TouchDesigner Course

Patrik Lechner

Vienna, March 5, 2020

Introduction/Notes

0.1. Notes

Video playback

0.1.1. Project Ideas

Simple

- Hipster filter
- Audio Reactive Noise Ball
- Toon Shader
- Replicator Experiments
- Oscilloscope/Spectroscope
- leap motion

Intermediate

- „Auto-Cutter“
- Particle stuff
- DMX
- big-data viz

Long Term

- Mapping
- Music Video
- Aiko Aiko
- Procedural Modelling (Bicycle, House, ...)
- (3d) Multi-Touch Interface
- vj-app
- (Light) Show Simulation
- Color Grading Software
- video Analysis

0.2. Course Structure

0.2.1. Lecture 1 introduction and 2D-processing

- students show what they want to do in form of images in a toe file.
- showing off what TD can do, what I did so far.
- Intro to TD
  - getting help
  - the GUI (quick overview)
  - the **OPs**
  - in Depth: **TOPs** and **CHOPs**
  - first guided projects
  - first python expressions (just something like `absTime.frame/100`)
  - a very quick look at GUIs
  - output window handling and MovieFileOut **TOP**. Possibly but rather not: off-line rendering.

0.2.2. Lecture 2, Rendering Setup and Procedural Modeling

- encapsulating Work, **COMPs**
- converting between OP families, and why
- GUIs again
- Select **OPs**
- Render Setup
- some fun SOPs: copy, group, noise, facet, particle, magnet, force, twist, lattice
- feedback **TOP**
- lights, light projection, shadows
- off-line rendering

0.2.3. Lecture 3, Replication and Instancing, compositing and more project structure

- instancing
- particle rendering

- replicator COMP
- Clones
- render pass TOP

0.2.4. Lecture 4

- Scripting
- Projection Mapping
- DATA/API visualization

0.2.5. Lecture 5

Working on the final Project

0.2.6. Lecture 6, project presentation

Contents

1. Lecture 1, Introduction and 2D-Processing

1.1. Notes

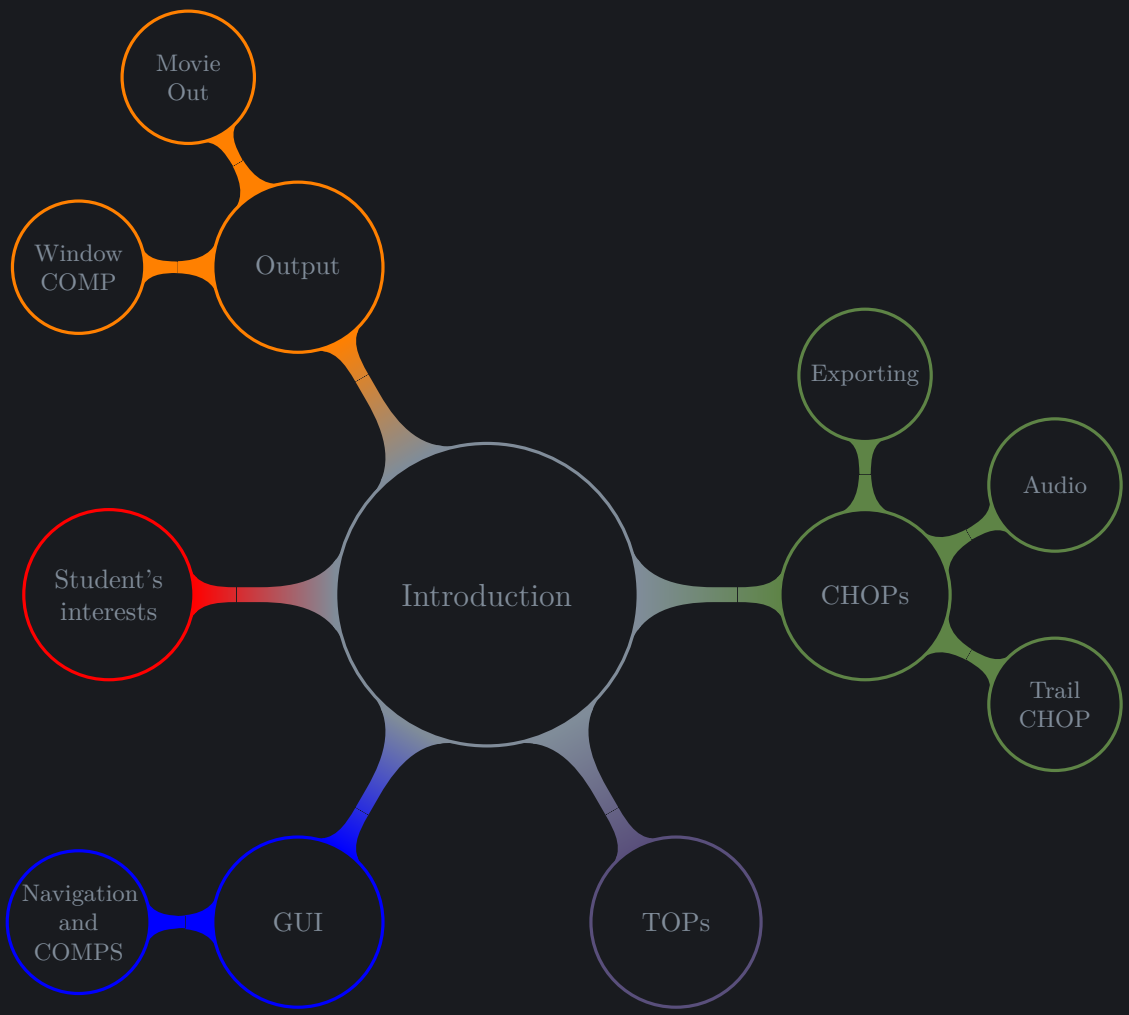

Figure 1.1.: Lecture Contents

1.2. About

About me: my vimeo site[1]
TouchDesigner is coming from sideFX/Houdini. Now made by the company *derivative*

Other products, more or less similar to TD:

- vvvv
- quarz composer
- Max/MSP/Jitter
- processing
- also game engines(Unity, Unreal, ...)

What can we do with TD?

- gravity[2]
- v squares Mapping Reel[3]
- NiN tour[4]

1.3. Getting Help

- Here (in the future hopefully)
- The Book Multimedia Programming using Max/MSP and TouchDesigner[5]
- The open source book introduction to TouchDesigner[6]
- the TD Wiki[7]
- the TouchDesigner OP Snippets. A useful collection of examples for each operator, accessible via TouchDesigner's help menu.
- The TD Forums[8]
- the TouchDesigner Help Group on Facebook[9]

[1] http://www.vimeo.com/titled
[2] https://www.derivative.ca/Events/2014/Gravity/
[3] https://vimeo.com/55818038
[4] https://vimeo.com/groups/touchdesigner/videos/71991897
[5] http://www.amazon.com/Multimedia-Programming-using-Max-TouchDesigner/dp/1849699712
[6] http://book.nvoid.com/
[7] http://www.derivative.ca/wiki088/index.php?title=Main_Page
[8] http://www.derivative.ca/Forum/
[9] https://www.facebook.com/groups/touchdesignerhelp/

1.4. The User Interface

In figure 1.2 we can see the graphical user interface of TD. Two good videos explaining the first steps(like navigating in the UI etc.) are:

- Part A[10]
- Part B[11]

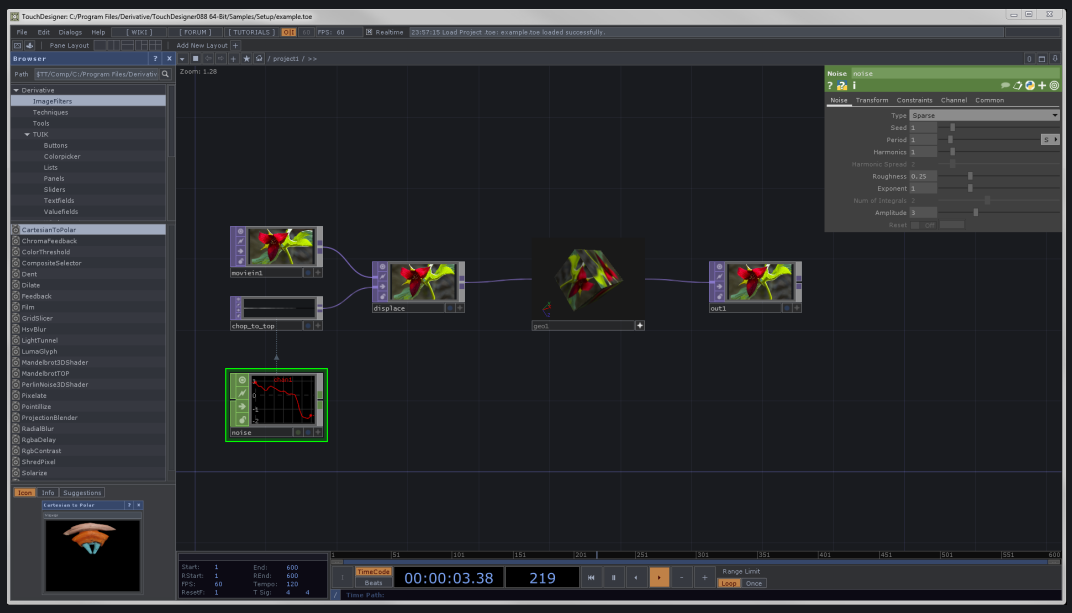

Figure 1.2.: The User Interface

To the left, there is the **Palette Browser** , which holds reusable code, **COMPs**, made by us or the factory. The big area in the center is called a **pane** . Here we can see our **network**, consisting of operators, or **OPs** for short.
To the upper right, we have the **Parameter Dialog**, see also fig. 1.3,which holds all the options for a currently selected **OP**[12].

[10] https://vimeo.com/10056323
[11] https://vimeo.com/10106354
[12] In TD we can select multiple **OPs** using box selection or the shift key. When we select similar **OPs** we can adjust all their values at once in the parameter dialog

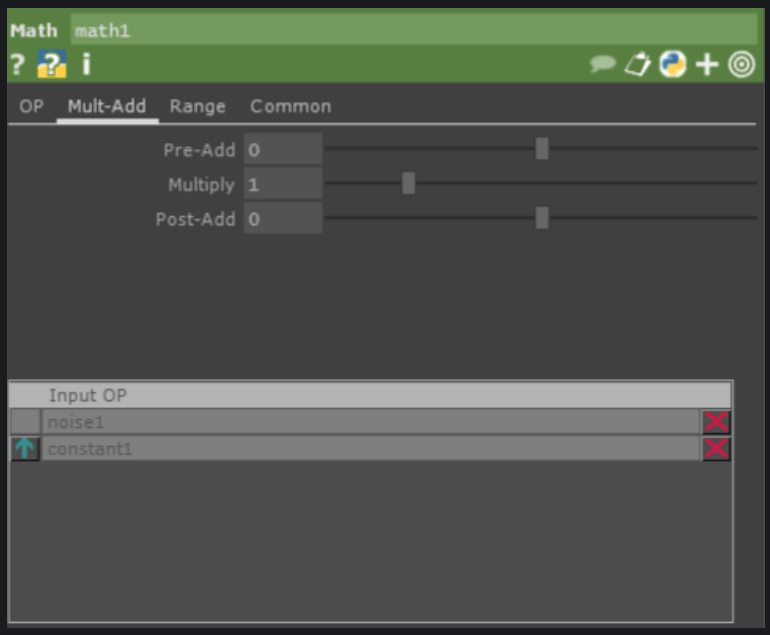

Figure 1.3.: the Parameter Dialog

We can add **OPs** by pressing tab or double clicking in an empty area of the network, to bring up the **OP Create Dialog**, as depicted in fig. 1.4.

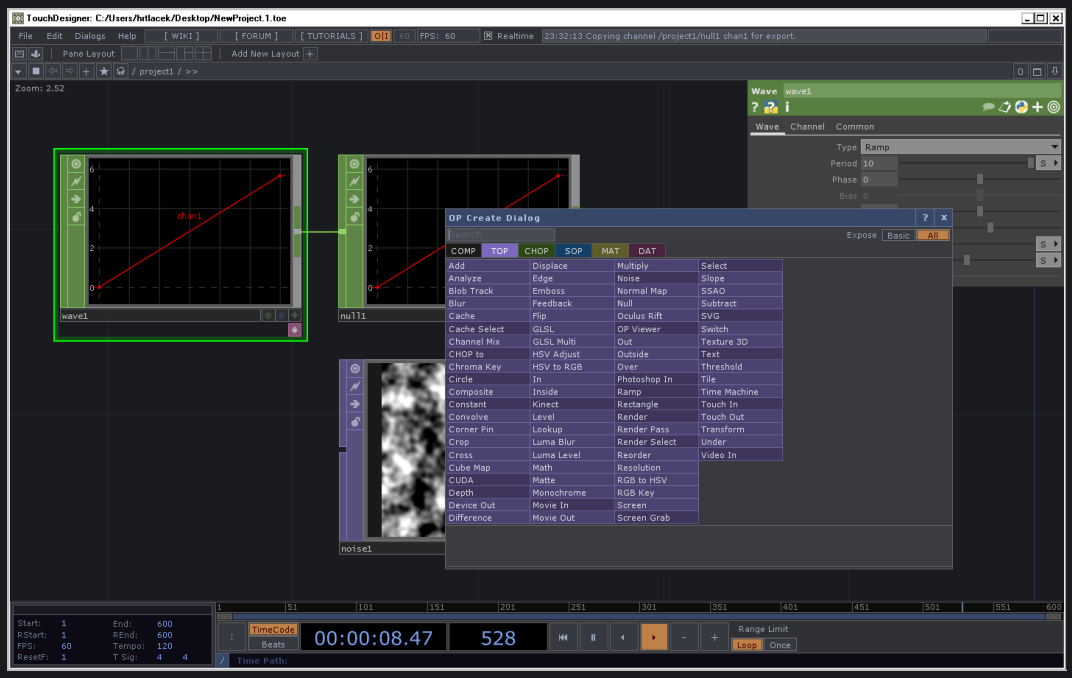

Figure 1.4.: The OP create Dialog

We can connect **OPs** by clicking at outputs and inputs of **OPs**. We can only connect **OPs** that are in the same *family*, so **TOPs** with **TOPs** and **CHOPs** with **CHOPs**[13]. We can disconnect **OPs** by right-clicking at the cord and choosing `disconnect` or by clicking at the inlet of the connected **OP** and dragging away to an empty area.

> If we look closely at the OP Create Dialog, we can see that some of the **OPs** are colored more dark and others are more light. The darker ones are *generators*, so they produce data, the lighter ones are called *filters* so they process data, and need input.

Let's have a quick look at the header of the parameter dialog: The header section in the parameter dialog displays the type and name of the operator and provides a number of buttons for basic operations as described below. The background color of the parameter header indicates the operator's family type.

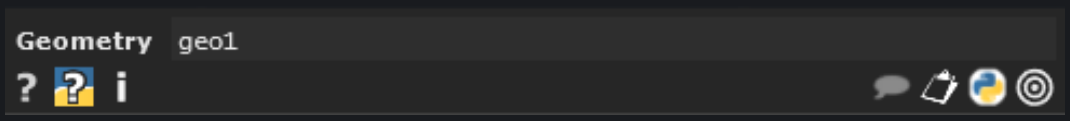

Figure 1.5.: Parameter Dialog Header

Top Section

- OP Type (Geometry **COMP** in this case)
- OP Name

Bottom Section

- Operator Help
- Python Class Help
- Operator Info
- Comment
- Clipboard
- Python/Tscript Operator Language Toggle
- Hide/Show Default Parameters

[13] There are ways to convert between families though, and **COMPs** can have inputs whose family we can define

The most important ones for us at the moment are the Operator Help, the Operator Info and the Show default Parameters buttons:
The operator info shows us important information about the data the Operator holds (such as resolution for **TOPs** or number of vertices for **SOPs**) but also displays errors if we messed up something.

> Usually we access the Operator info just by middle-mouse clicking on the **OP** itself, instead of clicking on the operator info button. That way you always have all the info always at your fingertips.

