Multimedia Programming Using Max/MSP and TouchDesigner

Patrik Lechner

Chapter No. 1
"Getting Started with Max"
In this package, you will find:

The author’s biography

A preview chapter from the book, Chapter no.1 "Getting Started with Max"

A synopsis of the book’s content

Information on where to buy this book

About the Author

Patrik Lechner started making electronic music at the age of 16, and soon discovered environments such as Pure Data and Max/MSP. From then on, he developed many tools for his own experimental music, and it wasn't long after this that he started creating generative 3D visualizations of audio material. Since then, he has devoted nearly all his life to real-time audio/video processing and generation.

Patrik worked as an audio engineer for an Austrian TV station for years, and taught Max/MSP both privately and at institutions. For instance, he conducted workshops for the audio engineers of the Burgtheater Vienna, and since 2012, he has been working for the University of Applied Sciences in St. Pölten (FH St. Pölten).

Patrik has worked on many multimedia projects, for example, an installation at the Festspielhaus Baden-Baden for the Institut für Creative\Media/Technologies, FH St. Pölten, and an interactive audio installation in Dubai. As an artist, he did audiovisual performances in Austria, Italy, Germany, Mexico, Canada, and Dubai, and regularly played at the Austrian Pavilion at the world exhibition in Shanghai 2010. He worked a lot with classically trained musicians, developed a real-time scoring system/piece for a string quartet that premiered in 2012, and frequently works with painters and artists from other fields.

For More Information:
I would like to thank the University of Applied Sciences, St. Pölten, for their continued support. I'd especially like to thank Alex Harker, Aya Georgieva, Brigitte Lechner, Camilo Ocampo, Cycling '74, Darwin Grosse, Derivative, Emmanuel Andel, Hannes Raffaseder, Isabelle Rousset, Jakob Doppler, Julian Rubisch, Martina Assum, Mathias Husinsky, Nathanaël Lécaudé, Peter Wyskovsky, Patrick Hollinsky, Pierre Alexandre Tremblay, Sebastian Zeiner, Thomas Seelig (MVC), Wolfgang Seierl (and institutions), and everyone on the Cycling '74 and Derivative.ca forums since they either technically or personally greatly supported this book.
Multimedia Programming Using Max/MSP and TouchDesigner

This book is about the creation of multimedia content with a strong emphasis on real-time generation of content. The two software packages, Max/MSP and TouchDesigner, are chosen as specialized tools to make the generation of audio and video material as flexible and intuitive as possible. Programming inside these languages/tools makes it easy to come up with any tool you might need in order to realize most abstract visions of artistic expression or simply to automate a process that needs to be done regularly. Have you ever lacked control over one of your favorite audio or video effects? Well, let's just build it ourselves! Have you always had an idea for your perfect individual synthesizer? By the end of this book, you'll be able to build it! Both artists and technically interested people are addressed, as the goals of the provided code are not only always headed towards practical needs of multimedia arts but also explained technically.

This book attempts to provide all the necessary tools and crafts in order to enable you to achieve both technical and artistic aims. All of them? Both software packages, Max/MSP and TouchDesigner, are well documented. Trying to replace this documentation of two fast-changing pieces of software would be inappropriate. This book relies on you to consult this documentation, and therefore, the content of this book will go a lot further. While the initial chapters will address people who have never worked with the software, at the end of this book, very advanced topics will be covered. The idea is to not only provide a very profound basis to start with multimedia programming, but also to rely on the documentation and integrated help systems of the software packages, thereby covering as much material as possible. This idea of both providing a solid basis and reaching for advanced techniques also makes it a good repository of concepts, techniques, and best practices. This is even true for me, the author, as I have also learned a lot during the course of writing this book, and have already found myself using it to look up techniques I forgot.

What This Book Covers

*Chapter 1, Getting Started with Max*, will allow us to learn what Max/MSP is about, what we can use it for, why we use it, and how to use it.

*Chapter 2, Max Setup and Basics*, will allow us to set up our custom Max system, go through the relevant settings, and start diving into programming with Max.

