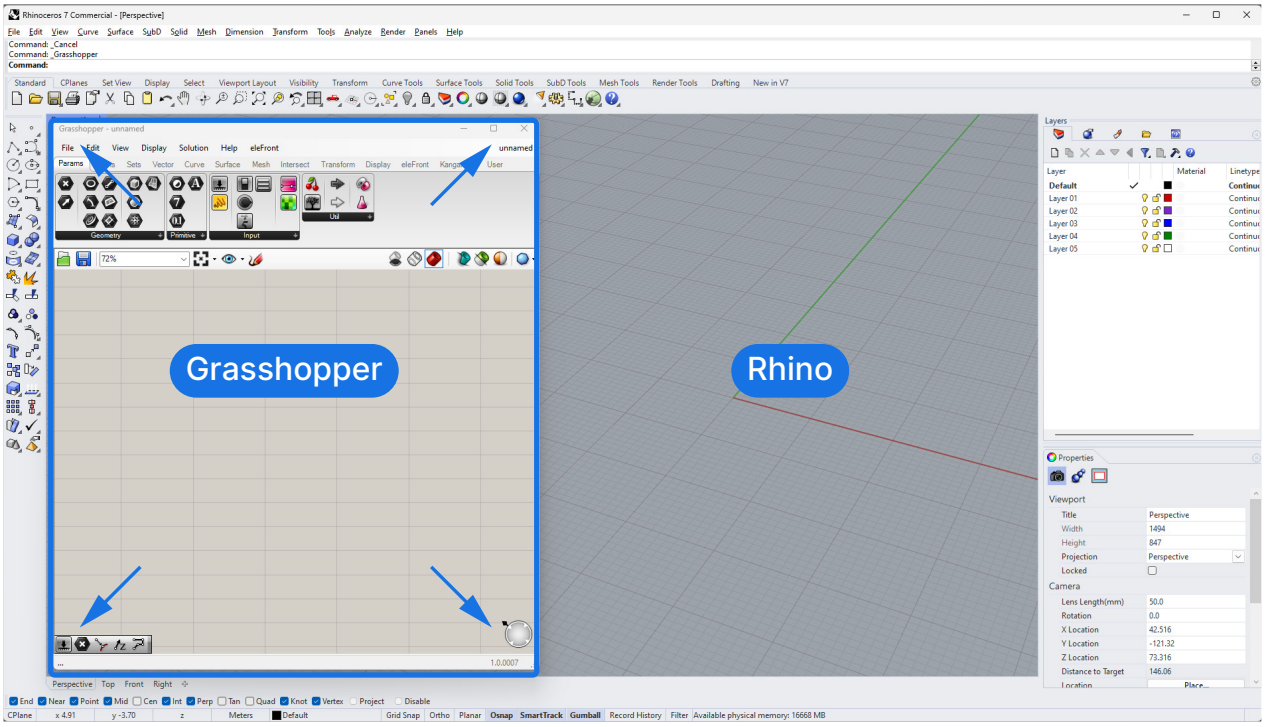
Setting Up Your Workspace

As we create our scripts in Grasshopper, we'll see a live preview of any geometry we generate directly in the Rhino viewport. One of the benefits of Visual Programming and the Grasshopper interface is that we see the result of our script as we develop it!

To that end we'll want to set up our workspace so we see both the script and the output side by side. This way we'll monitor if our changes in the script have the effect we want.

So let's move the Grasshopper window to the side of the screen, or to a second screen, if you have one.

This way we have a clear view of our script workspace as well as the resulting preview geometry.



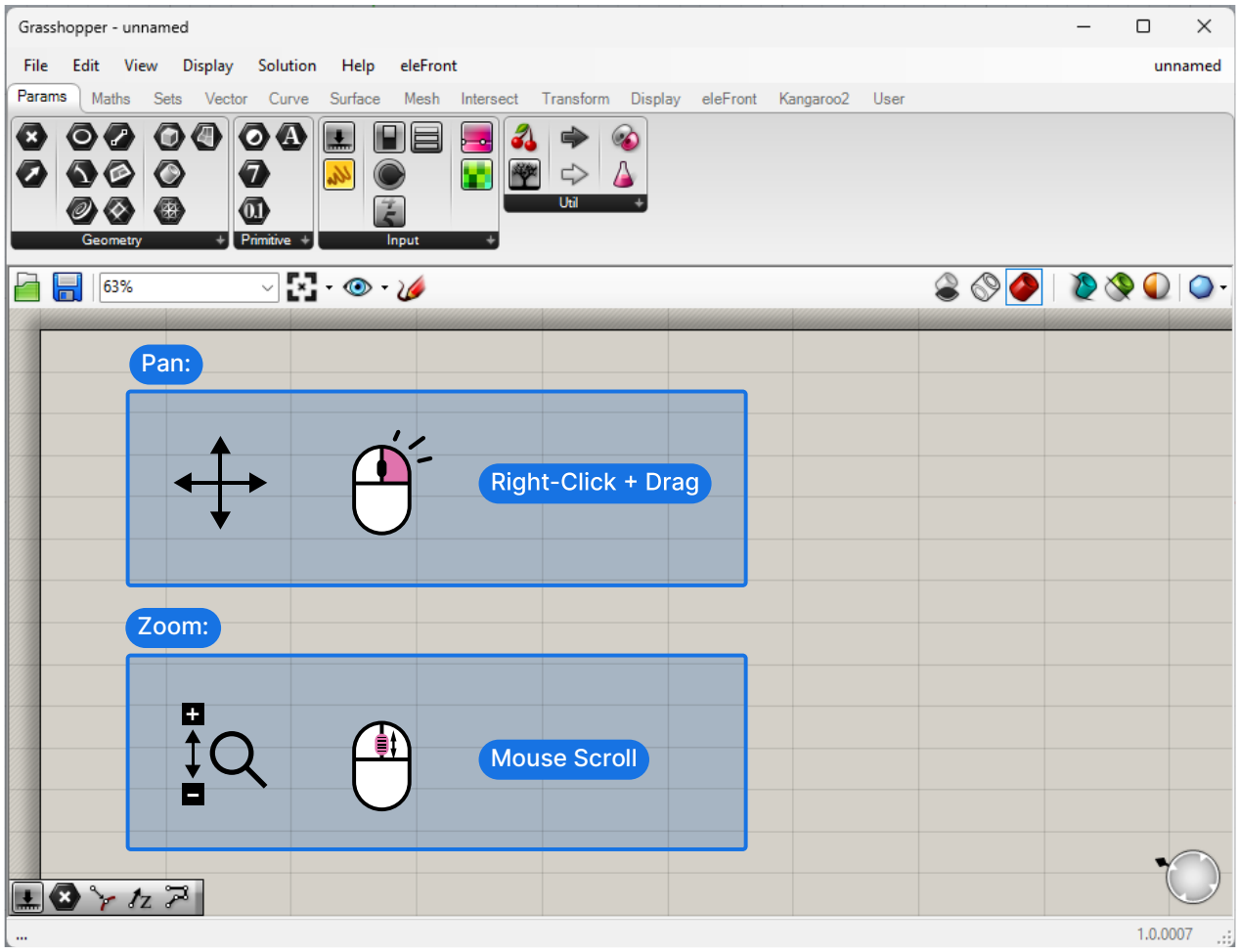
If you are using a single screen and you want to get a better view of the viewport, you can quickly **minimize** the Grasshopper window by **double-clicking** on the top bar of the window. Another double-click and the window is restored.

Navigating the Grasshopper Canvas

Now that we've positioned the Grasshopper window to our liking, let's take a closer look at the Grasshopper interface. The main workspace for our scripts is the Grasshopper canvas. This is where we'll place and connect our components to create scripts.

We can move and navigate the canvas just like the top view of the Rhino Viewport.

- Right-click and drag to pan.
- To zoom in and out, scroll the middle mouse button.



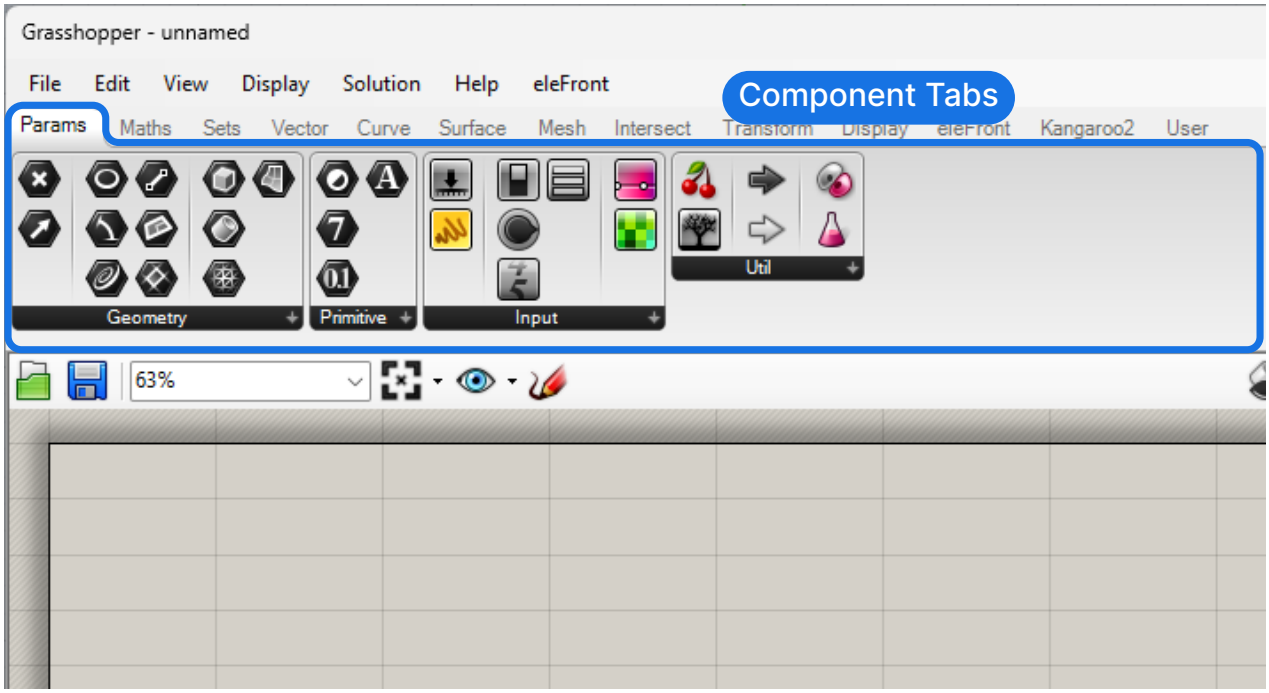
By zooming in and out we can keep an overview when needed, but also zoom in on a set of components to solve a specific problem.

Understanding Component Tabs

In the top portion of the Grasshopper interface, above the canvas, we find the component tabs. They contain all the components we are going to use to create our scripts and they're bundled in different tabs to make finding components easier.

There are different kinds of components in Grasshopper: Some mirror the geometry-related commands we know from Rhino, others are specific to Grasshopper and help us manage data and use mathematical formulas.

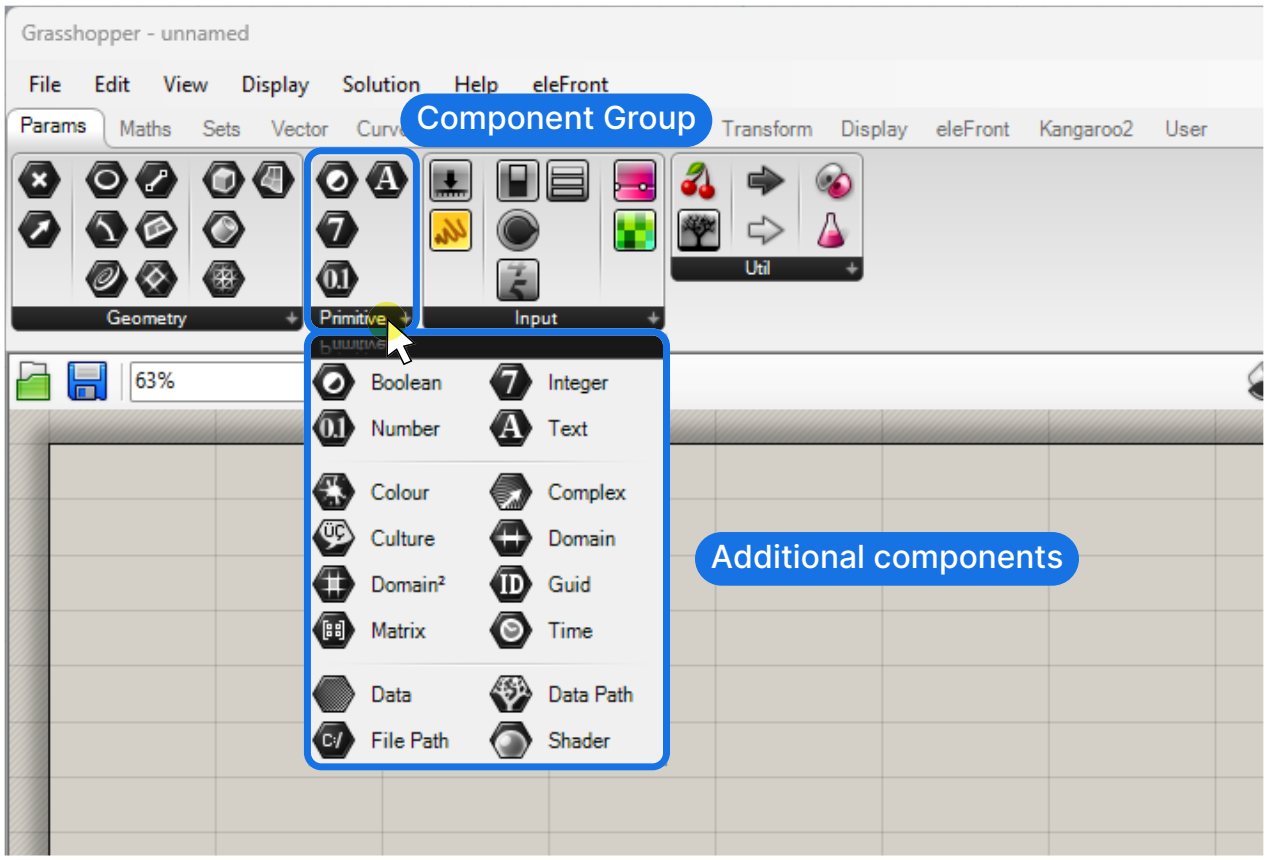
The first three component tabs: Params, Maths and Sets, allow us to interact with Rhino objects, create and manage variables and manage multiple objects with lists and data trees.



The following tabs contain the more familiar geometry-specific components. By and large these components have the same name as the Rhino commands, so as a Rhino user you'll already be familiar with most of the commands and what they do.

Let’s take a closer look at one of these tabs:

Within each tab, there are additional subsections which group the components further. If we click on the black bar on the bottom of each group, the drop-drown shows even more components, these are usually more specific and used less frequently.



Take some time to go through each of the component tabs to get a feeling for which commands are available in Grasshopper!

Adding Components to Your Script

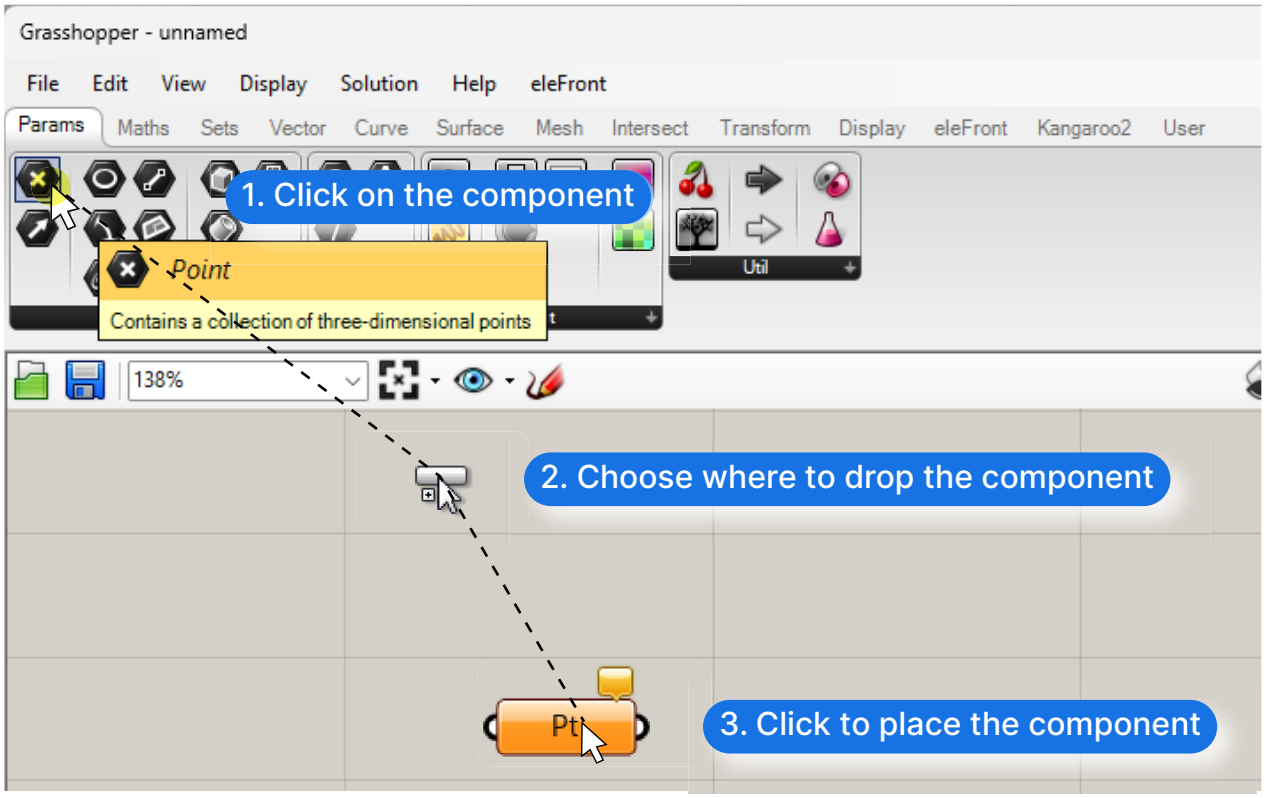
Now that we've explored the basic layout of the Grasshopper interface, let's bring it to life by adding some components. This is where the real fun begins as we start to see our designs take shape.

There's two ways to to add components to our script.

Let's say we want to add point container component. Container components are nodes that store certain geometry, without modifying it. There are container components for every geometry type available in Rhino: Curves, Surfaces, but also Numbers and Text.

Option 1: Adding Components With the Component Tabs

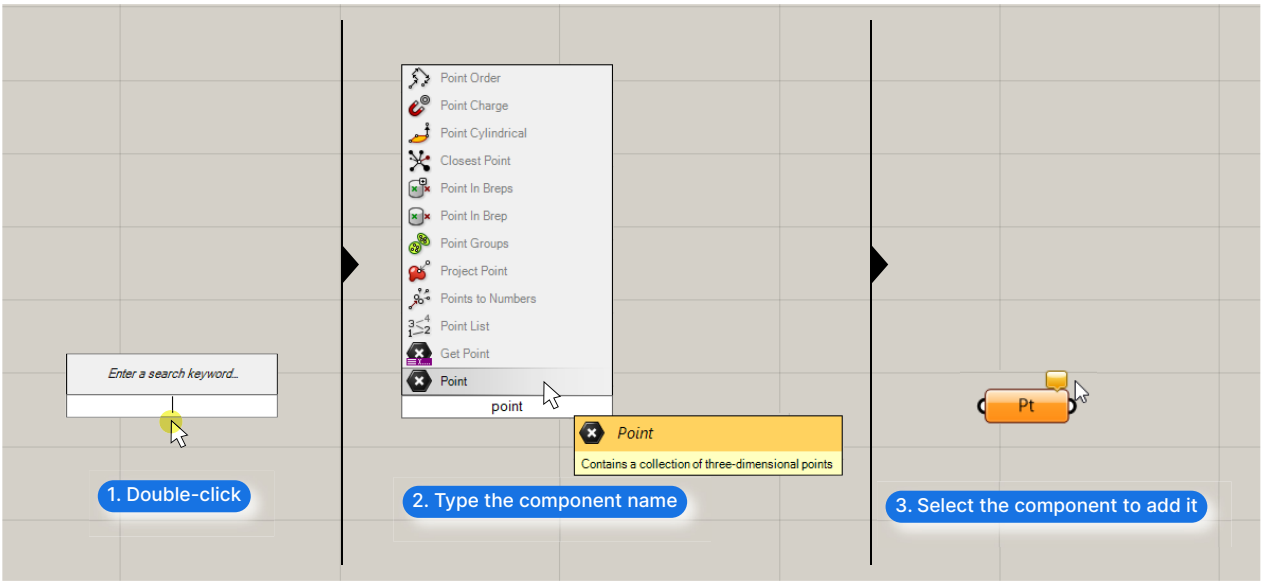
We can find components by going through the component tabs and subsections, "Params" and "Geometry" in our case, then we hover over the components to see their name and what they do, and then select the component we want and simply click on the canvas where we want to place it. This will add a component for us.



Option 2: Adding Components With the Search Bar

Another, quicker way of adding components is to use the component search bar. To activate it, double-click on the empty canvas and type the name of the component you're looking for.

Let's add another point component this way. We double-click and type 'Point' into the search bar



As we type, all the components that contain the search term will appear above the search bar. The closest match to your search term will be shown at the bottom, closest to your mouse pointer. If you are not sure which one to pick, hover over the results to get a short description of what the component does. Then just click on the component to add it to the canvas.

The component will be added exactly where you double-clicked to open the search bar.

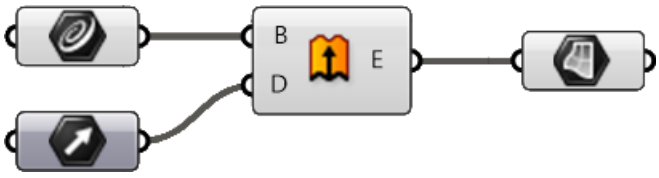
Adjusting the Component Display

Components a vital part of our scripts, so being able to recognize them and being able to tell what they do is paramount. The Grasshopper interface offers a few different options for their display.

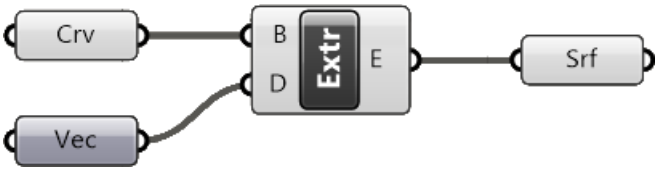
We can show a small icon in the middle of the components, opt for an abbreviation of the component name, or show the full name instead.

The image below shows the exact same components, in the three display modes.

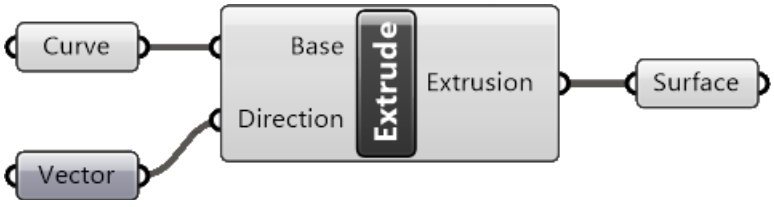
Icon Display



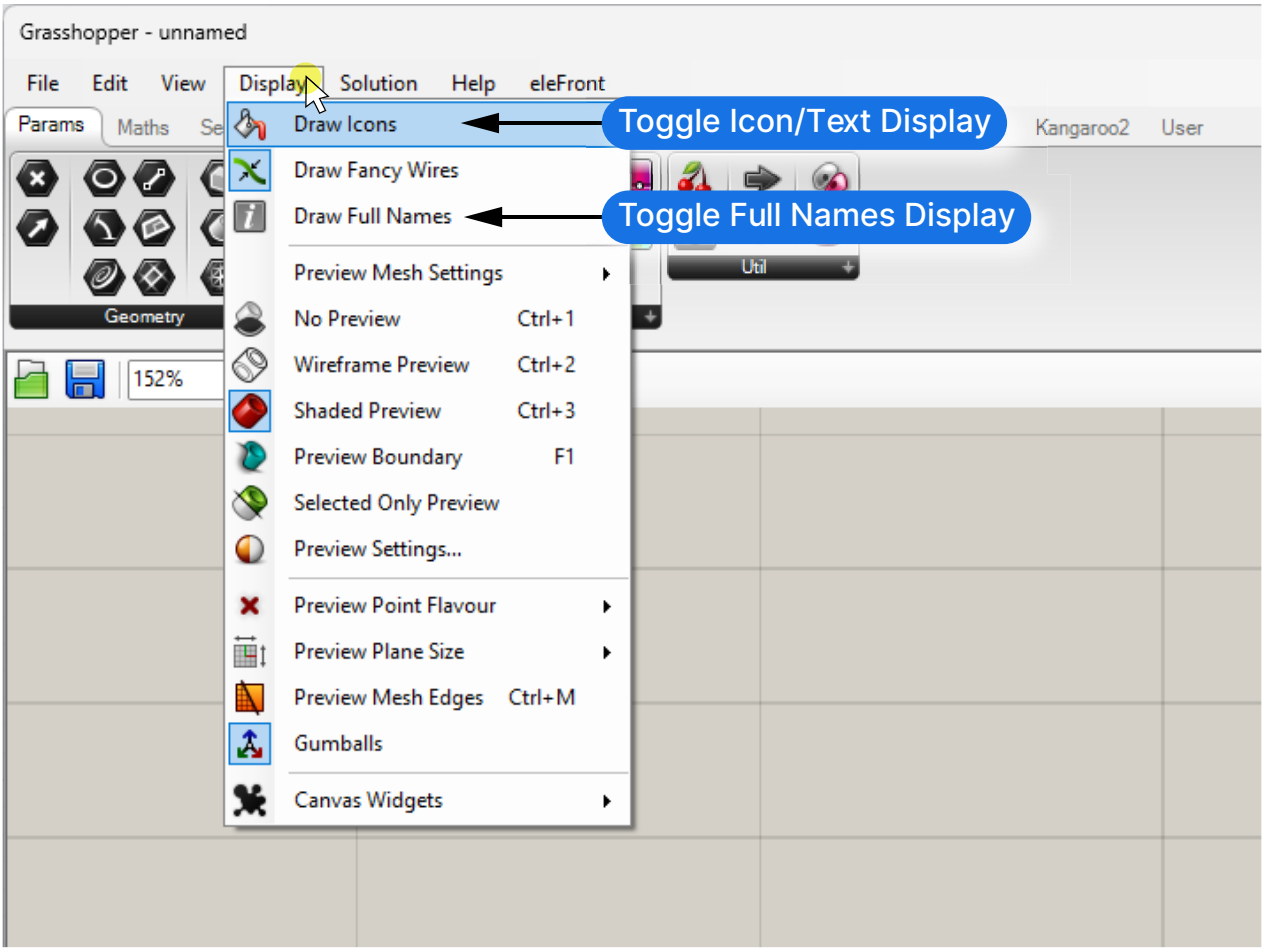
Text Display



Full Names Display



In the menu, under Display, and ‘Draw Icons’ we can toggle the graphical display of the components.



Usually people start out with the graphical icons and later on move to the text display. Personally, I find the abbreviations more helpful since the icons can be ambiguous. Choose the one that feels more intuitive for you!

Regardless of the component display, the inputs and outputs will be displayed with letters. When starting out with Grasshopper, the letters alone can be confusing. To help with that, we can display the full name of the inputs and outputs, by going to Display in the top menu and selecting ‘Draw Full Names’.

While this is great for starting out, the components will appear much larger on the canvas. As you get more familiar with the components, you can switch back to the single letter mode.

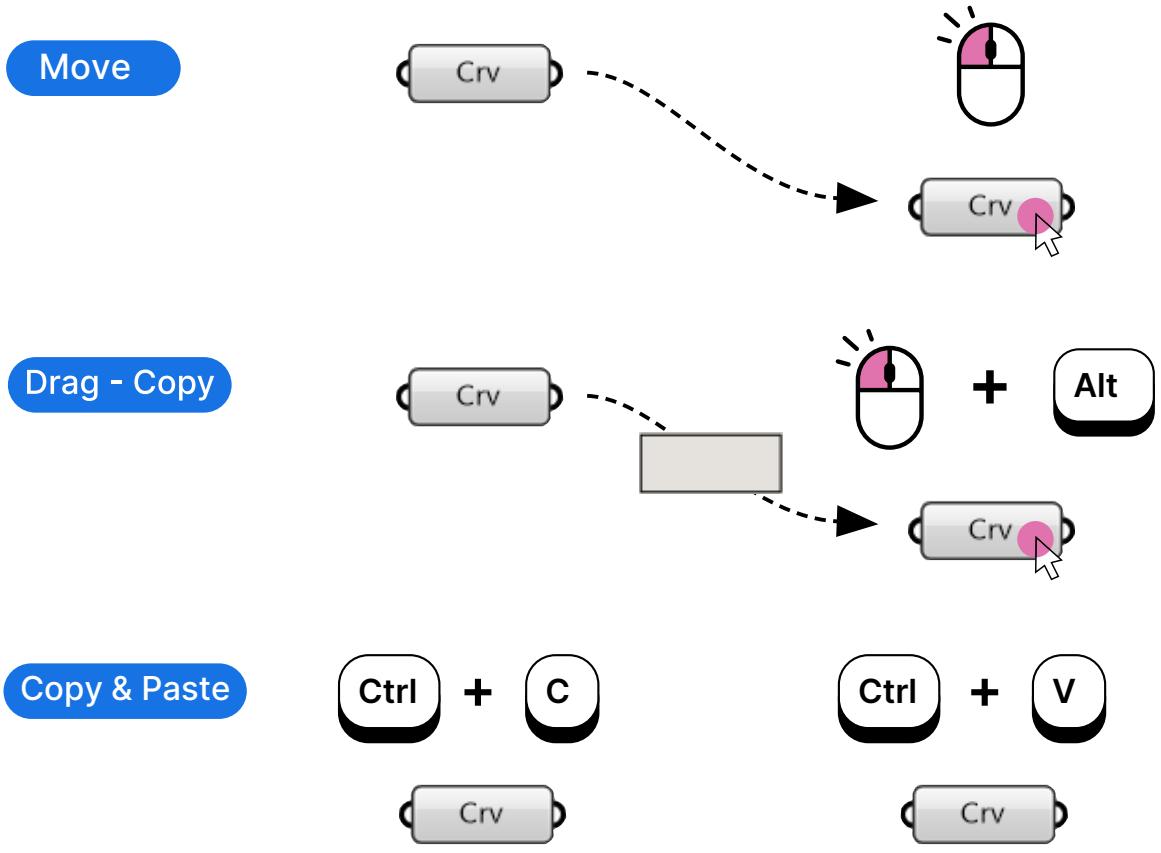
Moving and Copying Components

Now let's get comfortable interacting with components in the Grasshopper interface!

To **move** a component around on the canvas, simply click and drag it.

We can also copy components by clicking and dragging them, and then hitting **Alt** once. Instead of moving, a copy of the component will be created once you let go.

To **copy**, **cut and paste** components, use the same commands you would use in a text editor. Copy and paste any components on the canvas with **Ctrl + C** and **Ctrl + V**. Or cut and paste with **Ctrl + X** and **Ctrl + V**.



To **delete** components, simply select them and hit "delete" on your keyboard.

Understanding component health

Let’s keep two of the Point components on our canvas.

Now as you may have noticed, when we drop the point component onto the Grasshopper canvas, it turns orange. This is Grasshopper’s way of telling us that something is not quite right.

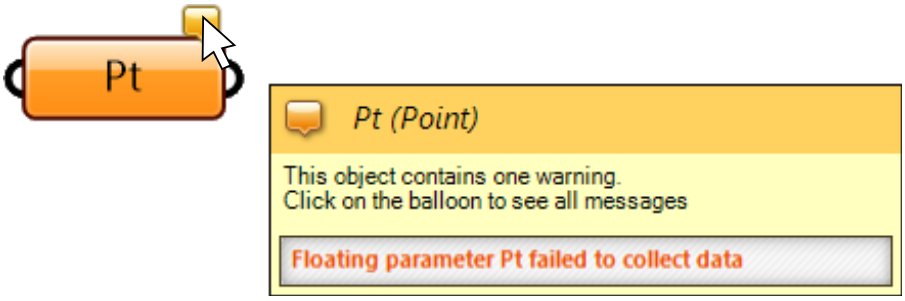
Grasshopper gives us cues about the “health” of components by coloring them:

- A **gray** component means everything’s fine.
- An **orange** component means that one or more inputs are missing.
- And a **red** component means that something went wrong. Usually because we provided the wrong input.

Component Health in Grasshopper

When a component is orange or red, we can get more information on what's wrong by hovering over the bubble on the top right corner.

Component Status Tooltip



In the case of our point, it says: "Parameter Pt failed to collect data" - in other words there is **no input**.

To fill them with data, we have two options. We can connect geometry from elsewhere in our script, or we reference geometry directly from the Rhino document.

Let's learn how to do that.

Referencing Geometry from Rhino

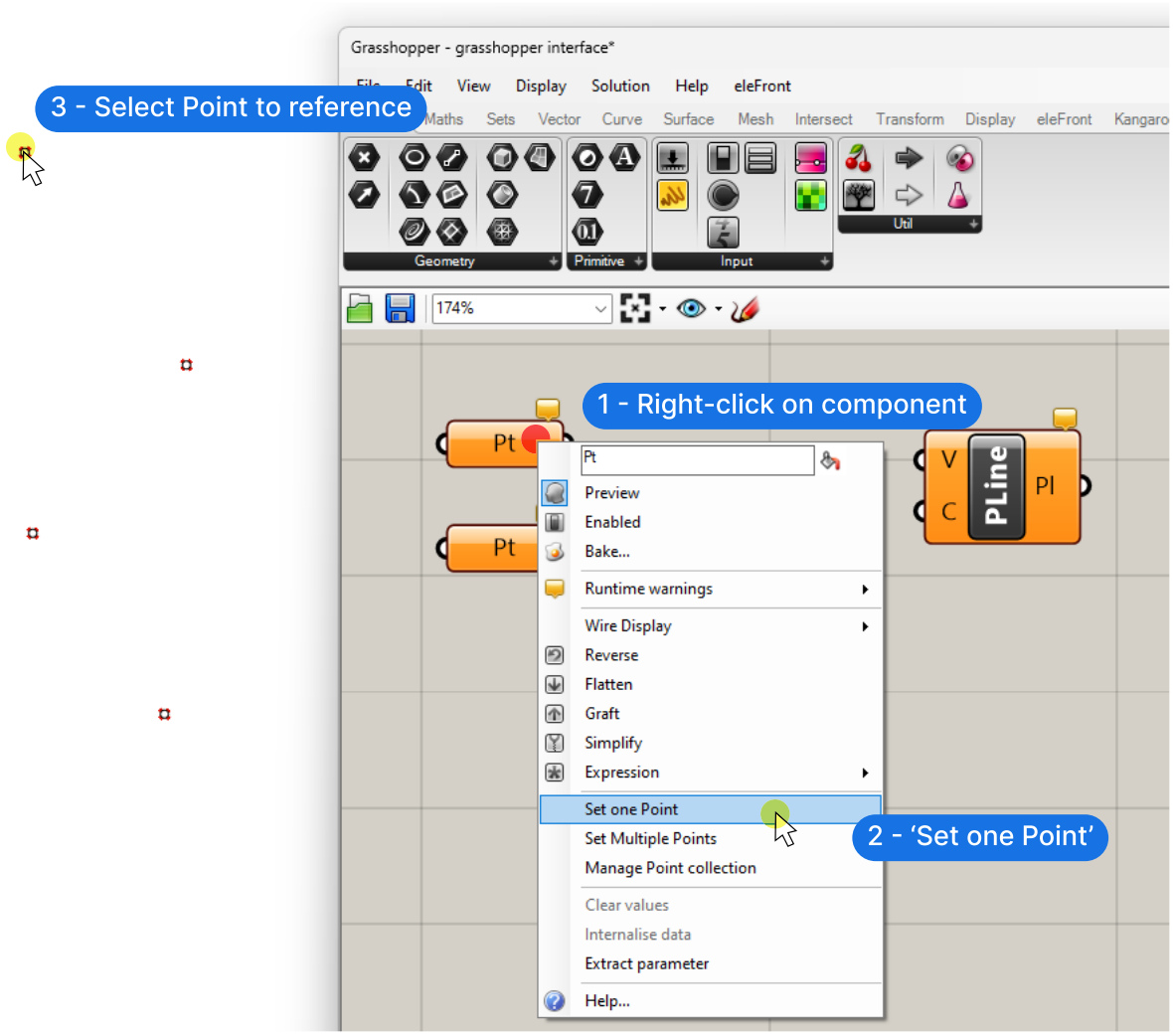
Grasshopper's possibilities are not limited to the Grasshopper interface. We can seamlessly combine native Rhino objects with parametric Grasshopper geometry by **referencing geometry**.

To reference geometry means creating a **link** between actual geometry in our Rhino file, and Grasshopper. If we move the geometry in Rhino, Grasshopper will pick up on it and update the script accordingly. This adds another layer of interactivity to our scripts!

Let's learn how to do it:
Let's create four points in the Rhino top view. These will be the points we'll reference.

To reference geometry in Grasshopper, **right-click** on the container component you want to link to Rhino geometry. Make sure the data container matches the kind of geometry you intend to reference to it. In our case we have a point container and can only reference points.

In the dropdown menu, chose between "**Set one point**" or "**Set multiple points**". Let's select 'Set one point'. As soon as we click, the Grasshopper window will minimize - that's normal! It's to clear the viewport so we can make our selection more easily. Let's select the first point.

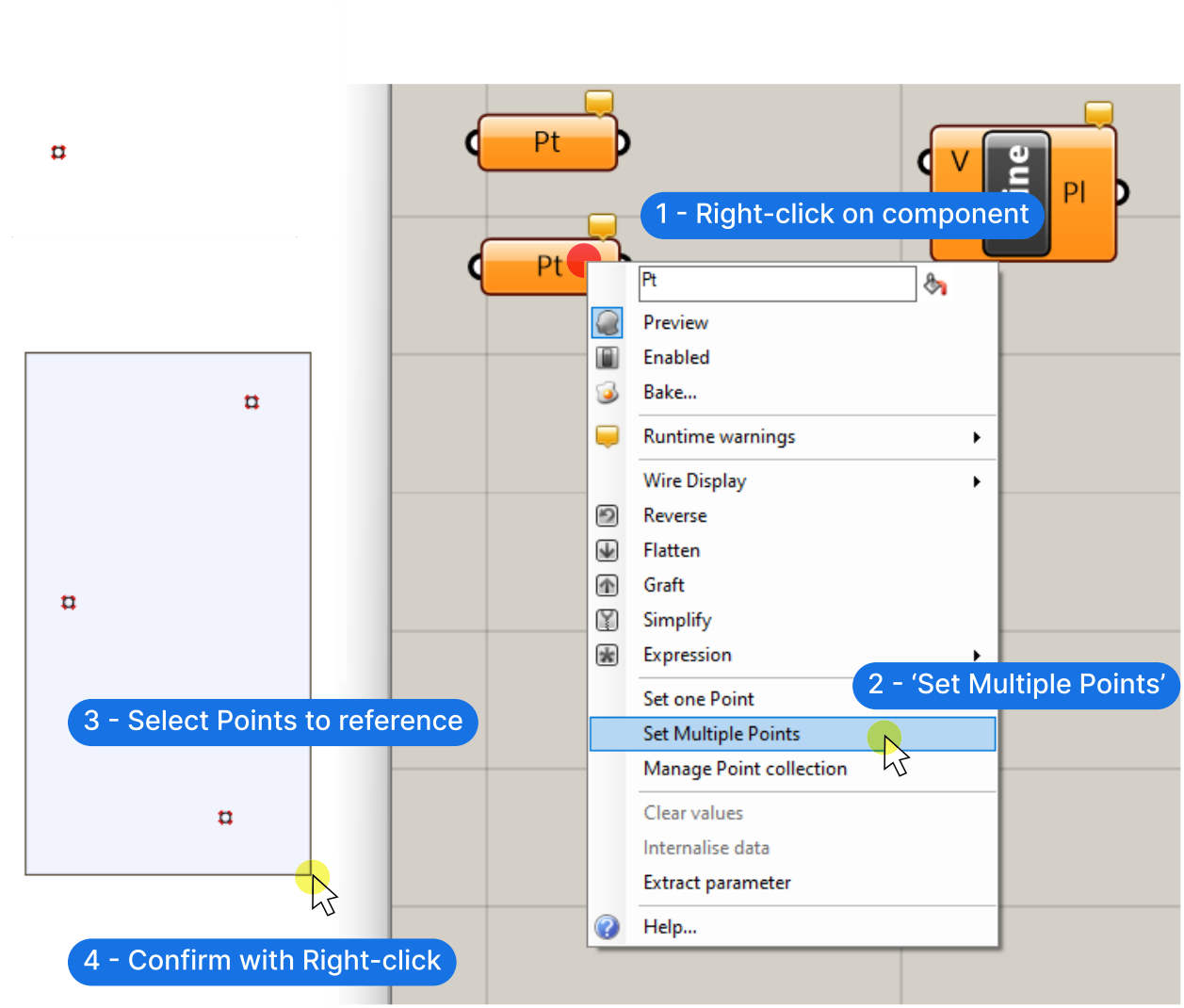


Done. The link is now created. The Grasshopper window will reappear as soon as we make the selection, and the component turns gray.

