Laboratório VISGRAF
Instituto de Matemática Pura e Aplicada

Creating Software for Interaction and Participation: A Documentation for Videolab, Klak, and MIDI in Unity

Santiago Guisasola, Luiz Velho (supervisor)

Technical Report TR-20-04 Relatório Técnico

February - 2020 - Fevereiro

Creating Interactive Software and Multimedia: A Documentation for Videolab, Klak, and MIDI in Unity v1.0

Santiago Guisasola

December 28, 2020

Contents

Acknowledgements

I would like to express my sincerest gratitude for the opportunity to work with the Vision and Graphics Laboratory at the Instituto Nacional de Matemática Pura e Aplicada. I am especially grateful to Dr. Luiz Velho for his support and guidance.

Chapter 1

Introduction

Unity is first and foremost a game engine: a software-development environment initially intended for the creation of video games.[1] It is a computational platform for designing technologies for interaction between human and human, human and computer, and even computer and computer.[2] Games can be predominantly visually-based (the music and sound accompany the graphics), predominantly sound-based (the graphics accompany the interactions with sound and music), or a mixture of the two. In other words, the user can either be thinking predominantly visually or sonically, or, in varying degrees, both, in the way that they embody or insert themselves into the game.

Human input is traditionally captured through a gamepad, joystick, or keyboard and mouse, and controls aspects, which are chosen by the game creators, of a multimedia scene (computer graphics / visual art, and sound). This control is achieved by using scripts to tie user input meaningfully to objects in a Unity scene. Additionally, with scripts, one can use parameters, properties, and statuses of objects to perform computations that generate anything from movement to deformations, and beyond. In Unity, scripts are short programs written in C#. Normally, user input and self-feedback within the game are harnessed together in creating the overall game and defining its dynamics.

From the Unity manual [7]:

> "Scripting is an essential ingredient in all applications you make in Unity. Most applications need scripts to respond to input from the player and to arrange for events in the gameplay to happen when they should. Beyond that, scripts can be used to create graphical effects, control the physical behaviour of objects or even implement a custom AI system for characters in the game."

In other words, scripts effect change in the game by taking user input or bypassing the human component entirely by capitalizing on self-feedback possibilities within the game. To fully take advantage of Unity's immense space of possibilities, scripting is a necessity.

The way Unity works can be thought of as follows. When working with 3D scenes in Unity, one is able to fill a vast 3D space with objects, lights, cameras, sound sources, and more.[3] Scripts inform Unity how these objects will behave as functions of each other, time, and user input. The location and behavior of all objects

[1] This documentation's attention is on Unity's capabilities of creating interactive multimedia environments that go beyond the traditional notion of a video game, but these environments will still be referred to as "games."

[2] Human input is not required in Unity, and entirely self-dependent games are possible. In these cases, the result can be thought of as a film that is largely driven through computation, and that, with the computational power afforded by Unity, can have procedural effects among other computational features.

[3] Unity also handles 2D games. The details of which, unfortunately, are beyond the scope of this project.

together determine which camera view within the game will be captured, rendered, and displayed when the game runs.

And that is what Unity does: a single still image (a *frame*) is created, wherein Unity takes the game's state in the frame and uses it to compute how the objects will move, ever so slightly, in the 3D space (according to user input and scripts, for instance), creating the next still image. Unity repeatedly generates and displays these frames at a rate that is known as the frame rate, which is measured in frames per second (fps). The standard for movies and television is 24 fps, although higher numbers are possible, which are particularly useful for sports (usually 30 fps) and slow-motion captures (e.g., 960 fps).

In short, a scene contains objects and scripts that breathe life into the objects. The scripts can also connect the objects with user input, which is traditionally captured using joysticks, gamepads, and the computer keyboard and mouse. These mechanisms make Unity a versatile platform for creating interactive multimedia that transcend traditional notions of video games.

Unity's wide-ranging interactive multimedia capabilities inspire questioning traditional forms of user input. If we are breaking free from video games as they stand, why not break free from the standard of joysticks, gamepads, and mouse and keyboard? A great feature of Unity is its extendibility: users can create *packages* that amplify the engine's space of possibilities. It is through packages that new forms of user input are found. Additionally, beyond user input, there are packages that redesign Unity's scripting environment. One such package, Klak [2], embodies Unity's scripting capabilities in the form of a visual programming environment.

Tradition in user input involves using joysticks, gamepads, or the computer keyboard and mouse, and these possibilities come pre-packaged with Unity. Thinking outside of the video game world, however, and inside the world of multimedia, the possibility of using MIDI for user input becomes an immediate desire.[4] Unity does not have inherent capabilities to work with MIDI input (or to generate MIDI output) but packages exist giving the engine these powers. Namely, the package MIDIJack, developed by Keijiro Takahashi [2], must be installed if it is desired to integrate Unity with MIDI communication. Alternatively, there are packages that encompass MIDIJack and other embellishing upgrades within a larger framework.[5]

The indispensability of scripting in using Unity to its full potential may be a deterrent to those who would like to participate in the creation of games and interactive media, but have little to no programming experience. The good news is that, as mentioned earlier, there is a Unity package that reshapes scripting in Unity. The visual nature of Klak allows, if desired, entirely bypassing the programming of scripts. Nevertheless, Klak teaches and uses the logic of scripting and computer programming (it has to, that is still what it is!) and can be a stepping stone to learning programming to those who are more visually inclined and who want to learn to program by creating interactive visual and sonic art. For more advanced users, Klak and scripts can be used in unison.[6]

With Klak, *patches* are made where one connects nodes (simple computational blackboxes that capture many of Unity's C# scripting capabilities) to each other in clever ways to produce desired computations using user input and endogenous properties. In other words, rather than having multiple scripts communicate with each other using global variables, Klak patches are visually connected to each other via *wires* in the same *patcher*

[4] Briefly, MIDI (Musical Instrument Digital Interface) is a communication protocol originally designed for digital music. MIDI does not create sound but instead relays messages containing information about the note played, how strong it was played, and more.

[5] One such package, Videolab, is the focus of this documentation, and will soon be discussed.

[6] In fact, the Klak nodes covered in this document can be altered and extended by changing their underlying scripts.

environment.

Both MIDIJack and Klak can be installed in Unity on their own, or as part of the Videolab package [20]. The Videolab package, which includes both Klak and MIDIJack, was created by Keijiro Takahashi and the consumer electronics company Teenage Engineering to supplement the capabilities of the OP-Z, an advanced sequencer and synthesizer created and manufactured by Teenage Engineering. However, **it is not required to own the OP-Z in order to use Videolab.**[7]

Videolab was gifted to the world on October 17, 2018.[8] Since then, many enthusiasts have joined the community, as evidenced by the numerous online groups and discussions revolving around the OP-Z and Videolab, and the YouTube tutorials and explorations. There is a common theme behind many comments: the people want more. Two things are clear:

1. Videolab is revolutionary.
2. There is widespread confusion on the details of Videolab and much of its possibilities remain a mystery for many participants.

The intention of this documentation is to directly address item (2) above. This is done by summarizing the essence of the Klak nodes, providing examples of how to use the nodes, and offering short tutorials on some of Unity's features that go hand-in-hand with Klak programming and a simple example of integration between Unity, Videolab, and Max MSP / Pure Data.[9] As mentioned, some tutorials already exist; this documentation makes an effort to refer to them when appropriate. In addition, several post-processing image effects that work very smoothly with Klak, called "Kino Image Effects," are included in Videolab and are briefly explained in the appendix. This documentation is not meant to be exhaustive, but should be enough to get an enthusiast started.

A reader who goes through the entire documentation in a linear fashion might encounter significant repetition (e.g., many properties and features of Klak nodes are present in several different nodes, as such they have almost or fully identical explanations and definitions). This was done intentionally for the readers who do not go through the documentation linearly, so that each node's explanation is self-contained, and so that the documentation can be used as a reference and not a book.

An additional intention (and hope) is that this documentation serves as an open-source starting point for the community to focus their efforts and collectively gather appropriate information, document it, share their wisdom, and enhance the overall understanding of Klak and Videolab. As such, this documentation is a call for communal collaboration. There may be errors and misinterpretations (and typos) present in the documentation, and acknowledgment of this is crucial so that the reader is open to discovering alternative and / or additional possibilities for the nodes and their properties. Hopefully, upon doing so, they will update the documentation accordingly. After this documentation is finalized as a document, it will be transferred to an online wiki platform, or similar online communal environment.

[7] The author of this documentation, for instance, does not own the OP-Z.

[8] This is the official public release date of Videolab.

[9] Max MSP and Pure Data can serve as intermediate steps that manipulate data coming from other platforms, such as Wii controllers and Mocap, translate these data into MIDI, and channel it into Unity for desired control of the game. Unfortunately, the details of these translations are beyond the scope of this documentation, and only a simple example of the use of these softwares is provided at the end of the documentation.

Above all, the main intention of this project is to increase the number of people participating in the creation of these technologies, and to increase the overall number of projects in the field. To put it another way, the goal is such that a person who has never programmed before, never used Unity, and maybe never played with a MIDI controller or software platform, will be able to create interactive audiovisual art and participate in this growing community, or at least get started.

It is worth noting that this documentation is also useful for those who have no interest in working with MIDI. This is because traditional means of capturing user input can be used, such as with a mouse and keyboard, a joystick, a gamepad, or even smartphones and tablets. Additionally, as stated earlier, it is entirely possible to create Unity games that take in zero user input. However, hopefully this documentation makes it clear, if it is not already, that MIDI provides an incredible enhancement to Unity.

Chapter 2

Terminology

Words and phrases that are commonly used throughout this document are defined and explained in this chapter.

Bang – A bang is a message in Klak akin to its appearance in other visual programming environments such as Max MSP and Pure Data. From the Max MSP manual [3]: "[The bang is] the message that tells many objects to *do that thing you do*. As a result, sending the bang message to other objects will normally cause them to send messages from their outlets." Note: what the Max MSP manual refers to as "object" we refer to as "node" in the context of Klak.

Child (and Parent) Objects – From the Unity manual [9]: "Unity uses a concept called Parenting. When you create a group of GameObjects, the topmost GameObject or Scene is called the 'parent GameObject', and all GameObjects grouped underneath it are called 'child GameObjects' or 'children'. [...] To make any GameObject the 'child' of another, drag and drop the intended child GameObject onto the intended parent GameObject in the Hierarchy."

DAW – A digital audio workstation (DAW) is a software for producing music and working with MIDI. Common DAWs include Garageband, Ableton Live, Pro Tools, Fruity Loops, and Cubase.

Euler Rotations (and Euler Angles) – Euler angles are a set of three values representing distinct angles in 3D space that describe the orientation of an object. An Euler rotation uses these angles to define a rotation of the object, and can be used in place of quaternions. Euler rotations are susceptible to gimbal lock, which is the loss of a degree of freedom in rotations due to the aligning (or "locking") of two of the variables used to describe the rotation. (Note: quaternions are not susceptible to gimbal lock.)

Float – A number with a decimal place.

GameObjects – From the Unity documentation [8]: "The GameObject is the most important concept in the Unity Editor. Every object in your game is a GameObject. This means that everything you can think of to be in your game has to be a GameObject. However, a GameObject can't do anything on its own; you need to give it properties before it can become a character, an environment, or a special effect. [...] Depending on what kind of object you want to create, you add different combinations of components to a GameObject. You can think of a GameObject as an empty cooking pot, and components as different ingredients that make up your recipe of gameplay."

Hierarchy – From the Unity manual [9]: "The Hierarchy window contains a list of every GameObject in the current Scene. [...] When you add or remove GameObjects the Scene (or when your gameplay mechanic adds and removes them), they appear and disappear from the Hierarchy as well. By default, the Hierarchy window lists GameObjects by order of creation, with the most recently created GameObjects at the the bottom. You can re-order the GameObjects by dragging them up or down, or by making them 'child' or 'parent' GameObjects."

Inlet – We refer to the left-hand input connections of Klak nodes as inlets.

Int – An integer, i.e., a number without a decimal place.

Interpolator – Many Klak nodes have the option of using an interpolator, which is a method for transitioning between two values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed. For example, the note off value of a particular MIDI note may be set to 0, and the note on value set to 1. If the interpolator is set to Direct, once the note is played, the value being sent out of the node will change directly and immediately from 0 to 1. On the other hand, if the interpolator is set to Exponential or Damped Spring, then, depending on its settings, the value will smoothly go from 0 to 1.

Klak Patches – The space where Klak nodes and wires are created.

List – All Klak nodes with outlets have an accompanying list that summarizes which other nodes these outlets connect to. It is possible to create connections between nodes (and even GameObjects) without wires through the list.

Max MSP – A visual programming environment for music and multimedia installations.

MIDI – Musical Instrument Digital Interface (MIDI) is a standard communication protocol originally developed for digital music, but that throughout its history has made its way to other types of digital communication (e.g., controlling lights).

MIDI Controller – A digital instrument that sends (and sometimes receives) MIDI information. A MIDI controller is usually connected to a DAW. A theme of this documentation is using MIDI to control Unity.

MIDI Note Number – A numerical representation of musical notes used by MIDI. For reference: Middle C (or C4) has MIDI note number 60. These values range from 0 to 127, although the lowest note on the piano (A0) has a MIDI note number of 21 (which has frequency 27.50Hz, close to the lower limit of human hearing, roughly 20Hz). Oftentimes, MIDI note values 0 - 20 are saved for other types of messages (e.g., percussive sounds).

MIDI Source Object – A MIDI Source object is created via the Unity menu bar, by first clicking GameObject, then MIDIJack, then MIDI Source. These are useful when there are more than one MIDI source that Unity is listening to.

MIDI Velocity – The force used to play a MIDI note, ranging from 0 - 127, where 0 represents no velocity (and thus no discernible sound) and 127 represents full force.

Nodes – We refer to the individual computational blackboxes in Klak as nodes.

Object Component – From the Unity documentation [16]: "Components are the nuts & bolts of objects and behaviors in a game. They are the functional pieces of every GameObject." Many GameObject's come prepackaged with components and all have the ability of being augmented by more components.

Object Hierarchy – see Hierarchy.

Outlet – We refer to the right-hand output connections of Klak nodes as outlets.

Parent (and Child) Objects – See Child (and Parent) Objects.

Patcher – see Klak Patches.

Pure Data – see Max MSP.

Quaternion - From the Unity manual [13]: "Quaternions are used to represent rotations. They are compact, don't suffer from gimbal lock and can easily be interpolated. Unity internally uses Quaternions to represent all rotations. They are based on complex numbers and are not easy to understand intuitively." In other words, don't worry about the details of quaternions, and trust Unity to handle them as rotations. Usually, you will be able to think about and work with more intuitive representations of rotations, and Unity will take care of working with their quaternion representations.

Scene - Scenes are the "vast 3D spaces" that can be filled with objects, as mentioned in the introduction of the documentation. From the Unity manual [14]: "Scenes contain the environments and menus of your game. Think of each unique Scene file as a unique level. In each Scene, you place your environments, obstacles, and decorations, essentially designing and building your game in pieces."

Shaders - From the Unity manual [10]: "Shaders are small scripts that contain the mathematical calculations and algorithms for calculating the Color of each pixel rendered, based on the lighting input and the Material configuration."

String - From the Unity documentation [15]: "[A string] represents text as a series of Unicode characters."

Vector3 - From the Unity documentation [17]: "[A Vector 3 is a] representation of 3D vectors and points. This structure is used throughout Unity to pass 3D positions and directions around. It also contains functions for doing common vector operations."

Wires - In this documentation we refer to the connections between Klak nodes as wires.

Chapter 3

Getting Started

This chapter outlines important information to get started with Unity, Klak, and Videolab. The first section details possible mishaps and initial confusions. Then, given the numerous possible configurations of the Unity workstation, an example of how to organize the workstation is provided. Finally, the chapter closes with a brief overview of the classes of Klak nodes and their functions.

3.1 Some Words of Warning and Advice

- **Naming the Klak nodes.**

Each Klak node added to the patcher comes with a name slot. For example, the `Note Input` node is initially simply called *Node Input*. If several of these are created and assigned to listen to different notes from the MIDI device, the patch can become very confusing very quickly because all of these nodes will be named the same thing: *Node Input*. By renaming these nodes, the patch becomes organized and easier to build upon and edit. For example, the specific `Note Input` node listening for the note C (in any octave) can be renamed *Note Input: C*.

- **Downloading the correct version of Unity.**

Unity is updated frequently and there is not always guaranteed compatibility across versions. In fact, because of this, developers often have many versions of Unity installed on their computer, which are managed through the Unity Hub. With the Unity Hub, specific versions of Unity can be opened for specific projects. In the case of Videolab, the best version to use (the one it was developed for) is version 2018.2. If Videolab is being used to build *videopaks* for the OP-Z, then using Unity version 2018.2 is necessary, since it is the only version that is compatible with version 1.1 of the OP-Z.

- **Selecting a GameObject.**

There are many situations where it is necessary to select a GameObject for a node's function and for other features, such as selecting the material of a GameObject. In such situations, in the node's inspector there is an empty field in which the user must select a GameObject. There are two ways of doing so. The first is to simply drag the GameObject into the empty field. The other is to click on the little circle next to the empty field and select the desired GameObject from the list of objects.

- **For a solid background color.**

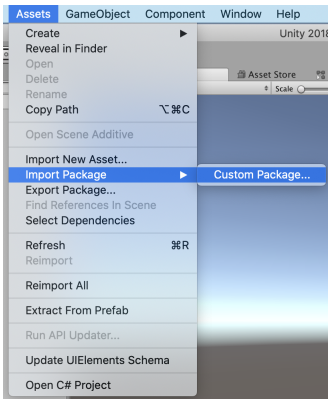
Select the Main Camera from the object hierarchy. In the camera's inspector, use the dropdown menu titled Clear Flags and select Solid Color.

- **Making changes while the scene is in play mode.**

Be careful. A common mistake, with sometimes devastating consequences (e.g., losing crucial adjustments) is making changes while the scene is in play mode. A way to think of play mode is that it is a test or debugging mode. Everything created can be tested, and during the testing, properties can be changed so that the user can observe in real time how certain changes will affect their game. **Changes made during play mode that the user would like to preserve should be copied, written down, or somehow recorded.** When play mode is ended, most changes made will be reversed. Be mindful of this.

- **Installing Videolab.**

Installing Videolab is simple. Download the latest version of the Videolab package on the Teenage Engineering GitHub page. From a Unity project, go to Assets on the Unity toolbar menu, then select Custom Package from Import Package. Simply select the Videolab package and load it into Unity.[1]

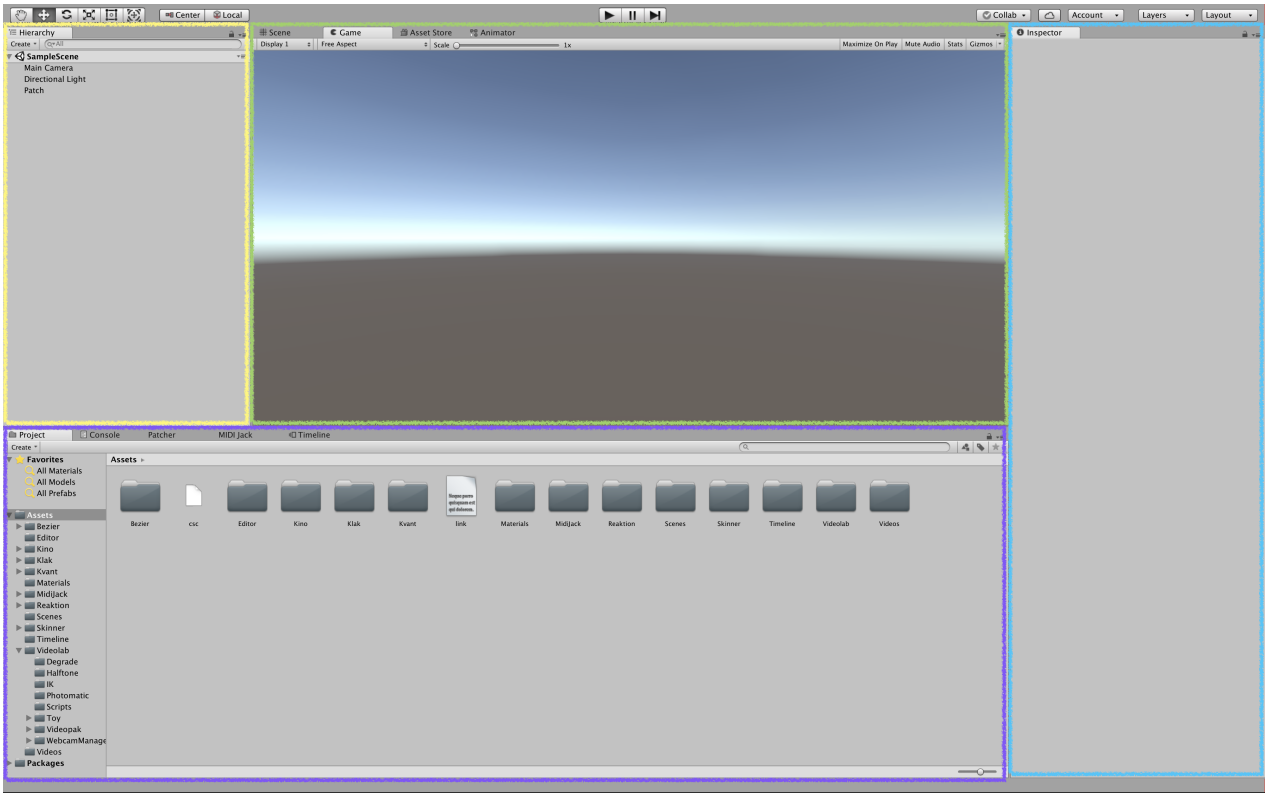


3.2 The Unity Workstation

There are many ways to organize the Unity workstation. On the top-right corner, there is a dropdown menu labeled *Layout* with options including *2 by 3, 4 Split, Default,* among others. The setup of the workstation can be adjusted as the user sees fit. The author uses the default workspace with additional commonly used windows.

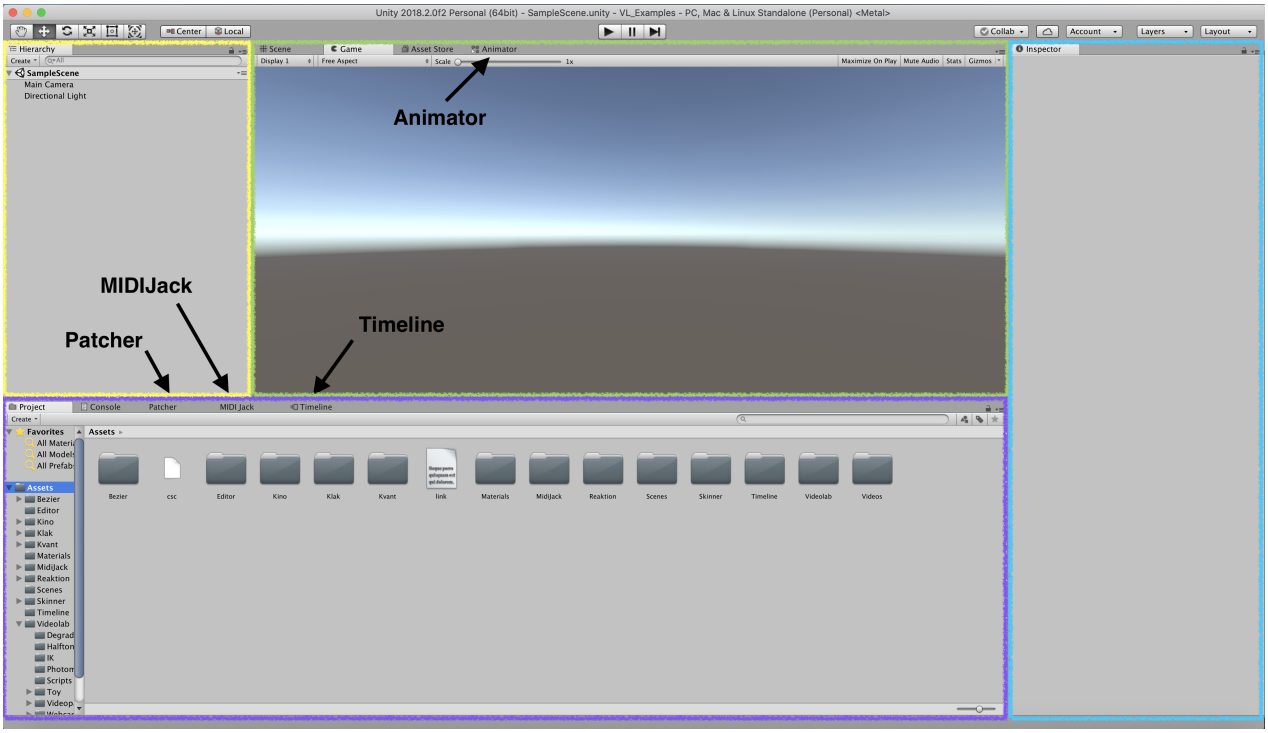
This is done by selecting the desired windows through the *Window* option in the workstation's menubar and dragging them to become a tab in the desired main window. The following image shows the four main windows of the default layout boxed out in yellow, green, blue, and purple.

[1]Please note: The author experienced difficulties importing the package using a PC. The workaround was to exclude the file "csc.rsp." The reader's experience may differ, but in case of the same difficulties, this workaround may save a significant amount of time and trouble. The author had to difficulties importing the full package into Unity. To exclude the "csc.rsp" file, when the Videolab package is being imported, the user has the option to include or exclude specific files. Simply uncheck the "csc.rsp" file.



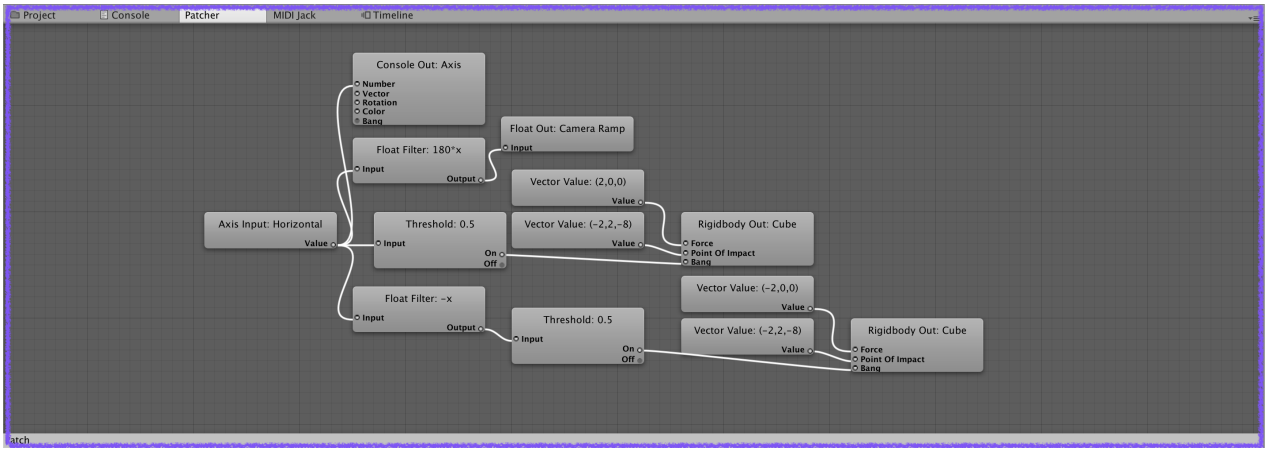
The next image shows the default workspace with additional windows clearly marked for:

- Klak Patches (the Patcher window on the bottom; purple box)
- MIDI input and output information flow (the MIDI Jack window on the bottom; purple box)
- The Timeline window (used in the Playable Director and Timelines short tutorial; on the bottom; purple box)
- The Animator window (used in the Animators short tutorial; on the bottom; green box)



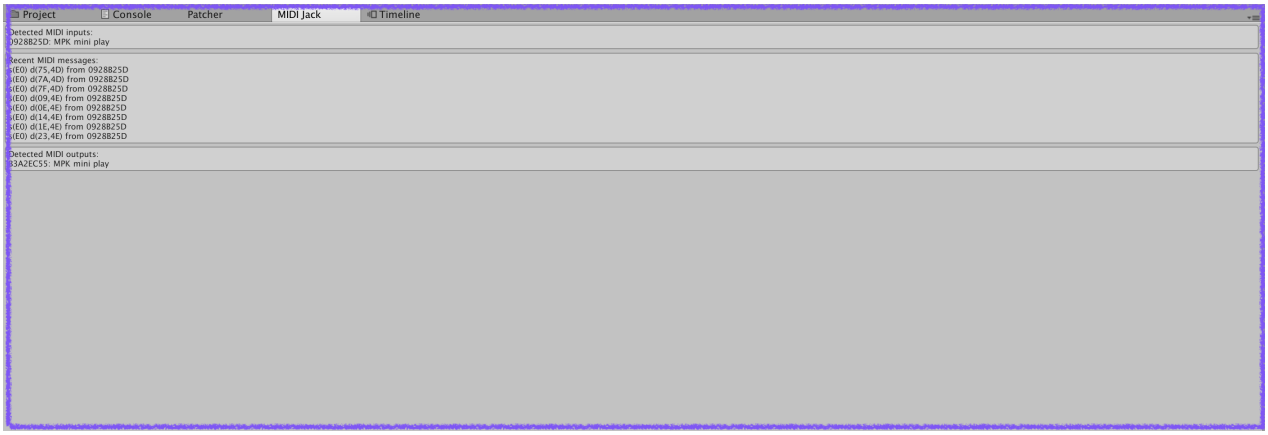
Klak Patch

The Klak Patch window appears automatically when a patch GameObject is created and the patcher is opened through the patch object's inspector. Alternatively, the window can be opened by accessing the *Window* option on the workstation's menubar, selecting Klak, then selecting Patcher. The window can then be placed as a tab in any of the main windows, and when selected appears as below. This window is necessary when making Unity scenes with Klak functionality.



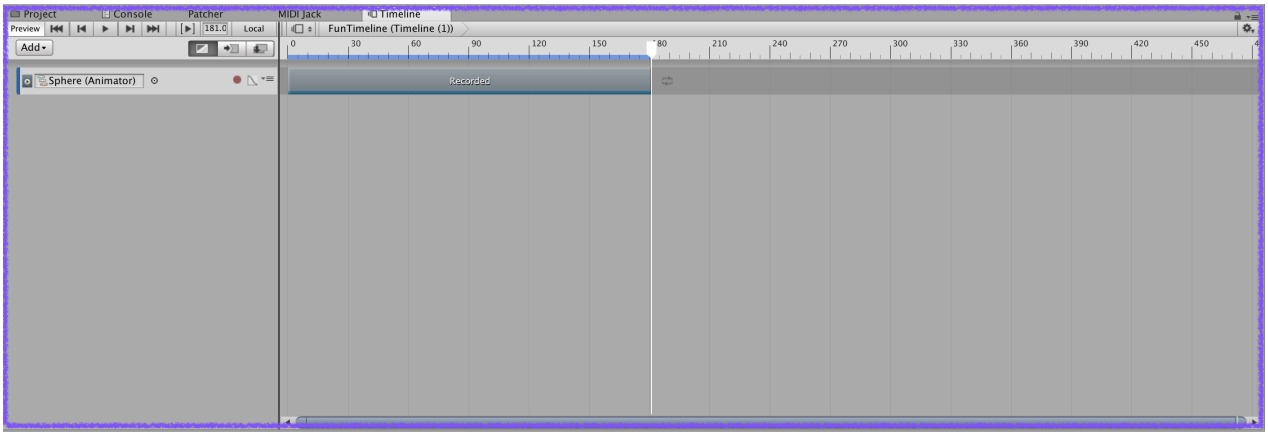
MIDI Jack

The MIDI Jack window is opened by accessing the *Window* option on the workstation's menubar and selecting MIDI Jack. This window shows incoming and outgoing MIDI data, detailing the message and the detected MIDI device(s). This window is optional and serves mostly (or entirely) for debugging.



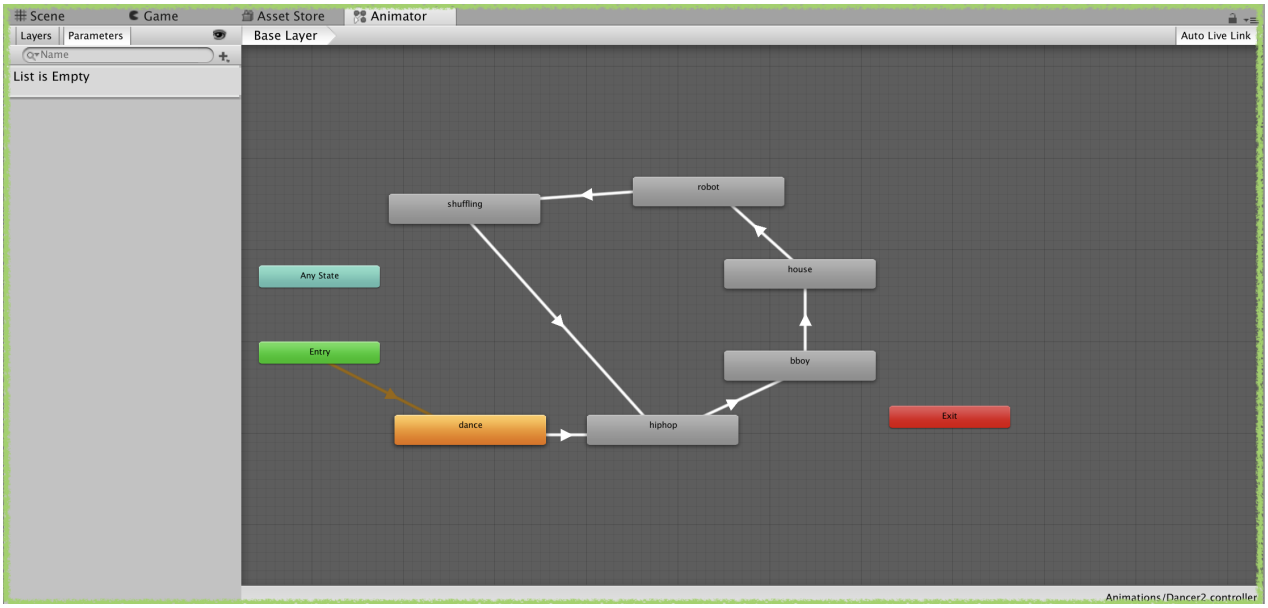
Timeline

The Timeline window is opened by accessing the *Window* option on the workstation's menubar and selecting Sequencing, then selecting Timeline. This window is necessary when creating Timeline animations, as detailed in the Playable Director and Timelines tutorial found in section 5.2



Animator

The Animator window is opened by accessing the *Window* option on the workstation's menubar and selecting Animation, then Animator. This window is necessary when manipulating animation sequences and cycles, as detailed in the Animator tutorial found in section 5.4.



3.3 Overview of Klak Nodes

The Klak patches are organized into seven classes: Input, Output, Conversion, Animation, Filter, Mixing, and Switching. The functionality of most of these classes is implied by their names.