*Chapter 3, Advanced Programming Techniques in Max*, starts with building a simple synthesizer, and then covers advanced techniques and introduces topics such as parameter saving, structuring our programs, and how to use Max in a professional manner.

For More Information:
Chapter 4, *Basic Audio in Max/MSP*, starts with how to specifically deal with audio, as the previous chapters dealt with Max in a more general way. Various synthesis techniques, sampling, and some digital signal processing theory is introduced.

Chapter 5, *Advanced Audio in Max/MSP*, takes a deeper look at audio-processing techniques. More sampling, granular sampling, and effects such as compression and reverbs are introduced, and spectral techniques are also discussed.

Chapter 6, *Low-level Patching in Gen*, specifically deals with audio in Max/MSP's Gen. Filters as well as simple physical modeling networks are discussed.

Chapter 7, *Video in Max/Jitter*, explores Max/MSP's video engine, Jitter, which is used to generate some simple 3D scenery.

Chapter 8, *Max for Live*, is about how Max and Ableton Live are a match made in heaven. We'll learn how to make this match for our patches, how to use Max for Live, and how to prepare our patches to be used inside Live.

Chapter 9, *Basic Visualization Using TouchDesigner*, introduces TouchDesigner to create real-time visualizations of our audio processes.

Chapter 10, *Advanced Visualization Using TouchDesigner*, discusses topics such as 2D compositing, time syncing, and 3D rendering.

Chapter 11, *3D Rendering and Examples*, explains in more detail how TouchDesigner is used for more complex and bigger 3D scenes and 3D rendering.

Chapter 12, *Connecting Our Software to the World*, explains how after creating complex programs we might want to connect them to the outside world using sensors, motors, multi-touch screens, and multispeaker setups. Some techniques for addressing such situations are explained.

For More Information:

Getting Started with Max

In this chapter, we will explore the fundamentals of Max. We will see what it is, how to use it, what we can use it for, and what Max is not capable of, or for which tasks it would be cumbersome to use it. You'll understand when it's appropriate to use Max and when it could lead to frustration. Max is quite different in comparison to other (text-oriented) programming languages. It has the strength of being very intuitive as you will see; it can do a lot in real time, so we can get very direct feedback to what we do. In this chapter, we will try to get a feeling of what Max is and what comes with it, and start looking at the general workflow. We will cover the following topics:

- Understanding Max and how it works
- MSP (audio and signal processing in Max)
- Jitter (video and matrix processing in Max)

Understanding the basic concepts of Max

Cycling'74, the company that produces the software, defines it as a toolkit for audiovisual/multimedia expressions that don’t demand much knowledge about programming. In fact, Max is a graphical programming language that lets us avoid the traditionally steep learning curve of text-oriented programming languages to some extent. We simply put boxes into an empty canvas, called a *patcher* or a *patch*, and connect them, patching them together.

**For More Information:**


---

**Downloading the example code**

You can download the example code files for all Packt books you have purchased from your account at [http://www.packtpub.com](http://www.packtpub.com). If you purchased this book elsewhere, you can visit [http://www.packtpub.com/support](http://www.packtpub.com/support) and register to have the files e-mailed directly to you. In case of Max examples, all examples (except for the first two chapters) are provided as so called Max projects. For each chapter, just open the corresponding *.maxproj* file.
Let's compare graphical programming with other representations of code for a minute. Look at this patcher:

Don't bother about the *vocabulary*, that is the object names, too much. This is a special patcher. It is also called a *gen*–patcher, and we use it here since it allows us to see the code generated under the hood. However, this *gen*–patcher is using a somewhat different vocabulary (object names). So, don't try to implement this right away; we'll have a slightly theoretical start for now. Can you already see what's happening?

