Machine Translated by Google

Hydra Functions

Machine Translated by Google

Machine Translated by Google

Machine Translated by Google

Readme!

This PDF is the fruit of two "intelligences" mine, very modest, and that of Artificial Intelligence. Indeed, my project to compile the functions available in Hydra for which
I often looked for additional explanation (I was going to say inspiration) as the dedicated page on the Hydra website hydra functions is, in my humble opinion, too concise and somewhat stingy with comments (perhaps examples too!).

So I asked for the precious help of AI (ChatGPT) in this case in order to satisfy my natural curiosity about this extraordinary program that is hydra video synth and in particular about the functions that we use to bring all these visual creations to life!

This document is far from perfect and is open to any improvements that its readers may suggest: not having fully mastered this text editor, some presentations are not ideally optimized, which you will see for yourself, but it has (as they say) the merit of existing!

Sometimes I had problems with the layout in Pages and couldn't find a solution. solution. I admit that I don't have a very good command of these text editors!
Some examples given by chatGPT are not correct and need to be corrected (which I did of course) and improved. Feel free to become a contributor by improving this document which will, I hope, be of use to the community!

Thanks for reading and maybe contributing too! :-))
Beryann parker, live coder and member of Toplap Strasbourg/France https://linktr.ee/beryann.parker

https://linktr.ee/beryann.parker

Hydra video synth:
https://hydra.ojack.xyz/

The hydra book:
https://hydra-book.glitch.me/#/

Machine Translated by Google

Machine Translated by Google

SOURCES

NOISE

In Hydra, the `noise` keyword is a powerful generative function that creates visual noise patterns often used to add texture or generate abstract shapes in visual compositions.

Here's an overview of the main aspects and settings of `noise` in Hydra:

1. **Random Pattern Generation**: The `noise` function generates random visual patterns
based on noise algorithms (e.g. Perlin noise or simple noise). This type of noise creates organic gradients and patterns, often with soft, flowing shapes
contrasting with geometric or regular patterns.

2. **`noise` parameters**:

   - **scale**: This parameter controls the density or resolution of the noise. For example, a high `scale` value will produce fine, detailed noise, while a low value will produce fine, detailed patterns. wider and coarser noise.

   - **offset**: Controls the displacement or position of the noise pattern, which can be useful for animating noise effects by gradually changing the offset over time.

3. **Typical uses of `noise` in Hydra**:

   - **Textures and Backgrounds**: Noise is often used as a background texture to add an organic dimension to a scene.

   - **Color modulation**: `noise` can be applied to the colors of other objects or visuals to give them a subtle and dynamic variation.

   - **Motion Effects**: By animating the `noise` parameters, it is possible to create smooth and natural motion effects, similar to waves, clouds or even sparkles.
of energy.

### Example of using `noise` in Hydra

Machine Translated by Google

Here is a simple example showing how to use `noise` to generate an animated textured background:

```
noise(3)                // The parameter determines the scale of the noise
  .color(0.5, 0.3, 0.9) // Applies a color to the pattern
  .modulate(noise(1), 0.2) // Modulation for more complexity
  .out()               // Send the animation result to the visual output
```

In this example, `noise(3)` defines a noise with a medium scale. Applying `.color` e
`.modulate`, additional variations can be introduced, and `.out()` displays the result

### Notice

Using `noise` in Hydra can be extremely effective in creating animations
abstract and organic. The successive modulations allow to simulate biological-like textures, which makes it a valuable tool for those who explore more expressive and fluid visuals in their creations.

VORONOI

In Hydra, the `voronoi()` function generates a visual effect inspired by Voronoi diagrams. Voronoi diagrams divide space into regions around central points, creating a mosaic effect where each region represents the area of influence of a particular point. This type of
diagram is commonly used in mathematics, natural sciences, and data visualization to model spatial structures, but Hydra leverages its rendering to produce interesting visual textures and patterns.

### Using `voronoi()` in Hydra

In Hydra, the `voronoi()` creates geometric textures with organic shapes, often used for background effects or as layers in more complex visual compositions. Here are the main parameters and their effect:

Machine Translated by Google

1. **Number of cells (first parameter)**: The first argument to `voronoi()` controls the number of cells in the diagram. The higher the number, the more dense and complex the diagram, while a lower number produces large, distinct cells.

2. **Speed (second parameter)**: This parameter determines how quickly cells move or transform over time, creating a dynamic animation effect.

3. **Edge Linearity (Third Parameter)**: The third parameter controls the level of contrast of the edges between cells. A higher value will make the edges sharper and more defined.
giving a more “chiseled” effect to the diagram.

### Example of use

```javascript

voronoi(10, 0.3, 0.5).out()
```

In this example:

- `10` defines a diagram with moderate cell density

- `0.3` gives a slow animation speed.

- `0.5` slightly softens the edges of cells, adding a subtle visual effect.

### Creative Applications

The `voronoi()` effect can be combined with other functions in Hydra to create abstract and textured visuals. For example, it is common to use it in overlay with effects like `osc()`, `noise()`, or `rotate()` to generate visual compositions with more depth and complexity.

### Example

```
voronoi(25, 0.1, 1.5)

  .modulate(osc(10, 0.5, 1))
```

Machine Translated by Google

```
.out()
```

Here the Voronoi diagram is modulated by an oscillation, which creates an interaction between the cells and adds an interesting distortion effect.

### My opinion

`voronoi()` in Hydra is a great tool for experimenting with mosaic-like visuals.
Its effect is particularly relevant for abstract and generative creations, especially when combined with other Hydra features. For real-time visual artists, it offers flexibility to create dynamic patterns, adaptable to live music or performances.
visual.

OSC

In Hydra, the `osc()` function generates an oscillating sine wave as a visual texture, often used to create patterns of regular stripes, ripples, and psychedelic effects. This function is fundamental to generative visuals because it produces fluidly moving patterns, ideal for real-time or synchronized creations with video.
music.

### Operation and Parameters of `osc()`

The `osc()` function typically takes three main parameters that influence the pattern of
the oscillation:

1. **Frequency (first parameter)**: Controls the number of oscillation bands displayed on the screen. A higher frequency increases the number of bands per unit space, creating a denser effect, while a lower frequency produces wider bands.

2. **Speed (second parameter)**: Controls the speed of the strips scrolling. This speed creates an effect of movement of the bands, often in a horizontal or vertical direction, and helps to give dynamism to the composition.

Machine Translated by Google

3. **Saturation Amplitude or Intensity (third parameter)**: Adjusts the intensity of the colors, influencing the saturation and contrast of the oscillation bands. With a high value, the bands are more saturated and visible, while a lower value can give smoother, almost transparent oscillations.

### Basic Example

```
osc(10, 0.1, 0.9).out()
```

In this example:

- `10` sets a frequency of 10 bands on the screen

- `0.1` indicates a slow scrolling speed, almost f

- `0.9` gives a strong saturation intensity, which makes colors vivid.

### Applications of `osc()` in Visual Compositions

The `osc()` function is versatile and can be used alone or combined with other functions to achieve various visual effects:

1. **Modulation**: By combining `osc()` with another function (like `voronoi()` or `noise()`), you can use oscillation as a modulation layer that alters the visual properties of the base function, adding undulating motion or dynamic texture.

Example :

```
osc(20, 0.3, 1.2)
  .modulate(noise(3))
  .out()
```

Here the oscillation is modified by noise, giving a band disturbance effect.

Machine Translated by Google

2. **Rotation and Depth Layer**: Combining `osc()` with `rotate()` creates spiral rotation effects on the strips, adding a sense of depth and three-dimensional movement.

Example :

```
osc(15, 0.2, 1)
  .rotate(Math.PI / 4)
  .out()
```

In this example, the oscillating bands are tilted, creating an interesting visual dynamic.

3. **Psychedelic Effects and Distortions**: For more intense visual effects, you can adjust the frequency and saturation parameters and combine them with functions like `modulatePixelate()` or `scrollX()`.

Example :

```
osc(50, 0.5, 2)
  .modulatePixelate(osc(10), 100)
  .scrollX(0.1)
  .out()
```

This example produces a complex psychedelic effect, with pixelated oscillating bands and a horizontal scroll

### My Opinion

Machine Translated by Google

The `osc()` function is essential for any artist working with Hydra, as it allows you to create rhythmic and fluid visuals, ideal for immersive environments or performances.
visuals in real time. Its creative potential is virtually endless, especially when exploring
combinations and variations with other functions. By manipulating parameters and layering, it is possible to create visuals that are constantly changing, react to the music, and captivate the audience's attention.

In Hydra, the `shape` function generates a simple geometric shape, often a polygon, that can be used as a basic visual element in an audiovisual composition. It is particularly flexible and allows you to create various visual effects by adjusting its parameters

SHAPE

### Syntax and Parameters

The `shape` function accepts several parameters, each of which affects the generated shape:

```
shape(sides, radius, smoothing)
```

1. **`sides`**: This parameter determines the number of sides of the shape. For example, a value of 3 creates a triangle, 4 creates a square, and so on. Higher values result in polygons with more sides, or even shapes that tend toward a circle at very high values.

2. **`radius`**: This parameter controls the relative size of the shape. A value of 0.5 results in a shape that takes up about half of the screen, while a value of 1 allows the shape to cover the entire screen. Values greater than 1 may cause the shape to extend off the screen.

3. **`smoothing`**: This parameter adjusts the smoothing of the edges of the shape. A higher value produces softer edges, while a lower or zero value produces sharper edges. The

Machine Translated by Google

Smoothing can be especially useful when you're looking to create more aesthetically pleasing shapes, especially for circles or multi-sided polygons.

### Usage Examples

1. **Basic shape (square):**

```
 shape(4, 0.3, 0.01)
.out()
```

This produces a medium sized square with a slight smoothing.

2. **Circle:**

```
  shape(100, 0.5, 1)
.out()
```

Using a high number of sides (like 100) visually results in a circle.

3. **Kaleidoscope effect:**

```
  shape(6, 0.5, 0.1)
.repeat(4, 4)
  .out()
```

By using `repeat`, we create a repeating effect which, combined with a hexagonal shape, can look like a kaleidoscope.

Machine Translated by Google

### Applications and Combinations

The `shape` function is often combined with other Hydra functions to generate dynamic and complex visuals. For example:

- **`rotate`** to rotate the shape,

- **`modulate`** to create distortions based on other shapes or textures,

- **`scale`** to adjust the size dynamically.

### ChatGPT Review

The `shape` function is an essential starting point for visual creation in Hydra. Its simplicity and flexibility make it very accessible, even for beginners, while providing a
depth that can be exploited by advanced users to design sophisticated visual compositions. It not only allows for the generation of basic aesthetic shapes, but can also serve as a structuring element for more elaborate effects when combined with other Hydra transformations.

GRADIENT

In Hydra, the `gradient` function generates a color gradient, making it an essential tool for creating fluid and dynamic visual effects. It allows you to manipulate hues in a way that
continues, generating color transitions that can be modified to add color.
depth and subtle movements to audiovisual compositions.

Machine Translated by Google

### Syntax and Parameters

The `gradient` function in Hydra is simply used with a single parameter that controls the number of color cycles in the gradient:

```
gradient(speed)

.out()
```

1. **`speed`**: This parameter determines how fast the gradient moves, creating an animated effect. A positive value scrolls the gradient to the right (or clockwise if the shape is circular), while a negative value scrolls it to the left (or counterclockwise). A value of zero creates a static gradient.

### How Gradient Works and Uses

The `gradient` generates a sequence of hues (usually colors in HSL space) that move continuously at a specified speed. This type of gradient is a continuous band of changing colors, allowing you to create dynamic moods or hypnotic effects.

### Usage Example

Basic Gradient:

```
gradient(0.5)

.out()
```

This code generates a gradient with a moderate speed, scrolling slowly to the right

2. **Static Gradient:**

```
gradient(0)

.out()
```

Machine Translated by Google

This version generates a fixed gradient, without any movement animation.

3. **Fast Gradient:**

```
gradient(3)
.out()
```

By increasing the speed, we create an intense visual effect where the colors change rapidly, giving a pulsating effect.

### Combinations with Other Hydra Functions

The `gradient` function is often used with other visual operations in Hydra to achieve more sophisticated effects. For example:

```
noise()
  .add(gradient(0.5))
.rotate(Math.PI / 2)
.colorama(0.1)
.out()
```

- **`rotate`**: The rotation effect applied to a gradient allows you to create color swirls or psychedelic rotation effects. For example:

```
rotate(Math.PI / 2)
```

This code rotates the gradient at right angles to achieve a vertical scrolling effect.

- **`colorama`**: This feature accentuates the impact of color changes, often used with `gradient` to achieve rapid color change effects.

### ChatGPT Review

Machine Translated by Google

`gradient` in Hydra is a fundamental tool for adding visual depth and fluidity to a composition. It acts as a "color texture" that can be layered o
combine with other functions to create a wealth of visuals. Although it is simple in its basic use, its combination possibilities make it very powerful, especially when used with modulation and distortion functions. Used effectively, `gradient` brings

both a dynamism and a hypnotic dimension that make Hydra's visual compositions captivating and immersive.

SRC

`src` is an **image or texture source**. It is a starting point for manipulating or combining visuals in a composition. Sources in Hydra typically include computer-generated patterns, cameras, or other video inputs. Here is more information on `src`

### How `src` works

- `src` is used to denote a **texture source** from a stream or input specific

- By default, it refers to the **first video source or camera** connected to the system (or the black screen at the start of a new session?).

- `src` can also be directed to other sources, such as **Hydra generated video** or external streams

### Basic example

Let's say you want to use a camera or video stream as a source in Hydra

```
s0.initCam()

src(s0)

.out(o0)
```

If a camera is configured as the primary source, it will be accessible directly as visual input.

### Advanced Usage

You can combine `src` with other functions to apply visual effects:

Machine Translated by Google

1. **Add filters:

```
s0.initCam()
src(s0)
.invert()
.modulate(osc(2),0.5)
.out(o0)
```

invert inverts the colors of the buffer `o0`.

2. **Apply masks or mixing effects:**

```
src(o0)
  .blend(osc(10, 0.1, 0.5), 0.5)
  .out()
```

Here, `src(o0)` is mixed with an oscillation (`osc`) for a dynamic effect.

3. **Create feedback:**

```
src(o0)
  .scale(1.01)
  .rotate(0.1)
  .out(o0)
```

This creates a feedback loop with a zooming and rotating effect.

### Links to other Hydra concepts

- **Buffers**: Buffers (e.g. `o0`, `o1`, etc.) are often combined with `src` to manipulate generated content in real time.

Machine Translated by Google

- **External Cameras**: If you plug in a USB camera or use an online video stream, `src
can capture and integrate this content into your visuals.

In summary, `src` is a fundamental building block in Hydra that allows you to define and manipulate visual sources. It is often used to work with input streams and create complex compositions in a visual live-coding environment.

SOLID

In **Hydra** the **`solid()`** function or method is used to create a solid color or uniform background in a visual. It generates a texture that fills the screen or assigned area with a constant color. Here is a more detailed explanation:

## **Syntax:**

```
solid(r, g, b, a)
```

## **Parameters:**

1. **`r` (red)**: Intensity of the red component (between 0 and 1).