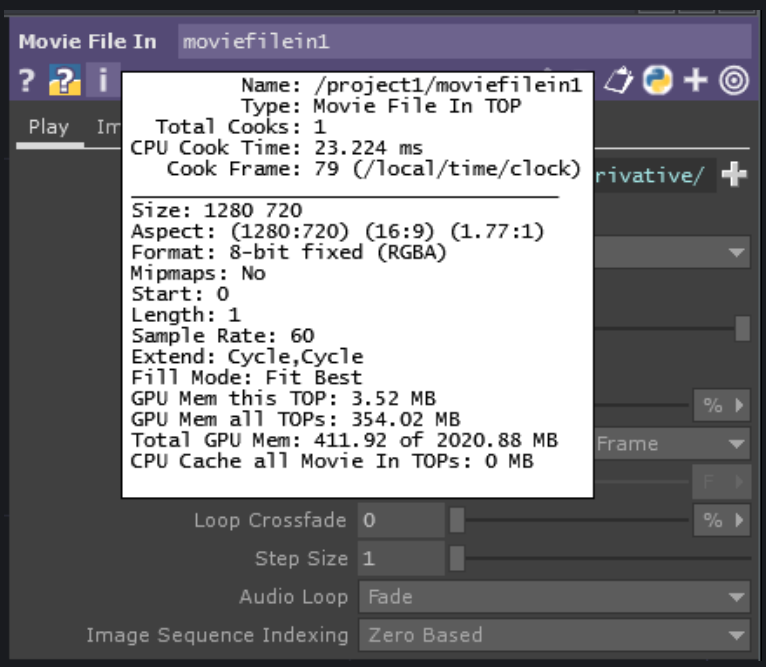

Figure 1.6.: CAPTION MISSING

1.5. The Operator Families

1.5.1. CHOPs

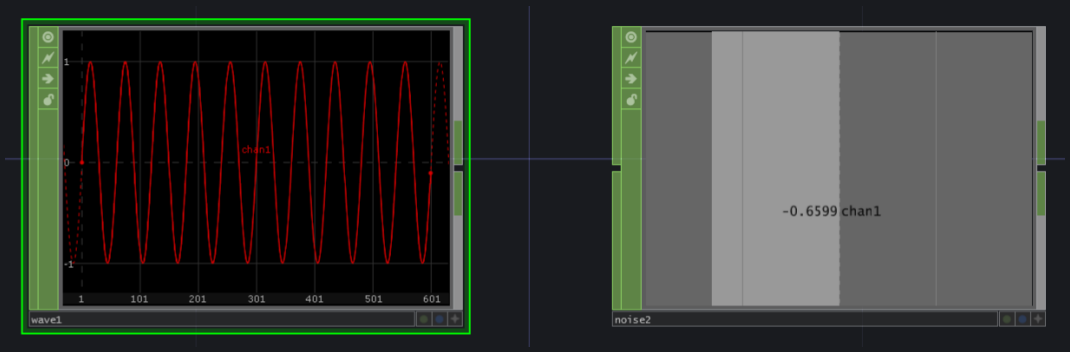

Figure 1.7.: chops

CHOPs are Channel Operators. Generally speaking, we could say, **CHOPs** are there for movement, animation, control of parameters, control signals and streaming numeric data. For example an audio file could be loaded using the Audio File In **CHOP**, an LFO[14] can be created with an LFO **CHOP**. **CHOPs** can contain any number of samples, and any number of channels. Also there are **CHOPs** for OSC in/output, Audio in/output, MIDI, DMX, Kinect, Leap Motion and many other interfaces.

1.5.2. TOPs

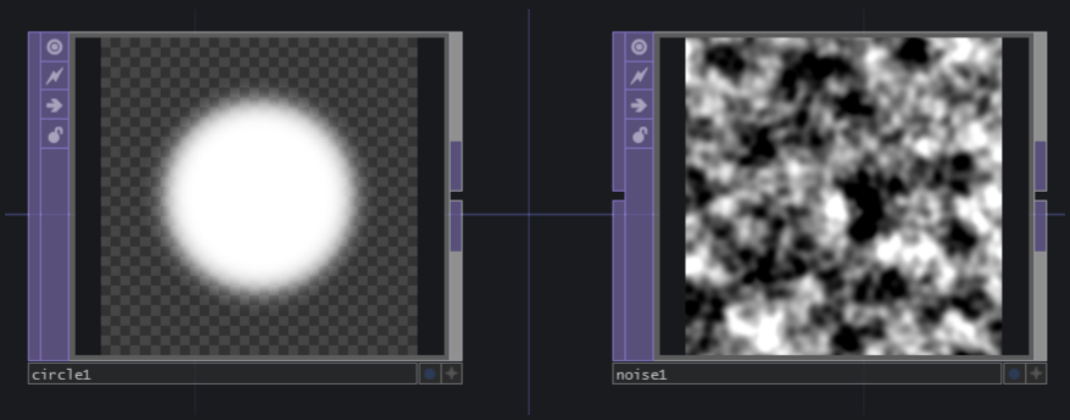

Figure 1.8.: tops

TOPs are Texture Operators. They hold image data, or other data in 2 dimensional/Image form. We find effects for image processing in this category. All of the Texture

[14] Low-Frequency Oscillator

Operators are GPU accelerated, and therefore can be very efficient. Don't forget that images are just data, if we want to use **TOPs** for working on other data than images, that's fine too, just the format has to be right. We will talk about moving between OP Families later.

1.5.3. COMPs

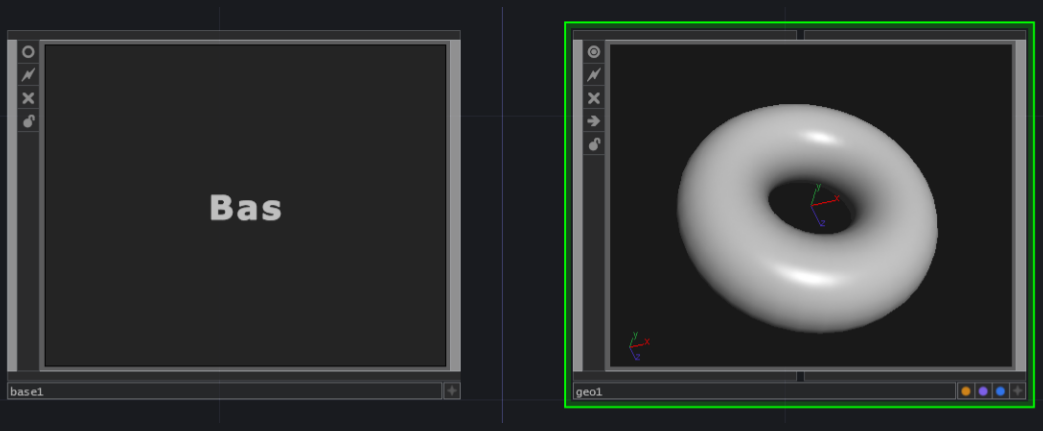

Figure 1.9.: comps

Components, or **COMPs** for short. If you come from Max/MSP or pd you can think of them as subpatchers. But in TD there are many different kinds of subpatchers. **COMPs** can contain other **OPs**. They allow us to build up hierarchies and to organize our networks in a more tidy way. But also, there are specialized **COMPs** for different purposes. If we just want to pack a network into some kind of self-contained box we should always use a Base **COMP** . If we want our result to have a GUI also, we should rather use a Container **COMP** . Later, in chapter 2 we will see specialized **COMPs** for 3D rendering such as a Geometry **COMP**.

1.5.4. SOPs

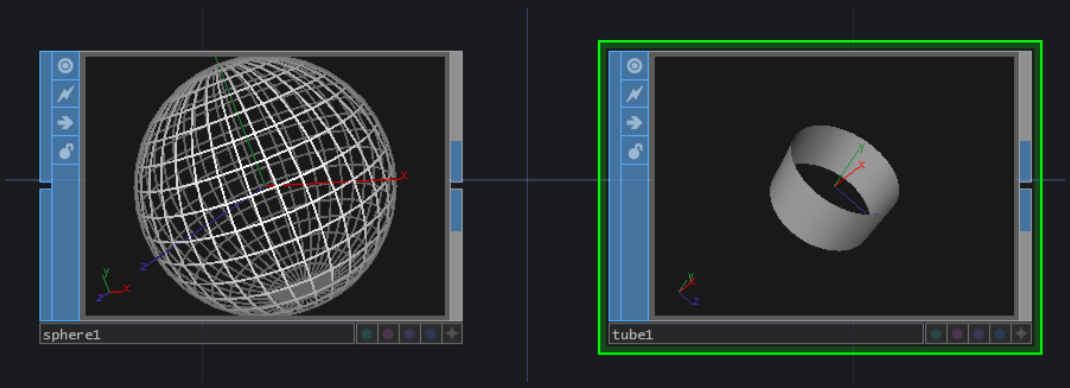

Figure 1.10.: sops

SOPs stands for Surface Operators. They are operators acting in 3d space. They let us model in a procedural way, transform, scale, extrude, twist and edit geometry procedurally. SOPs are processed on the CPU and then sent to a render TOP to be rendered to an image. Their CPU consumption heavily depends on the number of point they are dealing with. Also SOPs can contain different types of data: Polygons, NURBS, Meshes, primitives, bezier.

1.5.5. MATs

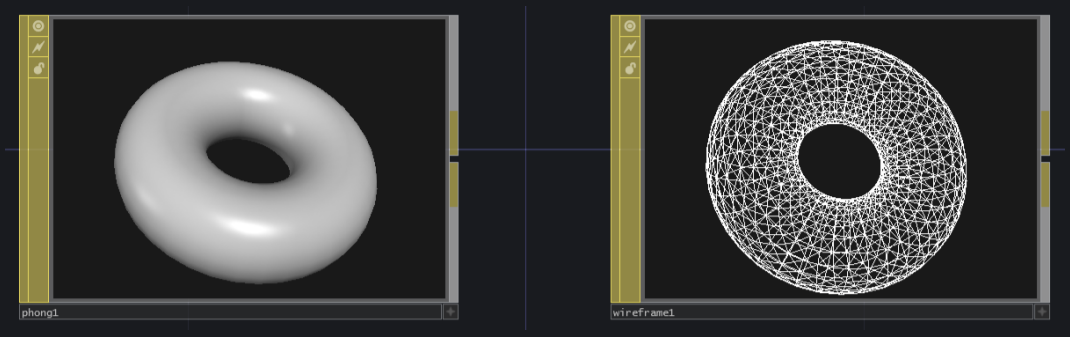

Figure 1.11.: MATs

MATs or Materials are used to shade geometry, so, this determines the look of our 3d objects. We are going to visit them later.

1.5.6. DATs

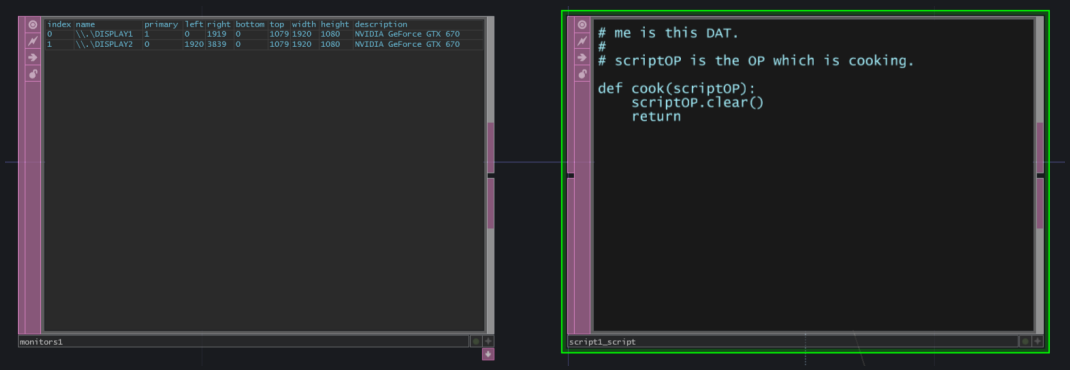

Figure 1.12.: DATs

DATs or Data operators, let us enter, edit and process text, code, tables and they handle incoming data as OSC, UDP, TCP/IP, MIDI etc. as well as output data using these protocols/interfaces.

1.6. TOPs Introduction

1.6.1. Simple 2D Generation and Processing

Effects on Web-Cam input

For a first simple example, let's consider 1.13. The idea is just that we take our webcam (or other video input device) and apply some effects to that stream of images.
As an exercise, look at the block diagram and implement it yourself in TD. The names of the Operators are identical to the labels, and remember you already know what OP family is the one for image processing, so make sure you're looking under **TOPs**. Don't forget to play around with the parameters of each **OP**!

Figure 1.13.: Basic real-time image processing Example.

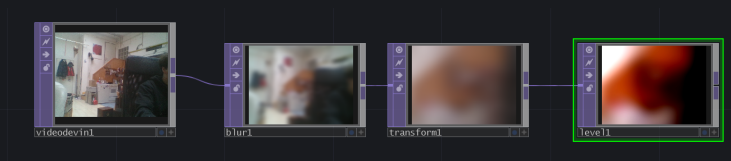

Figure 1.14.: The TD network representing the block diagram above.

Cooking and Resolution Inheritance

Before moving on to more spectacular results such as moving images and 3D scenes, let's make sure we throughly understand how TD manages resolutions. TD in general wants to help us and wants to make clever decisions about optimization. So it will make decisions about resolutions and when it is necessary to **cook** an **OP** .

> The Term 'cooking' in TD refers to processing. This is not specific to **TOPs**. If an **OP** cooks it means it does work, consumes CPU/GPU resources and produces output data. If it does not 'cook' it just outputs static data or no data and does not consume resources. If you look at the cords connecting the **OPs** in our last example, you should see a slight animation, indicating that data is indeed flowing (because our web cam continuously delivers frames to process.) In our next example, this won't be the case.

To explore resolution management, let's build a little network that produces a nice graphic with some simple shapes and some compositing. What we plan to do is depicted in Figure 1.15.

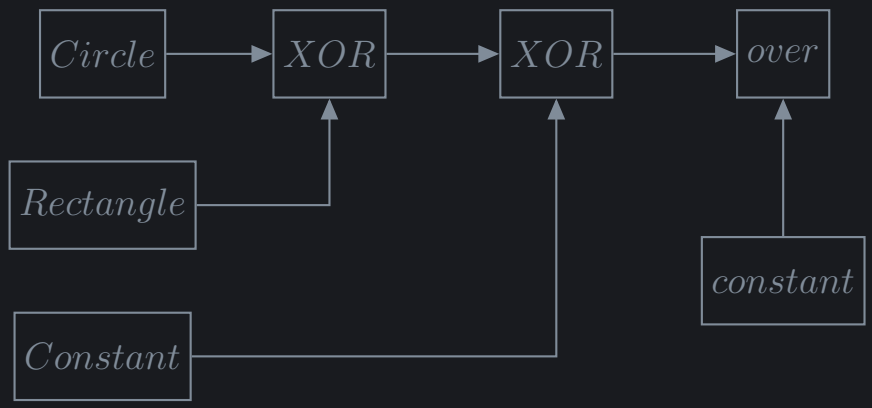

Figure 1.15.: Network for producing a simple geometric graphic

Give it a try before looking at the screen-shot in figure 1.17. The Blocks labeled *XOR* are just Composite[15] **TOPs**, whose mode has been switched to the XOR operation[16]. Play around with the parameters a bit, I adjusted the scale and position parameters of both the circle and the rectangle, and used the rotate parameter of the rectangle to arrive at what's depicted in Figure 1.17.
Now let's look at the resolutions we are operating with. When you middle mouse click on the **OPs**, you can see their resolutions. By default they are all at 256×256. Now

[15] http://derivative.ca/wiki099/index.php?title=Composite_TOP
[16] XOR stands for exclusive OR, a common boolean operation that will be positive if and only if **one** of the two inputs is positive. It will also be False/not positive if both of the inputs are True/Positive.

that's a bit poor, so let's do something about this. All **OPs** have a common page in their parameter dialog. It contains parameters common to that family of **OPs**, hence the name. In figure 1.16 you can see the common page of one of the Composite[17] **TOPs**.

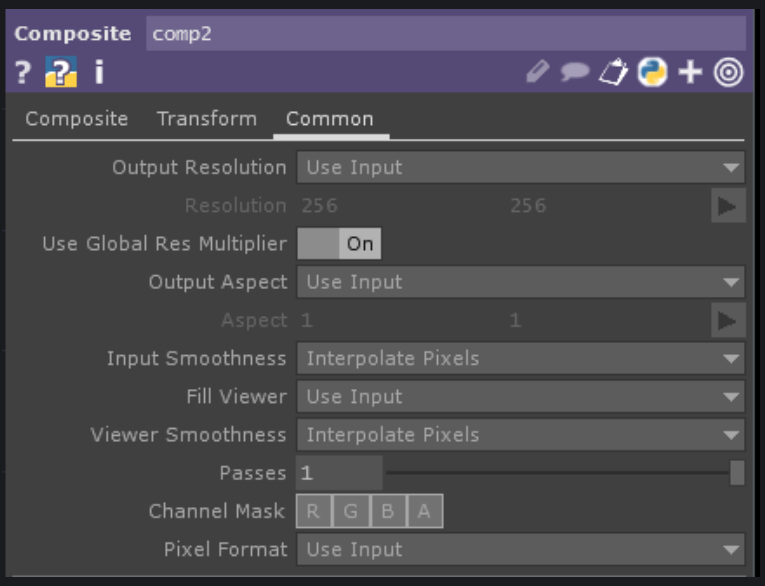

Figure 1.16.: TOP common pars

Since this **OP** is a „filter", so it has an input, it has the option to use its input to determine the output resolution. This will be default for all filter **OPs**.
If you now go to the Circle **TOP**[18], you will find it has a resolution parameter that is set to 256×256. Try changing it to 1024×1024. You will see, if you zoom in a bit, the resolution is quite a bit better, for the circle at least.
The composite **OPs** can take multiple input signals, so which resolution should they take?
By default, always the last **OP**'s resolution is used. So if you also change the resolution of the Rectangle **TOP**[19], the two Constant **TOP**[20]s you will find the whole network is running at a higher resolution.