Imagine a number, say 0, coming into our patcher at [\textbf{in} 1]. You can see that first, we add 1 to our incoming number, resulting in 1. Then, we multiply it with 0.5 (or divide it by 2), resulting in 0.5. Afterwards, we subtract 0.2 and get 0.3, which will be sent to the output of our little patcher. The program we see here doesn't do anything very useful, but it will hopefully illustrate differences in representing mathematical operations. By now, you have seen two representations of what's happening; the last few sentences describe what's happening in the patcher and the patcher itself. In essence, these sentences are like a recipe for cooking. Let's add another equation for reference:

\[
y[n] = (X[n] + 1) \cdot 0.5 - 0.2
\]

Don't be afraid of the notation. For simplicity, you can simply ignore the \( n \) subscriptions, but this is a common notation that you will encounter very often. The \( x \) parameter usually denotes an incoming value, and it corresponds to our [\textbf{in} 1] in our patcher; \( y \) corresponds to the output, [\textbf{out} 1]. The \( n \) parameter stands for a running index. Since we are usually dealing with a sequence of incoming numbers, we have to be able to address this fact. In some cases, we would, for example, like to combine the input with the previous input in some way. To give an example for this case, let's think of an expression that outputs the input plus the previous number's input:

\[
y[n] = x[n] + x[n-1]
\]
You don't have to understand what this is actually doing right now; this is just to make you familiar with another way of representing our code. We will later see how to create an \textit{n-1} term in \textit{max} (a one-sample delay), but now, let's concentrate on another form of representing our first patcher:

\begin{verbatim}
add_1 = in1 + 1;
mul_2 = add_1 * 0.5;
sub_3 = mul_2 - 0.2;
out1 = sub_3;
\end{verbatim}

This might look a bit overcomplicated for such a simple operation. It is the code that Max automatically generated for us when we created our patch. You can see that we are constantly assigning values to variables, for example the variable \textit{sub\_3} is assigned the value \textit{mul\_2} - 0.2, which is referring to the variable \textit{mul\_2} and so on, until we reach \textit{in1}. One can certainly write this program in a more elegant way, but let's stick to this version for now.

Think about the differences in these four representations of a system:

- Mathematical (the previous equation)
- Code (C++, as in the previous code)
- Data flow (Max patch) / block diagram (as in our patcher depicted previously)
- Text (a recipe that explains what to do in natural human language)

Each one of them has its strengths and weaknesses. A mathematical expression is concise and precise. On the other hand, mathematical expressions that describe a system can be declarative (meaning not giving us a recipe to get from the input to output but only describing a relation as it's the case for differential equations). Code, Max patches, and written recipes, on the other hand, are always imperative. They tell us what to do with the input so as to get the output. A Max patch has the advantage that the flow of data is always obvious. We always go from the outlets of an object to the input of another, typically using a line, going from top to bottom. A traditionally coded program doesn't need to be that way. This, for example, looks much like our mathematical representation and does not provide us with an impression of the order of operations as quickly as a Max patch does:

\begin{verbatim}
out1 = (in1+1)*0.5-0.2
\end{verbatim}

It is yet another valid version of our code, a bit tidier than the automatically generated code of course.
So, we can see that one major advantage of Max patching is that we can quickly see the order of operations, just like when we connect the guitar stomp boxes’ input to the output, as shown in the following figure:

I leave it to you to reflect on the differences of these representations in our text recipe.

**Modular basis for expressions**

We saw that Max can create code for us that looks very much like C++. There are some special cases, namely the gen domain, which we will see in *Chapter 6, Low-level Patching in Gen*, in which we can actually see and also export the code that Max is creating from our visual programming. You can think of Max as a high-level programming language in which we put together code we don't quite know. The details of this are both an advantage and a disadvantage of Max, but often, we won't care about the code itself.
We lose some control over what's actually happening, but there are lots of things we don't want to see and don't want to care about in typical multimedia programming. We usually don't want to deal with memory allocation when our aim is to quickly build a synthesizer, for example. A good tool for a certain task allows us to control all parameters that are of any interest for a certain task, not less and not more. For multimedia programming, Max is very close to this objective.

The real power of Max is in its modularity. Think of it like a basis, an infrastructure where you can not only patch but also embed text-oriented programming very easily. Numerous programming languages such as JavaScript, Java, Python, and others can be used within Max if we believe that a task requires these or is simply achieved quicker or better with a different approach than patching. Many people learned, for example, JavaScript simply because they wanted to improve their Max patching, so Max can serve you as a starting point to get into programming in general if you like, but only if you like. Of course, in general, it can be considered a good thing to be able to achieve a result in various ways by using different programming languages because you can always choose, and also because you have the opportunity to get many perspectives on programming methodology, problem solving, and problems themselves.