2. **`g` (green)**: Intensity of the green component (between 0 and 1).

3. **`b` (blue)**: Intensity of the blue component (between 0 and 1).

4. **`a` (alpha)**: Opacity of the color (between 0 and 1, optional). By default, the opacity is 1 (opaque).

## **Easy to use:**

Create an opaque red background:

```
solid(1, 0, 0).out()
```

Result: the screen will be completely red.

Machine Translated by Google

## **Advanced Usage:**

1. **Changing color with animations:**

Functions like `Math.sin()` can be used to animate values and create smooth transitions.

```
Solid(()=>Math.sin(time), ()=>Math.cos(time), 0.5)

.out()
```

In this example, the color changes dynamically based on time.

2. **Combination with other functions:**

The texture generated by `solid()` can be combined with other functions (e.g. `osc`, `shape`, etc.) to create more complex effects:

```
solid(0.1, 0.1, 0.8)

  .mult(osc(10, 0.5, 1))

  .out()
```

This produces a pattern where the uniform color is modulated by an oscillating wave.

3. **Fusion with textures:**

`solid()` can be mixed with other layers or operations like `add()`, `sub()`, or `blend()` can be applied:

```
 shape(4,0.2,0.5)

  .blend(solid(0, 0, 0.5, 0.5), 0.2)

  .out()
```

Here a semi-transparent blue layer is mixed with the current output.

Machine Translated by Google

## **Typical applications:**

- **Basic background** for visual compositions.

- Added specific color to mix with animated textures

- Generation of smooth transitions in a visual.

GEOMETRY

ROTATE

In **Hydra**, the `rotate` function is used to rotate a texture or image on the Z axis, applying a rotation in two-dimensional (2D) space. This allows you to visually transform the direction or orientation of a texture on the screen.

### Basic syntax

```
rotate(angle, speed)
```

### Settings

- **`angle`**: a number representing the angle of rotation, expressed in radians. The angle can be dynamic (for example using a function like `time` or a mathematical value) or static.

A permanent rotation can be set (example given in “hydra functions”:

```
osc(50).rotate( () => time%360 ).out(o0)
```

- **`speed`** : sets the rotation speed dynamically as in this example given in “hydra functions”:

```
osc(10,1,1)

  .rotate( () => time%360, () => Math.sin(time*0.1)*0.05 )
```

Machine Translated by Google

```
  .out(o0)
```

### Functioning

1. **Clockwise rotation**: if the angle is positive.

2. **Counterclockwise rotation**: if the angle is negative.

### Example of use

#### Example 1: Static rotation

```
osc(10, 0.1, 1)
  .rotate(Math.PI / 4) // Rotate 45 degrees (PI / 4 radians)
  .out()
```

Here an oscillating wave is generated and rotated by 45°.

#### Example 2: Dynamic rotation over time

```
osc(10, 0.1, 1)
  .rotate(time % (2 * Math.PI)) // Continuous rotation through 360°
  .out()
```

In this example, the texture rotates continuously, as the angle value changes over time.

#### Example 3: Rotation combined with other transformations

Machine Translated by Google

```
osc(10, 0.1, 1)
  .rotate(Math.sin(time) * Math.PI) // Oscillation of the angle between -180° and 180°
  .modulateRotate(osc(5).rotate(0.3))
  .out()
```

This code combines texture rotation with dynamic modulation.

### Practical cases

- **Create smooth animations** where textures loop in a loop

- **Psychedelic effects** by coupling `rotate` with functions like `modulate` or `kaleid`.

- **Manipulate complex patterns** to create optical illusions or geometric effects.

SCALE

In **Hydra**, the `scale()` function is used to adjust the size of system-generated visual elements. This can affect shapes, textures, or images, increasing or decreasing their scale in visual space.

### How `scale()` works

1. **Signature** :

```
scale(x, y, z)
```

  - `x`: Scale on the horizontal axis (width).

  - `y`: Scale on the vertical axis (height).

  - `z`: Scale on depth (only used in 3D contexts).(?)

  If you specify only one argument, it will be applied uniformly across all axes

Machine Translated by Google

2. **Simple example:**

```
shape(4) // creates a square
  .scale(0.5) // scales to 50%
  .out()
```

Here the square will be smaller than its default size.

3. **Independent scales:**

```
shape(4)
  .scale(1, 0.5) // normal width, but height reduced to 50%
  .out()
```

4. **Dynamic scale:**

You can use functions or variables to animate the scale:

```
shape(3)
  .scale(() => Math.sin(time) * 0.5 + 1) // dynamic variation
  .out()
```

### Creative Use

- **Zoom in/out**: Change the scale to create a progressive zoom effect.

- **Psychedelic effects**: Combine `scale()` with functions like `modulate()` or `rotate()` for more complex animations.

Machine Translated by Google

- **Dynamic Textures**: Adjust the scale of textures or images for smooth transformations

In summary, `scale()` is a powerful tool for manipulating the size of visual elements, either statically or dynamically. Combined with other functions, it allows you to create interactive and impressive visuals.

PIXELATE

In **Hydra**, the `pixelate()` function is used to apply a **pixelation** effect to a texture or visual output. It reduces the image resolution by grouping pixels into blocks, creating a retro or artistic style effect.

### How `pixelate()` works

1. **Signature** :

```
pixelate(x, y)
```

- `x`: Number of “pixels” (blocks) on the horizontal axis.

- `y`: Number of “pixels” (blocks) on the vertical axis.

If you specify only one argument (`x`), it will be applied uniformly to the horizontal axis e vertical.

2. **Visual effect**:

The function takes the current texture or image and subdivides it into blocks of sizes defined by the arguments. The smaller the values of `x` and `y`, the larger the blocks, and therefore the more pronounced the pixelation effect.

Machine Translated by Google

### Simple examples

1. **Uniform rasterization**:

```
shape(4)
  .pixelate(10, 10) // 10 horizontal and vertical blocks (?)
  .out()
```

This example will rasterize a square with uniformly sized blocks (?)

2. **Asymmetrical pixelation**:

```
voronoi(5)
  .pixelate(20, 5) // more horizontal blocks than vertical
  .out()
```

Here the image will be pixelated with horizontally wide but vertically thin blocks.

3. **Dynamic Pixelation**:

You can animate the parameters to create a moving effect:

```
osc(10, 0.1, 1)
  .pixelate(() => Math.sin(time) * 20 + 30, 15)
  .out()
```

This creates an effect where the horizontal blocks change size dynamically over time.

Machine Translated by Google

-### Creative use

- **Retro Effect**: Simulate the appearance of old screens or low-resolution graphics.

- **Glitch aesthetics**: Pair `pixelate()` with functions like `modulate()` or `kaleid()` for complex visuals.

- **Detail Hiding**: Reducing the resolution to simplify or abstract a texture

- **Interesting transitions**: Vary `x` and `y` to gradually move from a clear texture to a pixelated texture.

### Advanced example: combination with other effects

```
osc(20, 0.1, 0.8)
  .pixelate(10, 10)
  .modulate(noise(3), 0.5) // adds distortion through noise
  .out()
```

Here, `pixelate()` acts upstream, then the effect is mixed with dynamic noise for a unique rendering.

In summary, `pixelate()` is a versatile and very useful function for playing with resolution and visual abstraction.

REPEAT

In **Hydra,** the repeat function is used to duplicate a texture or shape on a grid, creating a tiled repeating effect. This can be useful for generating complex patterns or visual structures from a single texture.

Machine Translated by Google

Syntax of repeat

repeat(x, y)

- **x :** number of repetitions (or frequency) of the texture in the horizontal direction.
- **y :** number of repetitions in the vertical direction.

Functioning

The function modifies the texture periodically, replicating it a certain number of times depending on the values provided for x and y. The higher the values of x and y, the smaller the texture will be divided into.

Simple example

```
osc(10, 0.1, 1)

  .repeat(3, 2) // Repeats 3 times horizontally and 2 times vertically

  .out()
```

- Here an oscillator is generated, then repeated 3 times in width and 2 times in height, creating a grid of patterns.

Dynamic parameters

Values passed to repeat can be animated or dynamically generated to create more vivid visual effects. For example:

```
osc(20, 0.1, 1)

  .repeat(Math.sin(time) * 5 + 5, Math.cos(time) * 5 + 5)

  .out()
```

- The repetitions vary over time, making the effect dynamic

Combination with other functions

repeat is often combined with functions like modulate, scale or rotate to create complex patterns:

```
osc(5, 0.2, 0.8)

  .repeat(4, 4)

  .rotate(0.5)

  .modulate(osc(10), 0.3)

  .out()
```

- This produces a repetition with rotation and modulation to visually enrich the result.

Machine Translated by Google

Points to note

- If x or y is set to 1, there will be no repetition in that direction.
- Fractional values for x or y create interesting distortions.

In summary, repeat is a powerful tool in Hydra for introducing symmetry and repeating patterns into your visual compositions.

REPEATX

Here is a new explanation of the repeatX function in Hydra, with working examples.

repeatX function

The examples need to be reviewed!

repeatX is used to repeat a texture horizontally on the canvas. The parameters are:

```
.repeatX(frequency, offset)
```

- frequency: number of horizontal repetitions.
- offset *(optional) :* horizontal offset between repetitions, expressed in proportion to the texture width.

New examples

Example 1: Repeating a sine wave horizontally

```
osc(10, 0.5, 1) // Generates a sine wave

    .repeatX(4) // Repeats this wave 4 times on the X axis

    .out()                   // Display the result
```

Here the wave texture is repeated four times horizontally.

Example 2: Add an offset to repeats

```
osc(10, 0.5, 1) // Generates a sine wave

    .repeatX(6, 0.2) // Repeat 6 times with an offset of 0.2

    .out()                        // Display the result
```

Machine Translated by Google

With an offset of 0.2, each repeat is slightly shifted horizontally, creating a phase shift effect.

Example 3: Animate the offset over time

You can use the time variable to animate the shift.

```
osc(5, 0.1, 1)                                  // Generates a sine wave
    .repeatX(5, Math.sin(time) * 0.3) // The offset varies with a sinusoid
    .out()                                      // Display the result
```

The offset is controlled by a sine function that varies with time, giving a smooth wave effect

Example 4: Repeat a shape and change the color

```
shape(4, 0.5, 0.1) // Generates a shape with 4 sides (a diamond)
    .repeatX(3)                 // Repeat 3 times on the X axis
    .color(0.2, 0.4, 0.8) // Applies a color
    .out()                      // Display the result
```

This example shows a shape (diamond) repeated three times horizontally with a blue color.

Example 5: Combine repeatX with repeatY

To get a repeat grid, you can combine repeatX with repeatY:

```
shape(6, 0.3) // Generates a hexagonal shape
    .repeatX(5) // Repeat horizontally 5 times
    .repeatY(4) // Repeats vertically 4 times
    .out()             // Display the result
```

This produces a grid of hexagons.

REPEATY

In **Hydra Video Synth,** the repeatY function is a transformation that acts on a visual stream (or texture). It allows you to duplicate the image vertically by repeating its pixels on the Y axis, which creates a vertical repetition effect in the visual composition.

Machine Translated by Google

General operation of repeatY

The signature is typically as follows:

```
repeatY(yScale, yOffset)
```

- yScale: Sets the number of vertical repetitions. The higher this value, the more the image will be split and repeated into smaller fragments along the Y axis. A value of 1 means a single repeat (no duplication), while a value of 2 splits the image into two vertical repeats.
- yOffset: Sets a vertical offset applied after the repeat. This allows vertical repeats to slide on the Y axis, creating an animation or offset effect.

Example of use

```
osc(10, 0.1, 1)

   .repeatY(4, 0.2)

   .out()
```

- osc(10, 0.1, 1): Generates a visual oscillator with frequency bands.
- repeatY(4, 0.2): Repeats the oscillator 4 times on the Y axis and applies an offset of 0.2.
- out() : Sends the result to the visual output.

Visual effects obtained

- **With a high yScale :** You get finer image repetitions, as if it were was subdivided into several vertical segments.
- **With an animated yOffset :** You can create a fluid or cyclical movement by modifying dynamically this value, for example:

```
osc(10, 0.1, 1)

 .repeatX(6, Math.sin(time) * 0.5) (specify the role of Math.sin)

   .repeatY(6, Math.sin(time) * 0.5)

   .out()
```

Interaction with other functions

repeatY can be combined with other transformations to enrich the visuals:

- scale or rotate: To adjust the scale or angle of the image after repetition.
- modulate **or** blend: To mix the repeated stream with other fl

Machine Translated by Google

In summary

repeatY is an essential tool for creating vertical repeating patterns, tiling effects, or manipulating the dimensions of a visual composition in **Hydra Video Synth.** It opens up interesting creative possibilities, especially when combined with dynamic modulations or animations.

KALEID

General operation of kaleid

The function signature is as follows:

```
kaleid(nSegments)
```

- nSegments: The number of segments (or divisions) used for the kaleidoscope pattern. The larger this value, the more complex the effect will be with many repeating patterns. A low value (like 2 or 3) will produce an effect with little symmetry, while a high value (like 10 or 20) will generate finer and more detailed patterns.

Example of use

1. Basic application

```
osc(10, 0.1, 1)
   .kaleid(6)
   .out()
```

- osc(10, 0.1, 1): Creates a visual oscillator with colored bands.
- kaleid(6): Applies a kaleidoscope effect with 6 symmetrical segments.
- out() : Sends the result to the output stream

2. Dynamic Kaleidoscope

For a more lively effect, you can dynamically vary the number of segments using a function like Math.sin(time):

Machine Translated by Google

```
osc(20, 0.1, 2)

  .kaleid(Math.floor(6 + Math.sin(time) * 4))

  .out()
```

In this example, the number of segments fluctuates between 2 and 10, which animates the kaleidoscopic effect.

Visual effects

- **Increase** nSegments: Makes the image more complex, with more symmetrical patterns. many.
- **Decrease** nSegments: Simplifies the pattern, with larger and more obvious shapes
- **Combination with other transformations :**
  - rotate: Rotate the image for smooth movement of the pattern
  - modulate **or** blend: Add interactions with other visual flows
  - repeat **or** repeatX/Y: Reinforce repeating patterns for a more structured kaleidoscopic effect.

Advanced example with modulations

You can combine kaleid with oscillators, modulations, and other effects to create a complex composition:

```
osc(40, 0.2, 1)

  .kaleid(8)

  .modulateRotate(osc(10).kaleid(4), 0.2)

  .out()
```

- kaleid(8): Kaleidoscopic effect with 8 segments.
- modulateRotate: Adds a dynamic rotation modulated by another oscillator with a kaleidoscope.

In summary

The kaleid function in **Hydra** is a powerful tool for generating geometric patterns and hypnotic effects based on reflected symmetries. It is ideal for creating abstract visuals

Machine Translated by Google

animated mandalas or psychedelic compositions, especially when combined with other transformations or dynamic modulations.