In most cases, if not all, the Input and Output nodes will be the endpoints of connections within the patch. The Input allows the user to receive information from, for example, GameObjects, image effects, mouse & keyboard, MIDI controllers, and DAWs, but also allow the user to generate information that is sent directly out of the Input node's outlets without being tied to GameObjects, components, or controllers (e.g., pseudorandom numbers). The Output nodes, similarly, send information to, for example, GameObjects, image effects, MIDI controllers, and DAWs.

Conversion nodes take in information of a certain type and convert it. For example, the `Vector Components`

node takes in a vector3 (a 3D vector) and sends out three separate floats, one for each dimension.

There is only one Animation node: `Float Animation`. This node sends out successive floats in a time interval according to a curve defined in the node's inspector. This may be as simple as a line that goes from the value 0 to the value 1 in a single second, or it may be a sinusoidal curve that oscillates between -1 and 1 several times in a second.

There are just two Filter nodes: `Bang Filter` and `Float Filter`. `Bang Filter` can be turned on or off, allowing bang messages to be propagated through the node, or not. `Float Filter`, on the other hand, applies a linear transformation to an incoming float.

Mixing nodes for the most part, apply mathematical operations to two incoming pieces of data, sometimes of the same type or of different types. For example, a float can scale a vector, or two floats can be added, subtracted, multiplied, and so on.

Switching nodes use either floats or bangs as inputs and perform functions related to delaying and repeating bangs, or toggling between several options and sending bangs or floats depending on these options.

The next chapter goes over several example scenes using most Klak nodes, in which the following descriptions should become clearer. In addition, the appendix contains information on all Klak nodes.

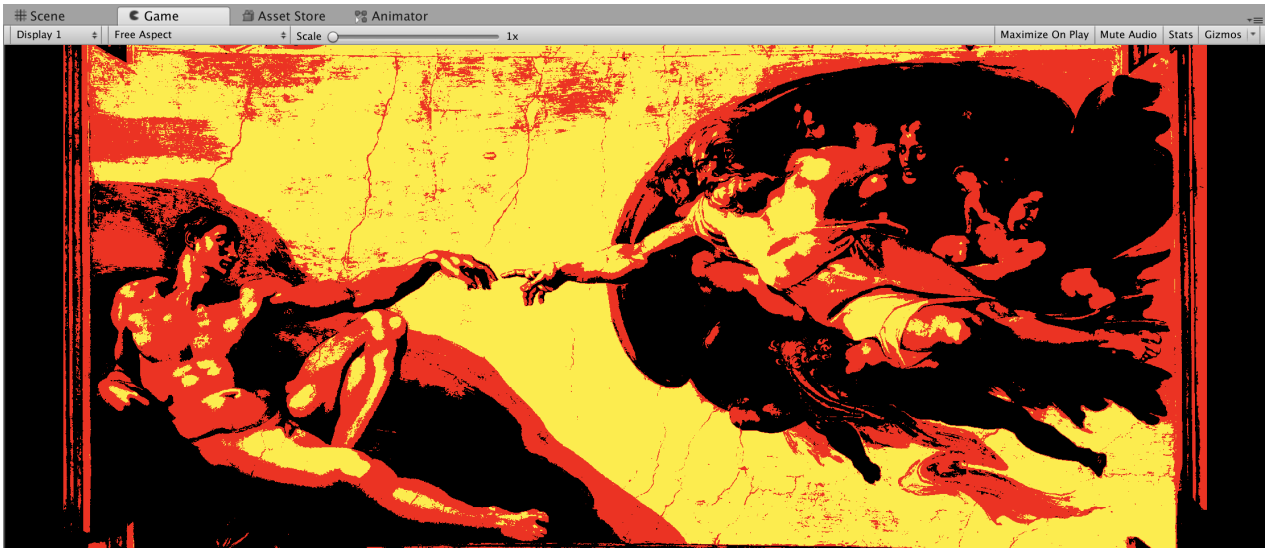
Chapter 4

Examples

This chapter goes through several Unity scenes exemplifying the function of most Klak nodes. The first seven examples (Adam, Boing, Peeler, Poly, Splash, Stripe, and Trail) are found in VideolabTest-master. These scenes were created by Keijiro and can be downloaded from his GitHub [2].[1] The next nine examples (Bounce, Button, Collider, ColorCubes, Dancer, Ramp, Video, Wall, and Warp) can be found in VL_Examples. These scenes were created by the author and can be downloaded from his GitHub (coming soon).

The reader is encouraged to see these scenes in action as they read through the examples in this chapter. It is invaluable for learning Klak to run the scenes, play with the patches, investigate the selected GameObject properties that are being received by Klak through Input nodes and that are sent out by Klak through Output nodes, among other exploratory and didactic activities. In particular, deconstructing the modules that activate various events so that each single event can be activated on its own is a fruitful exercise.

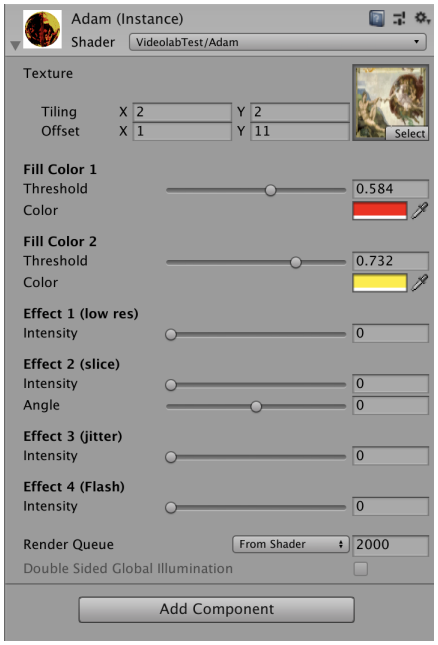
4.1 Adam



Nodes Used: `Note Input`, `Key Input`, `Random Value`, `Transform Out`, `Material Color Out`, `Material Float Out`, `Axis Rotation`, `Component Vector`, `Euler Rotation`, `HSB Color`, `Float Animation`, `Bang Filter`, `Float Filter`

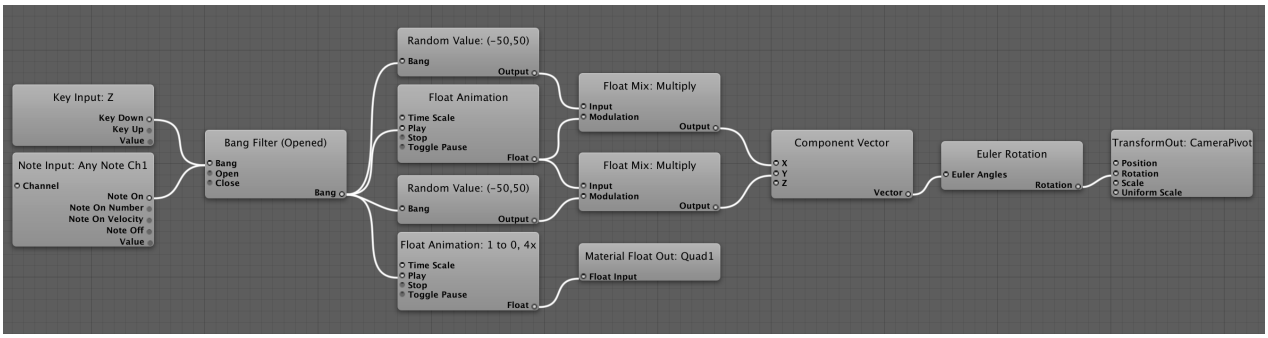
[1] Note that the author altered the names of the Klak nodes contained in Keijiro's scenes for the purposes of the images and explanations included in this chapter. Hence, when downloading these scenes from GitHub, the Klak nodes will not have the names shown here.

The Adam scene consists of a 3D box (the GameObject named *Quad*) containing Michelangelo's Adam painting on all faces. Quad's material has a number of variable properties. These are:



- The texture, which is the image currently set to Michelangelo's Adam painting.
- Two fill colors, which overlay different parts of the texture. These colors have float variables that manipulate the threshold of the color, which are accessible through `Material Float Out`. Additionally, the fill colors have color inputs which are accessible through `Material Color Out`.
- Four effects: low res, slice, jitter, and flash (all float variables go from 0 to 1, except the angle which goes from -1 to 1, and can be manipulated with `Material Float Out`).
    1. Low res - The higher the float variable Intensity, the more the image loses resolution and becomes pixelated.
    2. Slice - The float variable Intensity moves slices of the box in varying speeds and directions. The angle of the slices is controlled by the float variable Angle.
    3. Jitter - The float variable Jitter controls the intensity of the analog jitter image effect.
    4. Flash - The float variable Flash controls the intensity of the flash effect which is similar to lowering the threshold of both fill colors.

The user can control a few aspects of Quad and other aspects of the scene using either keyboard or MIDI input. The Klak patch creating this interactivity is comprised of 4 main modules. Images of these modules are provided below with explanations.



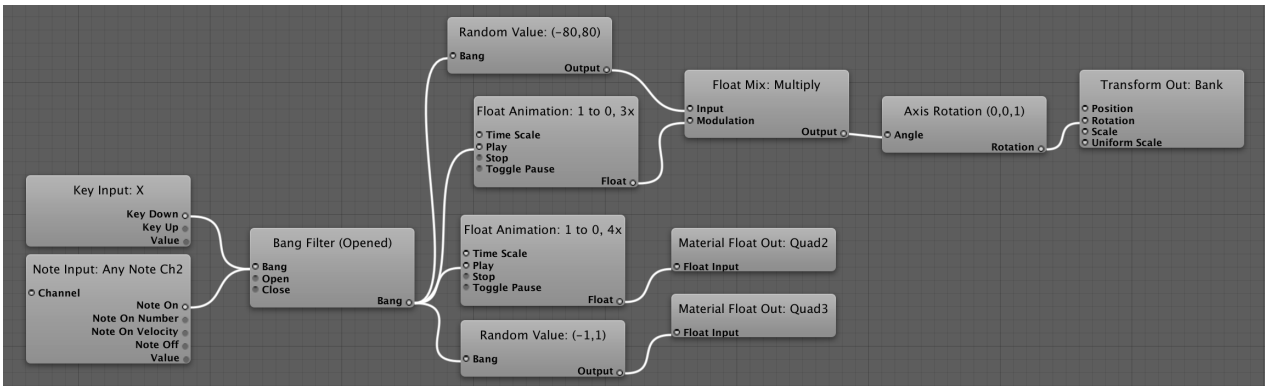
The above module is activated either when the Z key of the keyboard is pressed (`Key Input`), or a MIDI note is received through channel 1 (`Note Input`). The `Bang Filter` starts opened and simply transforms the incoming single bang into four separate bangs, activating two pseudorandom number generators (`Random Value`) and two `Float Animations`.[2]

[2] A closed `Bang Filter` would have to receive a bang in its Open inlet before functioning, as explained in the node's documentation.

The top `Float Animation` goes from 1 to 0 at a speed of 3 over the original duration of one second (i.e., it goes from 1 to 0 in about 0.33 seconds). These values get multiplied by the two random numbers (in the range from -50 to 50) and, through `Component Vector`, become the x and y components of the vector $(x, y, 0)$. Finally, these vectors are fed into `Euler Rotation`, which transforms the stream of vectors into a continuous sequence of rotations. These rotations get sent through `Transform Out` to manipulate the Camera Pivot GameObject, which is the second parent of the scene's camera.[3]

In short, each time the Z key is pressed or a MIDI note is received through channel 1, the camera pivots in a continuous and animated fashion over the course of roughly 0.33 seconds, each time in varying x and y intensities according to the random numbers generated. Note that each time the Z key is pressed or a MIDI note received through channel 1, the random number generators receive a bang and hence generate new random numbers.

The bottom `Float Animation` goes from 1 to 0 at a speed of 4 over the original duration of one second (0.25 seconds). Through `Material Float Out`, these values manipulate Quad's low res effect. In other words, Quad loses maximum resolution, then, continuously over 0.25 seconds, regains full resolution.



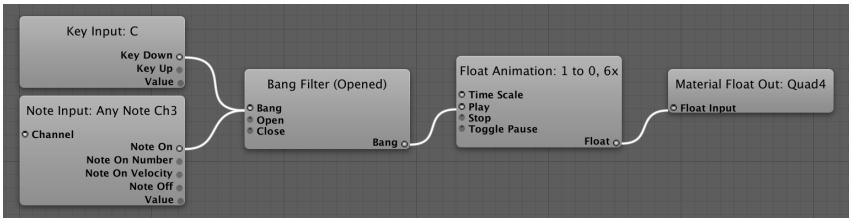
The above module starts very similarly to the first module covered earlier. As such, much of its description is the same. The module is activated either when the X key of the keyboard is pressed (`Key Input`), or a MIDI note is received through channel 2 (`Note Input`). The `Bang Filter` starts opened and simply transforms the incoming single bang into four separate bangs, activating two pseudorandom number generators (`Random Value`) and two `Float Animation`s.[4]

The top `Float Animation` goes from 1 to 0 at a speed of 3 over the original duration of one second (i.e., it goes from 1 to 0 in 0.33 seconds). These values get multiplied by the random number (in the range from -80 to 80) and, through `Axis Rotation`, animate a rotation about the z-axis. This rotation manipulates the Batch empty GameObject, which is the first parent of the scene's camera. Basically, this causes slight rotational jerks in the camera view.

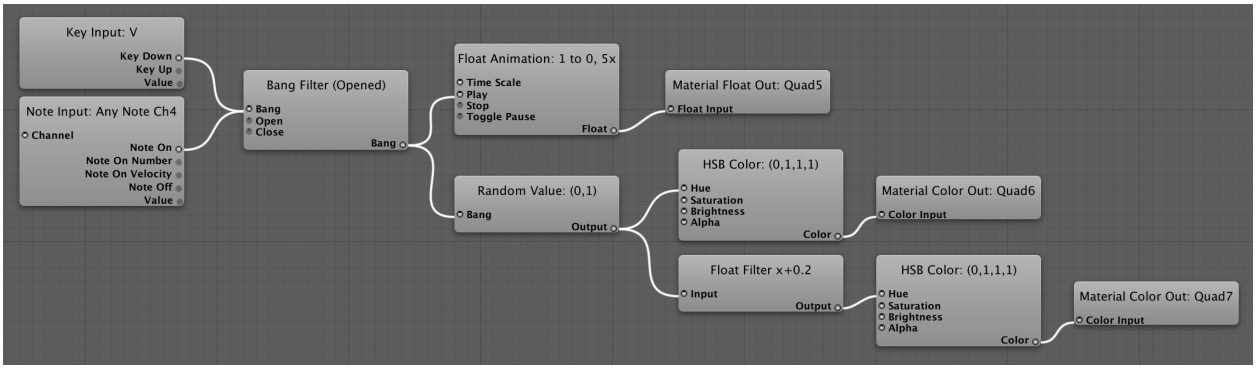
The bottom `Float Animation` goes from 1 to 0 in 0.25 seconds, and directly changes the intensity of Quad's slice variable. The `Random Value` (between -1 and 1) sets the angle of the slice effect.

[3] Making the camera a child of an empty GameObject is a way of achieving pivoting rotations since a rotation of the empty GameObject will rotate all of its children (including the camera) around its center of rotation. In fact, in Adam, the camera has two parents.

[4] A closed `Bang Filter` would have to receive a bang in its Open inlet before functioning, as explained in the node's documentation.



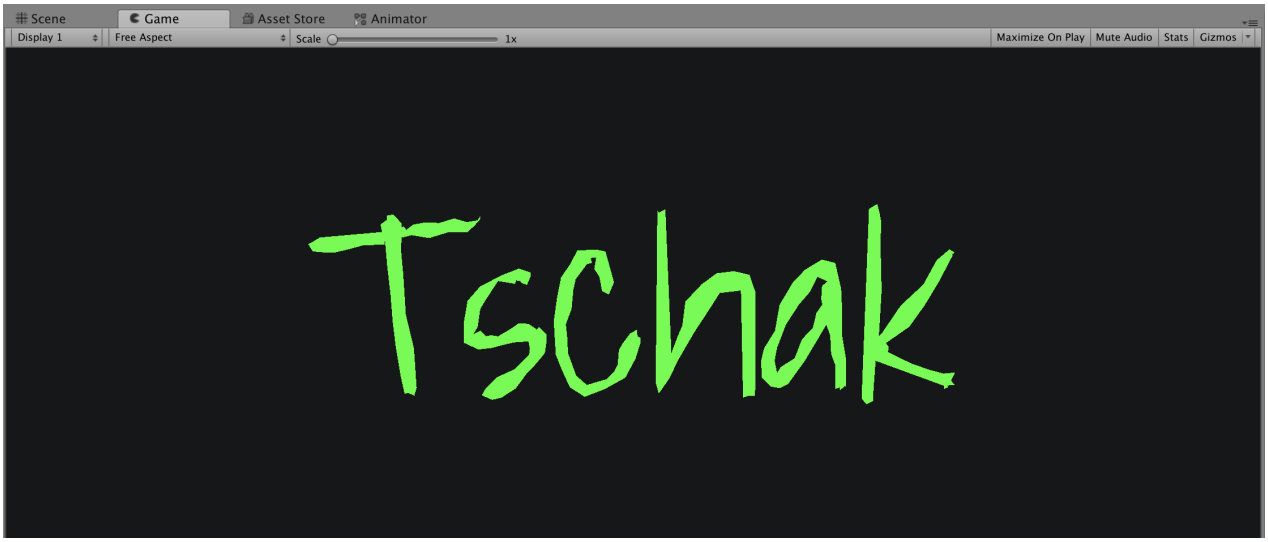
The above module is activated either when the C key of the keyboard is pressed (`Key Input`), or a MIDI note is received through channel 3 (`Note Input`). The `Bang Filter` is redundant here since it only outputs a single bang. The bang activates a very fast float sequence from 1 to 0 (about 0.17 seconds) that animates the jitter image effect.



Adam's last module is activated when either the V key of the keyboard is pressed (`Key Input`), or a MIDI note is received through channel 4 (`Note Input`). Once activated, two bangs are sent out of the `Bang Filter`, activating a `Float Animation` and a `Random Value`. The float sequence from 1 to 0 over 0.2 seconds generated by `Float Animation` is sent to Quad's flash material effect through `Material Float Out`. Basically, this causes Quad to be fully colored in by the second fill color for a brief moment.

The random number between 0 and 1 generated by `Random Value` becomes the hue of the first fill color and is slightly skewed before becoming the hue of the second fill color. The random float value becomes a hue through the `HSB Color` node and is sent to Quad's material through `Material Color Out`. One of the hues comes directly from the random float, while the other is a translation of the random float by 0.2, which is achieved with the `Float Filter` node.

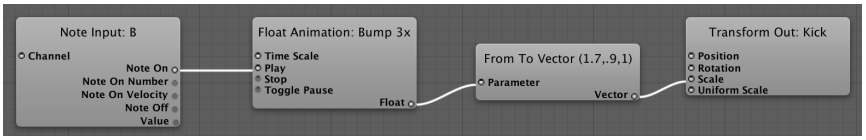
4.2 Boing



Nodes Used: `Note Input, Transform Out, Material Float Out, From To Vector, Float Animation, Float Filter`

The Boing scene contains 5 small modules, each of which activates words or effects (Boom, Tschak, Ah, Kick, and Boing) that grow on the screen before disappearing. Due to the similar nature of all modules, focus is only given to the module connected to the Kick effect.

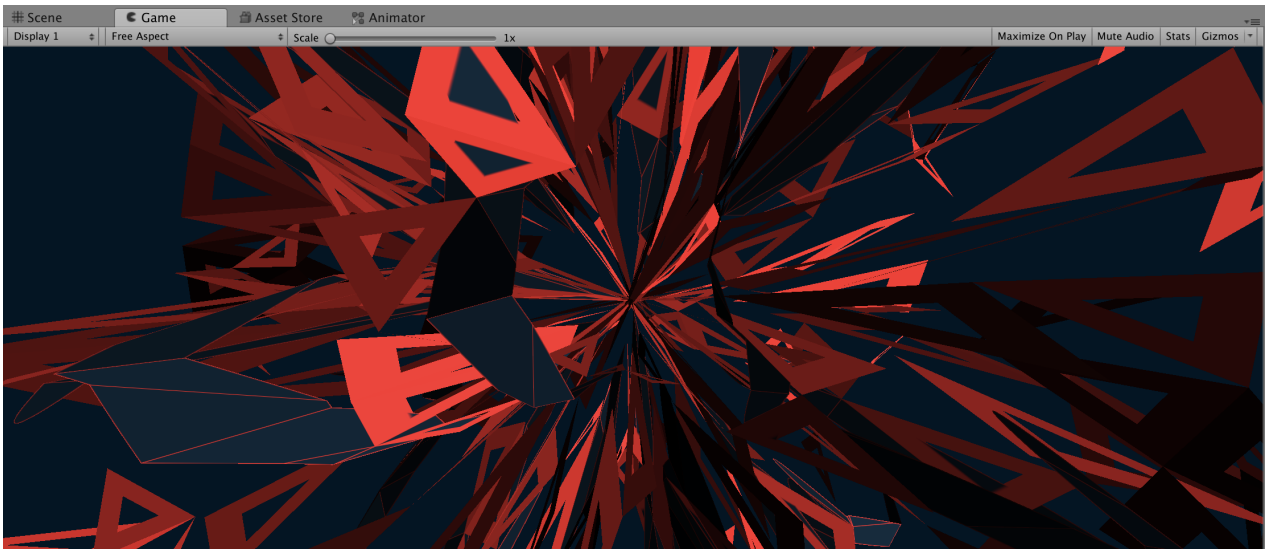
The Kick GameObject is a wobbly 2D disc with a shader that uses noise to wobble the object, has a color, and a polygon count (e.g., if the polygon count's variable is set to 4 or 5, the object will be a square or a pentagon, respectively). Since the nodes do not manipulate the Kick's shader variables, these variables will not be explored here. However, the reader is encouraged to manipulate the object's shader variables to observe and explore the effects. Further patchwork is additionally encouraged to extend the scene by altering these variables creatively.



Kick's patch module above is simple. A `Float Animation` is activated by `Note Input` when the note B is played by a detected MIDI instrument on any channel. The float sequence, in a period of about 0.33 seconds, sharply rises from 0 to 1, then goes more steadily back to 0. These floats, through `From To Vector`, smoothly animate vectors from $(0, 0, 0)$ (when the incoming float is 0) to $(1.7, 0.9, 1)$ (when the incoming float is 1). These vectors manipulate the Kick GameObject's scale through `Transform Out`.

Basically, when a B note is played, the Kick GameObject's scale quickly becomes $(1.7, 0.9, 1)$, then more steadily but still quickly goes back to $(0, 0, 0)$. In other words, the wobbly 2D disc rapidly emerges and covers the screen before shrinking and disappearing.

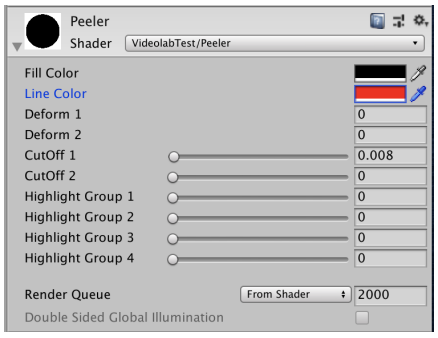
4.3 Peeler



Nodes Used: `Note Input, Key Input, Mouse Button Input, Event Out, Material Float Out, Float Animation, Float Filter`

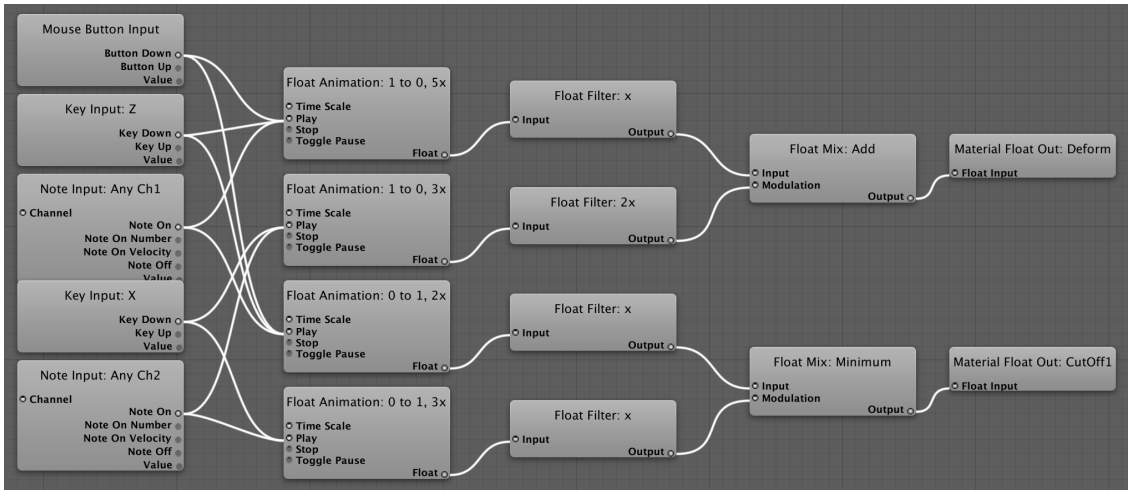
The Peeler scene contains an intricate setup. There is a parent-child hierarchy of empty GameObjects: the Camera Pivot (ultimate parent), the Distance (child of Camera Pivot), and the Snap Point (child of Distance). The Camera Pivot and Distance GameObjects have Brownian Motion scripts causing the objects to rotate and move. The scene's Main Camera object has a Smooth Follow script, making the camera follow the Snap Point object. Running the scene and observing the camera movements in scene mode (rather than game mode) can be an enlightening exercise.

The star of the scene is the Sphere object. The object's rotation is manipulated by a Brownian Motion script so that it is always slightly turning in different directions. Additionally, the Sphere has a shader with the following variables:



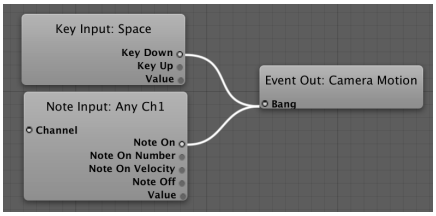
- Fill Color & Line Color are the color of the sphere and the color of the lines of the triangulation over the sphere's surface, respectively. These color variables can be manipulated with `Material Color Out`.
- Deform 1 & Deform 2 morph the sphere by causing different groups of vertices to move in varying directions and amplitudes. These float variables can be manipulated with `Material Float Out`.
- Cutoff 1 & Cutoff 2 cause a ghostly effect that seems to come from within the sphere. These float variables can be manipulated with `Material Float Out`.
- Highlight Groups 1-4 highlight different sets of triangles over the triangulation of the sphere. They can be manipulated with `Material Float Out`.

There are three separate patches in the Peeler scene. Attention is given only to two of the modules: those containing `Mouse Button Input` and `Event Out`, which are not found in any other VideolabTest-master example scenes.



The above module receives input from mouse clicks (`Mouse Button Input`), the Z and X keyboard keys (`Key Input`), and MIDI notes from channels 1 and 2 (`Note Input`). These inputs are connected in varying combinations to four `Float Animation`s, two of which go from 1 to 0 and the remaining two from 0 to 1. These float sequences undergo a linear operation through different `Float Filter`s, most of which are redundant (they take the input x and send out x unchanged). One of these `Float Filter`s doubles the value of x. The first two sequences of floats are added (`Float Mix`) and sent to the Sphere's deform shader effect. The minimum value of the last two sequences of floats is taken by `Float Mix` before manipulating the Sphere's Cut Off 1 shader effect.

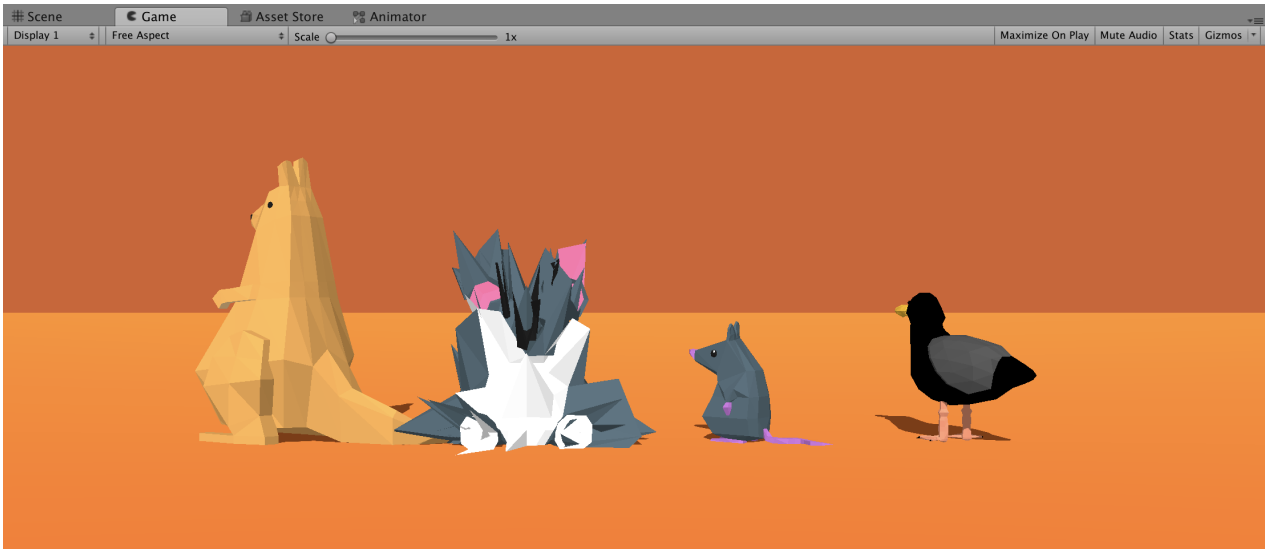
The inputs in this module each trigger two `Float Animation`s, ultimately triggering both effects of the module. Hence, any of the inputs in the module basically cause a short spike (above 0) in the Deform 1 shader effect, and a short dip (below 1) in the Cut Off shader effect.



The above module takes input from the spacebar on the keyboard (`Key Input`) and from any MIDI note on channel 1 (`Note Input`). This input simply bangs an `Event Out` node, which is set up to rehash the Brownian motion of one of the objects affecting the flux of the Main Camera. This effectively causes the Main Camera to reset its position.

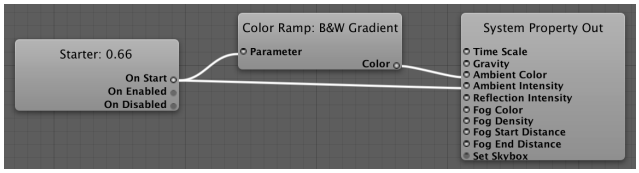
Note that the `Event Out` node does not have an empty field to select a GameObject and to further select where in the object the event will occur. Instead, the user must create an empty field by adding an entry to `Event Out`'s list in the node's inspector.

4.4 Poly



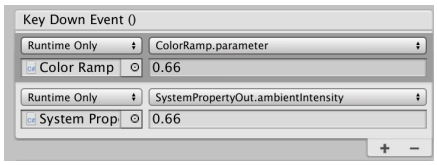
Nodes Used: `Note Input`, `Starter`, `Material Float Out`, `System Property Out`, `Color Ramp`, `Float Animation`, `Float Filter`

Poly has four animals that are continuously spinning in place. This scene is similar to the Peeler scene where the input deformed the sphere; here the input deforms the animals. Partly due to this, for this scene, attention is given only to the patch module containing the nodes `Starter` and `System Property Out`. This is also because these nodes are not found in any other VideolabTest-master examples.

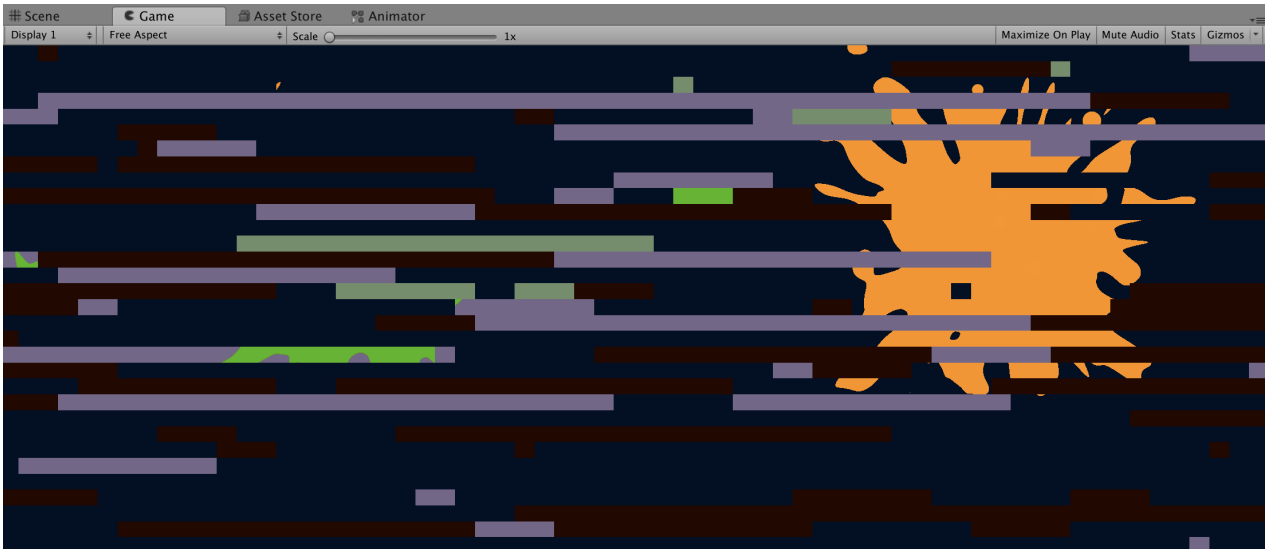


The above module is not activated by any input. Instead, it is activated by the `Starter` node, which sends the float 0.66 immediately as the scene starts. This float flows into `Color Ramp` to choose a gray value corresponding to the float 0.66 in a gradient from black to white. This color is sent to `System Property Out`'s Ambient Color. The float also flows into `System Property Out`'s Ambient Intensity.

The effect of this is that the scene lights up. To explore this and have control over when the effects are activated, the reader is encouraged to replace the `Starter` node with a `Key Input` node. The node can be left with the default option for the spacebar. After connecting the node's Key Down outlet to the `Color Ramp` inlet and `System Property Out`'s Ambient Intensity inlet, change the values in `Key Input`'s list from 1 to 0.66. This is shown on the right. And, of course, the reader is encouraged to try other values and inlets.