**When to use Max**

If you think of our previous different representations, you might notice that the last version might be the one that could be created in the fastest fashion. It's simply faster to type the following than it is to put objects in a Max patch, hopefully in a tidy way, and connect them:

```
    out1 = (in1+1)*0.5-0.2
```

If we know exactly what we want to achieve and how to achieve it, meaning we have a picture in our minds of all operations needed to accomplish a calculation, we will typically be faster in a text-programming language than in a graphical one. However, as soon as there is some doubt about how we want to do things, or what our objective really is, a graphical programming language that also doesn't need to compile each time we want to test the result will be more inspiring and faster. We can just try out things a lot quicker, which might inspire us. If you think about experimental music for example, the word suggests it's all about trying things out, doing experiments. With Max, we get our results really fast.
A word of caution should be said though. If we are working in Max, the target is often an aesthetic one, be it music, video art, or dancing robots. If we do so, there is often a fair amount of technical interest or necessity that drives us; otherwise, we could have done the job in a higher-level application. A very common danger is to lose the target of creating beautiful things while programming the tools for them night and day. In this regard, Max is also more dangerous than a Digital Audio Workstation (DAW) but a lot less than traditional programming languages.

It's hard to find general rules when the Max/MSP/Jitter package is the best tool for a problem, mainly because it is highly individualistic. If you just started Max and are a Java professional, it does not make sense to recommend using Max for everything it can do. Mostly, we use Max when it's the most efficient solution, meaning we can get the task at hand done in the fastest way and with the most satisfying overall result within Max. However, there are problems that have a structure that is very close to a Max way of thinking and others that don't. Real-time signal-processing certainly is one of the strengths of Max since it generally follows a block diagram form and is optimized for real-time processing. Don't forget that Max is designed to do signal-processing particularly well. Also, the ease with which we can often design an appealing GUI is an advantage. Problems that are easily solved in other programming languages (but partly can be done in Max) include recursive algorithms, problems that ask for object-oriented programming, database management, optimization problems, large-scale logical systems, and web-related problems. Some of these can of course be solved with a different language and can be embedded within Max.

Max – the message domain

Max is only one part of what you get when you purchase it. The full package actually consists of Max, MSP, Jitter, and Gen (optional). Max was the first building block and will usually serve as our infrastructure and control surface. It will handle data flying through our patches and sequence events, control the others (MSP, Jitter, and Gen), handle user input, handle the GUI, and so to speak the desk on which everything else is lying around. We'll now go through a simple Max patch, and later, a similar MSP and Jitter one, and we'll see what these are good at. For everything else in general, it's a good idea to use plain Max within the Max/MS/Jitter universe. Let's consider our first Max-only patch:
You could say that this patch is made up of the following three elements:

- Input (a button we can click on)
- Processing
- Output (the float number box, `[f1onum]`, at the bottom that lets us see the result)

So `[counter]` counts how many times we clicked on the button; then, we do some calculations on that number (you might see that it's the same calculation we looked at before) and output the result to the screen.

Notice that this patch isn't doing anything unless we click the button. This is an important concept and a major difference between the Max realm and the MSP and Gen~ domains.

We call the message that comes out of the button a **bang**. It is nothing more than an event. You could say it is how one Max object (one of the boxes) shouts to another: now! and the receiving object knows what to do if we patched everything together correctly. In the example given along with the counter, the bang message that comes from the `button` object simply tells the counter to increase the number output by one. You will meet this event-based concept everywhere in Max and it will be worthwhile to understand it thoroughly and have it in the back of your head while programming, but we will come to this again later in Chapter 2, Max Setup and Basics and in Chapter 3, Advanced Programming Techniques in Max.