[17] http://derivative.ca/wiki099/index.php?title=Composite_TOP
[18] http://derivative.ca/wiki099/index.php?title=Circle_TOP
[19] http://derivative.ca/wiki099/index.php?title=Rectangle_TOP
[20] http://derivative.ca/wiki099/index.php?title=Constant_TOP

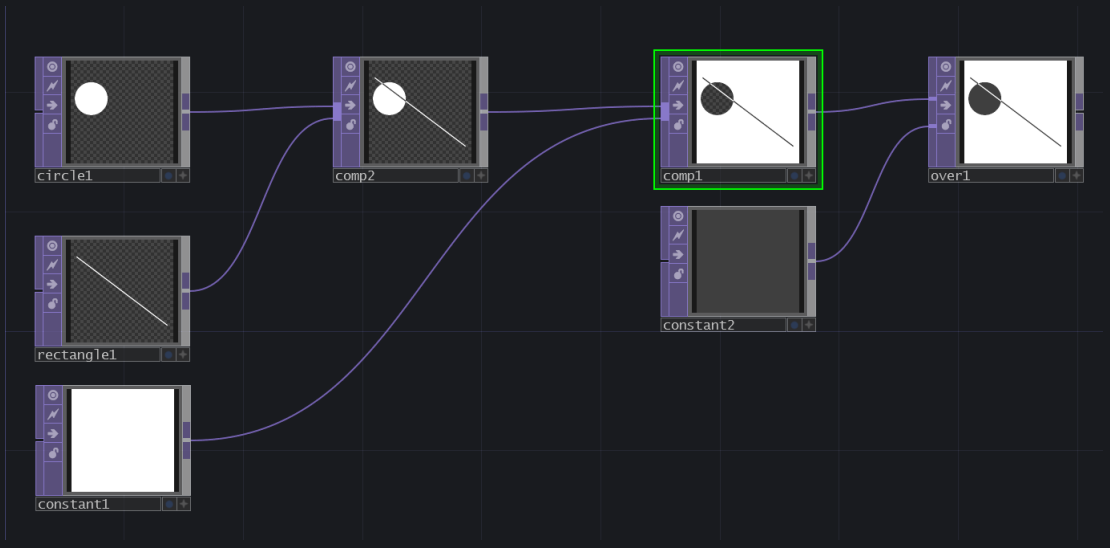

Figure 1.17.: CAPTION MISSING

Exercise: The last example was inspired by this[21]. Try to imitate the other examples there!

TOP common page

Looking at Figure 1.16, you can see a lot of parameters that rather seem technical, such as resolution and bit depth. Don't be fooled, some of these can be really interesting from an artistic/creative point of view. As an example we will explore the parameter `Channel Mask`. For a full description of all parameters, please look at the TOP Generator Common Page Wiki entry[22] or the TOP Filter Common Page Wiki entry[23].
Let's put a Transform **TOP**[24] after our current network. Use its parameters to move the whole image slightly to the right. After that, go to the Transform **TOP**[25]'s Common Page and under `Channel Mask`, switch off the button with the Label 'R'. This deactivates the **TOP**'s effect for the red channel. The result is a commonly used effect:

[21] http://derivative.ca/events/2018/BlokArtwork/
[22] http://www.derivative.ca/wiki099/index.php?title=TOP_Generator_Common_Page
[23] http://www.derivative.ca/wiki099/index.php?title=TOP_Filter_Common_Page
[24] http://derivative.ca/wiki099/index.php?title=Transform_TOP
[25] http://derivative.ca/wiki099/index.php?title=Transform_TOP

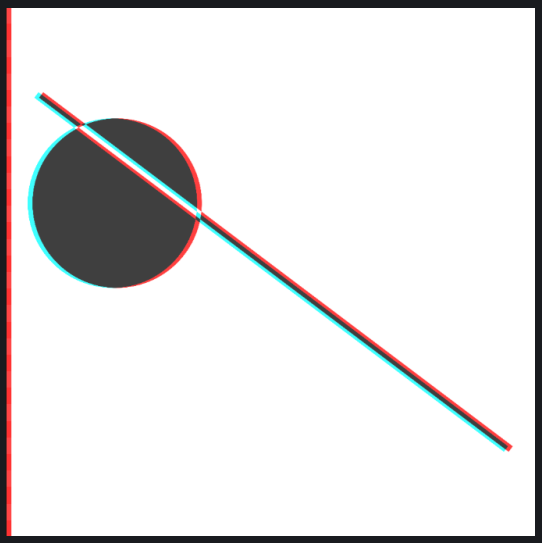

Figure 1.18.: All but the red channel Shifted

1.6.2. Simple Output

So we did process some live input and observed the results by just looking at the **OPs** viewers. But in reality, we need some kind of output, be it saving an image or video to disk, be it full-screen output to one or more projectors, LED walls, or maybe we output DMX to control lights.
As you can see, different situations demand different output techniques, but mainly to keep us motivated, let's just record a quick movie and try out full-screen output to one monitor.

Recording Movies and Images

Let's keep thing simple for now. Use our previous example and put down a Movie_File_Out **TOP**[26]. In its parameters, you can choose if you want to record an image or a movie, you can choose the location where the file will be saved and other parameters such as compression.
Just switch on record and switch it off again when you want to finish your recording. TD allows us to also record 'offline' so it wont loose any frames and takes time to render frames. This is especially important if we built a network that exceeds the real-time performance of our machine. We will get into this topic at a later point of this document.

[26] http://derivative.ca/wiki099/index.php?title=Movie_File_Out_TOP

Full-Screen Output

For this topic, we will build a very simple new network, but finally one with some movement. When looking at the block diagram in Figure 1.19 beware that our block diagrams are slowly going to get more abstract and less explicit. For example there is no *Output* **TOP**[27]. What's meant here is that this is going to be the output of our network, going to the screen. The network is supposed to load and play a movie file and distort it using the Displace **TOP**[28]. This should then be output using a Window **COMP**.

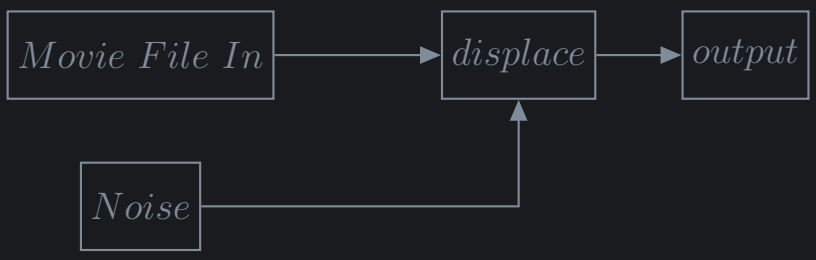

Figure 1.19.: Network for producing a simple geometric graphic

Build up the network and maybe lower the Displace **TOP**'s `Displace Weight` parameter to about 0.06 (both the x and the y value) to make the effect less extreme. Also switch off `Monochrome` of the Noise **TOP**[29].
Finally enter the following expression into the Noise **TOP**'s Translate z parameter as depicted in 1.20: `absTime.frame/100`.

> What does the expression `absTime.frame/100` mean? `absTime.frame` always returns the current absolute frame, so an integer that stands for the current frame number since we started TD. It is an ever increasing number.
> We can use that to translate the Noise. What does that mean? Noise is a collection of quasi-random values. These values are computed along a domain, and this domain can be shifted. Translate the noise **TOP**'s output in X or Y and you will immediately see that it just 'moves' the image left/right or up/down. Moving in Z moves in another dimension, seemingly creating a **continuous** set of new random values. Continuous means that the image does not 'jump', it gradually changes, which is often what we want.
> The division, so `/100` just slows down this behavior since that movement happens 100 times slower.

[27] There is an Out **TOP**, but it isn't meant here and we are going to see what it does soon.
[28] http://derivative.ca/wiki099/index.php?title=Displace_TOP
[29] http://derivative.ca/wiki099/index.php?title=Noise_TOP

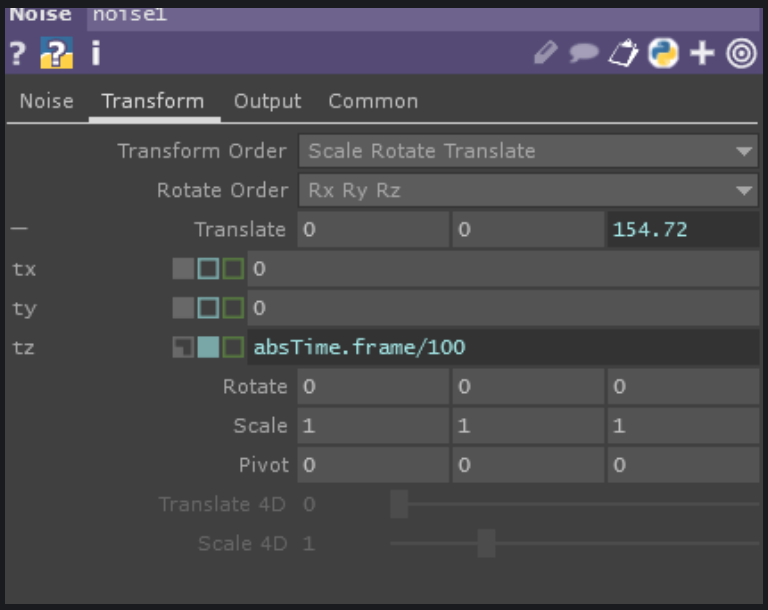

Figure 1.20.: Expression in Noise TOP

Figure 1.21.: CAPTION MISSING

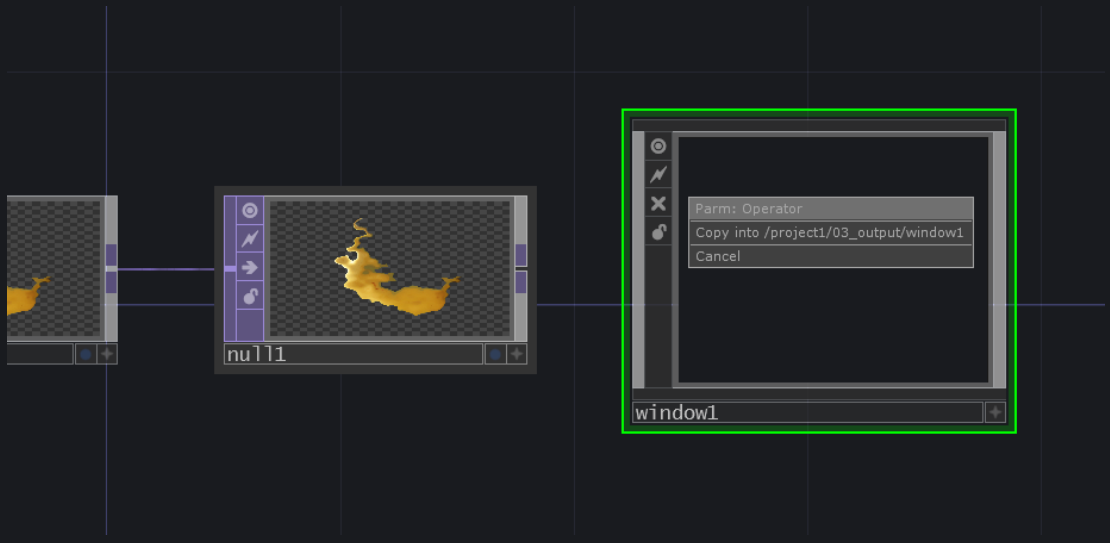

Figure 1.22.: CAPTION MISSING

Figure 1.23.: CAPTION MISSING

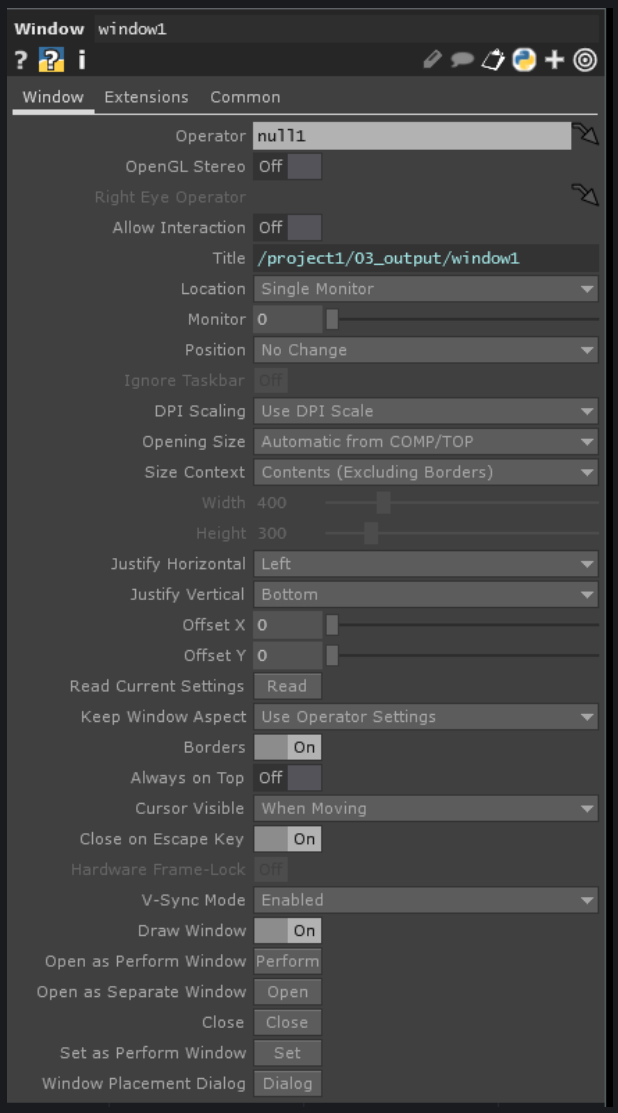

Figure 1.24.: CAPTION MISSING

1.7. CHOPs Introduction

- Audio Device In **CHOP**
- Audio Device Out **CHOP**
- Audio File In **CHOP**
- Audio Filter **CHOP**
- Analysis **CHOP**

1.7.1. CHOP exporting

1.7.2. Audio Analysis

If we want to control the appearance of something, be it geometry/visuals or dmx/lights by an audio signal, we generally use the structure depicted in Figure 1.25.

Figure 1.25.: Using Audio to control visual parameters.

Audio digital signals are numbers. For a stereo signal, we get always 2 number in any moment of time. Also we typically get 44100 numbers per second per channel. So we get a lot of numbers, be it from a microphone, a sound card input or a file.
These numbers themselves are not directly suitable to control something video related. Let's say we ignored on channel, so we get 44100 numbers per second. Let's assume we take these numbers to control the brightness of some video. First of all we will miss most of the action since our screens produce only 60 images per second if we are lucky. So we miss most of the 44100 values per second. And if we have really bad luck, these 60 images are all produced when the audio wave is passing through 0 or negative (audio signals typically move between -1 and 1). So our audio input could be really loud and our output could stay dark.

So we need some idea **what** we want to visualize. Loudness? The loudness of the bassdrum? The whole spectrum? We could think of many things and we will need some way to measure these. This process of measuring something about the audio is called *Feature Extraction*. For a simple measure of loudness we could use the *RMS*.

The next step might be to map this extracted measurement to a parameter, let's say the sized of a sphere. The linear RMS will range between 0 and 1. Now if we don't want the sphere to be 0 when the audio is silent, we will need to apply a *mapping*. This just means that we take our RMS and by multiplication and addition, bring it in another range, for example we could add 1. Then our sphere will have a size of 1 in case of silence and double the size in case of maximal input RMS. In TouchDesigner, the Math **CHOP** helps us a lot with this process. On its second page it offers us to map one range to another. We just have to tell it our input and output range and it does the remapping.
Beware that we could also apply a non-linear mapping, for example squaring or thresholding. We could look if the input is higher than some particular value and only then doe a certain action.