The role of Math.floor

You are right, the precision on the use of Math.floor is important to understand its role in the dynamic example of the kaleid function. Here is a detailed explanation of its role and usefulness in this context:

Role of Math.floor in the example

In this example:

```
osc(20, 0.1, 2)
  .kaleid(Math.floor(6 + Math.sin(time) * 4))
  .out()
```

Functioning :

1. Math.sin(time): Generates a value that oscillates between -1 and 1 over time, creating a fluid and continuous movement

2. 6 + Math.sin(time) * 4 : Converts this oscillation into a range of values from from **2 to 10.**

   - **Why?:** The Math.sin(time) function is multiplied by 4, which gives an amplitude of 8 (from -4 to +4). By adding 6, the result is centered around 6, with an oscillation between 2 and 10.

3. Math.floor: Rounds this result down to ensure that the parameter passed to kaleid is an **integer.**

   - **Why an integer?:** The kaleid(nSegments) function requires an integer to determine the number of segments. Without Math.floor, the oscillating value could be a decimal number, which would not work.
     correctly.

Without Math.floor

If Math.floor is omitted, kaleid will receive a decimal value, such as **6.73,** which can produce unexpected behavior or errors in Hydra. The segments would not be defined
correctly, because kaleid uses discrete divisions to create the symmetries.

Machine Translated by Google

Example without and with Math.floor

Without Math.floor:

```
osc(20, 0.1, 2)

    .kaleid(6 + Math.sin(time) * 4) // May cause problems with non-integer values


    .out()
```

- Here, 6 + Math.sin(time) * 4 produces values like **7.56** or **3.24,** which can create unexpected artifacts or effects.

With Math.floor:

```
osc(20, 0.1, 2)

    .kaleid(Math.floor(6 + Math.sin(time) * 4)) // Values are rounded to 7, 3, etc.


    .out()
```

- Using Math.floor ensures that kaleid only receives integers (eg: 6, 7, 8), which ensures smooth and predictable behavior

In summary

- Math.floor plays a key role in transforming a float value (like 7.56) into a u integer (like 7).
- This is essential for functions like kaleid which require **integer parameters** to work properly.
- Adding Math.floor therefore helps to avoid unexpected behavior and ensures that the kaleidoscopic effect remains stable and consistent.

SCROLL

In **Hydra,** the scroll() function is used to move or scroll a pattern or texture on the horizontal (x) or vertical (y) axis. This is a very useful feature to add movement to the generated visuals, creating a translation effect.

Syntax:

```
.scroll(xAmount, yAmount, speedX, speedY)
```

Settings:

Machine Translated by Google

1. **xAmount** *(required) :*
   - Sets the horizontal offset (x-axis)
   - The value can be positive (move to the right) or negative (to the left).
2. **yAmount** *(required) :*
   - Sets the vertical offset (y-axis)
   - The value can be positive (upward movement) or negative (downward movement).
3. **speedX** *(optional) :*
   - Scroll speed on the x axis
   - By default, no animation is applied if the speed is not specified
4. **speedY** *(optional) :*
   - Scroll speed on the y axis
   - By default, no animation is applied if the speed is not specified

Simple example:

Here is an example where a texture is generated and scrolled horizontally and vertically

```
speed = 0.001

shape(4)

 .scroll(0.1, 0.2,()=>Math.sin(time),()=>Math.sin(time)) /// Slowly moves the pattern on both axes with scroll speeds on the x and y axes


 .out()
```

Or :

```
osc(20, 0.1, 1)

      .scroll(0.1, 0.2, 0.01, -0.02) // Scrolls horizontally and vertically at
different speeds

      .out()
```

Use with modulators:

For more dynamic effects, oscillators or other functions can be used to modulate the scroll parameters.

```
shape(4)
```

Machine Translated by Google

```
    .scroll(() => Math.sin(time) * 0.1, () => Math.cos(time) * 0.1) // Dynamic animation based on time

.diff(osc(10))

    .out()
```

Notes:

- The scroll() function does not "cut" the image; it creates an infinite repetition effect thanks to to the cyclical nature of the patterns in Hydra.
- It is often used in combination with other functions (rotate, modulate, etc.) for more complex visuals.

Summary : scroll() in Hydra allows you to introduce fluid and repetitive movement into your visuals by moving patterns or textures on the horizontal and vertical axes, with speed control options for continuous or dynamic animations.

SCROLLX

In **Hydra,** the scrollX function is used to move a pattern, texture or flow video horizontally (on the X axis) across the canvas, creating a horizontal scrolling effect It can be combined with other functions to generate complex and dynamic animations.

Basic syntax:

```
scrollX(amt, speed)
```

Settings:

1. **amt** *(float) :* the amount of horizontal scrolling, expressed as a proportion of width of the canvas. For example:

   ÿ 0.5 : the image is offset by half the width of the canvas.

   ÿ 1 : The image is shifted by the entire width of the canvas (repeated in a loop).
2. **speed** *(float) :* the speed of the scrolling, expressed in cycles per second. A posi scrolls to the right, while a negative number scrolls to the left

Simple example:

```
osc(10, 0.1, 1)
    .scrollX(0.5, 0.1)
    .out()
```

- **osc(10, 0.1, 1)** generates a wave-like oscillation.
- **.scrollX(0.5, 0.1)** moves this oscillation horizontally by 50% of the canvas width, with a scrolling speed of 0.1 units per second

Machine Translated by Google

Example with dynamic modulations:

```
osc(20, 0.05, 0.8)
  .scrollX(() => Math.sin(time) * 0.3, 0.2)
      .out()
```

In this example:

- **Math.sin(time) * 0.3** dynamically oscillates the horizontal position between -0.3 and 0.3.
- Scrolling also occurs at a constant speed of 0.2 units per second

Advanced usage:

scrollX can be combined with textures, effects like modulate, or even video inputs (s0 for webcam, s1 for a secondary source). Here is an advanced example:

```
voronoi(10, 0.5, 2)
      .scrollX(0.1, 0.05)
      .modulate(noise(3), 0.2)
      .out()
```

This code generates a texture using voronoi, applies slow horizontal scrolling with scrollX, and modifies the result with a noisy texture

In summary:

- scrollX is perfect for creating horizontal motion effects or transitions fluid
- Combination with dynamic functions or modulators adds depth to your visuals.
- You can use it to sync movements with audio or other time parameters.

SCROLLY

Same reasoning as for scrollX but on the y axis.

Machine Translated by Google

COLOR

POSTERIZE

In **Hydra,** the posterize function is used to reduce the number of shades or levels in a texture, creating a posterized effect, where the image appears divided.
into distinct areas of color or value, without gradual transitions. This can produce a graphic rendering, similar to an image with a limited number of colors.

posterize syntax

posterize(steps, gamma)
Settings:

1. **steps** *(number) :* Sets
   the number of levels (or "steps") in each color channel. A value
   low (e.g. 2 or 3) produces a very segmented effect with little nuance, while a high value retains more detail.

2. **gamma** *(number) (optional) :* Adjusts
   the contrast of the color levels. A value greater than 1 increases the contrast between the levels, while a value less than 1 makes them more uniform. If this parameter is omitted, a default gamma is used.

Functioning :

The posterize effect acts on the color channels (red, green and blue) by quantizing their values
into a number of levels. Each pixel is adjusted to belong to one of these predefined levels

Examples of use:

Simple effect with 3 levels:

```
osc(10, 0.1, 1)

 .posterize(3)
```

Machine Translated by Google

```
.out()
```

Here the oscillator is reduced to only 3 color levels in each channel, creating a very graphic effect
With adjusted gamma:

```
osc(20, 0.1, 1)
 .posterize(5, 2)
 .out()
```

This applies 5 levels of color with increased contrast (gamma = 2), making the transitions between levels more pronounced.

Combined with other effects:

```
noise(5, 0.1
.posterize(4, 0.
 .kaleid(6)
.out()
```

Here the poster effect is combined with noise and a kaleidoscope effect, producing a complex.

Dynamic Level Modulation: You can dynamically animate the number of levels using a function like Math.sin :

```
osc(10, 0.1, 1)
.posterize(() => Math.floor(Math.abs(Math.sin(time) * 10)), 1.
 .out()
```

This varies the number of levels over time, creating an ever-changing effect.

Summary :

The posterize function is ideal for creating stylized and graphic effects by simplifying the textures or images. It is particularly useful in visual performances to achieve a unique and eye-catching rendering, especially when combined with other transformations such as scrolling, noise or oscillators.

Machine Translated by Google

SHIFT

Artificial Intelligence offers me an explanation that seems false to me! I take up the information given in the page dedicated to Hydra functions already mentioned above.

To be completed!

shift(r = 0.5, g, b, a) // a = alpha/transparency

Shift shifts (and "wraps") the values of r, g, b and/or a!

// default

```
osc()

.shift(0.1,0.9,0.3)

.out()
```

An example with saturate():

```
 osc(10, 0.1, 1)

  shift(0.2,2,0.9,0.3)

.saturate(20)

 .out()
```

INVERT

In **Hydra Video Synth,** the invert function is used to invert the colors of an image or video stream. More precisely, it subtracts each color value from 1, which is
to an inversion in the RGB color space.

Syntax

invert(amount)

Setting

- **amount :** A number between 0 and 1 that controls the intensity of the inversion.

Machine Translated by Google

ÿ **1 :** Complete inversion (each color is replaced by its opposite). **ÿ Intermediate values :** Apply a partial inversion, mixing the original and inverted colors.

Functioning

Color inversion is done by subtracting each color component from 1. For example, for a pure red pixel (R: 1, G: 0, B: 0), the full inversion gives:

• R: 1 ÿ 0 (inverse of 1), • G: 0 ÿ 1 (inverse of 0), • B: 0 ÿ 1 (inverse of 0), resulting in the color cyan (R: 0, G: 1, B: 1).

Example of use

```
osc(10, 0.1, 1) .invert(1) //
    Full inversion .out()
```

In this example:

• An oscillator (osc) generates visual waveforms. • The invert(1) function applies a total inversion of colors.

For a partial inversion:

```
osc(10, 0.1,
    1) .invert(0.5) // Invert to 50% .out()
```

Combination with other functions

invert is often combined with other functions like modulate, colorama or kaleid to produce dynamic and psychedelic visual effects.

For example :

```
osc(20, 0.05,

    1) .colorama(0.5) .invert(0.7) .modulate(noise(3), 0.2) .out()
```

This code generates a complex composition with inverted colors, modulation effects, and noise.

Creative applications

- Create dramatic contrast effects.
- Simulate a “negative mode” for visuals.

Machine Translated by Google

- Introducing visual variations into live performances.

In summary, invert is a powerful function to manipulate colors and bring dynamic contrast or unexpected visual effects in Hydra.

CONTRAST

In **Hydra** the **contrast** parameter allows you to modify the contrast of the colors of a texture o of an image generated in Hydra. In simple terms, contrast adjusts the difference between light and dark areas of an image.

Functioning

The main method for applying contrast is the .contrast() function. Here are its main features:

- **Syntax : .contrast(amount)**
- where amount is a number that controls the intensity of the contrast.
- **Settings :**

  ÿ amount : A numeric value that determines the contrast level.

  ÿ A value of **1** corresponds to the original contrast (no modification

  ÿ A value **greater than 1** increases contrast by reinforcing the differences between light and dark colors.

  ÿ A value **less than 1** decreases the contrast, making the colors more uniforms.

Simple example

Here is some example code in Hydra to illustrate the use of .contrast() :

```
osc(10, 0.1, 1)
    .contrast(1.5) // Increases the contrast
    .out()
```

In this example:

- osc(10, 0.1, 1) generates a wave with brightness variations.
- .contrast(1.5) increases contrast, making bright areas brighter and dark areas darker.

Combined effects

The contrast effect can be combined with other transformations like saturation, blur or blending effects to create dynamic visuals. For example:

```
osc(20, 0.1, 0.8)
```

Machine Translated by Google

```
.color(1, 0.5, 0.3)
.contrast(2)
.modulate(noise(3), 0.2)
.out()
```

Creative applications

- Accentuate texture differences for more impactful visuals.
- Create specific graphic styles (for example, a high-contrast effect often used in glitch art).
- Prepare an image to interact with additional effects in a visual workflow complex.

In summary, .contrast() is a key tool for adjusting the visual impact of textures in Hydra, allowing you to play with the dynamics between light and dark areas to enrich your visual creations.

BRIGHTNESS

In **Hydra Video Synth,** the **brightness()** function is used to adjust the brightness of a texture or video stream in your composition. This transformation changes the percept
general brightness of the image without directly affecting colors or contrast.
Here is a detailed explanation of how it works:

Functioning :

- The **brightness()** function takes a single numeric argument that represents the intensity of adjustment.
- The default value is usually 0, which means no change. applied to brightness.
- **Positive values** increase brightness, making the image clearer.
- **Negative values** decrease the brightness, making the image darker.

Syntax:

brightness(amount)

Setting :

- **amount :** (Number) Controls the amount of brightness adjustment.
  - ÿ **0 :** No effect.
  - ÿ **Values > 0 :** Increase brightness.
  - ÿ **Values < 0 :** Reduce brightness.

Machine Translated by Google

Example of use:

Increase brightness:

```
osc(10, 0.1, 1).brightness(0.5).out()
```

In this example:

- The oscillator is used as a source.
- Brightness is increased by **0.5,** making the image brighter.

Reduce brightness:

```
osc(10, 0.1, 1).brightness(-0.5).out()
```

Here the brightness is reduced by **-0.5,** making the image darker.

Creative use:

- **Create atmospheres :** Adjusting the brightness allows you to give a feeling of intensity or of darkness to a scene.
- **Modulate dynamically :** Combine **brightness()** with functions like **modulate()** for dynamic light variations.

```
osc(10, 0.1, 1)
    .brightness(() => Math.sin(time) * 0.5) // Oscillating brightness

    .out()
```

Important Notes:

- **Subtle changes :** Avoid values that are too extreme to preserve the details of the image.
- **Interaction with other effects :** The function can be combined with other transformations (eg, **contrast(), saturation())** for more complex compositions.

If you are exploring other visual manipulations, feel free to experiment with **brightness()** in combination with oscillators, video inputs, or modulations.

Machine Translated by Google

LUMA

In **Hydra Video Synth,** the **luma()** function is used to isolate or manipulate a specific part of a texture or video stream based on its brightness levels. It acts as a wire to select areas of the image based on their light intensity.

How **luma() works**

The function works on the **luminance channel** of a texture, which is the perceived brightness or lightness of the pixels. Areas with a brightness that matches the user-defined threshold will be preserved, while others will be dimmed (or made transparent).

Syntax:

```
luma(threshold, tolerance)
```

Settings:

1. **threshold** *(Number) :*

   ÿ Sets the brightness level to keep
   ÿ **0 :** Keeps very dark areas.
   ÿ **1 :** Keeps very bright areas.

2. **tolerance** *(Number) :*

   ÿ Controls the transition around the threshold. A high value softens the selection of luminance areas. ÿ **0 :** No softening (areas outside the threshold are completely transparent). ÿ **Higher values :** Creates a smoother transition between visible and transparent.

Example of use:

Isolate bright areas:

```
osc(10, 0.1, 1).luma(0.7, 0.05).out()
```

In this example:

- Areas of the image with a luminance greater than **0.7** are preserved. • A **tolerance** of **0.05** softens the transition slightly.