You can also select the geometry in Rhino first, and then right-click on the component and click set-one-point. The result is the same.

Referencing Multiple Objects

We can also reference several points to the same component. To do that, go to "**Set multiple points**", select the remaining points, and then right-click to confirm the selection.

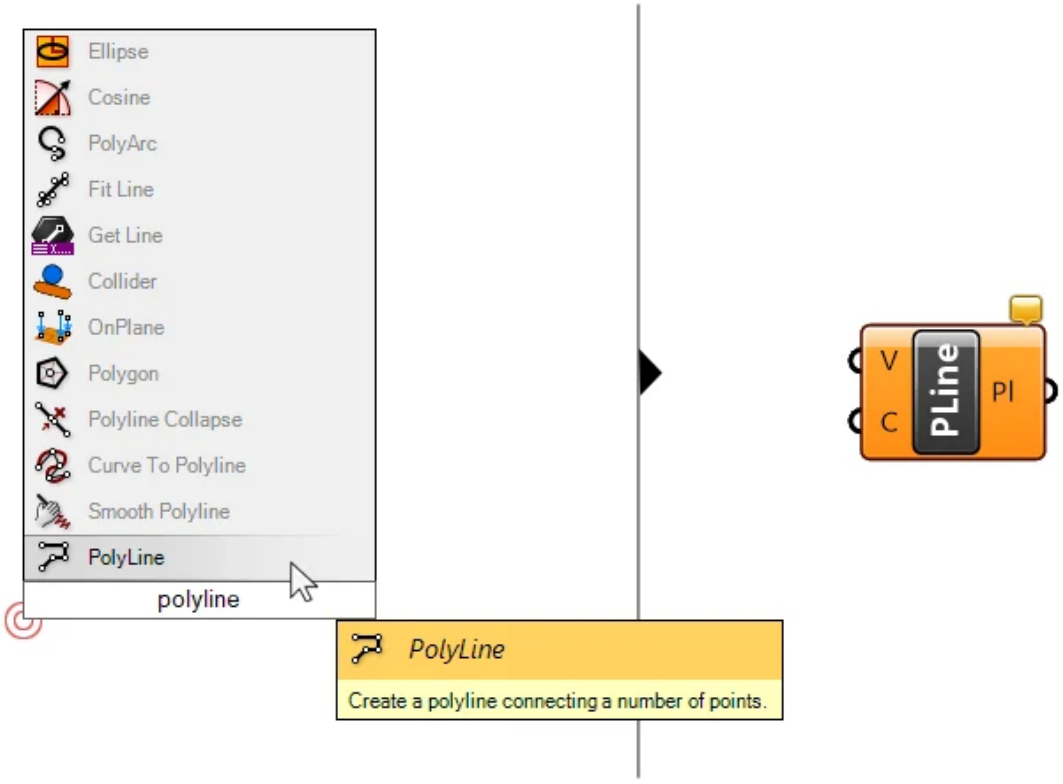


The point component will then contain multiple referenced points.

By hovering over the output we can check how many points each component contains.

Now let's create a **polyline** that connects these four points.

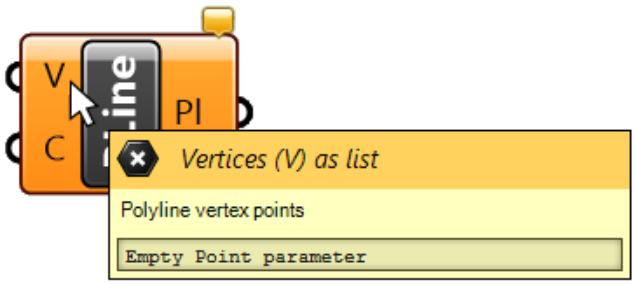
We could go to the component tabs and look for the component, but it's easier to simply double-click and type "Polyline". Let's hover over the first result to check if it does what we want, and then drop it on the canvas by clicking on it.



Understanding Inputs and Outputs

As we delve deeper into the Grasshopper interface, understanding component inputs and outputs becomes crucial. Components in Grasshopper can have multiple inputs and outputs. The inputs are on the left, and the outputs on the right. This is the only direction the information will flow!

We can find out what the required inputs are by **hovering** over them with the mouse pointer.



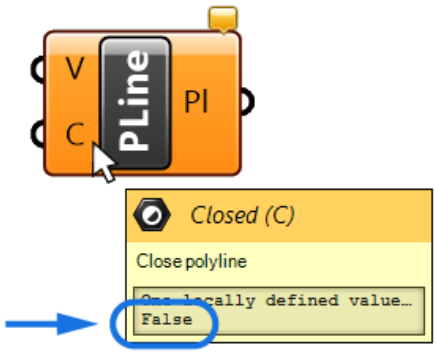
The polyline component for example, asks for the vertices (points) to connect with the polyline in the input V (for Vertices).

While the letters on the inputs give us a hint about the required input (e.g. V for Vertices), it's always good to check what the required inputs are by hovering over them. When starting out, connecting the wrong inputs is one of the most common mistakes! If a component turns red upon creating the connection, this is most likely the reason.

When in doubt, hover over both the output and the input of the components you are trying to connect and make sure they contain the same data type!

Predefined Inputs

Back to our Polyline component. The input C (for Closed) asks us whether we want to create a closed curve or not. By hovering over the 'C' input of the Polyline component, we can see that it already has a default value, which is 'False'.

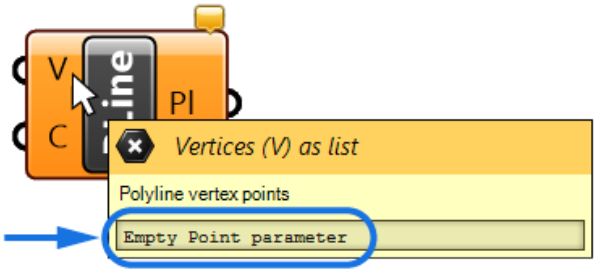


In Grasshopper, **True or False** values are called "**Boolean Values**". They specify whether an option should be on or off. Since the default input of the "Closed" input of the Polyline is False, it means that the polyline won't be closed. In other words, the two ends of the curve won't be connected.

Some Grasshopper components will come with **predefined, default values**. In that case we only have to provide an input if we want to change them.

It's just like running commands in Rhino: there are several additional options in the command line and usually these options have a default value.

The 'V' input of the Polyline component (the points to connect with the polyline), says "**Empty Point parameter**" in the tooltip, which is Grasshopper's way of saying that there is no data / no input.



Let's change that by connecting our referenced points!

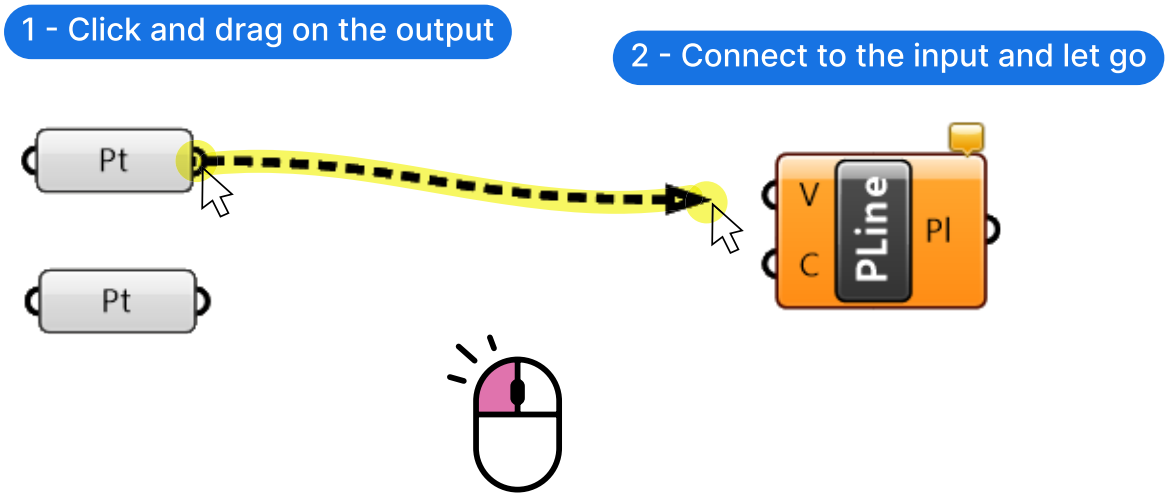
Creating Wire Connections between Components

Components alone are like static building blocks of our scripts. Only by creating connections between them we can bring them to life by allowing the information to flow through!

Let's learn how to **connect** components.

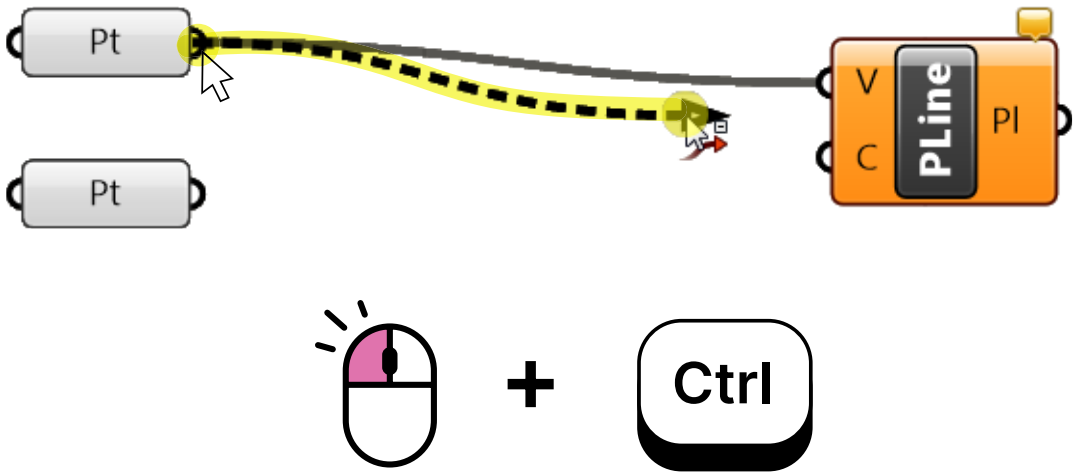
We are going to feed our referenced points into the vertices (V) input of the polyline component.

To create a connection, simply move your mouse to the output of the first component, until a little arrow appears, then click and drag to create a wire. As you drag it closer to components, it will snap to any available input. Let's connect it to the Vertices (V) input. Just let go to complete the connection.



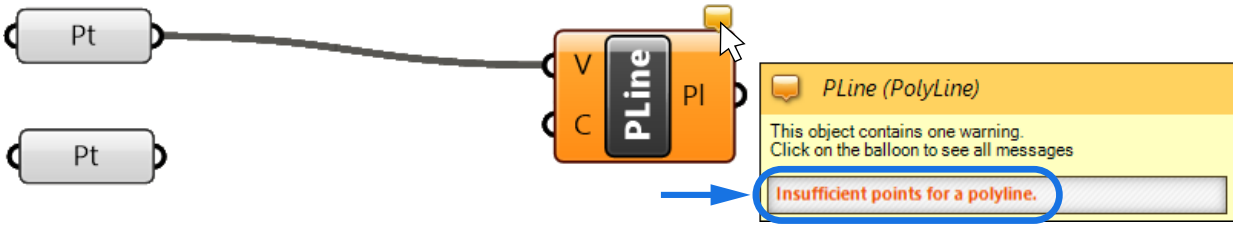
Removing a connection

To remove a connection, press Ctrl while repeating the exact same maneuver: click and drag the wire from output to input of the components you want to disconnect. A small red arrow and a minus sign will let you know that the connection will be removed. Make sure to keep pressing Ctrl until you let go of the mouse button.



We successfully connected the point.

But the polyline component is still orange, let's see what it says. By hovering over the orange bubble we see that it says "Insufficient points for a polyline". And that's true, we can't create a line from a single point.

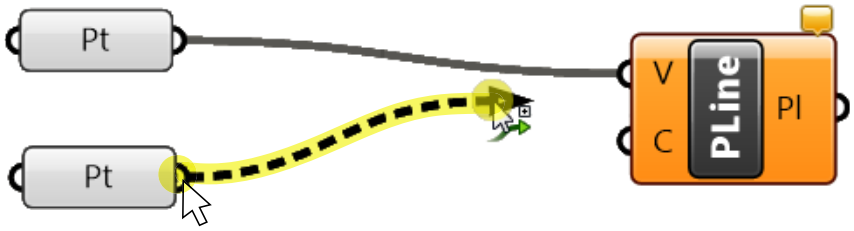


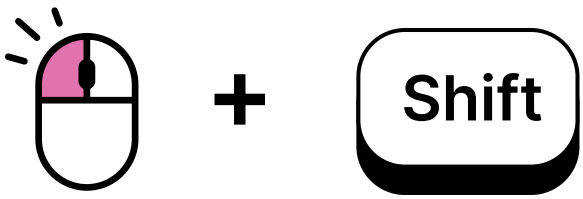
Connecting multiple inputs

So let's feed the remaining three points stored in the second point container into the same input.

If we connect the second point container in the same way, you'll notice that the previous wire disappears. That is the default wire behavior: **any new connection will replace the existing one**.

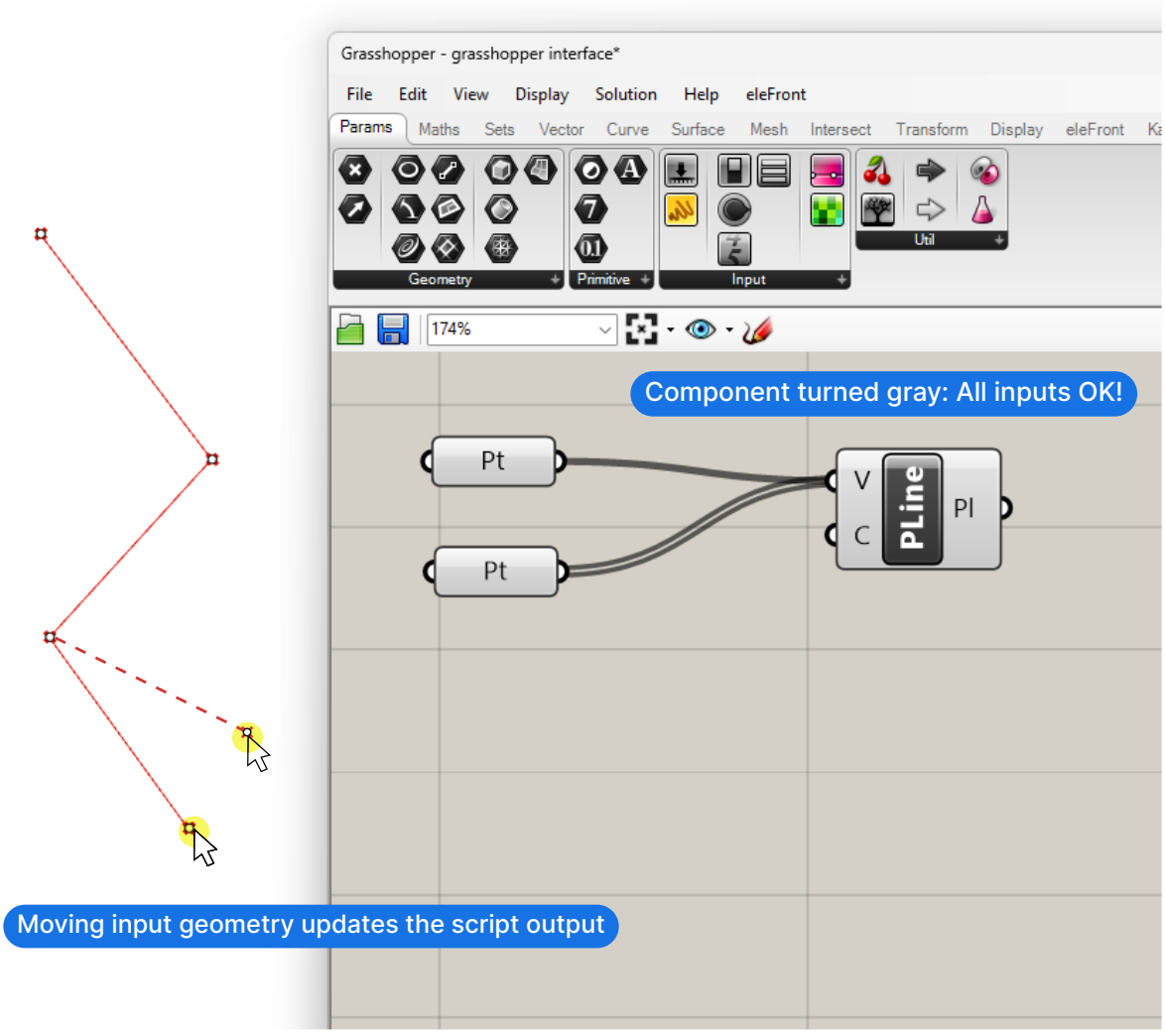
If we want to **add** the input to the existing one, we need to press Shift as we create the additional connection. **Click and drag**, and additionally press **Shift** before letting go of the wire, you will see a small green arrow and a little plus icon appear. This way both points are added to the input.





It’s important to note that whenever we are adding multiple inputs, the **order** in which we plug the points into the Polyline input will determine the order in which the points will be connected by the polyline.

Since we referenced the points from Rhino, if we move a point in Rhino, the polyline will automatically be updated!

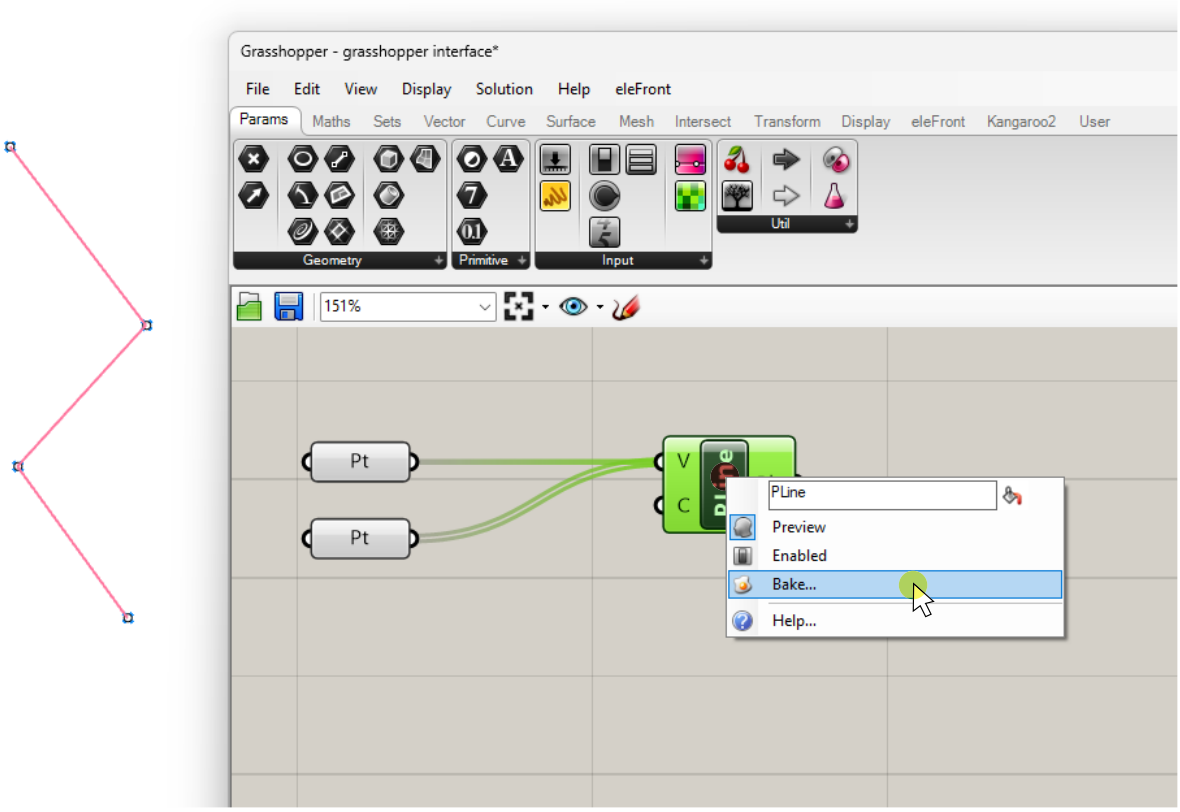


Baking Grasshopper Geometry to Rhino

As we referenced the points and created the polyline, it showed up in the Rhino viewport. You probably noticed that we can't select this curve in Rhino. The reason is that everything we create in Grasshopper remains a **preview** in the Rhino viewport until we export it into the Rhino document.

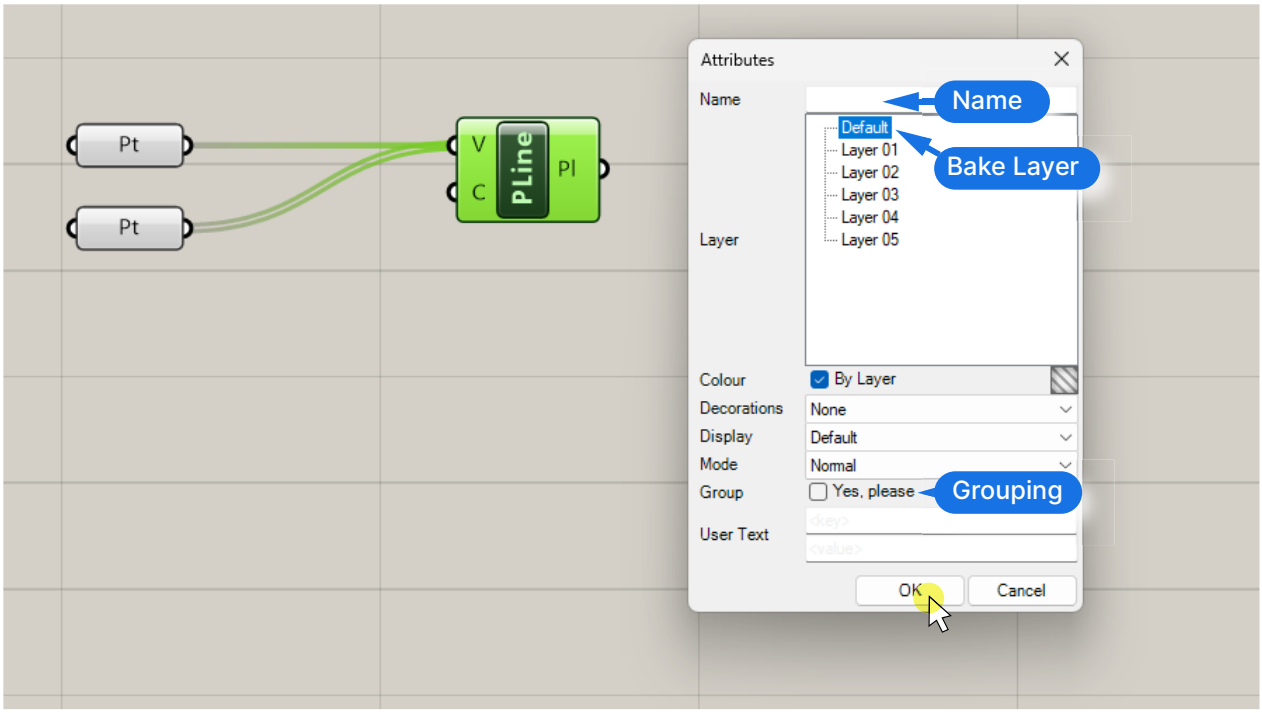
Exporting Grasshopper geometry to Rhino is called "baking" in Grasshopper.

We can bake any geometry we created in Grasshopper by selecting the component that contains it, **right-clicking**, and selecting **Bake**.



The pop-up window that will appear allows us to specify which layer we want to bake this object to, and on the bottom, whether we want to group the objects, in case of multiple objects. On top, we can give the object a name.

Click ‘OK’ to confirm.



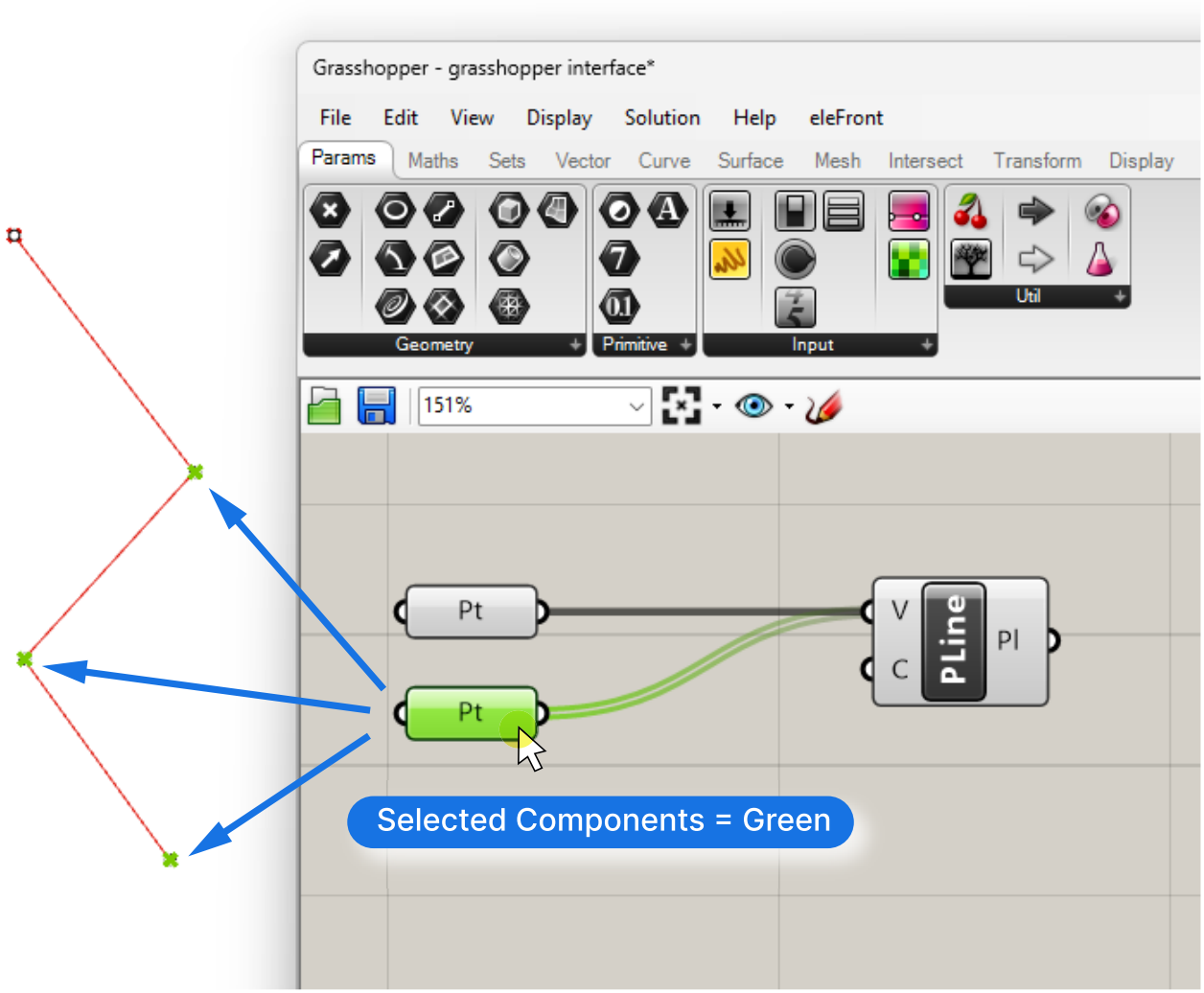
Now the curve is converted into a native Rhino object and we can select and manipulate it in any way we want.

It’s important to note though, that this baked geometry has no connection to the Grasshopper script anymore! If we move it to the side, we can see that the grass-hopper preview is still in the same place, unchanged.

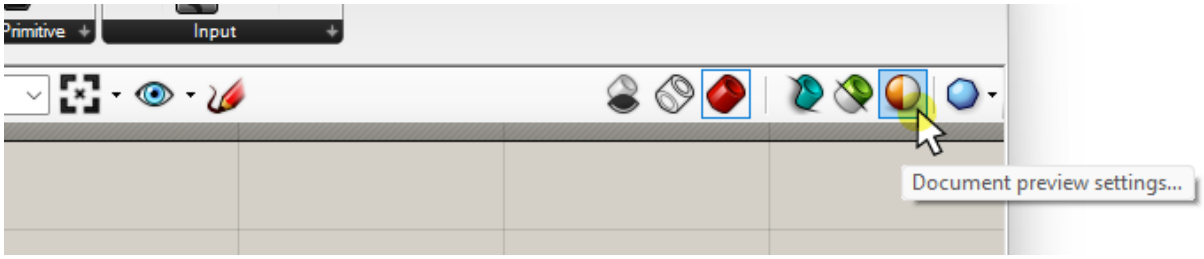
Customizing the Preview

To master the Grasshopper interface, we need to be able to control one of its main features: the preview.

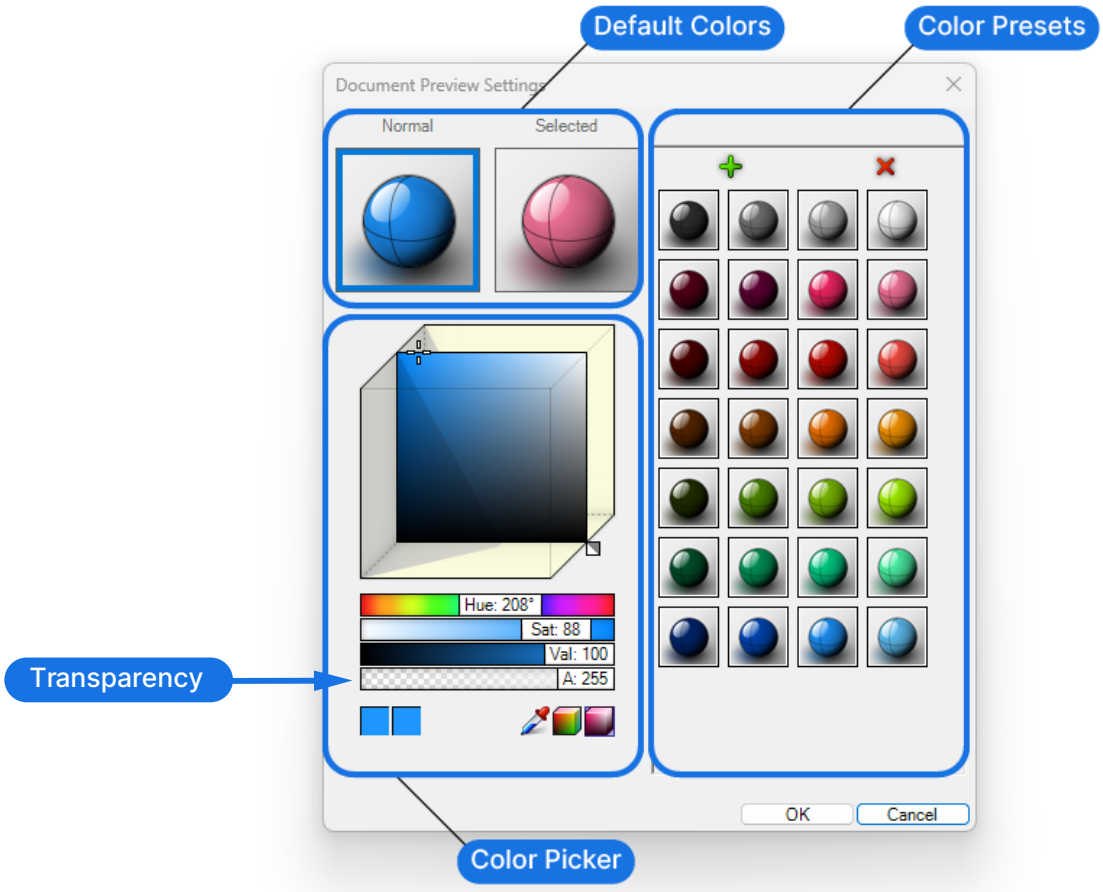
By default, all Grasshopper preview geometry in the Rhino viewport is displayed in red. If we select one or more components, their contents will be highlighted in green. It's a quick way to see which component generated which geometry.



You can change the default preview colors by clicking on the ‘Document preview settings’, the orange/white circle in the top right corner of the Grasshopper canvas.



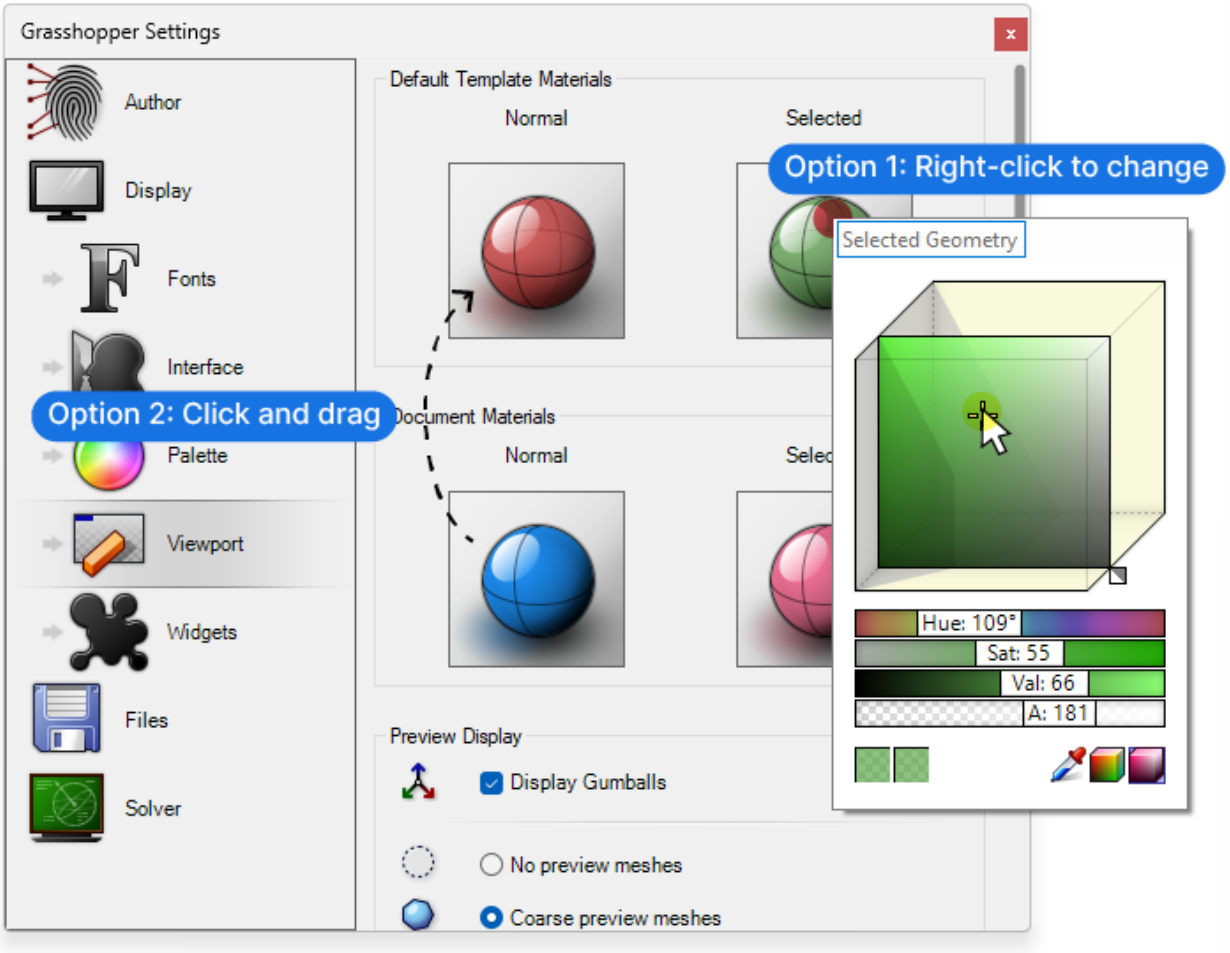
We can change the color but also the transparency of the preview. The transparency is controlled with the bottom slider (A). Below, I changed the unselected preview color to blue and the selected preview to magenta.



I find it most helpful to remove the transparency entirely to get the equivalent of a shaded mode in Rhino.

If we change the preview colors here, the settings will only be saved for the current Grasshopper script. Creating a new Grasshopper file will reset to the default red and green previews.

To change the default preview colors for all new files, we can go to **File > Preferences** and navigate to the **viewport** section.



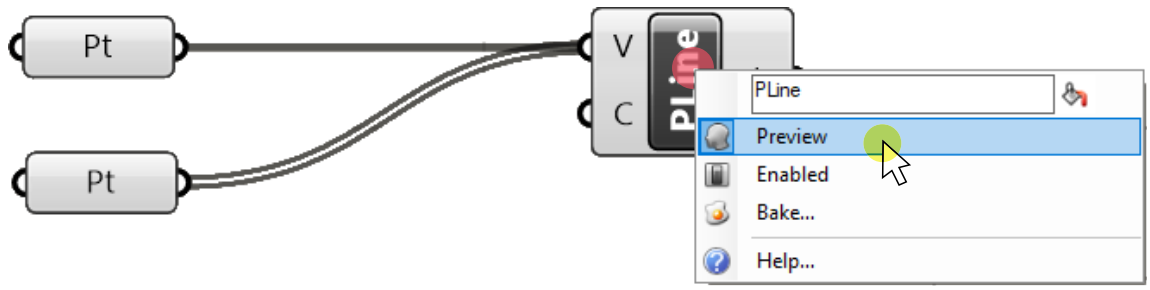
Here we see four spheres. On top, under “Default Template Materials”, are the default preview colors and on the bottom (“Document Materials”) the ones of the current script. We can change the default color and transparency by right-clicking on the spheres. If you’ve customized the preview in the current document, you can also click- and drag the colors from the bottom spheres.

Turning the preview of Components on and off

By default, any component we add to our script will be previewed in the Rhino viewport. While this works as long we only have a handful of components, once our scripts get more advanced, all the overlayed previews will make it impossible to see what's going!

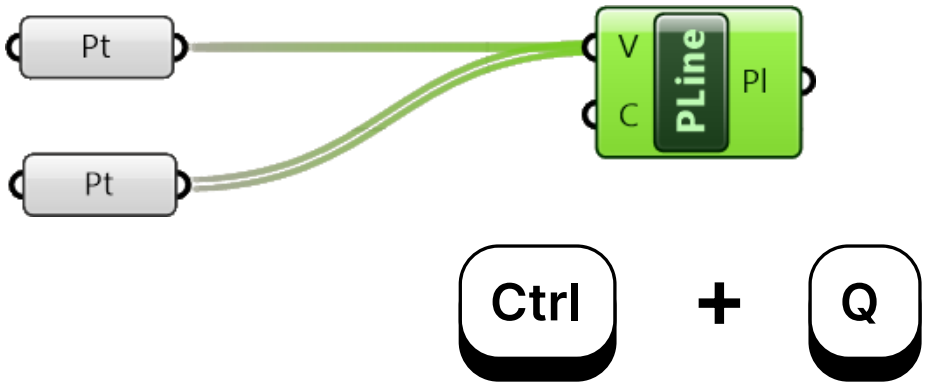
It's the equivalent of displaying all the intermediate steps leading up to a final geometry, when all we want to focus on is the final output!

To help with that, we can **disable the preview** of individual components. We can disable the preview by right-clicking on components and toggling the "Preview" option in the pop-up menu.



To differentiate components that are previewed from those that aren't, components with deactivated preview are shown in a darker shade of gray.