1.7.3. Simple Spectrum Display

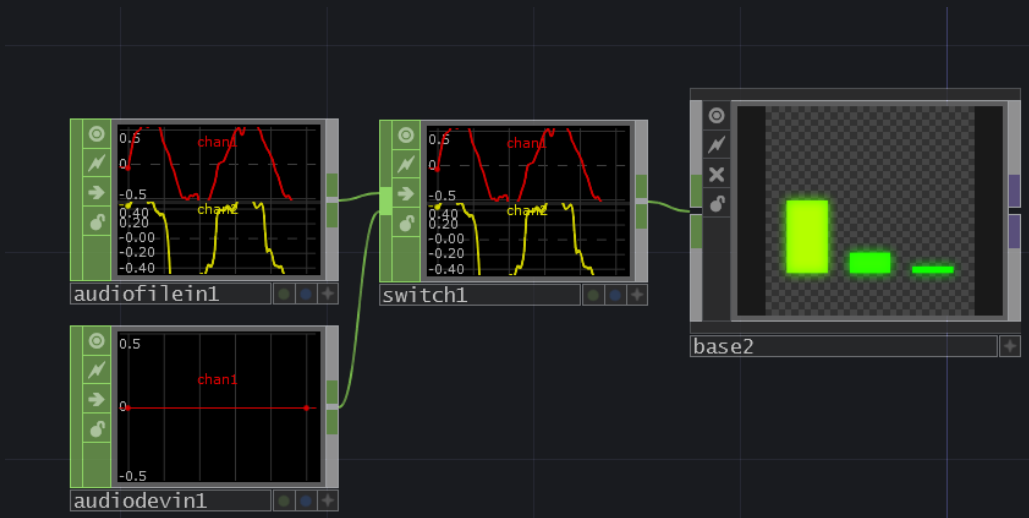

Figure 1.26.: CAPTION MISSING

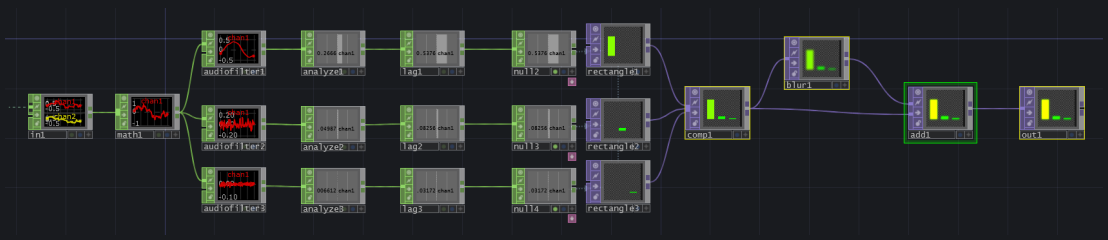

Figure 1.27.: CAPTION MISSING

2. Lecture 2, 3D Rendering

2.1. Notes

TOPs:

- feedback + scale
- simple reaction-diffusion

Panel COMP -> Make TOP network interactive
3d Render Setup
Classic torus + noise
Panes: 3d viewport
feedback+CircleSOP+noise SOP+Panel interaction
CHOP to SOP (oscilloscope/Spectroscope)
converting between OPs
render setup
SOPs:

- Torus
- Grid
- Sphere
- Noise
- facet
- transform

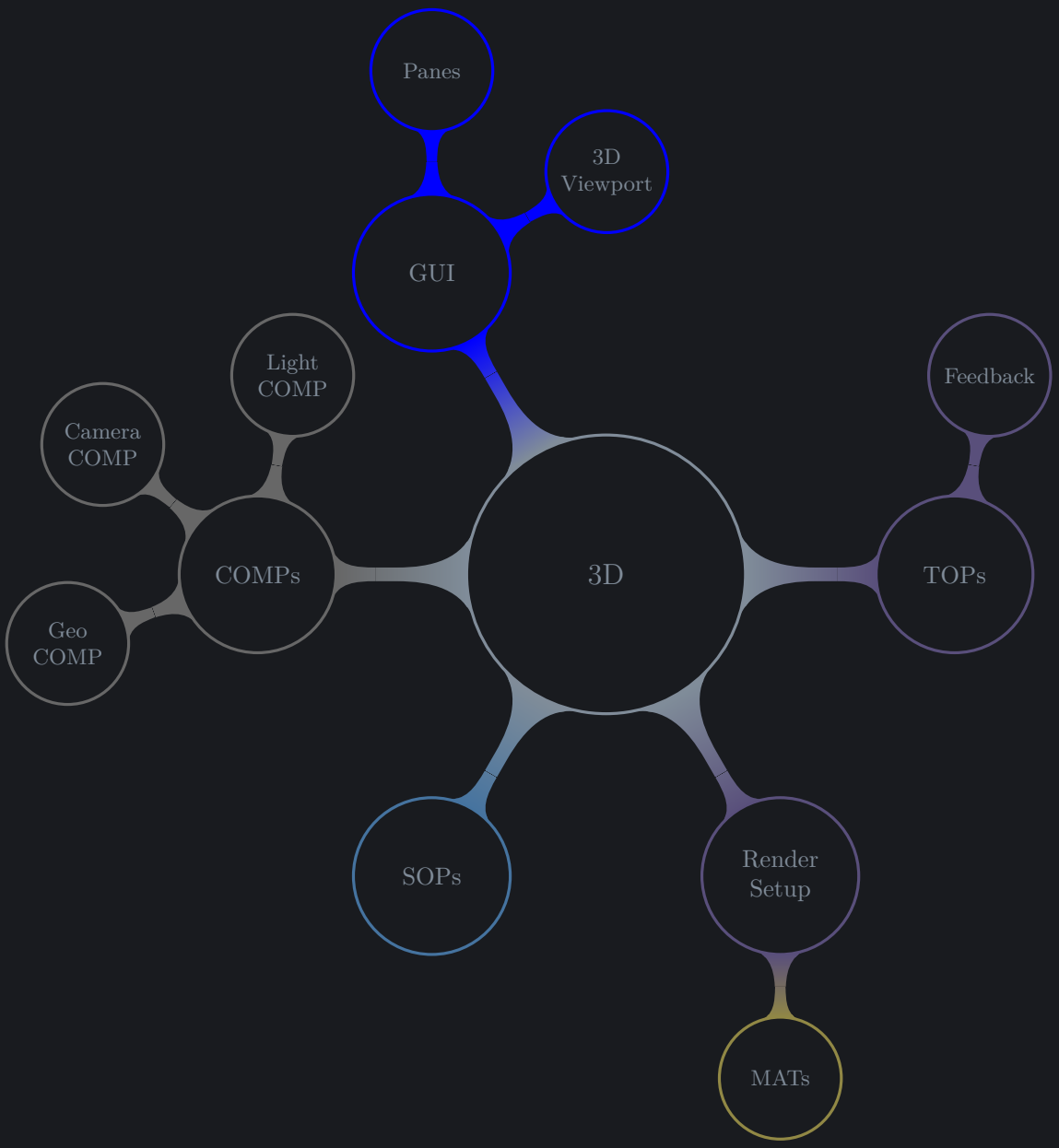

Figure 2.1.: Lecture Contents

2.2. More TOPs

2.2.1. Feedback Networks

Reaction-diffusion

Karl Sims Tutorial[1]

[1] http://www.karlsims.com/rd.html

Figure 2.2.: CAPTION MISSING

2.3. Basic Render Setup

For a minimal 3D render setup we need 4 things:

- Geometry **COMP**
- Camera **COMP**
- Light **COMP**
- Render **TOP**

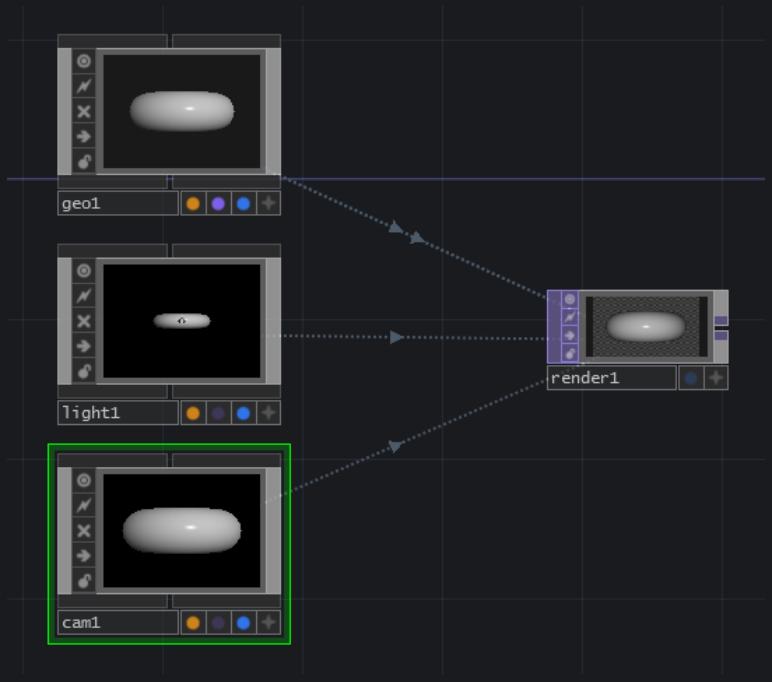

Figure 2.3.: A very basic 3D rendering setup

2.3.1. SOP Flags

2.3.2. MATs

2.3.3. Basic SOP manipulation

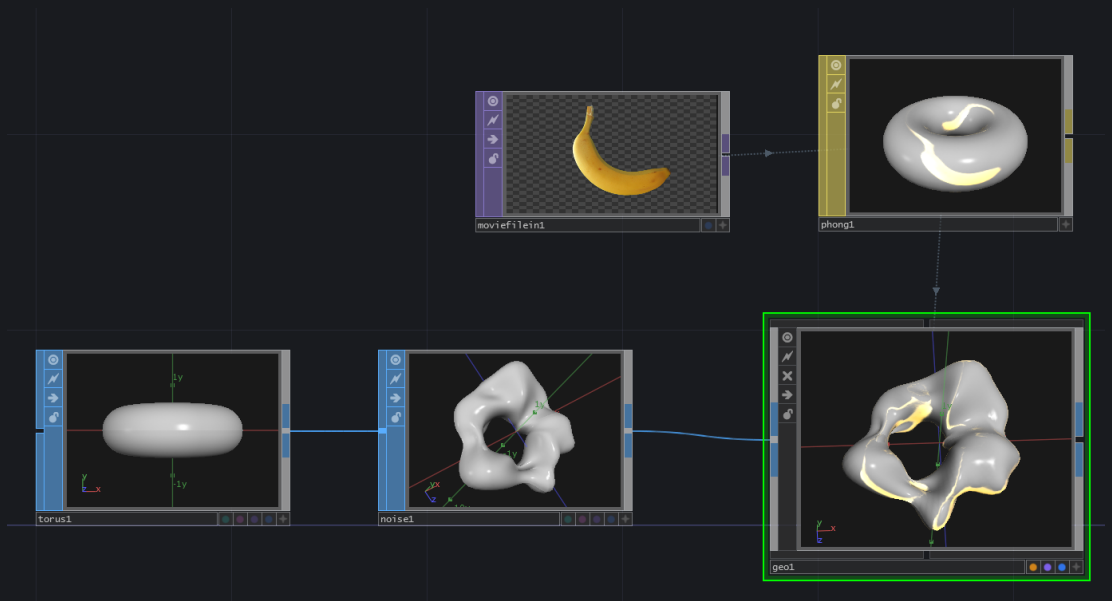

Figure 2.4.: CAPTION MISSING

2.3.4. The data inside a SOP

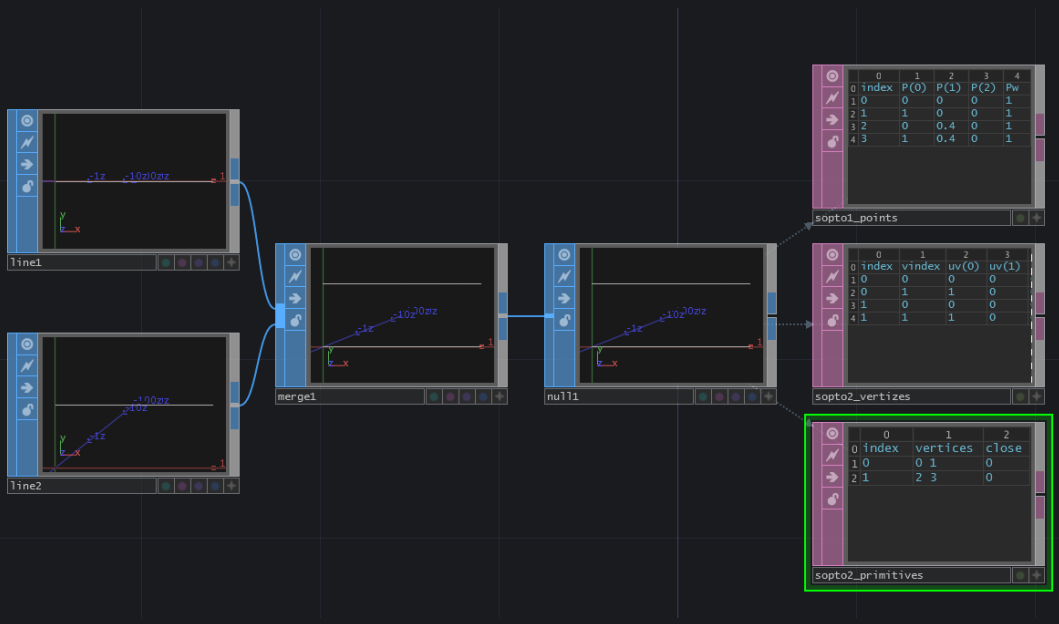

Figure 2.5.: CAPTION MISSING

What's the difference between a point and a vertex? To quote the TouchDesigner wiki[2]:
Each SOP has a list of Points. Each point has

- an XYZ 3D position value
- Other optional standard attributes like RGB color, alpha, texture UV and W, Normal XYZ.
- user-defined attributes.

Each polygon is defined by a vertex list, which is a list of point numbers.
So a point describes a position in 3D. A vertex is a building block of geometry that references a point. One point can be referenced by multiple vertizes.

3. Lecture 3

3.1. Notes

- **SOPs** explanations: Transform, Merge, Copy, Subdivide, Group
- Geometry Types, SOP to **DAT** for inspection
- Panel Interaction + Generative Design Example
- Instancing, SOP to **CHOP**
- Oscilloscope
- Trail **CHOP** + instancing