Create a mask with the dark areas:

```
osc(5, 0.1, 1).luma(0.2, 0.1).out()
```

Here :

- Dark areas (with luminance less than **0.2)** are isolated.

Machine Translated by Google

- The tolerance of **0.1** softens the edges of the mask.

Advanced usage:

1. **With a video or webcam source :**

```
s0.initCam()

src(s0).luma(0.5, 0.1).out()
```

- This keeps areas of a live video with brightness close to **0.5**.

2. **In combination with modulators**:

```
osc(10, 0.2, 1)
 .luma(() => Math.sin(time) * 0.5 + 0.5, 0.1)
 .out()
```

- The brightness **threshold** varies dynamically over time, creating an oscillating effect.

3. **Create dynamic masks :** You can use **luma()** to overlay or mask textures based on their brightness. For example:

```
osc(20, 0.1, 1).layer(noise(2))

  .luma(0.5, 0.05)

  .out()
```

### **Creative Applications:**
- **Create masks**: Isolate specific parts of an image to apply effects or combine them with other layers.

- **Dynamic Effects**: By varying the **threshold** or **tolerance** interactively, you can achieve captivating light animations.

- **Layering Elements**: Use `luma()` as a tool to compose different sources or streams in Hydra.

### **Important Notes:**

Machine Translated by Google

- The effect works best with textures that have a wide dynamic range (like oscillators or high-contrast video inputs).

- **Experiment**: Combine **`luma()`** with other functions like **`contrast()`** or **`brightness()`** to adjust textures before or after applying the filter.

In summary, **`luma()`** is a powerful tool for working with luminosity and creating expressive and precise visuals.

TRESH

In Hydra, the real-time video synthesis software designed by Olivia Jack, the tresh function (short for "threshold") is used to apply a threshold effect to an image or video texture. **This effect transforms pixel values according to a defined threshold creating binary (?) zones based on their brightness or other parameters.** This can give "posterization" type visual effects or very marked contrasts.

How **tresh works**

The tresh function compares the brightness (or other property) of pixels to a certain threshold.
Here is an example of its syntax:

```
tresh(threshold, tolerance)
```

- **threshold :** Determines the threshold value. Pixels with a value greater than or equal to this threshold become white (or another color defined by the modifications signal), while the others turn black.
- **tolerance :** Introduces a range around the threshold, allowing for a smoother transition between light and dark areas.

Examples of use

Here is an example in Hydra where we apply the tresh effect :

```
osc(10, 0.1, 1)
    .tresh(0.5, 0.1)
    .out()
```

1. **osc(10, 0.1, 1) :** Generates an oscillator (visual waves) with a frequency of 10.
2. **.tresh(0.5, 0.1) :** Applies a threshold of 0.5 with a tolerance of 0.1.
3. **.out() :** Sends the signal to the visual renderer.

Machine Translated by Google

Another example:

```
osc(30)
.layer(osc(15)
.rotate(1)
.thresh())
.out(o0)
```

Practical applications

- **Glitch or minimalist aesthetic :** By simplifying textures with very bright areas contrasting.
- **Interactive effects :** When combined with audio sources or other interactive inputs.
- **Stylized transformations :** With other functions like modulate or mult, to create complex visuals.

If you work with Hydra in a creative context, tresh is a powerful function to manipulate the clarity and contrast of your visuals. Feel free to combine it with other operations to enrich your compositions!

COLOR

In **Hydra Video Synth,** the color() function or method is used to generate a uniform texture with a color defined by its three basic components: **red (R), green (G),** and **blue.**
(B). It is essential for creating visual bases or adding colorful effects on other textures. Here are the details of how it works:

color() syntax

```
color(r, g, b, a)
```

Settings:

- **r (red) :** A value between 0 and 1 representing the intensity of red.
- **g (green) :** A value between 0 and 1 representing the intensity of green.
- **b (blue) :** A value between 0 and 1 representing the intensity of blue.
- **a (alpha)** *(optional) :* A value between 0 and 1 representing transparency (by default, the value is 1, i.e. completely opaque).

Functioning :

- When you use color(), Hydra generates a texture filled with a uniform color according to the specified values
- This texture can be used directly or combined with other textures via functions like blend(), add(), or modulate().

Machine Translated by Google

Example of use:

1. **Base color :**

   color(1, 0, 0).out()

2. This produces a full red color.

3. **Color with transparency :**

4. .gradient().

5. color(0, 1, 0, 0.5)

6. .out()

7. **This creates a semi-transparent green texture.**

8. **Combined effect with other textures :**

9. osc(10, 0.1, 1)

10. .color(1, 0, 0)

11. .out()

12. Here the oscillator is colored red.

13. **Add a colored gradient with colorama() :**

14. osc(20, 0.1, 1)

15. .color(0.5, 0.5, 1)

16. .colorama(0.3)

17. .out()

18.

Important Notes:

- Color values follow the **RGB** (Red, Green, Blue) model in a floating scale between 0 and 1.
- color() is often used as a simple tool to experiment with colors or create complex visual effects when combined with other Hydra functions.

Machine Translated by Google

SATURATE

In **Hydra Video Synth,** the **saturate()** function is used to adjust the **saturation** of a texture, that is, the intensity of the colors present in the generated image or texture. It allows you to amplify or reduce the vividness of colors, which is useful for creating visual effects dynamic and expressive.

saturate() syntax

```
texture.saturate(amount)
```

Settings:

- **amount :** A number (positive or negative) representing the intensity of the saturation applied.
  - ÿ **Positive values :** Increase saturation, making colors more vivid.
  - ÿ **Negative values :** Reduce saturation, reaching a desaturated or completely grayscale effect.
  - ÿ **Default value :** If no value is specified, it is considered as (no changes)

Functioning :

1. **Increase saturation :**
   When you apply a positive number, the colors become more intense, creating an oversaturation effect where the colors can appear "exaggerated."
2. **Reduce saturation :**
   With negative numbers, colors lose their intensity, giving a dull or monochromatic effect.
3. **Color contrast effect :**
   Textures combined with saturate() can produce captivating visual effects, especially if they contain a wide range of hues.

Examples of use:

Increase the saturation of a simple texture:

```
osc(10, 0.1, 1)
 .saturate(2)
   .out()
```

Machine Translated by Google

This makes the oscillator colors much more vivid.
Reduce saturation:

```
osc(15, 0.2, 1)

    .saturate(-1) .out()
```

This desaturates the texture, making the image almost grayscale.

Saturation combined with other effects:

```
osc(20, 0.05, 0.8)



    .kaleid(5) .saturate(3) .modulate(osc(10,
    0.1).saturate(-2)) .out()
```

Here, saturation amplifies visual contrasts in a texture modified by a desaturated secondary oscillate.

Transition to a desaturated effect:

```
osc(25, 0.05, 1)

    .saturate(() => Math.sin(time) * 2) // Dynamic variation with time .out()
```

Saturation evolves dynamically with a sine function.

Practical applications:

- **To stylize visuals :** Create vibrant effects or soften colors depending on the desired ambiance.
- **To play with detail levels :** High saturation can make textures more visually aggressive, while low saturation can generate a minimalist effect.
- **For dynamic transitions :** Combine saturate() with temporal modulations for evolving effects.

In summary, saturate() is a powerful tool for playing with the richness of colors in Hydra and gives visuals additional aesthetic depth.

Machine Translated by Google

HUE

In **Hydra,** a programming platform for visual live coding, the **hue** function is used to manipulate the hue of an image or texture. It allows you to shift the colors of a visual composition by adjusting their hue on the color wheel, while preserving their saturation and brightness. Here is an overview of its role and how it works:

How **Hue works**

The .hue() method changes the hue of each pixel in an image. The hue is adjusted by function of a specified value, usually expressed as a float between -1 and

- **Positive value :** Moves the colors clockwise on the circle chromatic.
- **Negative value :** Moves the colors counterclockwise.
- **Zero value (0) :** Does not change the hue, leaving the colors unchanged.

Syntax

hue(value)

- **value :** A float that represents the amount of hue shift
  For example :
  ÿ hue(0.2) : Shifts colors slightly towards red.
  ÿ hue(-0.3) : Shifts the colors slightly towards blue/green.

Simple example

Let's take a base texture that produces waves and apply a hue shift:

```
osc(10, 0.1, 1) // Generates an oscillating texture
    .hue(0.5) .out()           // Shifts the hue by 50%
                               // Send the output to the screen
```

Dynamic effect

To create a dynamic effect, you can animate the hue over time with time :

```
osc(10, 0.1, 1)
    .hue(Math.sin(time) * 0.5) // Continuous animation on the hue

    .out()
```

Machine Translated by Google

Creative combinations

hue can be combined with other functions like modulate, invert or brightness to enrich the visual effects.

```
osc(20, 0.1, 1)
    .hue(() => Math.sin(time) * 0.3) // Dynamic variation
    .modulate(noise(3)) .out()                              // Adding modulation
```

Practical application

- **Smooth transitions** in live performances.
- **Creation of lively color palettes** in loop or animation.
- **Harmonization of visuals** with music or other media by synchronizing the hue shift with external data.

In summary, **hue** is a powerful tool for color manipulation in Hydra, adding a dynamic and aesthetic dimension to your visual compositions.

COLORAMA

In Hydra, `colorama` is a function designed to manipulate the colors of a visual stream and applying a rotation of hues to them. It allows to give a psychedelic or kaleidoscopic effect to the composition, which can energize the visuals by introducing subtle or intense changes in the colors.

### Detailed description of `colorama`

The `colorama` function essentially changes the hue of each pixel in an image by applying a rotation of colors. It acts like a filter that scans the color spectrum (red, orange yellow, green, blue, purple) in a cyclical manner, creating variations in hues. By adjusting the `colorama` parameter, one controls the speed and intensity of this hue rotation, influencing so the final visual rendering

### Main parameter of `colorama`

Machine Translated by Google

The function takes a single main parameter that controls the intensity and speed of color rotation:

- **`amount` parameter**: This is a number (positive or negative) that represents the rate of color change. **Higher values make the hue rotation faster, while lower values produce more subtle and slower changes. A negative value reverses the direction of color rotation.**

### Example of using `colorama`

Here is a simple example showing how to apply `colorama` to animate the colors of a noise pattern in Hydra:

```
noise(3, 0.1)           // Creates a noise pattern with a certain scale and speed
  .colorama(0.5)          // Applies a hue rotation with medium intensity
  .out()              // Send the result to the visual output
```

In this example:

- `noise(3, 0.1)` generates a texture noise pattern.

- `.colorama(0.5)` adds a medium intensity hue shift effect. The colors change cyclically, creating a hue animation effect on the noise pattern.

### Typical uses of `colorama`

- **Create psychedelic effects**: Rapidly sweeping colors can create a vibrant, energetic effect, ideal for immersive visual creations.

- **Adding movement to static visuals**: On fixed shapes or patterns
`colorama` allows you to add dynamism by playing only on the tint of the colors, without modifying the shapes or positions

Machine Translated by Google

- **Enriching textures**: By applying `colorama` over a base texture (like a noise pattern or a gradient), one can achieve subtle, aesthetic changes that add depth to the visuals.

### Notice

`colorama` is a powerful tool in Hydra for visual artists looking to explore shifting color effects without having to alter the underlying structure of their visuals. It is particularly effective in combination with other modulation functions (comm `modulate` and `osc`) to create immersive and captivating animations. Used sparingly, `colorama` can also introduce interesting color nuances into abstract and psychedelic visual works.

R (G and B)

In **Hydra Video Synth, R** represents the **red** channel of an image or texture. It is one of the three components of the **RGB** (Red, Green, Blue) color model, which determines the hue and intensity of red light in visual rendering.

Role of R in Hydra

1. **Set the intensity of the red color**

   The value of **R** ranges from **0** to 1.

   0 : No red.
   1 : Full intensity red.

   When you specify a color, **R** controls the contribution of red to the overall color

Simple example:

```
solid(1, 0, 0)

 .out()
```

2.

- **1, 0, 0** means: full intensity red, no green, no blue
- Result: the entire screen will be red.

How R influences textures in Hyd

1. **Apply a red tint to a texture :**

   Use the **R** parameter in the function

Machine Translated by Google

```
color()
. osc(10, 0.1, 0.8)
 .color(1, 0, 0)
 .out()
```

ÿ

Here :

ÿ The texture created by osc() is completely tinted red.

2. **Dynamically modulate R :** You can bind the value of **R** to functions math or external inputs (e.g. time or audio) to dynamically change the intensity of red.

```
osc(10, 0.1, 0.8)

    .color(() => Math.sin(time), 0, 0)
    .out()
```

Math.sin(time) : The value of red oscillates between 0 and 1 depending on time.

6. **"Split Channel" Effects :** By isolating or manipulating only the red channel you can create glitch or abstract effects.

```
osc(10, 0.1, 0.8)

    .color(1, 0, 0) // Pure red
    .layer(osc(10, 0.1, 0.8).color(0, 0, 1).scrollX(0.02)) //
Added horizontally offset blue
    .out()
```

7.

ÿ Here, the red remains fixed, but a horizontal shift of the blue creates a visual effect interesting.

Using R in Visual Calculations

Hydra allows you to directly manipulate RGB channel values in shaders or textures:

1. **Extracting the red channel from a texture :**

2.

```
osc(10, 0.1, 0.8).r().out() (to be reviewed)
```

3.

ÿ The .r() method extracts only the red channel from a texture, resulting in a grayscale image based on the intensity of red.

3. **Red modulation :** You can use **R** to modulate other parameters, such as the speed or scale of a texture.

Machine Translated by Google

```
4. osc(10, 0.1, 0.8)
5.          .modulate(noise(3).r(), 0.5)
6.          .out()
7.
```

- Here, the noise is modulated only by the intensity of red.

Summary

The **R** channel in Hydra is essential for:

- **Define colors** or create red-dominated hues
- **Dynamically modulate the intensity of red** for animated or interactive effects.
- **Directly manipulate the red channel** for glitch or abstract effects in computer-generated video.
-

BLEND

In **Hydra Video Synth,** the add() function is a method for **combining two visual signals** by adding their corresponding pixel values. This type of operation is commonly used in video synthesis to superimpose or merge shapes, textures, or animations.

How add() works

When you apply add(), the values of each pixel from the two inputs (the textures or visual sources) are added together. This can produce **increased brightness** effects or interesting visual combinations.

Here are some key points about the behavior of add() :

- Pixel values are added **by color channel** (red, green, blue, alpha).
- If the sum of the values exceeds 1.0 (in the normalized 0 to 1 scale used by Hydra), it is often **clipped** to 1.0, which can result in areas of saturated or white color.
- Operations may be affected by **the opacity** (alpha channel) of the combined textures.

Syntax of add()

```
// add(texture2, intensity)
add(src, amount)
```

- **src :** The visual source to add to the main texture.
- **amount** *(optional) :* A multiplier that adjusts the intensity of the second texture before it is added.

Practical examples

Machine Translated by Google

1. Simple addition

```
osc(10, 0.1, 1) // Base oscillator
    .add(osc(30, 0.2, 1)) // Add a faster oscillator
    .out()
```

This example superimposes two oscillators with different frequencies, producing a complex texture.