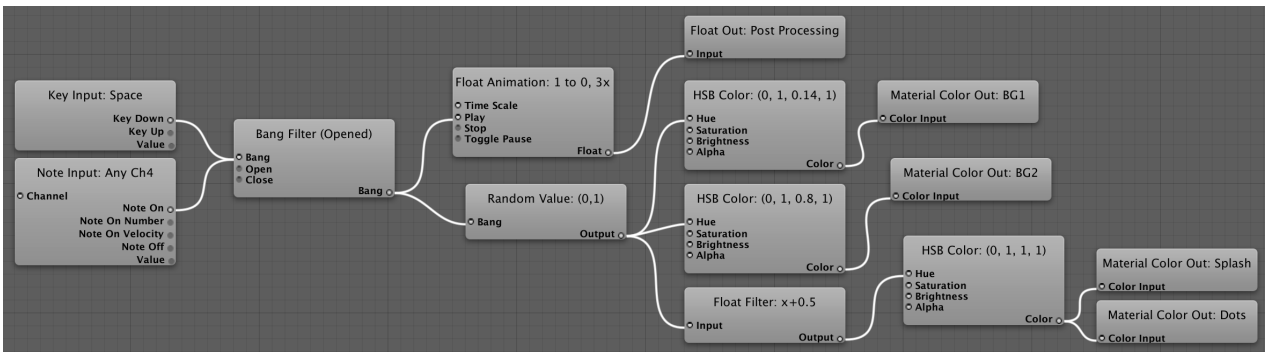


4.5 Splash



Nodes Used: `Note Input`, `Key Input`, `Random Value`, `Particle System Out`, `Float Out`, `Material Color Out`, `Material Float Out`, `HSB Color`, `Float Animation`, `Bang Filter`, `Toggle`
Associated Kino Image Effect: Digital Glitch

Splash is similar to Boing in the sense that they both cause an image to quickly appear and then disappear. Two major differences are that Splash has only three effects (Boing has five), and it uses Particle Systems. The GameObject BG is manipulated by its shader variables, and is an opening-and-closing shutter effect. The details of the shader variables will not be discussed because a discussion already exists for the shader variables in Adam and Peeler.



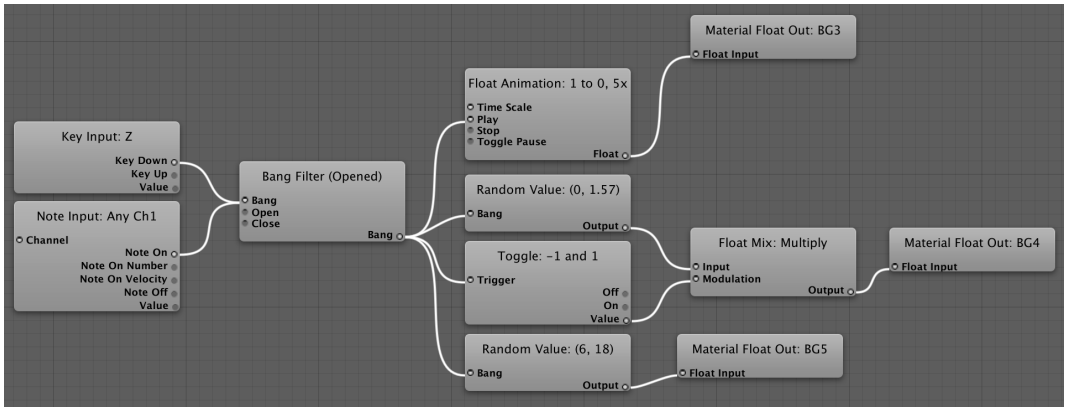
The above module causes a Digital Glitch effect and changes the color scheme of the scene. This is done by first taking in an input bang from the spacebar (`Key Input`) and from any MIDI note coming in from channel 4 (`Note Input`). The subsequent `Bang Filter` relays this bang to a `Float Animation` and a `Random Value`.

`Float Animation` smoothly generates floats from 1 to 0 in about 0.33 seconds, which are sent, through `Float Out`, to the Main Camera's Digital Glitch effect. This causes a Digital Glitch to appear with maximum intensity before it fades away in one-third of a second.

The random value between 0 and 1 (`Random Value`) ultimately alters four colors: the two colors of BG's shutter shader effect, and the colors of the particle systems Splash and Dots. The same random float is used as the hue (`HSB Color`) of BG's shutter effect's two colors (`Material Color Out`) which differ only in brightness.

The value of 0.5 is added to the same random float (`Float Filter`), which then becomes the hue of the color (`HSB Color`) that is sent to Splash and Dots (`Material Color Out`).

Hence, whenever the spacebar is pressed or a MIDI note played on channel 4, a Digital Glitch effect happens and the color scheme of the scene changes.

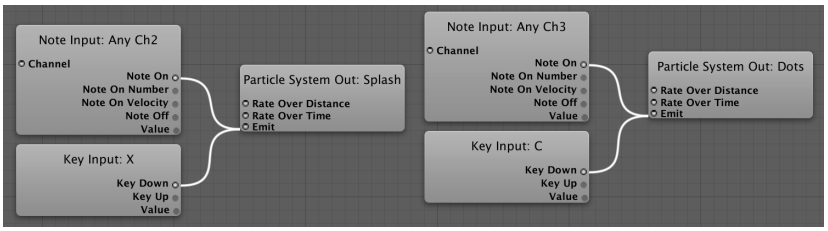


The above module is activated by the Z key on the keyboard (`Key Input`) or a MIDI note played on channel 1 (`Note Input`). When this happens, the bang goes through a `Bang Filter` to subsequently activate a `Float Animation`, two `Random Value`s, and a `Toggle`. Ultimately, BG's float-valued shader properties are manipulated (`Material Float Out`).

The top `Float Animation` generates a short 0.2 second sequence of floats from 1 to 0, which get sent to BG's threshold variable. This causes the shutter effect to quickly appear fully closed before opening once more.

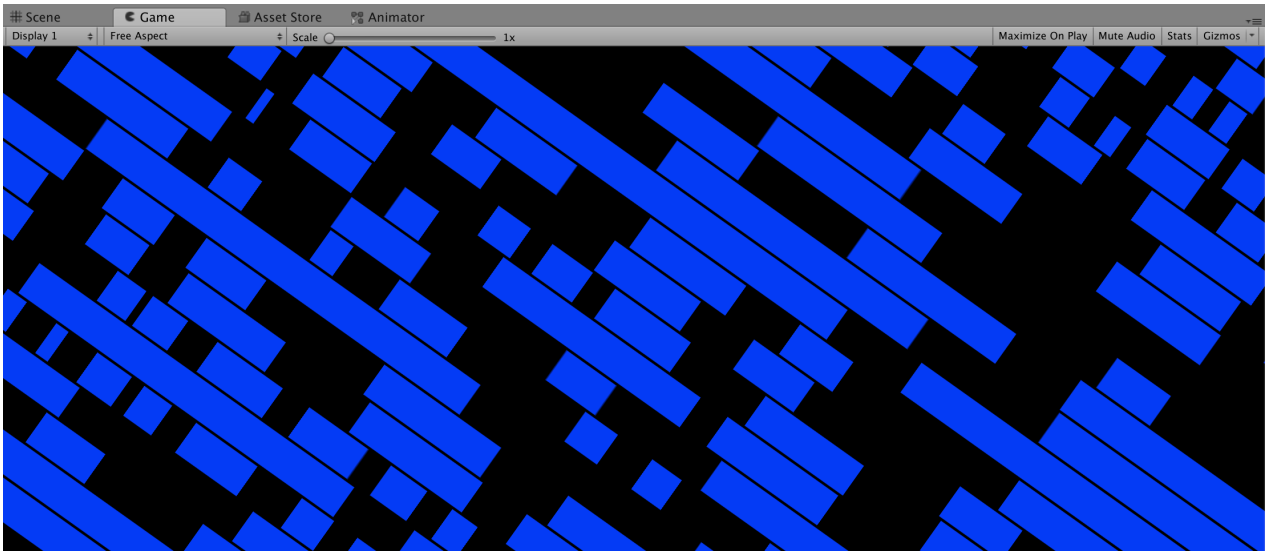
The `Random Value` and `Toggle` get multiplied in the `Float Mix` before being sent to BG's angle variable. The engineering behind this is clear: the random value is between 0 and $\pi/2$ (or, between 0 and 1.57 or between 0 and 90 degrees). The toggle alternates between the values of 1 and -1. As such, the angle of the shutter always alternates between an angle between 0 and $\pi/2$ and one between $-\pi/2$ and 0.

Finally, the bottom `Random Value` generates a float between 6 and 18 and is sent to BG's repeat effect, which is essentially the number of blinds squeezed into the shutter effect.



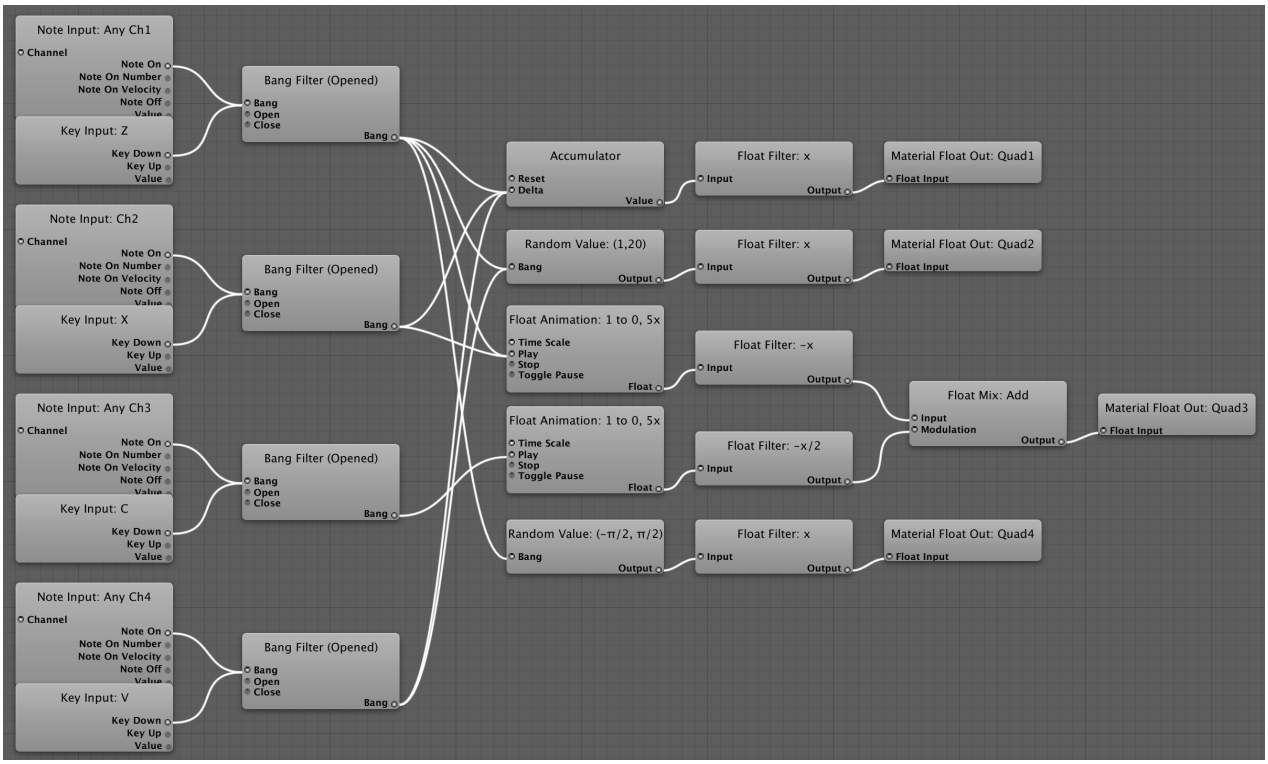
The above image contains two similar modules, the left activated by the X keyboard key (`Key Input`) or a MIDI note on channel 2 (`Note Input`), and the right is activated by the C keyboard key or a MIDI note on channel 3. The bang generated by these inputs are sent to the Emit inlet of a `Particle System Out` node, which causes the system to emit its particles. A short tutorial on Particle Systems can be found in chapter 5.

4.6 Stripe



Nodes Used: `Note Input`, `Key Input`, `Random Value`, `Float Out`, `Material Color Out`, `Material Float Out`, `Accumulator`, `HSB Color`, `Float Animation`, `Bang Filter`, `Float Filter`

Stripe consists of many ribbon like rectangles that move around, which are produced by a shader on the plane GameObject Quad. The patch dictates how these stripes rotate, grow, and move by manipulating Quad's shader variables. There are two modules in Stripe's Klak patch. We will go over only the first, which makes use of the `Accumulator` node.



The above module takes four keyboard inputs (the keys Z, X, C, and V, through `Key Input`) and four MIDI note inputs (in channels 1-4, through `Note Input`). The bangs generated by these inputs get sent to `Bang Filters`, which then get propagated in varying combinations to trigger an `Accumulator`, two `Random Values`, and two `Float Animations`.

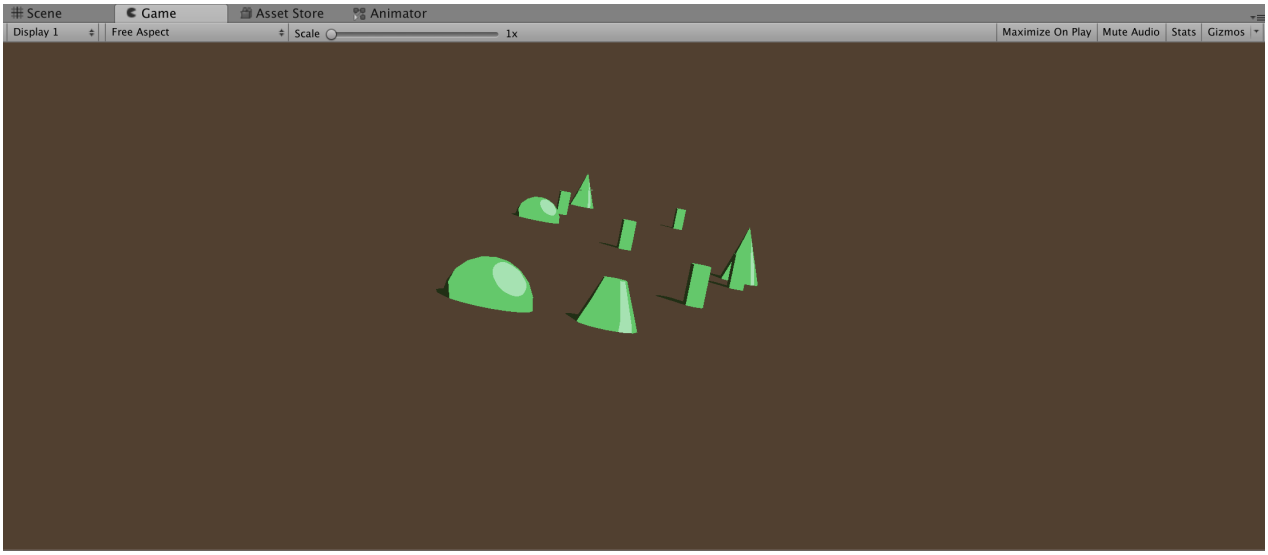
The first time the `Accumulator` receives a bang, it sends out a 1. The next bang causes it to send out a 2, then a 3, and so on. These values go through a redundant `Float Filter` (redundant because it receives x and outputs x), before getting sent to Quad's seed variable through `Material Float Out`. Hence, this simply moves up the value of the seed that generates Quad's broken tiles.

The first `Random Value` simply produces a float value between 0 and 20, which goes through another redundant `Float Filter`. This value ultimately goes out of a `Material Float Out` to change Quad's rows variable, which dictates the number of stripes in the effect.

Two `Float Animations` are triggered, both going from 1 to 0 in 0.2 seconds in the exact same way, and going through two similar `Float Filter`s, one simply multiplies the incoming floats by -1, and the other by $-1/2$. These floats are added in `Float Mix` before getting sent to Quad's threshold variable through `Material Float Out`. The threshold variable of Quad's shader goes from -1 to 1 and dictates the amount of empty space between the broken ribbons, with -1 meaning the ribbons fully take over the screen, and 1 means there empty space between the ribbons fully take over the screen. The consequence of this is a short animation that starts with no empty space between the ribbons and proceeds to grow the empty the space. Note that the two `Float Animations` just described can be activated at different times.

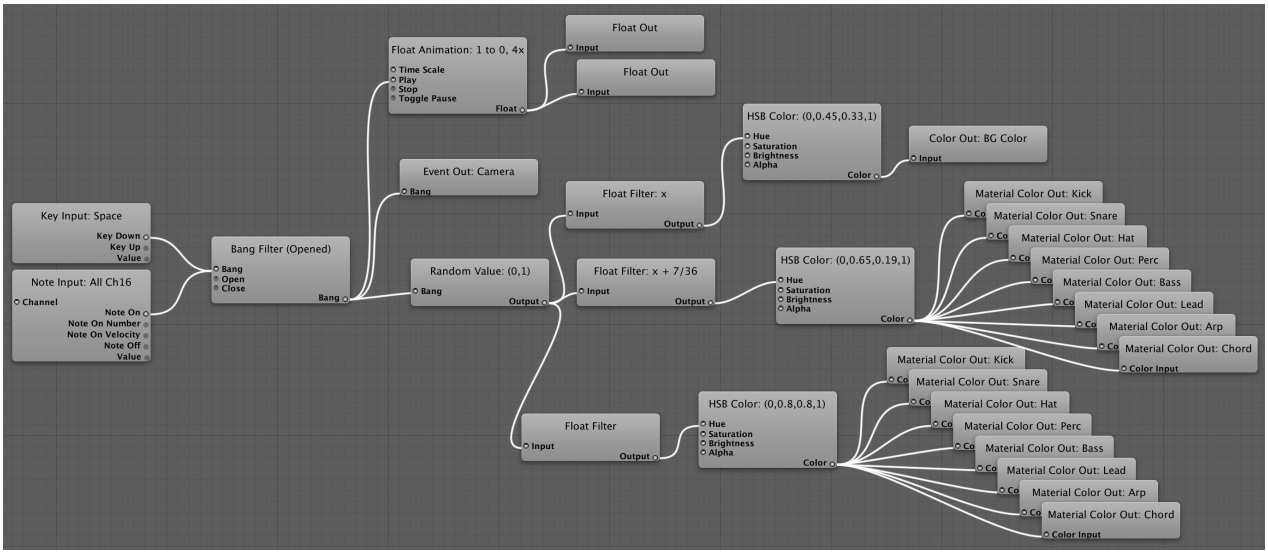
Finally, the last `Random Value` generates a number between $-\pi/2$ and $\pi/2$ (an angle between -90 and 90 degrees), which, after going through a redundant `Float Filter`, gets sent through a `Material Float Out` to manipulate Quad's rotation variable.

4.7 Trail



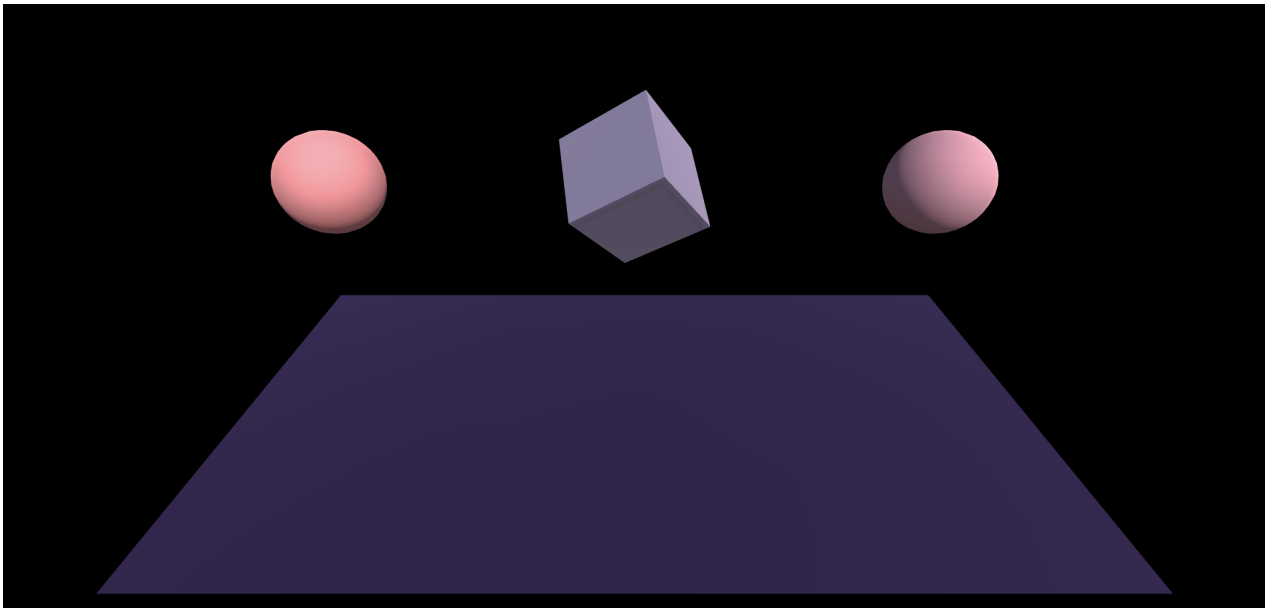
Nodes Used: `Note Input, Key Input, Random Value, Particle System Out, Color Out, Float Out, Material Color Out, HSB Color, Float Animation, Bang Filter, Float Filter`

Trail is a scene similar to Boing and to Splash. The main difference in Trail is that instead of the images disappearing, they trail off in the distance. Because most of the functionality of the patches in the scene are similar to those previously explained, focus is given to only a piece of a single module: the flow that ends with `Color Out`.



The above module is activated the spacebar is pressed (`Key Input`) or when a MIDI note is played on channel 16 (`Note Input`). The flow that ends in `Color Out` involves a `Bang Filter` which splits the incoming bang in three, followed by a `Random Value` between 0 and 1, which gets sent to a redundant `Float Filter` (redundant because the input x is unchanged). The random float between 0 and 1 ultimately makes its way as the hue of `HSB Color` with saturation 0.45, brightness 0.33, and alpha 1. The generated color is then sent to the background color of the Main Camera through `Color Out`.

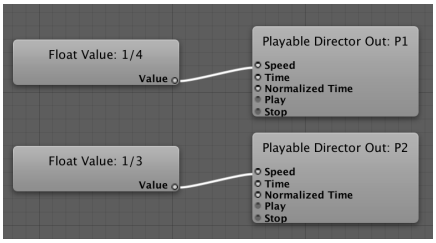
4.8 Bounce



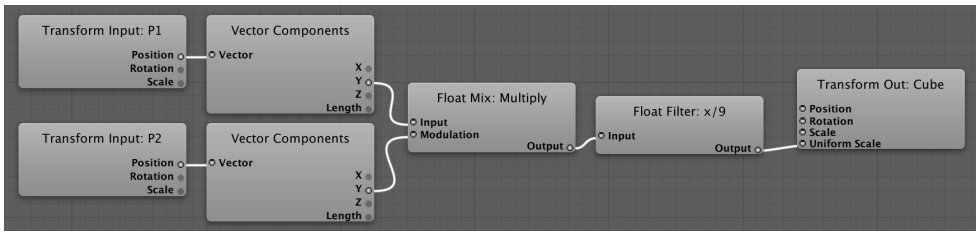
Nodes Used: `Float Value, Transform Input, Playable Director Out, Transform Out, Vector Components, Float Filter, Float Mix`
Associated Tutorial: Playable Director and Timelines

First, note that the Bounce scene has no human-computer interactivity. The scene has two bouncing spheres and a floating cube. The spheres bounce according to an animation created using the Playable Director and Timelines. Their bounces come from the same animation sequence, but become polyrhythmic when their speeds are adjusted accordingly. The cube's size is scaled according to the product of the cycles of the

polyrhythmic spheres; only when the spheres align does the cube reach its full size. There are two module classes in Bounce: one adjusts the speed of the animation for each sphere, and the other scales the cube.

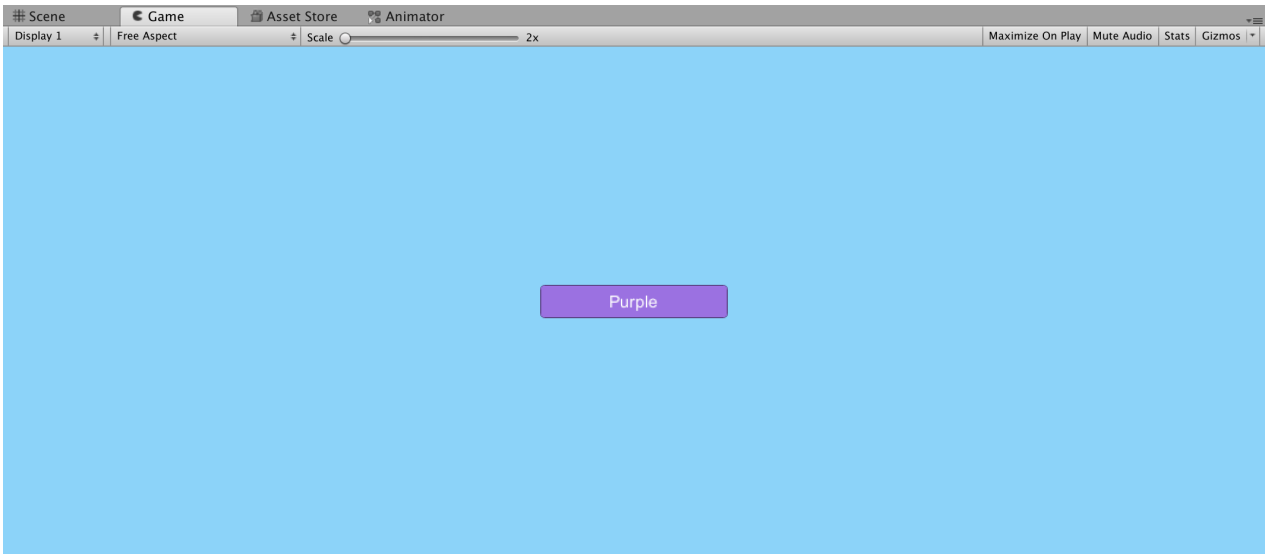


The above module simply scales the speed of the two animations, which is achieved with `Float Value` nodes adjusting the speed of Playable-Director-driven animations. The top `Float Value` node sends a 1/4 to the speed inlet of the `Playable Director Out` node that controls the left bouncing sphere. Similarly, the below `Float Value` node sends a speed of 1/3 to the right sphere's animation. The polyrhythmic effect is achieved since it takes the left sphere three full bounces and the right sphere four full bounces for the spheres to reset their collective cycle.



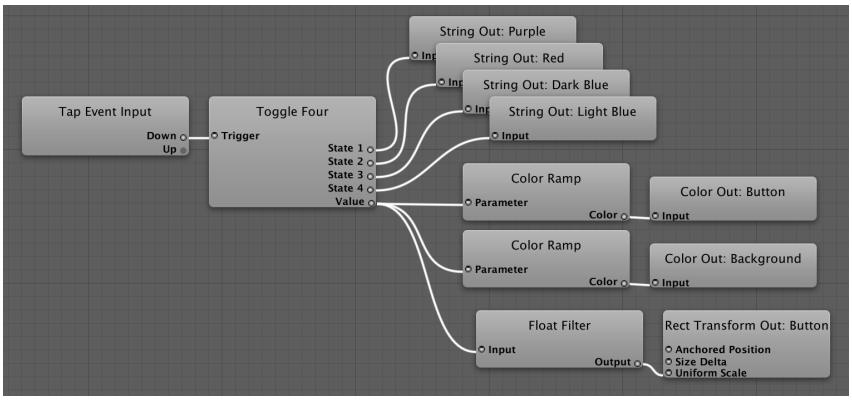
The two `Transform Inputs` grab the position vectors of the two bouncing spheres and send them to `Component Vectors`. Since the spheres only bounce along the y-axis, the y values are the only ones that are dynamic, going from the original height of the spheres, 3, to 0. These values get multiplied using `Float Mix`. Note that the product of these values is 9 just in case both spheres are at their original starting positions. This product gets scaled by 1/9 using `Float Filter` so that their range becomes 0 to 1, instead of 0 to 9. Finally, these floats are sent to the Uniform Scale inlet of a `Transform Out` node that is connected to the cube. The option to "Add To Original" in `Transform Out`'s inspector is off, so that the cube's overall scale ranges from 0 to 1, instead of 1 to 2.

4.9 Button



Nodes Used: `Color Out`, `Color Ramp`, `Float Filter`, `Tap Event Input`, `Rect Transform Out`, `String Out`, `Toggle Four`

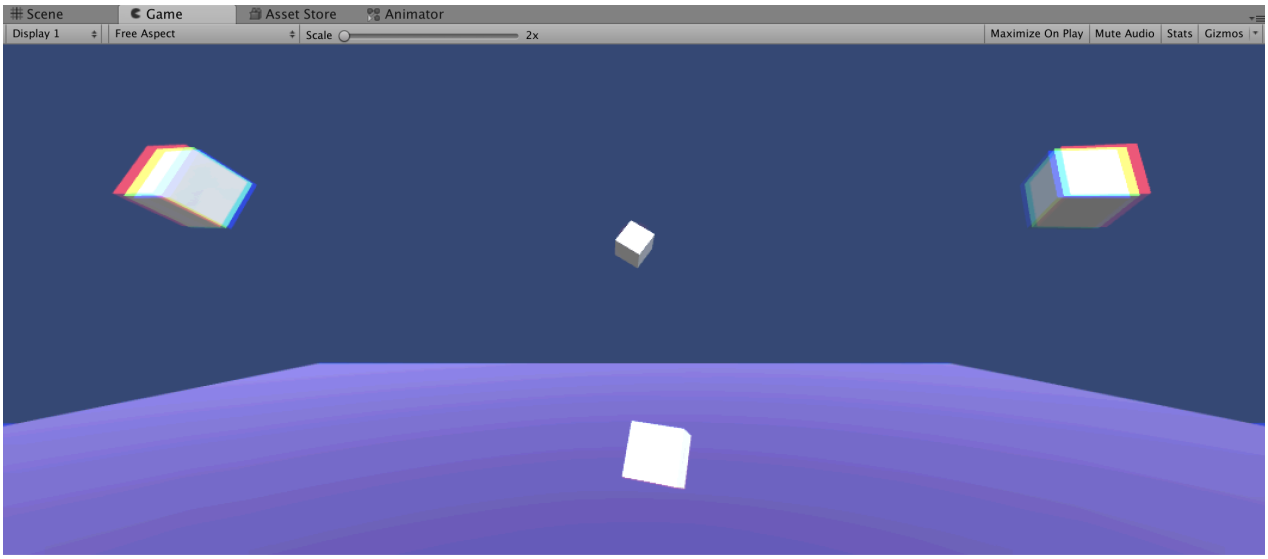
Button is a simple scene meant primarily to illustrate some of Klak's UI nodes and the `String Out` node. A colored button with text is in the middle of the scene. Once it is pressed, the background of the scene changes to the button's color, the button grows, and becomes a new color.



The `Tap Event Input` node detects when its associated UI element is pressed. In this scene, the associated UI element is the button. Once the button is pressed, it sends a bang to `Toggle Four`, which alternates between its state outlets. The first bang it receives causes it to send a bang out of its State 1 outlet and the corresponding value out of the Value outlet. The next bang will cause it to send a bang out of its State 2 outlet and the corresponding value out of the Value outlet. This process cycles, and every time a different `String Out` node is activated, sending in succession the strings "Purple", "Red", "Dark Blue", and "Light Blue" to the UI button.

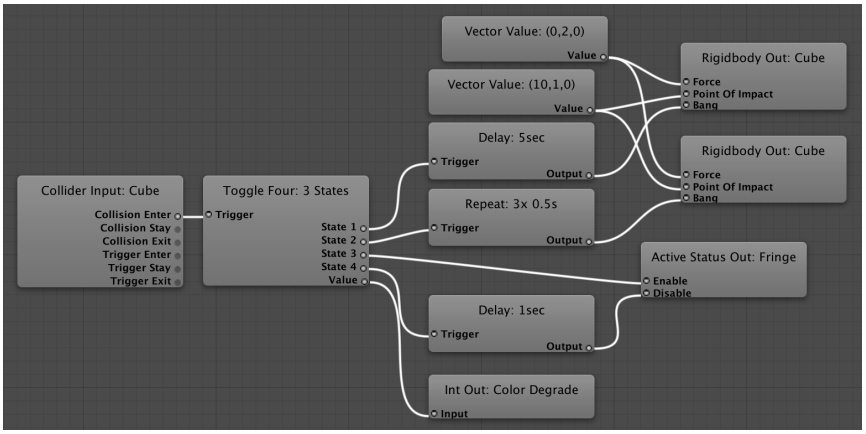
The values cycled activate two `Color Ramp`s, which change the color of the button and the Main Camera's background through `Color Out`. The two `Color Ramp`s have the same color sequences, where the button's sequence is earlier than the background color's sequence by one color. Lastly, the `Float Filter` multiplies the float received by 100 before sending to the uniform scale inlet of the `Rect Transform Out`.

4.10 Collider



Nodes Used: `Vector Value`, `Collider Input`, `Transform Input`, `Active Status Out`, `Rigidbody Out`, `Int Out`, `Rotation Out`, `Rotation Mix`, `Delay`, `Repeat`, `Toggle Four`
Associated Kino Image Effect: Fringe

The larger central cube of Collider is the only cube that has a Rigidbody and Physic object, both of which are necessary to detect collisions. The cube falls and rotates, causing all other cubes to rotate. Each time it bounces, the color degrades by one bit. The `Delay` node is used in a couple of ways, one of which is to give the central cube one last kick once it has settled.



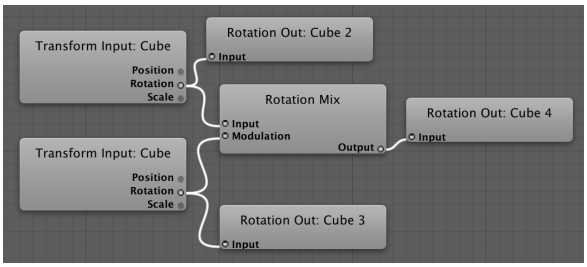
Once a collision has been detected by the `Collider Input` node, a bang triggers the `Toggle Four` node that performs numerous activities. Its sucessive bangs, in order, hit two `Delay` nodes, a `Repeat` node, and an `Active Status Out` node.

There are two `Vector Value` nodes which set up the `Rigidbody Out` to load the appropriate point of impact and force. With these vectors loaded, the `Rigidbody Out` node is activated when it receives a bang from the `Delay` and `Repeat` nodes, which delay the bang by 5 seconds, and repeats three bangs spaced out by 0.5 seconds, respectively.

The Main Camera's Fringe image effect is activated by the third bang coming from `Toggle Four` and deactivated, with a delay of 1 second, by the fourth bang.