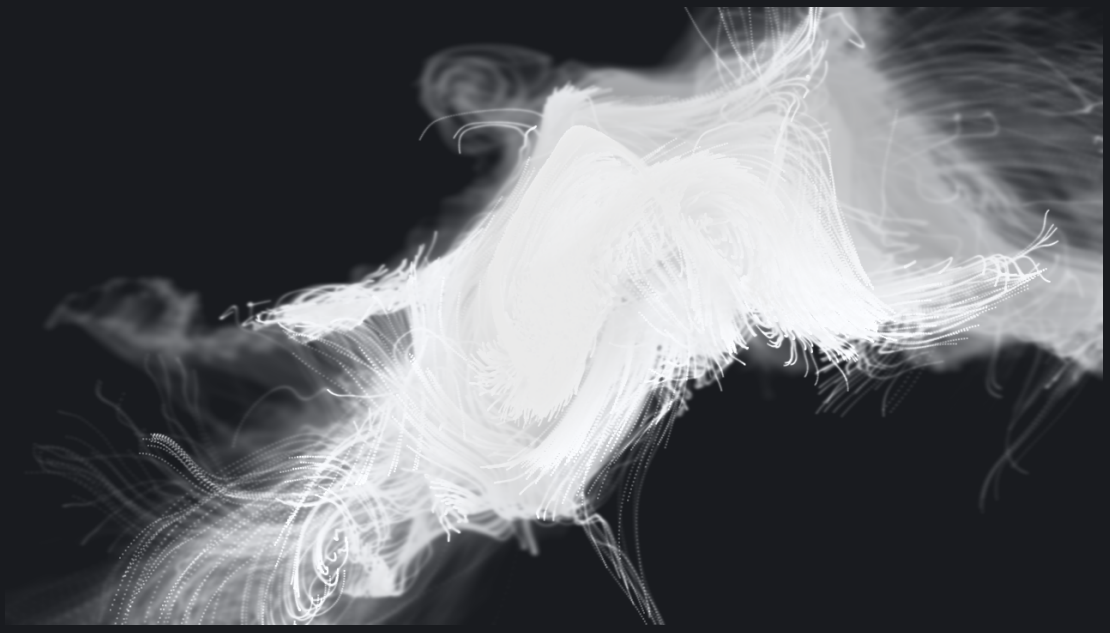

Figure 3.1.: Particle system with feedback

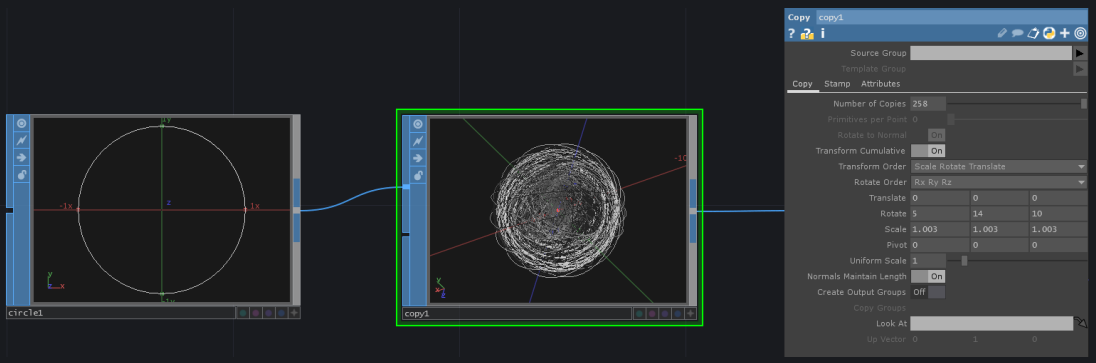

Figure 3.2.: CAPTION MISSING

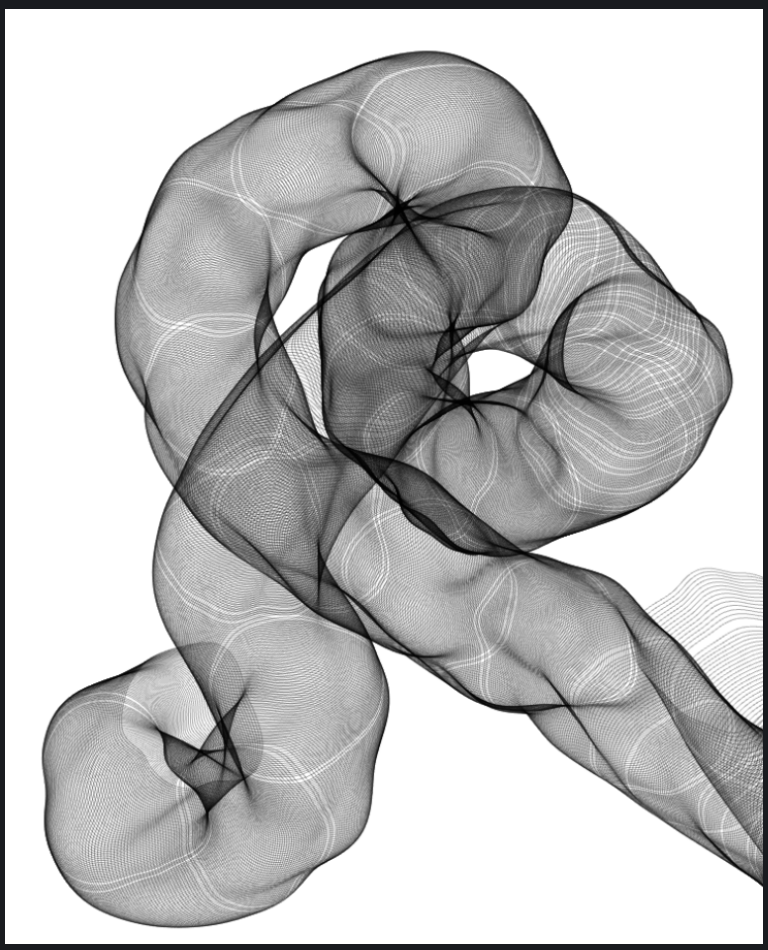

Figure 3.3.: CAPTION MISSING

Figure 3.4.: CAPTION MISSING

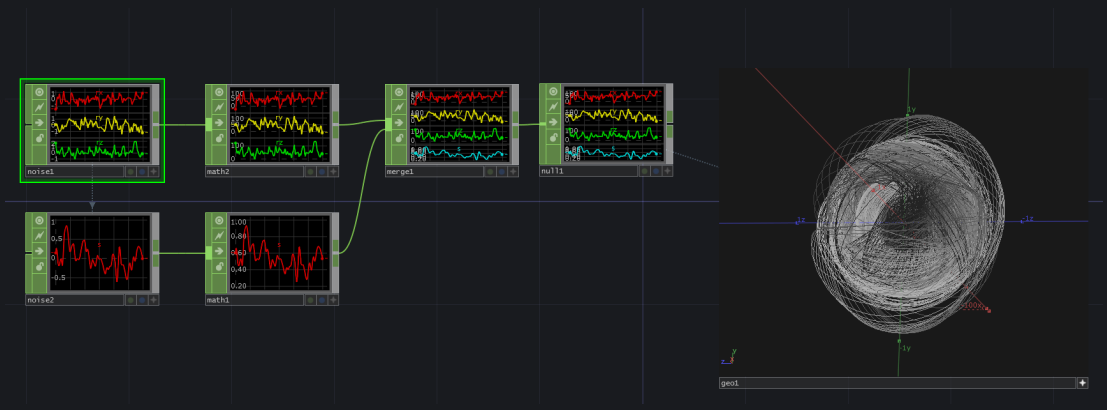

Figure 3.5.: CAPTION MISSING

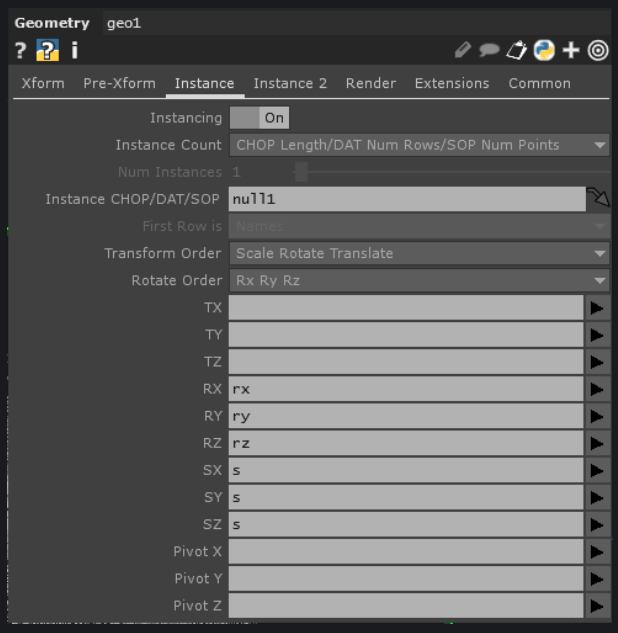

Figure 3.6.: CAPTION MISSING

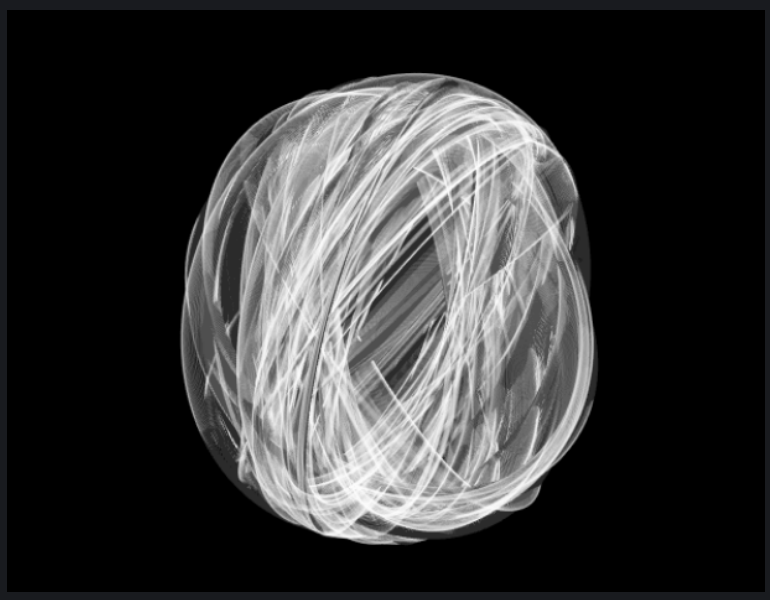

Figure 3.7.: 7200 circles in movement plus some material tweaking

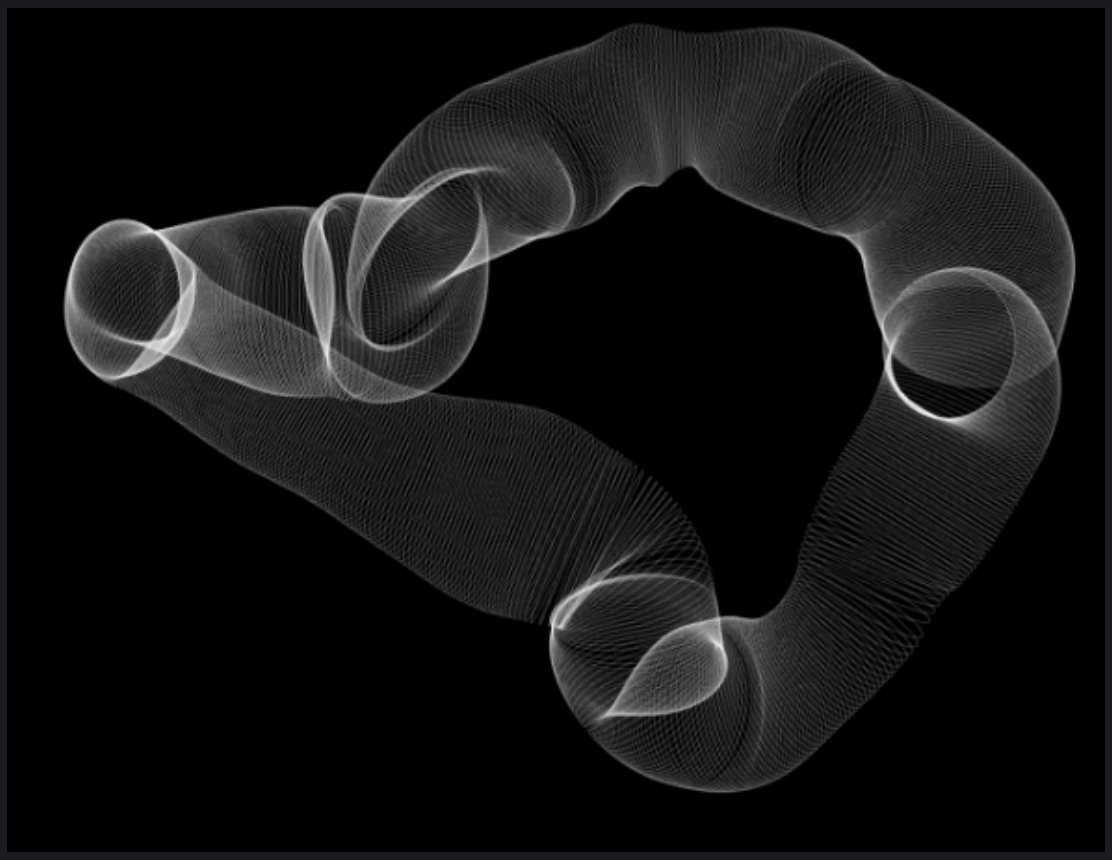

Figure 3.8.: Three dimensional version of the generative design noise circle example