2. With adjusted intensity

```
osc(20, 0.1, 1)
    .add(osc(40, 0.2, 1), 0.5) // Reduces the intensity of the
second texture
    .out()
```

The second oscillator contributes to the final image with only 50% of its intensity.

3. Application on a video

```
s0.initCam() // Initialize the camera
src(s0)
    .add(osc(15, 0.2, 1)) // Overlay a texture
swinging on the video
    .out()
```

This creates an effect where an animated texture is added to the camera image.

Creative applications

- **Texture Overlay :** Add shapes, patterns, or videos to enrich a composition.
- **Highlight Effects :** Increase brightness or create visual highlights.
- **Creating new patterns :** Combine multiple sources to generate unique patterns.

Addition is a simple yet powerful tool in Hydra, allowing you to modulate your visual creations and explore a multitude of combinations.

SUB

In **Hydra Video Synth,** the sub() function is used to perform a **pixel-by-pixel subtraction** between two visual signals. This allows **the brightness, colors, or patterns** of one texture to be subtracted from another, producing often darker or more subtle visual effects, such as **inverted silhouettes** or sharp contrasts.

Machine Translated by Google

How **sub()** works

- The pixel values of the main texture are **reduced** by those of the texture secondary (input source).
- The subtraction is performed **channel by channel** (red, green, blue, alpha).
- If a pixel value becomes negative (which can happen in a 0 to 1 scale), it is **clipped to zero,** resulting in a black color.

Syntax of sub()

```
// sub(texture2, intensity)
sub(src, amount)
```

- **src :** The visual source to subtract from the main texture.
- **amount** *(optional) :* A multiplier applied to the secondary texture before the subtraction.

Examples of use

1. Adjusting the intensity

```
osc(20, 0.1, 1)
    .sub(osc(40, 0.2, 1), 0.5) // Subtraction with intensity
reduced by 50%
    .out()
```

This produces a more subtle effect, where subtraction is less dominant.

2. Silhouette effect with video

```
s0.initCam() // Initialize the camera
src(s0)
    .sub(osc(15, 0.2, 1).0.5) // Subtracting a texture
swinging on the video
    .out()
```

This effect makes it appear as if the oscillator patterns are "erasing" parts of the video, creating dynamic dark shapes.

Creative applications

1. **Creating dramatic contrasts :**

    ÿ Use sub() to reveal or hide parts of a texture based on a secondary source.

2. **Reverse silhouette effects :**

Machine Translated by Google

ÿ Subtracting a light texture from a dark source can create visual effects resembling inverted shadows or etched patterns.

3. **Interaction between shapes and textures :**

ÿ Combine complex shapes by subtracting dynamic textures (like oscillators) for richer patterns.

Comparison with add()

Appearance	**add()**	**sub()**
Main effect	Make the image brighter	Make the image darker
Function	Adds pixel values	Subtracts pixel values
Applications	Overlays, bright flashes	Silhouettes, shadows, contrasts

The sub() function is a powerful tool for exploring dark and abstract variations in your Hydra compositions. It allows you to visually sculpt your textures by playing on contrasts and absences.

LAYER

(TO BE IMPROVED)

In **Hydra Video Synth,** a *layer* is a basic element that allows you to build visual compositions by stacking and combining different graphic sources. These layers interact with each other to produce complex and dynamic effects. Each *layer* can represent a video stream or a texture generated in real time.

Understanding Layers in Hydra

1. **Basic Concept: Visual Stacking**
A *layer* acts as a transparent sheet where you can draw, apply effects or combine sources. Layers stack to create a final composition, each layer that can interact with others.
Think of *layers* like drawings in design software: they can be manipulated individually or merged.

2. **Main Tools for Managing Layers**
Hydra offers simple commands for working with layers:

ÿ **layer(n) :** selects a specific layer, where n is the index (0 for l first layer, 1 for the next, etc.).
ÿ **blend() :** merges two layers with transparency.
ÿ **modulate() :** uses one layer as a mask or modulation for another.

Machine Translated by Google

ÿ **out() :** makes visible the output of one or more layers.

Examples of Composition with Layers

Example 1: Overlay with Transparency

Oscillators with different settings can be layered to produce rich patterns.

```
// A blue oscillator in motion

osc(5, 0.2, 1).layer(osc(0.5, 0, 1)).out()
```

Example 2: Mask Effect with Modulation

Layers can interact like masks to clip or modulate textures.

```
// solid animated

solid([1,0,0],[0,1,0],
 [0,0,1],1).layer(noise().rotate(1).luma()).out(o0)
```

Example 3: Visual Feedback

Layers allow you to create feedback effects, where the output of a layer is fed back as a source.

EXAMPLE SA REVOIR

```
// Layer 0: Base Oscillator layer(0).src(osc(20,
0.1).rotate(0.3)).out()

// Layer 1: Adding visual feedback
layer(1).src(o0).scale(0.9).rotate(0.1).blend(layer(0), 0.7).out()
```

Machine Translated by Google

Example 4: Video Merge

Layers also allow *you* to mix videos, such as a webcam or a pre-recorded video.

```
s0.initVideo("https://media.giphy.com/media/3o7abldj0b3rxrZUxW/
giphy.mp4")

src(s0).

layer(

    src(o0)

        .scale(0.9)

        .rotate(0.1)

        .blend(src(s0), 0.5))

 .out()
```

BLEND

In **Hydra Video Synth,** the **blend** function allows you to combine two visual sources (or layers) by adjusting their relative opacity. It creates a linear interpolation between two textures or videos based on a given parameter.

How **blend works**

The general syntax is as follows:

```
blend(src, amount)
```

- **src :** The visual source or layer you want to mix with the current layer.
- **amt :** A parameter between 0 and 1 that controls the mixing level.
  - ÿ 0 means only the current layer is visible (no merging).
  - ÿ 1 means that only the new source (src) is visible.
  - ÿ Intermediate values (e.g. 0.5) create a balanced blend between the two layers.

Machine Translated by Google

Simple usage example

```
osc(10, 0.1, 0.5) // Generates an oscillator
    .blend(osc(20, 0.2, 0.5), 0.5) // Blend with a second
source
    .out()
```

In this example:

- A low frequency oscillator is generated in the background.
- Another oscillator, with a different frequency, is merged 50% with the first.

Dynamic effects with blend

For more complex effects, you can animate the **amt** parameter using a function like time or modulate.

Example :

```
osc(10, 0.1, 0.5)
    .blend(osc(20, 0.2, 0.5), Math.sin(time) * 0.5 + 2) //
Time-based dynamic fusion
    .out()
```

Here, the amt parameter oscillates using a sine function, creating a smooth transition between the two layers over time

Creative applications

- **Subtle Overlay :** Add texture or patterns to a video or animation.
- **Smooth transitions :** Gradually move from one source to another.
- **Glitch or psychedelic effects :** Mixing shapes, colors or patterns that move dynamically.

In summary, the **blend** function is a powerful tool in Hydra for creating dynamic and varied visuals by combining sources precisely.

Machine Translated by Google

MULT

Multi -function operation

The syntax:

mult(src, amt)

- **src :** The visual source or layer to multiply with.
- **amt** *(optional) :* An intensity factor between 0 and 1 (or more) that controls how much the multiplication affects the current layer.
  - ÿ 0 : No multiplication (current layer remains unchanged).
  - ÿ 1 : Normal multiplication (the default).
  - ÿ A value greater than 1 amplifies the effect

Controlled multiplication with amt

Simple example with an amt :

```
osc(10, 0.1, 0.5) // Base oscillator
    .mult(shape(4, 0.5), 0.5) // Multiplies with a shape with
a reduced effect (50%)
    .out()
```

In this example:

- The oscillator is combined with a shape using an intensity of 0.5.
- This reduces the effect of multiplication, making the shape less dominant.

Example with a dynamic animation of amt :

```
osc(10, 0.1, 0.5)
    .mult(shape(3, 0.5), Math.sin(time)) // `amt` oscillates between -1 and 1

    .out()
```

Machine Translated by Google

- Here, the intensity of the mixture is dynamically modulated by a sine function, creating a pulsation.

Comparison between mult and blend

- **mult** multiplies pixel by pixel, often used to **mask** or modulate layers visual.
- **blend** performs linear interpolation, useful for **superimposing** or mixing two sources in a gentle way.

Creative applications of mult with amt

1. **Dynamic mask effect :**

2.

```
osc(20, 0.1, 0.8)
```

3.
```
.mult(noise(3, 0.1), 0.7) // Control the strength of the
masking
```
4.
```
.out()
```
5.

6. **Complex transitions :**

7.

```
osc(15, 0.2, 0.5)
```

8.
```
.mult(osc(5).rotate(0.5), Math.sin(time) * 0.5 + 0.5) // Mixing with a source in
smooth transition
```
9.
```
.out()
```
10.

11. **Visual Amplification :** Use amt values greater than 1 for intensive some areas:

12.

```
osc(10, 0.1, 0.5)
```

13.
```
.mult(noise(3, 0.2), 1.5) // Amplifies the patterns generated by the noise
```

14.
```
.out()
```
15.

Conclusion

The **mult(src, amt)** function is a powerful tool in Hydra for controlling how layers interact visually. The **amt** parameter adds additional flexibility allowing you to finely adjust the impact of the multiplication, whether to create masks subtle or intense visual effects.

Machine Translated by Google

DIFF

In **Hydra Video Synth,** the diff operator is used to calculate the absolute difference between two layers (or visual sources). This is a method for creating visual effects where the differences between two frames or streams are highlighted, often with very dramatic results.
dynamic or textured.

Use in Hydra

Here is a typical example of using diff in a Hydra patch:

```
osc(10, 0.1, 1)
    .diff(osc(15, 0.05, 2).rotate(0.5))
    .out()
```

Explanation :

1. The first layer is an oscilloscope (osc) with a frequency of 10, a modulation speed of 0.1 and an intensity of 1.
2. The second layer is another osc with a frequency of 15, a modulation of 0.05, an intensity of 2, and an applied rotation of 0.5.
3. The diff takes these two layers and calculates their absolute difference to create an effect contrasted between them.

Typical diff effects

- **Dynamic textures:** Oscillating or moving layers produce patterns alive and constantly evolving.
- **Moiré effects:** With similar frequencies between two oscillators, diff can reveal complex interference patterns.
- **Contrast effects:** It is useful to highlight areas of divergence between two visual streams

Creative exploration

- Combine diff with other operators like .add(), .sub(), or .modulate() to generate more complex effects.
- Apply it to video sources or cameras to transform scenes into abstract patterns.
- Pair it with dynamic parameters (.scale(), .rotate(), .kaleid()) to experiment with evolving compositions.

Advanced example with a webcam:

Machine Translated by Google

```
s0.initCam()
src(s0)
    .diff(osc(20, 0.1, 0.5).rotate(0.2))
    .out()
```

This creates an interaction between the webcam input and an oscillator, producing an abstract and organic rendering.

In summary, diff is a powerful operator for playing with visual contrasts and revealing details or patterns in your generative video compositions.

MASK

In **Hydra Video Synth,** the mask() function is used to apply a mask to a visual source. A mask is a technique for filtering or hiding certain parts of
of an image or video stream based on a defined texture or criterion. This is an operat
common in image processing to create dynamic or complex visual effects.

How **mask() works**

The mask works by taking a **texture** (visual source) and using **another texture** or **threshold** as a "template" to define which parts of the source image will be visible.

Basic syntax:

```
src(source).mask(texture, scale, offset) (??)
```

Settings:

1. **source :** the video source or image to apply the mask to.
2. **texture :** the texture used as a mask. The light parts of this texture will let the original image through, while the dark parts will hide the image.
3. **scale** (optional): scaling factor to adjust the size of the mask relative to the source.
4. **offset** (optional): offset (x, y) to adjust the mask position.

Simple example:

// Creating a base source

```
gradient(5)
    .mask(osc(10, 0.1, 1)) // Applies a mask based on a
oscillator
    .out()
```

Machine Translated by Google

In this example:

- The source o0 is displayed, but only a part will be visible.
- The mask is generated by osc(10, 0.1, 1), an oscillating texture.
-

Advanced example with scale and offset:

```
osc(10)
    .mask(shape(4, 0.5, 0.3), [1.5, 1.5], [0.5, 0.5])
    .out() (??)
```

In this example:

- The mask is based on a shape .
- The scale is adjusted with [1.5, 1.5] to make the mask larger.
- The offset [0.5, 0.5] shifts the mask position.(?)
-

Use cases:

1. **Creating dynamic effects :** Limiting animations to certain areas.
2. **Artistic layering :** Combine multiple visual sources with masks.
3. **Transition effects :** Gradually introduce or disappear visual elements.

The **mask** in Hydra is particularly useful for experimenting and creating complex visuals in live performances or art installations.

MODULATE

MODULATEREPEAT

In **Hydra Video Synth,** the **modulateRepeat** function is an operator that applies repetitive modulation to an input texture using a modulation texture. This function is used to create complex, repetitive effects, often related to dynamic spatial patterns or distortions.

Here is a detailed explanation of the settings and how it works:

Syntax

```
modRepeat(src, repeatX, repeatY, offsetX, offsetY)
```

Machine Translated by Google

Settings

1. **src :** The modulation texture.

   ÿ This is the source that will be used to modulate the current image or texture.

   ÿ For example, you can use an oscillator (osc()) or any other texture.

2. **repeatX :** Horizontal repeat frequency.

   ÿ Sets how many times the texture will be repeated on the horizontal axis.

   ÿ A larger value results in more frequent repetitions.

3. **repeatY :** Vertical repetition frequency.

   ÿ Sets how many times the texture will be repeated on the vertical axis.

4. **offsetX** (optional): Horizontal offset.

   ÿ Allows you to shift repeats horizontally for a dynamic effect.

5. **offsetY** (optional): Vertical offset.

   ÿ Allows you to shift repetitions vertically.

Functioning

- This function applies a repeating pattern based on the coordinates generated by the modulation texture (src).
- Repeat settings allow you to divide the screen into a grid defined by repeatX and repeatY.
- The effect is particularly useful for creating symmetries, mosaics, or geometric structures modulated by a dynamic source.

Example of use

```
shape(4,0.9)
    .add(osc(3,0.5,1))
    .modulateRepeat(osc(10), 3.0, 5.0, 0.5, 0.5)
    .out(o0)
```

- **shape(4,0.9)** generates a square.
- **.add(osc(3,0.5,1))** is used as modulation texture.
- **3, 5** means that the texture is repeated three and 5 times on each axis (x and y
- **0.5, 0.5** adds a slight offset to make the effect more dynamic.

Common applications

1. **Creating repeating patterns :** For visual effects inspired by mosaics or art generative.
2. **Synchronization with audio :** Using dynamic textures based on input sound.
3. **Geometric exploration :** Playing with dynamic values to experiment with abstract effects.

Machine Translated by Google

MODULATEREPEATX

In **Hydra Video Synth,** the modulateRepeatX function is used to manipulate a texture by repeating it along the X (horizontal) axis and modulating that repetition based on another texture or source. This creates dynamic and complex visual effects, often used to generate repeating patterns with interesting distortions.