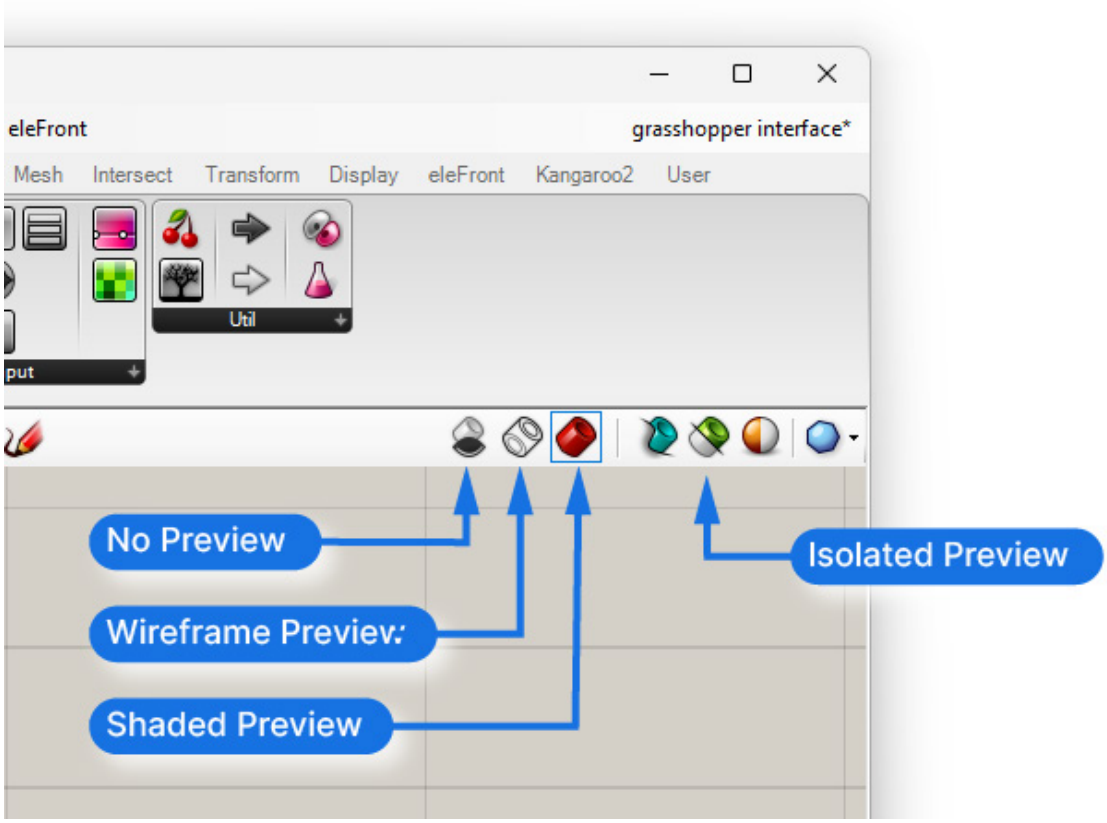
We can turn off the preview of multiple components at once, by selecting them and pressing **Ctrl + Q**. This shortcut will **toggle** the preview, meaning that if we press Ctrl + Q again, the preview switches back on.



Preview Modes

In some cases we'll want to quickly turn off the preview of all components or only preview a single component to check if it works as expected. Turning the preview of all components on and off manually would be a bit awkward. Luckily Grasshopper offers additional preview modes.

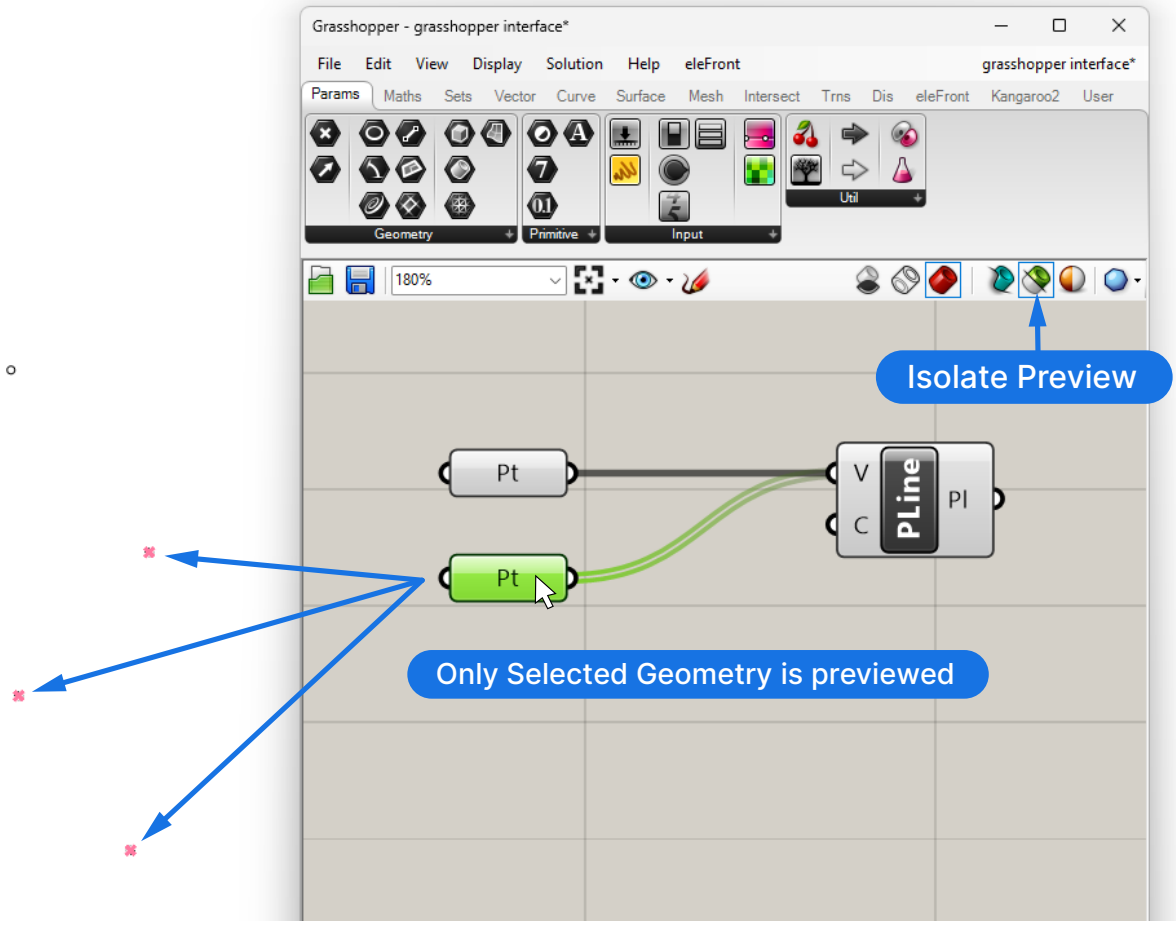
We change how the preview of Grasshopper geometry is displayed in the Rhino viewport with the buttons on the top right corner of the Grasshopper canvas.



The first three buttons control how the preview will be shown in the viewport, much like the Display Modes in the Rhino viewport.
The first one will disable the preview of the entire script.
The second one will show the preview geometry in wireframe mode, just like the Wireframe view in the Rhino viewport. And the third, red one, which is the default, will give us a normal, shaded preview.

To isolate the preview of specific components, we can activate the isolate preview mode - the half-green- half-white cylinder.

Once pressed, the preview in the viewport will only display the preview of the components currently selected on the Grasshopper canvas. This will be helpful in more elaborate scripts to understand which component contains which geometry.

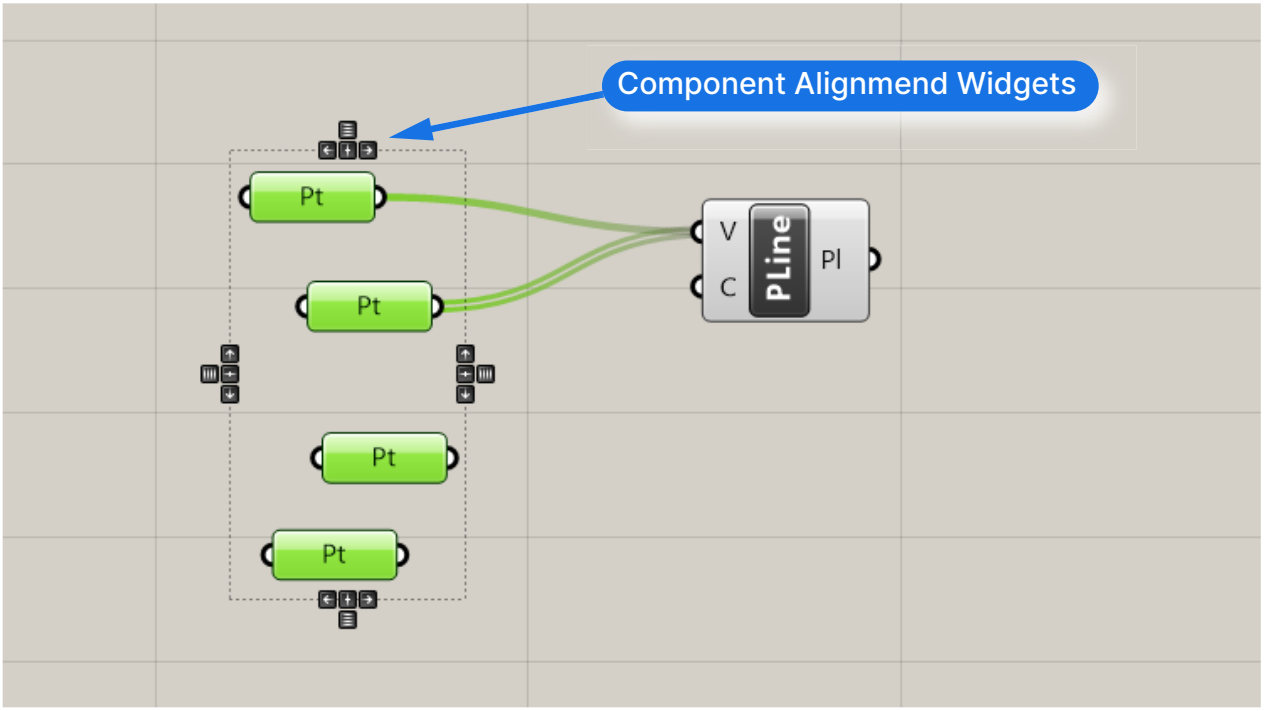


Click the same ‘Isolate Preview’ button once more to disable the isolate preview mode.

Keeping Your Grasshopper Definition Organized

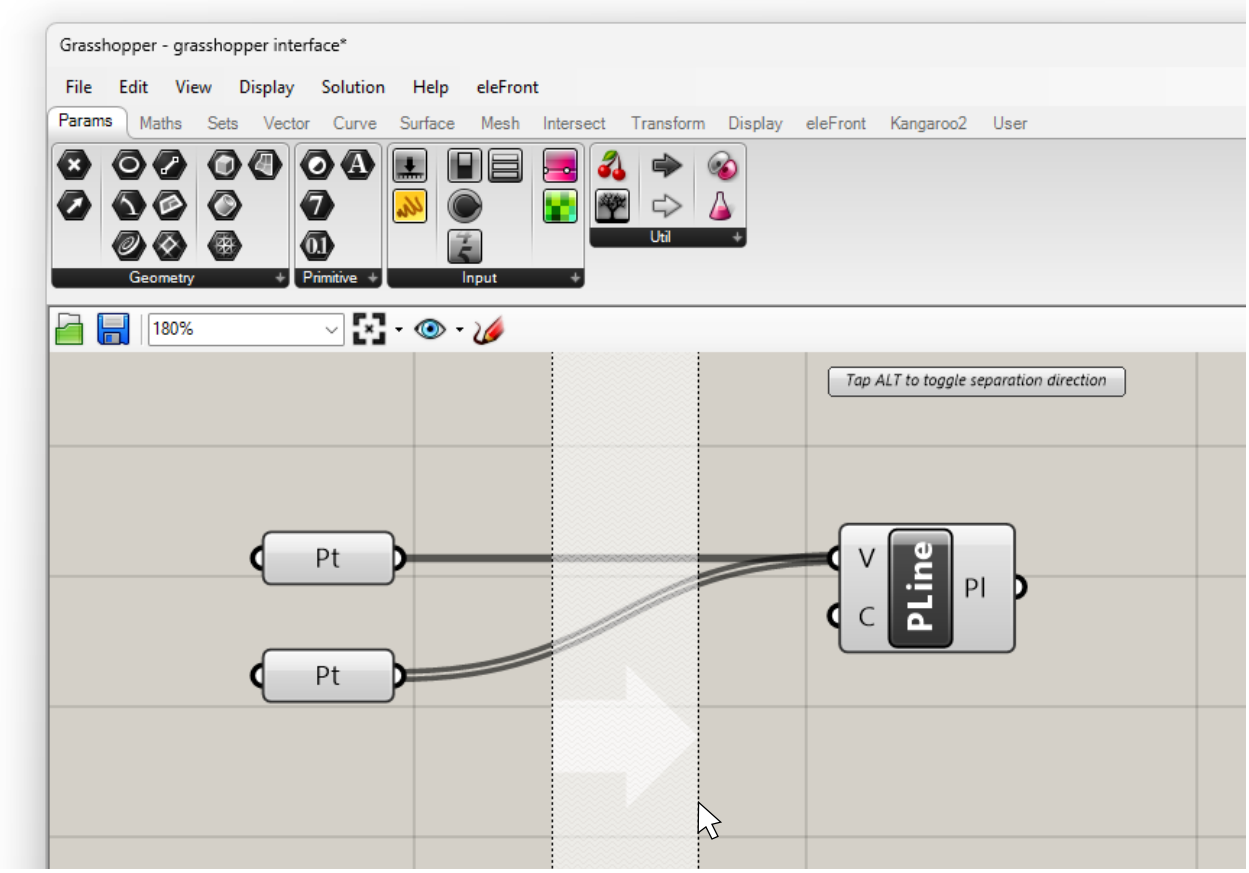
A drawback of the Grasshopper interface is that without some house-keeping, we risk ending up with a jungle of components and wires. To make sure we (and others after us) can makes sense of the script, it's a good idea to take some time to align and organize components.

When we select more than one component, a dashed bounding box with small black **alignment widgets** appears. We can use the arrows to align the components along one edge and space them out equally.



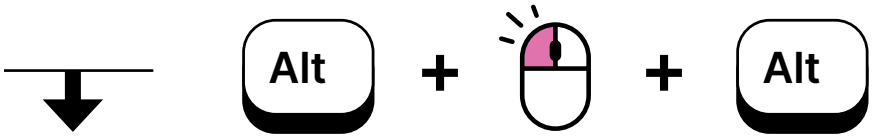
We can create space to add more components in-between existing components by holding down the **Alt** key and then clicking and dragging. This will shift all the components horizontally, to the right or left.

To use the same command to shift components vertically, press and hold the **Alt** key, start dragging, then hit the **Alt** key once again. This way you can shift all the components to up or down.



Shift horizontally:

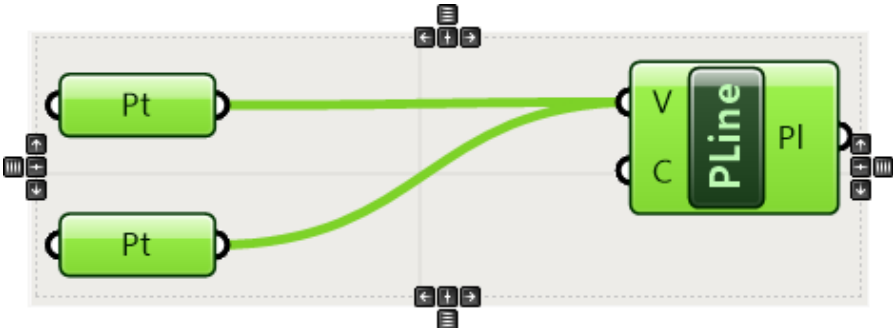
Shift vertically:



Grouping Components

It's best practice to group components and label them to clearly mark the different sections of a script within the Grasshopper interface. That way it will be easier for yourself and others to understand the different parts of a script.

To **group components**, select the components you want to group, and hit **Ctrl+G**.



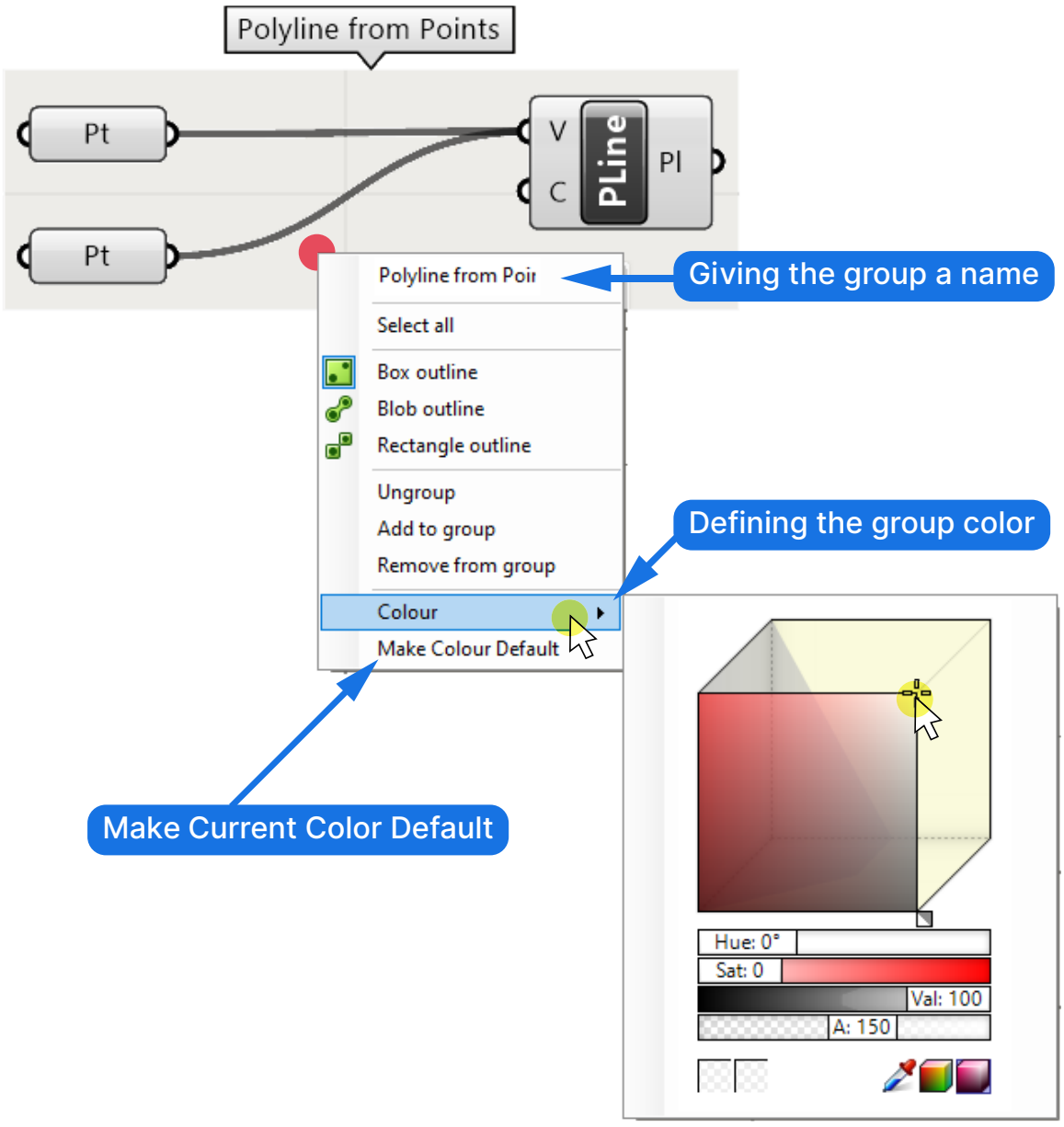
This will visually group the components. Click anywhere in the group and click and drag to move the entire group.

To delete the group, simply select it and hit 'delete' on your keyboard.

Customizing the Group

By right-clicking anywhere into the group we can add a **descriptive name** in the top field of the pop-up menu. We can also change the **color of the group** from the default yellow to a custom color, by right-clicking, going to 'Color' and adjusting it there.

To make the selected color the default color for any group you create in the future, right-click in the group once more and select ‘Make Colour Default’.



And there you have it! You’ve now mastered the Grasshopper interface, and you’re now equipped with the knowledge to navigate, create, and manipulate your own scripts.

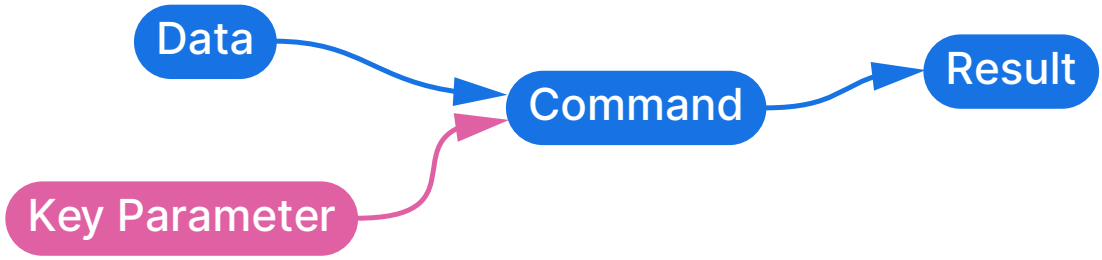
Next, we’ll learn how to define and set parameters!

3. Grasshopper Basics: Creating a Parametric Model

Now that you've mastered the Grasshopper interface, it's time to dive into creating your first script. While Grasshopper boasts a wide array of components, it's essential to understand that parametric design isn't just about connecting commands in a sequence. It's about building adaptable designs.

Parametric modeling involves using specific, variable **parameters** within our scripts. We need methods that allow us to easily interact with these parameters, ensuring our designs remain both interactive and adaptable.

On the other hand we don't want to specify too many parameters, as it would require too much manual adjustment. To avoid that, setting up **relationships** between different objects and their properties is crucial for a cohesive design.



In this chapter, we'll delve into the foundational tools in Grasshopper that make such agile adaptability achievable. We'll put a spotlight on essential components like the Panel, Number Slider, and Expression Editor.

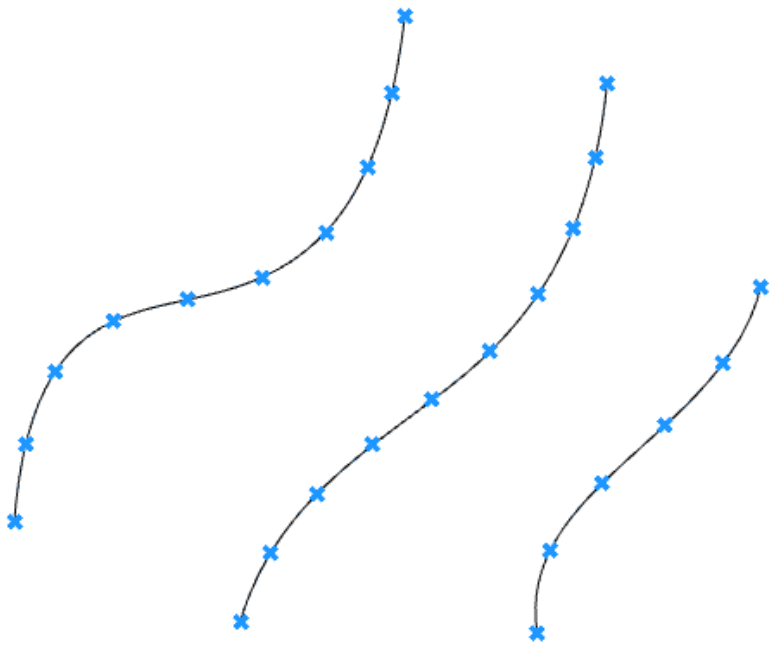
I'll guide you step-by-step using a simple example that illustrates the core idea of parametric modelling. By the end of this chapter, you'll have a hands-on understanding of how to build a parametric model and make your designs come alive with adaptability.

Let's dive in!

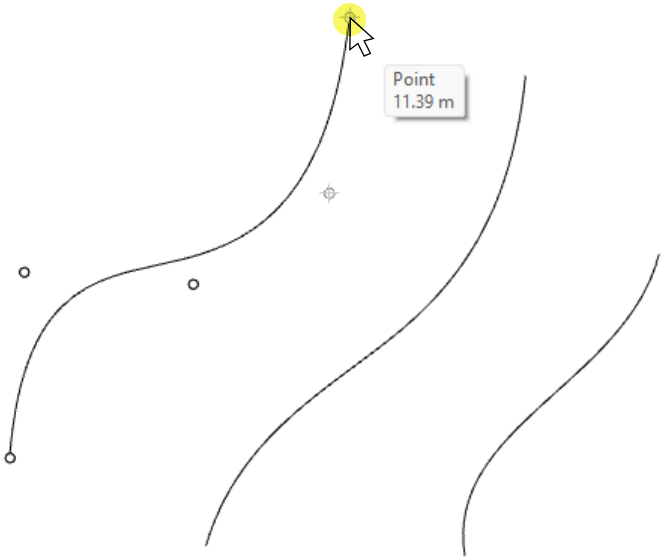
Our Project Goal: Crafting Dynamic Curves in Grasshopper

Understanding how to create a parametric model in Grasshopper is easiest when diving straight into a practical application. To showcase the concept behind parametric design, we'll construct a simple yet dynamic script that will divide curves of different lengths into equally spaced segments that are as close as possible to a target spacing we'll define.

Parametrically Subdivided Curves

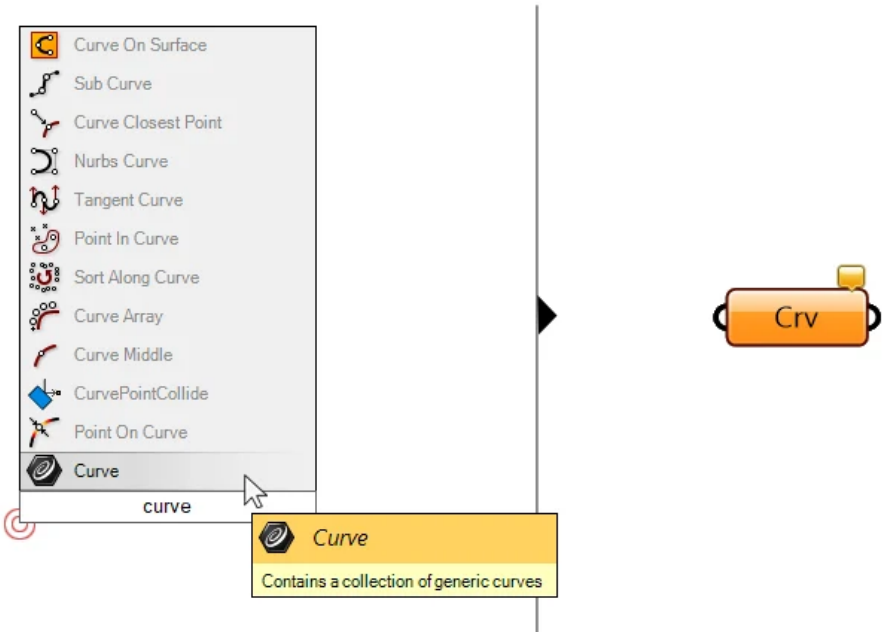


Let’s begin by sketching three distinct curves of varying lengths in Rhino. These curves will serve as the starting point of our script.

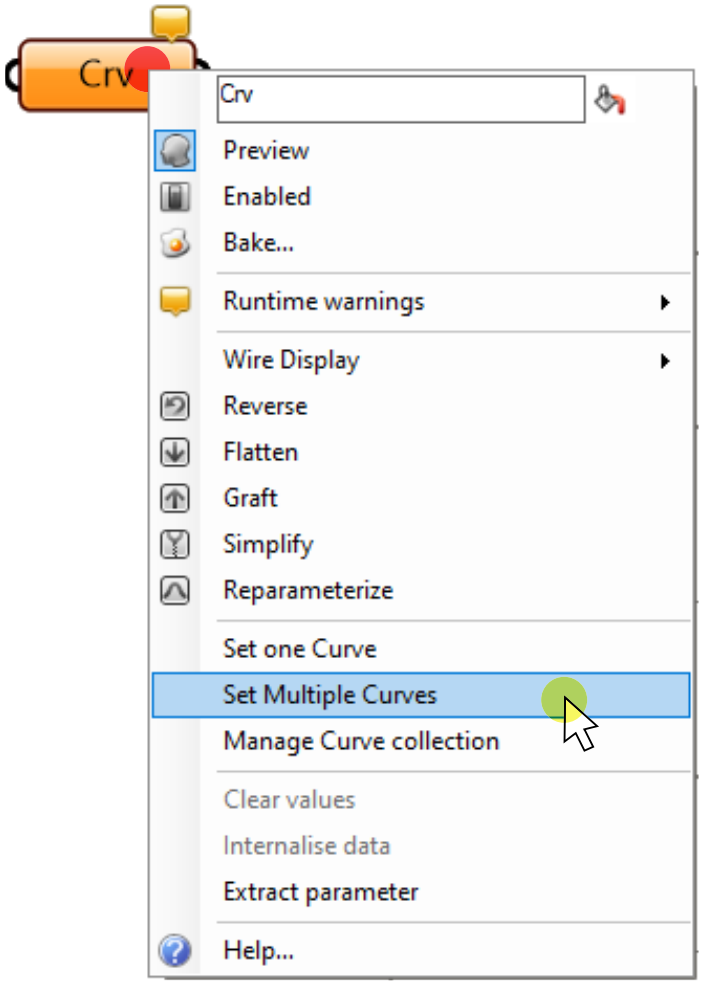


Next, let’s start up Grasshopper and reference these curves to a Curve container component.

Add a **Curve container** by double-clicking onto the Grasshopper canvas and typing “Curve”. Select the component with the black hexagon icon. If you hover over it, the tooltip should say: “Contains a collection of generic curves”.



To reference the curves to the Curve container, right-click on it, and select **Set Multiple Curves**. Then select all three curves in Rhino and confirm your selection with a right-click.



The Curve component should turn gray to show that it now contains data.

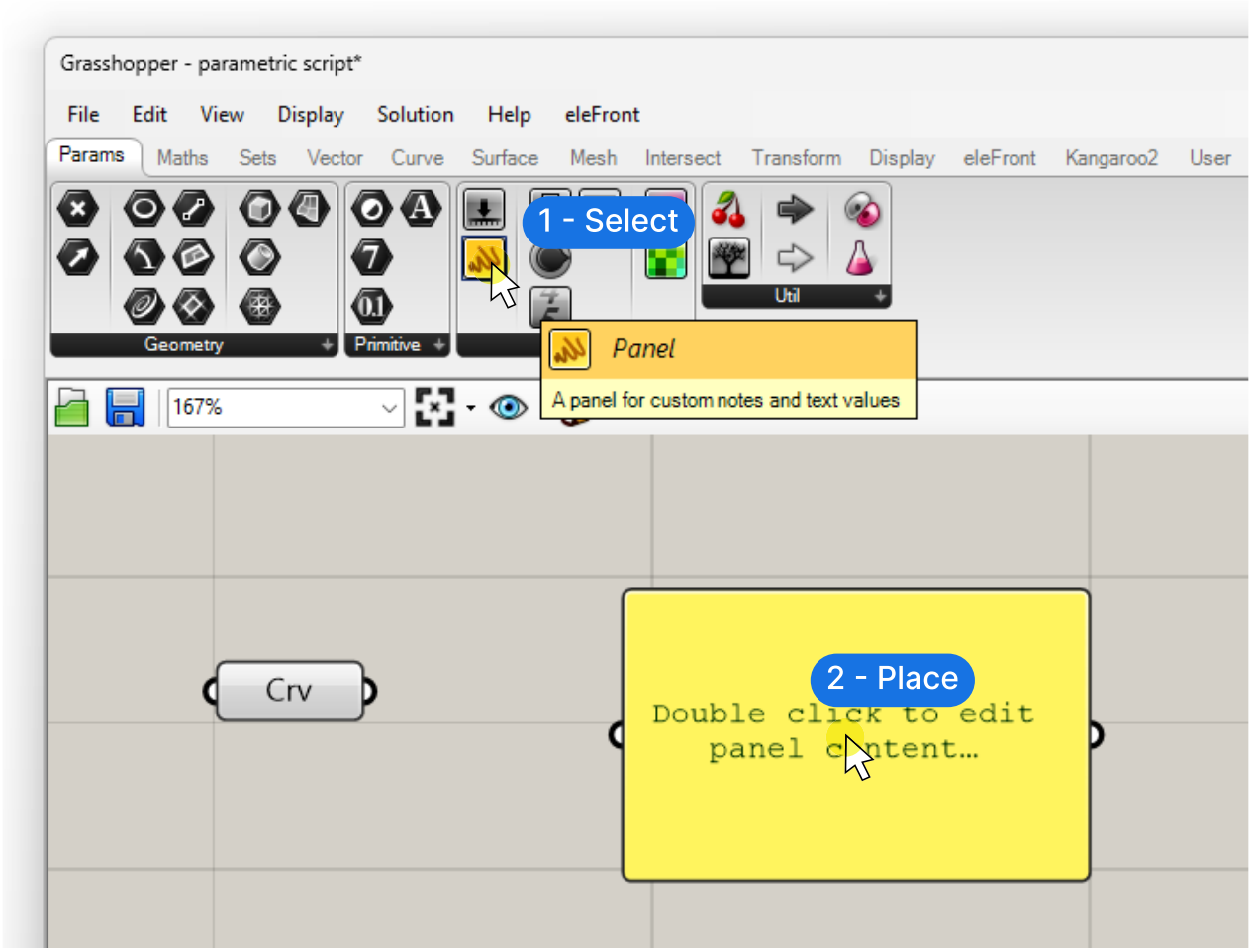
When we select the Curve component, the referenced curves are highlighted in the Rhino viewport, confirming that they are successfully linked. But there is a more precise way to check the output of a component: The Panel.

The Panel: Your Diagnostic Tool

The Panel allows us to see the contents and data structure of a component's output, in text form. So while we can see the preview of our selected Curve component in the viewport, the Panel allows us to double-check the exact number of items, and their type.

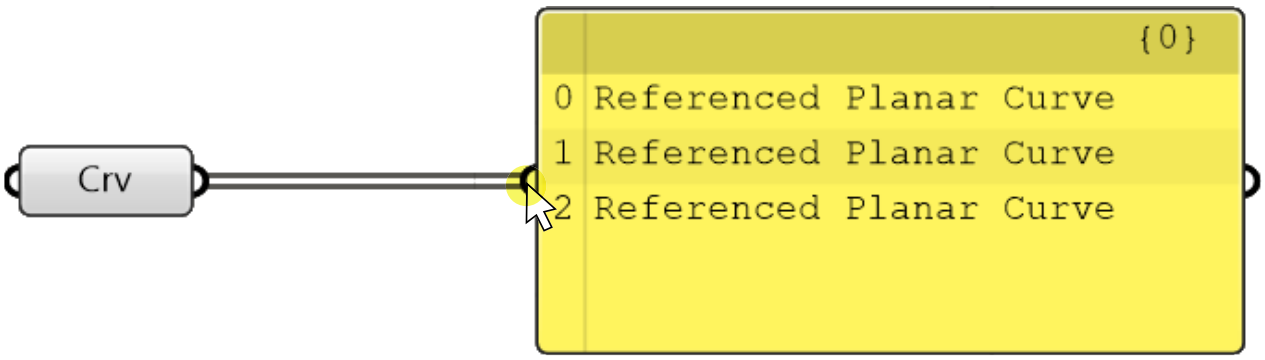
The Panel is an important part of our diagnostic toolset when building our scripts in Grasshopper. We can use it to ensure that the components output what we expect them to. We'll be using the Panel throughout our scripts to check data before connecting it to other components.

Now let's add a **Panel**, to see the output of our Curve component. We can select it from the standard component tab, it's the yellow icon, or we type "Panel" into the component search bar.



The Panel is like a simple text field and it has only one input and one output. To **display the contents of a component**, simply connect it to the Panel's input. Let's connect our curves.

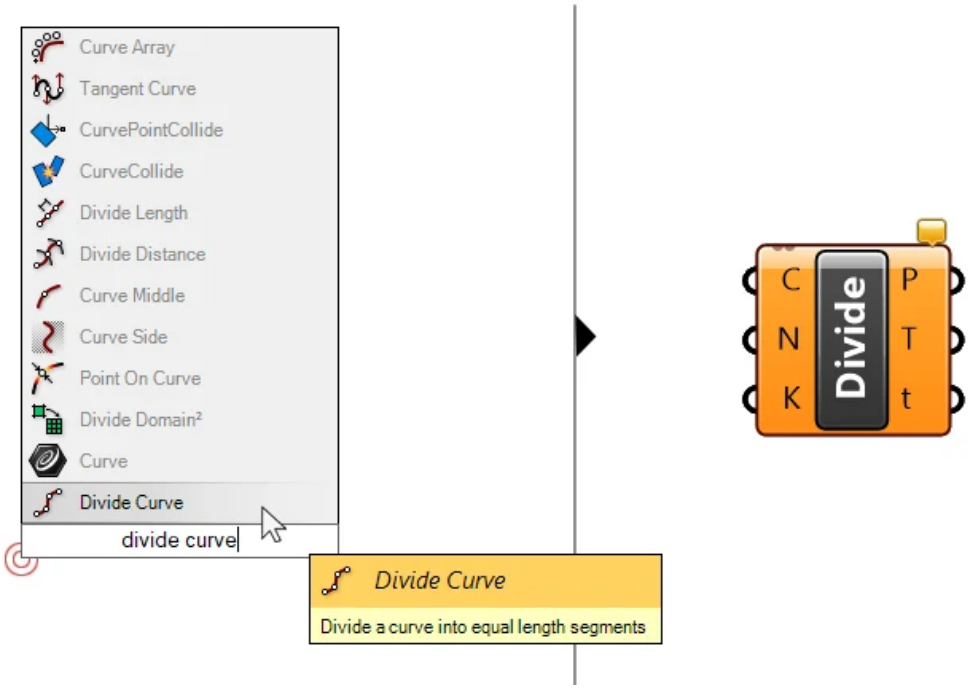
As soon as we connect a component, the Panel turns into a table and displays all the items the component contains in a list. A brief description of each element reveals the datatype. In our case, the text reads "**Referenced Planar Curve**".



Great! We've confirmed that our Curve component does in fact contain three curves, so let's move on to the next step: dividing the curves.

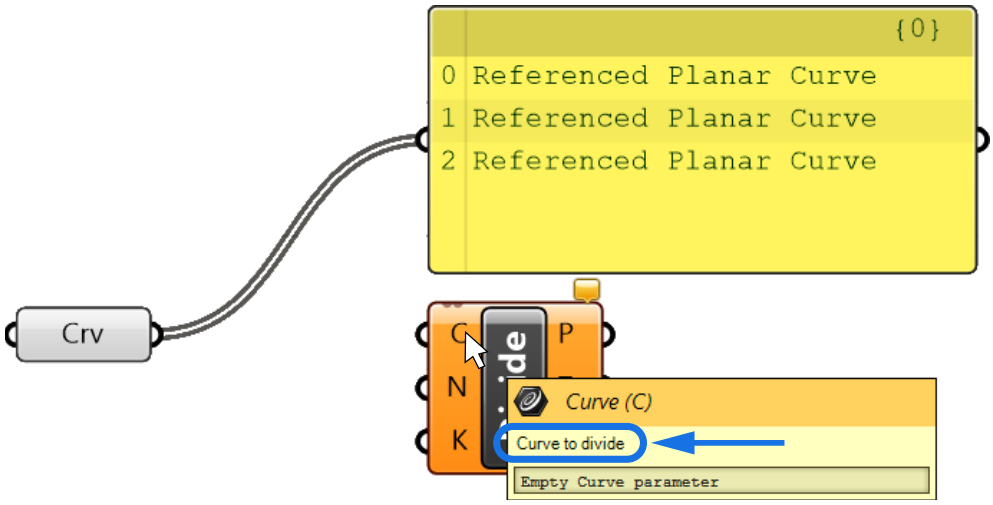
Dividing Curves in Grasshopper

Let's start by adding a **Divide Curve** component. We double-click onto the canvas where we want to component to be placed, type "Divide Curve", and select the first result.