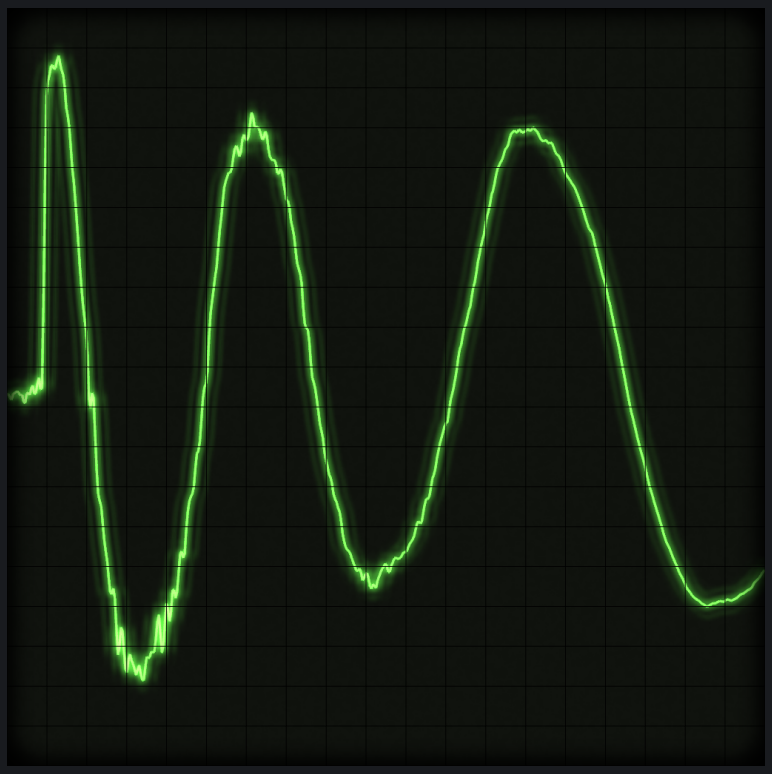

Figure 3.9.: CAPTION MISSING

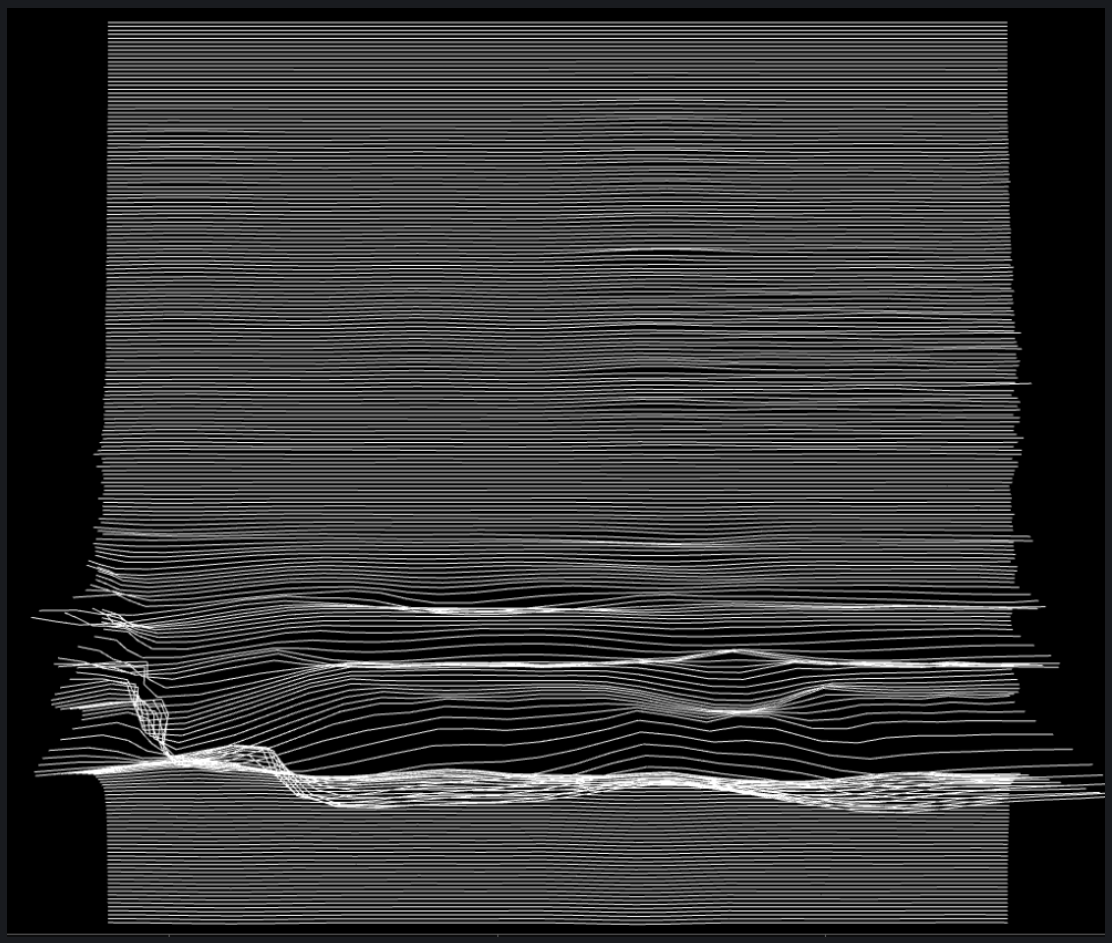

Figure 3.10.: Waterfall plot of an audio spectrum

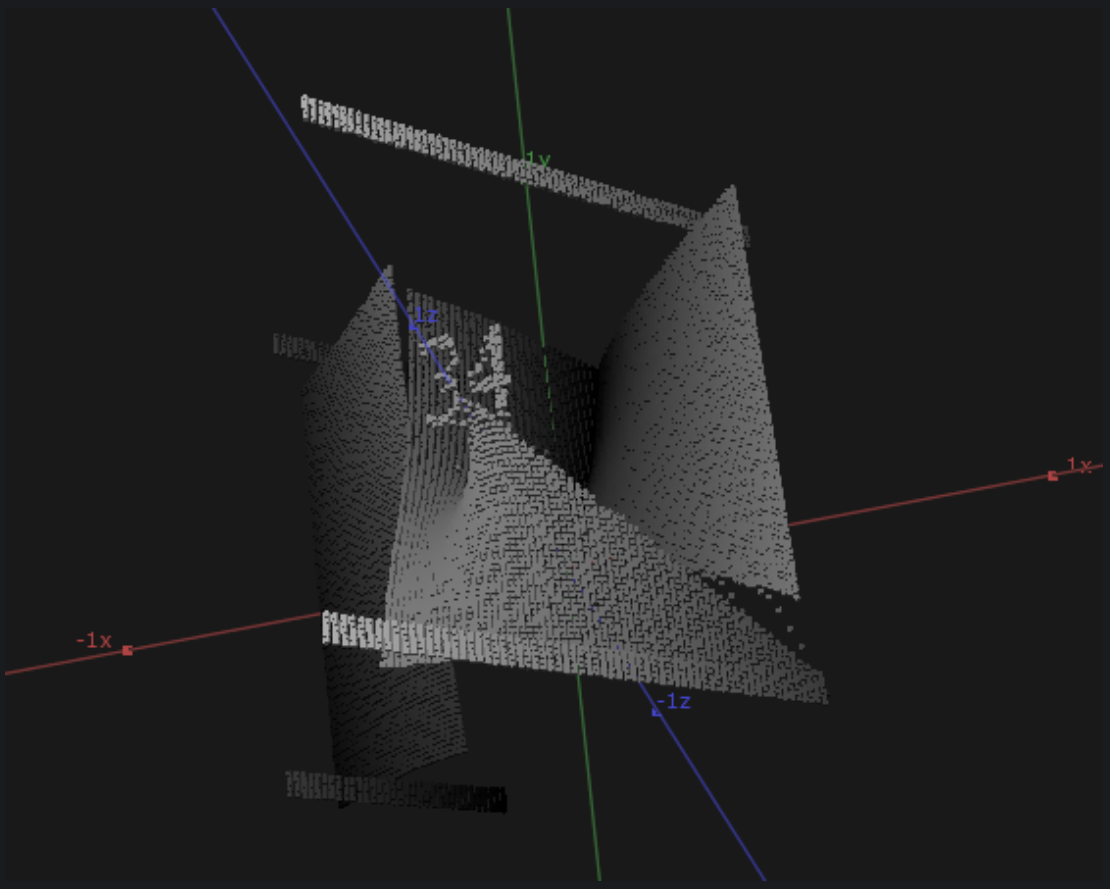

Figure 3.11.: CAPTION MISSING

Figure 3.12.: CAPTION MISSING

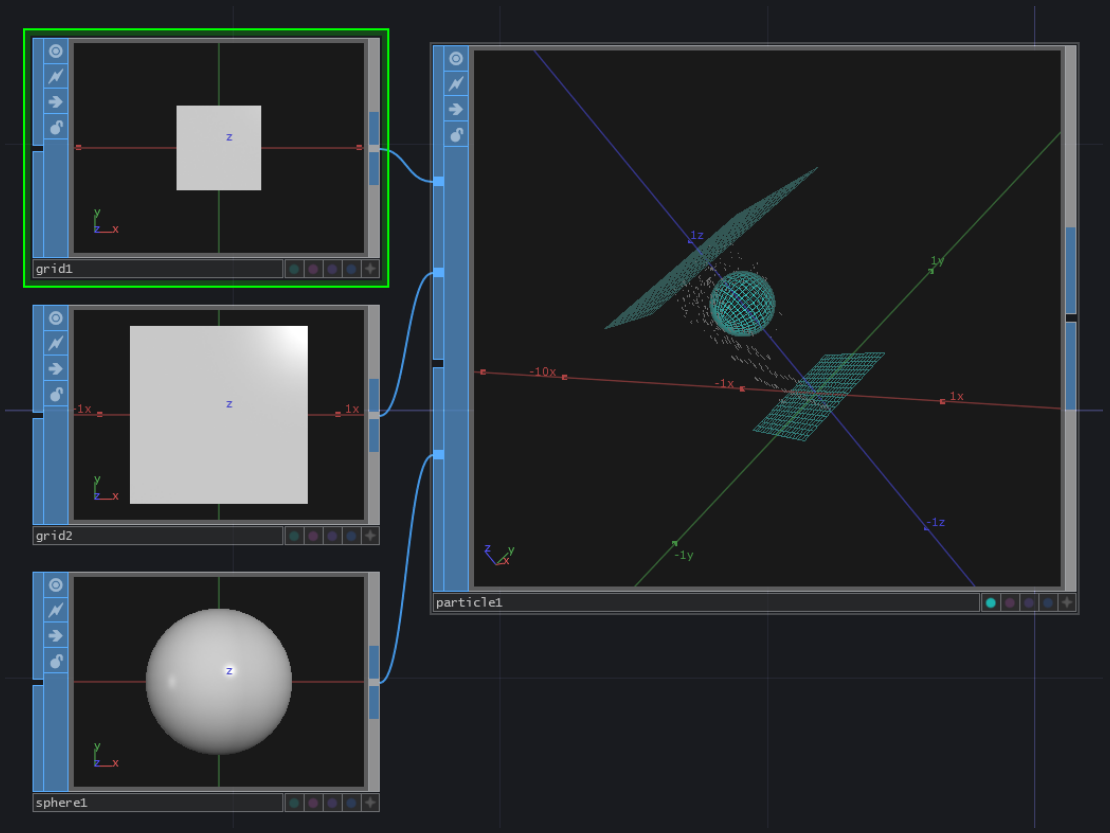

Figure 3.13.: CAPTION MISSING

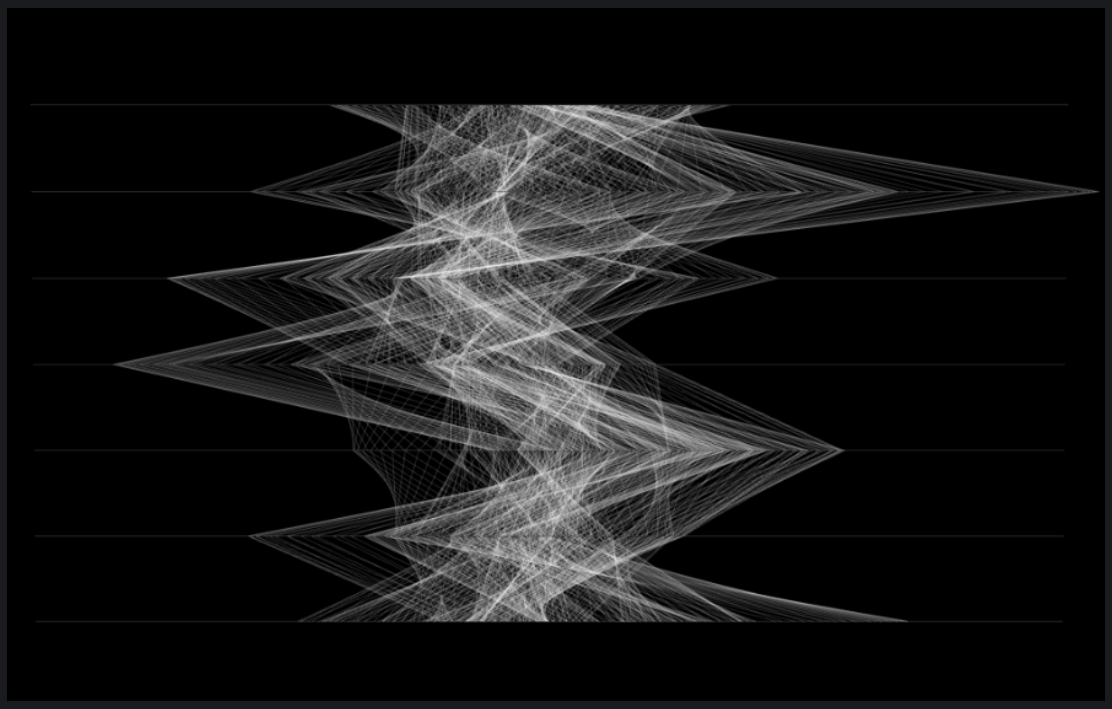

Figure 3.14.: 'parallel Coordinates', a technique for hyperdimensional data visualization

4. Lecture 4

4.1. Notes

- more proc modeling
- parent shortcuts
- custom parameters
- modules
- structured work
- Cloning
- Replicator
- ui building

4.2. Project Setup and Structure

There are a lot of ways to set up a project for live performance. Different projects will require different structures and setups. Here, we will have a look at one suggestion. This suggestion is designed to fulfill the following criteria:

- Being non-Linear and interactive
- Being somewhat tidy
- having multiple scenes
- having multiple FX
- each scene can be combined with each Effect
- Having a GUI
- each scene can react to audio without the need to duplicate the audio analysis
- versatile Output
- Minimize time for on-site configurations and adaptions

The project will be set up in a hierarchical way. In `root` we find a structure that looks like Figure 4.1.

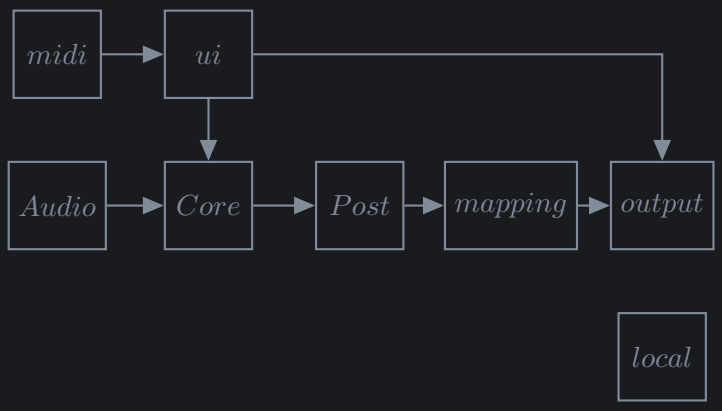

Figure 4.1.: Schematic view of our project

In practice, this might look like in Figure 4.2.

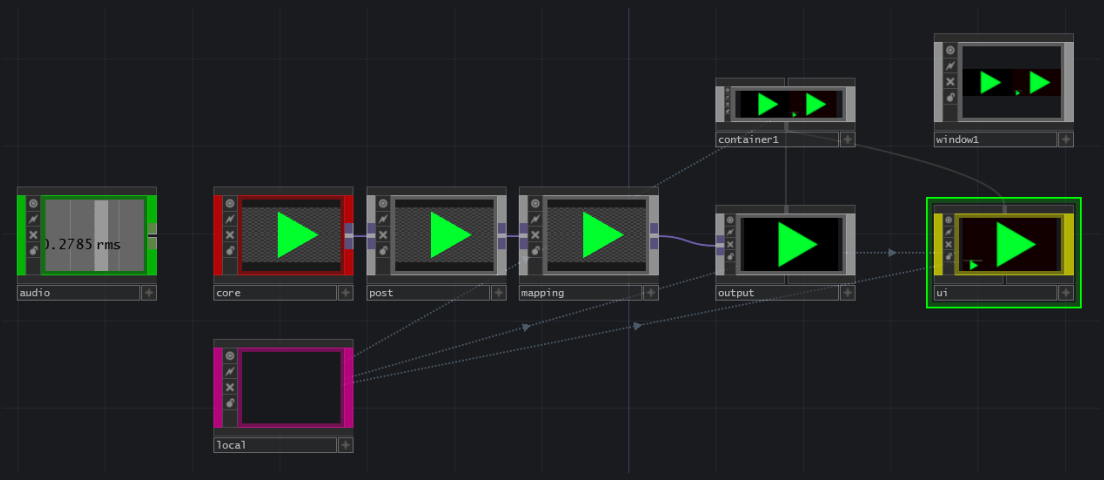

Figure 4.2.: Root level of our Project, inside `/project1/`

Notice that some arrows exist in the schematic but don't have corresponding patch cords in Figure 4.2. This is because the information is flowing via references and/or Select **OPs**.
Let's now have a look at what happens in each of these **COMPs**.

Midi We could also name this 'HID'(Human Interface devices). This could manage OSC[1]. Typically some form of hardware device is used to control elements on the UI which then control the actual parameters.

[1]Open Sound control, a very common protocol for inter device communication typically running on top of the UDP protocol

UI The UI Component is typically a Container **COMP**. It holds the Graphical User Interface we see during our Show. It contains Sliders and other controls, previews of currently unused scenes and shows us our actual output in case we don't see our projection. It should also show us some information about audio input levels (This is not shown in figure 4.1). Also it is used by the output, since we typically create one window spanning over all our monitors.
The UI can instantiate other UIs and can hold stored data.

Audio Here we find our audio input (Audio Device In **CHOP**) and all the audio analysis. The analyzed values (e.g. the current RMS) is then used by scenes in `core`.

Core Here we actually create our different scenes and effects. A very typical setup inside `core` could look like Figure 4.3. This is only one possibly topology of our image generation. We could also create our show using just one versatile algorithm with many parameters and possibly preset store/load facilities or we could go for a more interconnected matrix-like layout.

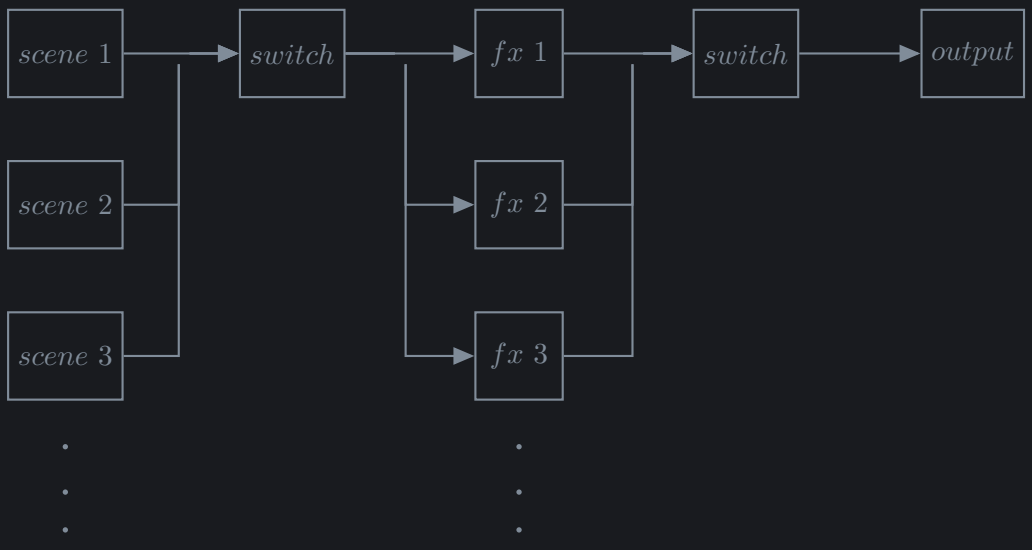

Figure 4.3.: Schematic view of `core`

In reality, this could look like Figure 4.4.

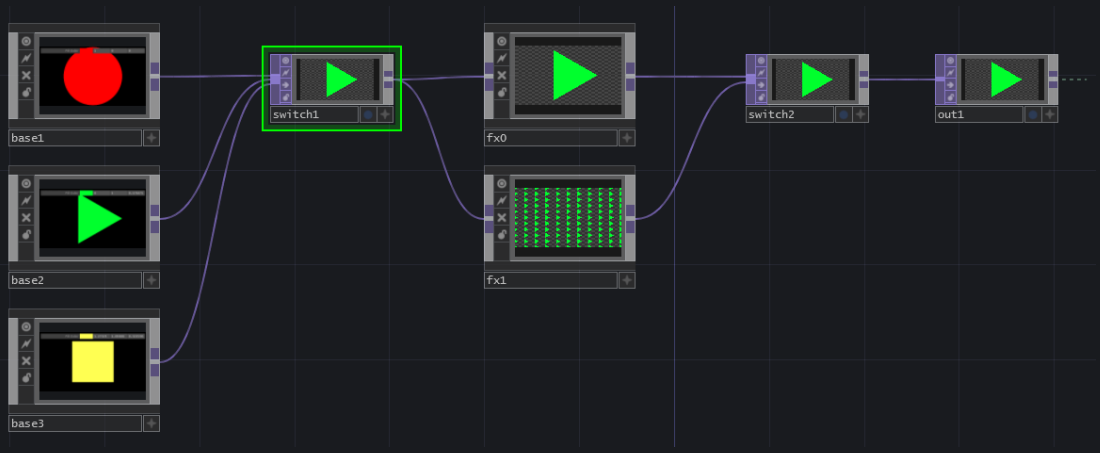

Figure 4.4.: inside `/project1/core`

Post Oftentimes we need some simple post fx such as contrast and color correction feature for adjustment on-site.

Mapping Here we might do our mapping for example using **Cam Schnappr** or **Kantan Mapper**. Both of these tools can be found in TouchDesigner's Palette.

Output Our output can be a Window COMP inside of `project1`. It should contain both our UI and the image we want to send to our projector(s).

Local The `local` COMP is a special COMP. It simply is a Base COMP that we renamed to `local`. This in turn makes TouchDesigner look in there for modules and variables. We will have a look at this at a later point, but to put it simply: It allows us to store states, paths and other variables that make our network more manageable.