Syntax

```
modRepeatX(source, reps = 3, offset = 0.5)
```

Settings

1. **source**

   The texture or source used to modulate the repeat. This can be another video output or a generator in Hydra (like osc, shape, etc.).

2. **Reps**

   Controls the number of repetitions default to 3

3. **offset** *(default: 0.5)*

   Sets the modulation offset, influencing how the pattern is applied to texture.

Functioning

- **Horizontal Repeat :** modulateRepeatX starts by dividing the image or texture into multiple repeating sections along the X axis.
- **Dynamic Modulation :** Then the specified source (first parameter) is used for distort or disrupt these repetitions according to its own variations.
-

Practical Example

Here is a simple example to visualize the effect of modulateRepeatX :

```
noise(2).modulateRepeatX(gradient(), 30,0.9)
 .colorama(2)
 .out()
```

- **noise(2)** generates an oscillating texture.
- **modulateRepeatX(gradient(), 30, 0.9)** uses another oscillation (gradient()) to modulate the repetitions of the first texture.

Machine Translated by Google

- The result is a series of horizontal patterns, with dynamic variations depending on the modulator oscillation.
-

Visual Effects

- This can produce "wave" type effects, smooth deformations or distortions geometric depending on the source used.
- Ideal for adding textural disturbances into real-time visual compositions.

MODULATEREPEATY

In **Hydra Video Synth,** the modulateRepeatY function is used to manipulate a texture by repeating it along the Y (horizontal) axis and modulating that repetition based on another texture or source. This creates dynamic and complex visual effects, often used to generate repeating patterns with interesting distortions.

Syntax

modRepeatY(source, reps = 3, offset = 0.5)

1. **source**

   The texture or source used to modulate the repeat. This can be another video output or a generator in Hydra (like osc, shape, etc.).

2. **Reps**

   Controls the number of repetitions default to 3

3. **offset** *(default: 0.5)*

   Sets the modulation offset, influencing how the pattern is applied to texture.

Functioning

- **Horizontal Repeat :** modulateRepeatX starts by dividing the image or texture into multiple repeating sections along the Y axis.
- **Dynamic Modulation :** Then the specified source (first parameter) is used for distort or disrupt these repetitions according to its own variations.
-

Practical Example

Here is a simple example to visualize the effect of modulateRepeatX :

Machine Translated by Google

```
gradient(2).modulateRepeatY(gradient(), 30,0.9)
 .colorama(2)
 .saturate(20)
 .out()
```

- **gradient(2)** generates an oscillating texture.
- **modulateRepeatY(gradient(), 30, 0.9)** uses another oscillation (gradient()) to modulate the repetitions of the first texture.
- The result is a series of horizontal patterns, with dynamic variations depending on the modulator oscillation.

MODULATEKALEID

In Hydra, the modulateKaleid function is a visual modulation effect that applies a kaleidoscope transformation to a video stream or texture based on another stream texture used as a modulator. This effect creates symmetrical patterns by distorting the original image using kaleidoscope properties. This allows for adding elements of symmetry and dynamic variations in a visual composition.

Here is a detailed explanation of its settings and how it works:

Syntax

```
modKaleid(texture, nSides)
```

Settings

1. **input :** The source or texture used to modulate the kaleidoscope effect. This can be a texture generated in Hydra, a video or an image.
2. **nSides :** (Number of sides) Defines the number of symmetries in the kaleidoscope. Pa example :
   - ÿ 3 creates a triangular effect.
   - ÿ 6 generates a hexagonal effect.
   - ÿ A high value adds more symmetries.

Simple example of use

```
osc(10, 0.1, 1)
    .modulateKaleid(osc(5, 0.2, 1), 6)
    .out()
```

Code Analysis

1. **osc(10, 0.1, 1) :** Generates an oscillation texture.

Machine Translated by Google

2. modKaleid(osc(5, 0.2, 1), 6, 0.5) :

ÿ Use another oscillator (osc(5, 0.2, 1)) as modulation source.

ÿ Applies symmetry with 6 sides (nSides = 6).

3. .out() : Outputs the visual composition.

Effects and applications

- **Dynamic Pattern Creation :** Ideal for generating psychedelic animations.
- **Reactive Effects :** Can be combined with other sources (like audio) for synchronized effects.

- **Visual complexity :** By manipulating parameters dynamically (through functions or controllers), you can achieve captivating compositions.

Example with dynamic interaction

```
voronoi(10, 5)
    .modulateKaleid(osc(10, 0.2, 0.5), Math.sin(time) * 10 + 3)
    .out()
```

Here, the number of sides and the intensity change over time thanks to the sinu function

Conclusion

modulateKaleid is a powerful tool for adding symmetry and visual depth to Hydra compositions. By playing with parameters and using different input textures, it allows you to create a wide variety of unique effects.

MODULATESCROLLX(MODULATESCROLLY)

In **Hydra,** the modulateScrollX function is a visual effect that distorts a texture or visual source by applying modulation to horizontal scrolling **(ScrollX),** depending on of a texture or modulator source. This effect allows you to create fluid movements, dynamic distortions or wave effects in the horizontal axis.

Syntax

```
modScrollX(texture, scrollX = 0.5, speed)
```

Settings

1. **input :**

ÿ The source or texture used to modulate the scrolling

Machine Translated by Google

ÿ This can be a texture generated in Hydra (like osc, noise, etc.) or a video/image.

2. **scrollX :** Default modulation intensity at 0.5

3. **Speed:** scrolling speed

Basic example

```
osc(10, 0.1, 1) // Generates a visual oscillation.
    .modulateScrollX(osc(5, 0.2, 1), 0.5, 0.2) // Applies horizontal modulation.

    .out() // Sends the result to output.
```

Explanation

1. **osc(10, 0.1, 1) :** Oscillation texture with lines.
2. **modScrollX(osc(5, 0.2, 1), 0.5, 0.2) :**
   ÿ Use another oscillator as a modulator.
   ÿ Modulation intensity set to 0.5.
   ÿ Horizontal scrolling at 0 speed.

Dynamic effects and combinations

You can achieve captivating visual effects by combining modulateScrollX with other modulators or by changing the parameters dynamically:

Example with dynamic change:

```
gradient(0.2)
    .modulateScrollX(noise(3, 0.5), 0.8, Math.sin(time) * 0.3)
    .modulate(noise(2, 0.1), 0.2)
    .out()
```

Analysis :

- **Math.sin(time)** and **Math.cos(time)** allow you to vary the intensity and speed over time
- Result: a horizontal movement which fluctuates according to sinusoidal cycles

Machine Translated by Google

Creative applications

1. **Dynamic horizontal ripples :** Use a source like noise or gradient to add waves or distortions.
2. **Reactive effects :** Synchronize the scrollX or amount parameter with music or other signals.
3. **Abstract Transitions :** Combine with blend, rotate, or other modulations to achieve complex visuals.

Summary

modulateScrollX and therefore modulateScrollY is an essential feature to add subtle or dramatic horizontal movements in your Hydra compositions. By playing with the parameters and using different modulation sources, it allows you to generate fluid, hypnotic or chaotic animations according to your artistic needs

MODULATE

In **Hydra Video Synth,** the **modulate** function is used to modify or perturb a texture. or a visual signal based on another texture or signal. This creates dynamic and interactive effects, often used to generate complex and fluid visuals. It acts as a modulation, that is, a way of combining two inputs (a source and a modulating texture) to produce a unique visual result.

How modulate works

The **modulate** command works by overlaying or perturbing an input texture with a second texture. This can include translations, rotations, or deformations depending on the pixel values of the modulating texture.

The general syntax is as follows:

```
src(o0) // Original source
    .modulate(src(o1), amount, offset)
    .out(o0) // Visual output
```

Settings:

1. **texture :** The texture or signal used to modulate (e.g., another output, a form or video).

   Example: src(o1), shape(4), etc.

2. **amount :** A number (or function) between 0 and 1, which determines the intensity of the modulation. A value close to 1 means that the modulation will have a very visible effect while a value close to 0 attenuates its impact.

Machine Translated by Google

3. **offset** (optional): Offsets the texture used for modulation. This can add additional variations and enrich the visuals.

Example of use:

Basic example:

```
osc(10, 0.1, 0.8) // Input oscillator
    .modulate(osc(20, 0.2), 0.5) // Modulation with another oscillator

    .out(o0)
```

Here, an oscillator with a frequency of 20 modifies the base oscillator (frequency 10) with an intensity of 0.5.

Example with a shape:

```
shape(4, 0.5, 0.1) // A square .modulate(noise(3), 0.8) //
    Modulated by a noise .out(o0)
```

In this example, a square shape is disrupted by a noise signal, creating a fluid deformation effect.

Variations of modulate:

Hydra offers several modulate variants for specific uses

• **modulateScale :** Modifies the scale based on a texture • **modulateRotate :** Modifies the rotation based on a texture • **modulatePixelate :** Pixelates one texture based on another • **modulateHue :** Modifies the hue (color) based on another texture

Example with modulateScale:

```
osc(10, 0.1, 0.8)
    .modulateScale(osc(20, 0.2), 0.5) .out(o0)
```

This creates a zoom effect based on a second texture.

Summary :

The **modulate** function is a powerful tool for enriching your visuals in Hydra. By playing with textures, frequencies, and parameters, you can create dynamic and organic visual effects. It is a central element for those exploring generative visuals and wanting to add interactions between their sources.

Machine Translated by Google

MODULATESCALE

In **Hydra Video Synth,** the modulateScale function is used to modulate the scale (size or zoom) of one texture relative to another texture. This allows for dynamic visual effects where one texture affects the scale of another, often in interactive and generative compositions.

How **modulateScale works**

```
modulScale(texture, multiple = 1, offset = 0)
```

Settings

1. **texture :** The texture or source used to modulate the scale. This can be a flow video, wobble, noise or other shape generated in Hydra.
2. **multiple(amount)** (default: 1): The strength of the modulation applied. A value positive increases the effect, while a value close to zero reduces it.
3. **offset** (default: 0): An offset applied to the modulator texture, allowing the effect to be controlled at a more subtle level.

Result

The main texture is enlarged or reduced depending on the brightness or intensity values of the modulating texture. For example:

- Lighter areas of the modulating texture may cause zooming in.
- Darker areas may cause zooming out.

Example of use

Here is an example of a visual effect with modulateScale :

```
osc(10, 0.1,
    1) .modulateScale(noise(5),0.2,0.9) .out()x
```

In this example:

- A fast oscillation (osc(10, 0.1, 1)) is created as the main texture.
- A slower oscillation (osc(5)) modulates the scale of the first texture, generating a pulsating or vibrating effect.

Practical case

modulateScale is particularly useful for:

- Create organic animations where textures appear to “breathe” or “pulse.”
- Add visual complexity by combining dynamic sources.

Machine Translated by Google

- Generate effects reactive to music or other stimuli if Hydra is synchronized with external data.

You can experiment with more complex sources (like noise, shape, or videos) to see how they influence the modulation

MODULATEPIXELATE

You are right, the parameters of the modulatePixelate function in **Hydra Video Synth** are different from those mentioned earlier. Here is a corrected and detailed description of the function :

Function: modulatePixelate

The modulatePixelate function applies pixel modulation based on a source texture. It combines the values of the original texture and the source texture to produce an effect where the size and structure of the pixels change dynamically.

Syntax

.modulatePixelate(texture, multiple, offset)

Settings

1. **texture :**

    ÿ The texture used as modulation source. This can be a generated texture (like osc(), voronoi(), noise()) or a video/camera input (like s0, s1).

    ÿ This texture influences the distribution and dynamics of pixelation.

2. **multiple :**

    ÿ Sets the **multiplier factor** for pixel size

    ÿ The larger this value, the smaller the pixels become, producing an effect of increased density.

3. **offset :**

    ÿ Sets an offset applied to the pixelization effect.

    ÿ This allows to control the variation and distribution of the areas affected by the pixelation.

Machine Translated by Google

Simple usage example

```
osc(30, 0.1, 1)
    .modulatePixelate(noise(5), 100, 0.1)
    .out()
```

Decomposition:

- **Basic texture :** osc(30, 0.1, 1) generates an oscillating texture with a frequency of 30.
- **Modulation texture :** noise(5) creates a noisy texture used to modulate the pixelation.
- **Modulation parameters :**
    - ÿ multiple = 10 controls the size of the pixels, making them quite small.
    - ÿ offset = 0.1 adds a slight variation to the positioning of the pixelation.

Advanced example

```
  speed = 0.2
voronoi(15, 0.3, 0.1)
    .modulatePixelate(osc(10, 0.2, 0.8), 50, 0.2)
    .out()
```

Decomposition:

- **Base texture :** voronoi(15, 0.3, 0.5) creates a mosaic pattern based on a Voronoi diagram.
- **Modulation texture :** osc(10, 0.2, 0.8) is a slow oscillation used to modulate the size of pixels.
- **Modulation parameters :**
    - ÿ multiple = 50 creates pixels of medium size.
    - ÿ offset = 0.2 adds a subtle variation to make the effect more dynamic.

Creative applications

1. **Retro and pixelated aesthetics :** recreate visuals similar to old low-resolution screens. resolution.
2. **Dynamic visualizations :** combine modulatePixelate with textures interactive or reactive (e.g. audio or video).
3. **Structured chaos effect :** by modulating with textures like noise() or moving camera inputs.

Machine Translated by Google

MODULATEROTATE

You are right, in **Hydra Video Synth** the parameters of the modulateRotate function are **texture, multiple,** and **offset.** Here is a precise and complete description of this function.

Function: modulateRotate

The modulateRotate function applies dynamic rotation modulation to a texture, based on another source texture. This creates an effect where the rotation angle varies based on the values of the modulating texture, providing dynamic and often hypnotic visuals.

Syntax

```
.modulateRotate(texture, multiple, offset)
```

Settings

1. **texture :**

   ÿ The texture used to modulate the rotation.
   ÿ This can be a generated texture (osc(), noise(), voronoi()) or a external input (s0, s1).
   ÿ This texture determines how the rotation is modulated spatially and temporally.

2. **multiple :**

   ÿ Controls the **multiplication of rotation modulation.**
   ÿ The higher the value, the more the angular variations are amplified.
   ÿ Positive values change the rotation clockwise, and negative values negative counterclockwise.

3. **offset :**

   ÿ Adds a **constant offset** to the rotation angle.
   ÿ This offset influences the starting position or base of the rotation

Simple usage example

```
osc(30, 0.1, 1)
    .modulateRotate(noise(4), 1.5, 0.2)
    .out()
```

Decomposition:

• **Basic texture :** osc(30, 0.1, 1) generates an oscillating wave with a frequency of 30.

Machine Translated by Google

- **Modulation texture :** noise(4) provides a noisy texture to modulate the rotation.
- **Modulation parameters :**

    ÿ multiple = 1.5 amplifies the rotation by a moderate factor

    ÿ offset = 0.2 adds a slight base offset to the rotation.

Result: The oscillating wave is distorted by dynamic rotation influenced by noise, with a fluid and slightly offbeat movement

Advanced example

```
voronoi(20, 0.3, 0.5)
    .modulateRotate(osc(10, 0.2, 0.8), -2, 0.1)
    .out()
```

Decomposition:

- **Base texture :** voronoi(20, 0.3, 0.5) generates a mosaic pattern based on a Voronoi diagram.