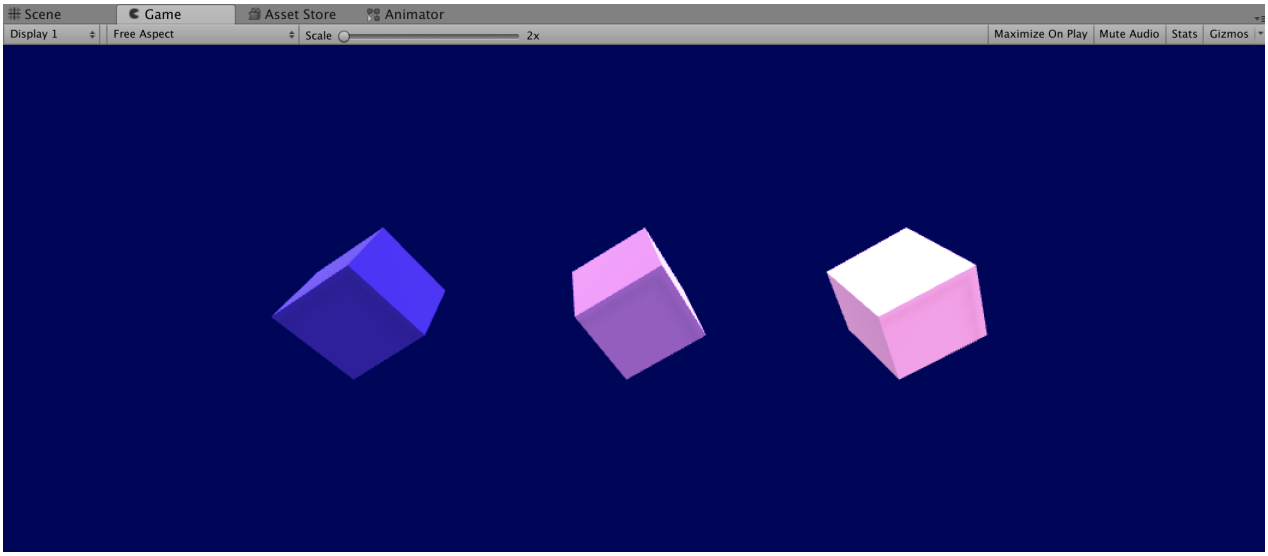
Finally, the color of the entire image is incrementally degraded with each successive collision, which is achieved with the `Int Out` node.

In other words, as the central cube tumbles and falls, the collisions encourage further tumbling (through `Rigidbody Out`) and each successive collision degrades the color of the entire image.



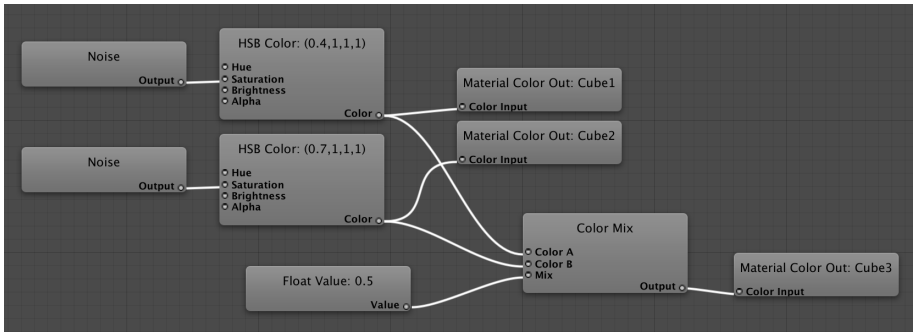
The above module takes the central cube's rotation and sends it to the floating cubes. The rotation is faithfully sent to cubes in the corners, and it is mixed through `Rotation Mix` before causing the upper middle cube to spin.

4.11 ColorCubes



Nodes Used: `Material Color Out, HSB Color, Float Value, Noise, Color Mix`

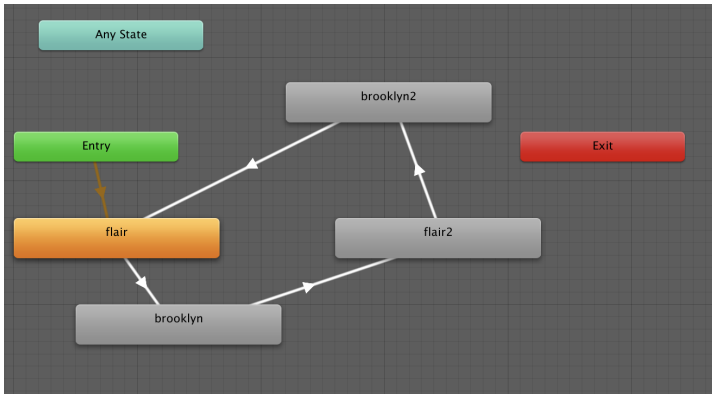
ColorCubes is a very simple scene consisting of three cubes. The outermost cubes have their colors manipulated by pseudorandom noise (`Noise`). The central cube's color is created by mixing the colors of the two outermost cubes (which are continuously changing, through `Color Mix`).



4.12 Dancer

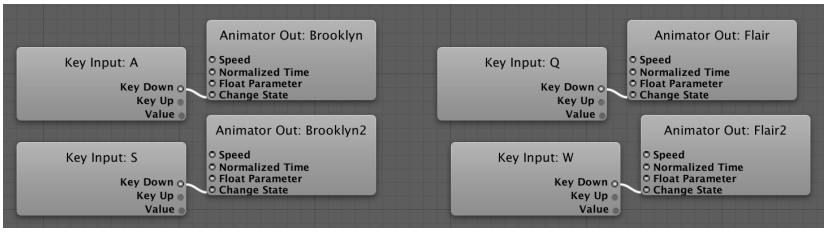
Nodes Used: `Key Input, Noise, Active Status Out, Animator Out, Float Filter`
Associated Tutorial: Animators
Associated Kino Image Effect: Bloom

The Dancer scene consists of a colorful and dynamic environment where a human mannequin cycles through two dance moves: the Brooklyn and the Flair.[5] The color effects are achieved through a combination of Directed Light objects of different colors arranged in the scene, and the Photomatic Effect and Bloom scripts attached to the Main Camera. Animating the mannequin is done using the Animator and two FBX animation files downloaded from Adobe's Mixamo [5]. There are two main modules in the Klak patch: one for controlling the animation, and the other for manipulating the lights and colors.



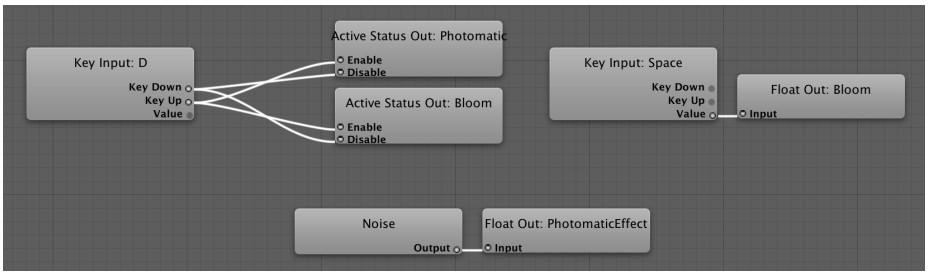
The animation sequence defined in the Animator, shown above, has two copies of the same FBX files: two copies of Brooklyn and two copies of Flair.

[5]The commonly-used spelling for the dance move is "flare." However, on Mixamo the animation is saved under the name "flair." Because of this, for consistency with the file names used in Unity, the spelling "flair" will be used.



Using the above module, it is not possible to repeatedly restart the same animation using the same `Key Input` to `Animator Out` flow sequence. For example, if the A keyboard key is pressed, the animation sequence will go to the Brooklyn animation. However, pressing A repeatedly does not create a glitchy or scratchy effect. To overcome this, two of the same animation files were used in the sequence: Brooklyn and Brooklyn2, and Flair and Flair2. The scratchy effect is then achieved by repeatedly pressing the two keys associated with the animation. In the case of Brooklyn these are the A and S keys, and in the case of Flair, it is the Q and W keys.

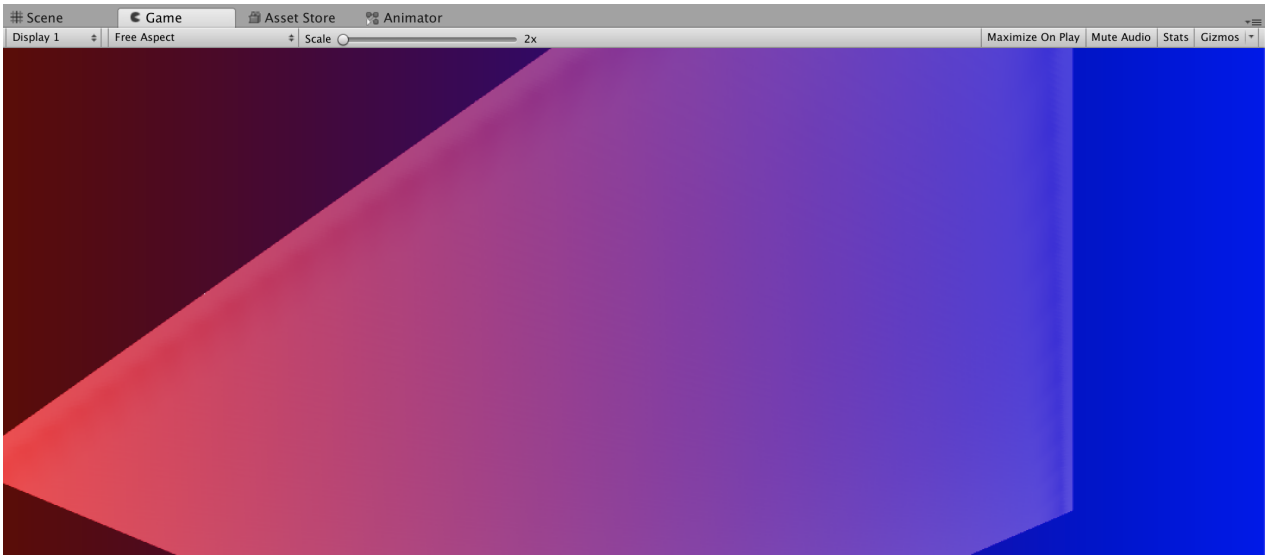
The four `Key Input` nodes in the above module each produce a bang when one of A, S, Q, and R is pressed. This action sends a bang out of the Key Down outlet of the appropriate `Key Input` node and into the Change State inlet of `Animator Out`. In the `Animator Out`'s inspector, the string-valued entry called Change State To has the same state name as the desired state name in the Animator.



Dancer's second module controls the colors and lights. A `Key Input` node sends bangs to two `Active Status Out` nodes when the D keyboard key is pressed and released, turning the Main Camera's Photomatic and Bloom effects off and on, respectively. Another `Key Input` node sends a value of 0.5 when it is pressed, and 0 otherwise, to the Main Camera's Bloom effect. The consequence of the former is that all color and light effects are stripped away, revealing the mannequin in full detail. The consequence of the latter is that the brightness becomes concentrated on the dancer.

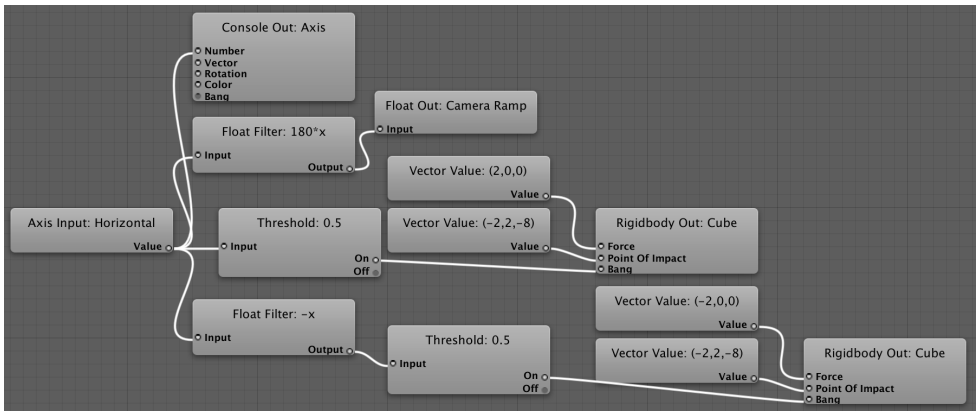
Lastly, `Noise` sends a continuous stream of pseudorandom values to the hue of the Main Camera's Photomatic effect. This causes the colors on the scene to drift.

4.13 Ramp



Nodes Used: `Float Out`, `Float Filter`, `Axis Input`, `Vector Value`, `Rigidbody Out`, `Console Out`, `Threshold`
Associated Kino Image Effect: Ramp

In the Ramp scene, either a joystick or the keyboard left and right keys can be used (`Axis Input`) to move the post-processing Ramp effect's angle. However, doing so also causes the object in the middle of the scene to begin rotating very subtly. If one direction of rotation is sustained for too long, the object will start to move too far. Because most the patch's functionality is present in other examples, attention is given only to the `Threshold` node.

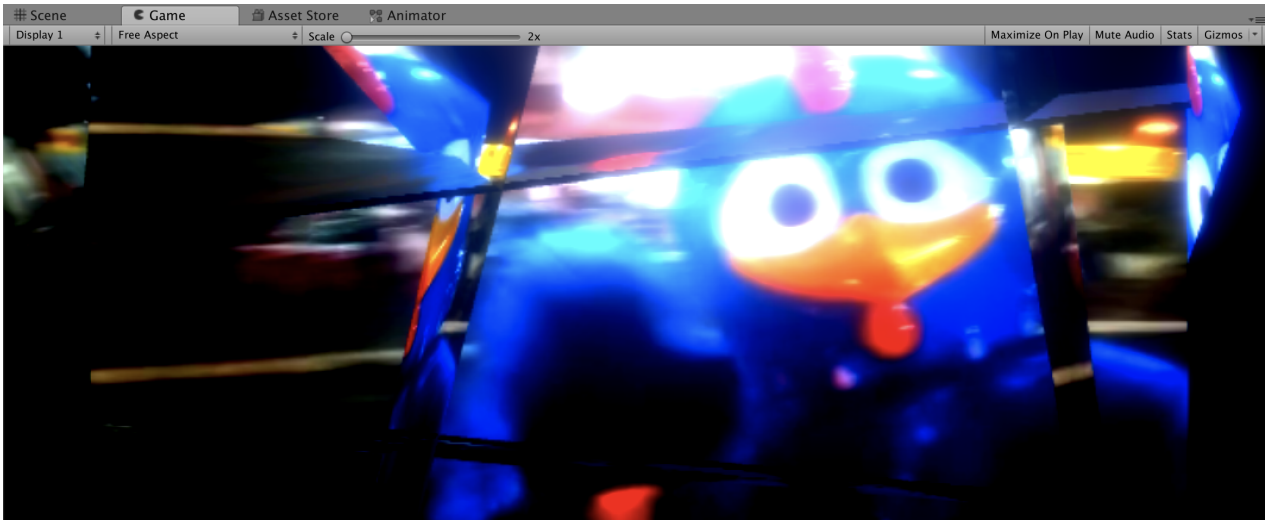


First, note that the `Console Out` node provides no function to the patch except for displaying data in the console window. This node is crucial for debugging.

`Axis Input` produces a float value between -1 and 1. When the left arrow key is pressed (or the joystick is pushed to the left), the node's output values tend to -1, and when it is pushed to the right, the values tend to 1. Hence, the first `Threshold` is activated only when `Axis Input` is pushed to the right long enough for it to produce values above 0.5. The other `Threshold` is activated only then `Axis Input` is pushed to the left, since these values first get multiplied by a -1 in `Float Filter`.

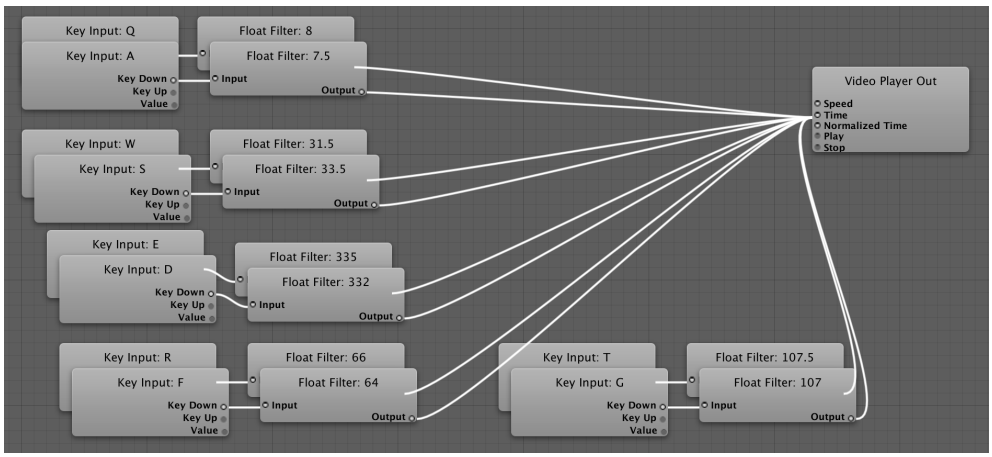
Once these thresholds have been met, a bang gets sent to `Rigidbody Out` nodes to push the object, using a similar impact mechanism involving `Vector Value`s as the Collider example.

4.14 Video

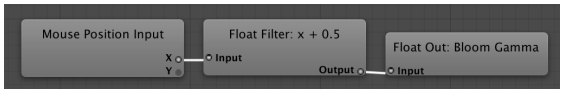


Nodes Used: `Key Input, Float Out, Float Filter, Mouse Position Input, Video Player Out`
Associated Tutorial: Video Player
Associated Kino Image Effect: Bloom

The Video scene displays a video by the Brazilian street dancing group Carreta Furacão. Various keyboard keys allow the user to jump to specific timestamps of the video. Notice in the Klak patch how the Key Inputs come in doubles. This is to allow the user to jump back and forth, creating a glitch effect, between very similar timestamps.



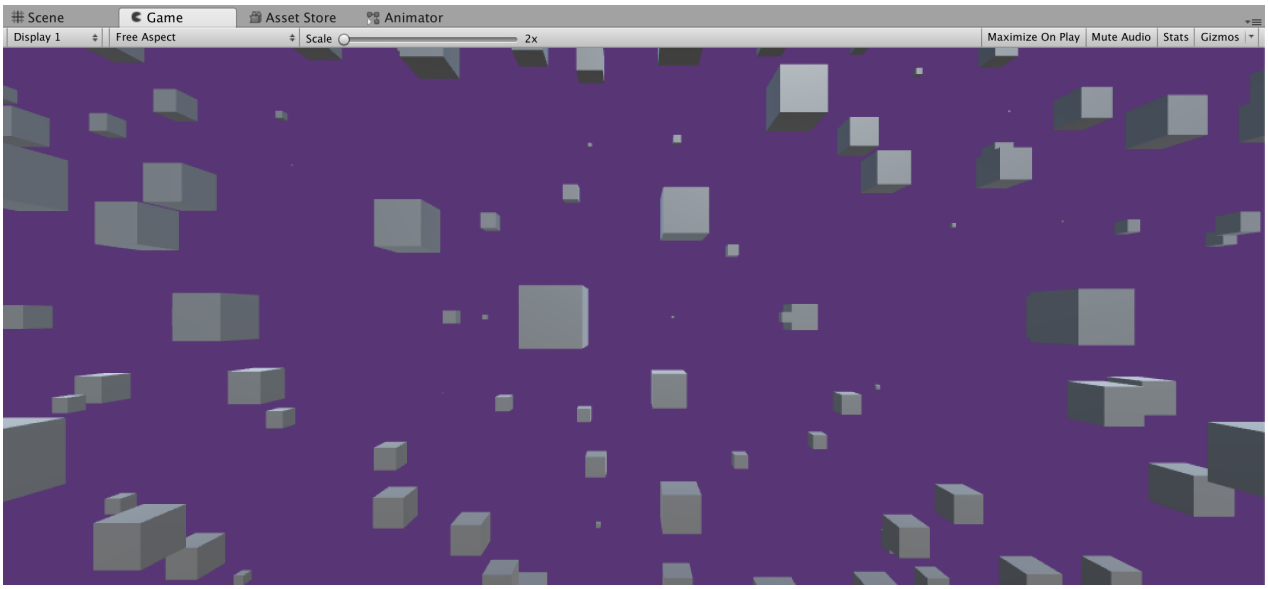
The above module is very intuitive. Each of the keyboard inputs (`Key Input`) first bangs a `Float Filter` so that the bangs become float-valued. More specifically, they become timestamps. Then, these float values simply go into the time inlet of `Video Player Out`, which controls the Carreta Furacão video.



The last module in Video picks up the mouse's x-axis position (`Mouse Position Input`), scales it by adding 0.5, and then sending it to the Main Camera's Bloom image effect.

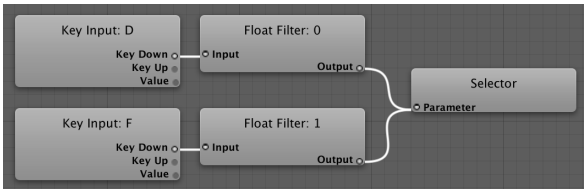
In other words, each pair of keys (Q, A), (W, S), (E, D), (R, F), and (T, G) cause the video to jump to very similar positions in the video. Moving the mouse in the x direction causes the image to bloom its highlights (left extreme of the mouse's x-axis) or to become dark (the right extreme).

4.15 Wall



Nodes Used: `Key Input, Mouse Position Input, Selector`

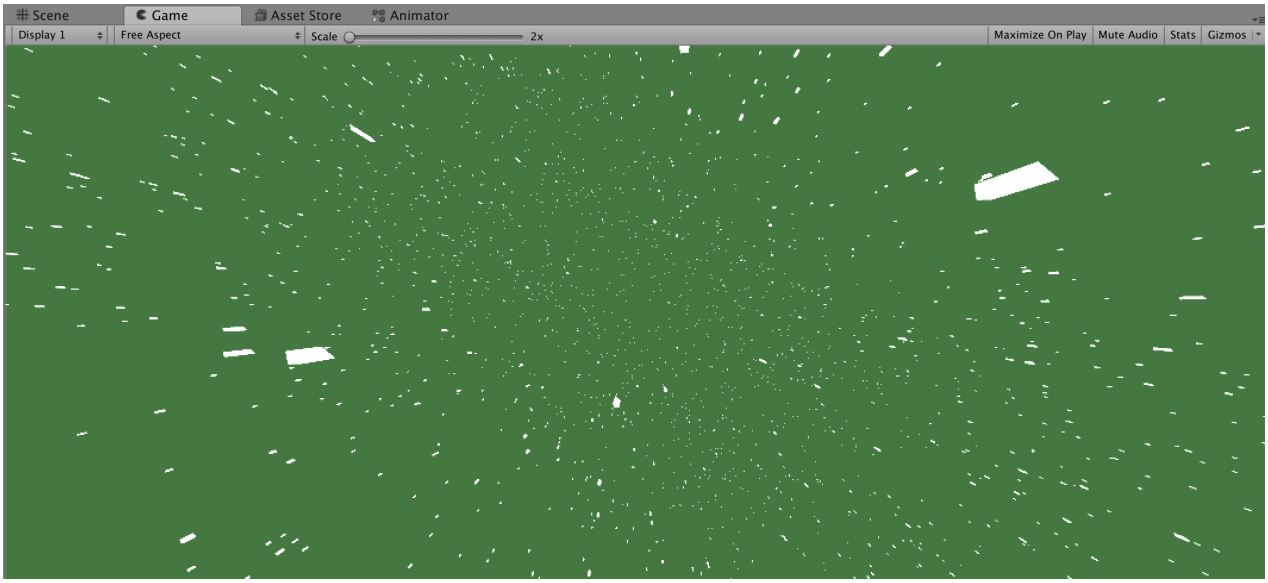
Wall is by far the simplest scene in the examples. There is a wall of moving cubes and two cameras. The Klak interaction consists only of switching between the two cameras, offering different points of view of the scene.



As stated above, this is simplest scene in the examples. There are two `Key Inputs`, listening to the D and F keyboard keys. When D is pressed, it activates the value 0 in the `Float Filter`, while F activates the value 1. Finally, these float values are sent to `Selector`, which simply selects one of two cameras in the scene, providing alternative vantage points.

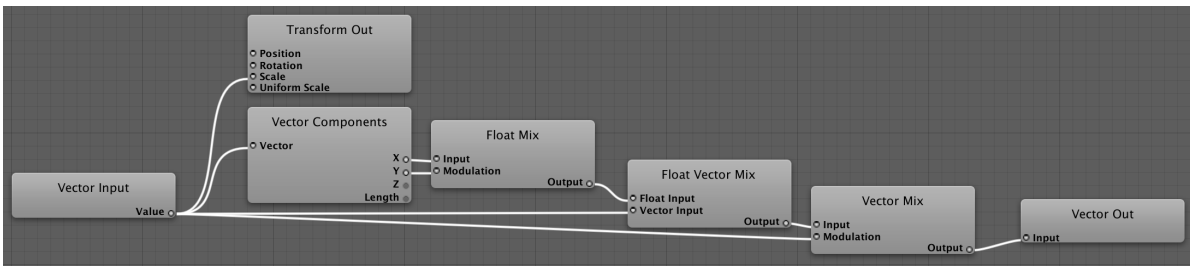
Notice how the cameras become activated and deactivated in the Object Hierarchy when the D and F keys are pressed.

4.16 Warp



Nodes Used: `Vector Input`, `Vector Out`, `Vector`, `Vector Components`, `Float Mix`, `Float Vector Mix`, `Vector Mix`

Warp resembles a space voyage in hyperdrive with twists and turns. It is composed of the Warp GameObject which on its own produces a starry image.



The above module uses the `Vector Input` node to take the Main Camera's position. This position is first sent to the scale inlet of the `Transform Out` node, which is associated with the Warp GameObject. This creates the sensation of the stars being stretched and pulled.

The initial position vector of the Main Camera is also decomposed through `Vector Components` to extract the x and y vector values. These values are added in `Float Mix`, before being used to scale the original position vector (`Float Vector Mix`). Finally, this vector is then added to the original position vector (`Vector Mix`) before being sent to alter the rotation scale of the Brownian Motion script of the Main Camera.

Chapter 5

Short Tutorials

The tutorials in this chapter are split into two categories: those that utilize features within Unity (the first four tutorials), and the single tutorial that requires using software outside of Unity (the last tutorial, which uses Max MSP / Pure Data).

These tutorials are very short and to-the-point, and are meant to get the reader started using the explained features. There are many resources on the internet containing tutorials and exercises that utilize these features, and the reader is encouraged to study these features more deeply, and to practice creating patches that make use of them.

5.1 Particle Systems

Particle systems are like geysers: they have a source from which particles emanate. The user has control over the shape of the particles and many of their features, including the amount of particles, their durations, the rate at which they are emitted over distance and over time, and many others. Wondrous objects can be created using particle systems; choosing the right shape and adjusting the properties can create anything from a field of grass to a nebula.

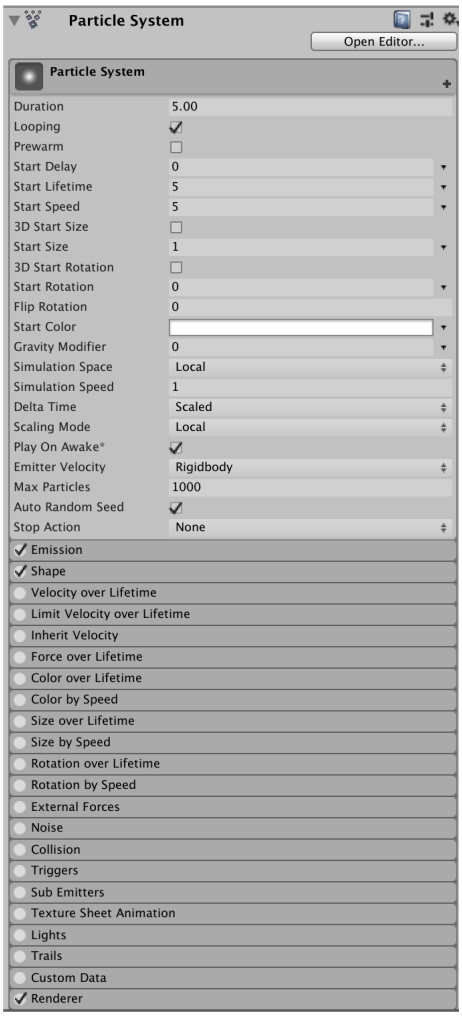
From the Unity Manual [11]:
"A particle system simulates and renders many small images or Meshes, called particles, to produce a visual effect. Each particle in a system represents an individual graphical element in the effect. The system simulates every particle collectively to create the impression of the complete effect."

In other words, particle systems allow the user to create a "fog" made of images or meshes. The user has extensive control over the properties of the individual components making up the fog.

Create a new scene and set the Clear Flags option of the Main Camera to Solid Color. Create a particle system by either right clicking in the Hierarchy and selecting Particle System from Effects, or by clicking GameObject from the Unity menu and selecting Particle System from Effects. Make sure the particle system's transform position is set to $(0, 0, 0)$. When the scene is played, a simple geyser-type system of particles should be emanating and shooting upwards.

WARNING: Make sure play mode is ended before making changes that are intended to persist.

With the particle system GameObject selected in the hierarchy, the system's inspector will be displayed. This is where specific changes to the system's properties can be made.



The Particle System's properties can be found in the inspector, as shown on the left.

This is a short tutorial meant to get the reader started with particle systems. As such, most details are omitted and the reader is encouraged to explore the numerous particle system properties. Hovering the mouse cursor over the name of the property in the inspector displays a short description of the property.

For example, the description for the Gravity Modifier says "Scales the gravity defined in Physics Manager." Change the Gravity Modifier from 0 to 1, and press play. What was before a geyser-type system shooting upwards, with the particles never making their way back down, is now affected by gravity: the particles barely make their way upwards before falling back down. Can the gravity be negative?

Note: The little button with a downward arrow to the right of some properties allows the user to change the input from a constant value, to a random value. In some cases, the value can come from a curve, or between two curves.

For another example, in the system's inspector, open the emission module, and set the Rate Over Time to 100. This causes the system to emit a much larger amount of particles only dependent on time. On the other hand, as it stands, changing the value of Rate Over Distance has no effect. This is because Rate Over

Distance only affects the emission of particles as the system moves in space.[1]

Altering the look of the particles is done through the renderer module (the last module in the system's inspector). Open this module and change the default material "Default-Particle" either by dragging a material object to the material slot, or by clicking the small circle to the right of the slot and selecting an available material.[2]

Particle systems have many other properties under many other tabs. The small gray circle to the left of the tab names indicate whether the tab and its properties are active (a small checkmark appears when the tab is active). The reader is encouraged to explore these tabs and properties, and to come up with a rudimentary (or complex!) particle system.

Once a working particle system is in place, create a Patch GameObject by right-click inside the hierarchy and selecting Patch from Klak (this can also be done by clicking GameObject on the Unity menu). With the patch object selected, open the Patcher through the object's inspector. This will create a window for the patcher that can be placed as a tab in another set of windows within the Unity environment.

This tutorial will give a brief explanation of how certain Klak nodes can be used to manipulate particle systems. These nodes are `Float Out`, `Event Out`, and `Particle System Out`.[3]

Using `Float Out` with particle systems, the user is able to manipulate the following float-valued particle system properties: time, start delay, playback speed, emission rate, start speed, start size, start rotation, start lifetime, gravity modifier, and others.With `Event Out`, the user can effect "bang" style properties, such as pause. Finally, with the `Particle System Out` node, the user is able to change the rate over distance and the rate over time, as well as cause the system to emit more particles.

Some helpful tips:

Changing simulation space from local to world is useful when the particle system is attached to a moving object. In local, the particle system is always following the object. In world, the particle system generator is following the object, but the position of the particles and the way they fall (or rise) is dependent on where they were generated in the world.

For start color, the default is a single color. By clicking the small downward arrow, a gradient or one of several random options can be chosen instead.

Velocity Over Lifetime module: particles can have their velocity defined in terms of a specific vector, or direction, over time. This way, instead of shooting straight up (like the default geyser), the particles can move in other directions. Additionally, other properties such as rotation about the center of the system (formally called Orbital, which can create spiral and cylindrical effects) is defined in this module.

The Emission module defines the emission of the particles. Particles can spawn over time, or over distance. Additionally, particle bursts can wait for specific triggering events.

[1] One way to do this is to attach a script defining motion to the particle system.

[2] There are many free assets with materials made especially for particle systems, which are available in Unity's Asset Store. The reader is encouraged to search, discover, and load such assets into Unity to explore creating particle systems.

[3] Other Klak nodes can have interesting effects on particle systems and the reader is encouraged to explore these, some of which are `Bool Out`, `Color Out`, `Int Out`, `Rotation Out`, `Transform Out`, `String Out`, and `Vector Out`.

A sample of commonly manipulated properties: duration, lifetime rotation, shapes, and speed.

5.2 Playable Director and Timelines

Associated Example: Bounce

Playable Director and Timelines let the user control animations that are created by defining static properties of the animations at given periods of time. For example, to simulate a bounce, a sphere could have position (0,1,0) at time 0, then (0,0,0) after 1 second, and then back to (0,1,0) after another second. Additionally, its scale can be altered to reflect the deformation that happens when the sphere hits the ground, which in this case would be the point (0,0,0). Unity will then, as smoothly as possible, animate the sequence.

From the Unity Manual [12]:

> The Playable Director component stores the link between a Timeline instance and a Timeline Asset. The Playable Director component controls when the Timeline instance plays, how the Timeline instance updates its clock, and what happens when the Timeline instance finishes playing.

The Timeline Asset stores the information about the movements, deformations, and other forms of animation that are defined along the timeline. The Timeline instance is a particular manifestation of the animation. The bridge between the two is the Playable Director component, which tells the Timeline instance how to behave according to the information recorded in the Timeline Asset.

Create an empty GameObject, recenter its transform position if necessary, and rename it to Timeline. In the Window option of the toolbar, click Sequencer, then Timeline. Place the Timeline window where most convenient in the Unity workspace. A suggestion for how to organize the Unity workspace is given in the introduction.

With the Timeline window in view, click the empty GameObject, revealing the option (a "Create" button in the Timeline window) to create a Director component and a Timeline asset. After choosing a name and location, this will automatically create the objects for you and the empty GameObject is automatically assigned to them. The empty GameObject now has two components, the Playable Director (which references the timeline object you just created) and the Animator, which is verifiable in the empty GameObject's inspector.

In the Playable Director component of the empty GameObject, we can control a few properties, such as *Play on Awake*, and whether the animation should loop or play once. The Animator component is not needed (but can be useful, which is currently not discussed in this documentation). In light of these, for the purpose of this short introduction to Playable Director and Timelines, the Animator component can be removed from the empty GameObject.

With the empty GameObject selected (which is named Timeline), in the Timeline window, the individual timeline for the GameObject can be deleted. This is because the empty GameObject is not being animated, it is being used as an organization scheme to control *other* animations. An alternative to all of this is to create the Timeline Asset in the object that will be animated. The advantage of doing this with the empty GameObject comes when managing multiple animations, since this method is a great for keeping track of all

of the timeline-style animations being created. So, delete the individual timeline for the GameObject.

At this point there are no longer any tracks. The simplest way to go forward from here involves dragging the GameObject that will be animated into the empty track list in the Animator window, and then selecting "Add Animation Track." Create a Sphere object and drag it into the Timeline window, making sure that the empty object named Timeline has been selected.

Now, click the record button. Here, the GameObject that is tied to the timeline can be manipulated to build an animation. This is done by dragging the tab that indicates the frame (alternatively, seconds could be selected by clicking the gear on the upper-right corner of the Timeline window). Drag the tab to the frame you would like to change properties of the object (such as position, rotation, and scale), and change the properties as desired. Create several of these to define the animation.

To later edit the specific frames (or seconds), double click on the timeline, which opens the Animation window, where the specific frames/seconds keys can be changed.

Several animations can be added to the same timeline (or separate timelines can be created) by following a similar process as described above.