4.3. Designing a self-contained Component, our first Scene

Let's have a look at an example scene first:

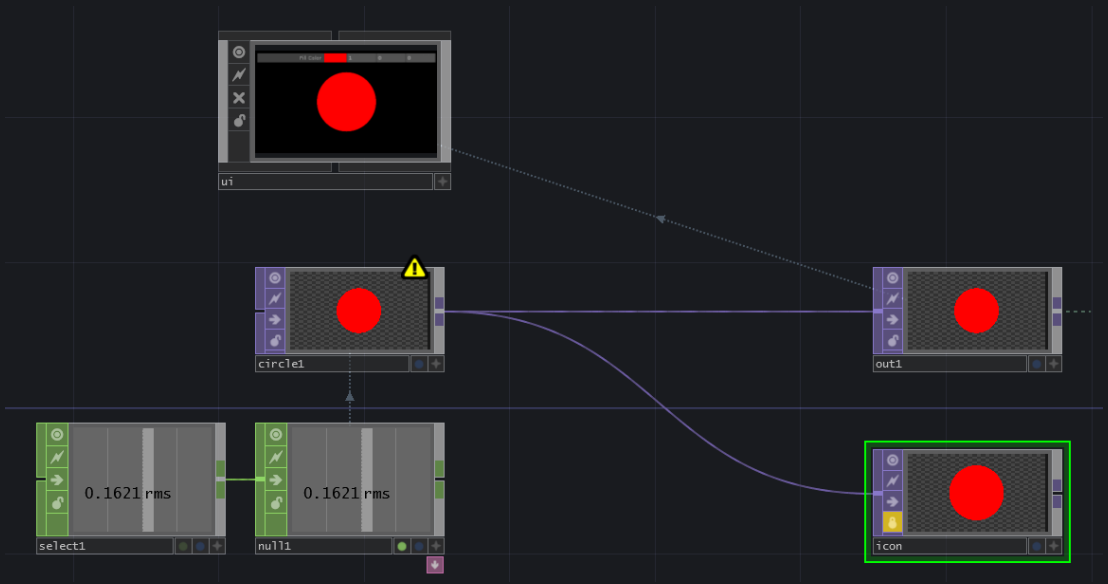

Figure 4.5.: example scene, `/project1/core/scene0`

What we have is some super simplistic imige generator, that is controlled via the RMS of the audio. It's output is sent to the **COMP**'s output. Also its output was cached at some point into a Null **TOP** with the Lock Flag enabled to save the image. The Null **TOP** is renamed to `icon`. This is a convention to later fetch an unanimated preview of what the scene looks like.
Also we have a simple GUI in form of a Component **COMP**.
So let's now see why we chose to layout our exmaple scene this way and what concepts were used here and why.

Ideally our scenes are self contained in a sense that we can drag and drop them in any network, they work instantly and we never have to look inside. How can we achieve this? Basically via

- Custom Parameters[2]
- a decent GUI
- Sticking to certain standards
- enough interconnectivity
- Testing. Make sure there are no errors. Because there are errors.

We will now have a look at how to build a GUI and how to expose parameters. An excellent video about how to build components can be found here[3]

[2] https://docs.derivative.ca/index.php?title=Custom_parameters
[3] https://www.youtube.com/watch?v=3mgn5SJg1oI

4.3.1. Custom Parameters

The Custom Parameters feature allows us to add parameters to a Component. We can just right-click on a Component an choose `Customize Component...` to arrive at the window shown in Figure 4.6.

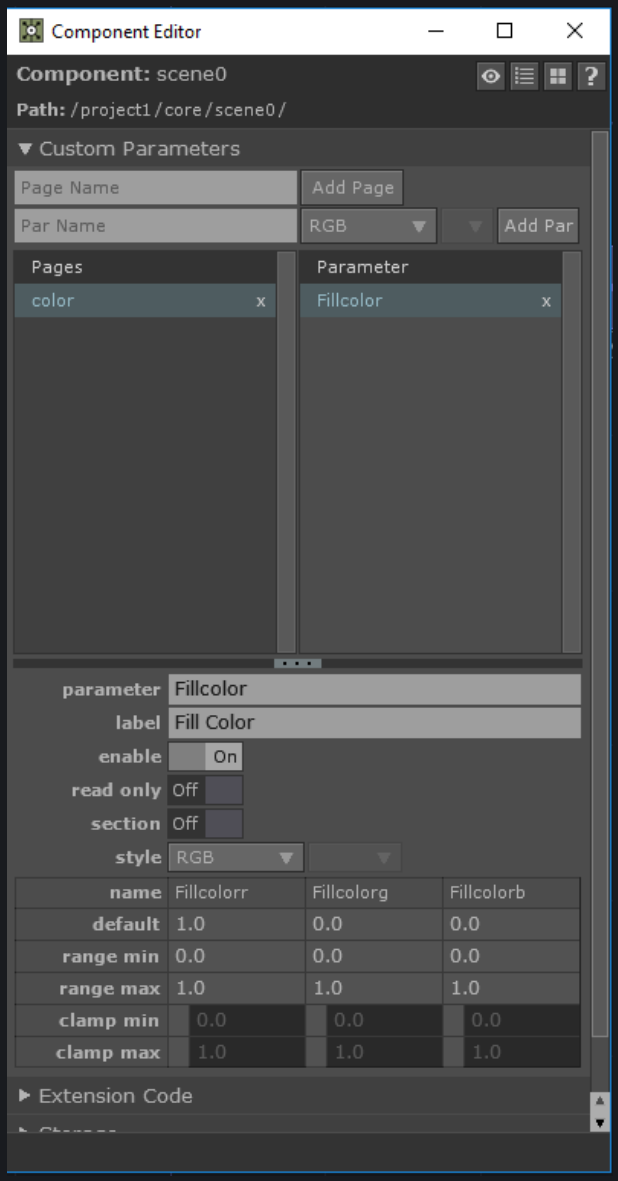

Figure 4.6.: The Component Editor

Here we can add pages and parameters to our component's parameter dialog.
Often we will just need to control a parameter of one of our **OPs** inside the **COMP** directly. We would need to look at the parameter, find the correct type for our custom parameter, and link it to the parameter inside via an expression or CHOP exporting. But there is a nice trick: We can open the Component Editor, go inside our Component

and drag the parameter we want to control to the Parameter list in the Component Editor. This will create a new Parameter with the same name, type and ranges as the original parameter. Also there appears a small pop-up, asking us if we want to automatically create a reference, so the two are linked.

4.3.2. Simple GUI

TouchDesigner is really good for GUI building.

Figure 4.7.: Image taken from here[4]. GUIs built in TouchDesigner for the motion picture Alien Covenant

But before getting to complicated, let's look at how we can get the most simplistic GUI going. For our scenes, it would be great if we had a GUI that showed the output of our scene and the important parameters. GUIs are built using one or multiple Container **COMP**s.
The Container **COMP** allows us to align children elements in an orderly hierarchical manner and it generates events and data for mouse interaction such as mouse rollover or click events.
Another very handy **COMP** is the Parameter **COMP**. We can just point it to a parameter of a chosen OP and it displays it.
So, to make our example scene display a very simple GUI, we do the following:

1. create a Container COMP
2. rename it to 'ui'
3. point its `background TOP` parameter to out Out TOP
4. inside the Container COMP, create a Parameter COMP.
5. adjust the Parameter COMP to point to important parameters.
6. in the `OP Viewer` parameter of our `scene0` Base COMP, write `./ui`, so the scene displays its ui

4.3.3. Connections without Patch Cords, the Select OPs

Figure 4.8.: CAPTION MISSING

4.3.4. Configuring TD after Install

There are not a lot of things that have to be configured after a fresh install of TD. One of the things that are highly recommended to be adjusted is the text editor. Sometimes, we write actual text-code inside TouchDesigner, be it **python**, **GLSL**, **TScript**,

or another language. We can write this code inside TD itself, but we can also use an external text editor, such as Sublime Text or Notepad+, which provides syntax highlighting, auto-completion and more.
In order to conveniently use such an external editor, we just need to go to TD's preferences and specify the path to the external editor.
After that, anytime we right-click on a **DAT** and choose *Edit contents* or press ctrl +e , our text editor opens, we can edit the file, save it and will have edited that **DAT**.

4.3.5. Scripting

important helper functions:

```
dir(object) #get attributes of the object
help(object) #call help on object
```

Referencing OPs

OPs and their contents can be referenced in an absolute or relative way. Also, it makes sense to save paths inside variable, as we will see in 4.3.5. to reference a Null **CHOP** at `/project1/null1` for example, we can use the expression:

```
op('/project1/null1')
```

We can access the first channel in that Null **CHOP** just by issuing `op('/project1/null1')[0]`. Also, a **CHOP**'s channels may be accessed by their name, so if we had a **CHOP** with a channel named `tx` at the same location, we can use `op('/project1/null1')['tx']`. Both of these expressions are useful for animating parameters, and are an alternative to **CHOP exporting**(see 1.5.1).

> What should I use, **CHOP** exporting or python references? Python expressions and references tend to be cleaner and more readable. **CHOP** exporting is faster performance-wise. The team at derivative is constantly working to make python expressions compile to native C code in the backround so simple expresions and references are basically already as fast as **CHOP** exporting.
> So, if in a certain project, performance is a very critical issue, one should still stick to **CHOP** exporting. Otherwise, python expressions are preferable due to readability and overall handyness.

Variables

In TouchDesigner, there are a number of variable types [5]. Within scripts, variables may be defined in a standard pythonic way, such as `a = 5`. For variables that may be accessed throughout a network, **component variables** are the most useful ones, although there are a couple of options to get a variable-like behavior[6], without defining an actual variable.
Variables in TD should not be used for processes that require them to be written very often, such as driving an animation at 60 FPS with a variable. A (time sliced) **CHOP** channel makes more sense in such situations. Think of variables rather for purposes of storing application resolution, setting and getting the state of state machines, or similar „semi-static“ values.

Component variables in TD are bound to a **COMP**. They can be bound to the **root Component**, but may live in any component. There are many ways to get the value of a variable, as well as setting it. In order to create a variable called `varName` with the value `5` in the component `/project1` we can execute the command:

```
op('/project1').setVar('varName', 5)
```

This will create a Base **COMP**: `/project1/local`. The `local` **COMP** is a special **COMP** that is automatically referenced in certain situations, such as retrieving a variable or module. We could have created a Base **COMP** and renamed it to `local` manually too. Inside this **COMP**, our variables are stored via **DAT**s.

missingImage

For defining variables that can be retrieved everywhere in a network, we typically define a component Variable that is attached to the root component (at `/`). In order to do so, we can execute the command:

```
root.setVar('varName', 5)
```

This will store our new variable inside `/local`. We can now get the value of `varName` by evaluating

```
var('varName')
```

in a parameter or by using an Evaluate **DAT** for example, but also simply in the **Text Port**.

[5] Variables on the TouchDesigner Wiki
[6] Other options, that might be more useful depending on the situation are: Referencing a Null **CHOP**, extensions, the storage system, modules containing variables and more

We can override Variables inside specific **COMPs**. Let's assume we made a complex network, and we use two root Component variables to set the resolution for all **TOPs** in the network. This is good practice, since it makes it easy to change the resolution quickly and globally. But maybe we have some subprocess that is also quite complex and which needs a different resolution. We could make new variables. But what if we could just override the global ones inside just this subprocess? Well, we can, we just need to redefine the variables inside that specific **COMP**.

Assume our complex subprocess is a base **COMP** at `/project1/sub`. We can just define:

```
root.setVar('resXY', [1280, 720])
op('/project1/sub').setVar('resXY', [256, 256])
```

This way, everywhere in our network, `resXY` will evaluate to the list `[1280, 720]`, but everywhere inside `/project1/sub` it will evaluate to `[256, 256]`.

Modules

Modules are a way to organize variables, functions, classes. Any Text **DAT** can hold a module. let's assume we made a Text **DAT** at `/project1/myTools`. In there, we wrote:

```
x = 5
def multiplyByTwo(inp):
        return inp*2.
```

testThis

We can now use this Text **DAT** as a module. In order to do so, we have couple of options. One would be to use it in another script inside another Text **DAT**. For example:

```
import myTools
print (myTools.multiplyByTwo(5))
```

Another and maybe more common way would be to use it in a parameter, by the MOD (module on demand) class. If our module is inside the search Path (so in the same **COMP** for example, or, as we see in a moment, inside `local`) we can use the expression:

```
mod.myTools.multiplyByTwo(5)
```

For information on how to reference a module, and for information about how to import

it as efficiently as possible, look here[7]

[7] http://www.derivative.ca/wiki088/index.php?title=MOD_Class

5. Lecture 5

Notes

Contents

Pitch Visualization (Bach Piece)

Communication with Max/MSP

Projection Mapping

GLSL

Procedural Modeling

https://www.shadertoy.com/view/MdfBRX

- group SOP
- latice SOP
- magnet/force/metaball SOP
- Texture SOP
- Model SOP

5.1. Projection Mapping

Projection Mapping[1]

5.1.1. 2D Mapping

There are a couple of **TOPs** that enable us to do simple mapping tasks:

- Crop **TOP**[2]
- Corner_Pin **TOP**[3]

[1] https://docs.derivative.ca/index.php?title=Projection_mapping
[2] http://derivative.ca/wiki099/index.php?title=Crop_TOP
[3] http://derivative.ca/wiki099/index.php?title=Corner_Pin_TOP

- Transform TOP[4]
- Displace TOP[5]

So if we jsut need some slight adjustment of our output we can come up with our own mapping tool. This keeps our network simple and is a good choice if we need nothing fancy.
For more complex 2D mapping scenarios TouchDesiger comes with some pretty advanced tools that helps us to properly map one or multiple textures to the physical environment.

Stoner

The Stoner[6] COMP can be found in the Palette.

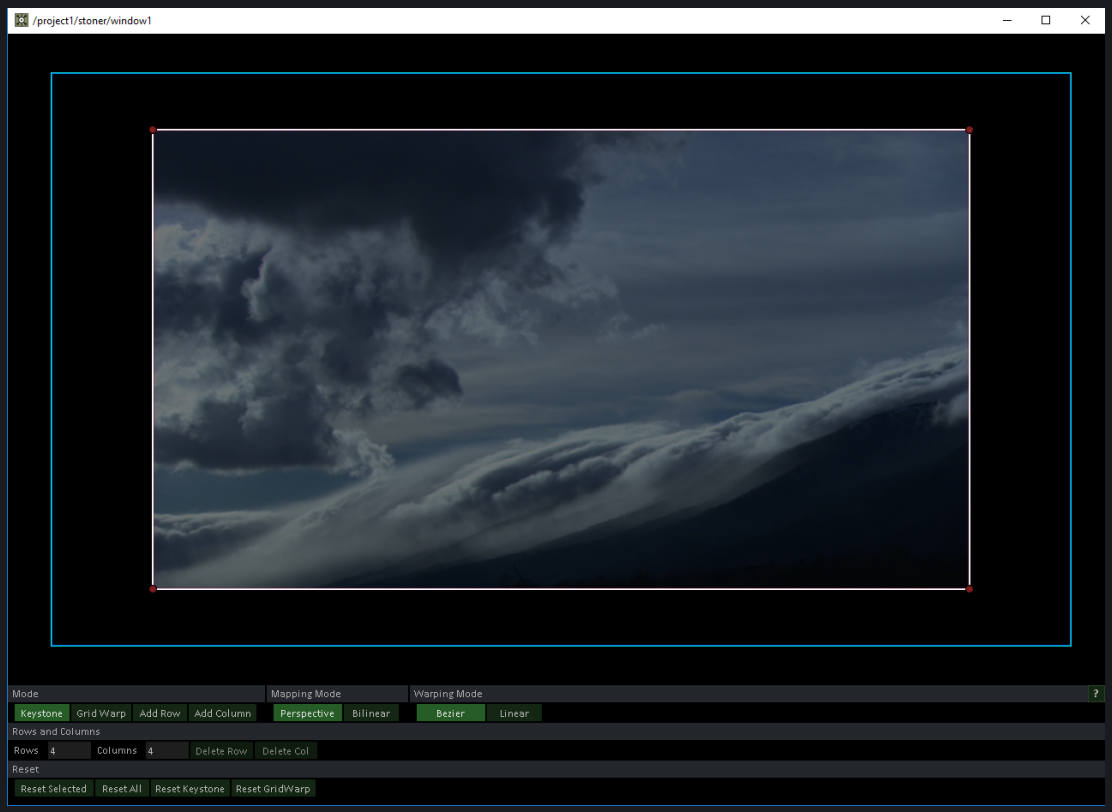

Figure 5.1.: CAPTION MISSING

Kantan Mapper

The Kantan Mapper[7] COMP can be found in the Palette.