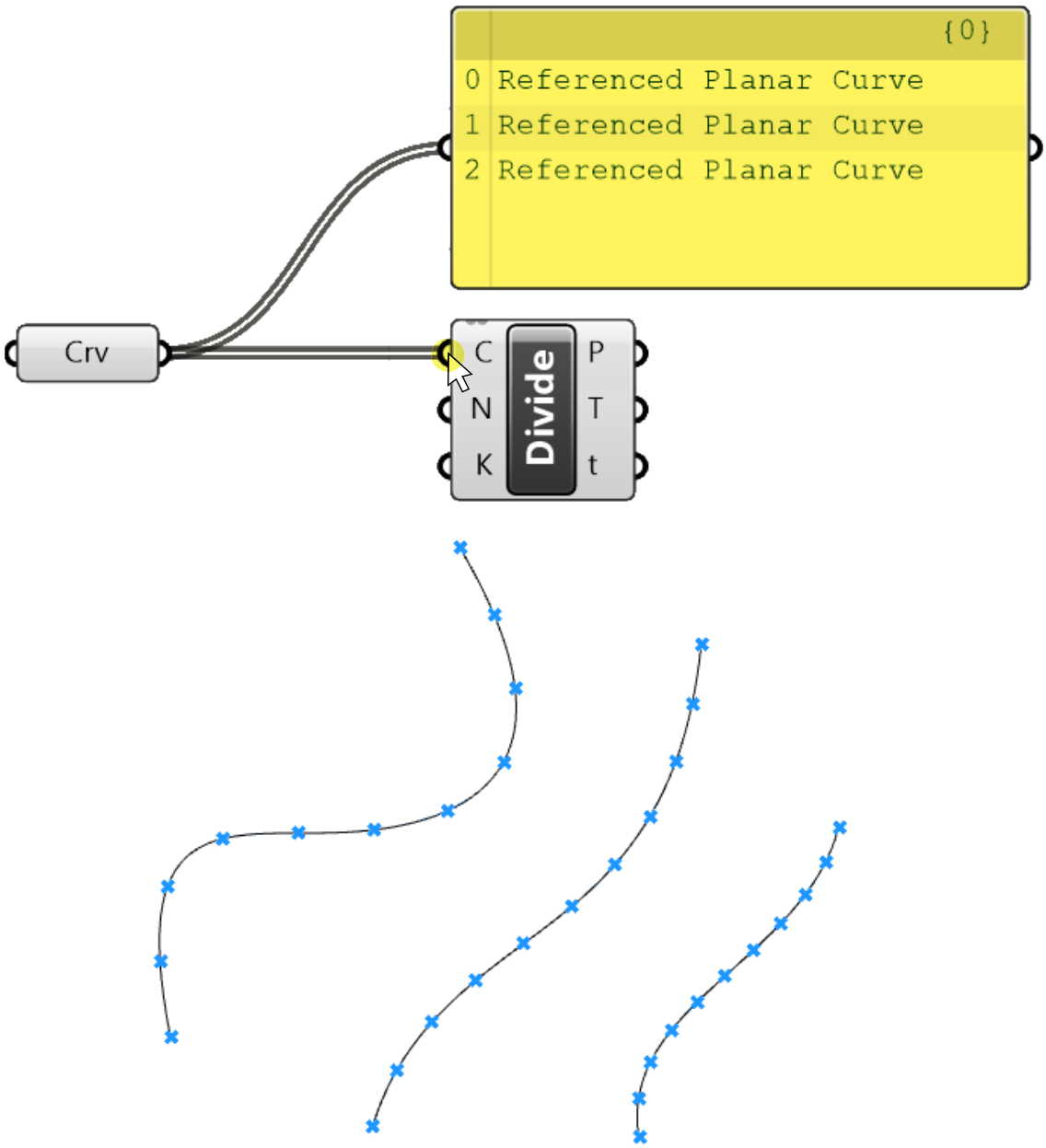


The Divide Curve component creates a specified number of equally spaced points on the curve and outputs additional information about the curve at those points.

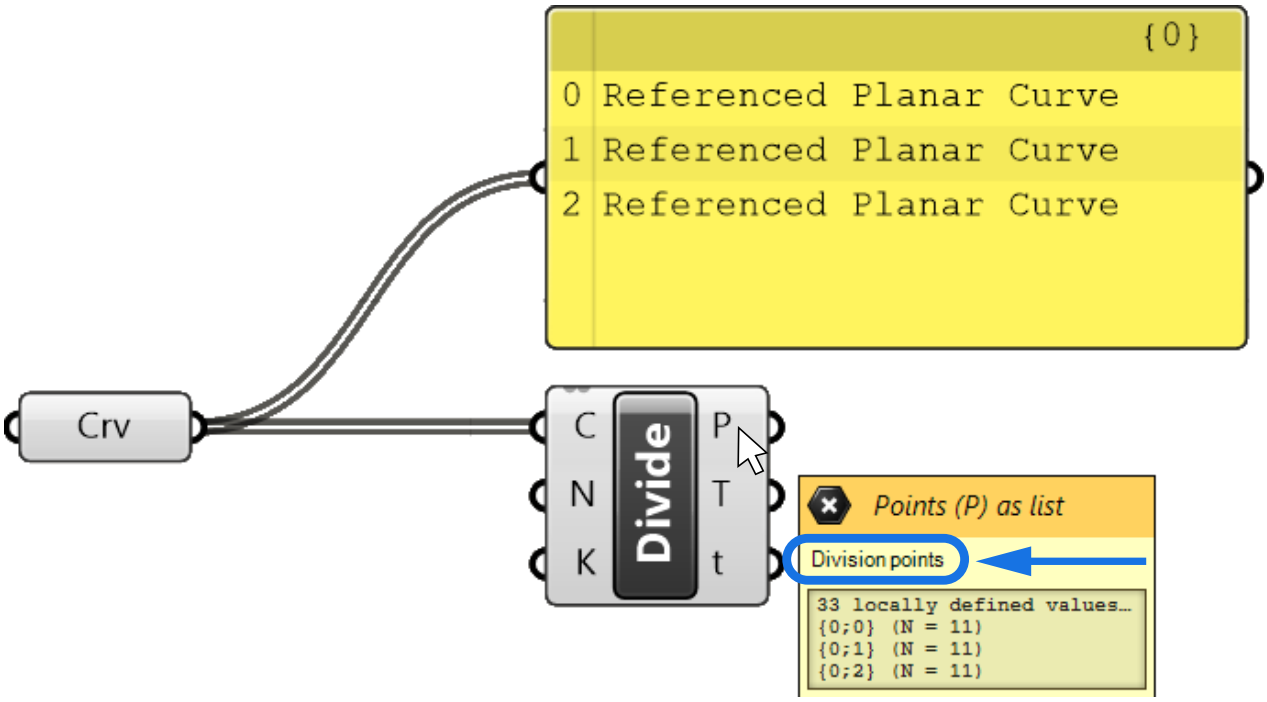
Let's learn about the required inputs by hovering over them: The first one, 'C', is the Curve to divide. Let's connect our Curve container output.



Make sure to create the connection between the actual curve container, and not the panel. The output of the panel is not the curves themselves, but exactly what's shown in the panel: a list of text items.



We see that division points already show up on our curve, that's because the **default value** of the next input N, which is the **number of segments**, is 10. In a moment, we'll add a number slider to dynamically adjust this number. But first, let's check out the outputs by hovering over them:



The first output is the **division points (P)**. These are simply points, equally spaced along the curve.

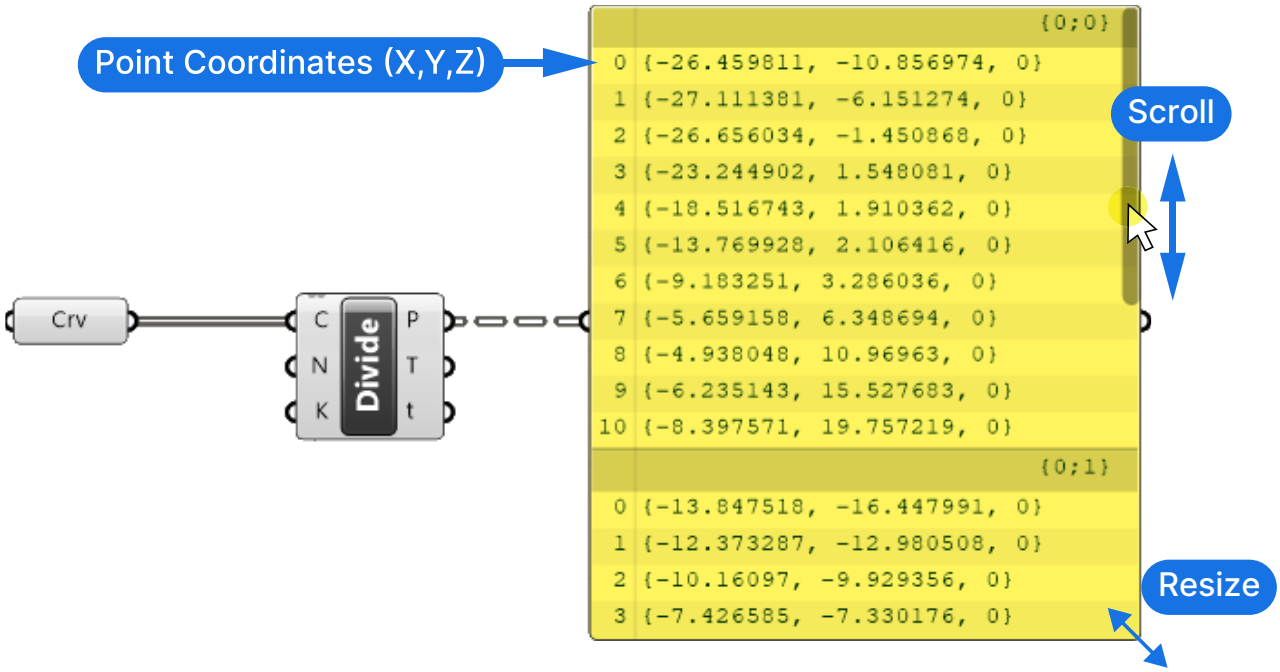
In addition, the Divide Curve component also outputs the **tangent vectors (T)** at the division points, and lastly the **curve parameters (t)** that describe the location of the division points on the curve. We could use that information to for example, split a Curve at those points.

For now we are only interested in the points themselves.

Let's re-use our existing Panel and connect the Points (P) output to see what it looks like.

Evaluating the result with the Panel

Since we generated a lot of points, the list inside the Panel exceeds the Panel boundary. We can resize the Panel to see more of the contents or scroll through with the scroll bar on the right hand side.



Because a Point is simply defined by its three X,Y and Z coordinates, which are just numbers, the Panel actually shows us all three coordinates of the Point instead of just writing “Point”, as it did with the curves.

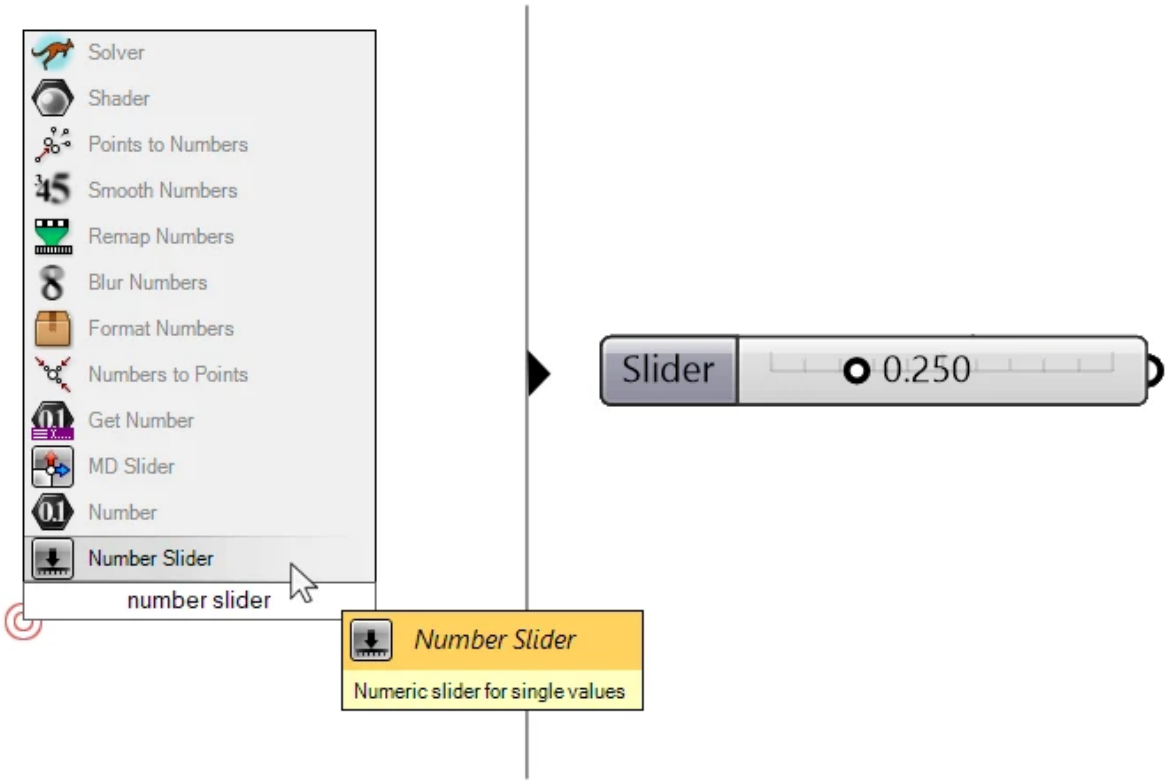
We divided each of the three curves into 10 segments, generating 11 points per curve (we get one more point than we get segments, because the component outputs both the start and endpoints).

The Panel shows the points in three sets of 11 points, each in their own **list**. This “grouping” into separate lists is Grasshopper’s way of keeping track which points belong to which curve. In more advanced scripts, this so-called data structure will become important for controlling which objects are processed together.

Let’s add a number slider to interactively adjust the number of divisions to make our script more dynamic!

Number Slider: Adjusting Design Parameters

Building a parametric model in Grasshopper is only useful if we are able to interact with its parameters. In Grasshopper, the **Number Slider** is one of the primary tools to do so. Let's type "Number slider" into the component search bar, and select the component to add it.

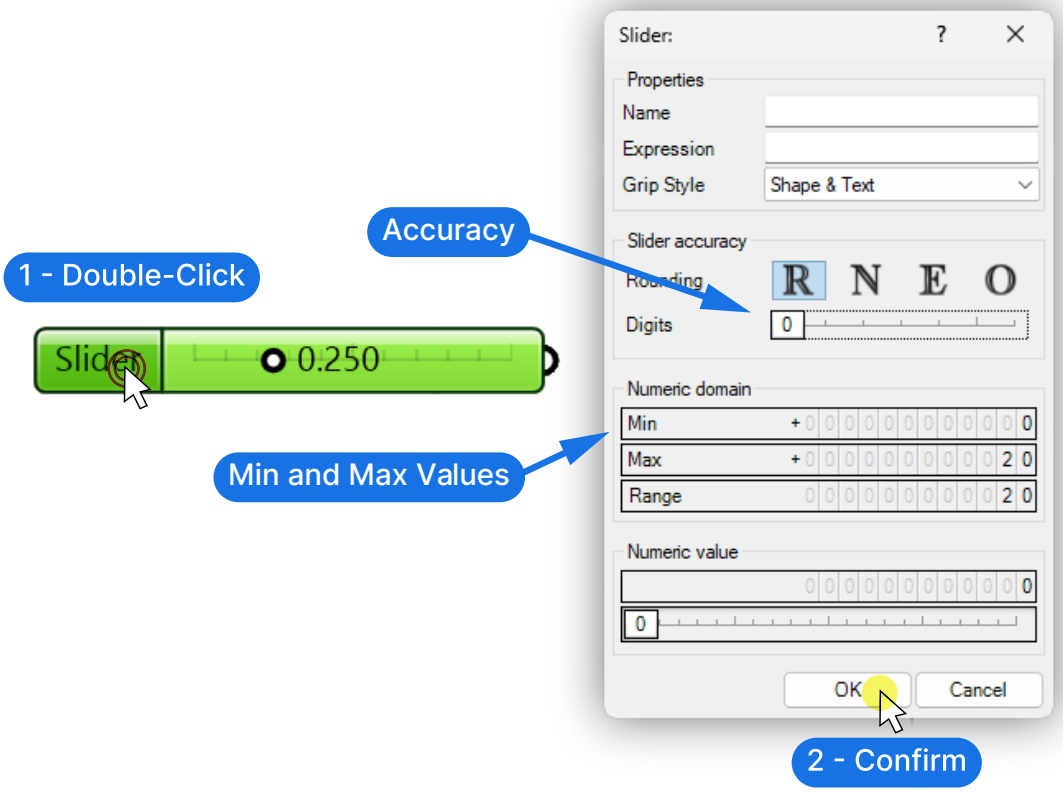


As the name suggests, we can "slide" the dot to change the output value of the component.

When we drop the number slider onto the canvas this way, the slider will come with a preset value of 0.250 and a preset range between 0 and 1.

We can change the **range** and **precision** of the number slider by double-clicking onto the left, dark gray part of the component.

In the menu that pops up, we can define the domain or minimum and maximum value of our slider. We can change the values by double-clicking into the fields and typing a number.



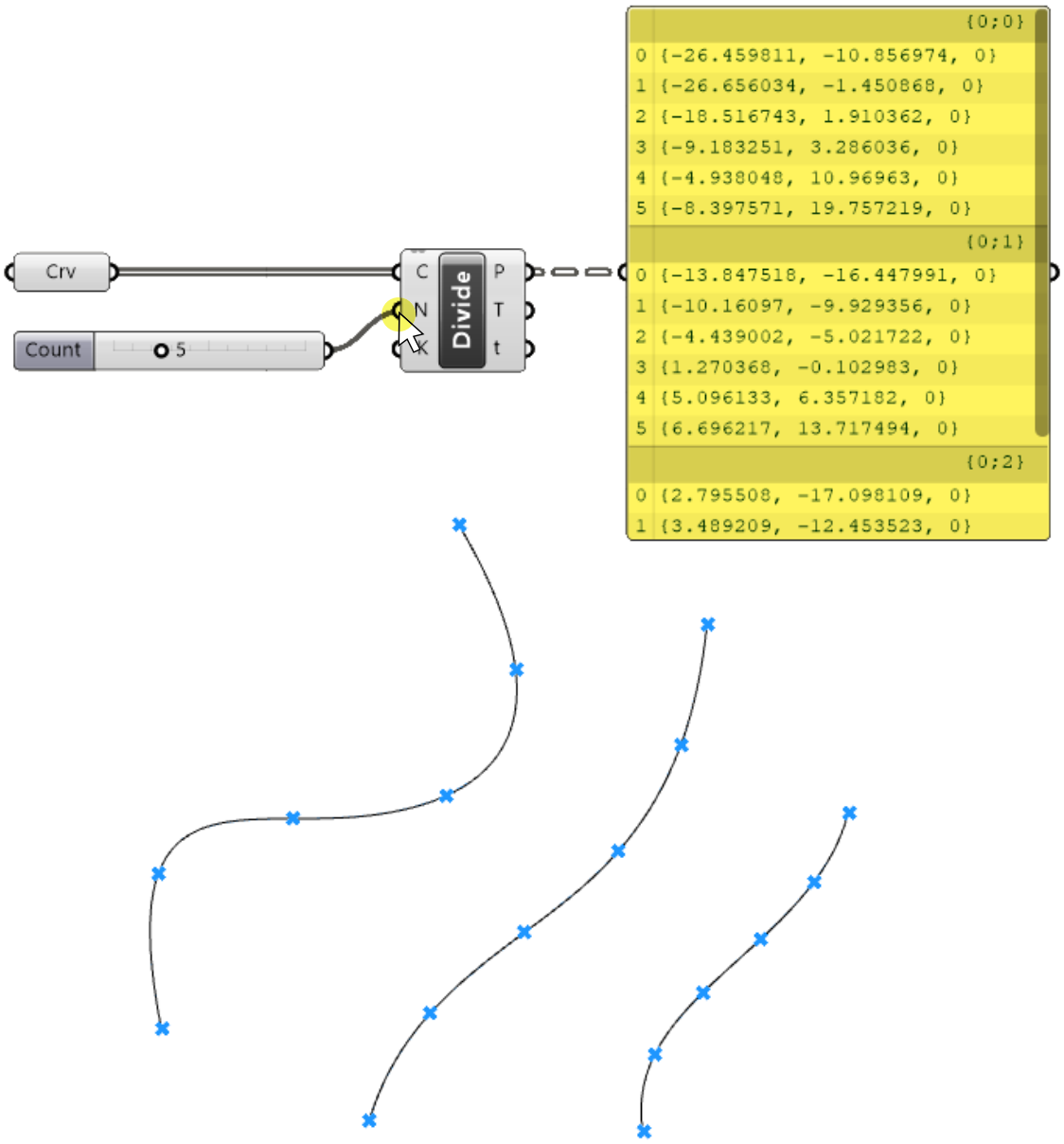
Let's change the minimum to 0, and the maximum to 20.

Above, we can adjust the slider accuracy by specifying the number of digits after the comma. Since we need whole numbers to define the number of segments, let's set it to 0. Let's hit OK to confirm.

The slider range and precision is now updated. To change the values, we simply move the slider by clicking and dragging. We can also manually set a specific number by double-clicking into the slider area and typing the number.

Let’s set it to 5 and plug it into the “Number of Segments” (N) input of the Divide Curve component. We can slide through the numbers and check in the preview if it works as expected.

As we move the slider, the Panel also updates to show the resulting output.

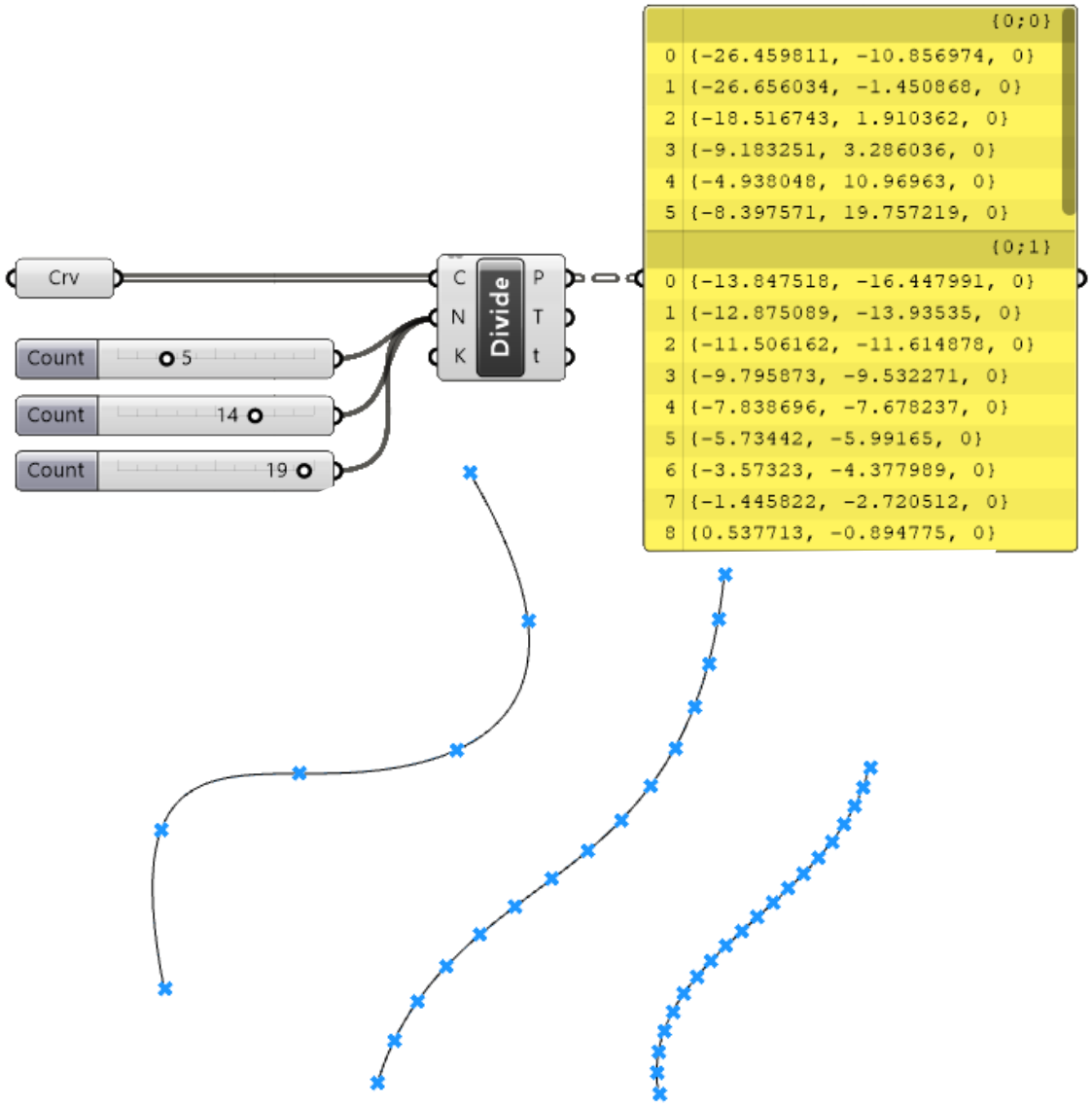


Advanced Panel Functions: Writing and Data

Our goal is to subdivide the curves into segments of similar length. That means that we'll need to specify a different number of subdivisions for each curve. Let's learn how to do that.

Since we have three curves, we need to provide three different numbers to the "Number of Segments" input.

One way to do it is to add two more Number Sliders and connect them all to the "Number of Segments" input. (Press Shift as you create the connection to add instead of replace a connection).



But we can also type the numbers out in a Panel. Here's how:

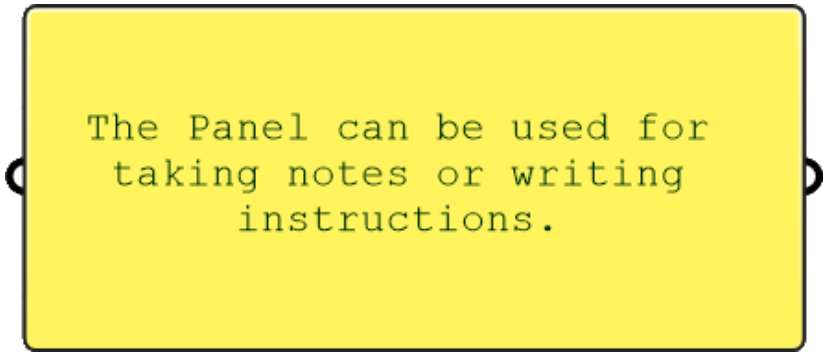
The Panel is not limited to just displaying the content of components - we can also use it to type in information. We can only do that if there is no input connected - if a component is connected, it will override our manual input.

Let's drop a new Panel component onto the canvas.

We can add text just like in a simple text editor by double-clicking on it. It can be used as standalone component to add notes or instructions in a script.

1 - Double-Click into the Panel

2 - Write the note



But we can also use it as a way to enter numbers that will drive our script. To do that requires an extra step.

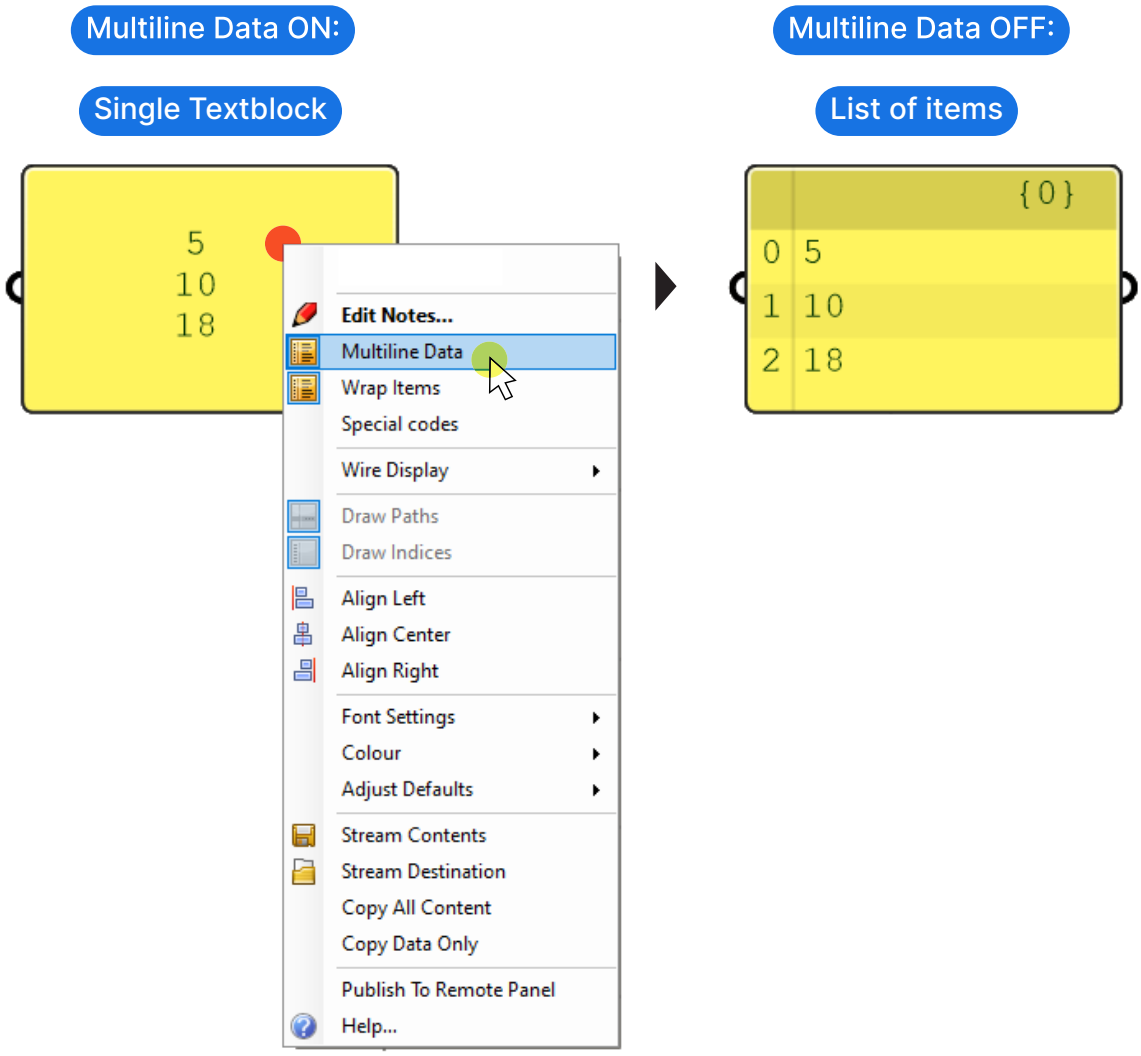
Using the Panel as input method

Let's start by typing three numbers followed by enter to get three numbers on three lines.

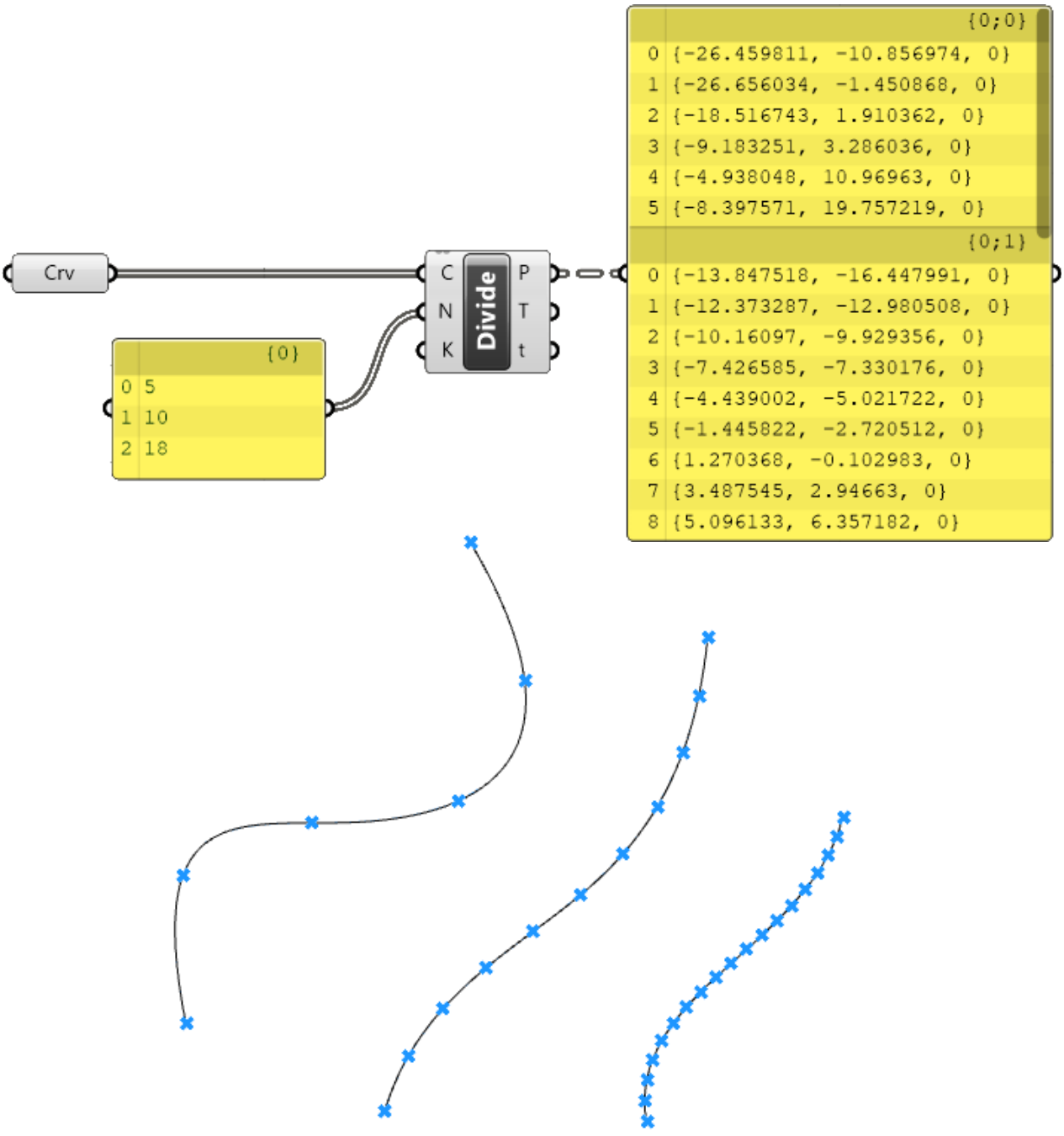
By default, anything we type into the Panel is considered a single text block, meaning one text object. Even if our numbers look like a list of three values, to Grasshopper they are still a single text block.

If we want each line to be considered as a separate value, we need to right-click onto the Panel and turn off Multi-line Data.

As you can see, the preview now shows the same content as a list of numbers.



If we connect these three numbers to the “Number of Segments” input of the Divide Curve component, we get a different number of division points for each curve, as we can also see in our Panel component.



We successfully specified a different number of divisions for each curve, but we still have to change the the numbers manually. Let’s find a way to specify the number of divisions parametrically, next!

Parametric Division: Adapting to Curve Lengths

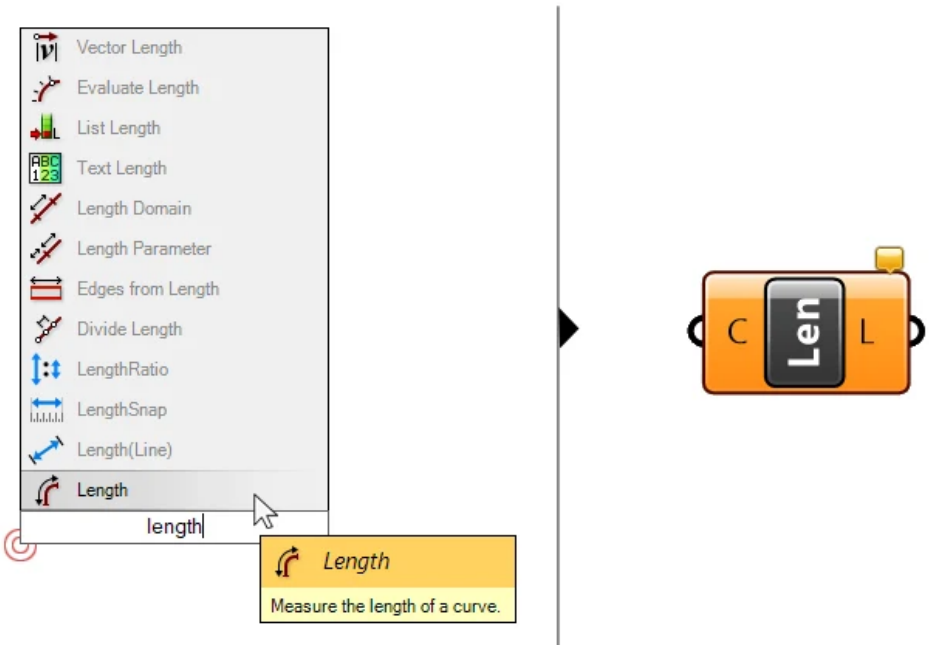
To create parametric relationships in our Grasshopper model, we need to take one property of an object and use it to define another. In our example, we want to create a **parametric relationship** between the length of the curves and the number of subdivisions. That way, we'll end up with a similar spacing between the points across all curves.

To turn this parametric relationship into components we need to further break it down into **logical steps**. In our case, we can specify a **target length** for the segments and divide the length of the curves by it. We'll get a number that we can round to a whole number to get the segments per curve.

Let's implement this logic.

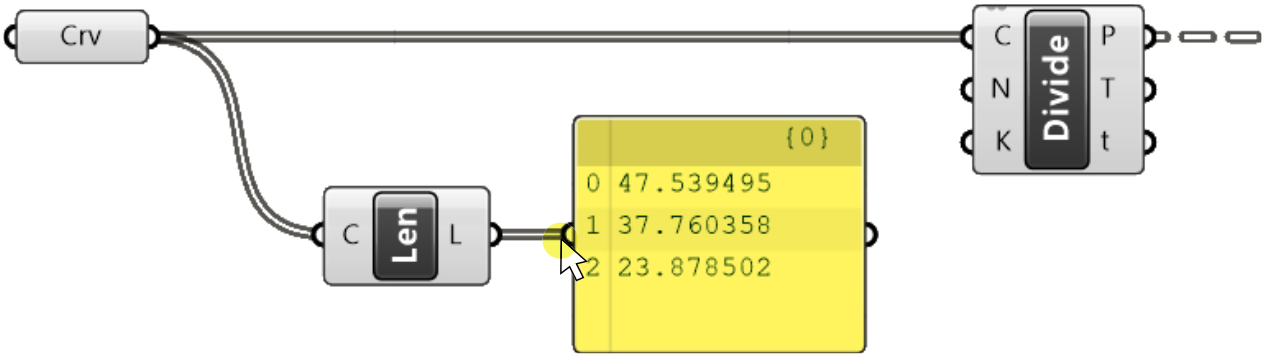
Extracting the length of the curves

First we need to get the length of our curves. We can use a component called "**Length**" for it. Double-click on the canvas to open the component search bar and type "Length".
There are many components with the word "length" in it. Make sure to pick the component that says "**Measure the length of a curve**" when you hover over it.



The Length component takes a curve as an input and, you guessed it, outputs the curve's length. Let's connect our Curve container.

The output of the Length component is a number. We can check and see the output by connecting it to a Panel component.



Next let's define our **target spacing**.

Advanced variable definition with the number slider

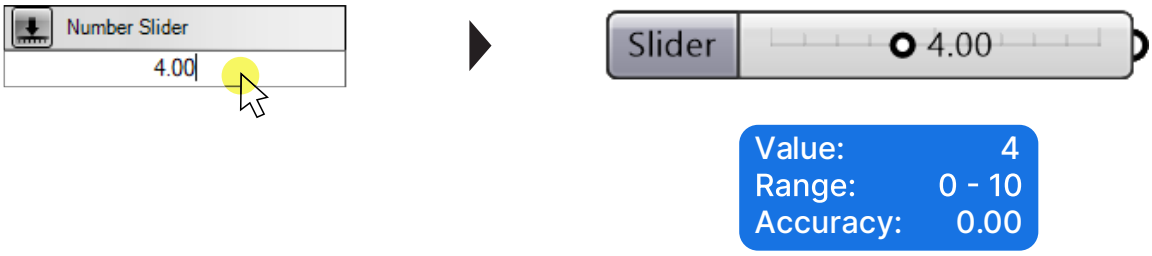
Let's add a new Number Slider that will define the target spacing of our Points on the curves. Let's use a more advanced method of inserting a Number Slider this time:

If we type a number directly into the **component search bar** and hit Enter, Grasshopper will create a slider for us, with that number as the value.