For More Information:
Max Signal Processing

Without further investigation, we’ll dive into analogies for the example we just looked at in all the other worlds; first, let’s look at Max Signal Processing (MSP):

As you can see, we are still doing the same small calculation. MSP is the audio part of Max, and so all the operations that we are applying are suited for audio signals but can be used for all sorts of things, such as doing this calculation. You can recognize an MSP object by its little tilde (~) at the end of its name. The tilde isn’t used a lot in many languages, which results in partly strange locations on computer keyboards. Please refer to the English Wikipedia entry on the tilde to find a key combination to create a tilde on your keyboard.

This time, our input is an [adc~ 1] object. The adc term stands for analog to digital conversion; it’s just our audio input. Beware, this is a tricky patcher; I’ve hidden many things for didactical reasons, but if you wish, go ahead and look at the patcher itself (by double-clicking on p counter). Instead of monitoring whether a button has been pushed, we are checking whether the audio input level is high. Essentially, you can clap your hands instead of pushing a button (this is a very primitive clap detector though). Again, we count how often we clapped, process that count, and output a number.

The important difference to the Max version is that we are always dealing with streams of numbers in MSP, namely we are getting sample rate number of values every second, even if nothing is happening, for example, adding two constants. Refer to the following screenshot:

For More Information:
Here, we are adding 0 and 0 and as a result, we get 0 of course. Seems like an inexpensive process regarding CPU right? Well it is, but bear in mind that we are calculating $0 + 0 = 0$ 44,100 times per second in my case of a 44.1 kHz sample rate. So, the takeaway message here really is that in MSP, if we have a static input, it doesn't mean that we are not processing, so it doesn't mean we are not crunching numbers all the time. We will learn how to deal with this problem using the \texttt{poly\textendash} object in a later chapter.

This difference in the Max world is one reason why this simple patch became complicated (the things I have hidden in the screenshot). To really stick to the analogy instead of outputting a number, we can have an output sound, for example, a sine wave with the resulting number as audible sound, but let's keep things simple here for now.

To sum it up, we can say that we use MSP when we want to process audio (there are exceptions, though). However, there are other situations in which we would want to use MSP due to the way in which it treats data. MSP processes floating-point numbers with a 64-bit resolution called double precision, whereas Max represents floats with 32-bits (the MSP \texttt{buffer\textendash} object also stores its values \textit{only} with 32-bits). So we essentially have more precision and can represent both bigger and smaller numbers in MSP than in Max. Also, and this is maybe even more important, we not only have a bigger resolution of our values but in time as well. The default scheduler interval of Max runs at 1 ms, so at 1000 Hz it is able to represent signals with a maximum frequency of 500 Hz, in theory. Without going into that theory too deep (Sampling theory and the Nyquist rate), you can imagine that if we process 1,000 values per second, we can't work with signals at higher frequencies. However, that's the job of MSP, and we tend to use MSP for all time-critical processes such as drum sequencers and everything where timing should be really tight. We'll use MSP simply for all high frequency ($\geq 100$ Hz might be a good border here) data that we manage to create in or get into our software.
Jitter, Matrix, and video processing

Let’s take a look at the Jitter version:
Don't be afraid! I know it looks complicated and we won't go over every detail since it's needless to understand everything at this point. The difficulty here is only that if we still follow our analogy, we have to analyze incoming video as the input of our system. It actually still is our senseless small calculation, but since Jitter is made for matrix calculation and especially for video material, we do everything in matrices here. Our clap detector becomes a flash light detector, and doing the actual counting in Jitter is also not a task you should start with when learning Jitter. However, if you look closely, you can see that at the very bottom, there are three [jit.op] objects that are doing the actual processing. Everything else is just to get a trigger signal out of our video input and to also count these triggers within this video context. This is a highly complicated way to achieve our initial goal, but it should show you that we can also calculate anything with matrices. Many things are hidden in there, which you can take a look at later.