[4] http://derivative.ca/wiki099/index.php?title=Transform_TOP
[5] http://derivative.ca/wiki099/index.php?title=Displace_TOP
[6] https://docs.derivative.ca/index.php?title=Palette:Stoner
[7] https://docs.derivative.ca/index.php?title=Palette:kantanMapper

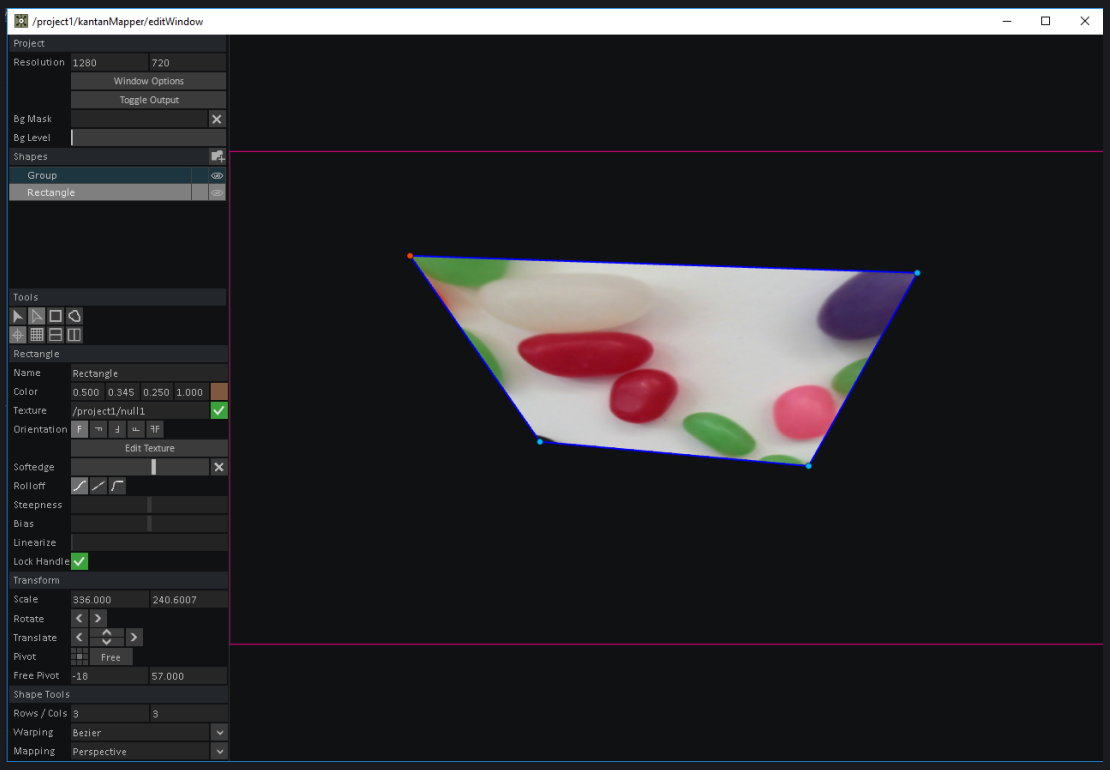

Figure 5.2.: CAPTION MISSING

External Tools

Also we can use other tools than TouchDesigner for mapping. We can send our video material to other software packages via the DirectX_Out TOP[8] or the Syphon_Spout_Out TOP[9]. Popular mapping tools are for example:

- Madmapper[10]
- Visution MAPIO2[11]
- Resolume Arena[12]
- Isadora[13]

5.1.2. 3D Mapping

For 3D mapping, we need a 3D model of the physical structure we want to be mapped. Obviously, two scenartios could come about here:

[8] http://derivative.ca/wiki099/index.php?title=DirectX_Out_TOP
[9] http://derivative.ca/wiki099/index.php?title=Syphon_Spout_Out_TOP
[10] https://madmapper.com/
[11] https://visution.com/
[12] https://resolume.com/software
[13] https://troikatronix.com/

1. Either we start with a 3d Model and Build it in reality or
2. we start from a physical Object and create a 3D model of it.

For scenario 1, we could start off in a CAD program like Solid Works or Auto CAD, design a structure and build it/3D-print it. In Scenario two we are confronted with a physical object and need to get an accurate 3D model of it. Sometimes we can obtain construction plans and accurately rebuild the object this way. In other cases we need to somehow measure the object, since there are no plans. This can be done using a laser or using photogrammetry. Autodesk provides software that does this(Recap[14]), but there are many programs that try to extract a 3D model from a series of photos or a video like VisualSFM[15].

When we have a 3D Model of the structure we want to map we can use CamSchnappr in TouchDesigner to automatically estimate the objects position, the projectors position and the lens parameters of the projector to automatically align our projection with the physical world. The CamSchnappr[16] **COMP** can be found in the Palette.

5.2. GLSL

GLSL is a C-style language. It enables us to write code that runs on the GPU. Since the GPU has a massively parallelized architecture, some processes can be faster to calculate on it by a huge margin.
In TouchDesigner we have a couple of ways to write such a program wich is typically called a *Shader*[17]. We shouldn't forget though, that all **TOPs** *are* shaders and calculated on the GPU and **MATs** are as well.
But if we want to write GLSL code we can do so by using the following **OPs**:

- GLSL **TOP**[18]
- GLSL_Multi **TOP**[19]
- GLSL **MAT**

Also, the GLSL Wiki page[20] is a good starting place to get oriented how to write shaders in TD.

[14] https://www.autodesk.com/products/recap/overview
[15] http://ccwu.me/vsfm/
[16] https://docs.derivative.ca/index.php?title=Palette:CamSchnappr
[17] If you want a more exact definition of the word shader, go here
[18] http://derivative.ca/wiki099/index.php?title=GLSL_TOP
[19] http://derivative.ca/wiki099/index.php?title=GLSL_Multi_TOP
[20] https://docs.derivative.ca/index.php?title=GLSL

Learning how to code in GLSL is out of the scope of this course. But we will go ahead and use some shader code we find online and slightly change it so it runs inside TouchDesigner. A highly recommended resource for learning how to write GLSL is The Book of Shaders[21]. Another interesting resource is Shadertoy[22]. Here we can find tons of shaders and look at them online. We can then just grab the code and port it to TouchDesigner, which is what we are going to do next.

I more or less randomly picked a shader from Shadertoy, this one[23]. Its output looks like this:

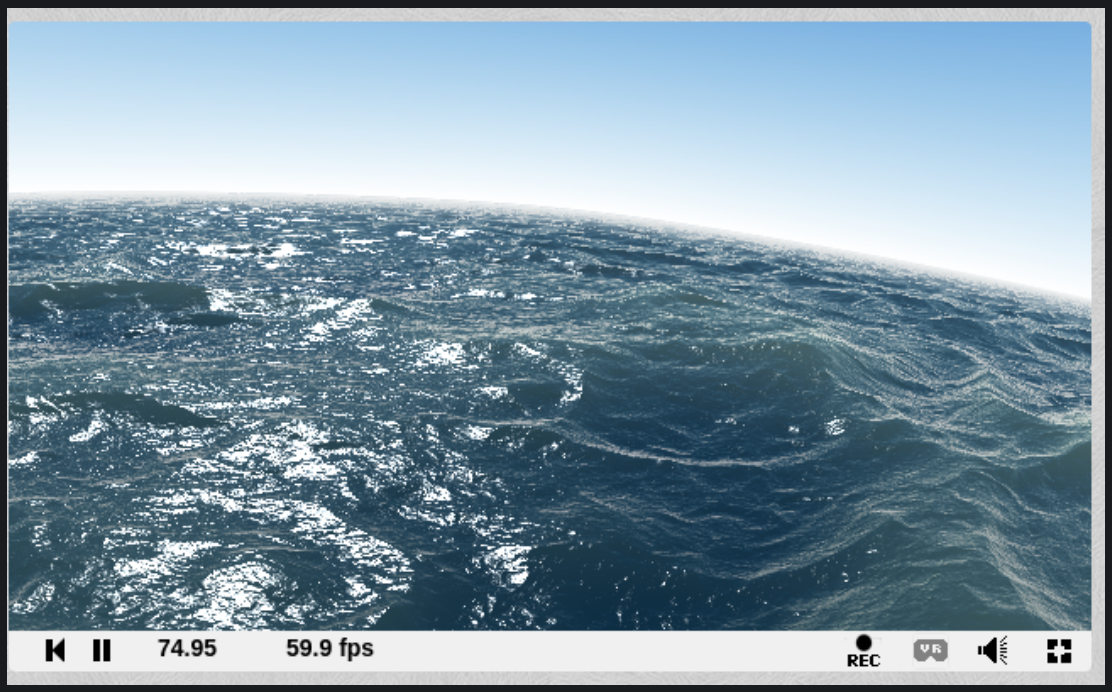

Figure 5.3.: CAPTION MISSING

And its code looks like this (don't panic now):

```
1  /*
2   * "Seascape" by Alexander Alekseev aka TDM - 2014
3   * License Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported
       License.
4   * Contact: tdmaav@gmail.com
5   */
6
7  const int NUM_STEPS = 8;
```

[21] https://thebookofshaders.com/
[22] https://www.shadertoy.com/
[23] https://www.shadertoy.com/view/Ms2SD1

```
8  const float PI        = 3.141592;
9  const float EPSILON   = 1e-3;
10 #define EPSILON_NRM (0.1 / iResolution.x)
11
12 // sea
13 const int ITER_GEOMETRY = 3;
14 const int ITER_FRAGMENT = 5;
15 const float SEA_HEIGHT = 0.6;
16 const float SEA_CHOPPY = 4.0;
17 const float SEA_SPEED = 0.8;
18 const float SEA_FREQ = 0.16;
19 const vec3 SEA_BASE = vec3(0.1,0.19,0.22);
20 const vec3 SEA_WATER_COLOR = vec3(0.8,0.9,0.6);
21 #define SEA_TIME (1.0 + iTime * SEA_SPEED)
22 const mat2 octave_m = mat2(1.6,1.2,-1.2,1.6);
23
24 // math
25 mat3 fromEuler(vec3 ang) {
26        vec2 a1 = vec2(sin(ang.x),cos(ang.x));
27     vec2 a2 = vec2(sin(ang.y),cos(ang.y));
28     vec2 a3 = vec2(sin(ang.z),cos(ang.z));
29     mat3 m;
30     m[0] = vec3(a1.y*a3.y+a1.x*a2.x*a3.x,a1.y*a2.x*a3.x+a3.y*a1.x,-a2.y*a3.x);
31        m[1] = vec3(-a2.y*a1.x,a1.y*a2.y,a2.x);
32        m[2] =
              vec3(a3.y*a1.x*a2.x+a1.y*a3.x,a1.x*a3.x-a1.y*a3.y*a2.x,a2.y*a3.y);
33        return m;
34 }
35 float hash( vec2 p ) {
36        float h = dot(p,vec2(127.1,311.7));
37     return fract(sin(h)*43758.5453123);
38 }
39 float noise( in vec2 p ) {
40     vec2 i = floor( p );
41     vec2 f = fract( p );
42        vec2 u = f*f*(3.0-2.0*f);
43     return -1.0+2.0*mix( mix( hash( i + vec2(0.0,0.0) ),
44                       hash( i + vec2(1.0,0.0) ), u.x),
45                  mix( hash( i + vec2(0.0,1.0) ),
46                       hash( i + vec2(1.0,1.0) ), u.x), u.y);
47 }
48
49 // lighting
50 float diffuse(vec3 n,vec3 l,float p) {
51     return pow(dot(n,l) * 0.4 + 0.6,p);
```

```
52 }
53 float specular(vec3 n,vec3 l,vec3 e,float s) {
54     float nrm = (s + 8.0) / (PI * 8.0);
55     return pow(max(dot(reflect(e,n),l),0.0),s) * nrm;
56 }
57
58 // sky
59 vec3 getSkyColor(vec3 e) {
60     e.y = max(e.y,0.0);
61     return vec3(pow(1.0-e.y,2.0), 1.0-e.y, 0.6+(1.0-e.y)*0.4);
62 }
63
64 // sea
65 float sea_octave(vec2 uv, float choppy) {
66     uv += noise(uv);
67     vec2 wv = 1.0-abs(sin(uv));
68     vec2 swv = abs(cos(uv));
69     wv = mix(wv,swv,wv);
70     return pow(1.0-pow(wv.x * wv.y,0.65),choppy);
71 }
72
73 float map(vec3 p) {
74     float freq = SEA_FREQ;
75     float amp = SEA_HEIGHT;
76     float choppy = SEA_CHOPPY;
77     vec2 uv = p.xz; uv.x *= 0.75;
78
79     float d, h = 0.0;
80     for(int i = 0; i < ITER_GEOMETRY; i++) {
81         d = sea_octave((uv+SEA_TIME)*freq,choppy);
82         d += sea_octave((uv-SEA_TIME)*freq,choppy);
83         h += d * amp;
84         uv *= octave_m; freq *= 1.9; amp *= 0.22;
85         choppy = mix(choppy,1.0,0.2);
86     }
87     return p.y - h;
88 }
89
90 float map_detailed(vec3 p) {
91     float freq = SEA_FREQ;
92     float amp = SEA_HEIGHT;
93     float choppy = SEA_CHOPPY;
94     vec2 uv = p.xz; uv.x *= 0.75;
95
96     float d, h = 0.0;
```

```
97      for(int i = 0; i < ITER_FRAGMENT; i++) {
98          d = sea_octave((uv+SEA_TIME)*freq,choppy);
99          d += sea_octave((uv-SEA_TIME)*freq,choppy);
100         h += d * amp;
101         uv *= octave_m; freq *= 1.9; amp *= 0.22;
102         choppy = mix(choppy,1.0,0.2);
103     }
104     return p.y - h;
105 }
106
107 vec3 getSeaColor(vec3 p, vec3 n, vec3 l, vec3 eye, vec3 dist) {
108     float fresnel = clamp(1.0 - dot(n,-eye), 0.0, 1.0);
109     fresnel = pow(fresnel,3.0) * 0.65;
110
111     vec3 reflected = getSkyColor(reflect(eye,n));
112     vec3 refracted = SEA_BASE + diffuse(n,l,80.0) * SEA_WATER_COLOR * 0.12;
113
114     vec3 color = mix(refracted,reflected,fresnel);
115
116     float atten = max(1.0 - dot(dist,dist) * 0.001, 0.0);
117     color += SEA_WATER_COLOR * (p.y - SEA_HEIGHT) * 0.18 * atten;
118
119     color += vec3(specular(n,l,eye,60.0));
120
121     return color;
122 }
123
124 // tracing
125 vec3 getNormal(vec3 p, float eps) {
126     vec3 n;
127     n.y = map_detailed(p);
128     n.x = map_detailed(vec3(p.x+eps,p.y,p.z)) - n.y;
129     n.z = map_detailed(vec3(p.x,p.y,p.z+eps)) - n.y;
130     n.y = eps;
131     return normalize(n);
132 }
133
134 float heightMapTracing(vec3 ori, vec3 dir, out vec3 p) {
135     float tm = 0.0;
136     float tx = 1000.0;
137     float hx = map(ori + dir * tx);
138     if(hx > 0.0) return tx;
139     float hm = map(ori + dir * tm);
140     float tmid = 0.0;
141     for(int i = 0; i < NUM_STEPS; i++) {
```

```
142         tmid = mix(tm,tx, hm/(hm-hx));
143         p = ori + dir * tmid;
144         float hmid = map(p);
145                 if(hmid < 0.0) {
146                 tx = tmid;
147             hx = hmid;
148         } else {
149             tm = tmid;
150             hm = hmid;
151         }
152     }
153     return tmid;
154 }
155 
156 // main
157 void mainImage( out vec4 fragColor, in vec2 fragCoord ) {
158         vec2 uv = fragCoord.xy / iResolution.xy;
159     uv = uv * 2.0 - 1.0;
160     uv.x *= iResolution.x / iResolution.y;
161     float time = iTime * 0.3 + iMouse.x*0.01;
162 
163     // ray
164     vec3 ang = vec3(sin(time*3.0)*0.1,sin(time)*0.2+0.3,time);
165     vec3 ori = vec3(0.0,3.5,time*5.0);
166     vec3 dir = normalize(vec3(uv.xy,-2.0)); dir.z += length(uv) * 0.15;
167     dir = normalize(dir) * fromEuler(ang);
168 
169     // tracing
170     vec3 p;
171     heightMapTracing(ori,dir,p);
172     vec3 dist = p - ori;
173     vec3 n = getNormal(p, dot(dist,dist) * EPSILON_NRM);
174     vec3 light = normalize(vec3(0.0,1.0,0.8));
175 
176     // color
177     vec3 color = mix(
178         getSkyColor(dir),
179         getSeaColor(p,n,light,dir,dist),
180         pow(smoothstep(0.0,-0.05,dir.y),0.3));
181 
182     // post
183         fragColor = vec4(pow(color,vec3(0.75)), 1.0);
184 }
```

So, we can put down a GLSL **TOP** in TouchDesigner and can just copy paste the code

into the Text **DAT** called `glsl1_pixel` that should be attached to the GLSL **TOP**. Just copy the contents of the file above what's in there already. When we did that, we see a couple of errors.

6. Lecture 6

6.1. Notes

A. Shortcut Cheat Sheet

shortcut	action
h	home view
d	toggle display flag
r	toggle render flag
shift + a	toggle all viewers Active
u	go up in component Hierarchy
i	go down in component Hierarchy
p	toggle Parameter Dialog Visibility
q	toggle adaptive homing in Geometry Viewports
Ctrl + e	Edit Dat in external Editor

Index