Depending on the number we type, Grasshopper will generate a range, or minimum and maximum value. This range depends on the value we enter. By typing 4 for example: you'll get a slider between 0 and 10.

We can also define the precision of the Number Slider this way.

If we type “4.00”, the slider will be between 0 and 10 and have a precision of 2 digits after the comma.



Let’s use this number slider to define our target spacing.

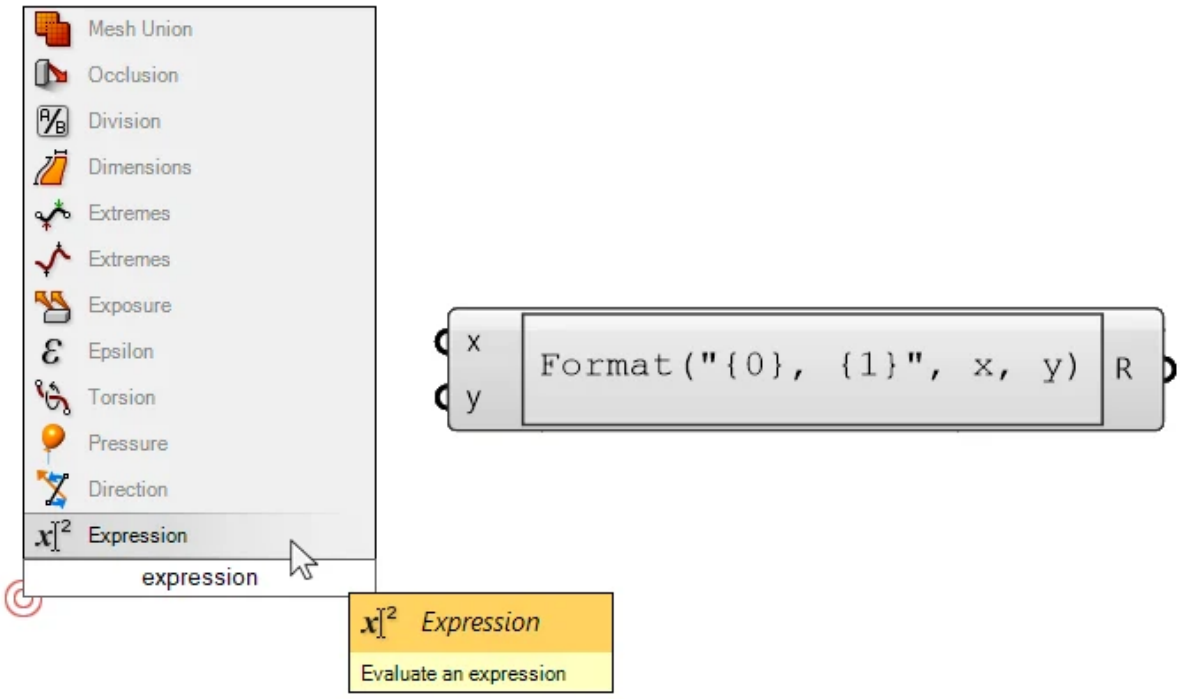
So, we have the curve length and the target spacing, next, we’ll define a parametric relationship between the two values!

The Expression Editor: Mathematical Operations Simplified

It’s time for some math! To create a parametric relationship between the curve length and the spacing, we’ll use a simple **division**. We’ll divide the curve length by the target spacing to get the number of segments.

Throughout our scripts, we’ll use **mathematical expressions** to control different object properties. In 99% of cases, it will be very basic operations: additions, divisions, multiplications, subtractions or a combination of them. No need for advanced algebra!

To enter any mathematical formula in Grasshopper, we can use the **Expression** component. Let’s double-click on the canvas and type “Expression” and add the component.

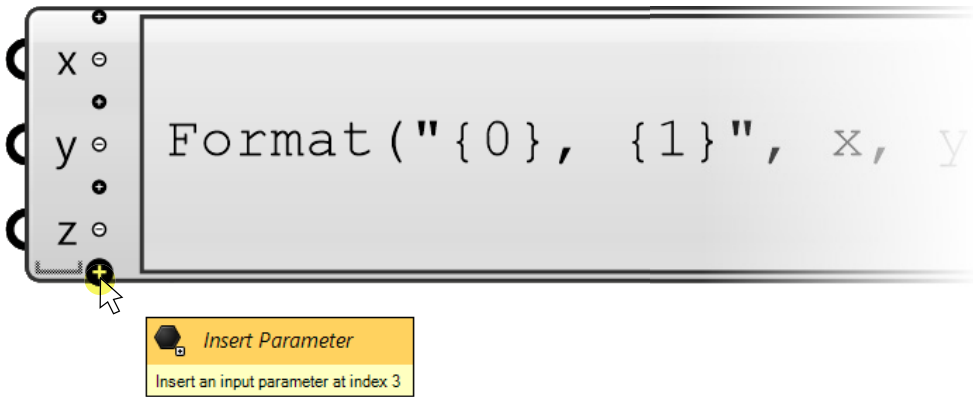


Don't mind the notation that shows up in the middle of the component: it is a template that we'll replace with our own formula in a second!

First let's look at the anatomy of the component: we have simple x and y inputs, which will be numbers, and one resulting number output (R).

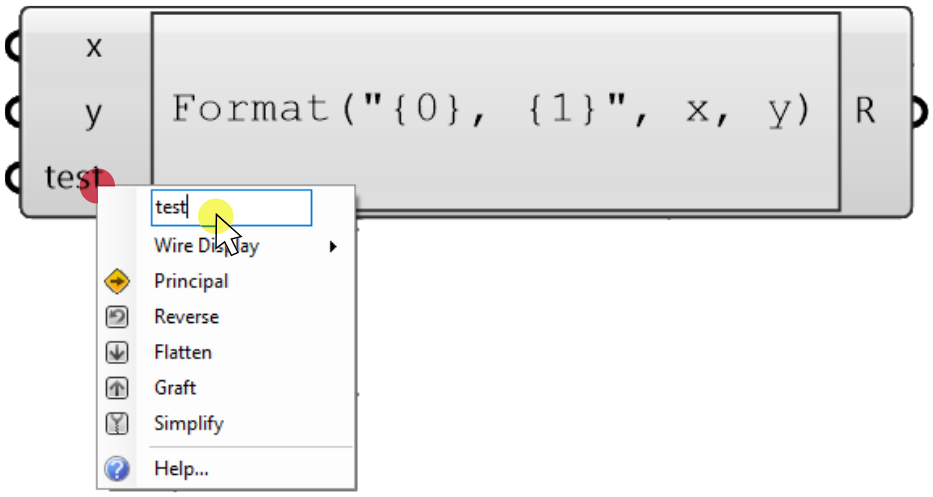
If we zoom in close to the inputs of the expression component, small plus and minus signs appear (you'll get a tooltip with a countdown the first time you do it).

Zoom in to Add & Remove Input Variables



By clicking the plus and minus symbols, we can add or remove inputs from our formula. These inputs will be the variables we can use in our formula. We can give the inputs any name we want by right-clicking on them, as long as we use the same name in our formula as well.

Specifying Input Variable Names



Although there are different mathematical components, we could for example use the “Division” component, I recommend to always use the Expression editor for mathematical operations. Instead of using ten different components, we can use one that can do everything. The main advantages are that we can easily type and modify any formula, that we see the formula directly, and that we can do multiple operations in one component.

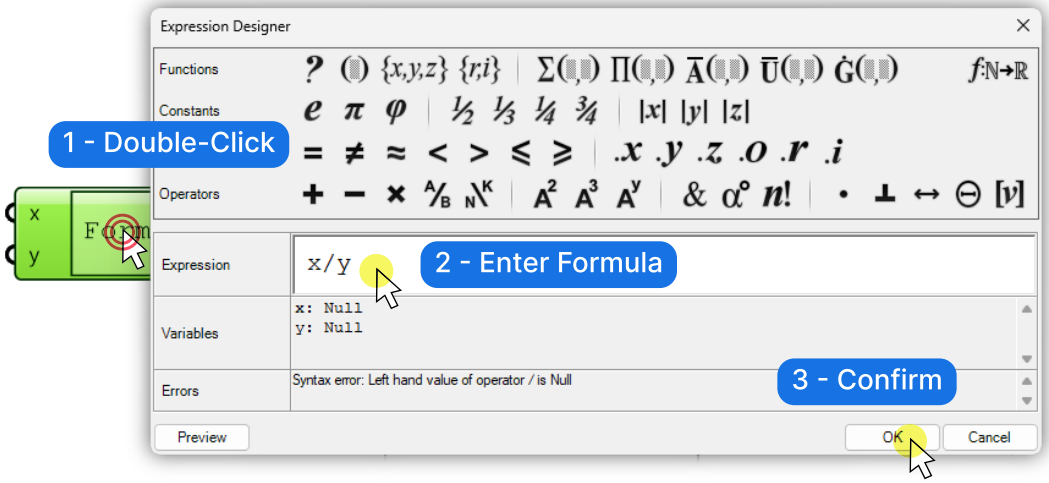
Now let’s add our formula!

Entering a formula in the expression editor

To open the expression editor, double-click on the central formula area. In the editor that pops up, we can define any mathematical formula we need. Let's delete the existing template and simply type

```
x / y
```

"x" will be the **curve length**, and "y" the **target spacing**.

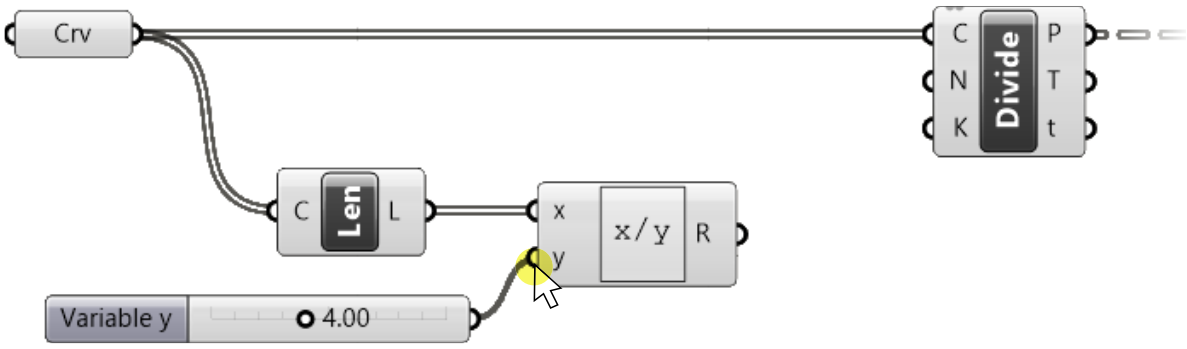


All the buttons above the expression input field can help us with typing in the formula, but for most cases, we can just type in the formula using our keyboard. Up in the top right corner we can access a list of additional commands with a short description. You can find the notation for Sine and Cosine etc. here, if you are feeling adventurous!

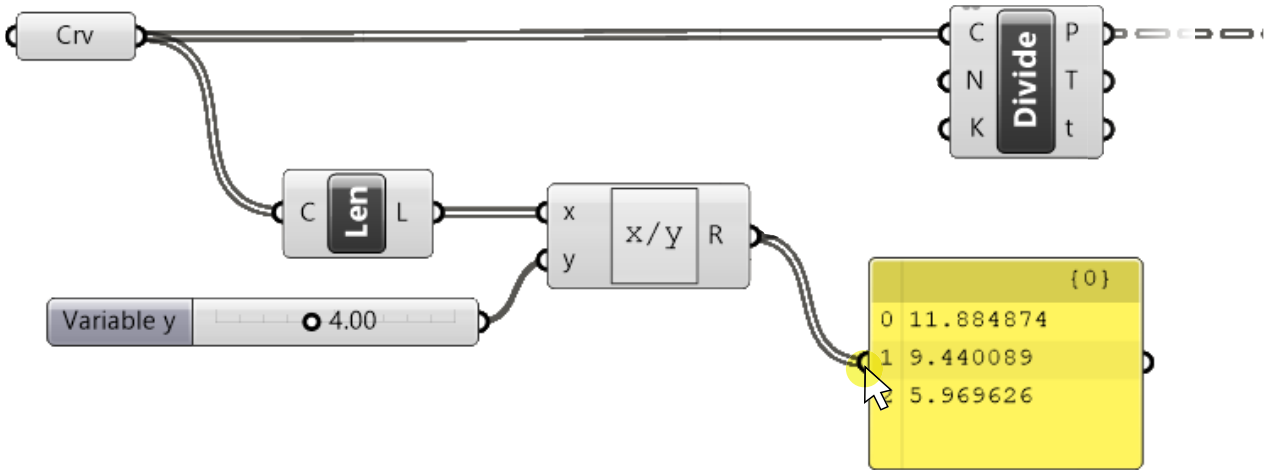
Once we are happy with our formula we confirm with OK.

When we confirm, the component turns red because we haven't provided any inputs yet. Let's solve that.

Let’s connect our curve lengths to the x input, and the Number slider to the y input.

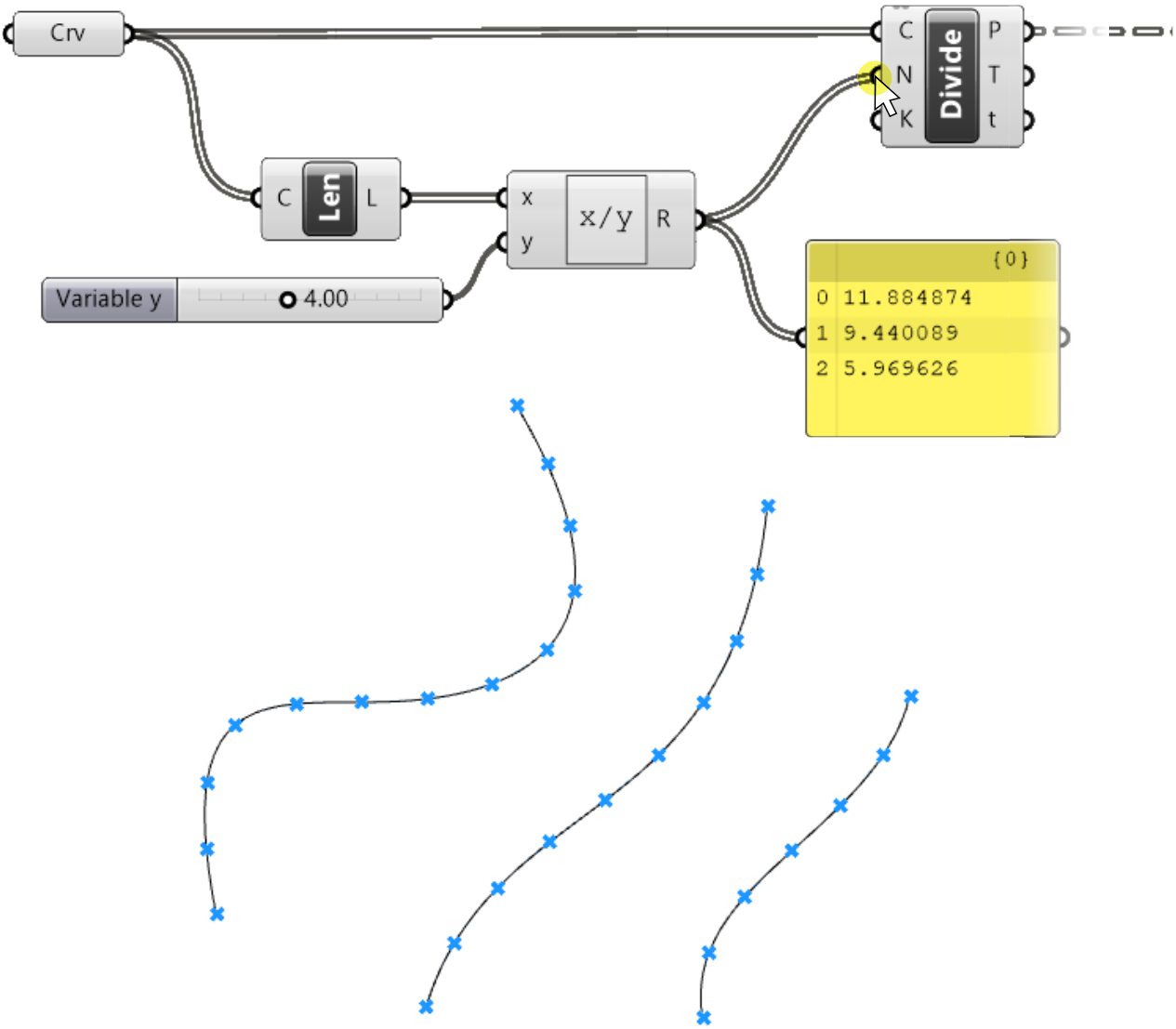


We can once again check the output by plugging it into our **Panel** component.



Our output is not a whole number because our curve lengths and spacing aren’t either.

Let’s plug this result into the **segment number (N)** input of the Divide Curve component.

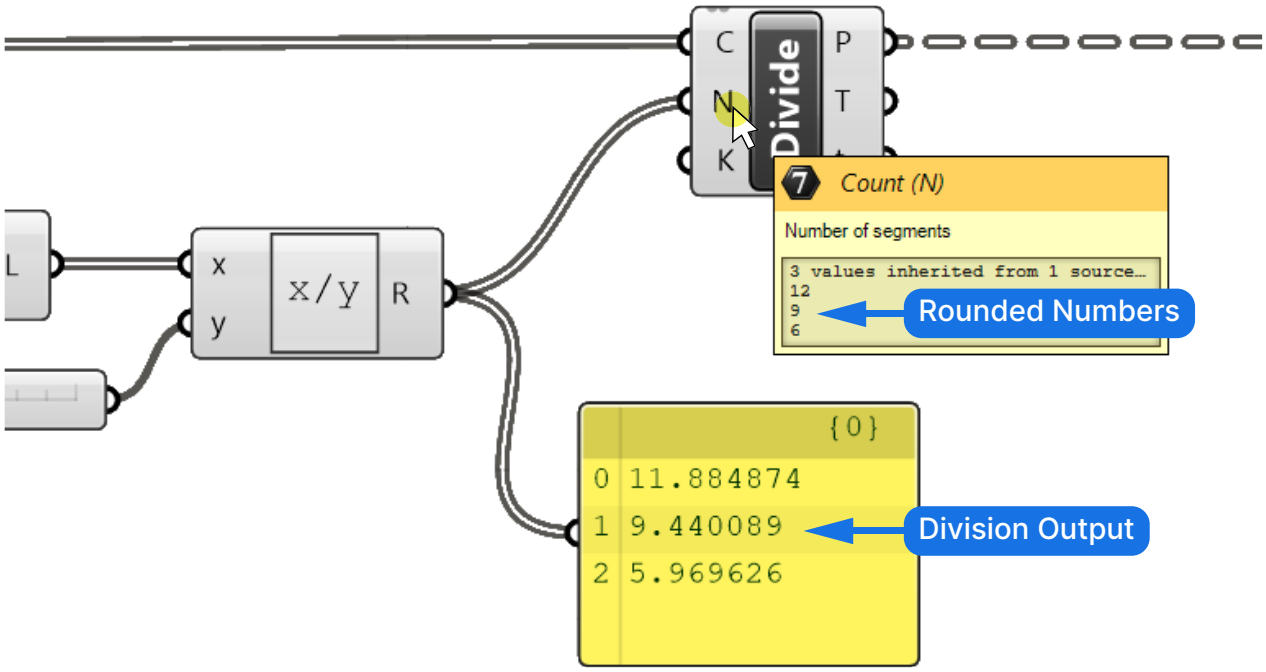


The curves are now parametrically divided into segments of similar length!

But the number of segments needs to be a whole number and the number we specified has several digits after the comma. Let's understand what happend.

Grasshoppers automatic datatype conversion

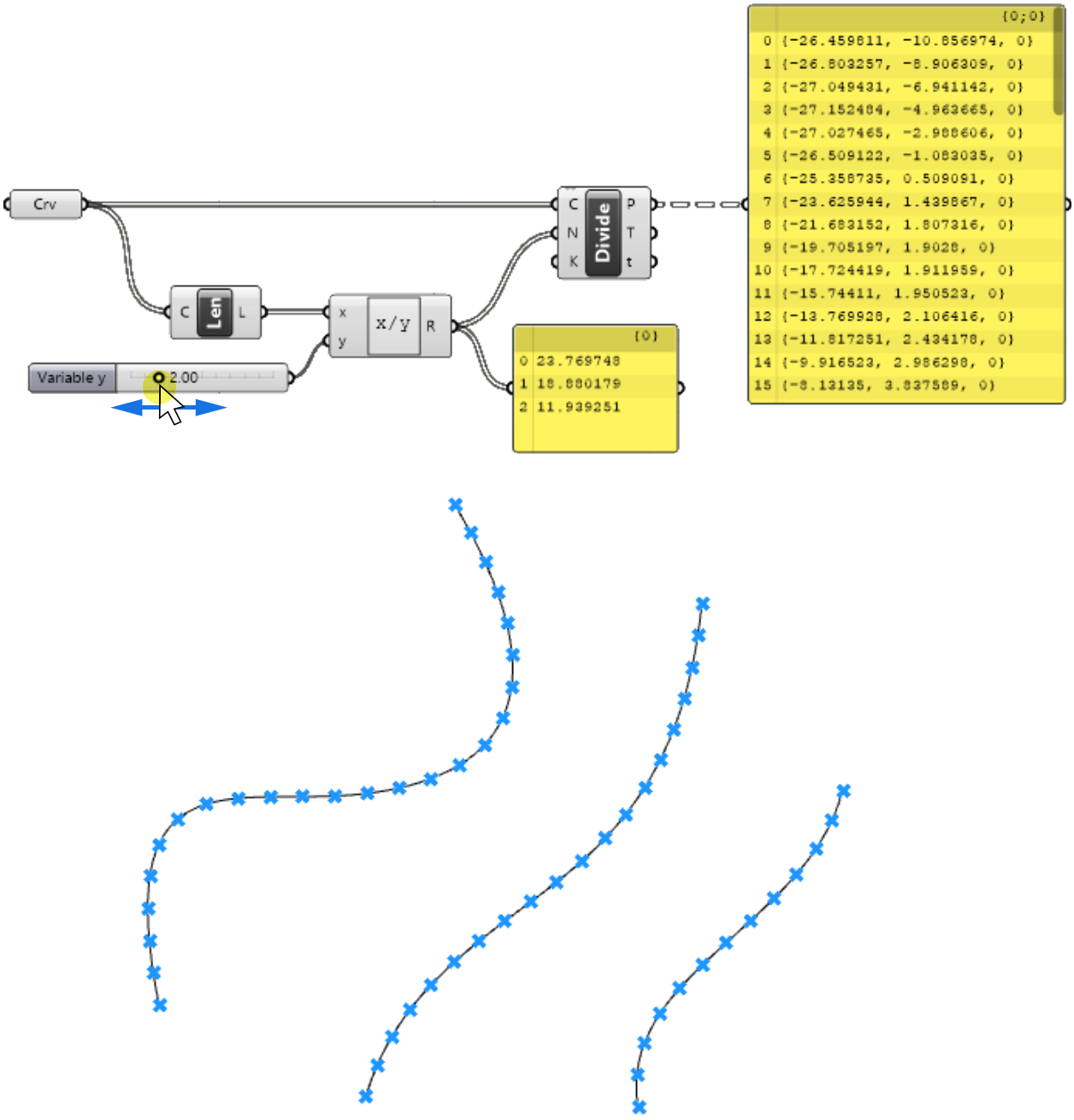
We can see by hovering over the segments (N) input that Grasshopper turned the numbers into whole numbers for us.



Grasshopper will often automatically convert one data type into another one, if possible, to make work easier for us.

In this case a floating point number like 11.884874 will be turned into a whole number (integer) by rounding to the nearest number: 12.

If we now change the value on our slider, we can see that each curve has a different number of points, and that the spacing between the points is similar for all curves. We can also see the update in our Panel component, showing us the different point lists.



That's it! We've created a simple parametric model in Grasshopper!

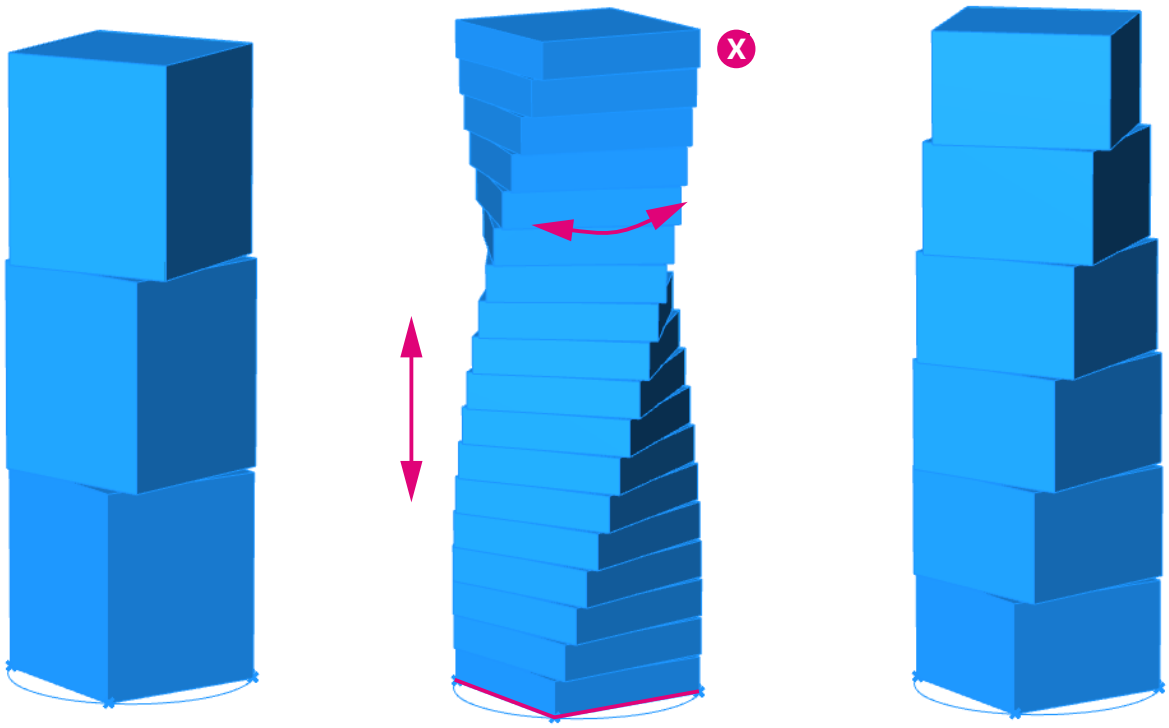
4. Design Exercise: Parametric Tower Design

Now that you've mastered the Grasshopper interface and you know how to create and control design parameters, it's time to put together a full script!

In this chapter, we'll build a complete script step-by-step. We'll explain the logic behind the script, get to know a first set of key components like Move, Extrude, Rotate, and learn about vectors and number series.

To exemplify the core concepts of parametric design, we'll build a simple script that creates a parametric tower made of stacked, rotated segments.

Let's dive in!



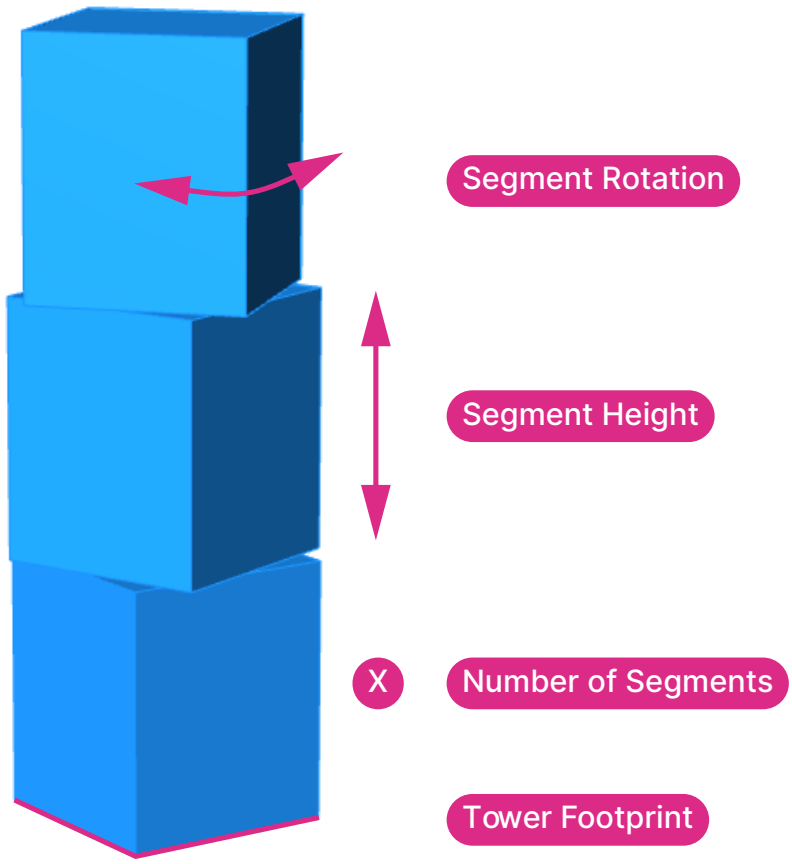
Defining our Project Goal

Every scripts starts with a clear idea of what we want to create.

Our goal for this script is to create a tower made of segments, where each segment is rotated more than the previous, resulting in a “twist”. We want our parametric model to be set up so we can easily adjust the number of segments, their height, the footprint of the tower and the tower segments’ rotation.

Before we start adding components to our script, we need to take a moment to sketch out the design logic. In other words, we need to figure out a series of geometrical operations that will allow us to generate this form in Grasshopper.

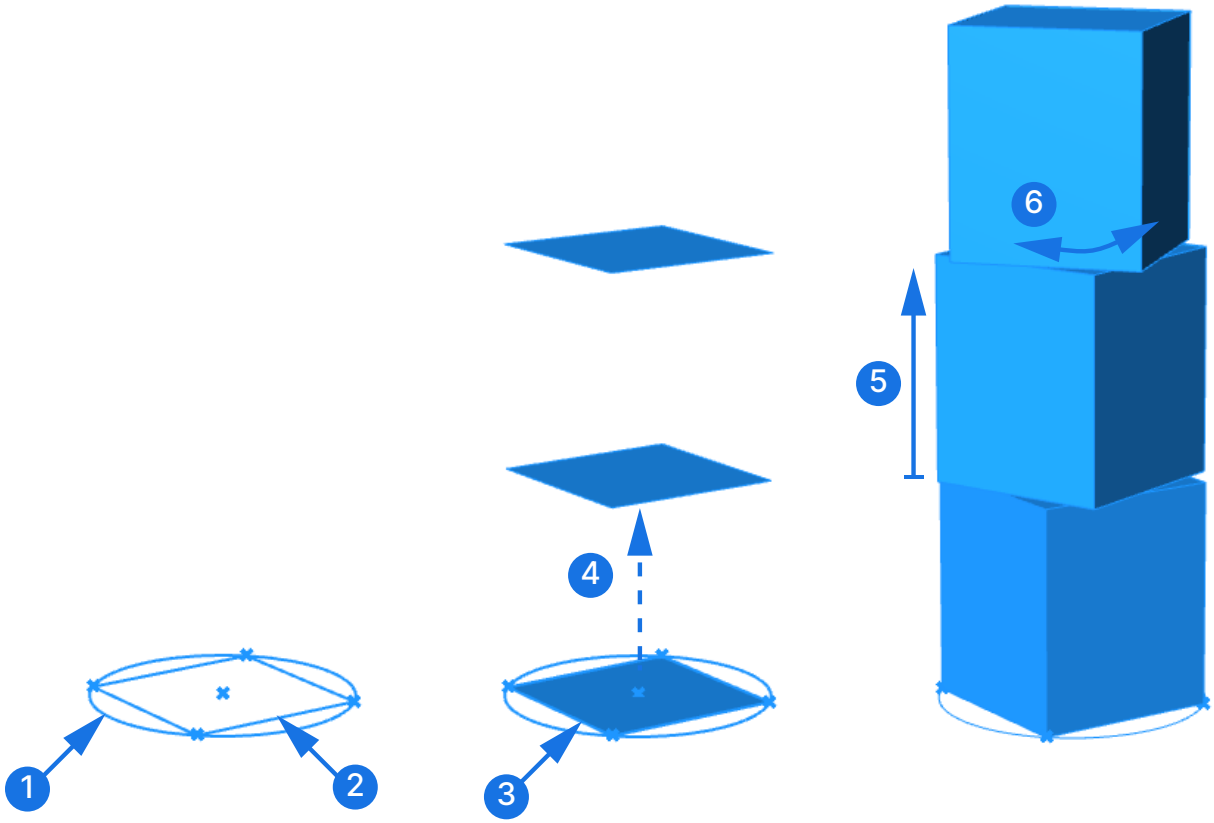
Design Parameters:



Design Logic: The Blueprint for Our Script

There are many different ways to create this form, and the one outlined below is just one of many. Here are the steps we'll take to build our script:

1. We will start by creating a **circle**. We'll control the tower's footprint by adjusting the circle's radius.
2. Next, we'll divide the circle to get four points that we then connect with a NURBS Curve to create a base rectangle.
3. Then we'll generate a surface from the rectangle.
4. Once we have the surface, we'll move it vertically to three different heights.
5. We're going to extrude the surfaces to create volumes.
6. And finally we're going to rotate each of the volumes by a different amount.



Once the script is complete, we can make adjustments at each of these steps to control and fine-tune our design.

Now that we've come up with a logic, let's open Grasshopper and turn this logic into components!

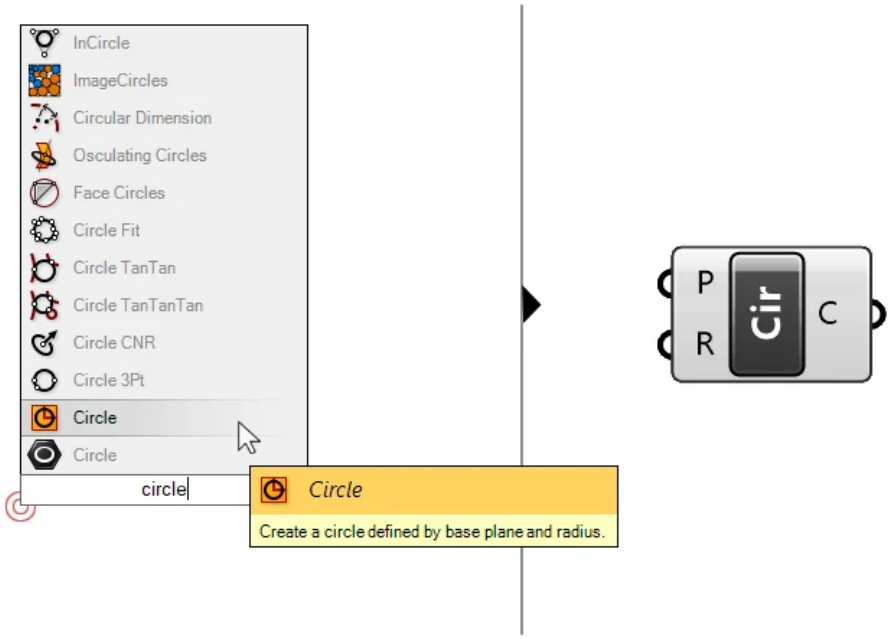
Step-by-Step Tutorial in Grasshopper

Let's start with Step 1: drawing a circle.

Let's double-click onto the canvas to open the component search bar and type "circle" to find a suitable Grasshopper component.

The first result is a data container - not the component we are looking for. We don't want to just reference a circle from Rhino, we want to **create a circle** from scratch to be able to adjust its radius with a slider from within Grasshopper. If we hover over the second result, the tooltip says "Create a circle defined by base plane and radius", that's the one we want. Let's click on it to drop it to the canvas.

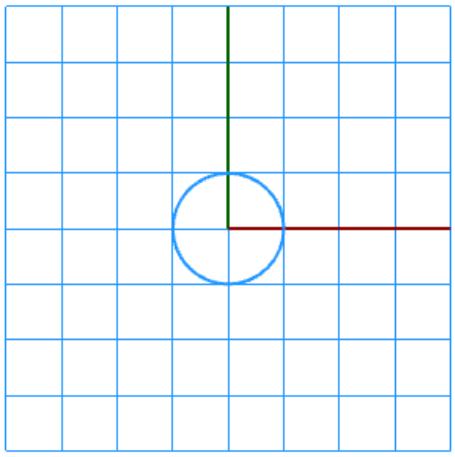
The component has two inputs, the **Baseplane (P)** of the circle and its **Radius (R)**.

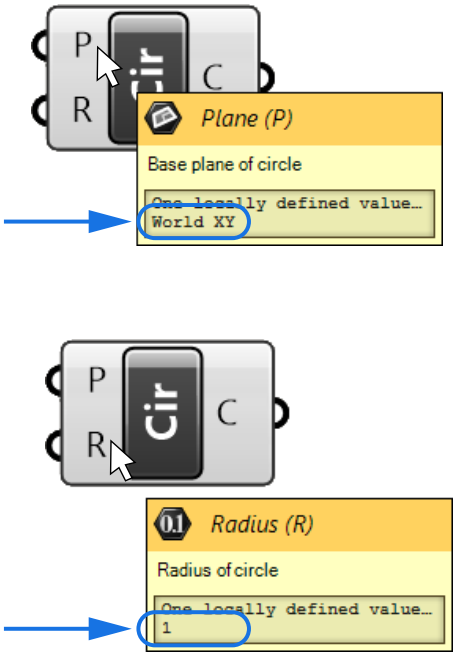


When we hover over the inputs we can see that both already have a **default values:** the Baseplane is set to “World XY”, which is the ground plane in Rhino located at the origin, and the radius has a default value of 1.

Since the component comes with default inputs, we already get a preview in the Rhino viewport, without having specified any of the inputs. We see both the circle as well as the plane, which is shown as a grid.

Reading a Component’s Default Inputs



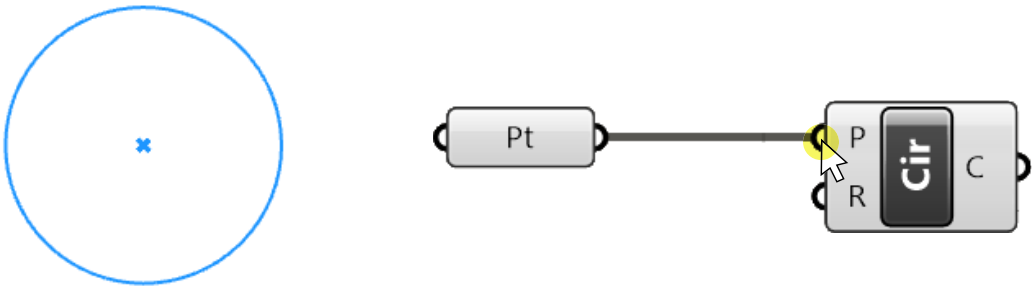


Let’s make sure that we have complete control over the circle by referencing a point from Rhino as the baseplane and adding a Number Slider to control the radius.

Defining the Circle's Baseplane

Let's start by creating a point in Rhino. Next, we add a **Point container** component by typing "Point" in the component search bar and selecting the first result. Then we right-click on it, select "**Set one Point**", and finally select our point in Rhino. Now we can connect the point to the Circle component's base plane input.

As a result, we can control the circle's location by moving the point in Rhino.