Jitter processes are somewhat similar to Max processes. It runs at the scheduler rate (see Chapter 2, Max Setup and Basics and Chapter 3, Advanced Programming Techniques in Max) that Max does, in contrast to MSP that runs at audio rate. Also, nothing is processing if the input is static or if we trigger calculations. Usually, if we are really working with video signals, we want to achieve frame rates between 25 and 60 fps; therefore, it's also similar to how things work in MSP: a stream of data. The difference in MSP is that we are in more direct control of the rate; we have to drive the system with a [metro] object that is sending out bangs at a given rate. In the Jitter context, we will typically use a [qmetro] object. It's the same as the [metro] object with the difference that it waits for other processes (like drawing the last frame) to get completed. Refer to Chapter 2, Max Setup and Basics for the scheduler and priority. In this case, it's our [qmetro] object that is sending out a bang (resulting in the computation of the next frame), each of which is 30 milliseconds; therefore, we are running our computations at ~33 fps (1/interval in seconds = fps or also Hz).

Jitter data format

Jitter matrices are divided into planes. Planes are similar to what one often calls a channel (for example, an alpha channel, a red channel, and so on) in video technique. Let's consider the format of a standard video signal in Jitter; we have an alpha plane, a red plane, a green, and a blue one for video. Each of these planes is a two-dimensional matrix in itself; it has, for example, 1,080 columns and 720 rows, so one cell per pixel. Or you could think of it this way; each cell or pixel, in the case of a video signal, needs to store four values; therefore, we represent each pixel with 4 cells each on an individual plane. This leads us to an imagination that is depicted later; take it with a grain of salt.
The plane count is something different than the dimensions of a matrix. As soon as we have a plane count greater than 1, we can think of it as adding one dimension, independent of what the plane count might be. For a detailed explanation on Jitter matrices, refer to http://cycling74.com/docs/max5/tutorials/jit-tut/jitterwhatisamatrix.html.

The following diagram shows a two-dimensional Jitter matrix with 4 planes illustrated in three dimensions. If our plane count is greater than one, then we can imagine our matrix to have one additional dimension:

Now, since you have an idea of how Jitter handles data, you can imagine that as soon as we are confronted with multidimensional data, it's a good idea to do processing with Jitter (or even with a shader). We have tools to process arrays or lists in Max and MSP, but as soon as data gets two or more dimensions, we'll tend to use Jitter. We can have up to 32 planes and use up to 32 dimensions. Obviously, this data can grow quite quickly; therefore, we have the advantage of having great control over the bit depth, but we'll take a look at this in more detail in Chapter 7, Video in Max/Jitter of this book.
Summary
We have seen that Max is inspired by block diagrams and an I connect one device to another workflow that is reinforcing experimentation and visual thinking. Being familiar with Max helps us sometimes to choose between Max/MSP/Jitter for a certain task or use something different. I didn't outline differences between Max and other visual programming languages, concentrating on multimedia expressions like Max's brother environment Pd, Reaktor, vvvv, Usine, Bidule, SynthMaker, TouchDesigner Audulus, and others. There are simply way too many out there. However, we made some progress on understanding the different territories within Max and what their advantages and disadvantages are. In short, we learned that high frequency or very high-timing accuracy leads us to MSP and multidimensional (>2) data that tends to ask for the use of Jitter. The Max domain itself is here for everything else and often it's used to build bridges between the other environments as well.

In the next chapters, we will dive right into Max, and we'll see how to configure it to our needs and customize it, and get some small projects going. We'll get to know Max a lot more and soon, we will build a simple synthesizer, getting ready for more audio processing.

Exercises

1. Try to think of your projects, ideas, and reasons why you actually want to learn Max and why you bought this book. Take the project apart in your head or think of some bits of code necessary to achieve the whole idea if you put them together the right way. In what environments (Max, MSP, Jitter, or something else) would these be written? Try to think of cases where it's not as clear as in a simple synthesizer.

2. Think about a project that incorporates audio and video. What processes are done in which environment? Draw a simple flowchart and think about where it's best to go from Jitter to MSP or the other way around.

3. Open our MSP counter/processing patch from the beginning of this chapter. Go inside [p counter] (a subpatcher, we'll learn about this shortly) by double-clicking on it. Try to think about what's happening in there and why it's necessary.
Where to buy this book


Free shipping to the US, UK, Europe and selected Asian countries. For more information, please read our shipping policy.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.