A common effect that can be created with Playable Director and Timeline is to create a bounce or a wobble. Unity will take care of extrapolating the in-between frames.

If the animation is being looped, it is important that the last frame defined matches with the first frame. In this way, there won't be a discontinuity between the first and last frames of the loop.

The `Playable Director Out` node, allows the user to manipulate the speed of the animation, jump to a specific time of the animation (either objective time or normalized time, where the entire length of the animation is normalized to a length of 1 and real number values between 0 and 1 can specify a relative point in time in the animation), and to play and stop the animation. More information can be found in the section of `Playable Director Out`.

It is also possible to, in the timeline window, convert the animation with they key frames defined by right clicking and selecting "convert to clip track" (it's still changeable in the animator window). Now the user is able to treat the entire animation as a single track, including the possibility to move the animation around (useful for when there is more than one), copy and paste it, and change the length in time of the animation.

Here the user has additional options, for example, Animation Extrapolation, where the user can select "ping pong" (among others) which will make the animation loop back and forth. This is important if in the same timeline there is a clip track that goes longer than the selected clip track, so that the shorter clip does not end and stay still, unless, of course, this is desired.

It is possible to group the tracks, by clicking Add (in the top-left corner) and then Track Group. This allows the user to organize the timelines (for example, the objects that wobble, the objects that bounce, and even have a Camera group using Activation Tracks).

Activation Tracks can be used with several objects such as cameras. In this situation two or more cameras can be used, and a timeline can be made to activate and deactivate certain cameras.

5.3 Video Player

Associated Example: Video

Video Player allows the user to use video files in the Unity scene. These files are played on GameObjects such as spheres, cubes, or others.

From the Unity Manual [18]:

> Use the Video Player component to attach video files to GameObjects , and play them on the GameObject's Texture at run time.

First, a video file in an appropriate format is needed. The type of video file depends on the operating system being used. More details can be found in the Unity manual [19]. Additionally, the GameObject that will play the video must be added to the scene.

Once the video file is ready, and the GameObject has been created, a Video Player Object must be created in the Object Hierarchy. This is done by selecting the GameObject option of Unity's menubar, then selecting Video followed by Video Player. Now, the video file must be dragged into or chosen in the Video Clip component option in the Video Clip object's inspector.

Lastly, changing the Render Mode in the Video Clip object's inspector to Material Override allows using a GameObject to display the video. Drag the GameObject that will be displaying the video into the Renderer source in the Video Player Object's inspector. The scene is ready to play the video file on the GameObject!

5.4 Animators

Associated Example: Dancer

Animators are used to connect animation files and are behind many interesting applications. This short tutorial focuses on creating Unity projects that shift animation states. To achieve the goal of the tutorial, several FBX files are needed (at least two).[4]

Once two or more FBX files have been downloaded, to keep the project organized, in the Assets folder, create a folder for the animations (e.g., call it "Animations"). Drag the FBX files from wherever they are stored on the computer into the newly created folder.

Notice that clicking on the arrow on the mid-right edge of the FBX file in the Unity folder reveals several subfiles. The most important subfile that will be used in this tutorial is that with a rectangle containing a play button in its center. Remember that these are the files that will be used to create the animation sequence.

Drag one of the downloaded FBX files into the Object Hierarchy. Notice how if the scene enters play mode, the animation does not move. This is because the GameObject for the FBX file does not have an Animator Controller.

Right click in the same folder to create an Animator Controller. Give it a name. With the object selected in the Object Hierarchy, in its inspector, under the Animator component, an empty controller is revealed. Drag

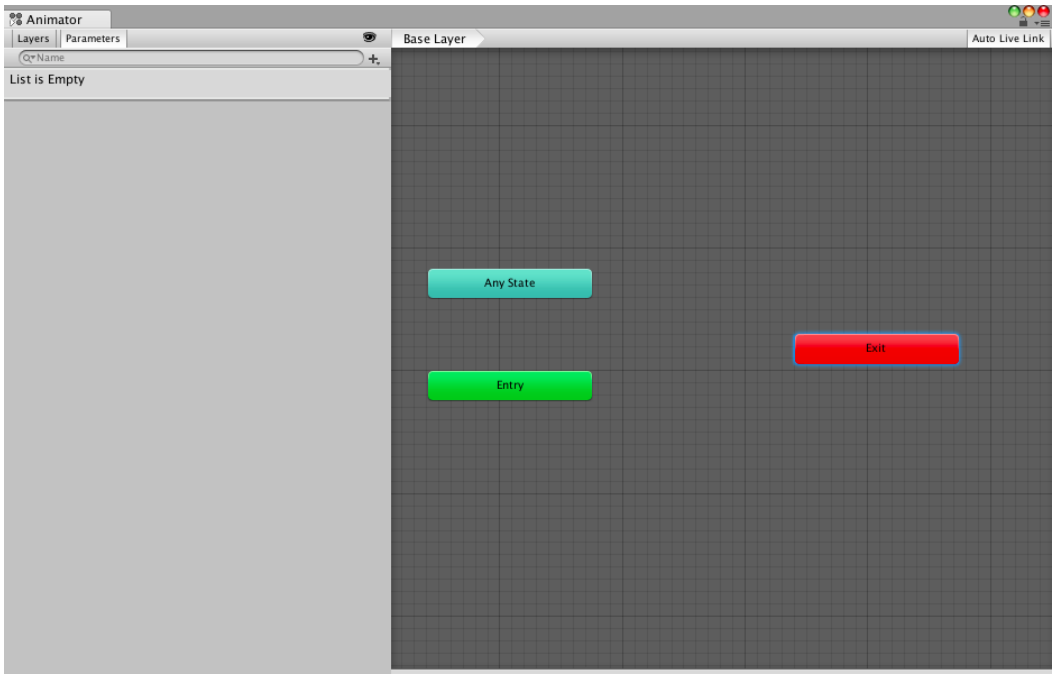
[4] A great website for this end is Adobe's Mixamo: https://www.mixamo.com/

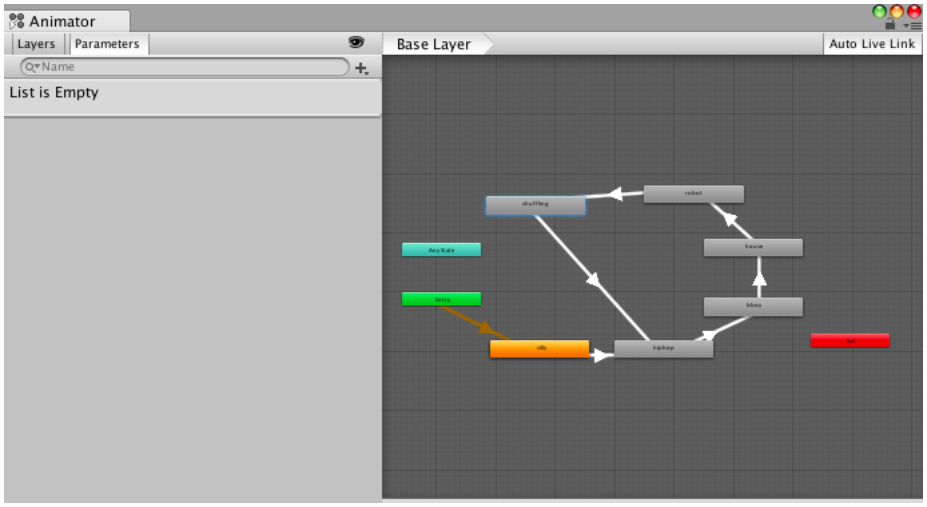
the controller that has just been created into this slot.

If the Unity workspace does not already have an Animator window, go to Window>Animation>Animator. Place this window where convenient, (recommendation: place it as a tab next to Scene, Game, and Asset Store, as shown in the introduction). In this space, a network of transitions between animations can be created. This space is where the rectangles with play buttons will be placed. Each of these separate animations are called *states*. Transitions can be created from one state to another. Doing so and entering play mode will result in the animation following these transitions.

An important thing is to name each of these states and give them tags. This will allow Klak to access the animation states and jump from state to state. Connecting the states in a loop allows the animation to run indefinitely, which may be desired. Klak can then be used to jump between states in the loop. Otherwise, Klak can still jump between states, but the animation will end once the state ends.

The first image below shows a newly created Animator Controller, void of any states and transitions. The image below it shows an Animator Controller with several states and a loop.





Although it may be hard to read, one of the above states is named "bboy," which is an animated breakdance sequence downloaded from Adobe's Mixamo. To access these states with Klak, an Animator Out node must be created and receive a bang input in its Change State inlet. The bang is what will trigger the change in animator states. In the node's inspector, select the animator created during the exercise in the empty animator slot. Still in the inspector, in the string box Change State To, write the name of the state (in this example: "bboy").

The intention of this short introduction is to get the reader started in creating animator sequences and accessing the states therein with Klak.

5.5 Pure Data and Max MSP

Pure Data (PD) and Max MSP (Max) are visual programming environments for creating interactive computer music and multimedia installations. Both were created by Miller Puckette. PD is open source and free, and Max is paid. There are good reasons to use either of them. Due to their visual nature, they are similar to the Klak patching environment in Unity both visually and logically. Please note that if Videolab is being used with the OP-Z and iPad, then, it may not possible to use PD or Max.

PD, Max, and Videolab all fascilitate interactive computer music. As such, it can be useful to use them together. A person with high levels of expertise using Unity and scripting may be able to extend Klak's functionality internally. For others, however, it may be easier to use PD or Max to enhance the capabilities of your Videolab creations.

The focus of this short section is on MIDI manipulation. Although Klak has a node for `Note Input`, it does not receive input from the Pitch Bend parameters of MIDI controllers. A way to extend `Note Input` to incorporate this functionality is to rework the scripts behind Klak. Another way is to build a PD or Max patch that receives the Pitch Bend information and relays it to Unity.

The example of using Pitch Bend is easy enough. In Max, the input node is "bendin" and the output node is "ctlout n," where n is the appropriate knob number that will be communicating with Unity's MIDIJack, which comes with the Videolab package. Max's "bendin" node knows to listen to MIDI Pitch Bend data. The "ctlout n" node stands for *control out*, which sends MIDI control messages (i.e., knob messages), rather than standard MIDI data. This distinction is important because Pitch Bend relies on a MIDI feature called aftertouch, which allows continuous streams of values to be sent through the communication channel (which

is not possible with the standard MIDI-Out port).

In the image below, the intermediate node displays the number (that represents the pitch bend), and the channel number used is 7. The 7 was chosen arbitrarily. Whatever number is chosen must be reflected in the `Knob Input` node's inspector, where the Knob Number is selected. Make sure that "Is Relative" is not selected. It is also best to select "Direct" as the option for interpolator, since the pitch bend already comes in a continuous stream that is sensitive to small changes.

The value in Unity is then received in `Knob Input`, and is output through the node's "Value" output as a real number between 0 and 1.

This is a very simple example, but very powerful, as now the pitch bend data can be used to further manipulate the Unity scene. Additional possibilities integrating Max and PD with Unity and Videolab include using Wii controllers and Mocap. More details on Max and PD are unfortunately beyond the scope of this document. The internet, however, is full of resources for those wanting to learn and add this extra layer to their Unity MIDI projects.

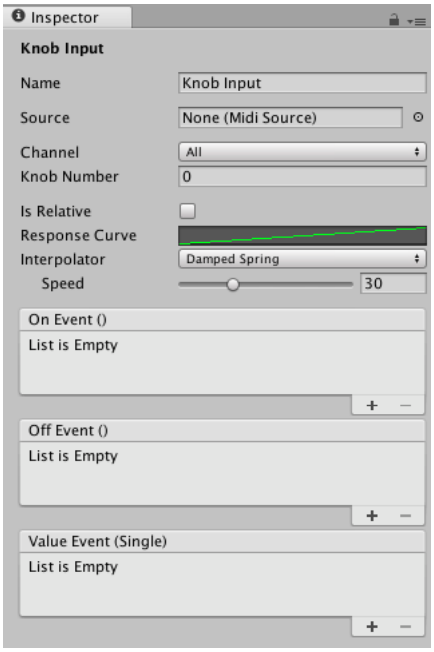
Appendix A

Input

A.1 Input > MIDI > Knob Input

Description: `Knob Input` listens to a specified MIDI Channel (or all channels) and knob number for knob change sequences. The MIDI channel can be manipulated via the inlet (Channel) or pre-defined in the inspector. When a MIDI knob is turned and picked up by `Knob Input`, through the outlets, the user may trigger a bang when the knob is activated (On) or deactivated (Off), and propagate the numerical value of the turning knob (Value).

Inlet: Channel (float)
Outlets: On (bang), Off (bang), Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Source - The MIDI source is specified by dragging the MIDI Source Object in the object hierarchy to Source in the node's inspector, or by clicking the small circle (to the right) and selecting the object. Note that a MIDI Source Object must exist in the object hierarchy. If there is only one MIDI source connected to the computer, this feature is automatic (so the existence of the object is not required). In other cases, it is useful for restricting which MIDI source should be interpreted for the node.

Channel - The MIDI channel may be altered during runtime by sending floats to the Channel inlet of the Knob Input node. Alternatively, it may be specified in the inspector. There are 16 channels and an option for all channels. Note that the Channel inlet receives floats rather than integers. The node rounds floats up and all values below 1 get mapped to 1, and above 16 to 0.

Knob Number - An integer between 0-15 must be chosen to reflect the desired knob.

Is Relative - If this checkbox is activated, the knob values are relative to their starting point upon turning. Otherwise, they are absolute values.

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

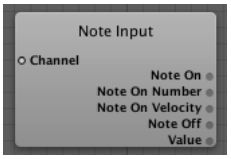
Lists - Summary of the node's output connections.

Examples:

MIDIKlak [2]: Knob Event (Trigger), Knob Event (Value)

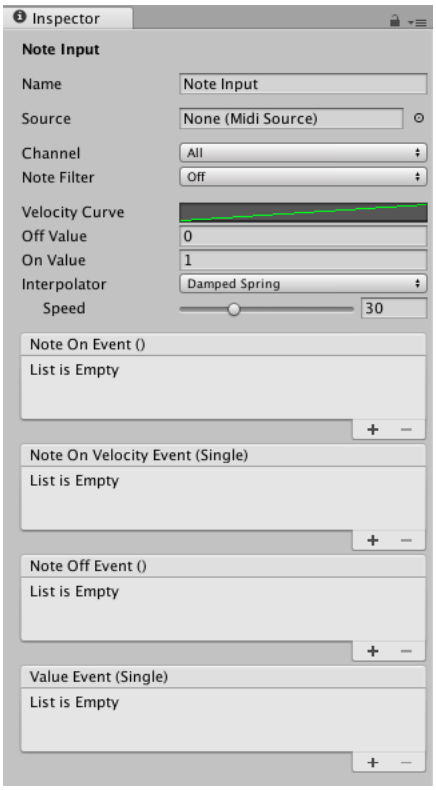
YouTube: Ninety Six, Milo

A.2 Input > MIDI > Note Input



Description: `Note Input` listens to a specified MIDI Channel (or all channels) for note-on and note-off messages. The MIDI channel can be manipulated via the inlet (Channel) or pre-defined in the inspector. When a MIDI note is played and picked up by `Note Input`, through the outlets, the user may trigger a bang when the note is pressed (Note On) or released (Note Off), propagate the numerical MIDI note value when the note is pressed (Note On Number), propagate the numerical MIDI note velocity when the note is pressed (Note On Velocity), and finally, send float values that are pre-defined in the inspector both for when the note is pressed (On Value) and released (Off Value).

Inlet: Channel (float)
Outlets: Note On (bang), Note On Numer (float), Note On Velocity (float), Note Off (bang), Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Source - The MIDI source is specified by dragging the MIDI Source Object in the object hierarchy to Source in the node's inspector, or by clicking the small circle (to the right) and selecting the object. Note that a MIDI Source Object must exist in the object hierarchy. If there is only one MIDI source connected to the computer, this feature is automatic (so the existence of the object is not required). In other cases, it is useful for restricting which MIDI source should be interpreted for the node.

Channel - The MIDI channel may be altered during runtime by sending floats to the Channel inlet of the Note Input node. Alternatively, it may be specified in the inspector. There are 16 channels and an option for all channels. Note that the Channel inlet receives floats rather than integers. The node rounds floats up and all values below 1 get mapped to 1, and above 16 to 0.

Note Filter - The user may impose a note filter, which is done in the inspector. If the filter is off, any played note will activate Note Input. Otherwise, the option of Note Name allows the user to specify which note (C, C#, D, D#, E, F, F#, G, G#, A, A#, B) will activate Note Input, and Note Number allows the user to specify a range of MIDI note numbers which will activate the node.

Velocity Curve - The curve maps the velocity value to a user-defined curve.

On Value - This value is pushed out of the Value outlet when the specified note (or any note when unspecified) is pressed.

Off Value - Same as On Value but when the note is released.

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

Lists - Summary of the node's output connections.

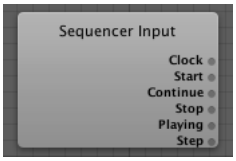
Examples:

MIDIKlak [2]: Knob Event (Trigger), Knob Event (Value)

YouTube: Ninety Six, Milo

VideolabTest-master [2]: Adam, Boing, Peeler, Poly, Primitive, Sphere, Splash, Stripe, Text1, Text2, Tilt Brush, Trail, Worm

A.3 Input > MIDI > Sequencer Input



Description: Details on the `Sequencer Input` node are omitted from this documentation because:

1. This node requires a MIDI sequencer (such as the OP-Z).
2. The author does not own a MIDI sequencer.

Inlets: None
Outlets: Clock (bang), Start (bang), Continue (bang), Stop (bang), Playing (float), Step (float)

Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

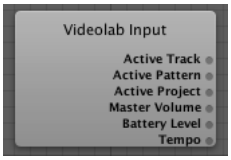
Source - The MIDI source is specified by dragging the MIDI Source Object in the object hierarchy to Source in the node's inspector, or by clicking the small circle (to the right) and selecting the object. Note that a MIDI Source Object must exist in the object hierarchy. If there is only one MIDI source connected to the computer, this feature is automatic (so the existence of the object is not required). In other cases, it is useful for restricting which MIDI source should be interpreted for the node.

List - Summary of the node's output connection.

Example:

YouTube: Andy

A.4 Input > MIDI > Videolab Input



Description: Details on the `Videolab Input` node are omitted from this documentation because:

1. This node requires the OP-Z sequencer and synthesizer.
2. The author does not own the OP-Z sequencer and synthesizer.

The readers are encouraged to fill this page with necessary details. More information can be found on the teenageengineering GitHub wiki.

Inlets: None
Outlets: Active Track (float), Active Pattern (float), Active Project (float), Master Volume (float), Battery Level (float), Tempo (float)

Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

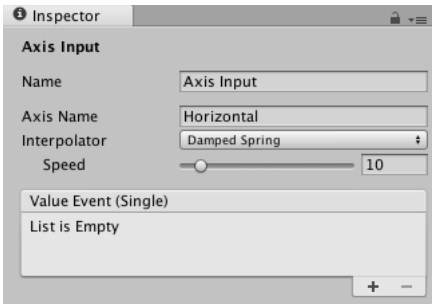
Source - The MIDI source is specified by dragging the MIDI Source Object in the object hierarchy to Source in the node's inspector, or by clicking the small circle (to the right) and selecting the object. Note that a MIDI Source Object must exist in the object hierarchy. If there is only one MIDI source connected to the computer, this feature is automatic (so the existence of the object is not required). In other cases, it is useful for restricting which MIDI source should be interpreted for the node.

List - Summary of the node's output connection.

A.5 Input > Axis Input

Description: `Axis Input` can be thought of as a joystick input, with capabilities of capturing directional keyboard or mouse movements. The default string in Axis Name is *Horizontal*, which outputs a -1 at the left horizontal extreme of the joystick (or keyboard direction keys) and all in between values, with 0 for the neutral position, and +1 for the right horizontal extreme. Other options for Axis Name are *Vertical*, *Mouse X*, and *Mouse Y*. *Vertical* captures the vertical joystick movements, with extremes of vertical down being -1 and vertical up being +1. *Mouse X* and *Mouse Y* are analogous to *Horizontal* and *Vertical*, but map the extreme mouse locations to the respective values.

Inlets: None
Outlet: Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Axis Name - The options for Axis Name are *Horizontal*, *Vertical*, *Mouse X*, and *Mouse Y*.

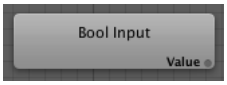
Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

List - Summary of the node's output connection.

Example:

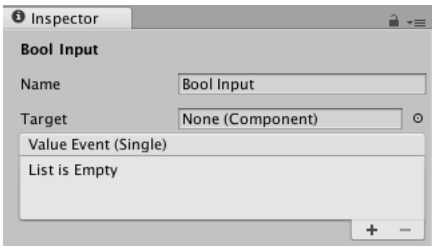
VL_Examples: Ramp

A.6 Input > Generic > Bool Input



Description: `Bool Input` receives a boolean value (0 or 1) from a selected GameObject and propagates it through the Value outlet. Once a GameObject is selected, a hierarchy is unveiled allowing one to select the desired object's boolean property. For example, if the selected GameObject is the Main Camera, the user is then able to select the specific component of the camera (Transform, Camera, or AudioListener), which then allows the user to select the desired property. Note that the functionality of this node is oftentimes not intuitive, and that hidden object properties may appear under the unveiled hierarchy, so exploration is encouraged.

Inlets: None
Outlet: Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected, from which the desired boolean property to be propagated is chosen.

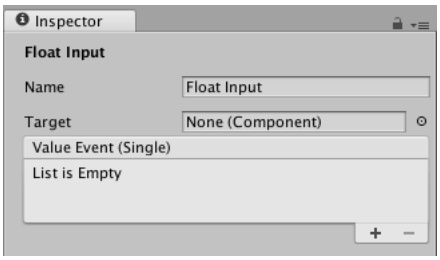
List - Summary of the node's output connection.

Example: Me1

A.7 Input > Generic > Float Input

Description: `Float Input` receives a float value from a selected GameObject and propagates it through the Value outlet. Once a GameObject is selected, a hierarchy is unveiled allowing one to select the desired object's float-valued property. For example, if the selected GameObject is the Directional Light, the user is then able to select the specific component of the light (Transform or Light), which then allows the user to select the desired property. Note that the functionality of this node can sometimes be non-intuitive, and that hidden object properties may appear under the unveiled hierarchy, so exploration is encouraged.

Inlets: None
Outlet: Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected, from which the desired float-valued property to be propagated is chosen.

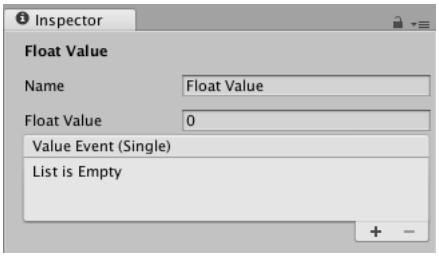
List - Summary of the node's output connection.

Example: Me1

A.8 Input > Generic > Float Value

Description: `Float Value` continuously outputs a user-defined float value through the Value outlet.

Inlets: None
Outlet: Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Float Value - Value that this node will continuously output through the Value outlet.

List - Summary of the node's output connection.

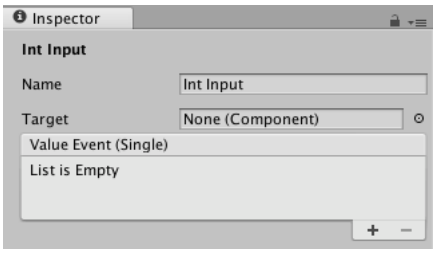
Examples:

VL_Examples: Bounce, ColorCubes

A.9 Input > Generic > Int Input

Description: `Int Input` receives an int value from a selected GameObject and propagates it through the Value outlet. Once a GameObject is selected, a hierarchy is unveiled allowing one to select the desired object's int-valued property. For example, if the selected GameObject is the Main Camera, the user is then able to select the specific component of the camera (Transform, Camera, or AudioListener), which then allows the user to select the desired property. Note that the functionality of this node can sometimes be non-intuitive, and that hidden object properties may appear under the unveiled hierarchy, so exploration is encouraged.

Inlets: None
Outlet: Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected, from which the desired int-valued property to be propagated is chosen.

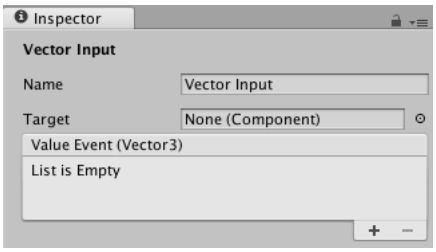
List - Summary of the node's output connection.

Example: Me1

A.10 Input > Generic > Vector Input

Description: `Vector Input` receives a 3D vector from a selected GameObject and propagates it through the Value outlet. Once a GameObject is selected, a hierarchy is unveiled allowing one to select the desired object's boolean property. For example, if the selected GameObject is a simple cube, the user is then able to select the specific component of the cube (Transform, MeshFilter, MeshRenderer, or BoxCollider), which then allows the user to select the desired property. Note that the functionality of this node is oftentimes not intuitive (sometimes it is!), and that hidden object properties may appear under the unveiled hierarchy, so exploration is encouraged.

Inlets: None
Outlet: Value (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected, from which the desired vector-valued property to be propagated is chosen.

List - Summary of the node's output connection.

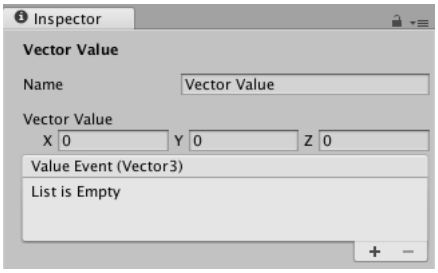
Example:

VL_Examples: Warp

A.11 Input > Generic > Vector Value

Description: `Vector Value` continuously outputs a user-defined 3D vector through the Value outlet.

Inlets: None
Outlet: Value (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Vector Value - User-defined vector value that the node will continuously output.

List - Summary of the node's output connection.

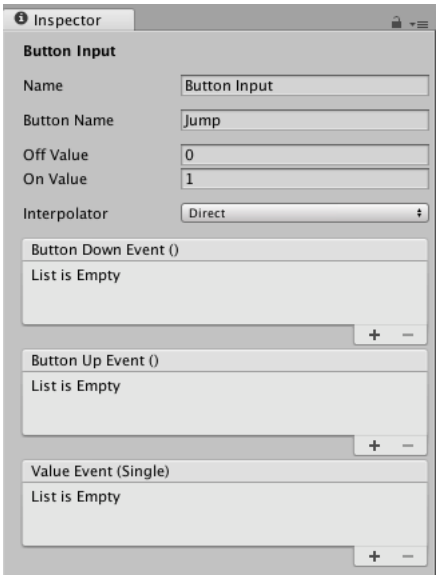
Examples:

VL_Examples: Collider, Ramp

A.12 Input > Button Input

Description: `Button Input` can be thought of as a joystick / gamepad input, with capabilities of capturing keyboard or mouse input. The default string in Button Name is *Jump*. For details on how the keyboard and mouse are mapped to button names, and to change these mappings (and the button names) according to the user's preference, the user should click Edit in Unity's menu, then select Input from Project Settings.

Inlets: None
Outlet: Button Down (bang), Button Up (bang), Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Button Name -

Off Value - User-defined value that is continuously sent through the Value outlet while the button is not being pressed.

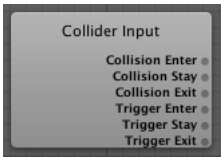
On Value - User-defined value that is continuously sent through the Value outlet while the button is being pressed.

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

List - Summary of the node's output connection.

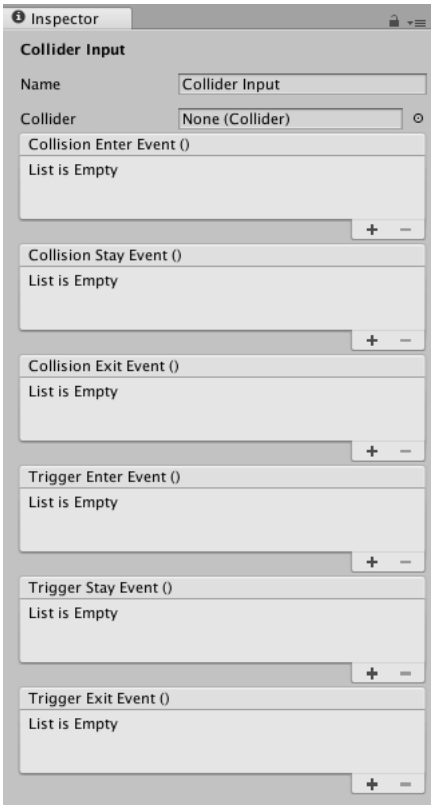
Example: Same as `Key Input`

A.13 Input > Component > Collider Input



Description: `Collider Input` receives collision events from GameObjects that have a collider component. For example, cubes hitting each other, or spheres bouncing on a plane. The node sends bang messages out of its various outlets, each for specific scenarios of the collision.

Inlets: None
Outlet: Collision Enter (bang), Collision Stay (bang), Collision Exit (bang), Trigger Enter (bang), Trigger Stay (bang), Trigger Exit (bang)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

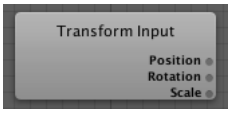
Collider - A target GameObject with a collider component must be selected, from which the collider events will be received and propagated through the chosen outlet(s).

List - Summary of the node's output connection.

Example:

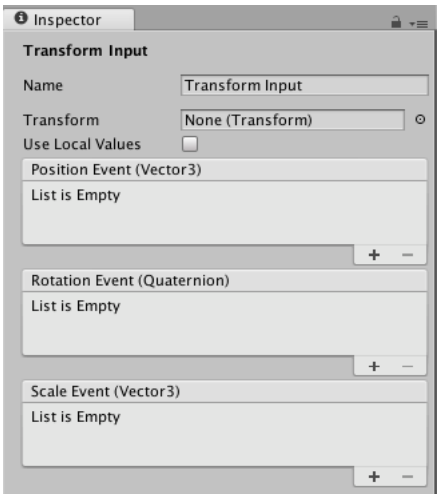
VL_Examples: Collider

A.14 Input > Component > Transform Input



Description: `Transform Input` receives information from the Transform properties of a GameObject: position, rotation, and scale. When the game runs, the node propagates the initial value through the wires and updates the value according to changes to the appropriate GameObject's transform data. There is an option to use local values, which is important only in the case of the selected GameObject being a child of a parent object. In such a case, if the option is selected, the values are calculated in relation to the parent object's transform value, and not in reference to the global scene. If the option is selected when the object is not a child, it produces the same values as when the option is not selected.

Inlets: None
Outlet: Position (vector3), Rotation (quaternion), Scale (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Transform - A target GameObject must be selected, from which the transform values will be received and propagated through the chosen outlet(s).

Use Local Values - Checkbox to select whether to use the global transform values or to use values relative to a parent object. If no parent object exists, using local values has no effect.

List - Summary of the node's output connection.

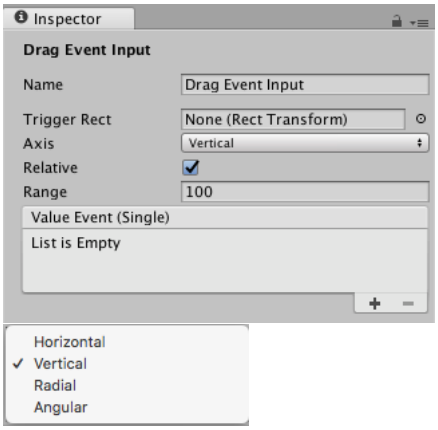
Examples:

VL_Examples: Bounce, Collider

A.15 Input > UI Event > Drag Event Input

Description: `Drag Event Input` receives float information from a UI element that is being dragged, with options for the direction of the movement: horizontal, vertical, radial, and angular. A UI element must be selected in `Drag Event Input`'s inspector. The option for Axis reflects the possible directions of movement. The value pre-defined as the range serves as the denominator the drag's produced values, i.e., higher values of the range make the drag's produced values smaller. The checkbox "Relative" produces values relative to the canvas (when checked) or global values (when unchecked).

Inlets: None
Outlet: Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Trigger Rect - A target Rect Transform from a UI element must be selected. It is from this component that the dragging activity will be picked up.

Axis - Drop down menu with options for how to interpret the dragging: Horizontal, Vertical, Radial, and Angular.

Relative - If this checkbox is activated, the values are scaled relative to the canvas. Otherwise, they are absolute values.

Range - User-defined range that divides the produced values.

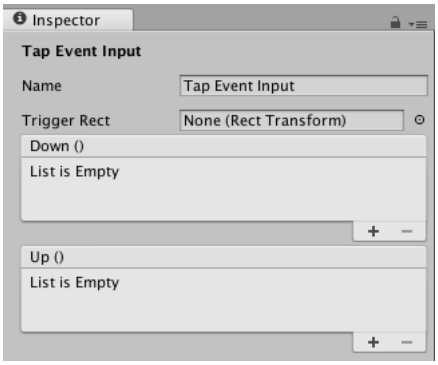
List - Summary of the node's output connection.

Example: MeUI

A.16 Input > UI Event > Tap Event Input

Description: `Tap Event Input` detects when a UI element has been pressed and released, sending a bang through its outlets when these activities occur.

Inlets: None
Outlet: Down Event (bang), Up Event (bang)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Trigger Rect - A target Trigger Rect from a UI element must be selected. It is from this component that the tapping activity will be picked up.

List - Summary of the node's output connection.

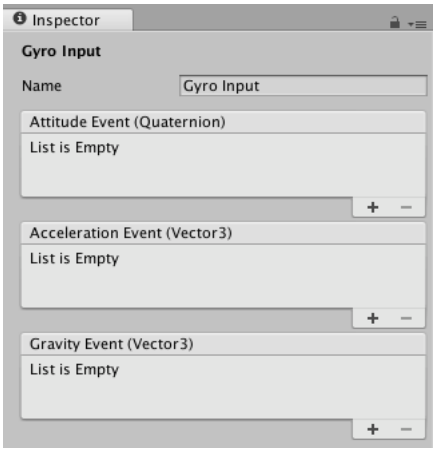
Example:

VL_Examples: Button

A.17 Input > Gyro Input

Description: `Gyro Input` receives gyroscope details from an appropriate device (e.g., smartphones and tablets). The available gyroscopic data are attitude, acceleration, and gravity. These are, respectively, the device's orientation in space, the speed at which the device is moving, and the direction of gravity as a vector.

Inlets: None
Outlets: Attitude (quaternion), Acceleration (vector3), Gravity (vector3)



Inspector:

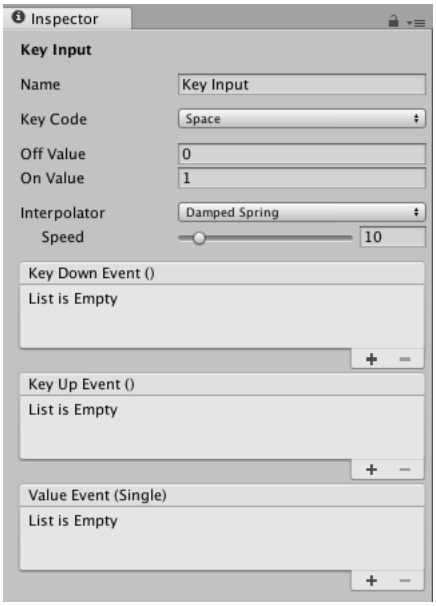
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

List - Summary of the node's output connection.