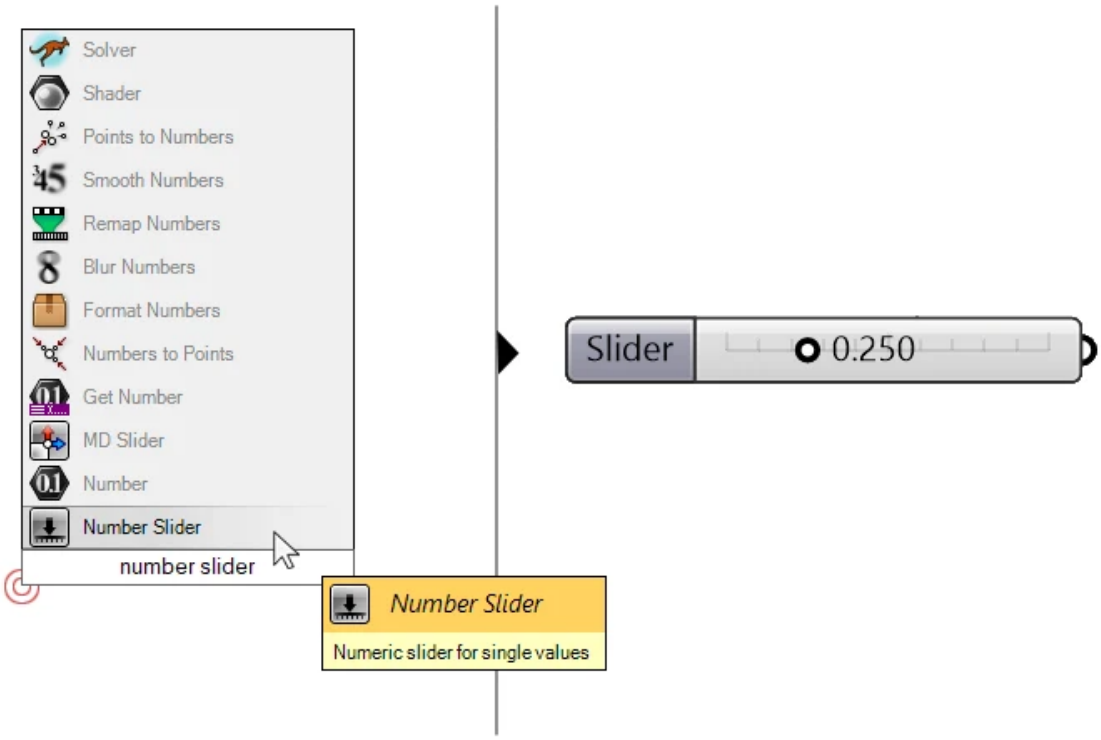


Technically, a Point is not a base plane. A base plane is made of a point location as well as X and Y vectors that define the plane's orientation. But thanks to Grasshopper's automatic data type conversion, if we just provide a point, it will be converted into a plane.

If we don't specify X and Y vectors, Grasshopper will use the default Rhino X and Y directions for the orientation, resulting in a horizontal plane located at the point we referenced.

Adding a Number slider

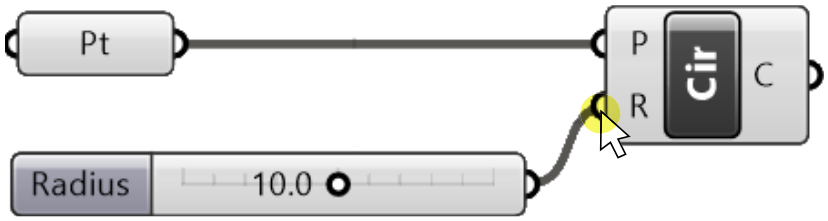
Next, let's plug a **Number Slider** into the Radius (R) input to make it adjustable. Let's type 'number slider' into the component search bar to add the component.

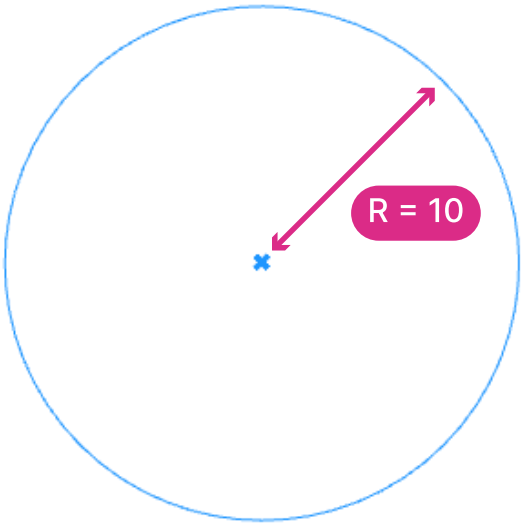


We'll also change the Min and Max values to 1 and 20 respectively, and set the slider accuracy to 1, by double-clicking on the part of the component that says "Slider", and adjusting the values in the window that pops up.

Let’s plug it into the Radius (R) input of the Circle component.

We can slide through the numbers and check in the preview if it works as expected. I will set the value to 10 for now.





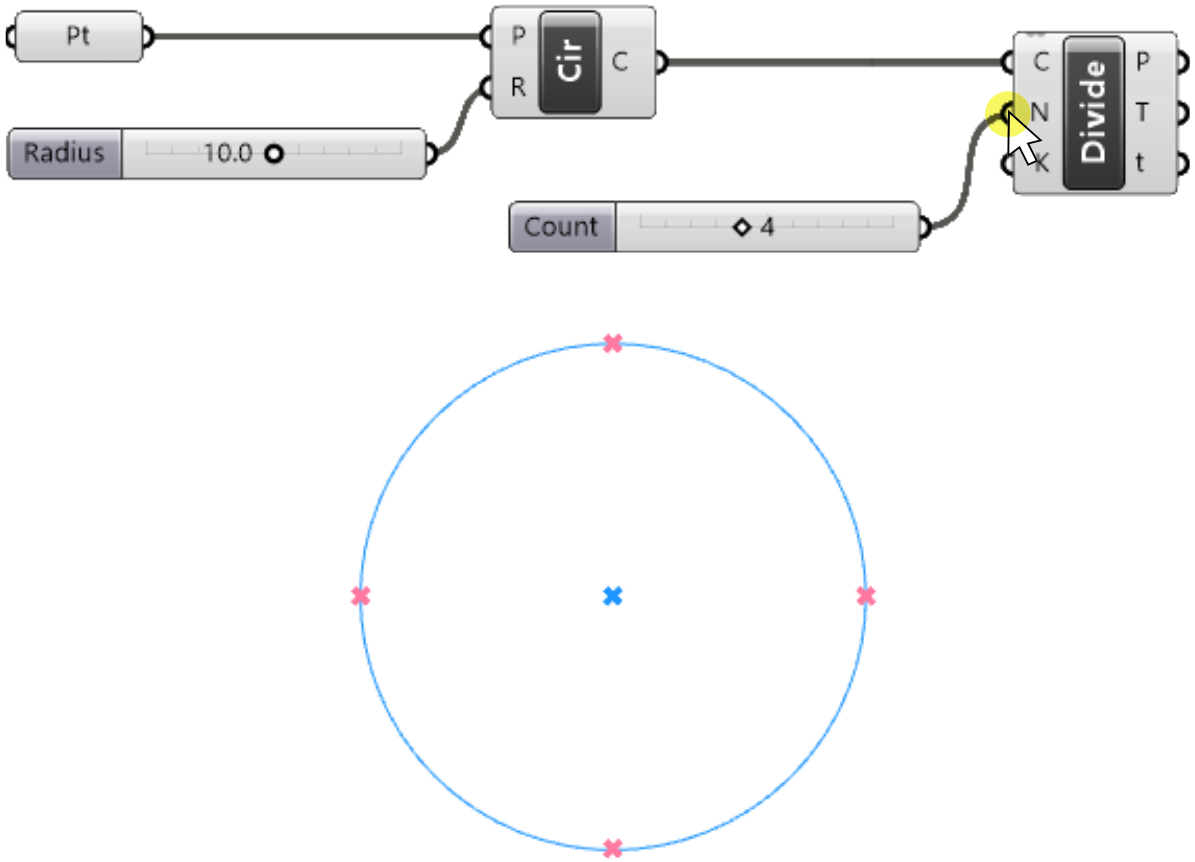
Dividing the Curve

Next, let’s use the **Divide Curve** component to generate four points along the circle. You can find it by typing “divide curve” into the component search bar.

Let’s connect our circle to the first input (C), which is the curve to divide.
As soon as we do that, the default 10 division points will appear on the curve.

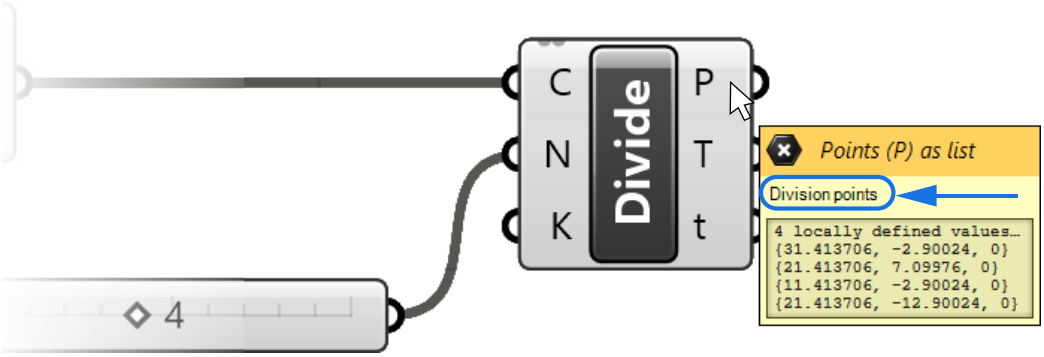
Now let’s add another Number Slider to control the number of segments by typing “4” into the search bar, and hitting Enter. We get a slider with a range of 0 to 10 and the value set to 4.

Next, we connect it to the Number of Segments (N) input of the divide component.



The preview in the viewport looks good: we get four equally spaced points along the curve.

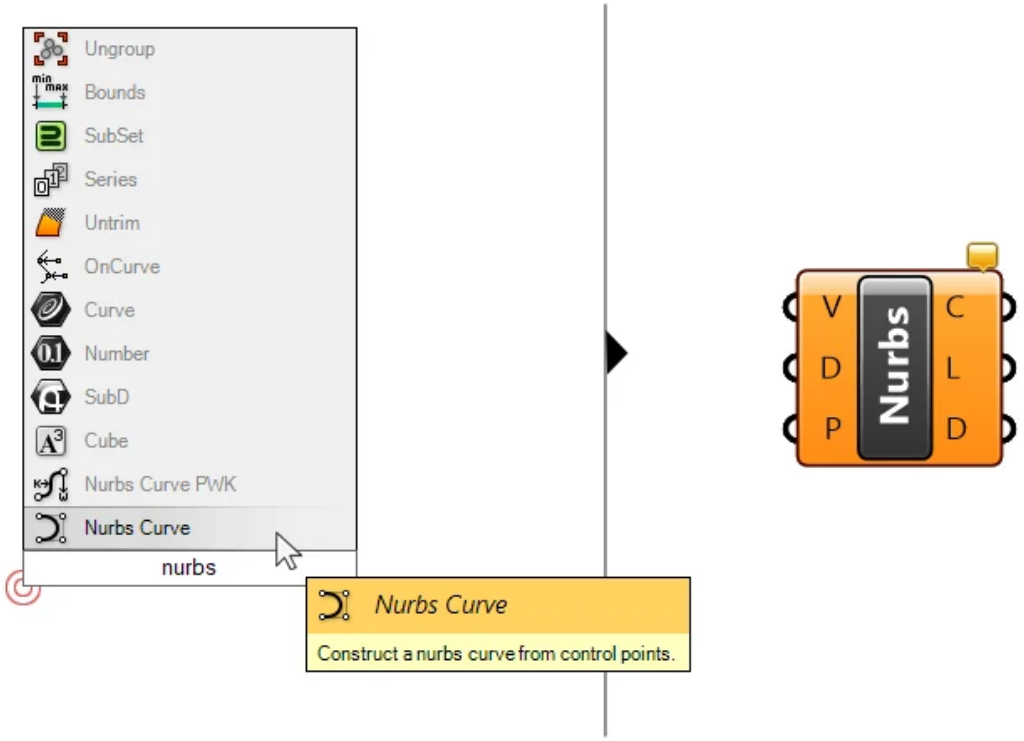
Let's take a look at the outputs of the component: The first output P contains the four **division points**, that's the one we're interested in.



Creating a NURBS Curve From Points

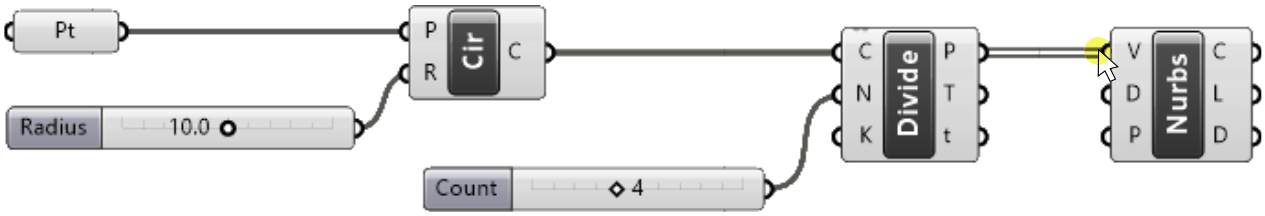
Next, let's connect these four points to create a rectangle. In Rhino we create curves by defining its controlpoints with a series of mouse clicks. In Grasshopper, we can't select points in the viewport, so instead we need to create a **list of points** in the correct order, and plug it into a curve component. All curves in Rhino are build on the same mathematical model: they are **NURBS Curves**. In Grasshopper, we can access the curve command through the "NURBS Curve" component.

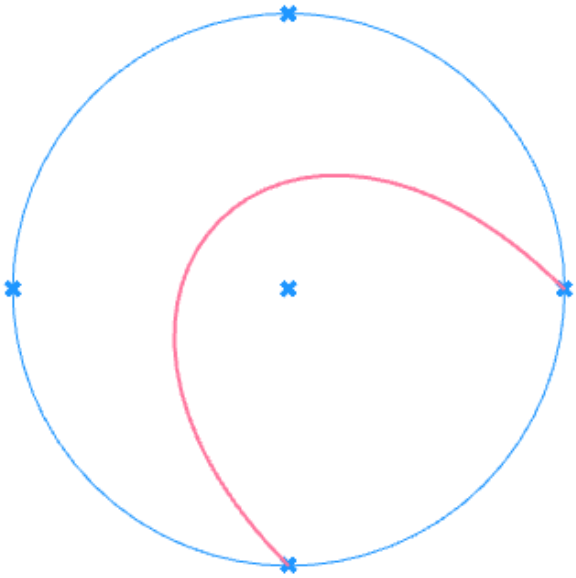
Let's type "NURBS Curve" and drop it on the canvas.



Let's take a look at the inputs it requires. The first input asks for **Vertices (V)**, which are the control points of the curve. Let's connect our division points to this input. They are located in the "P" output of the Divide Curve component.

A look in the viewport shows that it's not quite what we had in mind.





Let's understand what's going on.

First of all, we notice that the curve is smooth instead of simply connecting the points. This is because the default **Curve Degree (D)** in the second input is set to "3".

To understand what that means, we need to take a look at how Rhino constructs NURBS Curves.

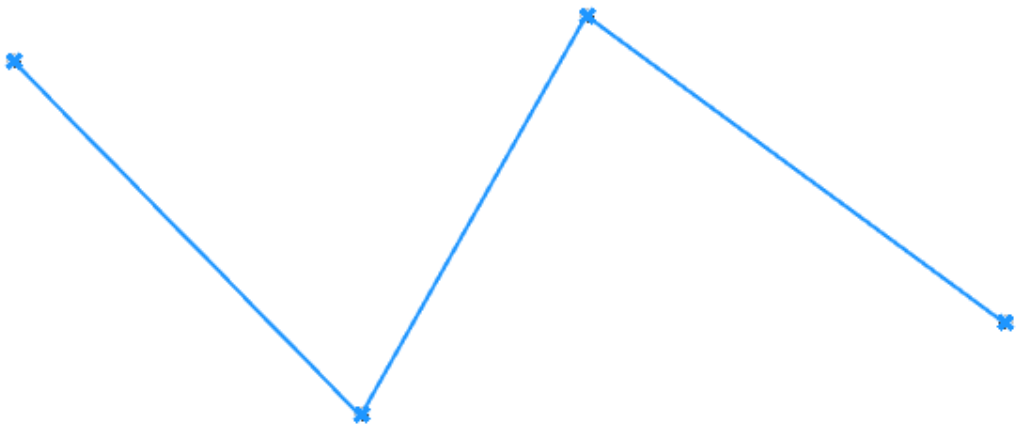
Understanding NURBS Curve Degrees

As we delve deeper into this tutorial, understanding how NURBS curve are constructed in Grasshopper is crucial. The curve degree of NURBS curves defines how **smooth** a curve is. It defines the geometric **continuity** of the curve. You can think of the degree as a way to control the relationship between the controlpoints and the resulting curve.

NURBS Curves come in three (or more) degrees:

Degree 1

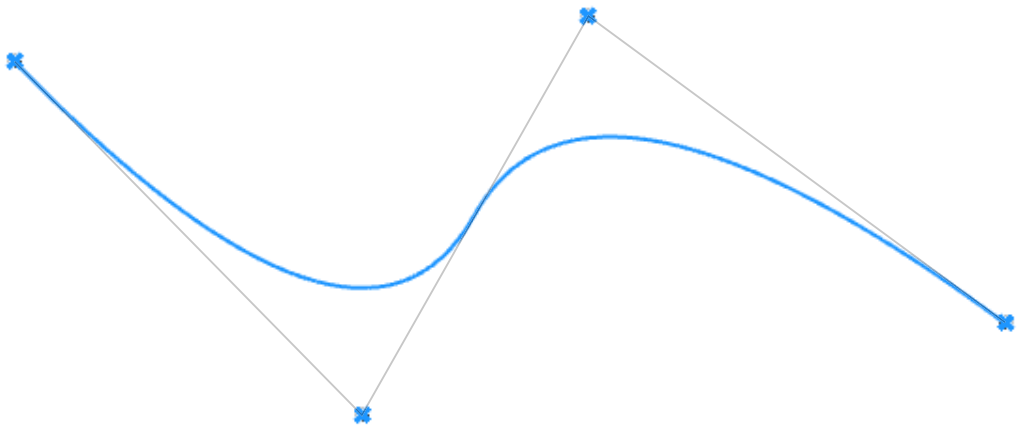
If the curve degree is 1, the controlpoints are connected with **straight lines**, creating a **polygonal** output. All curve segments meet in sharp corners, without any curvature. Two neighboring segments share one controlpoint, but all they share is its **position**. A NURBS curve with a degree of 1 has what's called **positional continuity**.



Curve Degree 1 (positional continuity)

Degree 2

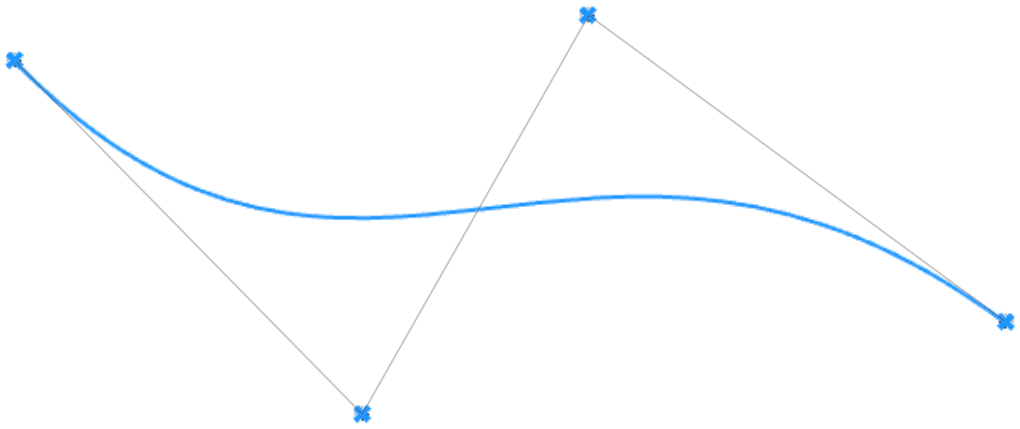
If the degree is 2, it means that the continuity is **tangential**. The resulting NURBS curve now flows in such a way that the curve segments connecting the control-points are tangents to the curve. The curvature now considers 2 controlpoints, one to either side of a vertex, to define the curvature.



Curve Degree 2 (tangential continuity)

Degree 3

A NURBS Curve of degree 3 blends the curvature across three controlpoints at a time and appears the smoothest. In this case we have **curvature continuity**.



Curve Degree 3 (curvature continuity)

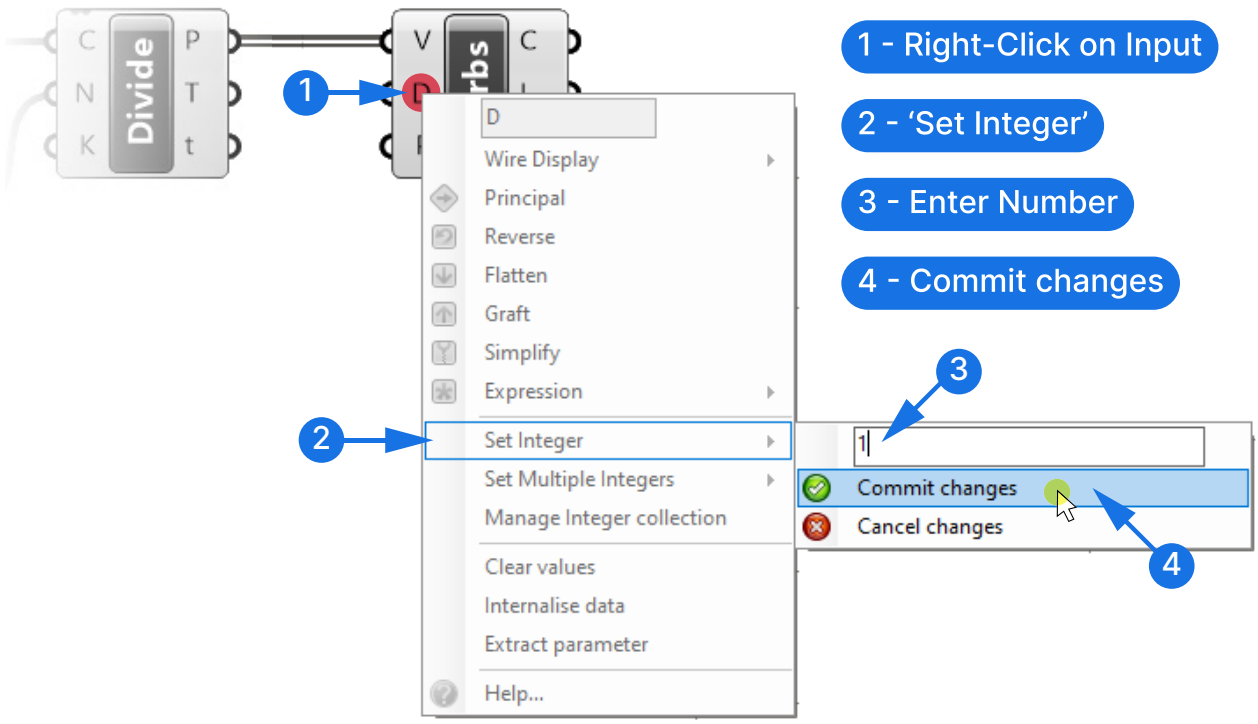
While there are even higher degrees than 3, there is no practical reason to use them. In practice, we'll be using either NURBS curves of degree 1 to create polylines, or NURBS curves of degree 3 to create smooth curves.

So if we want a polyline to create our tower base, we have to change the Curve degree to 1.

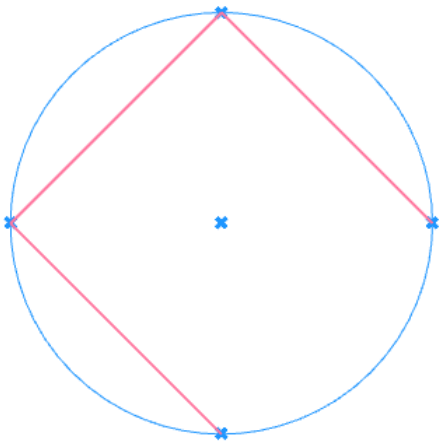
We could add a Number Slider once again, but since we are unlikely to change this value after setting it once, we can set the number inside the component directly.

To do so, right-click on the Curve Degree input, go to Set Integer, and change the number to 1. Make sure to click Commit Changes to apply the change.

Defining Parameters Inside Components

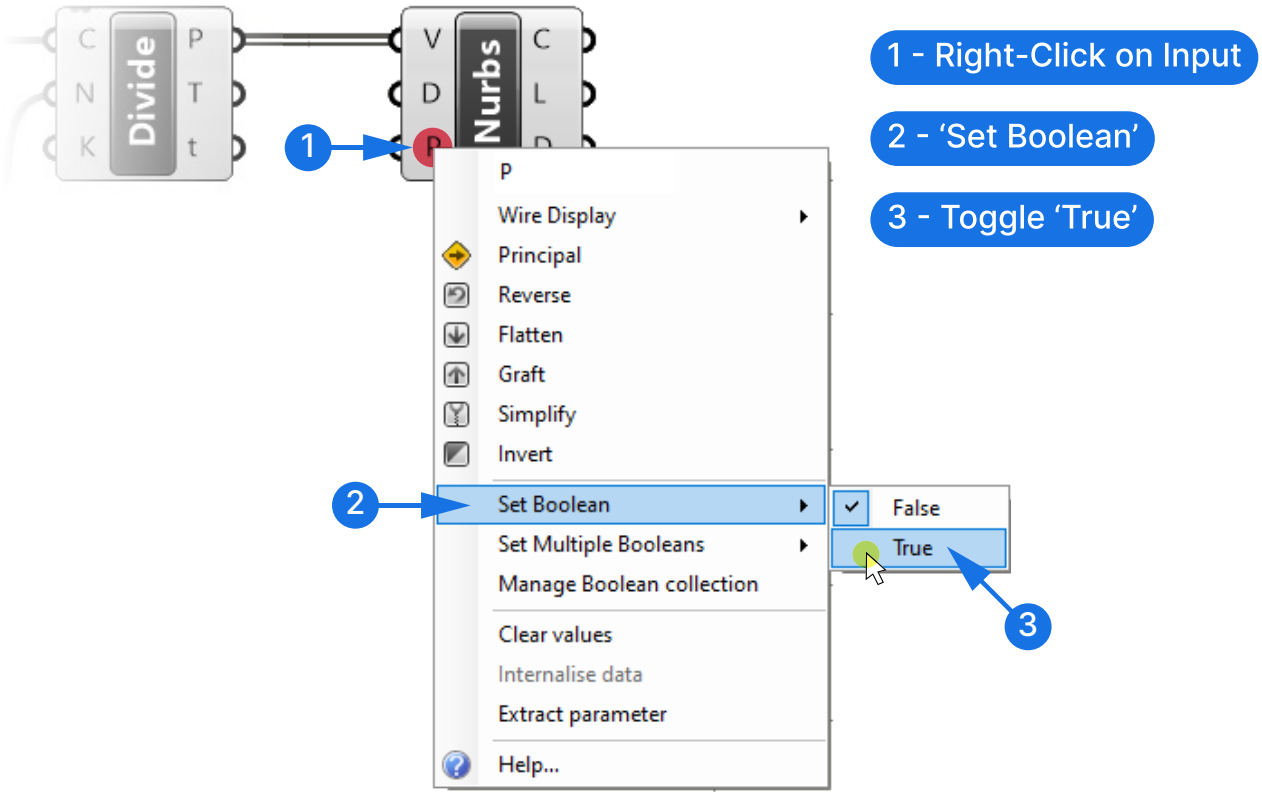


Our preview looks good. The only problem now is that the curve is not **closed**.

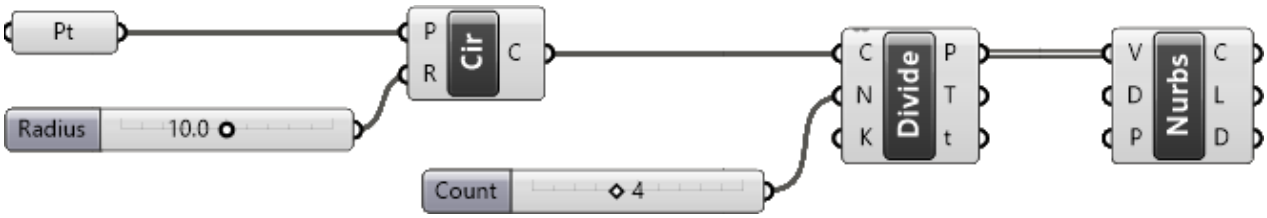


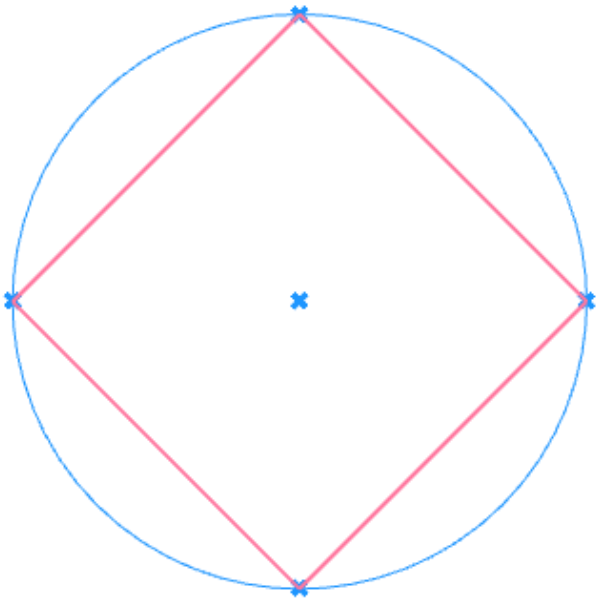
The last input **P** of the NURBS Curve component defines whether the curve is periodic or not, or closed or not. The default is set to **False**, so let's change it to **True**.

Once again let's do this directly in the component, as we won't need to change this later. We right-click on the third input P, go to **'Set Boolean'** and change the value to **True**. "Boolean" is the term used to describe a value that is either True or False, or 1 or 0.



We now have our base rectangle ready!



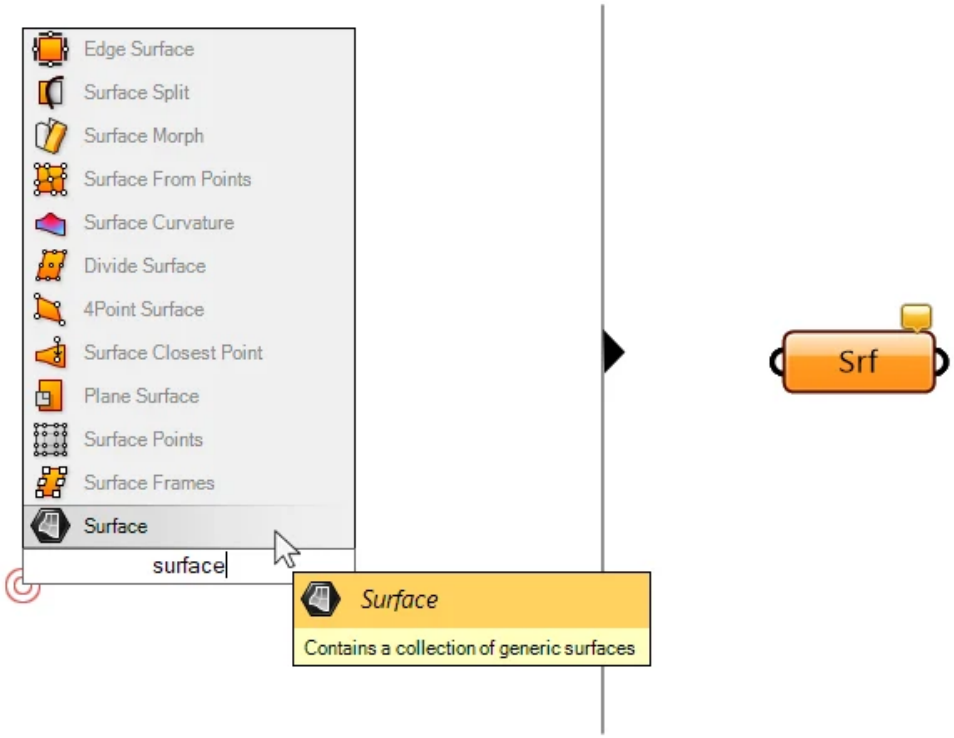


Next, we'll turn it into a **surface**.

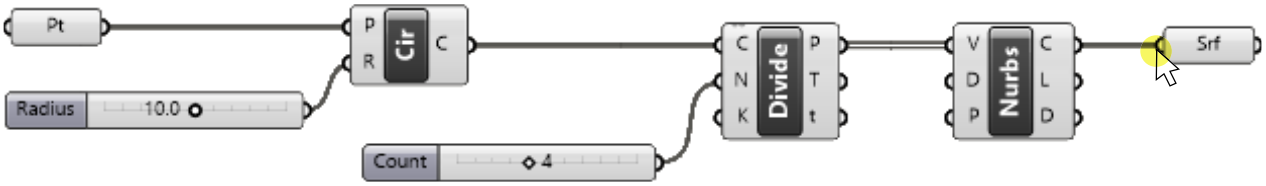
Converting a Curve into a Surface

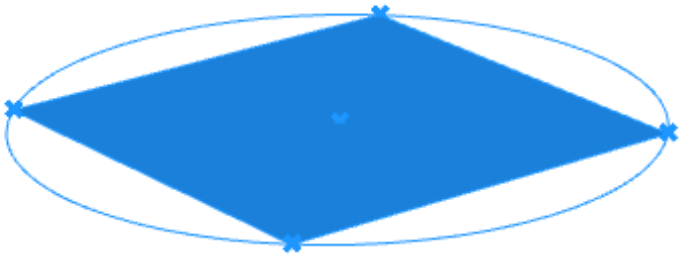
Next, let's turn the closed curve into a Surface, by taking advantage of the automatic conversion functionality of Grasshopper's data containers. Besides storing surfaces or referencing them from Rhino, the Surface component will convert any closed, planar curves we connect to it into surfaces.

Let's type "Surface" into the component search bar and add the **Surface container** component.



We can now connect the NURBS curve we generated. Again, we have several outputs for the NURBS Curve, but the one we need is the first output C, the 'Resulting NURBS Curve'. Let's connect it to the Surface container.





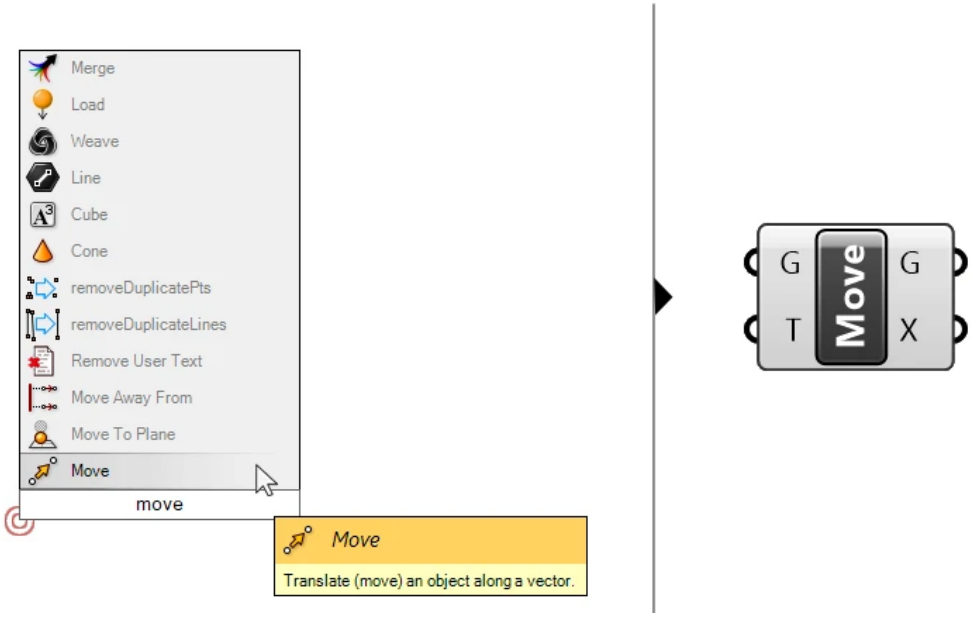
We can now see the shaded preview of the surface in the viewport.

Let’s switch to a perspective view to prepare for our next step: moving the surface to three different heights.

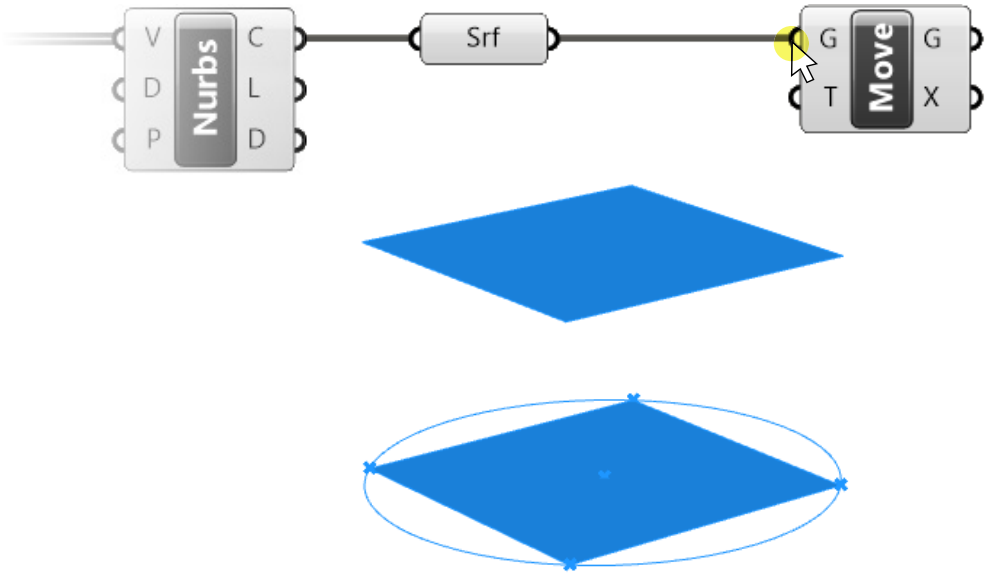
Moving the Surface

Let's start with what we know: we need to move the surface, so let's search for the **Move** component in the search bar and click on it.

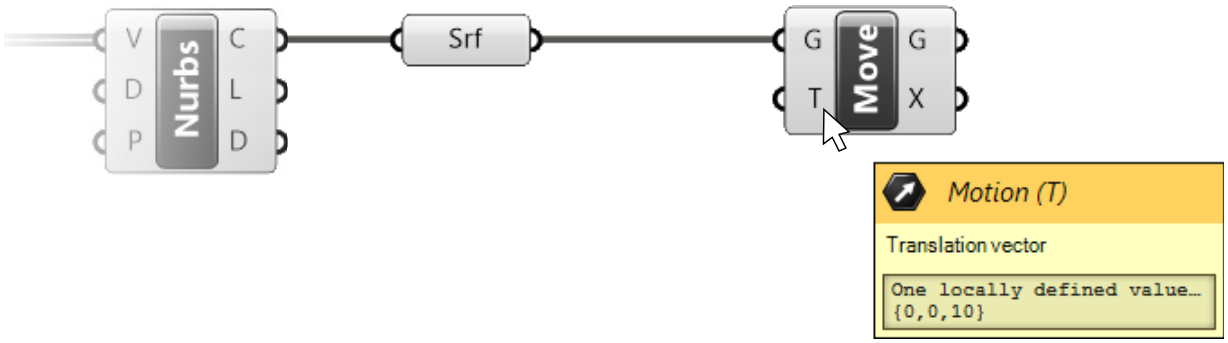
There are two inputs, the geometry (G) to move, and the translation vector (T).



If we plug the Surface container output into the Geometry (G) input of the move component, we see that the surface is already moved up.



That's because, once again, there is a **default vector** inside the translation vector input. We can see the vector by hovering over the translation vector (T) input.



It shows one locally defined value of:

```
{0,0,10}
```

The numbers represent X,Y,Z coordinates. A vector is made of the three X,Y,Z-coordinates, just like a point.

```
{X-coordinate, Y-coordinate, Z-coordinate}
```

The direction and length of the vector is defined as going from the origin {0,0,0} to the X,Y,Z coordinates specified. In Rhino, the Z-axis is the **vertical axis**, as shown in the little widget in the bottom left corner of the Rhino perspective viewport.

The default vector of the Move component therefore describes a **vertical movement** of 10 units.