- **Modulation texture :** osc(10, 0.2, 0.8) provides a slow oscillation used to modulate the rotation.

- **Modulation parameters :**

    ÿ multiple = -2 applies an amplified rotation in the counterclockwise direction

    ÿ offset = 0.1 adds a subtle offset for dynamic variations.

Result: The Voronoi pattern rotates with angles modulated by the oscillation, creating a hypnotic movement.

Creative applications

1. **Kaleidoscopic effect :** Combine modulateRotate with kaleid() to create rotating symmetrical patterns.
2. **Responsive Visualizations :** Use audio-reactive textures to modulate rotation in sync with the music.

3. **Smooth Distortions :** Combine complex textures like noise() or voronoi to achieve abstract and dynamic effects.

MODULATEHUE

In **Hydra,** the modulateHue function dynamically changes the hue of a texture o of a video stream based on the hue variations of another texture. It works by manipulating the hue channel in the HSV color model, creating sophisticated color shifting or mixing effects. Here's a more in-depth explanation, followed by specific examples.

Machine Translated by Google

Full syntax:

```
modHue(texture, amount, offset)
```

Detailed settings:

1. **source** *(texture) :*

   ÿ The texture or image used as a modulator.
   ÿ It influences how the hue of the main texture will be changed.

2. **amount** *(decimal number) :*

   ÿ The intensity of the modulation effect.
   ÿ A positive value strengthens the hue shift, while a negative value reverses the shifts.

3. **offset** *(decimal number, optional) :*

   ÿ Shifts the hue values of the main texture before applying modulation.
   ÿ This adds extra variation to enrich the visual effect.

Detailed operation:

modulateHue does not directly change the main texture, but overlays a modification based on hue variations in the modulator texture. Unlike other functions like modulate (which acts on brightness), modulateHue applies only to the hue channel, which preserves the original color intensity and saturation while creating a chromatic rotation effect.

Examples of use:

ALL EXAMPLES ARE TO BE REVIEWED!

1. Simple hue modulation:

```
osc(20, 0.1, 1) // A fast oscillating wave
    .modulateHue(osc(10, 0.05, 0), 0.5) // Modulation with a
slower wave
    .out()
```

• **Description :** The first wave is modulated by a second, slower wave. This creates an effect where hues shift and oscillate dynamically across the color spectrum.

2. Adding a hue offset :

```
gradient(1) // Generates a basic gradient
```

Machine Translated by Google

```
    .modulateHue(osc(10, 0.1, 0), 0.7, 0.5) // Modulation with
hue shift
    .out()
```

- **Description :** The offset of 0.5 adds a fixed offset to the hues before modulation, c which results in a constant rotation in the colors, in addition to the dynamic modulation effect.

3. Reverse modulation:

```
voronoi(10, 0.3, 2) // Voronoi Texture
    .modulateHue(osc(5, 0.1, 0), -1.0) // Modulation with inversion

    .out()
```

- **Description :** With a negative amount , the hues are inverted. The effect becomes more dramatic, and the hue variations follow an offset opposite to the modulator.

4. Superposition of multiple modulations:

```
osc(30, 0.05, 0.8)
    .modulateHue(gradient(2), 0.3) // First modulation with a gradient

    .modulateHue(osc(15, 0.2, 0), 0.5, 0.2) // Second
modulation with a wave
    .out()
```

- **Description :** By combining multiple modulations, complex and organic visual effects can be achieved. Here, a gradient and a wave together influence the texture main.

5. Modulation of a video source:

```
s0.initCam() // Initializes a camera as a source
src(s0)
    .modulateHue(osc(10, 0.1, 0), 0.4) // Wave-based modulation

    .out()
```

- **Description :** The image captured by the camera is modified in real time, with tints dynamic oscillating.

Tip: Combine with other functions

To enrich the visuals, you can use modulateHue in combination with:

- **blend :** Mix the modulated texture with another.

Machine Translated by Google

- **add :** Add an additional texture.
- **layer :** Superimpose several layers of effects.

Example :

```
osc(10, 0.2, 1).modulateHue(noise(3), 0.8)
    .layer(gradient(1).scale(0.5))
    .out()
```

Effects achieved with modulateHue :

1. **Smooth color transitions :** Ideal for fluid or hypnotic effects
2. **Dynamic Interactions :** Textures interact in real-time to create visuals complex.
3. **Chromatic Rhythms :** Perfect for syncing to music or inputs exterior.

If you have a specific goal for your visuals, please mention it: I can help you adjust the settings or suggest other variations.

EXTERNAL SOURCES

INITCAM

In **Hydra Video Synth,** the initCam() function is used to initialize and integrate a camera as a video source into your visual compositions. Here is a revised explanation, with examples consistent with existing functionality in Hydra.

How **initCam() works**

1. **Camera initialization :**

   When you call initCam(), Hydra attempts to access a camera connected to your computer, such as a built-in or external webcam.

2. **Creating a source :**

   Once initialized, the camera is associated with one of the video sources (e.g. s0, s1, etc.). You can use this source in your shaders to generate interactive visuals or apply effects.

Machine Translated by Google

Basic examples:

Live camera display

```
s0.initCam()
src(s0).out()
```

- Here, the live video from the camera (s0) is displayed without modification.

Applying simple visual effects:

You can manipulate the camera source to create more complex visuals:

```
s0.initCam()
// Add rotation and feedback
src(s0)
    .rotate(() => Math.sin(time) * 0.1) // Dynamic rotation
    .modulate(src(o0), 0.2).out()                              // Feedback
```

Advanced examples:

Mixing with an oscillator:

Combine the camera input with an oscillator for a dynamic effect:

```
s0.initCam()
osc(10, 0.1, 0.8) .mult(src(s0))              // Creating an oscillator
    of the camera                               // Multiplication with video

    .modulate(noise(3), 0.3) // Modulation with noise
    .out()
```

Important Notes:

- **Permissions :** Make sure your browser has the necessary permissions to access to the camera.
- **Technical limitations :** If multiple cameras are connected, Hydra uses the first detected camera by default.
- **Compatibility :** Some configurations or browsers (like Firefox) may have restrictions to access the camera.

Conclusion :

The initCam() function is a great tool for integrating live video streams into Hydra e manipulate them in real time, allowing the creation of interactive and immersive visual performances.

Machine Translated by Google

INITIMAGE/INITVIDEO

In **Hydra Video Synth,** the initImage() function is used to load an image as a visual source into your compositions. This allows you to integrate a static image as a texture or visual element into real-time generated graphical manipulations and effects.

How **initImage() and initVideo() work**

1. **Uploading an image/video :**
   You provide a path or URL to an image you want to use.
   Hydra loads this image and associates it with one of the video sources (s0, s1, etc.).
2. **Use of the source :**
   Once the image is loaded, it can be used in your shaders and manipulated with Hydra's transform, blend, and effects features.

Simple example:

Show an image

```
s0.initImage("path/to/image.jpg") // Replace with the path or URL of your image or video

src(s0).out()
```

• This loads and displays the image as it is

PLEASE NOTE THE RIGHTS ASSOCIATED WITH IMAGES OR VIDEOS, THEY MAY BE A SOURCE OF RESTRICTION IN THEIR IMPORTATION!!

PERSONALLY I PREFER:

https://commons.wikimedia.org/wiki/Main_Page TO IMPORT PHOTOS OR VIDEOS. YOU CAN ALSO UPLOAD YOUR OWN

IMAGES AND VIDEOS ON THE SITE FOR USE BY THE

CONTINUED ONLINE.

Example with manipulation:

Add transformations to the image:

You can apply effects or modify the loaded image to create dynamic visuals

```
s0.initImage("https://upload.wikimedia.org/wikipedia/commons/
0/02/1966_Buick_Skylark_convertible_%2814986026094%29.jpg")
// Apply rotation and modulator effects
```

Machine Translated by Google

```
src(s0)
    .rotate(() => Math.sin(time) * 0.2) // Dynamic rotation
time based
                                                        // Increased scale
    .scale(1.2) .modulate(osc(10, 0.1, 0.5)) oscillator  // Modulation with a

    .out()
```

Advanced example:

Mixing an image with other sources:

Combine the image with a camera or oscillator for a more complex composition:

```
s0.initImage("https://upload.wikimedia.org/wikipedia/commons/
9/99/Plymouth_Special_De_Luxe_4-Door_Touring_Sedan_1936.jpg")
s1.initCam()
// Mixing the image with the camera video
src(s0)
    .blend(src(s1), 0.5) // Blend 50% with camera
    .modulate(osc(15, 0.1), 0.3) // Modulation with an oscillator

    .out()
```

Possible uses:

1. **Visual Background :** Load an image as a base for animated effects.
2. **Textures and Masks :** Using an image to modulate or texture other sources.
3. **Dynamic Mixes :** Combine images with other video sources (camera, oscillators, noise, etc.).

Important Notes:

- **Path or URL :** The image must be accessible from the specified path or URL.
- **Format Compatibility :** Hydra supports common image formats like JPG and PNG.
- **Image Size :** Large images may be reduced or distorted depending on the transformations applied.

Conclusion :

The initImage()/initVideo function expands the creative possibilities in Hydra by integrating static images as visual sources. It is particularly useful for combining static visuals with real-time animations and effects, making your visual performances even richer and more diverse.

Machine Translated by Google

INITSCREEN

TO BE CHECKED!!

In **Hydra,** the initScreen() function initializes and configures the screen or canvas on which
The visuals generated by Hydra are displayed. This function is usually used at the beginning to
correctly configure the visual rendering environment. Here are its main roles

1. **Creating the rendering canvas :**(?)

   initScreen() creates the HTML canvas where Hydra will display the visuals. This can include managing canvas dimensions, placement on the page, and basic visual settings.

2. **Initializing rendering parameters :**

   The function configures the WebGL environment needed for rendering graphics
   This includes managing shaders and textures to generate the dynamic visuals.

3. **Connecting to the DOM** (?)

   If you're working in a browser or embedded interface, initScreen() associates the canvas with an existing HTML element (or creates a new one if necessary).

Example :

Display a source and transform it

```
//select a window//select a window s0.initScreen()

src(s0).pixelate(4)
.out()
```

SYNTH SETTINGS

RENDER

In **Hydra,** a library and coding environment for real-time visual creation, the render() function is used to **display the visual output of one channel in another**
channel, or to manipulate visual content by combining video streams or textures generated
by other parts of the scene.

How **render() works**

1. **Input source :**

   By default, Hydra has four main channels (or buffers): o0, o1, o2, and o3.
   These channels contain visual streams generated by the command chains

Machine Translated by Google

2. **Advanced manipulations :**
   By combining render() with other functions like src(), blend(), or modulate(), you can layer, create visual feedback effects, or mix different elements to enrich your visual.

Practical examples

1. Basic usage:

```
osc(10, 0.1, 1.5).out() // Generates an output on o0
render(o0)                                        // Redirect output to channel 1
```

2. Combine streams with render and src

```
osc(20, 0.05, 0.8)
        .out(o0)
noise(10)
        .out(o1)
src(o1)
        .blend(src(o0), 0.726)
        .out(o2)
src(o2)
        .mult(src(o2), 0.726)
 .pixelate()
        .out(o3)
render() //displays the 4 windows
```

3. Feedback effect:

```
  osc(30, 0.1, 1)
    .modulate(src(o0), 0.1) // Adds a modulation effect based on o0

    .out()

//render() to display on the entire screen
```

Why use render() ?

- **Flexibility :** This allows working with multiple visual layers without being limited to a single output.
- **Dynamic compositions :** Create complex effects or interactions between different channels.
- **Performance optimization :** By rendering a stream in another channel, you avoid recalculate some parts.

Machine Translated by Google

If you have a specific project or effect in mind, I can help you implement it in detail.
using render () in your context!

SERESOLUTION

Doesn't work on my Mac in chrome or Safari!

In **Hydra,** a coding platform for real-time visual creation, the **setResolution** function allows you to set the resolution of the visual output (the canvas or the rendering)
This controls the width and height of the rendering in pixels. Here's a detailed explanation:

Usefulness of setResolution

1. **Image quality control:**

   ÿ Higher resolution produces sharper, more detailed visuals.
   ÿ Lower resolution may give a pixelated or stylized effect, which can be used intentionally in artistic creations.
2. **Impact on performance:**

   ÿ Lower resolution reduces the load on the graphics processing unit (GPU), which is useful if your system has limited resources.
   ÿ Higher resolution may require more computing power, which could slow down rendering.
3. **Adaptation to the screen or presentation context:**

   ÿ You can adjust the resolution to match the dimensions of a specific screen or a particular configuration (such as projectors or monitors).
   LED).

setResolution syntax

setResolution(width, height)

- **width :** The width in pixels.
- **height :** The height in pixels.

For example :

```
setResolution(1920, 1080) // Sets the resolution to 1920x1080 (Full HD).

setResolution(800, 600) // Sets the resolution to 800x600.
```

Default behavior

- By default, Hydra uses the resolution of your screen or render window.

Machine Translated by Google

- If you do not set a resolution with setResolution, Hydra adjusts automatically render to window size.

Practical examples

1. Intentional pixelated effect:

setResolution(320, 240) // Low resolution for pixelated rendering. osc(10, 0.1, 0.8).out()

2. **Creating high-resolution visuals for a project:**

```javascript

setResolution(3840, 2160) // 4K resolution for highly detailed visuals.

voronoi(10).out()

## Advice

- Experiment with different resolutions to see how it affects the aesthetics and performances.
- Make sure to adapt the resolution to your hardware to avoid slowdowns useless.

Do you need help with a specific project with Hydra? 😊

## HUSH

In **Hydra,** a visual live coding platform, the hush() command is used to stop any rendering or visual output in progress. This is a handy command to "clean" the screen or interrupt visual effects without having to restart the program or manually delete lines of code.

### Main functionality of hush() :

- **Stop rendering:** As soon as the hush() command is called, Hydra stops immediately all active visual rendering streams. This includes oscillators, shaders or any other visual effect generated.
- **Cleanup Context:** This can be useful when you want to start with a clean base to test new code or remove visuals without leaving your session.

### Example of use:

```
osc(10, 0.1, 0.8).out()
// After seeing the visual in action, you want to stop it:

hush()
```
```

Machine Translated by Google

Practical use cases:

1. **Quick Interruption:** During a live performance, you might want to stop the temporary visuals without wasting time editing code.
2. **Experimentation:** When writing and testing new visuals, you can use hush() to stop what is being displayed before launching another visual. **Limitation:**

hush() does not remove any code or settings defined in your session. Any variable definitions, functions, or configurations remain active. For a complete restart, you will need to reset your environment manually.

SETFUNCTION

In **Hydra,** the setFunction() command allows you to define or redefine a custom function for the live coding environment. This is a way to modify or extend Hydra's existing functionality by adding custom behaviors or effects, directly accessible in your session.

CF HERE: https://hydra.ojack.xyz/docs/docs/learning/extending-hydra/glsl/

Main functionality of setFunction() :

- **Add custom functions:** You can define your own functions and use them like any other native Hydra command.
- **Redefine existing functions:** You can override the default behaviors of built-in functions if you want to customize their results or effects.