A.18 Input > Key Input

Description: `Key Input` listens for a single keyboard key to be pressed (specified through the Key Code dropdown menu), and triggers bangs when the key is pressed (Key Down) and released (Key Up), and sends float values for these same events (Value). The float values are defined in the node's inspector: Off Value is the value that will be propagated through the Value outlet when the key is released, and On Value is the value that will be propagated when the key is pressed.

Inlets: None
Outlet: Key Down (bang), Key Up (bang), Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Key Code - Dropdown menu of keyboard characters

Off Value - Float input, gets sent out at Value when key is released.

On Value - Same as Off Value but when note is pressed.

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

List - Summary of the node's output connection.

Examples:

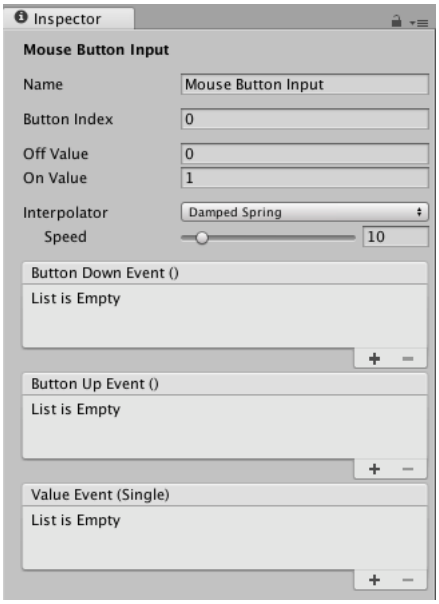
VideolabTest-master [2]: Adam, Peeler, Sphere, Splash, Stripe, Text2, Tilt Brush, Trail, Worm

VL_Examples: Dancer, Video, Wall

A.19 Input > Mouse Button Input

Description: `Mouse Button Input` listens for a mouse button click. One of three options can be specified in Button Index in the node's inspector: 0, 1, or 2. Usually, 0 is the left mouse button (primary button), 1 is the middle button, and 2 is the right button (secondary button). A bang is triggered and sent out the Button Down outlet when the specified mouse button is clicked and through the Button Up outlet when the button is released. Float values for these same events– click and release– are defined in the node's inspector and propagated out the Value outlet.

Inlets: None
Outlets: Button Down (bang), Button Up (bang), Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Button Index - String form to define which mouse button activity will be received by the node. The possible entries are 0, 1, or 2, for the left button (primary button), middle button, or right button (secondary button), respectively.

Off Value - Float value defined to be propagated through the Value outlet when the defined mouse button is released.

On Value - Float value defined to be propagated through the Value outlet when the defined mouse button is pressed.

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

List - Summary of the node's output connection.

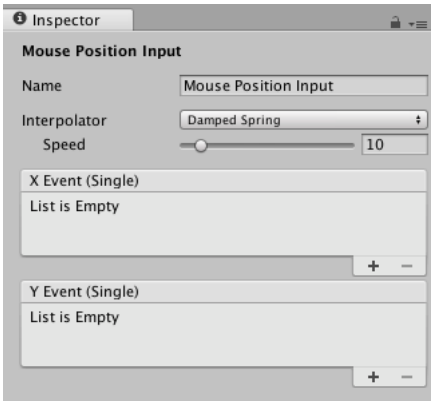
Example:

VideolabTest-master [2]: Peeler

A.20 Input > Mouse Position Input

Description: `Mouse Position Input` receives the x and y coordinates of the mouse cursor.

Inlets: None
Outlet: X (float), Y (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

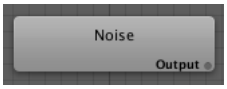
Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

List - Summary of the node's output connection.

Example:

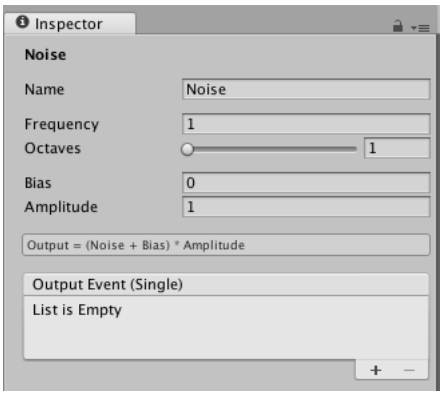
VL_Examples: Video, Wall

A.21 Input > Noise



Description: `Noise` uses Unity's Perlin noise to continuously generate pseudorandom numbers between -1 and 1. These random numbers can be further manipulated through the Frequency, Octave, Bias, and Amplitude, all of which are defined in the node's inspector. The generated output is (Noise + Bias) × Amplitude. Behind the scenes, Frequency scales time so that if the frequency is doubled, for example, the underlying pattern changes at twice the speed. Roughly, Octave combines several iterations of noise to generate more complex forms of randomness, where the octaves progressively get smaller. In other words, Frequency and Octave change the actual pseudorandom values generated, while Bias and Amplitude scale this generated value. Note that this is different from `Random Value` in two significant ways: `Noise` continuously generates pseudorandom values using Unity's Perlin noise, while `Random Value` generates a uniformly distributed pseudorandom value only when it receives a bang (with the option to send such a value when the game starts).

Inlets: None
Outlet: Output (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Frequency - Float input that defines the frequency.

Octaves - Slider allowing the user to choose between 8 integer octaves (1-8).

Bias - Float input that will scale the noise.

Amplitude - Float input that will scale the noise.

List - Summary of the node's output connection.

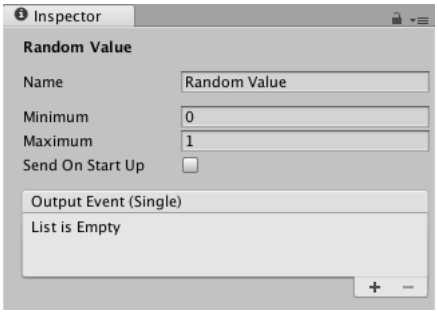
Example:

VL_Examples: ColorCubes, Dancer

A.22 Input > Random Value

Description: `Random Value` uses Unity's uniformly distributed Range pseudorandom number generator to generate single values between a user-defined range each time the node receives a bang. Note that this is different from `Noise` in two significant ways: `Random Value` generates a uniformly distributed pseudorandom value only when it receives a bang (with the option to send such a value when the game starts), while `Noise` continuously generates pseudorandom values using Unity's Perlin noise.

Inlet: Bang (bang)
Outlet: Output (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Minimum - User-defined lower bound for the pseudorandom number generator.

Maximum - User-defined upper bound for the pseudorandom number generator.

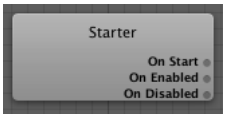
Send On Start Up - Check-box. If selected, will output a pseudorandom number when the game starts.

List - Summary of the node's output connection.

Examples:

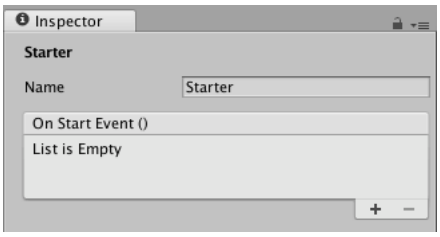
VideolabTest-master [2]: Adam, Primitive, Splash, Stripe, Text2, Trail, Worm

A.23 Input > Starter



Description: `Starter` sends a bang (or an int value of 1) in three situations: when the game starts (through the On Start outlet), when the node is enabled (through the On Enabled outlet), and when the node is disabled (through the On Disabled outlet). Because there is no inlet to the node, it is only possible to trigger the On Enabled and On Disabled messages by accessing `Starter`'s active status through alternative means. Some of Klak's nodes have a checkbox allowing the user to select whether the node should send messages as soon as the game starts. `Starter` is useful for triggering nodes at start-time that do not have this option.

Inlets: None
Outlet: On Start (bang), On Enabled (bang), On Disabled (bang)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

List - Summary of the node's output connection.

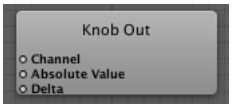
Examples:

VideolabTest-master [2]: Poly

Appendix B

Output

B.1 Output > MIDI > Knob Out

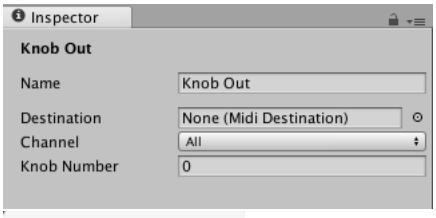


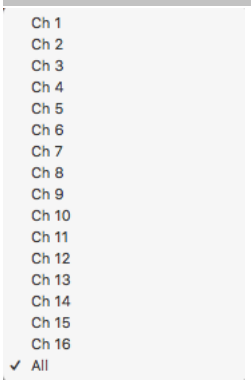
Description: Details on the `Knob Out` node are omitted from this documentation because:

1. This node requires a MIDI device with knobs (such as the OP-Z).
2. The author does not own a MIDI device with knobs.

The readers are encouraged to fill this page with necessary details.

Inlet: Channel (float), Absolute Value (float), Delta (float)
Outlets: None





Inspector:

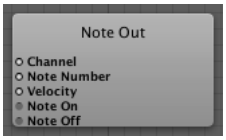
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Destination - Unity is oftentimes able to handle this automatically. However, in cases where the user has several MIDI destinations and would like to limit the outgoing information to a specific device, a MIDI Destination GameObject can be created by accessing the GameObject > MIDIJack > MIDI Destination.

Channel - The default option "All" ensures that any channels that are listening will receive the MIDI message. This is also able to accommodate situations where the user would like to limit the information being sent to a specific channel.

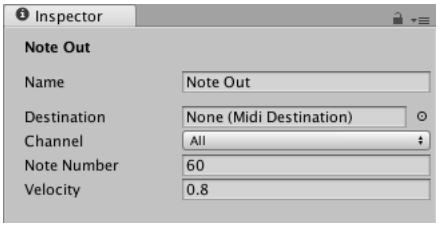
Knob Number - An integer value between 0-15 specifies which knob will be manipulated by the node.

B.2 Output > MIDI > Note Out



Description: `Note Out` combines separate pieces of MIDI information (the channel, note number, velocity, and information about note on and off) into a MIDI message that is sent to a device (e.g., a MIDI controller or DAW) that is able to receive and interpret these messages. The node is intuitive in the sense that it receives these pieces of information through its inlets and sends it to the selected MIDI destination defined in the node's inspector.

Inlets: Channel (float), Note Number (float), Velocity (float), Note On (bang), Note Off (bang)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

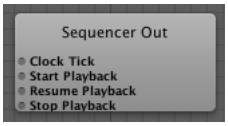
Destination - Unity is oftentimes able to handle this automatically. However, in cases where the user has several MIDI destinations and would like to limit the outgoing information to a specific device, a MIDI Destination GameObject can be created by accessing the GameObject > MIDIJack > MIDI Destination.

Channel - The default option "All" ensures that any channels that are listening will receive the MIDI message. This is also able to accommodate situations where the user would like to limit the information being sent to a specific channel.

Note Number - The user is able to use a predefined MIDI note number that gets sent to the MIDI destination whenever the node's Note On inlet receives a bang. This value can be changed during runtime by sending floats to the Note Number inlet.

Velocity - The user is able to use a pre-defined MIDI velocity that gets sent to the MIDI destination whenever the node's Note On inlet receives a bang. This value can be changed during runtime by sending floats to the Velocity inlet. Usually MIDI velocities are whole numbers between 0 and 127. This node, however, scales the velocities to floats between 0 and 1.

B.3 Output > MIDI > Sequencer Out

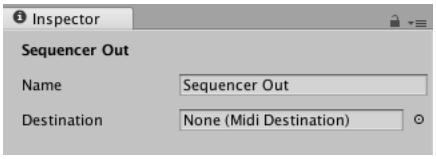


Description: Details on the `Sequencer Out` node are omitted from this documentation because:

1. This node requires a MIDI sequencer (such as the OP-Z).
2. The author does not own a MIDI sequencer.

The readers are encouraged to fill this page with necessary details.

Inlet: Clock Tick (bang), Start Playback (bang), Resume Playback (bang), Stop Playback (bang)
Outlets: None

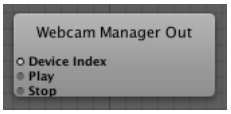


Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

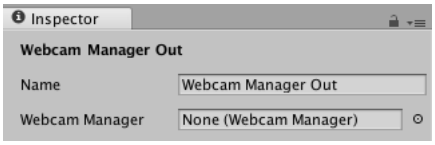
Destination - Unity is oftentimes able to handle this automatically. However, in cases where the user has several MIDI destinations and would like to limit the outgoing information to a specific device, a MIDI Destination GameObject can be created by accessing the GameObject > MIDIJack > MIDI Destination.

B.4 Output > Videolab > Webcam Manager Out



Description: `Webcam Manager Out` can stream the feed from a webcam on a 3D object or a UI canvas. This is done by creating an empty GameObject and adding the Webcam Manager component to it through its inspector. This will then give the option to select the appropriate input webcam device. A material must be created and added to the GameObject with the Webcam Manager component. This links the material to the webcam feed. Whenever this material is added to other objects, (e.g., a cube), the feed will be displayed on the object.[1] For information on displaying the webcam feed on a UI component, see the teenageengineering GitHub wiki. The `Webcam Manager Out` node allows manipulation of the webcam feed through its various inlets. The Device Index inlet receives floats that choose from the possibly numerous appropriate devices. Play and Stop can each receive a bang, causing the feed to play or stop, respectively. The empty GameObject with the Webcam Manager component must be added to the `Webcam Manager Out`'s Webcam Manager slot in the inspector.

Inlet: Device Index (float), Play (bang), Stop (bang)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Webcam Manager - A target GameObject with a Webcam Manager component must be selected.

Example:

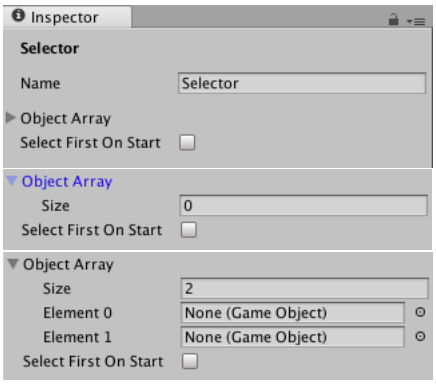
VL_Examples: Webcam

[1] This information was taken from the teenageengineering GitHub wiki [cite].

B.5 Output > Selector

Description: `Selector` allows toggling through various GameObjects, by turning their active status on or off. In the `Selector`'s inspector, the user defines the object array size, which will create an array with the selected number of slots for GameObjects. The float input is rounded down to match the numbers $1, 2, \ldots, n$, where n is the user-defined number of objects. If the option Select First On Start is not on, all active statuses are preserved until `Selector` starts receiving floats through its input. If it is on, no matter what the active statuses are, all will be turned off except for the GameObject in Element 0 of `Selector`'s inspector.

Inlet: Parameter (float)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Object Array Size - Int input that defines the number of GameObject slots.

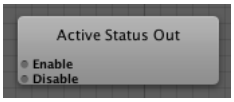
Element i - For each integer i between 0 and the object array size (minus 1, since the count starts at 0), there is a slot to select GameObjects.

Select First on Start - Checkbox so that first object is selected on start. If it is on, all objects when the game runs are turned off, except Element 0. If it is off, all active statuses are preserved until `Selector` starts receiving floats through its inlet.

Example:

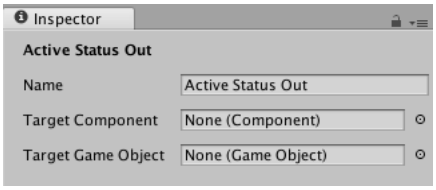
VL_Examples: Wall

B.6 Output > Component > Active Status Out



Description: `Active Status Out` toggles the active status of a selected GameObject or object component. The difference is turning an entire GameObject on or off, or toggling just one of its components. Note that the same `Active Status Out` node can toggle the active status of a particular GameObject and a component of an entirely different object. The node has two bang inlets: Enable and Disable, which turn the GameObject's active status on and off, respectively.

Inlets: Enable (bang), Disable (bang)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target Component - When a GameObject is selected, a dropdown menu is unveiled, allowing the user to select the specific component to be toggled by the node.

Target GameObject - A target GameObject is selected to be toggled by the node.

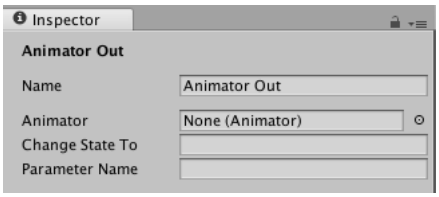
Examples:

VL_Examples: Collider, Dancer

B.7 Output > Component > Animator Out

Description: `Animator Out` allows control of an Animator Controller Asset through manipulation of its speed, selection of where to begin (or jump to) within the animated sequence, and ability to change the state of the animation (i.e., the animation sequence to play). A brief tutorial on Animators is given at the end of this document. The Speed inlet receives a float and adjusts the speed of the animation accordingly, in an intuitive way. The Normalized Time inlet receives a float between 0 and 1, where 0 is mapped to the beginning of the animation, and 1 to the end; this way, the value of 0.5, for instance, gets mapped to precisely the midpoint of the animation, for all animations, independent of their lengths. The Change State inlet is activated with a bang, and changes the state of the animation according to a string defined in the inspector that matches one of the animation states (i.e., animation sequence).

Inlets: Speed (float), Normalized Time (float), Float Parameter (float), Change State (bang)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Animator - A target animator component must be selected, upon which control of the animation through the inlets and the node operations is achieved.

Change State To - The string entered here must match state name within the Animator Controller Asset. When the Change State inlet receives a bang, the Animator Controller Asset will jump to this animation sequence.

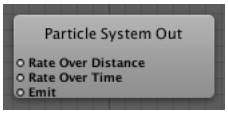
Parameter Name - The string entered here reflects the name of the parameter the user desires to use.

Examples:

VL_Examples: Dancer

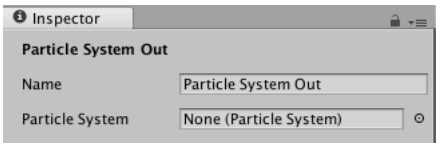
YouTube: Andy, Milo

B.8 Output > Component > Particle System Out



Description: `Particle Systems Out` allows control of a Particle System through manipulation of its rate over distance and time, and forcing the system to emit particles. A brief tutorial on Particle Systems is given at the of this document. The inlet Rate Over Distance receives a float which affects the rate at which the particle system's emitter generates new particles over distance, i.e., as the emitter moves. The inlet Rate Over Time is similar, where the float instead affects the rate at which the particle system's emitter generates new particles over time (this does not require movement). For example, dirt particles should be generated as an object moves along a dirt field (which happens with Rate Over Distance). On the other hand, if the object is standing still it would not make sense for dirt particles to be emitted (which would happen with Rate Over Time). Behind the scenes, both of these manipulate the Emission module of the particle system. Finally, the inlet Emit receives a bang causing the particle system to emit more particles.

Inlet: Rate Over Distance (float), Rate Over Time (float), Emit (bang)
Outlets: None



Inspector:

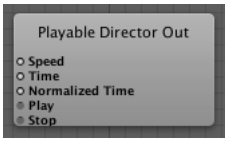
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Particle System - A target Particle System Component must be selected, upon which the rate over distance and time can be manipulated, and the system can be forced to emit more particles.

Examples:

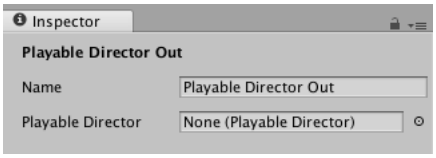
VideolabTest-master [2]: Primitive, Splash, Text1, Trail

B.9 Output > Component > Playable Director Out



Description: `Playable Director Out` allows control of a Playable Director and associated Timeline Object. Through this ensemble it is possible to create animations by defining the state of the objects that are being animated along specific periods of time. Unity then "fills in the blanks." More information can be found in a short tutorial at the end of this document. The `Playable Director Out` node allows control of the animation through manipulation of its speed, selection of a specific instance of time or a normalized unit of time (i.e., the exact time in seconds where the animation should jump to or the normalized time where 0 represents the beginning and 1 the end), and by playing or stopping the animation sequence.

Inlet: Speed (float), Time (float), Normalized Time (float), Play (bang), Stop (bang)
Outlets: None



Inspector:

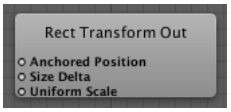
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Playable Director - A target Playable Director component must be selected, upon which control of the animation through the inlets and the node operations is achieved.

Example:

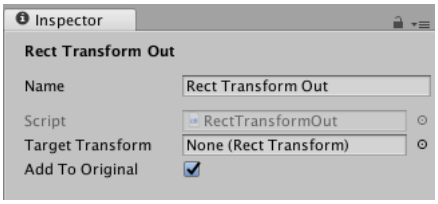
VL_Examples: Bounce

B.10 Output > Component > Rect Transform Out



Description: `Rect Transform Out` allows manipulation of a Rect Transform Object. The rect transform is the 2D extension of the standard transform, which is centered around a point (1D). The rect transform is rectangle wherein UI elements are placed. The inlets of `Rect Transform Out` allow the user to manipulate the chosen Rect Transform Component's anchored position (the position of the object relative to its set anchors), its size delta (the size of the object relative to its anchors), and its uniform scale.

Inlets: Anchored Position (vector3), Size Delta (vector3), Uniform Scale (float)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

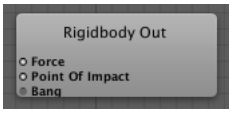
Target Transform - A target Rect Transform component must be selected, upon which control of its anchored position, size delta, and scale is achieved.

Add to Original - When this checkbox is active, the new values do not replace the original values, but add to them.

Example:

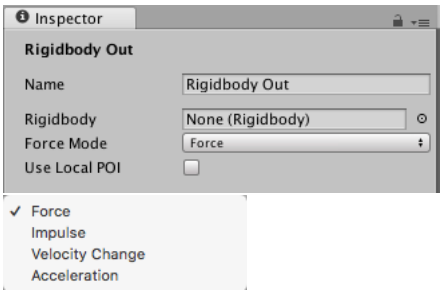
VL_Examples: Button

B.11 Output > Component > Rigidbody Out



Description: `Rigidbody Out` allows the user to manipulate the Rigidbody component of a GameObject. Objects with a Rigidbody component go through physics simulations, meaning, for instance, that the object is affected by gravity and impacts. In other words, such an object's motion will be dictated at least partly by Unity's physics engine. Through the `Rigidbody Out` node, the user can directly send data to an object's Rigidbody component. More precisely, these data detail the force being applied to the object and the point of impact. The Bang inlet of the node immediately applies the defined force at the defined point of impact.

Inlets: Force (vector3), Point of Impact (vector3), Bang (bang)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Rigidbody - A target GameObject with Rigidbody Component must be selected, upon which the selected force can be applied.

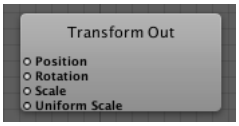
Force Mode - A dropdown menu gives the user a choice of Force, Impulse, Velocity Change, or Acceleration, as the way in which the node will interpret the force vector received through the Force inlet.

Use Local POI - A checkbox allowing the user the choose whether the point of impact will be local (if there is a parent) or global.

Examples:

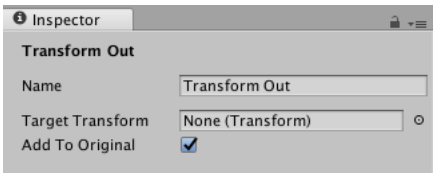
VL_Examples: Collider, Ramp

B.12 Output > Component > Transform Out



Description: `Transform Out` allows the user to manipulate a desired GameObject's transform information: its position, rotation, scale, and uniform scale.

Inlet: Position (vector3), Rotation (quaternion), Scale (vector3), Uniform Scale (float)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target Transform - A target GameObject must be selected whose transform will be manipulated by the node.

Add to Original - An intuitive checkbox. If checked, the values received by the various inlets of the node are added to the original values. Otherwise, they replace the original value.

Examples:

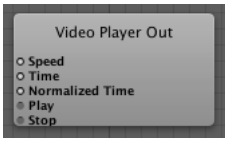
VideolabTest-master [2]: Adam, Boing

VL_Examples: Bounce

MIDIKlak [2]: Knob Event (Trigger), Knob Event (Value), Note Event (Trigger), Note Event (Value)

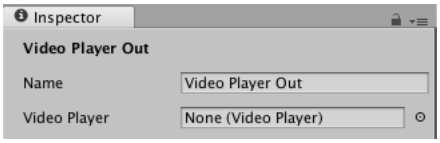
YouTube: Ninety Six, Milo

B.13 Output > Component > Video Player Out



Description: `Video Player Out` allows the user to manipulate a Video Player object, which does exactly what its name entails: play videos. Through the node's various inlets the user is able to change the speed of the video playback, jump to a specific point in time (absolute or relative), and play or stop the video.

Inlet: Speed (float), Time (float), Normalized Time (float), Play (bang), Stop (bang)
Outlets: None



Inspector:

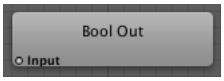
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Video Player - A target Video Player Component must be selected, upon which control of the video through the inlets and the node operations is achieved.

Example:

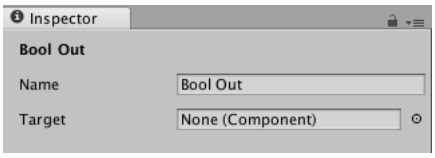
VL_Examples: Video

B.14 Output > Generic > Bool Out



Description: `Bool Out` sends a boolean value to an appropriate boolean property of a GameObject. Once a GameObject has been selected in `Bool Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific property of the GameObject that will be manipulated.

Inlet: Input (float)
Outlets: None



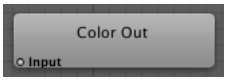
Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected to receive the boolean value.

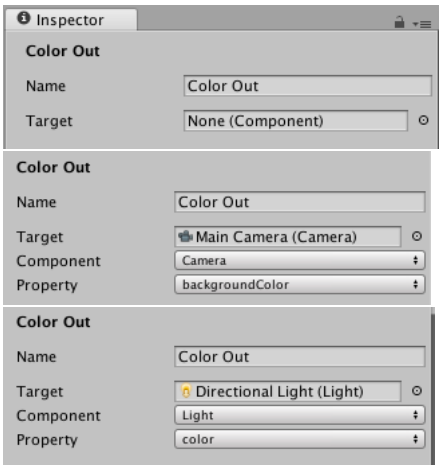
Example: Me1

B.15 Output > Generic > Color Out



Description: `Color Out` receives a color input and sends it to an appropriate GameObject. These include the camera background color and the light color. For this node to work with the camera, the camera's Clear Flags option must be set to solid color. In `Color Out`'s inspector, a target component must be selected (the camera or the light). Below are images of the inspector showcasing when the camera and the light are selected in `Color Out`'s inspector.
Open Question: Does `Color Out` control anything beyond the camera background color and the light color? All material colors are controlled by `Material Color Out`.

Inlet: Input (color)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

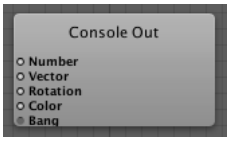
Target - Possible target GameObjects include the camera and the light. Once a GameObject is chosen, the correct component and property must be selected so that the node successfully manipulates the object's color.

Examples:

VideolabTest-master [2]: Primitive, Trail, Worm

VL_Examples: Button

B.16 Output > Generic > Console Out



Description: `Console Out` prints the values received through its inlets to the console. The purpose of this node is debugging. If more than one `Console Out` is being used, or if several parameters are being sent to the console using `Console Out`, then it is recommended that each `Console Out` receives a different name, so that the data in the debugging console is better organized. Although this node has does not affect anything in the game, it is **very** useful.

Inlets: Number (float), Vector (vector3), Rotation (quaternion), Color (color), Bang (bang)
Outlets: None

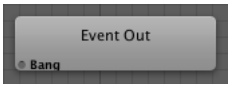
Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Example:

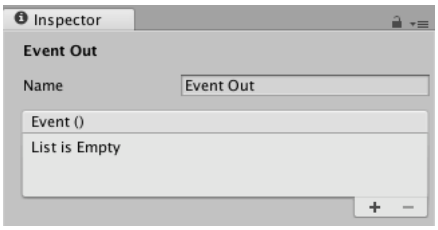
VL_Examples: Ramp

B.17 Output > Generic > Event Out



Description: `Event Out` invokes an event. Because `Event Out` has no outlets, the event must be defined using the list in the node's inspector. This is done by clicking the "+" in `Event Out`'s list, which reveals an empty slot for a GameObject to receive the event.

Inlet: Bang (bang)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

List - Summary of the node's output connection.

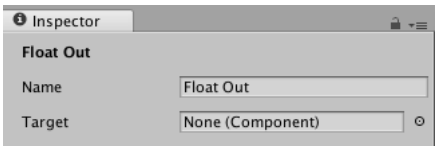
Examples:

VideolabTest-master [2]: Peeler, Sphere, Text2, Worm

B.18 Output > Generic > Float Out

Description: `Float Out` sends a float value to an appropriate float property of a GameObject. Once a GameObject has been selected in `Float Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific property of the GameObject that will be manipulated.

Inlet: Input (float)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected to receive the float value.

Examples:

VideolabTest-master [2]: Splash, Stripe, Trail, Worm

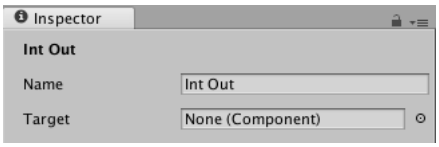
VL_Examples: Ramp, Video

YouTube: Milo

B.19 Output > Generic > Int Out

Description: `Int Out` sends an int value to an appropriate int property of a GameObject. Once a GameObject has been selected in `Int Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific property of the GameObject that will be manipulated.

Inlet: Input (float)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected to receive the int value.

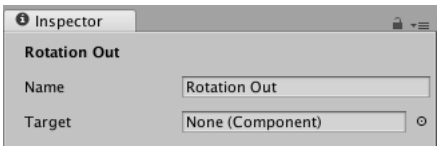
Example:

VL_Examples: Collider

B.20 Output > Generic > Rotation Out

Description: `Rotation Out` sends a rotation (quaternion) to an appropriate property of a GameObject. Once a GameObject has been selected in `Rotation Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific property of the GameObject that will be manipulated.

Inlet: Input (quaternion)
Outlets: None



Inspector:

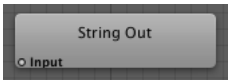
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected to receive the rotation.

Example:

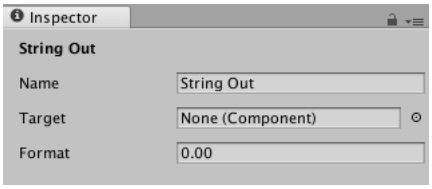
VL_Examples: Collider

B.21 Output > Generic > String Out



Description: `String Out` sends a string to an appropriate string property of a GameObject. Once a GameObject has been selected in `String Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific property of the GameObject that will be manipulated. This is a mysterious node since it receives a float input, and its inspector contains the field Format, which is float-valued.

Inlet: Input (float)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

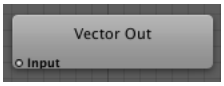
Target - A target GameObject must be selected to receive the string.

Format - Mysterious float-valued property of the node.

Example:

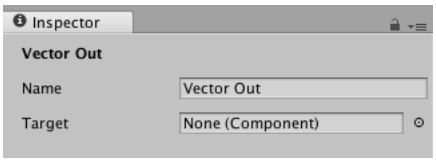
VL_Examples: Button

B.22 Output > Generic > Vector Out



Description: `Vector Out` sends a vector to an appropriate vector property of a GameObject. Once a GameObject has been selected in `Vector Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific property of the GameObject that will be manipulated.

Inlet: Input (vector3)
Outlets: None



Inspector:

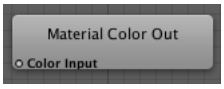
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected to receive the vector.

Example:

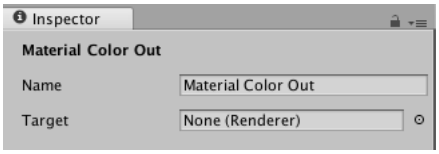
VL_Examples: Warp

B.23 Output > Renderer > Material Color Out



Description: `Material Color Out` allows the user to manipulate the color of materials. Its sister node is `Color Out`. However, there is an important distinction between these nodes. `Color Out` affects color properties that are not based on materials, such as the camera's background color. Once a GameObject has been selected in `Material Color Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific material color property of the GameObject that will be manipulated.

Inlet: Color Input (color)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected from which the appropriate material property is chosen to receive the color.

Examples:

VideolabTest-master [2]: Adam, Splash, Stripe, Text2, Trail, Worm

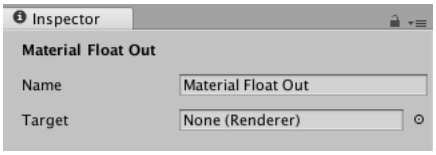
VL_Examples: ColorCubes

YouTube: Milo

B.24 Output > Renderer > Material Float Out

Description: `Material Float Out` allows the user to manipulate the float-valued properties of materials. Its sister node is `Float Out`. However, there is an important distinction between these nodes. `Float Out` affects float properties that are not based on materials. Once a GameObject has been selected in `Material Float Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific material float property of the GameObject that will be manipulated.

Inlet: Float Input (float)
Outlets: None



Inspector:

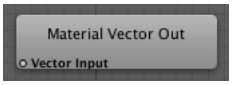
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected from which the appropriate material property is chosen to receive the float.

Examples:

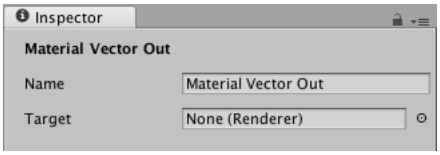
VideolabTest-master [2]: Adam, Boing, Peeler, Poly, Primitive, Sphere, Splash, Stripe, Text1, Text2, Tilt Brush, Worm

B.25 Output > Renderer > Material Vector Out



Description: `Material Vector Out` allows the user to manipulate vector-valued properties of materials. Its sister node is `Vector Out`. However, there is an important distinction between these nodes. `Vector Out` affects vector properties that are not based on materials. Once a GameObject has been selected in `Material Vector Out`'s inspector, a menu hierarchy is unveiled, allowing the user to select the specific material color property of the GameObject that will be manipulated.

Inlet: Vector Input (vector3)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Target - A target GameObject must be selected from which the appropriate material property is chosen to receive the vector.

Example: Me1

B.26 Output > Rumble Out

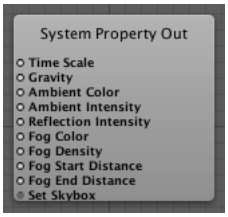
Description: `Rumble Out` trigger's a vibration on the device. The device (e.g., a smartphone or tablet) must be able to produce vibrations.