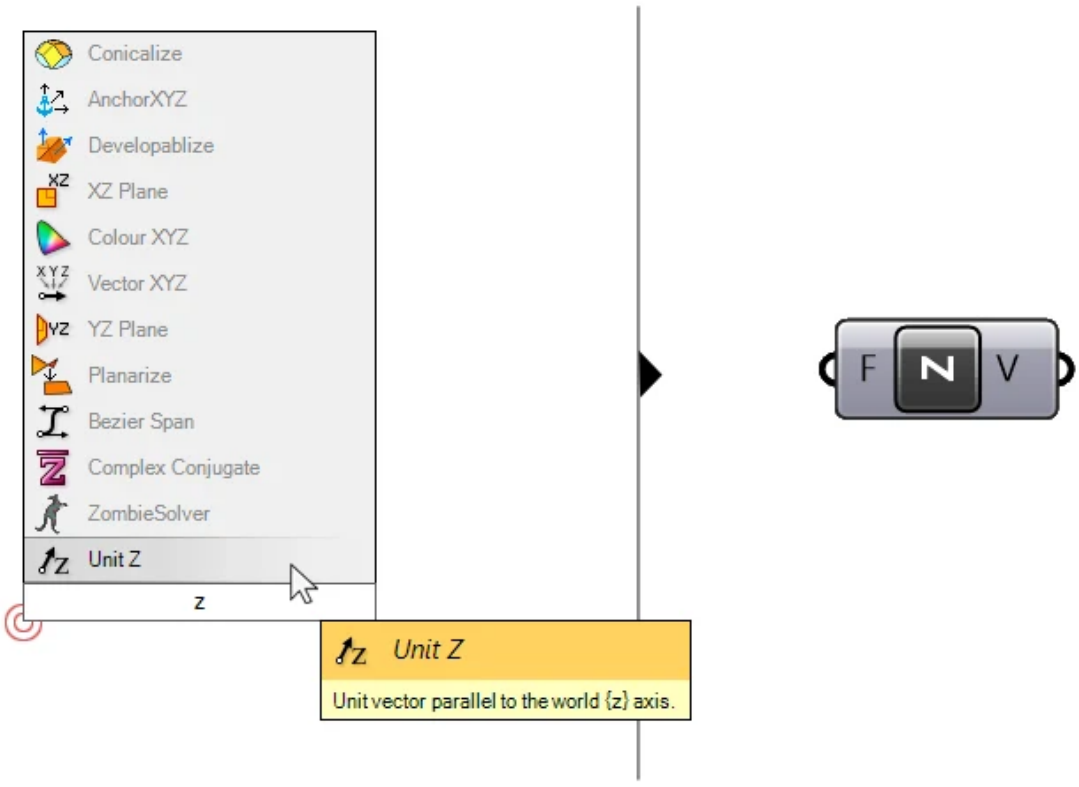
Let's define our own vector!

Constructing a Vector in Grasshopper

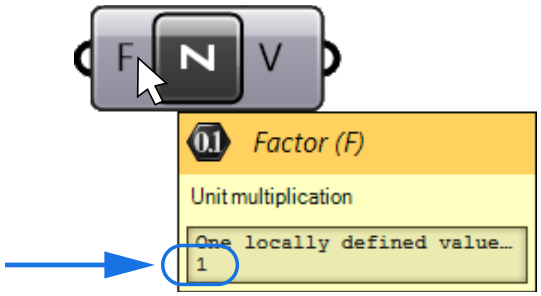
In Grasshopper, there are several ways to create vectors. We can for instance, create a vector from scratch by defining X,Y, and Z values, with the **Vector XYZ** component. But in practice, we'll rarely create vectors manually this way, because defining vectors just with X,Y,Z values is far from intuitive.

Often we simply want to move geometry along one of the principal axis: X, Y or Z. In our case we only want to move the surface vertically, along the Z-axis.

For that, Grasshopper offers three **Unit vector** components, one for each axis. Let's add a **Unit-Z** base vector. Simply type 'z' into the component search bar, and the "Unit Z" component should show up as the first result. Click on it to add it to the script.

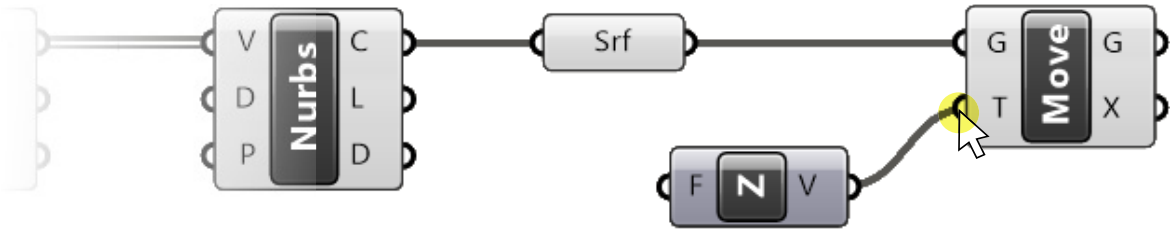


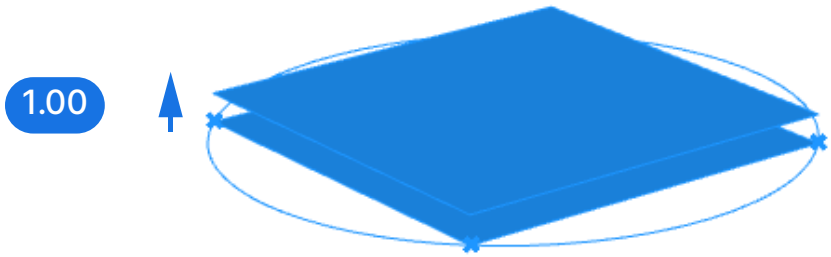
The direction of this vector is locked to the Z direction. The way to control the length of the vector is by specifying it in the Factor (F) input of the component. Any number we connect will define the length of the vertical movement.



By hovering over the input, we see that it comes with a default value of 1. Meaning that this Unit Z vector describes a vertical movement of 1 unit upwards.

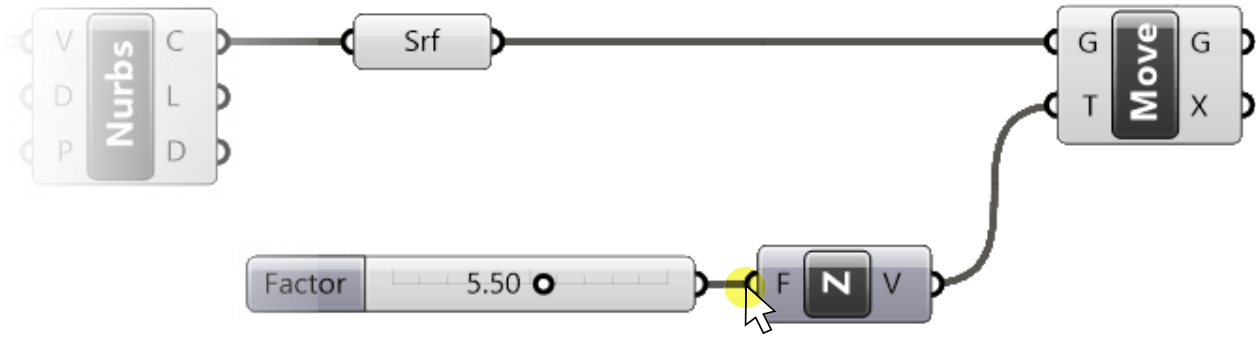
Let's connect the Unit Z component to the translation vector input (T) of the move component. The preview in the viewport shows that the surface is now moved up by 1 unit.

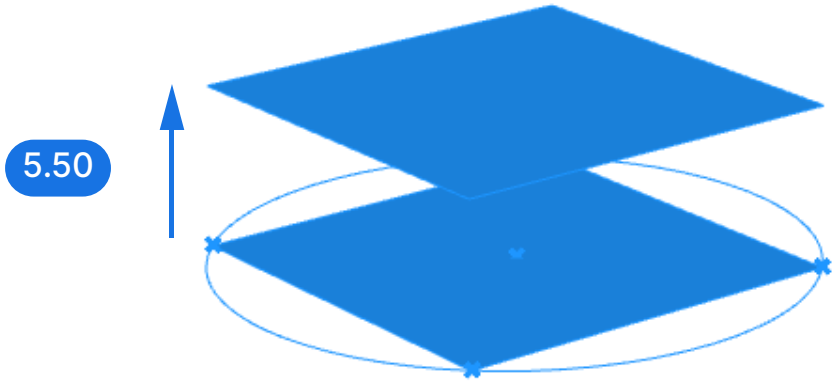




We can define the **length** of our translation vector by adding a **Number Slider** and connecting it to the **Factor (F)** input of the Z-Unit component.

Once we do that, we have defined both the **direction** (along the Z axis) and the **length** of our vector (in the example below, 5.50).

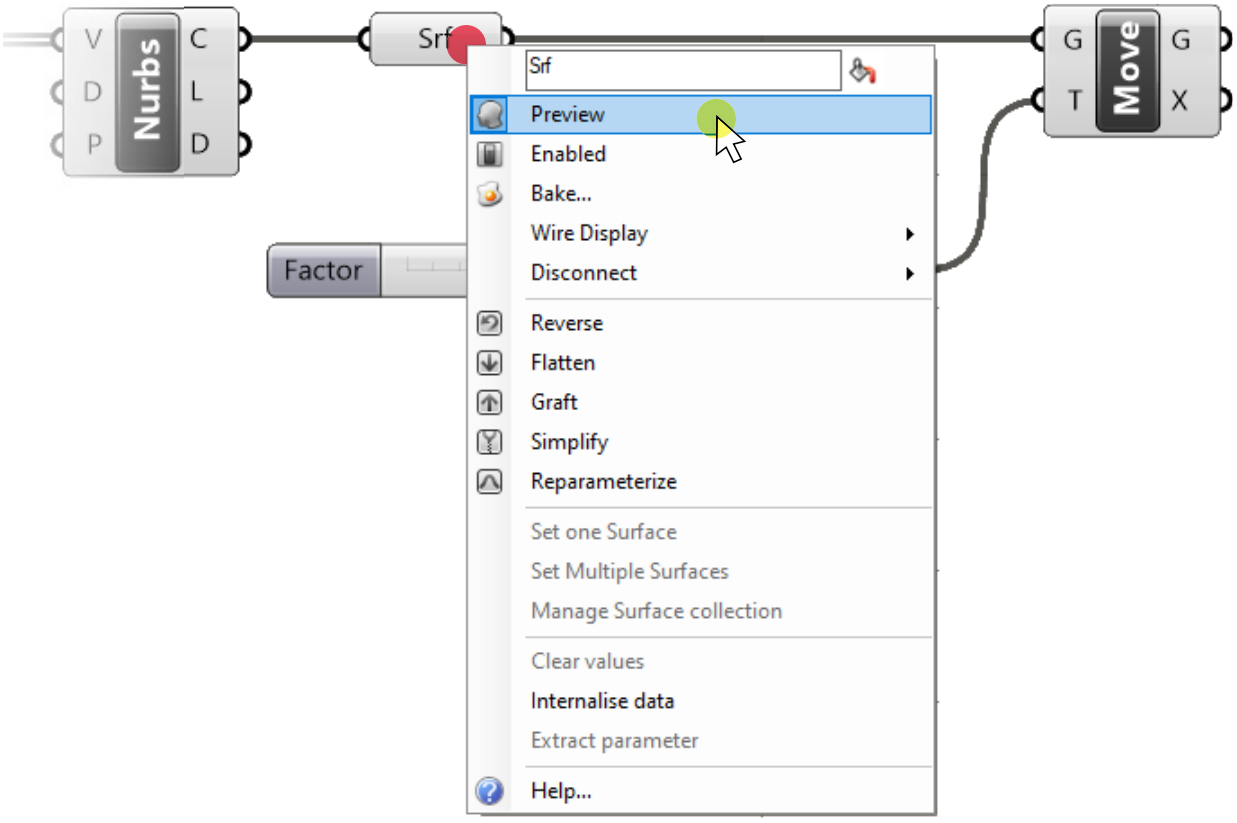


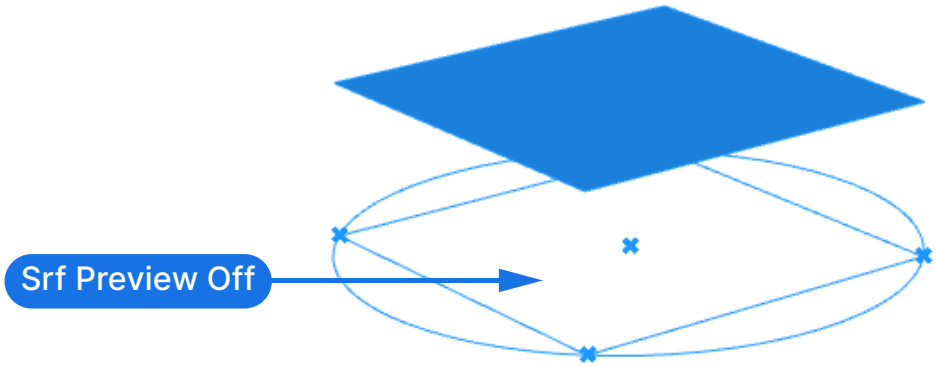


At this point you may be wondering why we still see the base surface on the ground level, even though we moved it vertically.

The reason is simple: In Grasshopper, all component outputs exist at the same time. By using the Move component, we moved the surface vertically, but the original surface still lives, unchanged, in the Surface container! This is also the reason why there is no 'Copy' component in Grasshopper.

To hide the base surface, we can right-click on the Surface container component and toggle the **Preview**. The Surface component will turn dark gray and the surface on the ground plane will disappear.





We successfully moved the surface vertically by an amount we specified, using the **Move** and **Unit Z** components. To move it to three different heights, we need to connect three different vectors to the translation input of our move component.

Let's do that next!

Generating a Series of Numbers

To move the surface to three different heights, we need three Z vectors:

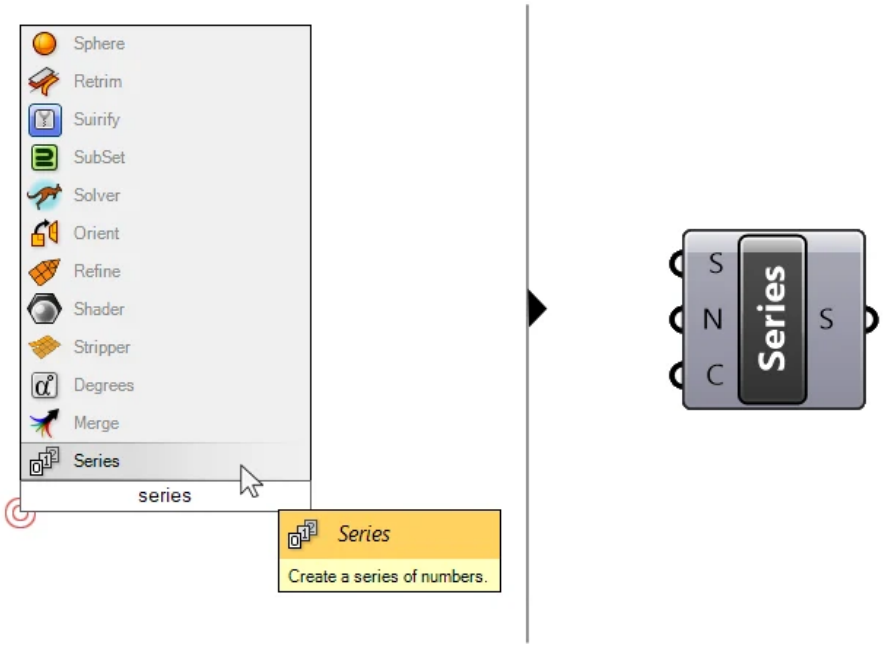
- the base surface can stay where it is, which means we move it by a value of 0.
- The second surface would be moved by a specified amount, let's say, 10,
- and the third one by twice that amount, 20, to ensure that the segments are spaced equally.

This means we'll need three numbers: 0, 10, 20.

We could add three sliders and plug them all into the F input of the Unit Z component to generate the three vectors. But then when we change one, we'd have to manually adjust the others to keep the equal spacing. Instead, we can describe the relationship between these numbers.

Mathematically speaking, we are creating a **Series of numbers**. We can use Grasshopper's **Series** component to generate the numbers for us.

Let's add it to the canvas by double-clicking onto the canvas and typing "series", and adding the component by clicking on it.



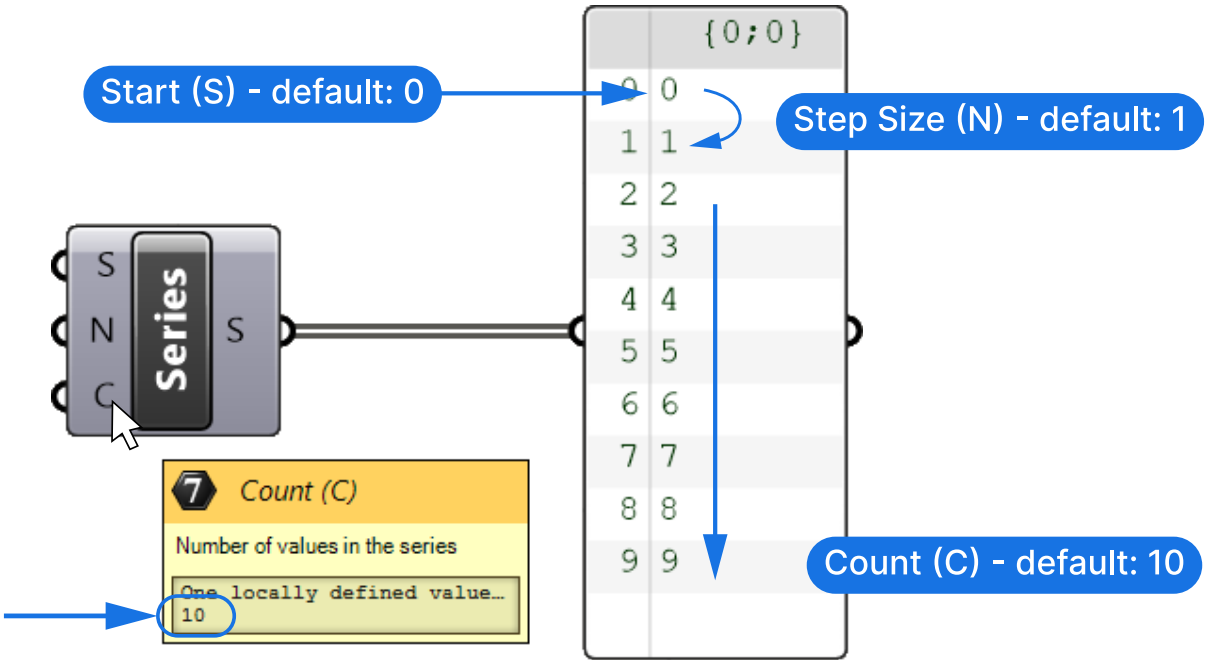
As the name suggests, the Series component creates a series of numbers. By checking the inputs we can see that it allows us to define

- the **Start (S)** or first number of the series,
- the **Step size (N)** for each successive number,
- and the **Count (C)** of the numbers to generate.

The default inputs are a starting point of 0, a step size of 1 and a count of 10, resulting in the following output:

```
0, 1, 2, 3, 4, 5, 6, 7, 8, 9
```

We can visualize the output by connecting it to a Panel component.



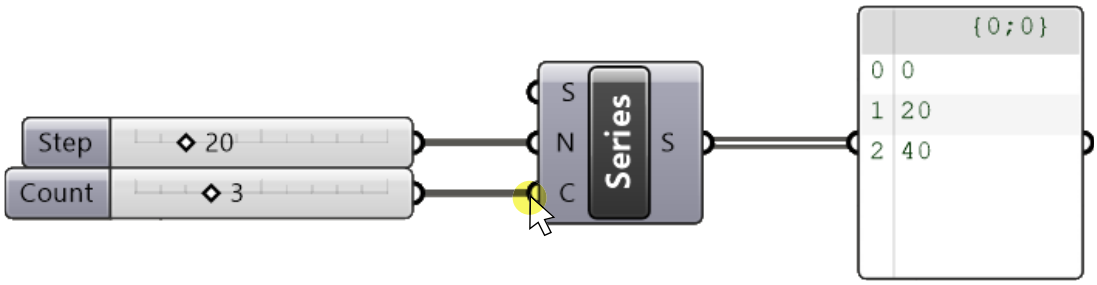
Tip: To change the default color of the Panel component, right-click on the Panel, go to **Colour** and change the Color there. To make every future Panel have the same color, right-click on the Panel again, go to **Adjust Defaults**, and select "**Make Colour Default**".

Let's add two **Number Sliders**, one to define the **Step size (N)** and one to define the **Count (C)**. We can leave the Start of the series at the default value of 0.

Let's type 20 into the search bar and hit enter to add a Number Slider with a range from 0 to 100 and the value 20. This number will control the height of the tower segments.

And let's do the same again with a value of 3. This number will control the number of tower segments to create.

Let's connect the Number Sliders to the Series component.

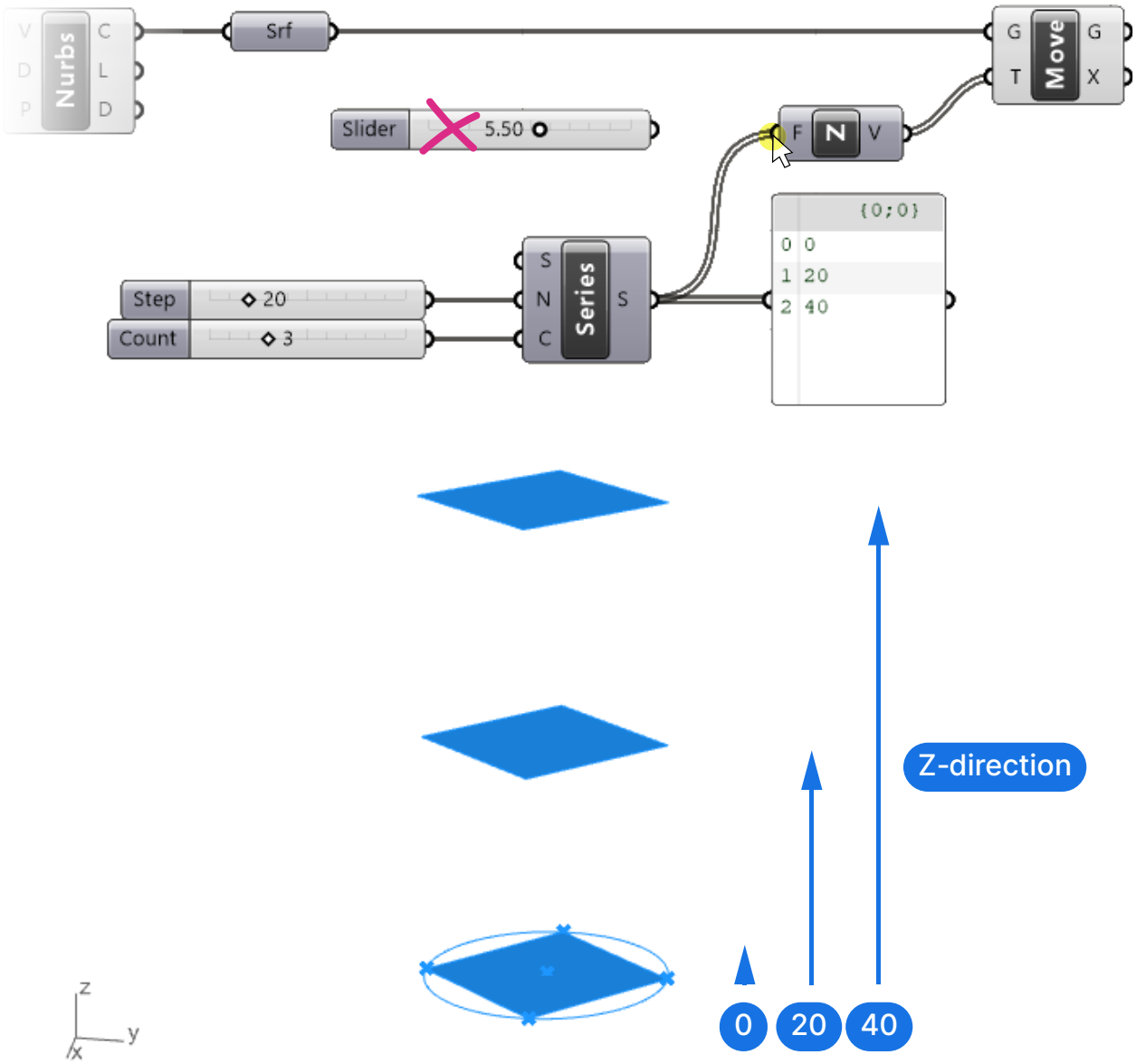


We can see in the Panel that the numbers are what we expect them to be: the series starts at 0, has a step size of 20 and contains three numbers total:

```
0, 20, 40
```

Nothing has changed yet in our preview, because we still need to turn these numbers into vectors along the Z-axis.

To do so, we connect the resulting series output (S) to the Factor (F) input of our Unit Z component. In doing so we'll replace the existing connection to the slider, which we can now delete.

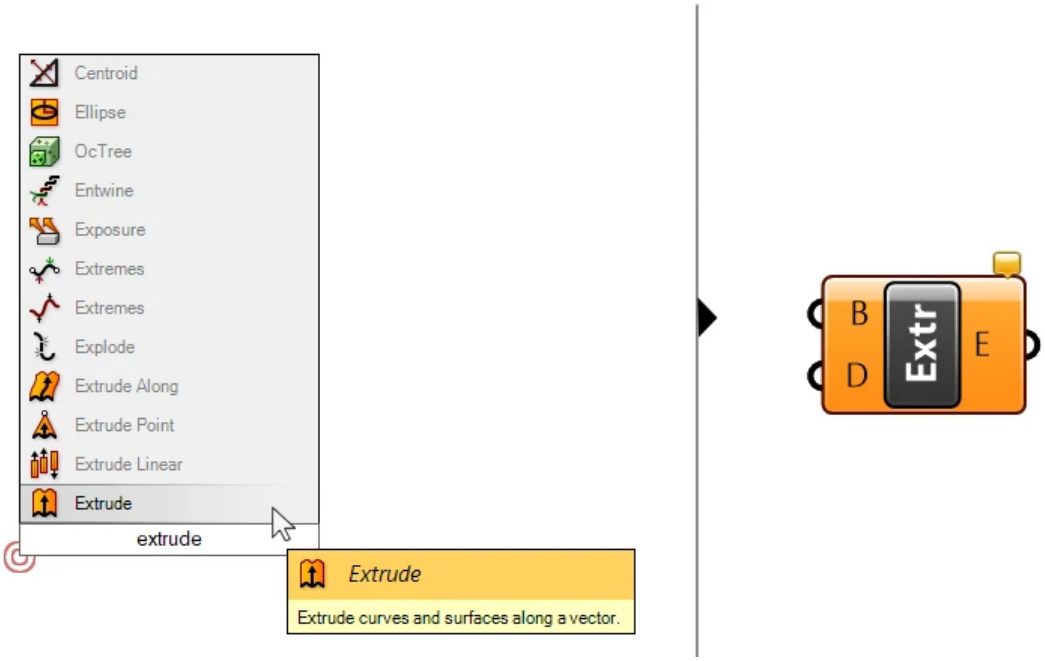


As soon as we do that, we can see our three resulting surfaces in the viewport. If we select the Move component, the surfaces are highlighted in the viewport. The first surface has been moved by a factor of 0 along the Z axis, so it's still on the ground plane.

Next step: extruding the surfaces!

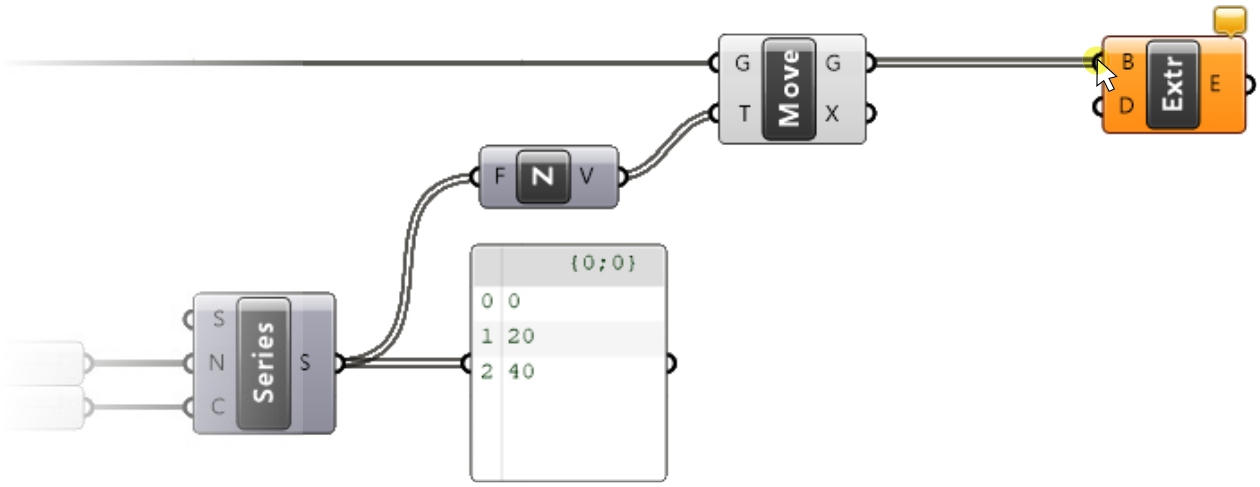
Extruding the Surfaces

Let's drop an **Extrude** component onto the canvas and check its inputs.

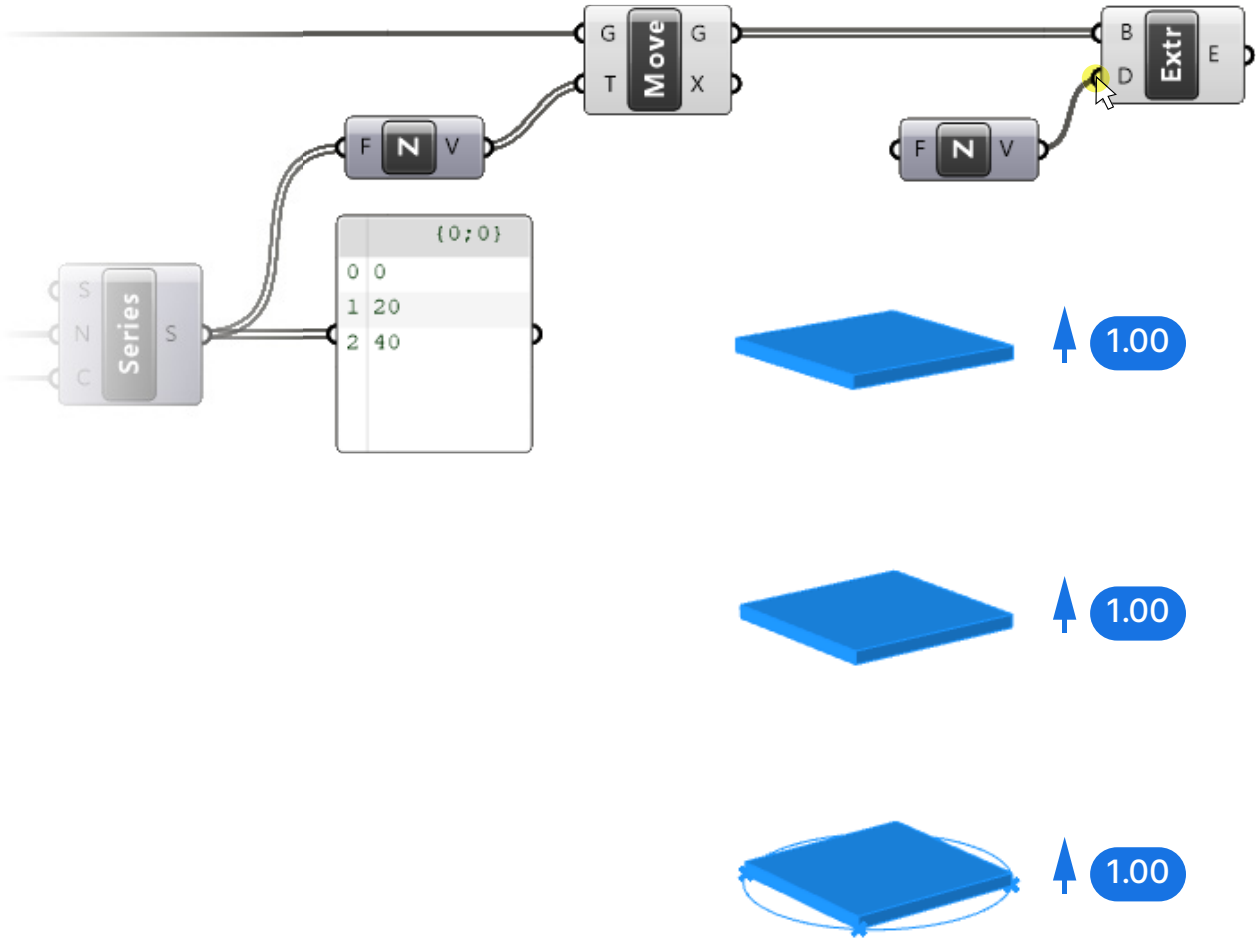


The Extrude component can extrude both **curves and surfaces (B)**, and needs an **extrusion direction (D)** in form of a vector.

Let's go ahead and connect our surfaces.



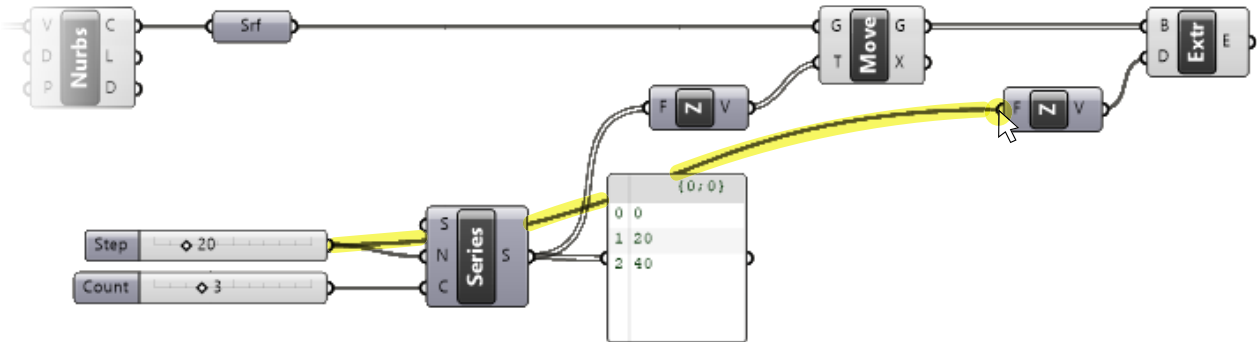
Nothing happened in the viewport, because the extrusion direction input doesn't come with a default value. Let's set up the extrusion direction, which is once more vertical, or along the Z-axis. So let's add another **Unit Z** component and connect it.

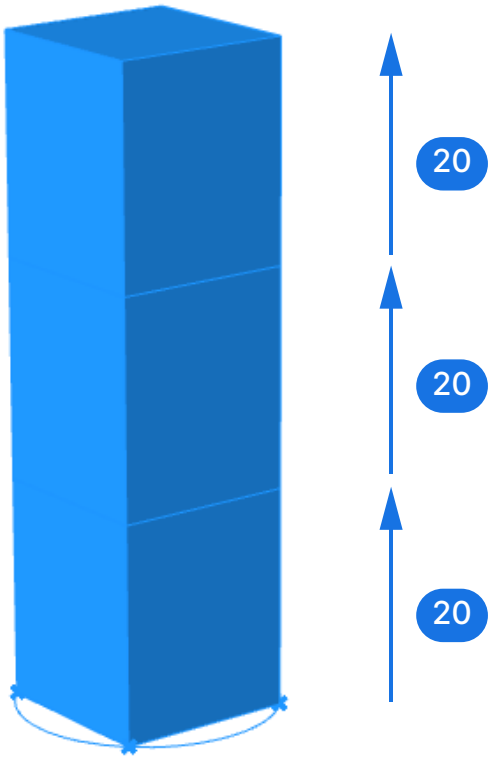


Our surfaces are now extruded by one unit, because, as we've seen before, the Unit Z Factor (F) input has a default value of **1**.

To stack the tower segments, we need to extrude the surfaces vertically by the **same amount as the distance between the moved surfaces**, which is 20.
We could add another slider and give it a value of 20, but if we change the step value of our series later, we would need to also manually change this extrusion height.

Instead, we connect the **same** Number Slider we used to define the step size to the Factor (F) input of the Unit Z value for the extrusion. This way, these two values will always be linked. One Number Slider now controls two different parameters.



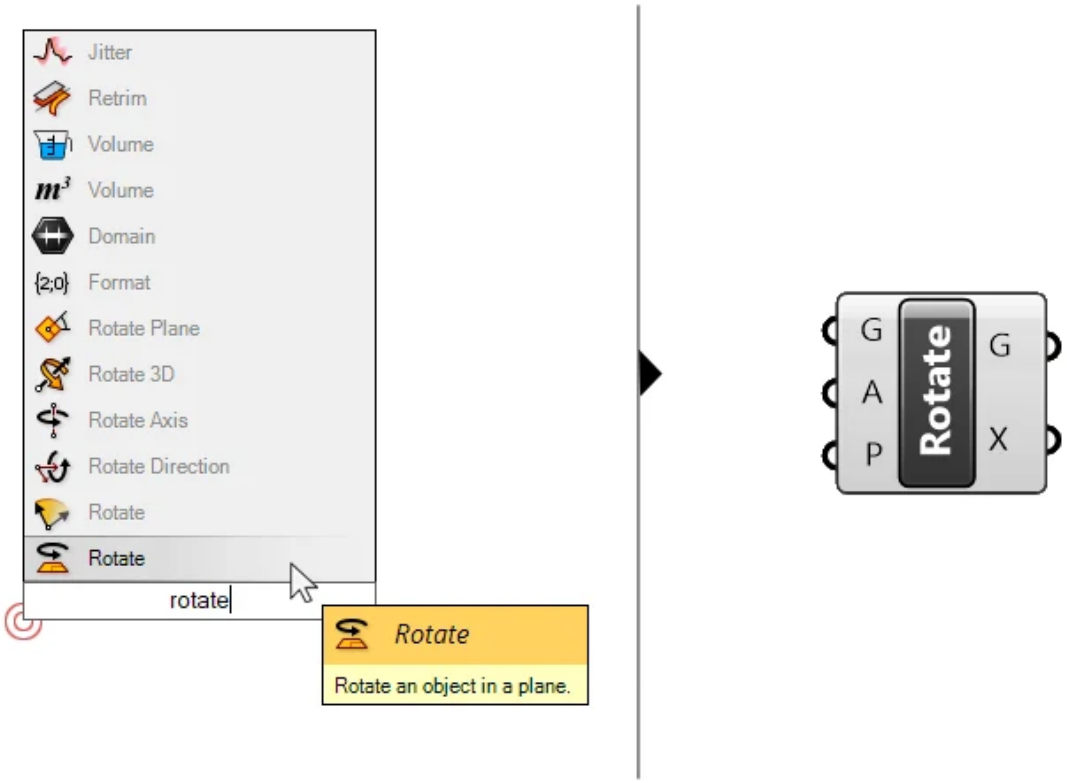


Perfect, we have our three extruded segments! The last step is to **rotate** the extrusions.

Rotating the Tower Segments

Let's find the right component to rotate the tower segments. If we type "rotate" into the search bar, we get many results. Grasshopper has several components that can help us to rotate objects, they are all slightly different. Carefully read the popup descriptions to pick the right one.