General syntax:

setFunction(name, func) • **name :**
The name you want to give to the new function or the one you want to redefined

- **func :** A JavaScript function that describes the desired behavior.

Example of use from the Hydra reference document:

```
// from https://www.shadertoy.com/view/XsfGzn

setFunction({name: 'chroma',

    type: 'color',

    inputs: [

        ],
```

Machine Translated by Google

```
                  `
    glsl:

       float maxrb = max( _c0.r, _c0.b );

       float k = clamp( (_c0.g-maxrb)*5.0, 0.0, 1.0 );

       float dg = _c0.g;

      _c0.g = min( _c0.g, maxrb*0.8 );

      _c0 += vec4(dg - _c0.g);

       return vec4(_c0.rgb, 1.0 - k);

`})

osc(60,0.1,1.5).chroma().out(o0)
```

Practical use cases:

1. **Code reuse:** If you frequently use a set of effects, you can group them into a custom function to simplify your session.
2. **Contextual Redefinition:** During a live performance, you can adjust or redefine functions to suit a specific mood or style without changing your entire code.
3. **Extending functionality:** Create non-native effects or behaviors in Hydra by leveraging JavaScript.

Limitation:

- Changes made with setFunction() are limited to the current session If you restart Hydra, the definitions will be lost unless they are saved. in a file or script

SPEED

In **Hydra,** speed() is a function that controls the **overall speed of the animation** in the system. It acts as a time multiplier, influencing all sources, oscillators, e other temporal modulators in your visual composition.

Use :

speed = x

- **factor :** A number (positive, negative, or 0) that determines the relative speed of the animation.

Machine Translated by Google

Effects of the factor parameter :

1. **Normal speed :**

   ÿ By default, the speed is set to 1.

   ÿ If you don't specify anything, the animations advance at their normal pace.

2. **Accelerate :**

   ÿ A value greater than 1 speeds up the animation.

   ÿ Example: speed = 2 // Doubles the speed.

   ÿ

3. **Slow down :**

   ÿ A value between 0 and 1 slows down the animation.

   ÿ Example: speed = 0.5 // Reduces the speed by half.

   ÿ

4. **Pause :**

   ÿ If you set the speed to 0, all animations freeze.

   ÿ Example: speed = 0

   ÿ

5. **Reverse the animation :**

   ÿ A negative value makes the animations play in reverse.

   ÿ Example: speed(-1) // The animation moves backwards at the speed normal.

   ÿ

Practical example:

Let's say you have an animation defined as follows

```
osc(10, 0.1, 1).out()
```

- To speed up the overall speed: speed = 2
-
- To reverse and slow down the animation: speed = -0.5
-

Important points:

- speed() affects **the entire project,** not just a particular source or string.
- If you want to change the speed of a specific animation, use parameters local modulators (as in osc(freq, speed, amp)).

In summary, speed() is a powerful tool for controlling the overall dynamics of your visuals, ideal for adjusting the mood or experimenting with temporal pacing in Hydra.

Machine Translated by Google

BPM

~~https://hydra.ojack.xyz/functions/#functions/bpm/0~~

The speed of all arrays in a sketch can be changed using the bpm parameter of hydra synth.

```
bpm = 60
osc(60,0.1,[0,1.5]).out(o0)
```

WIDTH/HEIGHT

Examples from Hydra documentation. ChatGPT gave us a totally wrong answer!

Scroll widthwise:

```
shape(99).scrollX(() => -mouse.x / width).out(o0)
```

Scroll in height direction:

```
shape(99).scrollY(() => -mouse.y / height).out(o0)
```

TIME

Role of time in Hydra

- **time** is a dynamic variable that represents the elapsed time (in seconds) since the launching the program or rendering graphics.
- It is continuously updated and can be used to introduce variations or animations in the visuals.

Explanation of the example

Here is a breakdown of the line of code:

```
shape(2, 0.8)
    .kaleid(() => 6 + Math.sin(time) * 4)
    .out(o0)
```

1. **shape(2, 0.8) :**

   ÿ Creates a geometric shape, here a **polygon** with 2 sides (therefore a straight line).
   ÿ The second parameter 0.8 controls the "sharpness" of the edges of the shape.
2. .kaleid(() **=> 6 + Math.sin(time) * 4) :**

Machine Translated by Google

- ÿ Applies a **kaleidoscope** effect to the shape.
- ÿ The function passed to kaleid dynamically determines the number of "segments" (symmetries) in the kaleidoscope.
- ÿ **Math.sin(time)** oscillates between -1 and 1 over time, creating a motion fluid and cyclical
- ÿ **6 + Math.sin(time) * 4** dynamically adjusts the number of segments, ranging from 2 (6 - 4) to 10 (6 + 4).

3. .out(o0) :

- ÿ Sends the final rendering to output channel o0, which displays the animation on the screen

Expected visual result

- A shape (here a straight line) is transformed into a kaleidoscopic pattern whose number of segments changes fluidly and continuously to the rhythm of the sinusoidal function

Why use time ?

- The time variable allows you to create a dynamic and organic animation that evolves without manual intervention.
- Without time, the number of segments in the kaleidoscope would remain fixed, and the animation would be missing. of fluidity

If you need additional details or other examples, please don't hesitate to ask! 😊

MOUSE

From the hydra documentation:
https://hydra.ojack.xyz/functions/#functions/mouse/0

Example:

```
shape(99).scroll(
    () => -mouse.x / width,
    () => -mouse.y / height)
    .out(o0)
```

ARRAY

Fast

Example:

```
shape([3, 6, 9].fast(2)).out()
```

This is explained by the structure of Hydra where **fast() acts directly on an array ([3, 6, 9]),** and not as an independent method of shape(). Here is a more detailed explanation:

Machine Translated by Google

Decrypting the code:

```
shape([3, 6, 9].fast(2)).out()
```

1. **[3, 6, 9] :**

   ÿ An array containing the values that will be used for the parameters of the shape() function.

   ÿ Here, the values correspond to the number of sides of the shapes: a triangle (3), a hexagon (6), and a nonagon (9).

2. .fast(2) :

   ÿ Applied to the array [3, 6, 9], this method causes Hydra to cycle through the array values at **twice the normal speed.**

3. **shape() :**

   ÿ Uses the array to dynamically alternate between values, creating shapes with different numbers of sides.

4. **out() :**

   ÿ Displays the result of this composition on the screen

What appears visually

- A rapid alternation between a triangle, a hexagon and a nonagon.
- If you increase or decrease the value of .fast(), it will change the speed at which these shapes alternate.

Advanced example: combination with other transformations

You can enrich this effect by adding rotations or modulations:

```
shape([3, 6, 9].fast(4))
    .rotate(0.1)
    .modulate(osc(5, 0.1))
    .out()
```

Explanation :

1. **[3, 6, 9].fast(4) :**

   ÿ Transitioning between shapes is even faster thanks to .fast(4).

2. **rotate(0.1) :**

   ÿ Adds a constant rotation to the shape.

3. **modulate(osc(5, 0.1)) :**

ÿ The oscillator provides dynamic modulation to the composition.

SMOOTH

In **Hydra,** the smooth() function is used to **smooth out transitions** between values, creating smoother, more gradual visual effects. It is particularly useful when working with
with modulations or tables to avoid sudden jumps or abrupt changes in an animation.

How **smooth() works**

- **Temporal smoothing:** The smooth() method acts as a filter that applies a linear (or similar) interpolation between values, slowing down transitions to make them smoother

- **Application :**

  ÿ It is often used in conjunction with arrays (e.g., [value1, value2].smooth()).

  ÿ Can be combined with oscillators or visual sources to produce smoother animations.

Basic example with a table

```
shape([3, 6, 9].smooth(0.5)).out()
```

Explanation :

1. **[3, 6, 9] :**

   ÿ An array containing values that dynamically change the number of sides of the form.

2. .smooth(0.5) :

   ÿ Applies a smoothing to the transitions between values in the table. The value 0.5 controls the degree of softening:

   ÿ A lower value (e.g. 0.1) makes transitions faster.

   ÿ A higher value (eg 1) makes them slower and more gradual.

3. **shape() :**

   ÿ Use these values to determine the number of sides of the shape (triangle, hexagon, nonagon), but the transitions between them are now smooth.

4. **out() :**

   ÿ Display the result on the screen

Machine Translated by Google

Combined example with oscillators

```
osc(10, 0.1, 1)
    .modulate(shape([3, 6, 9].smooth(0.3)))
    .out()
```

Explanation :

- **[3, 6, 9].smooth(0.3) :**

  ÿ Smooths transitions between different shapes to avoid abrupt changes in modulation.

- **Visual result:**

  ÿ The oscillations of osc() are modulated by shapes that evolve in a fluid manner, creating an organic visual effect.

Important points about smooth() :

1. **Controlled softening:**

   ÿ The value passed to smooth() controls the intensity of the effect. Too high a value can slow down transitions to the point of being imperceptible.
2. **Creative applications:**

   ÿ Improves visual aesthetics by removing choppy transitions.
   ÿ Ideal for music-synchronized animations or for meditative and immersive effects.

3. **Combinations:**

   ÿ Combine smooth() with modulators, oscillators or transforms for complex and smooth animations

EASE

In Hydra, the ease operator is not used in isolation, but is actually part of an array in specific contexts. This is because it is typically used in
association with other dynamic parameters to modulate complex transitions or interpolations.

Here are some explanations and examples to clarify the use of ease as a part of a
painting :

Working with tables

Machine Translated by Google

Hydra allows the use of **dynamic arrays of values** to create modulated behaviors. When ease is used in an array, it acts on a set of values and applies interpolated transitions for each of them.

Example of use in a table

1. **Interpolate multiple values of a color :**

```
2.
   osc(10, 0.1, 1)
3.    .colorama([0.1, 0.5, 0.9].ease('easeInOutCubic'))
4.    .out()
```

ÿ Here, an array [0.1, 0.5, 0.9] is interpolated using ease, making the transitions between colorama values smoother over time.
'easeInOutCubic' refers to the mathematical function that performs interpolations between values. For a detailed list of these 'ease' functions, see
ÿ specifying the rate of change of a parameter over time cf: https://easings.

5. **Complex animation with a dynamic table :**

```
6.
   shape(4)
7.    .scale([0.5, 1, 1.5].ease('easeInOutQuint'))
8.    .out()
```

ÿ In this example, the shape size gradually changes between the values 0.5, 1, and 1.5 using ease, creating a cyclic animation.

Important Notes

- **Correct syntax :** Make sure to use .ease() with an array to benefit from its interpolation properties.
- **Dynamic Visual Effects :** By combining ease with other parameters like modulate, color, or scale, you can produce very complex and organic visuals.

Machine Translated by Google

OFFSET

From the docs: https://hydra.ojack.xyz/docs/docs/learning/sequencing-and-interactivity/arrays/

Another one of the methods Hydra adds to Arrays, allows you to offset the timing at which Hydra will switch from one element of the Array to the next one. The .offset method takes a Number from 0 to 1.

Another of the methods that Hydra adds to arrays allows you to offset the time at which Hydra will move from one element of the array to the next. The .offset method takes a number between 0 and 1.

Example:

```
noise([2, 10, 30].offset([1, 0.25, 0.5]))
 .color(1,0,3)
     .out()
```

Each value in the array creates a unique offset on the pattern in time and space as in the example below:

```
shape(4)
    .scale(0.5)
    .repeat([2, 3, 4].offset([0.2, 0.4, 0.6])) .out()
```

FIT

You are absolutely right, and I apologize for the previous mistake. In **Hydra Video Synth,** the .fit() method applied to an **Array** is used to **resize the values of an array to fit within a specified range.** This allows for manipulating data p

make them usable in Hydra functions.

How .fit() **works :**

The .fit() method takes two arguments, **min** and **max,** and maps the values in the existing array to that range.

This is useful, for example, when working with randomly generated data or with values that do not fit the scale needed for a specific effect.

Machine Translated by Google

General syntax:

```
array.fit(min, max)
```

Example of use:

Let's say you have an array with values from 0 to 10, but you want to fit them to a range from 0 to 1 (more useful in Hydra).

```
[0, 2, 5, 10].fit(0, 1)
// Result: [0, 0.2, 0.5, 1]
```

This converts the initial values into a range proportional to the new specified range.

Practical example in Hydra inspired by:

https://hydra.ojack.xyz/?code=YnBtJTIwJTNEJTIwMTIwJTBBYXJyJTIwJTNEJTIwKCklM0QlM0UlMjAlNUIxJTJDMiUyQzQlMkM4JTJDMTYlMkMzMiUyQzY0JTJDMTI4JTJDMjU2JTJDNTEyJTVEJTBBb3NjKDUwJTJDLjElMkNhcnIoKS5maXQoMCUyQ01hdGguUEkpKSUwQSUwOS5zY2FsZShhcnIoKS5maXQoMSUyQzIpKSUwQSUwOS5vdXQoKQ==

```
bpm = 120 arr
= ()=> [1,2,4,8,16,32,64,128,256,512]
osc(50,.1,arr().fit(0,100)) .scale(arr().fit(1
        ,10)) .out()
```

Why it's useful:

- **Data adaptation :** Allows data to be made usable for functions Hydra like modulate, osc, or scale.
- **Creativity :** Gives more control over the scale and scope of dynamically generated visual effects.

FFT/SETSMOOTH/SETCUTOFF/SETBINS/SETSCALE

https://hydra.ojack.xyz/docs/docs/learning/sequencing-and-interactivity/audio/

Audio reactivity # FFT

functionality is available via an audio object accessible via 'a'. The editor uses https://github.com/meyda/meyda for audio analysis. To view the ff bins

```
a.show()
```

Machine Translated by Google

Set the number of fft bins

```
a.setBins(6)
```

Access the value of the leftmost bin (lowest frequency):

```
a.fft[0]
```

Use value to control a variable:

```
osc(10, 0, () => a.fft[0]*4).out()
```

It is possible to calibrate the reactivity by changing the minimum and maximum value detected (Represented by blurred lines on the fft). To set the minimum value detected

```
a.setCutoff(4)
```

Adjusting the scale changes the detected range

```
a.setScale(2)
```

The fft[] will return a value between 0 and 1, where 0 represents the cutoff and 1 represents the maximum.

You can set a smoothing between audio level readings (values between 0 and 1). is no smoothing (more unstable, faster reaction time), while 1 means the value will never change.

```
a.setSmooth(0.8)
```

To hide the audio waveform:

```
a.hide()
a.setBins(5) // number of bins (bands) to separate the audio spectrum
```

```
noise(2) .modulate(o0,()=>a.fft[1]*.5) // listen to the 2nd band .out()
```

```
a.setSmooth(.8) // smooth audio responsiveness from 0 to 1, uses linear interpolation
a.setScale(8) // upper limit of sound volume (corresponds to 0)
a.setCutoff(0.1) // sound volume at which to start listening (corresponds to 0) a.show() // show
what Hyd is listening to // a.hide()
```

```
render(o0)
```

Machine Translated by Google

Machine Translated by Google