Inlet: Vibrate (bang)
Outlets: None

Inspector:

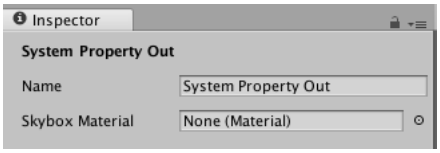
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

B.27 Output > System Property Out



Description: `System Property Out` works with the RenderSettings class in Unity, affecting visual elements in the scene, such as fog and ambient light. Although the node's inspector asks for a material, this is unnecessary for functionality.

Inlet: Time Scale (float), Gravity (vector3), Ambient Color (color), Ambient Intensity (float), Reflection Intensity (float), Fog Color (color), Fog Density (float), Fog Start Distance (float), Fog End Distance (float), Set Skybox (bang)
Outlets: None



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Skybox Material - A target material component *may* be selected for specific manipulation of a material. Otherwise the scene as a whole is affected by the node.

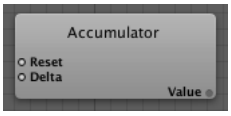
Examples:

VideolabTest-master [2]: Poly

Appendix C

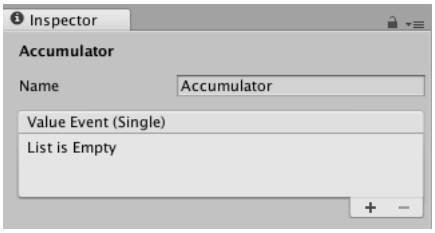
Conversion

C.1 Conversion > Accumulator



Description: `Accumulator` keeps a running sum of the floats it receives. Note that certain nodes continuously send their value while they are activated (or deactivated!). The `Accumulator` would then accumulate perhaps more than is desired. Hence, if discrete single values are desired to be accumulated, special attention should be taken concerning the way `Accumulator` receives values. The floats to be accumulated are sent to the Delta inlet. To reset the accumulation, a bang is sent to the Reset inlet. The accumulated values are sent out of the Value outlet.

Inlets: Reset (bang), Delta (float)
Outlet: Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

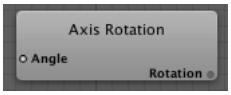
List - Summary of the node's output connection.

Examples:

VideolabTest-master [2]: Stripe

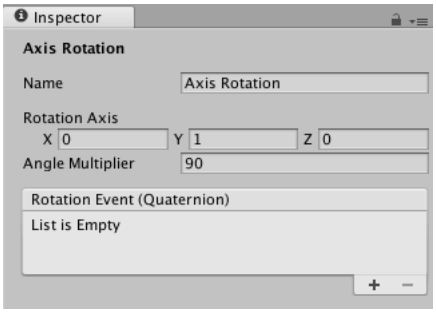
YouTube: Andy

C.2 Conversion > Axis Rotation



Description: `Axis Rotation` converts a float into a rotation along a user-defined axis. Like `Euler Rotation`, the x-axis is roll, y is yaw, and z is pitch. The Angle inlet is scaled by the Angle Multiplier (defined in the inspector)

Inlet: Angle (float)
Outlet: Rotation (quaternion)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Rotation Axis - User-defined vector of rotation. The vectors $(1, 0, 0), (0, 1, 0)$, and $(0, 0, 1)$ correspond, respectively, to the x (roll), y (yaw), and z (pitch) axes.

Angle Multiplier - Scales the incoming float into degrees. (Usually scaling floats in Klak is achieved with `Float Filter`.)

List - Summary of the node's output connection.

Examples:

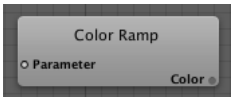
VLExamples: Airplane

VideolabTest-master [2]: Adam

MIDIKlak [2]: Knob Event (Value)

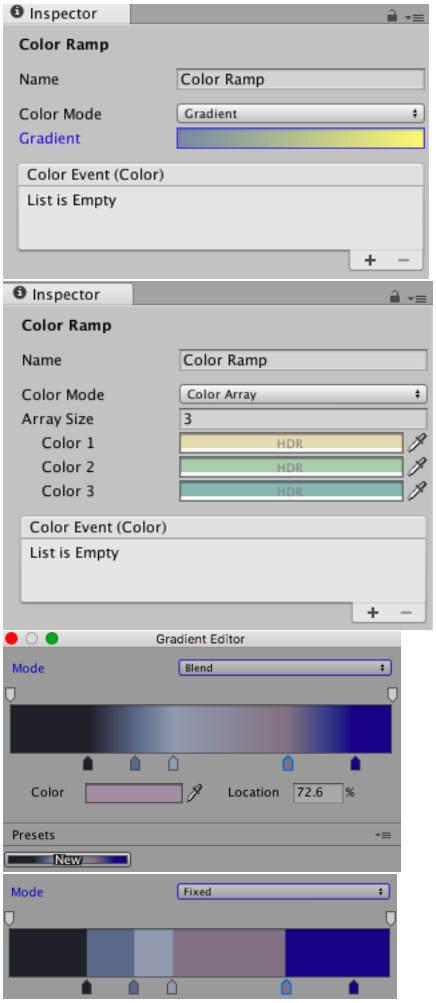
YouTube: Ninety Six

C.3 Conversion > Color Ramp



Description: `Color Ramp` smoothly transitions between a set of colors or a gradient, parametrized by floats in the range of 0 to 1.

Inlet: Parameter (float)
Outlet: Color (color)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Color Mode - A dropdown menu has the option of Gradient or Color Array.

Gradient - A gradient with the options of Blend and Fixed can be created using the Gradient Editor. The lower tags control the colors, and can be moved around to control the gradient. Additional tags can be inserted by clicking in the region of the tabs. The upper tags control the alphas of the gradient, and are manipulated like the lower tags.

Color Array - A sequence of colors that is equidistantly distributed along the interval from 0 to 1. For example, if 5 colors are created, they will be mapped, in order, to 0, 0.25, 0.5, 0.75, and 1. The areas in between these points contain “in between" colors.

List - Summary of the node’s output connection.

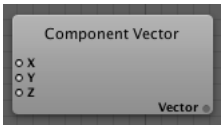
Examples:

VideolabTest-master [2]: Poly

VL_Examples: Button

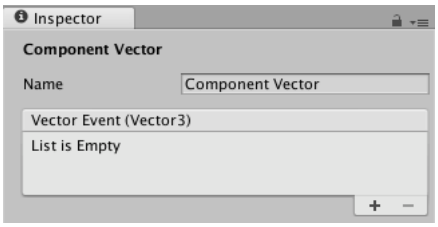
YouTube: Milo

C.4 Conversion > Component Vector



Description: `Component Vector` creates a vector3 from three float inputs.

Inlets: X (float), Y (float), Z (float)
Outlet: Vector (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

List - Summary of the node's output connection.

Examples:

VLExamples: Mouse-Position Rotate

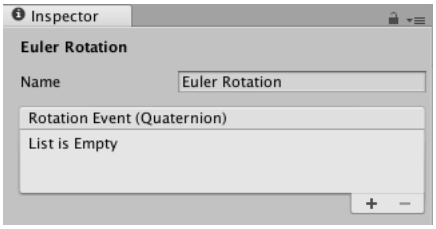
VideolabTest-master [2]: Adam

YouTube: Milo

C.5 Conversion > Euler Rotation

Description: Euler rotations are defined by three angles, which control the roll, yaw, and pitch of the rotating object. In Unity, an object's rotation in the inspector is manipulated via Euler Rotations. The `Euler Rotation` node allows a vector3 to manipulate an object's rotation. Behind the scenes, the node converts vector3's into quaternions, where x is roll, y is yaw, and z is pitch.

Inlet: Euler Angles (vector3)
Outlet: Rotation (quaternion)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

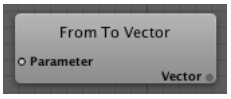
List - Summary of the node's output connection.

Example:

VLExamples: Mouse-Position Rotate

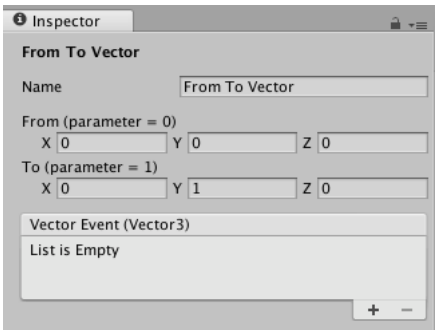
VideolabTest-master [2]: Adam

C.6 Conversion > From To Vector



Description: `From To Vector` parametrizes an infinite line between two user-defined vectors. When the float received via the inlet is 0, the node sends the first user-defined vector through its outlet, and when it is 1, the second vector is sent. The line works for values outside of the range from 0 to 1.

Inlet: Parameter (float)
Outlet: Vector (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

From (parameter = 0) - The node will output this user-defined vector when the input float is 0.

To (parameter =1) - The node will output this user-defined vector when the input float is 1.

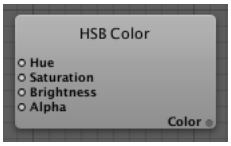
List - Summary of the node's output connection.

Examples:

VLExamples: Line Cube

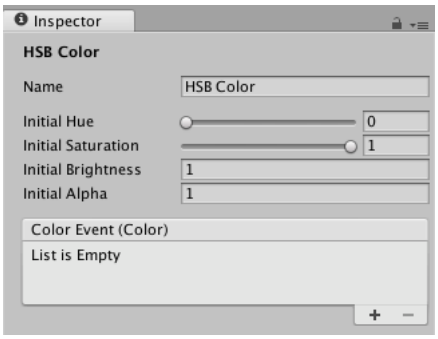
VideolabTest-master [2]: Boing

C.7 Conversion > HSB Color



Description: `HSB Color` starts with user-defined (or default) values for hue, saturation, brightness, and alpha. The node has four inlets: hue, saturation, brightness, and alpha, which can be manipulated during runtime. Each color component is scaled to use floats between 0 and 1.

Inlets: Hue (float), Saturation (float), Brightness (float), Alpha (float)
Outlet: Color (color)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Initial Hue - A slider allows the user to choose a float from 0 to 1 to serve as the initial hue.

Initial Saturation - A slider allows the user to choose a float from 0 to 1 to serve as the initial saturation.

Initial Brightness - A float input box allows the user to input a float to serve as the initial saturation.

Initial Alpha - A float input box allows the user to input a float to serve as the initial alpha.

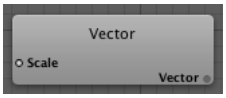
List - Summary of the node's output connection.

Examples:

VideolabTest-master [2]: Adam, Primitive, Splash, Stripe, Text2, Trail, Worm

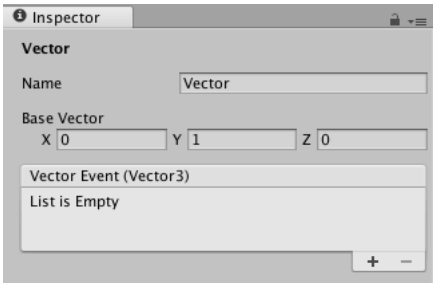
VL_Examples: ColorCubes

C.8 Conversion > Vector



Description: `Vector` multiplies a user-defined base vector by an input float.

Inlet: Scale (float)
Outlet: Vector (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Base Vector - Define the base vector that will get multiplied by the incoming float.

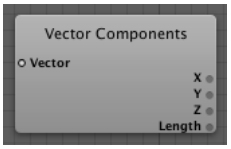
List - Summary of the node's output connection.

Examples:

VL_Examples: Warp

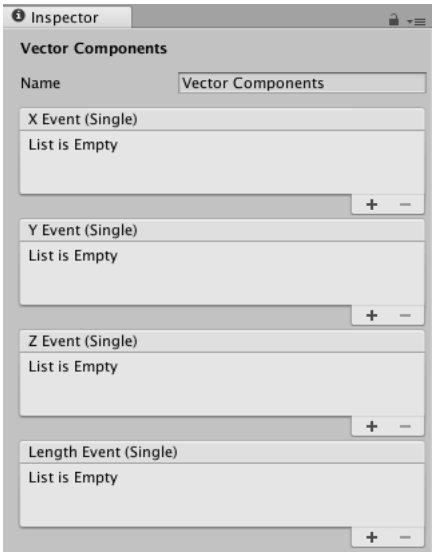
MIDIKlak [2]: Knob Event (Trigger), Knob Event (Value), Note Event (Trigger), Note Event (Value)

C.9 Conversion > Vector Components



Description: `Vector Components` decomposes a vector into its x, y, and z coordinates, and calculates the length of the vector.

Inlet: Vector (vector3)
Outlets: X (float), Y (float), Z (float), Length (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Lists - Summary of the node's output connections.

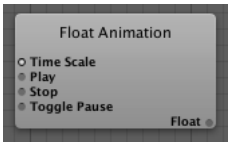
Example:

VL_Examples: Warp

Appendix D

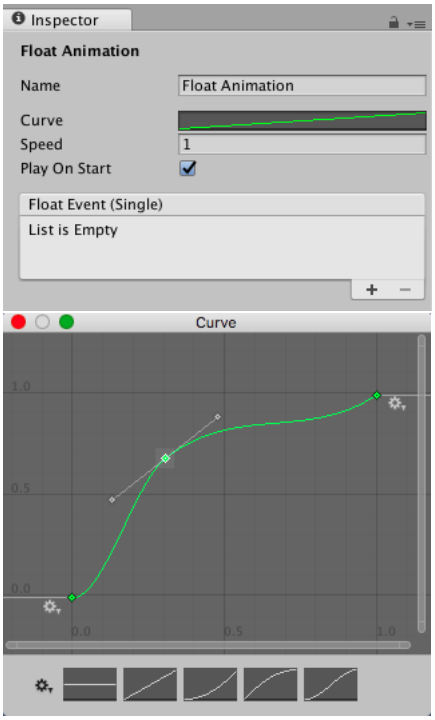
Animation

D.1 Animation > Float Animation



Description: `Float Animation`, despite its name, does not involve actual animations. The node outputs float values from a user-defined curve along time, and as such, is an animation of float values. Once `Float Animation` has reached its endpoint, it continuously ends out its last value. To prevent this, the node must be either paused (by sending a bang to its Toggle Pause inlet) or stopped (by sending a bang to its Stop inlet). If the animation is paused, it can be continued from where it left off by sending another bang to Toggle Pause. If the animation is stopped, it can only be restarted by sending a bang to Play. The Time Scale inlet receives floats that alter the speed of the animation.

Inlets: Time Scale (float), Play (bang), Stop (bang), Toggle Pause (bang)
Outlet: Float (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Curve - The default curve is a line with slope one, increasing from 0 to 1 uniformly across time. The user can change the curve by moving the already existing keys (the moveable points that shape the curve) and by creating additional keys.

Speed - The default time for float animation is 1 second. This can be adjusted either in the animation inspector, by dragging the end point farther down the x-axis, or by setting the speed. The speed is adjustable in runtime through the input Time Scale.

Play on Start - Option to play immediately when the scene is loaded.

List - Summary of the node's output connection.

Examples:

VideolabTest-master [2]: Adam, Boing, Peeler, Poly, Primitive, Sphere, Splash, Stripe, Text1, Text2, Tilt Brush, Trail, Worm

MIDIKlak [2]: KnobEvent (Trigger), NoteEvent (Trigger)

YouTube: Milo

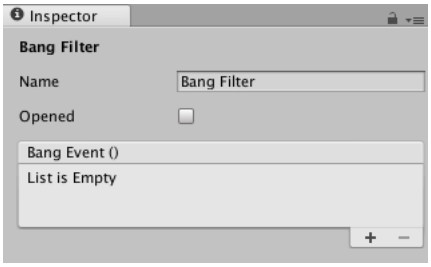
Appendix E

Filter

E.1 Filter > Bang Filter

Description: `Bang Filter` can be thought of as a faucet. The faucet is open when the node has received a bang in its Open inlet, and it is closed when the node has received a bang in its Closed inlet. Bangs sent to the Bang inlet of `Bang Filter` are only pushed through the node and sent out of its Bang outlet if the node is open.

Inlets: Bang (bang), Open (bang), Close (bang)
Outlet: Bang (bang)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Opened - Checking this option allows bangs to flow through the `Bang Filter` when the scene is loaded. During runtime, the open and close options are manipulated via the inlets Open and Close.

List - Summary of the node's output connection.

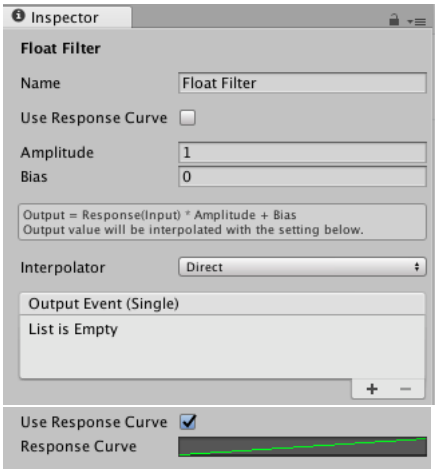
Examples:

VideolabTest-master [2]: Adam, Splash, Stripe, Text2, Trail, Worm

E.2 Filter > Float Filter

Description: `Float Filter` applies a linear function to an incoming float with the option of using a response curve. The response curve first maps the float received through the Input inlet according to the curve. The curve's output is then manipulated by the Amplitude and Bias. If the option Use Response Curve is not selected, then the float received through the Input inlet is manipulated directly by the Amplitude and Bias.

Inlet: Input (float)
Outlet: Output (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Use Response Curve - If this option is selected, the float is first mapped by the user-defined function, and the result is transformed by the Amplitude and Bias.

Amplitude - This user-defined value scales the incoming float.

Bias - This user-defined value is added to the scaled float (the product of the float and the Amplitude).

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

List - Summary of the node's output connection.

Examples:

VideolabTest-master [2]: Adam, Boing, Peeler, Poly, Primitive, Stripe, Text1, Text2, Tilt Brush, Trail, Worm

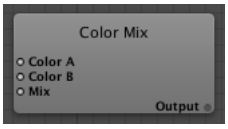
VL_Examples: Bounce, Button, Dancer, Ramp, Video

YouTube: Milo

Appendix F

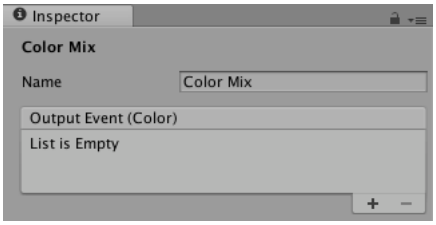
Mixing

F.1 Mixing > Color Mix



Description: `Color Mix` takes two RGBA color inputs, Color A and Color B (both 4-dimensional vectors), and mixes them using an incoming float value x. The mix is computed as x(Color A) + $(1 - x)$(Color A).

Inlets: Color A (color), Color B (color), Mix (float)
Outlet: Output (color)



Inspector:

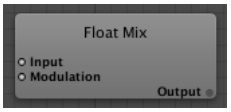
Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

List - Summary of the node's output connection.

Example:

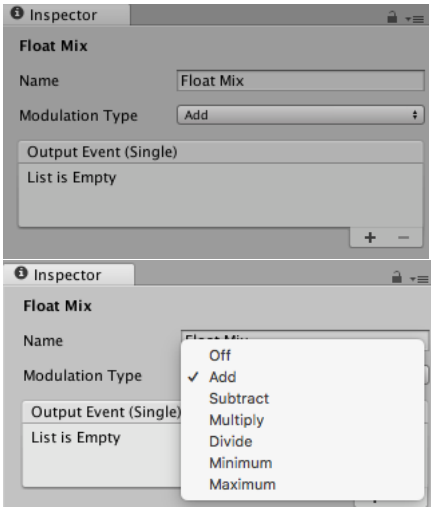
VL_Examples: ColorCubes

F.2 Mixing > Float Mix



Description: `Float Mix` performs a user-defined mathematical operation on the input and modulation floats. These operations are addition, subtraction, multiplication, division, minimum, and maximum. The node only sends out the final computation when it receives updated values through either one of its inlets (or both). If a continuous stream is desired, at least one of its inlets should be receiving a continuous stream of floats.

Inlets: Input (float), Modulation (float)
Outlet: Output (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Modulation Type - Dropdown menu with options Off, Add, Subtract, Multiply, Divide, Min, Max. Denote the input and modulation floats as x and t, respectively. Then,

Off outputs the input float x.

Add: $x + t$

Subtract: $x - t$

Multiply: xt

Divide: x/t

Minimum: $\min(x, t)$

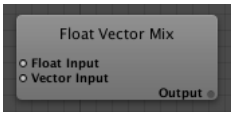
Maximum: $\max(x, t)$

List - Summary of the node's output connection.

Examples:

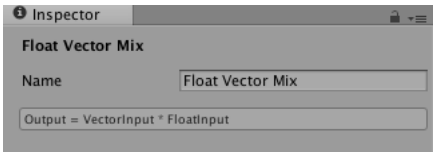
VL_Examples: Bounce, Warp

F.3 Mixing > Float Vector Mix



Description: `Float Vector Mix` scales a vector by a float. The node only sends out the final computation when it receives updated values through either one of its inlets (or both). If a continuous stream is desired, at least one of its inlets should be receiving a continuous stream of data.

Inlets: Float Input (float), Vector Input (vector3)
Outlet: Output (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Example:

VL_Examples: Warp

F.4 Mixing > Rotation Mix

Description: `Rotation Mix` computes the product of two quaternions. Quaternions themselves represent rotations, and their product represents the composition of the rotations. In other words, first the object is rotated by the quaternion Input, and then it is rotated by the quaternion Modulation. Note that quaternion multiplication is not commutable, so the product of Input and Modulation is not the same as the product of Modulation and Input.

Inlets: Input (quaternion), Modulation (quaternion)
Outlet: Output (quaternion)

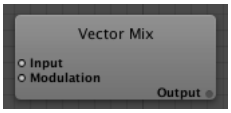
Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Example:

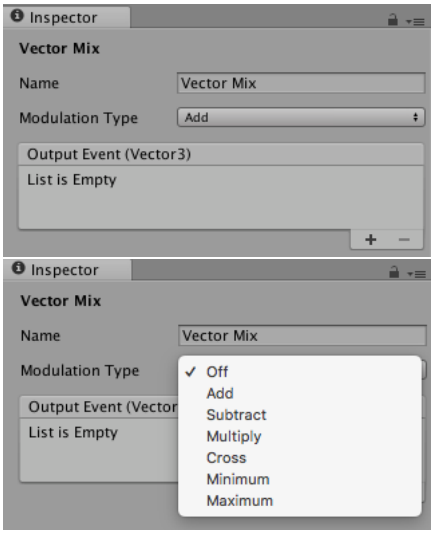
VL_Examples: Collider

F.5 Mixing > Vector Mix



Description: `Vector Mix` applies a user-defined vector operation to the input and modulations vector. The node is activated whenever there is a change in either the input or the modulation vectors.

Inlets: Input (vector3), Modulation (vector3)
Outlet: Output (vector3)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Modulation Type - Dropdown menu with options Off, Add, Subtract, Multiply, Cross, Min, Max. Denote the input and modulation vectors as $v = (v_1, v_2, v_3)$ and $t = (t_1, t_2, t_3)$, respectively. Then,

Off outputs the input vector $v = (v_1, v_2, v_3)$.

Add: $(v_1 + t_1, v_2 + t_2, v_3 + t_3)$

Subtract: $(v_1 - t_1, v_2 - t_2, v_3 - t_3)$

Multiply: $(v_1 \cdot t_1, v_2 \cdot t_2, v_3 \cdot t_3)$

Cross[1] : $v \times t$

Minimum: $(\min(v_1, t_1), \min(v_2, t_2), \min(v_3, t_3))$

Maximum: $(\max(v_1, t_1), \max(v_2, t_2), \max(v_3, t_3))$

List - Summary of the node's output connection.

Example:

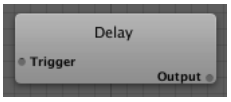
VL_Examples: Warp

[1]The cross product produces a vector perpendicular to both v and t and thus normal to the plane containing v and t.

Appendix G

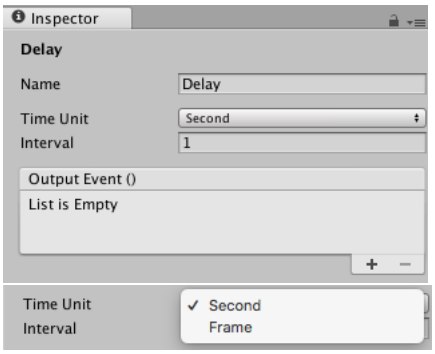
Switching

G.1 Switching > Delay



Description: `Delay` simply delays a bang by specified amount of time in seconds or frames.

Inlet: Trigger (bang)
Outlet: Output (bang)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Time Unit - Dropdown menu with options Second and Frame.

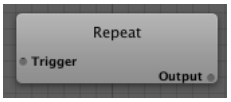
Interval - The amount of seconds or frames the bang is delayed.

List - Summary of the node's output connection.

Example:

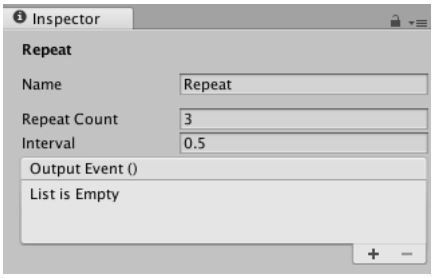
VL_Examples: Collider

G.2 Switching > Repeat



Description: `Repeat` transforms a single bang into several bangs sent in equally spaced intervals of time.

Inlet: Trigger (bang)
Outlet: Output (bang)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Repeat Count - Specify the amount of times to repeat the bang.

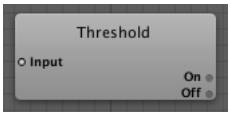
Interval - Specify the amount of seconds between repetitions of the bang.

List - Summary of the node's output connection.

Example:

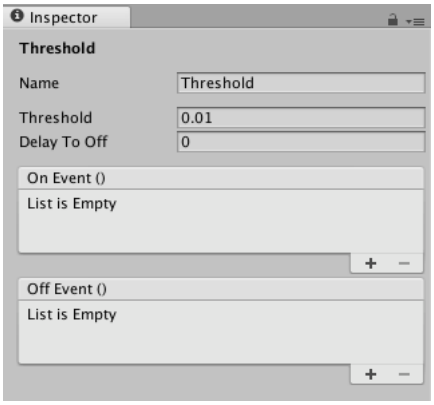
VL_Examples: Collider

G.3 Switching > Threshold



Description: `Threshold` allows the user to define a threshold value that, once met (*i.e.*, the incoming float x is such that $x \geq$ threshold), a bang is sent through the On outlet. In addition, the user may define a time delay in seconds such that after the bang through the On outlet, a bang is sent through the Off outlet. If the incoming float value x does not satisfy the threshold, no bangs are sent either through the On or the Off outlets.

Inlet: Input (float)
Outlets: On (bang), Off (bang)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Threshold - Select the value to serve as the threshold. All incoming float values greater than or equal to the threshold will activate the `Threshold` node and send a bang through the On outlet.

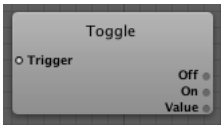
Delay to Off - Select the amount of seconds to wait to send a bang through the Off outlet after a bang has been sent through the On outlet.

List - Summary of the node's output connection.

Example:

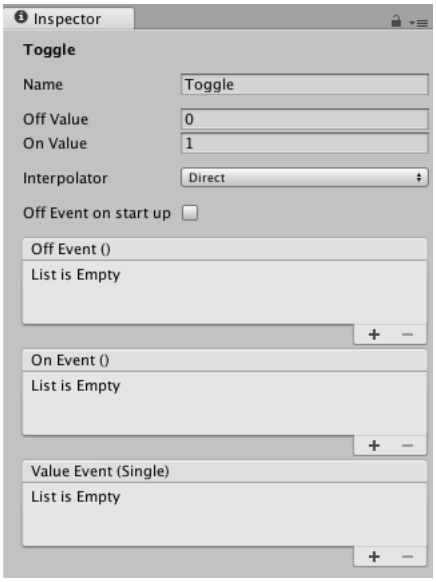
VL_Examples: Ramp

G.4 Switching > Toggle



Description: `Toggle` cycles through sending a bang from its On outlet to sending one from its Off outlet, in alternating fashion, each time a bang is received through its Trigger inlet. Every odd-numbered bang received (e.g., the first, third, fifth, and so on) causes the node to send a bang from its On outlet; every even-numbered bang received causes the node to send a bang from its Off outlet. In addition, Toggle sends a user-defined On Value and an Off Value from its Value outlet, depending on the order of the bang. Note that the values sent through the Value outlet are done so in continuous fashion.

Inlet: Trigger (bang)
Outlets: On (bang), Off (bang), Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

Off Value - The float that will be sent through the Value outlet during the off-cycle (until the node is toggled to send an On bang).

On Value - The float that will be sent through the Value outlet during the on-cycle (until the node is toggled to send an Off bang).

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

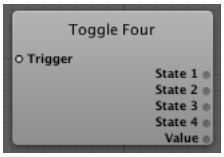
Off Event at Startup - If checked, the node will send the Off bang when the scene runs.

List - Summary of the node's output connection.

Examples:

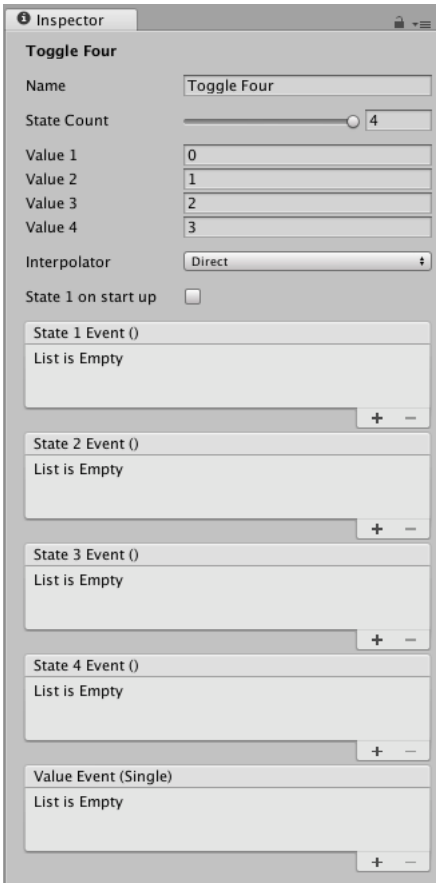
VideolabTest-master [2]: Splash, Text2

G.5 Switching > Toggle Four



Description: `Toggle Four` is a customizable version of the `Toggle` node. Whereas `Toggle` has two states (on and off), `Toggle Four` has, by default, four states (state 1, state 2, state 3, and state 4). The number of states may be defined to be 2, 3, or 4. `Toggle Four` receives a bang as an input, which cycles through the states. In each state, the node outputs a bang from the corresponding state's outlet, and a value from the node's Value outlet. This value is sent continuously, even as it changes.

Inlet: Trigger (bang)
Outlet: State 1 (bang), State 2 (bang), State 3 (bang), State 4 (bang), Value (float)



Inspector:

Name - Specifies the name of the node, which can be used to organize the patch. This is an optional feature.

State Count - Slider to choose int value from 2 to 4. This defines the number of states of `Toggle Four`.

Value i - for i=1 to 4, the user can define Float values to be sent continuously through the Value outlet, depending on the state of the node.

Interpolator - Interpolators are methods of transitioning between values. This can be done directly by jumping from one value to another, or smoothly by following a curve. A dropdown menu contains the options Direct, Exponential, and Damped Spring. The latter two have an option for speed.

State 1 on Startup - The option to send the State 1 bang and value when the game runs.

List - Summary of the node's output connection.

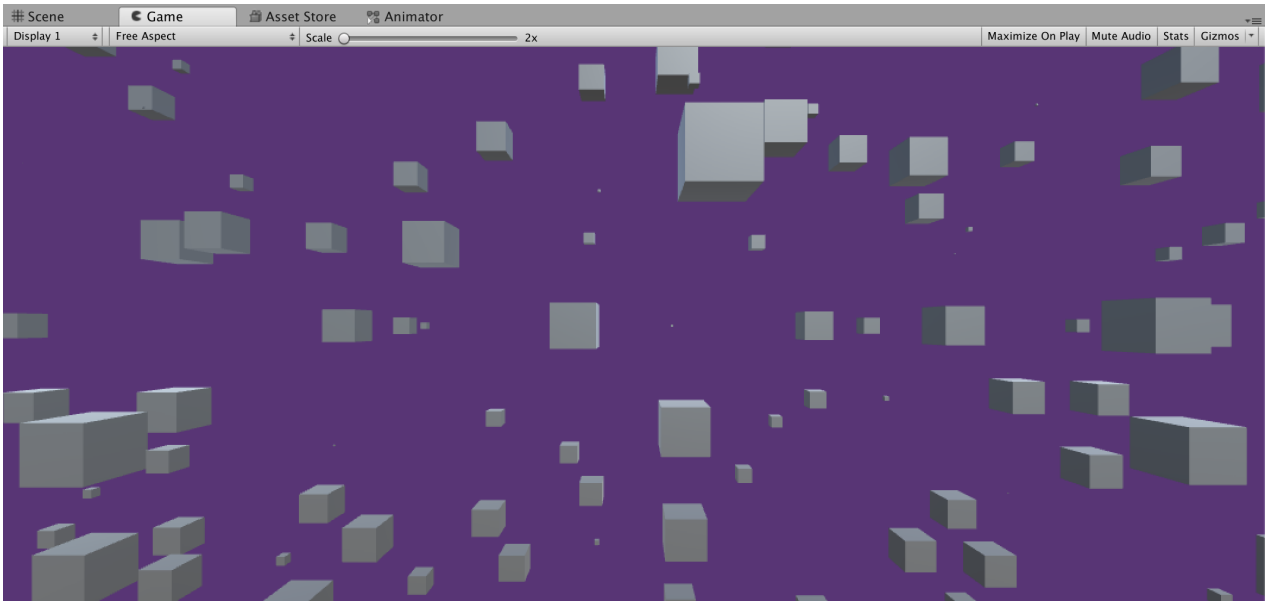
Examples:

VL_Examples: Button, Collider

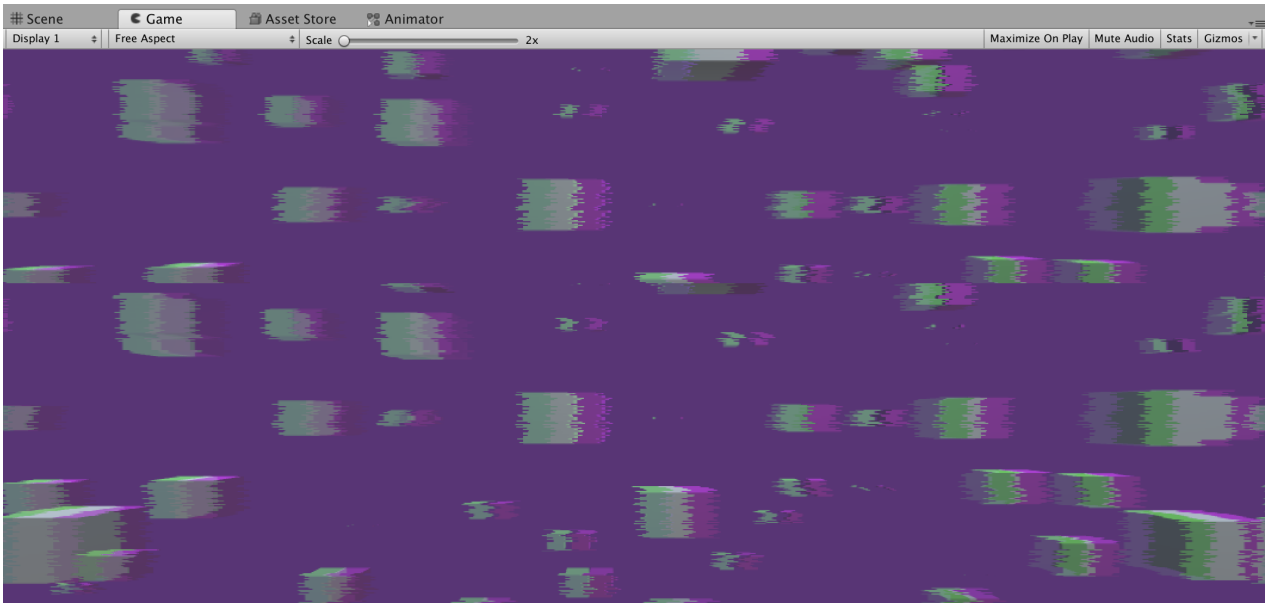
Appendix H

Kino Image Effects

Videolab includes several post-processing image effects that are easily manipulable with Klak. This section provides a short description for each of these effects. The image below is of the scene before any post-processing effect is applied. In the description of each effect, an image showing the altered scene is provided.