We want the one that says "**Rotate an object in a plane**" - it's the first result. Let's select it.
You know the drill! Let's check the inputs for the rotate component:



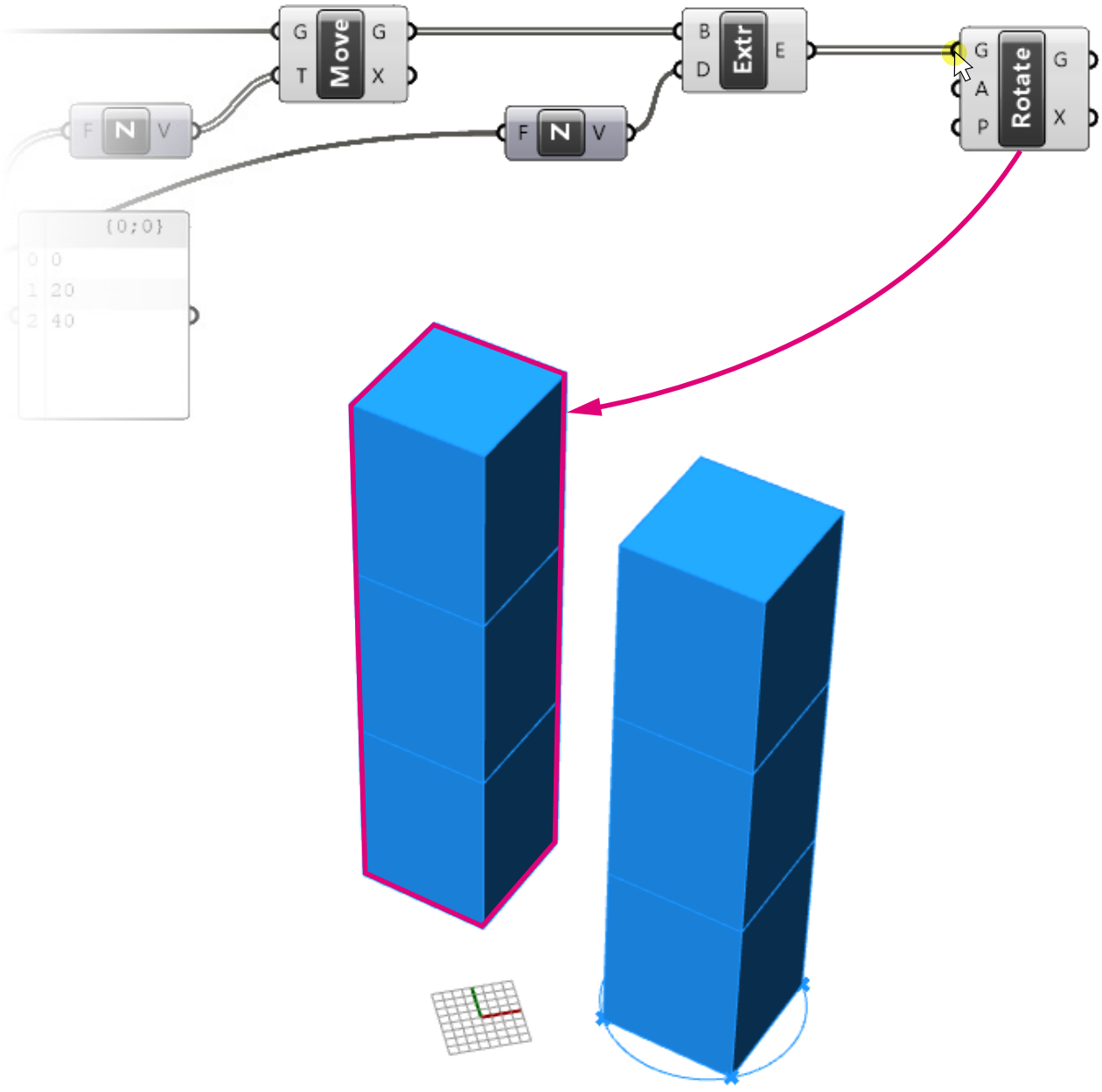
The first input **(G)** is the **geometry to be rotated**.

The second one **(A) is the rotation angle.** It's important to note that the angle input must be in radians, not degrees. Luckily we can easily convert degrees to radians, I will show you how in a second. There is already a default value in here, 0.50 * Pi, which is the radians equivalent of 90 degrees.

The last input **(P) is the rotation plane**. When we rotate an object in a plane, the rotation plane defines both the location and rotation axis of the rotation.

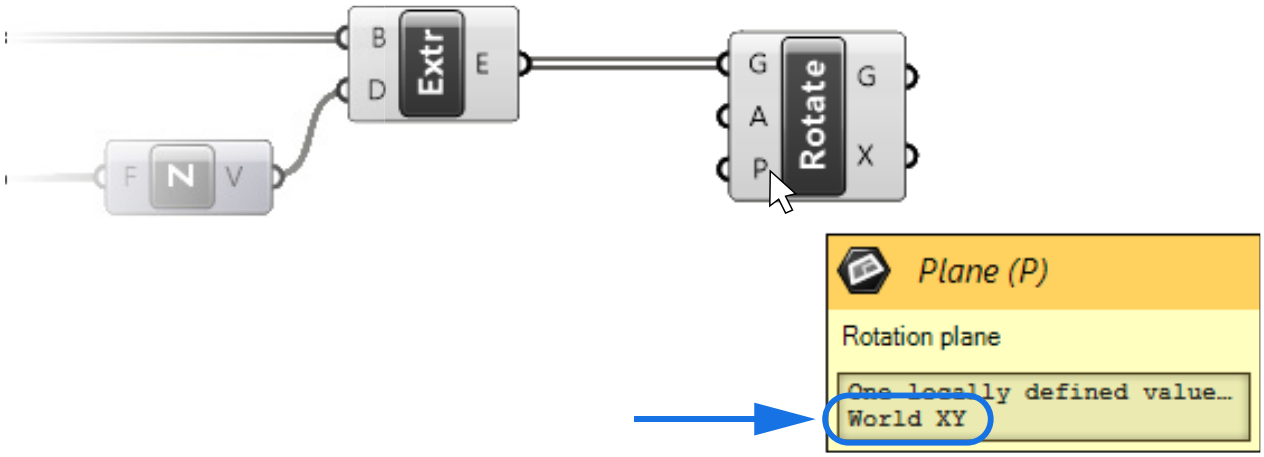
Let's begin by connecting our extruded volumes output (E) to the Geometry (G) input of the Rotate component.

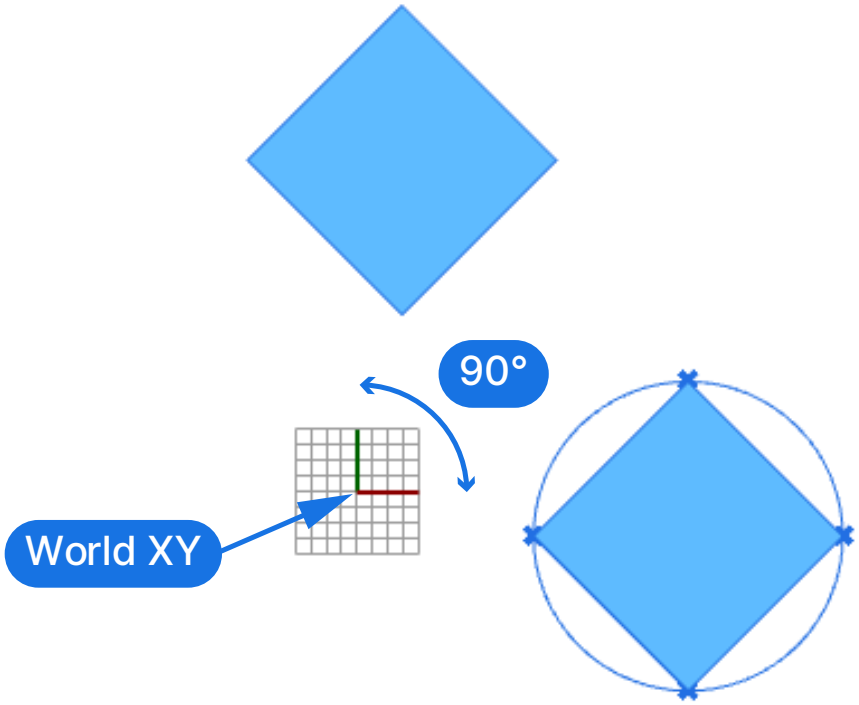
If we check in our viewport, we now have what looks like a copy of our tower in what seems a random place!



The reason is simple: let's go to the top view to understand what happened.

Since we haven't specified a rotation plane yet, the **default rotation plane** is the **World XY** Plane, which is the Rhino origin. That, combined with the **default rotation** input of **90 degrees**, means that the component took our tower segments and simply rotated them 90 degrees around the origin. The exact location of the rotated segments depends on the location of the point in Rhino that we referenced the circle to.



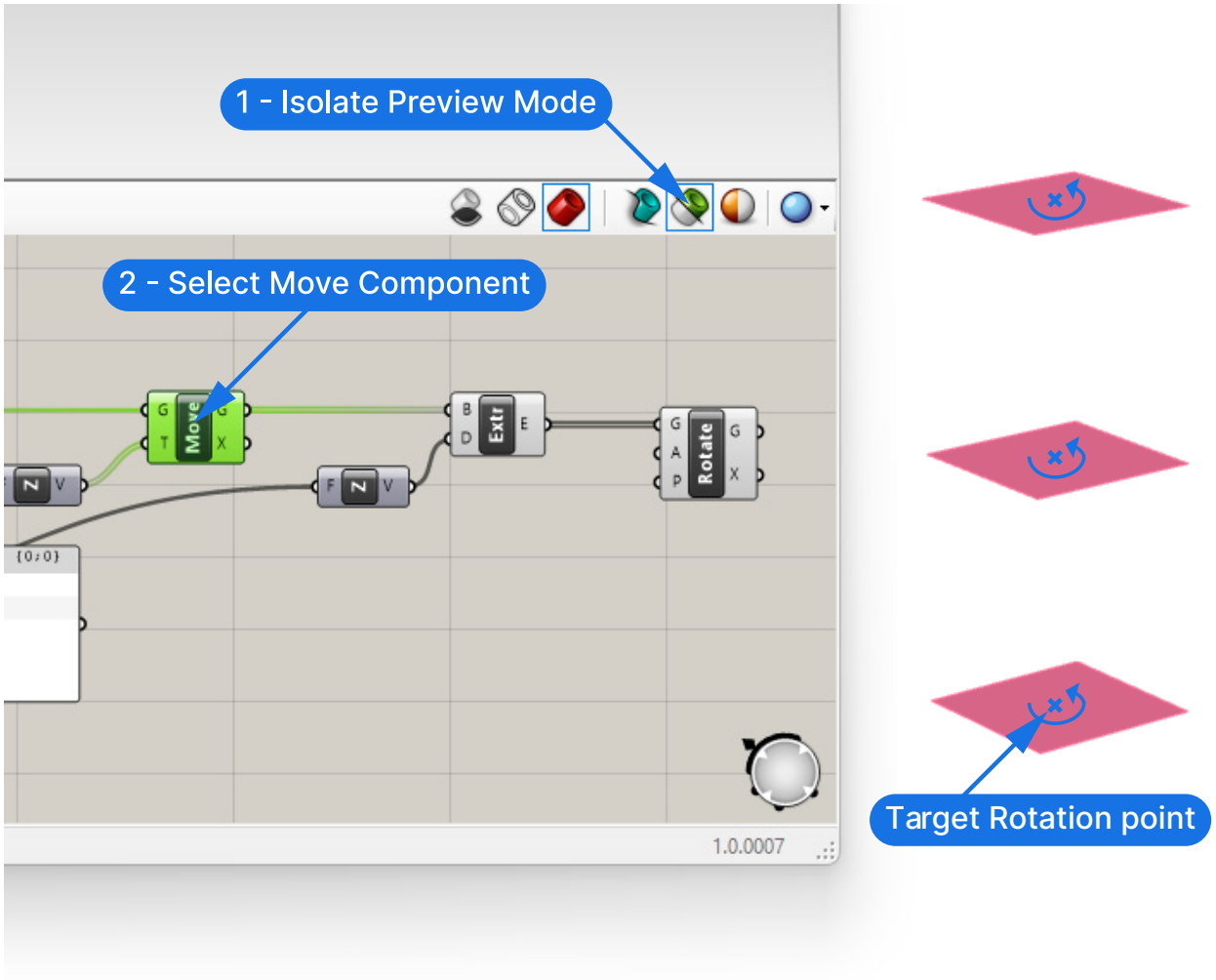


Let’s specify the rotation plane to fix it.

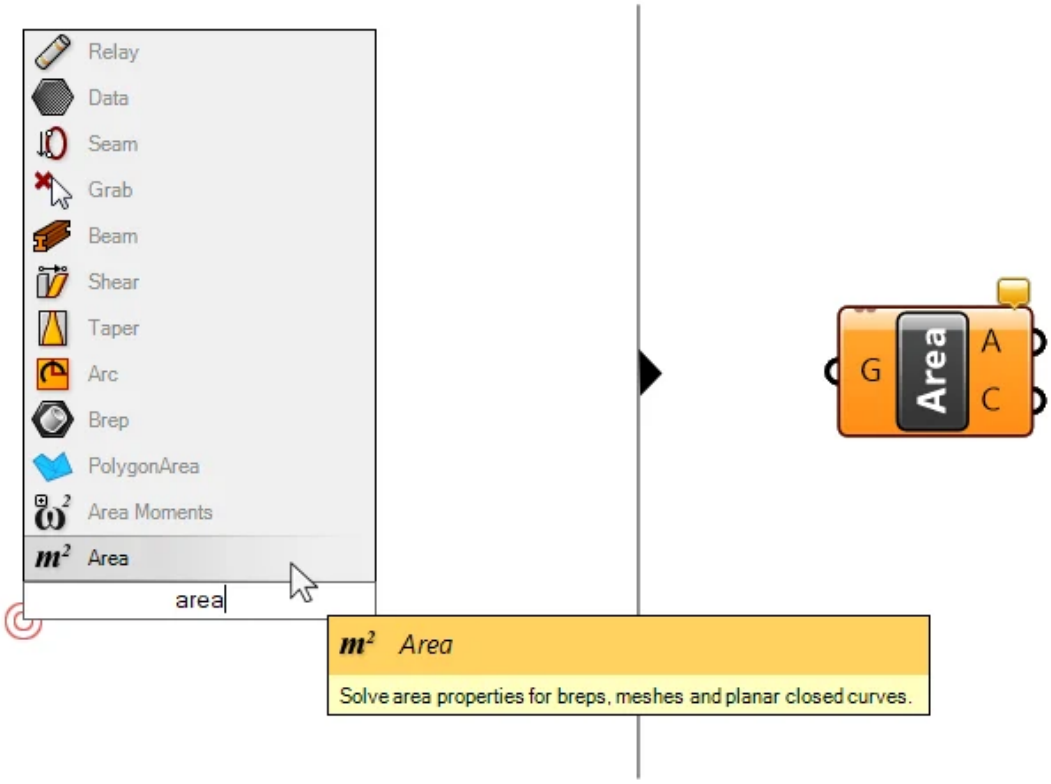
Just like we did for the base plane of our circle, we can specify the base plane with a **point**. We want the rotation point for each segment to be the **centerpoint** of the segment.

Let’s activate the **isolate preview mode** and select the **Move** component. This will only show the preview of the moved surfaces.

We want the rotation of the tower segments to happen around the center of each of these surfaces.



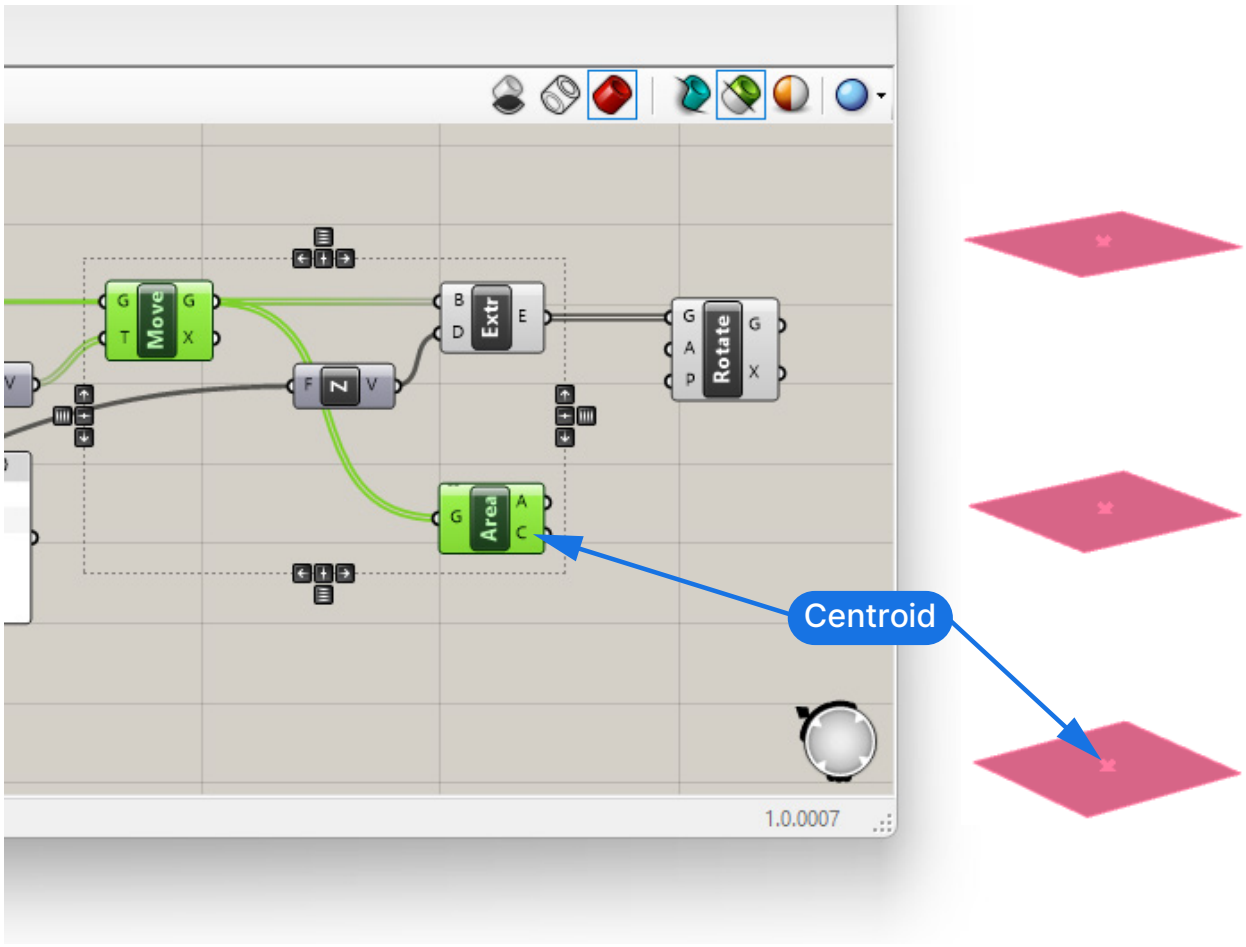
To find the center point of a curve or surface, we can use the **Area** component. Let's add an Area component to our canvas and check out its inputs and outputs.



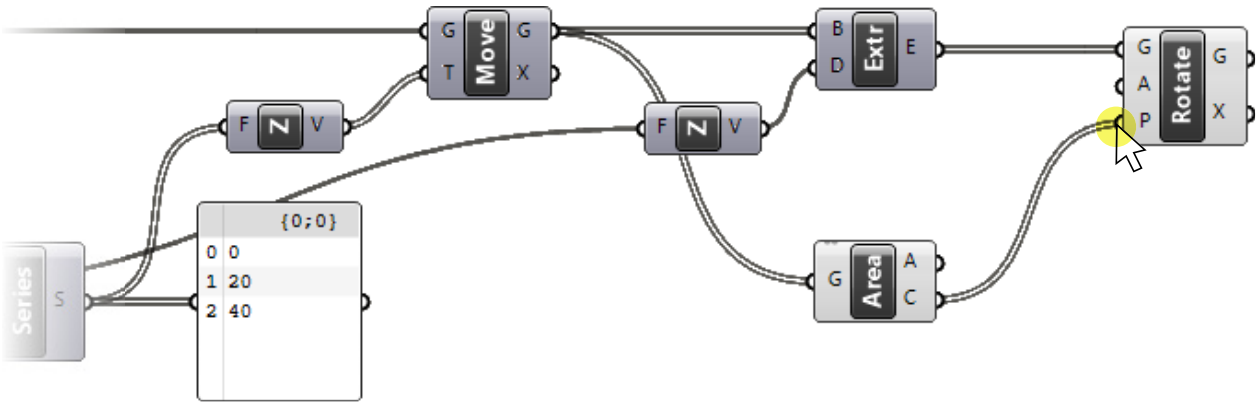
It accepts closed curves and surfaces, basically any **closed, planar geometry** as an input (G). The first output is the **Area (A)**, a number, and the second one is the **Centroid (C)**, the center of the input geometry.

Let's plug our moved surfaces output into the Area component.

When we select both, we can see the surfaces and the corresponding centroids or centerpoints.



Let’s turn the isolate preview mode back off and connect the Centroids (C) to the Rotation Plane (P) input of the Rotate component.



If we now select our Rotate component, we can see that the preview lights up right on top of our existing extrusions. The rotation now occurs around the centerpoints of the tower. But since the geometry is rotated by the default 90 degrees, and we are dealing with a square, we don't really see the rotation.

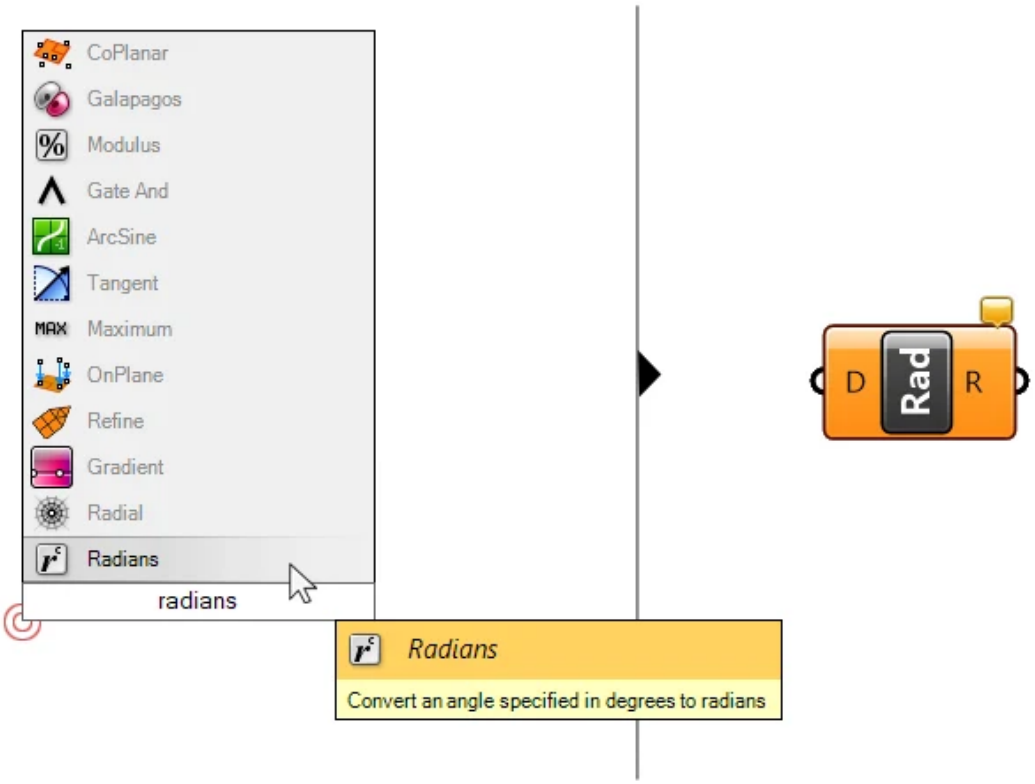
Let's solve that by specifying a custom rotation.

Controlling Rotation: Degree to Radians Conversion

We'll add a **Number Slider** with the value of 45, by typing 45 into the search bar and hitting enter.

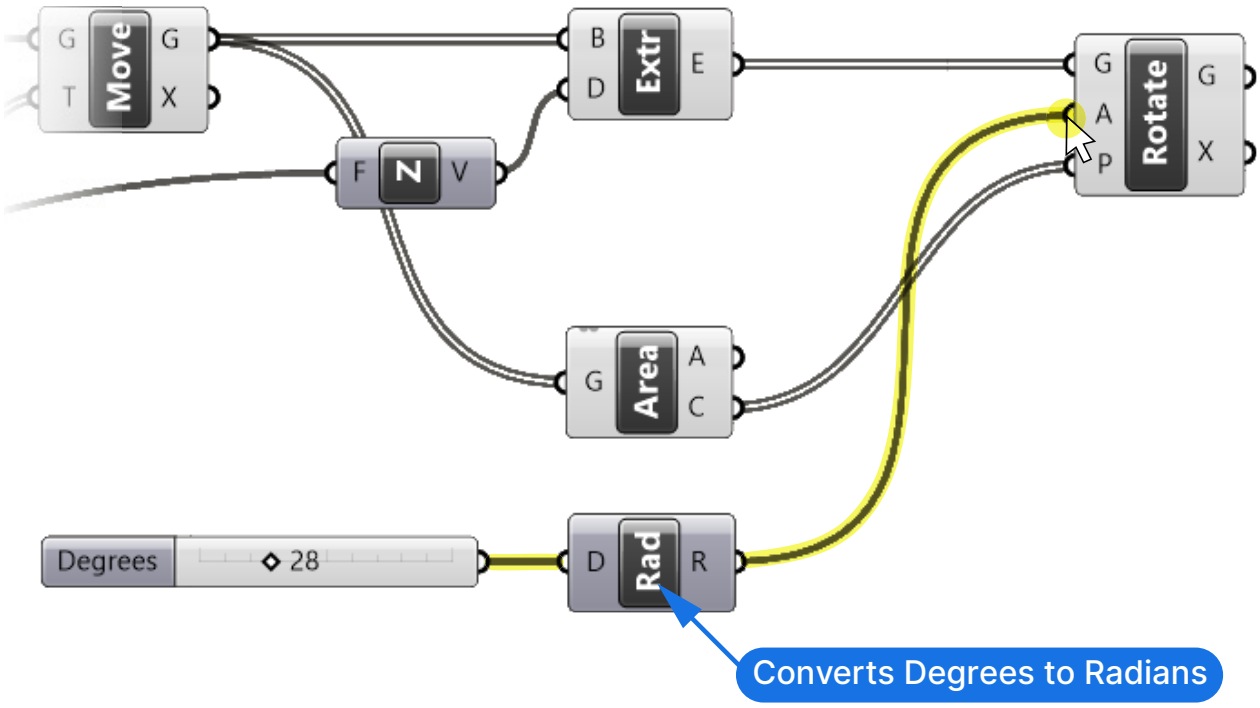
This will be our rotation angle (A) in **degrees**. Let's connect it to the A input of the Rotate component. If we scroll through the Number Slider, the tower rotates erratically. That's because while we are expecting the rotation to be in **degrees**, the Rotate component interpretates it as **radians**.

To convert the number we specify with the number slider to radians, we need a component that converts Degrees to Radians, it's simply called **Radians** - let's find it in the search bar.



We can see from the description what it does: it converts an angle specified in degrees to radians.

Let's connect the Number Slider to the degree (D) input and then the resulting radians angle to the Angle (A) input of the Rotate component.



The number slider value is now correctly converted to degrees and we can control the rotation correctly.

It's hard to tell what's going on in the preview because of the overlayed geometry. Let's turn off the preview of the moved surfaces as well as the extrusions, so we just see the rotated segments.

To do so, select the components and then hit **Ctrl + Q** to turn off the preview or right-click on each component individually and toggle "Preview" off.

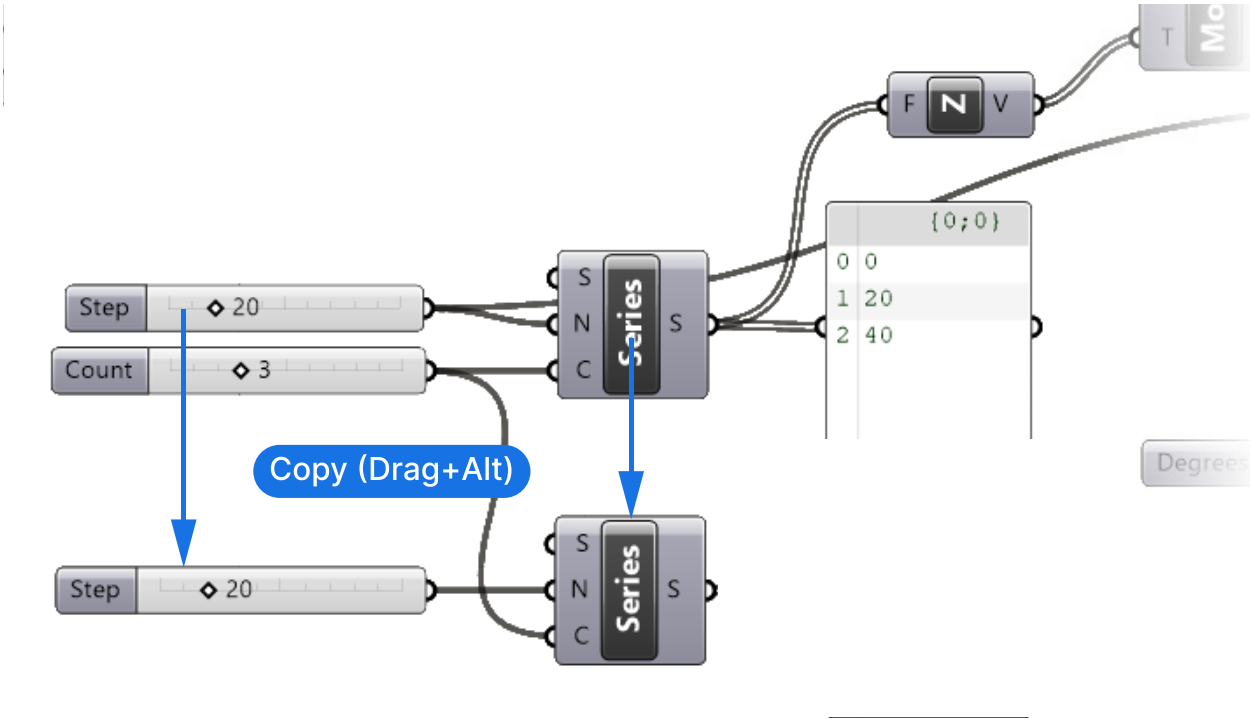
We are now controlling the rotation of the tower with the Number Slider!

Rotating the Tower Segments

So, our rotation is set up, but currently all segments are rotated by the same amount. If we want to rotate each segment by an incremental amount, we need to provide three different angles, one for each segment. Let's use the Series component once more to create a series of numbers.

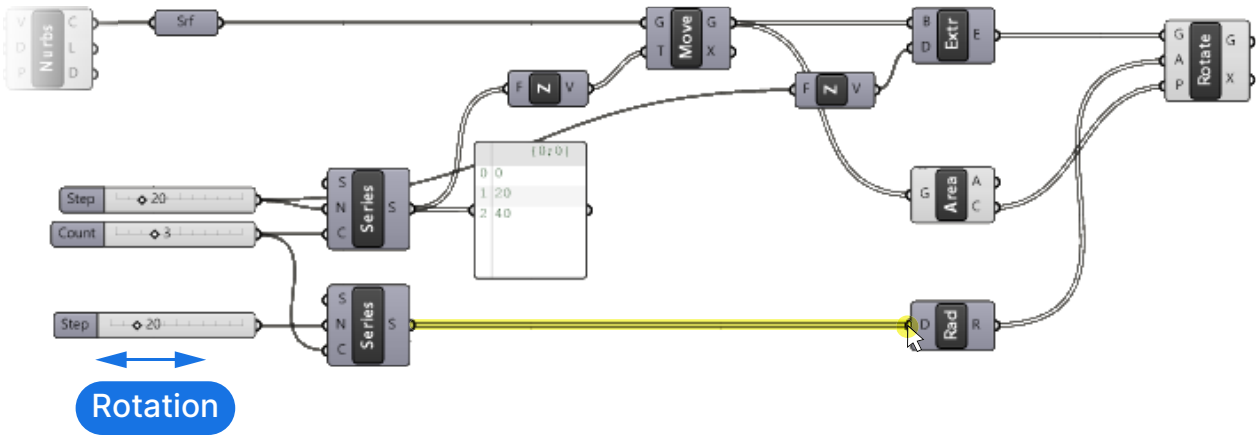
Instead of creating a new Series component, let's **copy** the existing one, by dragging it and tapping **Alt** once, before letting go. When copying a component this way, all the inputs also remain connected.

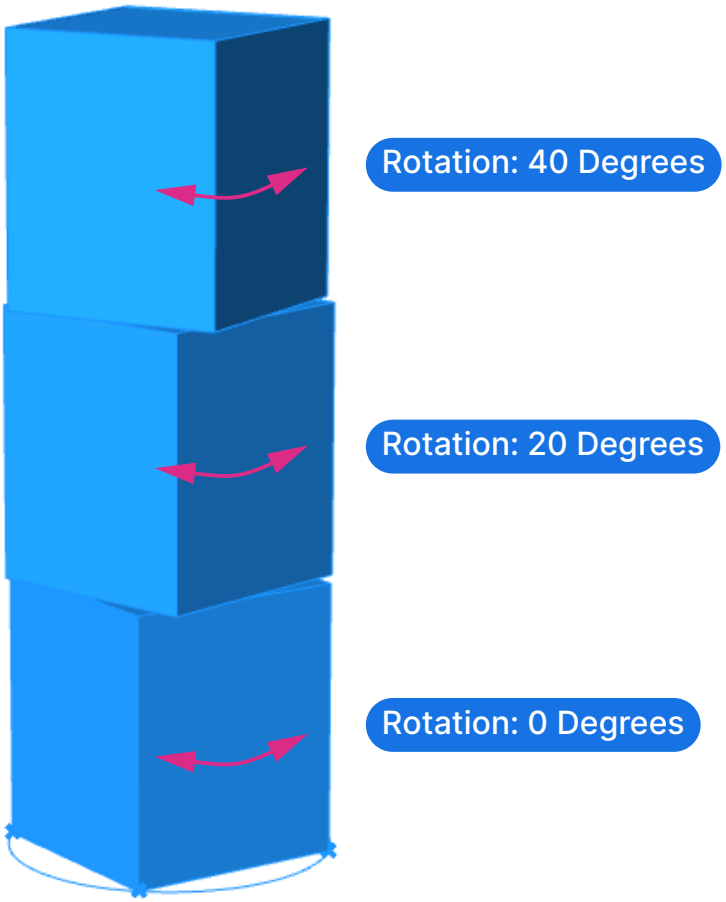
Let's see which one we need to change.



We can keep the default starting value for the series of 0, and also the Count (C), which should match the number segments we create with the other series component, so we have as many rotation angles as we have segments. What we do want to change is the **Step size (N)** of the series. This is the number that will define the rotation. So let's copy the "Step" slider and replace the Step (N) input of our new Series component.

Now we can connect the output of this number series to the Degree (D) input of the Radians component. And remove the old number slider.





By adjusting the step size of our new Series component, we can specify the increment of the rotation angle of the tower segments. Adjusting this number will control the “twist” of the tower.

And here is the power of parametric design: We can now go back and change the values at every step of the script to tweak our design!

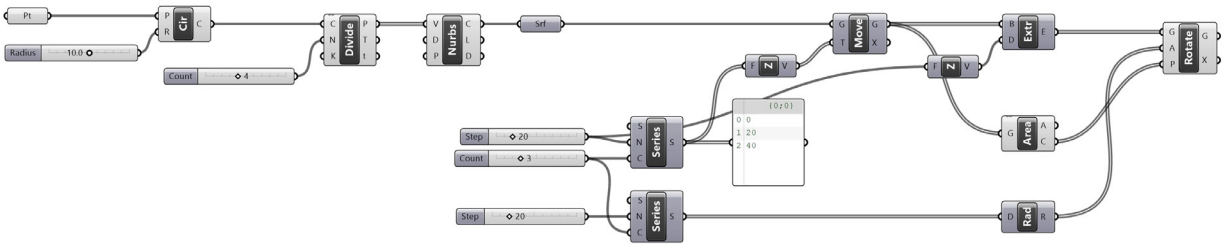
- By changing the radius of the circle, we change the overall footprint of the tower.
- The number of curve divisions defines the shape of our tower segments, a value of three will give us a triangular outline, and six a hexagonal shape and so on.
- We can then adjust the height of the tower segments, the count and the rotation with the Series inputs.

Experiment with the input values to see what kind of result you can achieve!

Once you are happy with the result, you can bake it by right-clicking the final component and selecting **“Bake”**.

Congratulations, you completed your first Grasshopper script!

Complete Script Overview:



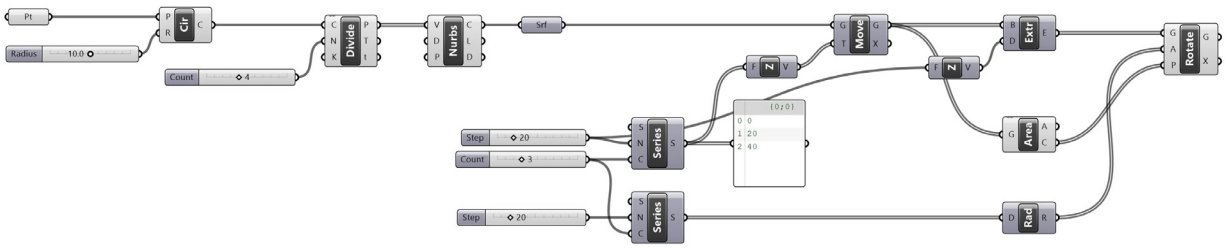
How to Define the Starting Point a Script

A common question when starting with Grasshopper is where to start a script. To reach the result of our script, we don't have to start with a point as we've done in this tutorial. We could directly reference a circle in Rhino instead of creating it in Grasshopper, or start the script with a referenced surface in Rhino.

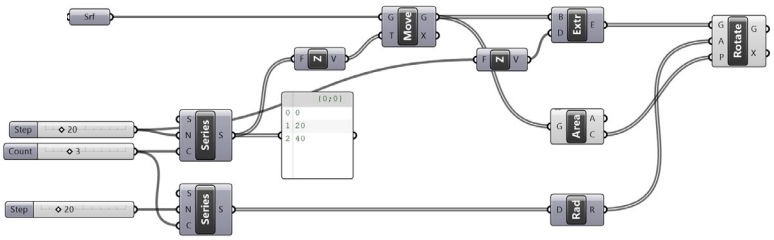
The starting point of our Grasshopper script simply depends on which values we want to be able to change **parametrically**. If we don't need to change the footprint of the tower, we could start the script by referencing a surface in Rhino.

In the script we build, we have complete control over the design: from the location of the tower to its final rotation.

Starting Point: Point



Starting Point: Surface



So what‘s next?

Alright, you’ve built a solid foundation in Grasshopper, but where do you go from here? Grasshopper is vast and multifaceted, so it’s essential to have a structured approach to deepen your expertise. I’ve seen many learners wander aimlessly, feeling overwhelmed. So, based on my experience, here’s a roadmap to ensure you progress systematically:

Stage 1: Embrace Data Management

Here’s where things get intriguing. Understand that everything in Grasshopper operates on data. While it’s tempting to sidestep, a strong grasp of data management can truly elevate your designs. Get familiar with lists, understand data trees and paths, and master the art of data matching. This knowledge is key to unlocking Grasshopper’s true potential.

Stage 2: Master Geometry

With data management under your belt, it’s time to dive deeper into Grasshopper’s geometric foundations. Curves, surfaces, volumes – understand them, play with them. Extract their properties like tangents, normals, and evaluation points.

Stage 3: Algorithms

Now, roll up your sleeves and dive into the world of algorithms. Depending on your domain, this could mean exploring the magic of randomness in architectural designs, creating gradient effects, or even extracting performance data from your designs.

At the end of the day everyone has their journey. Whether you’re using Grasshopper for every design, just starting, or considering its integration, the goal is progress. Keep pushing boundaries, and if you need a deeper dive or more structured guidance, keep reading!

A Faster Way to Master Grasshopper

In this e-book, I've only scratched the surface of what Grasshopper has to offer. There's so much more to learn!

Like any sophisticated design tool, Grasshopper has its learning curve, which prevents many designers from unlocking its potential.
Instead of wasting time reading all the blog posts, watching hours of video tutorials, and getting stuck countless times, why not have me as your guide to learn Grasshopper much faster?
While teaching architects and designers, I've developed a proven step-by-step system/framework to get the results from Grasshopper in the fastest and most efficient way.

- No more guesswork. No wasted hours.
- Proven and efficient process to learn Grasshopper
- Benefit from time-saving templates, and other downloads
- Engage with hands-on tasks and design exercises to refine your skills
- Lifetime 24/7 access to the course material
- Free course updates

This course will let you bypass many of the pitfalls I encountered in the past. Instead of investing months or even years piecing things together, you'll be a professional Grasshopper user in a fraction of the time.
Offering lifetime access, updates, and a certificate upon completion, this course is designed with your success in mind. To discover more, follow this link or click the image below.

Final Words

Grasshopper is a tool that's truly transformed how I approach design. It has saved me a lot of time and helped me become a more agile designer, bringing my designs to the next level. This Grasshopper e-book for beginners has shown you its potential as a bridge between your architectural vision and the tangible, adaptable designs that can be realized.

We've touched upon key features of parametric modeling: we started by getting our heads around the idea of visual programming and how Grasshopper and Rhino work together. Then we jumped into the interface, learned how to define and set design parameters and step-by-step, built a parametric model of a tower. However, this is just scratching the surface.

I genuinely hope this e-book has ignited your curiosity and given you a solid foundation. And if you're eager to improve your skills 5x or 10x faster and become a Grasshopper power user, check out my Grasshopper Pro Course. It's a solid next step.

Keep exploring and designing. The best is yet to come.

This e-book was created by Thomas Jeremy Tait, www.hopific.com
Contact the author via LinkedIn