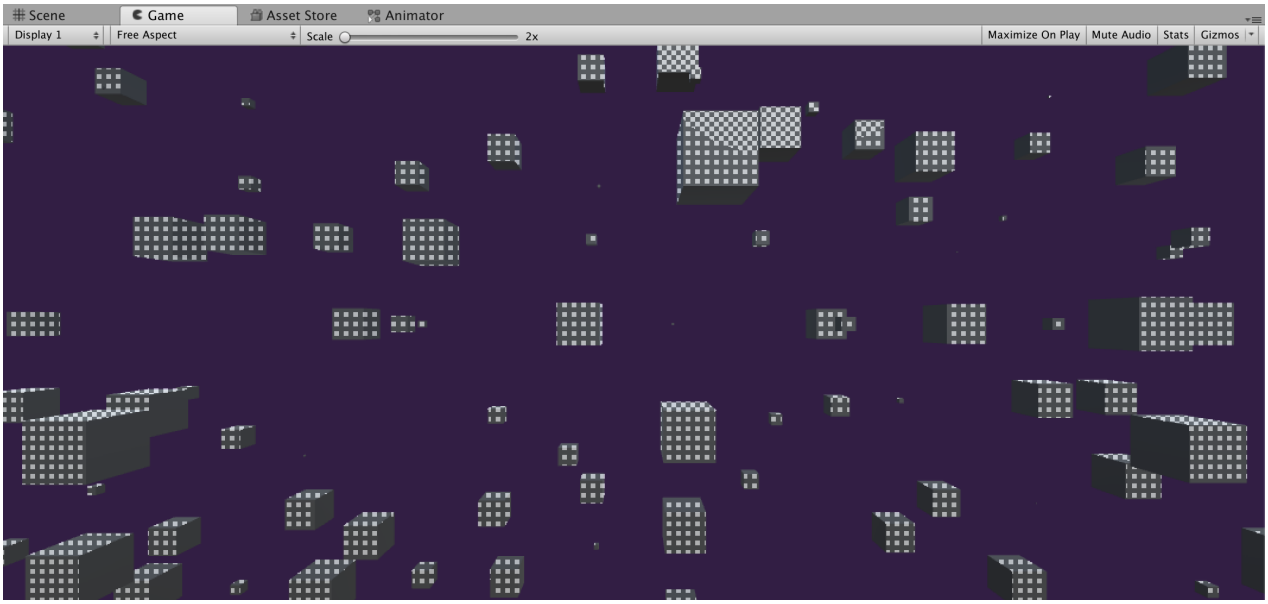


H.1 Analog Glitch



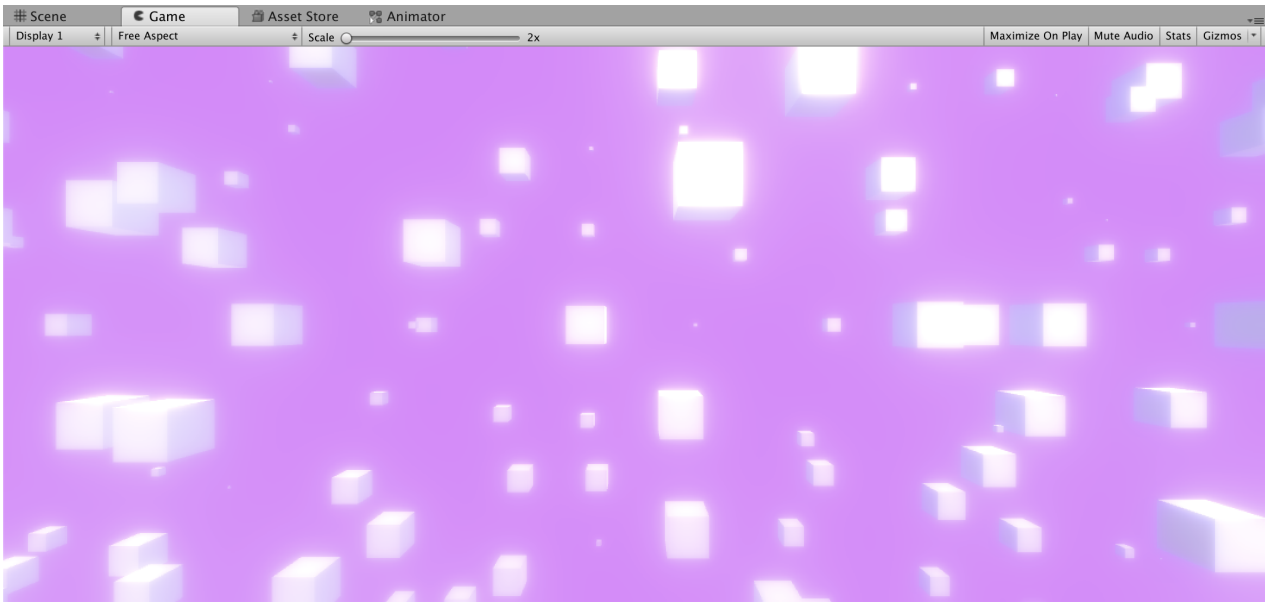
Analog Glitch simulates image glitches produced by an analog television. There are four float-valued variables: Scan Line Jitter, Vertical Jump, Horizontal Shake, and Color Drift. These variables can be manipulated by the `Float Out` node.

H.2 Binary



Binary produces an effect that mixes pixelation and opacity to produce a halftone-type effect. There is a dropdown menu allowing the user to select the Dither Type (different techniques for producing the pixelation), an int variable for the Dither Scale (how strong the pixelation is), two color variables, and a float-valued variable for opacity. Dither Scale can be manipulated using `Int Out`, the colors with `Color Out`, and the opacity with `Float Out`.

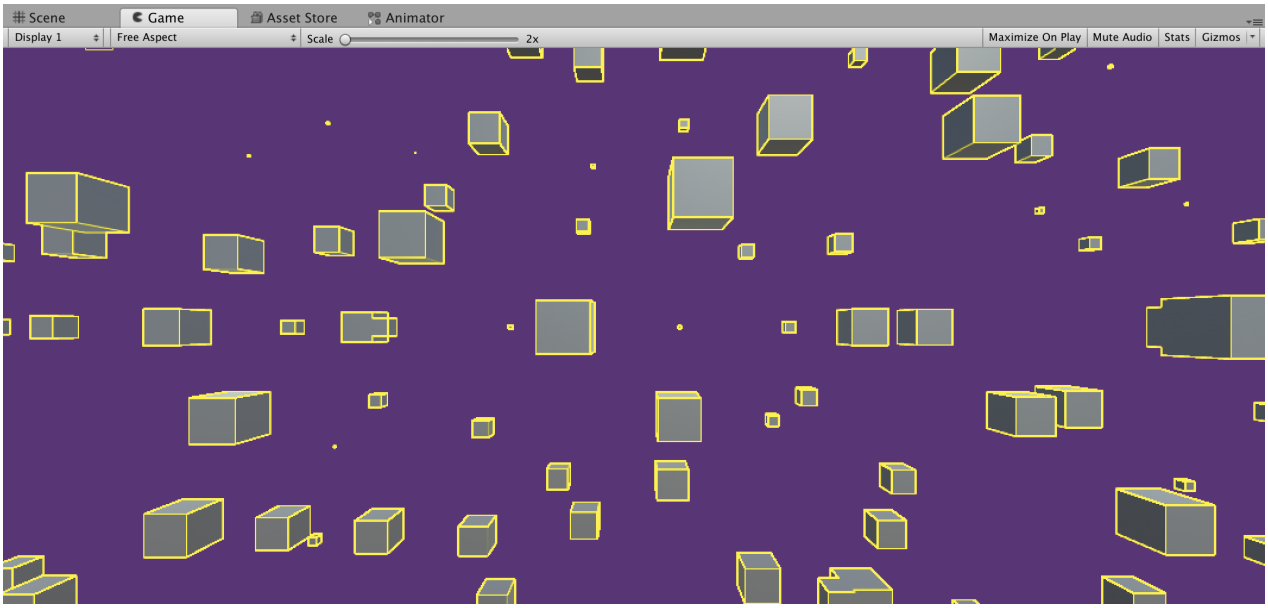
H.3 Bloom



Associated Examples: Dancer, Video

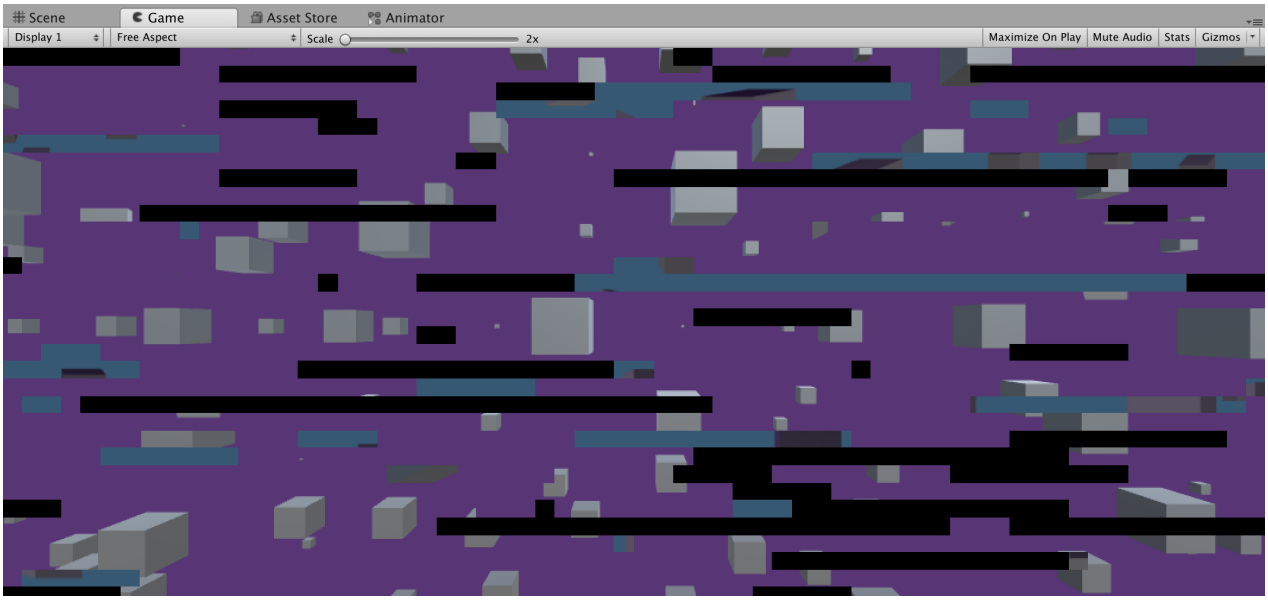
Bloom is an effect that intensifies the brightness of the image. This is achieved with a number of variables: the threshold (gamma), soft knee (ranges from 0 to 1), intensity, radius (ranges from 1 to 7), and the two boolean options high quality and anti-flicker. The first four variables are float-valued and can be manipulated with `Float Out`. The last two are buttons that can be turned on and off in a variety of ways, including `Bool Out` and `Active Status Out`.

H.4 Contour



Contour produces a highlighted contour around the scene's GameObjects. There are two color variables and six float-valued variables. They can be manipulated through the `Color Out` and `Float Out` nodes, respectively.

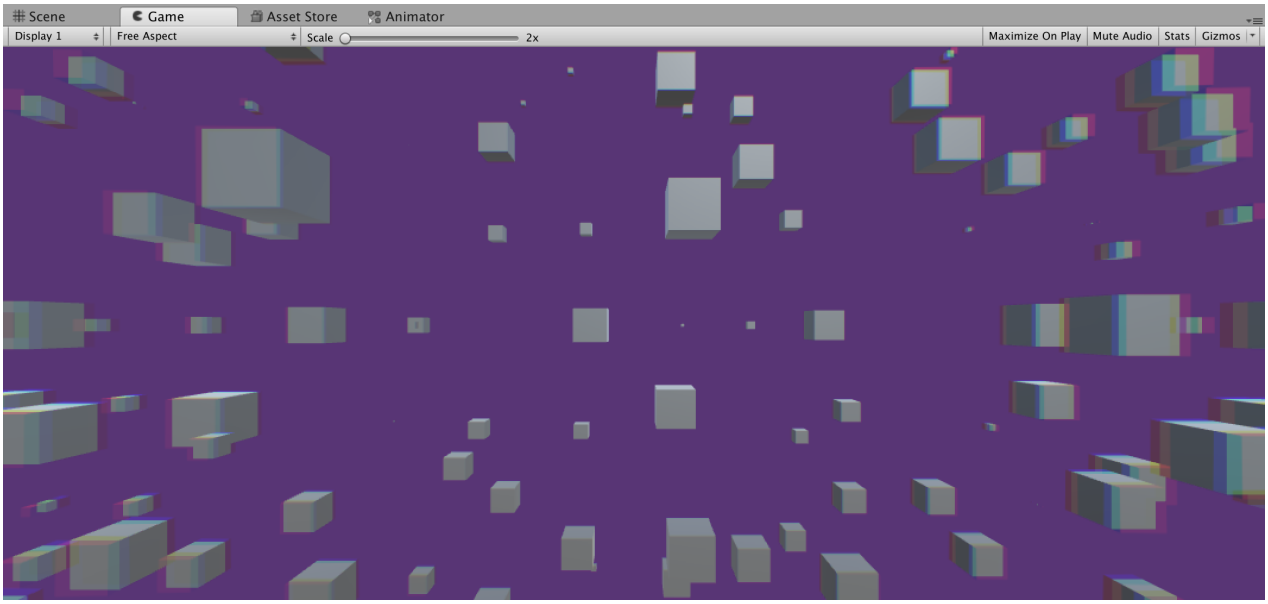
H.5 Digital Glitch



Associated Example: Splash

Digital Glitch is the digital version of analog glitch, producing broken pixelated horizontal lines on the screen. There is only one float-valued variable (ranging from 0 to 1) which can be manipulated with the `Float Out` node.

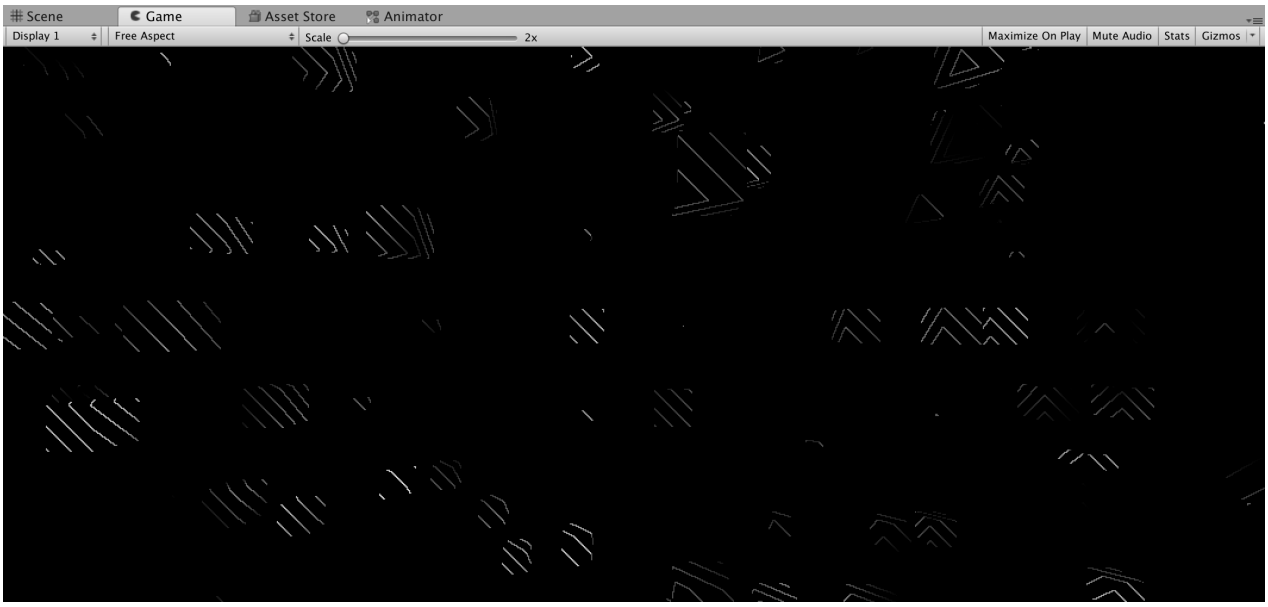
H.6 Fringe



Associated Example: Collider

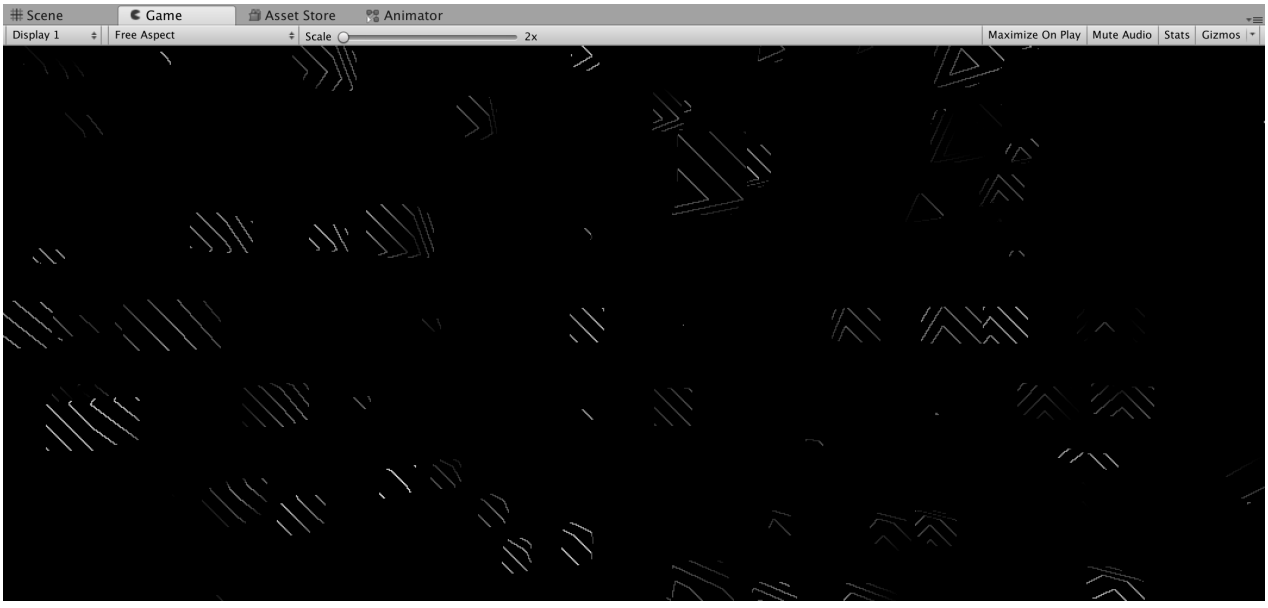
The Fringe effect does chromatic aberrations along the corners of the image. There are three float-valued variables that enable the effect, each of which can be manipulated with a `Float Out` node.

H.7 Isoline



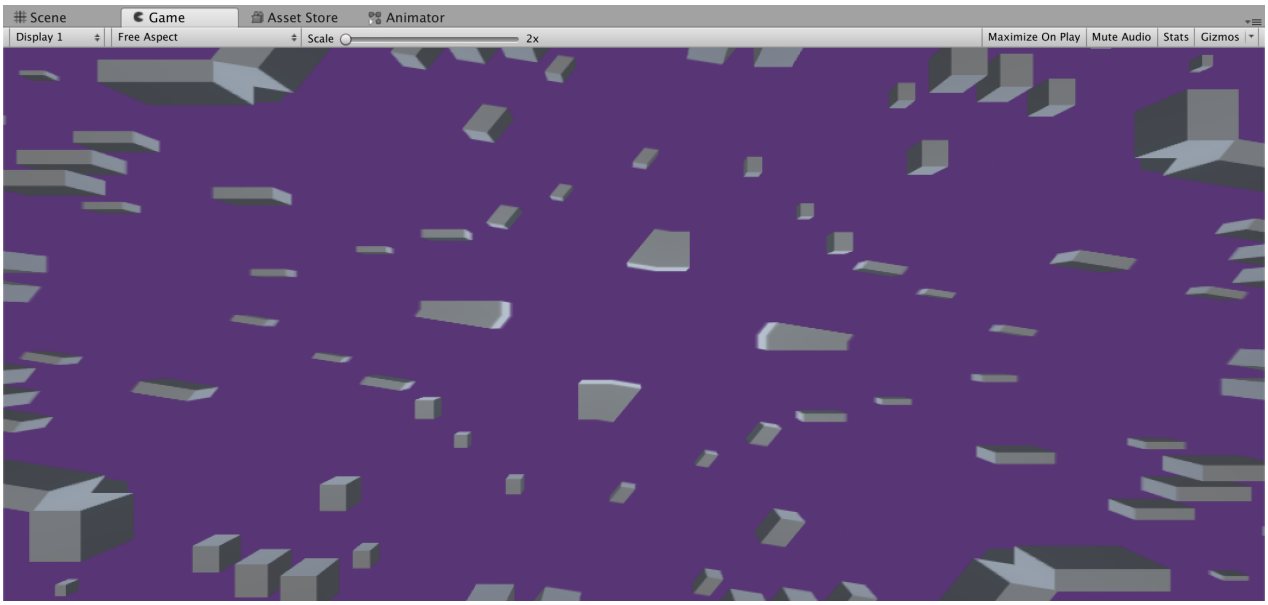
The Isoline effect casts contour lines on the game's objects. There are two colors (the background color and the line color) (`Color Out`), five float-valued variables (`Float Out`), and two vector-valued variables (`Vector Out`).

H.8 Isoline Scroller



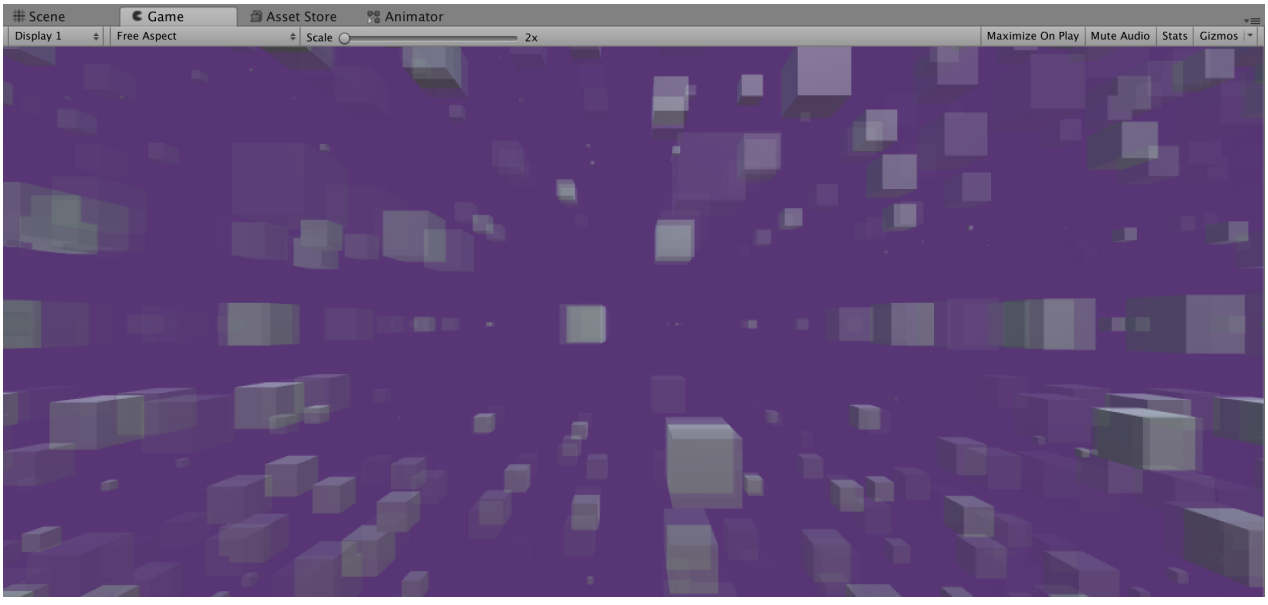
The Isoline Scroller effect depends on the Isoline effect and uses a script to cause the isolines to move along the image. There is a vector-valued variable for direction (can be manipulated with `Vector Out`, and a float-valued variable for speed (`Float Out`).

H.9 Mirror



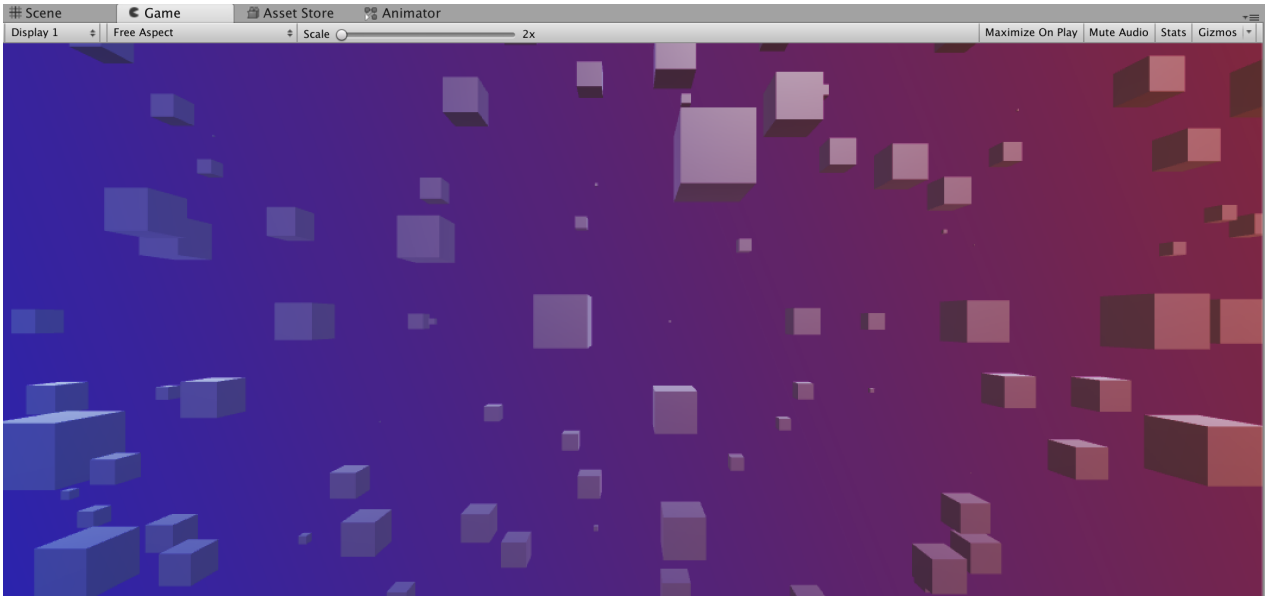
Mirror produces a kaleidoscope effect. There is an int-valued variable which denotes the number of repetitions in the effect (accessed through `Int Out`, two float-valued variables (offset and roll, accessed through `Float Out`). Finally, there is a button for the option of symmetry (accessed through `Active Status Out` or `Bool Out`).

H.10 Motion



The Motion effect is akin to taking high-action and high-speed photos. The shutter speed and exposure are manipulated, and several frames are blended to create the motion effect. There are two int-valued variables (shutter angle and sample count, both accessible through `Int Out`). Lastly, there is a float-valued variable denoting the strength of the multiple frame blending. This variable is accessible through `Float Out`.

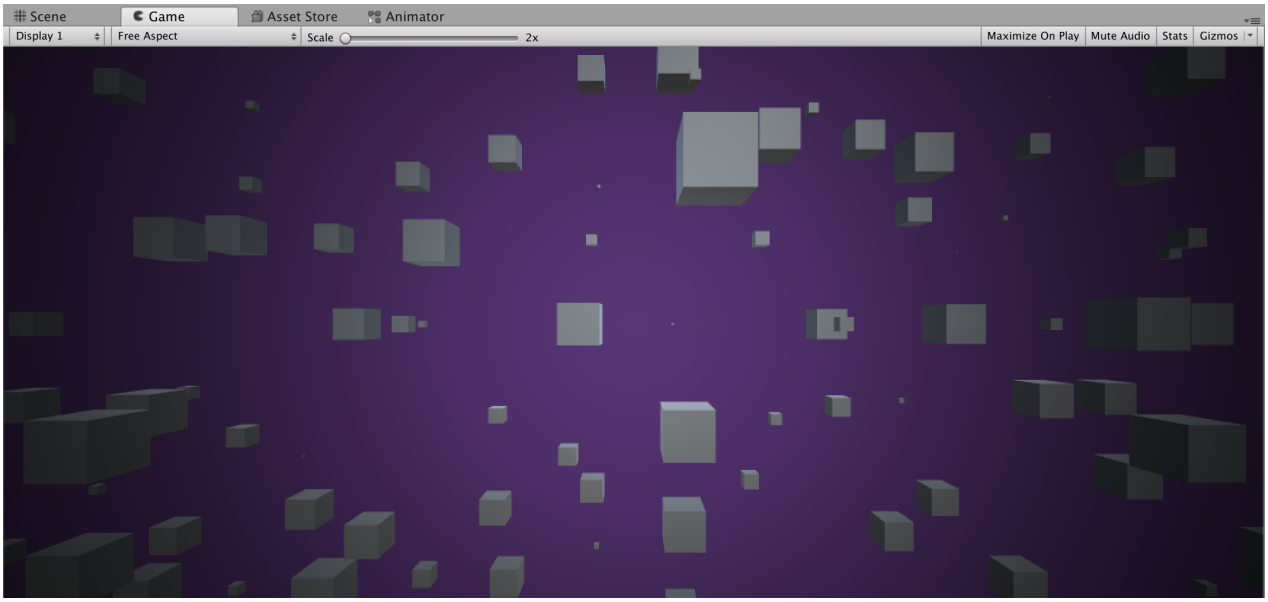
H.11 Ramp



Associated Example: Ramp

The Ramp effect blends two colors according the selected blend mode (available through a dropdown menu). The colors can be manipulated with `Color Out` nodes. The int-valued angle can be manipulated with a `Int Out` node, and the float-valued opacity variable (between 0 and 1) can be manipulated with a `Float Out` node.

H.12 Vignette



Vignette produces a vignette around the image and has a single float-valued variable called falloff, which is the intensity of the vignette (can be manipulated with a `Float Out` node).

H.13 Vision

The Vision effect inverts the image's colors depending on the source (accessible via the dropdown menu). There are two float-valued variables: repeat and blend ratio, which are accessible through the `Float Out` node. Lastly, there is a button option to use depth normals, which is accessible through the `Active Status Out` or `Bool Out` nodes.

Bibliography

[1] Andy Tanguay. (2018, November 03). OP-Z and Unity - Animation Triggering for Absolute Beginners - I mean like BEGINNER. Retrieved November 20, 2020, from https://www.youtube.com/watch?v=MV05M0Fkkws

[2] Keijiro - GitHub. Retrieved November 20, 2020, from https://github.com/keijiro/

[3] Max Basic Tutorial 2: Bang!. Retrieved November 20, 2020, from https://docs.cycling74.com/max8/tutorials/basicchapter02

[4] birthCenter Productions. (2019, January 15). OP-Z Videolab tutorial 1: Setup and Triggering Animation. Retrieved November 20, 2020, from https://www.youtube.com/watch?v=8quDzNrlviI

[5] Mixamo. Retrieved November 20, 2020, from https://www.mixamo.com/

[6] Ninety Six. (2018, October 27). OP-Z Videolab install and first steps. Retrieved November 20, 2020, from https://www.youtube.com/watch?v=-ucthiCRQRI

[7] Unity Technologies. Unity User Manual (2018.2). Retrieved November 20, 2020, from https://docs.unity3d.com/2018.2/Documentation/Manual/index.html

[8] Unity Technologies. GameObjects. Retrieved November 20, 2020, from https://docs.unity3d.com/560/Documentation/Manual/GameObjects.html

[9] Unity Technologies. The Hierarchy window. Retrieved November 20, 2020, from https://docs.unity3d.com/Manual/Hierarchy.html

[10] Unity Technologies. Meshes, Materials, Shaders and Textures. Retrieved November 20, 2020, from https://docs.unity3d.com/Manual/Shaders.html

[11] Unity Technologies. Particle System. Retrieved November 20, 2020, from https://docs.unity3d.com/Manual/class-ParticleSystem.html

[12] Unity Technologies. Playable Director component: Timeline: 1.2.17. Retrieved November 20, 2020, from https://docs.unity3d.com/Packages/com.unity.timeline@1.2/manual/play_director.html

[13] Unity Technologies. Quaternion. Retrieved November 20, 2020, from https://docs.unity3d.com/ScriptReference/Quaternion.html

[14] Unity Technologies. Scenes. Retrieved November 20, 2020, from https://docs.unity3d.com/Manual/CreatingScenes.html

[15] Unity Technologies. String. Retrieved November 20, 2020, from https://docs.unity3d.com/ScriptReference/String.html

[16] Unity Technologies. Using Components. Retrieved November 20, 2020, from https://docs.unity3d.com/Manual/UsingComponents.html

[17] Unity Technologies. Vector3. Retrieved November 20, 2020, from https://docs.unity3d.com/ScriptReference/Vector3.html

[18] Unity Technologies. Video Player component. Retrieved November 20, 2020, from https://docs.unity3d.com/Manual/class-VideoPlayer.html

[19] Unity Technologies. Video file compatibility. Retrieved November 20, 2020, from https://docs.unity3d.com/Manual/VideoSources-FileCompatibility.html

[20] Teenage Engineering – Videolab. Retrieved November 20, 2020, from https://teenage.engineering/products/op-z/videolab