DEIMOS USER STUDY REFERENCE DOCUMENT (Immersive Data Visualisation Morphs and Transitions)

Thank you for taking the time to participate in our user study! This document contains instructions for the asynchronous portion of the user study, including walkthroughs and reference material for how to use the Deimos toolkit (Dynamic Embodied Interactive MOrphS).

SOFTWARE AND FILES TO DOWNLOAD

- Deimos Unity project files
  - Download ZIP or clone via Git
- Unity 2021.3.17f1
  - Any other Unity 2021.3.x version should also work
- Visual Studio Code
  - We strongly recommended VSCode for IntelliSense to provide auto-complete suggestions with JSON schemas
  - Visual Studio 2022 *should* also work with this
  - See here for a list of supported editors with JSON schemas

INSTRUCTIONS

Steps to take:

1. Ensure you have the above software and files downloaded
2. Follow the three Walkthroughs to familiarise yourself with Deimos and its grammar
   - *DxR Grammar and Extended Functionalities*, *Basic Morphs*, and *Advanced Morphs*
   - Some parts of the first one can be skipped if you are comfortable with the DxR and/or Vega-Lite grammar
3. Browse through the examples presented in the *ExampleMorphsScene* scene (found in *Assets/Deimos/Examples*)
   - You may test them for yourself, study how they work, and/or recreate these on your own
4. Create your own morphs either from scratch or based on the suggestions provided in Self-Exploration section
   - You should use the *UserStudyScene* scene (found in *Assets/Deimos/User Study*)
   - Please save any morphs which you create, whether successful or not, in this scene. We would like you to show these to us in the interview Zoom session
   - More resources are provided in the Frequently Asked Questions and Grammar Reference sections
5. OPTIONAL: Test these morphs in VR
   - Note that the toolkit has only been tested with an Oculus Quest 2 via Oculus Link, and not as a standalone build in neither VR or MR

Things to keep in mind:

- Please note down any thoughts you might have, whether good or bad, as you test Deimos.
  - We are broadly interested in the usability of the toolkit, the toolkit's utilisation of key immersive analytics characteristics, and future research directions and improvements
- Feel free to email us directly (benjamin.lee1@monash.edu) if you encounter any major roadblocks or bugs that impede your progress
- Remember that Deimos is not intended to be production ready, so expect some jank and bugs here and there!

WALKTHROUGHS

DxR Grammar and Extended Functionalities

Deimos uses DxR for its data visualisations, therefore knowledge of the DxR grammar is necessary for proper usage of the toolkit. We highly recommend you to go through the DxR Grammar Docs, especially the examples.

Note that the DxR Interactions and GUI features are currently unsupported in Deimos and are likely prone to breaking or completely non-functional. Please ignore these features for the purposes of this study.

Creating Visualisations at Runtime

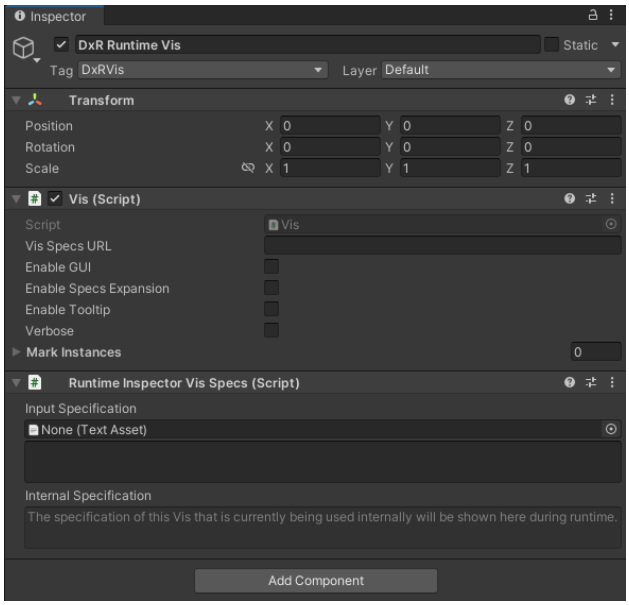
Helper scripts are provided to help you easily author and edit DxR visualisations and their specs (in JSON) at runtime. We highly recommend you make use of them—they are there for a reason! As an aside, base DxR includes similar functionality as well, but was not included due to it not being in the branch which we initially forked the project from.

To create a DxR visualisation GameObject, open the context menu in the Hierarchy, and select *Morphs* > *DxR Runtime Vis*. The new GameObject will have a *RuntimeInspectorVisSpecs* component. There are two ways of supplying a vis spec: 1) as a JSON file in the Assets folders; or 2) as a string in the inspector itself.

To use a JSON file, open the context menu in the Project panel, and select *Create* > *Morphs* > *DxR Vis Specification* (or create your own file manually). Then drag and drop this new file into the *Input Specification* parameter in the *RuntimeInspectorVisSpecs* component. Its contents should be visible in the textbox below. This JSON file will now be used to generate the visualisation. If any changes are made to the file during runtime, you can click on the *Update Vis* button to update the visualisation. The JSON file itself will never be modified by Deimos directly.

To use a string, simply type (or copy and paste) the specification text into the large textbox. Visualisations can also be updated at runtime using the *Update Vis* button. Note that this manner of input does not work if a JSON file is set. You can however click on the *Remove JSON File* button to remove the JSON file reference and easily edit the specification in the Inspector, which can be useful if you do not want to modify the file itself.

Datasets can be referenced by their filename and extension alone, so long as the dataset itself is stored in the *Assets/StreamingAssets/DxRData* folder.



Base DxR Runtime Vis component

DxR Runtime Vis component with a visualisation specification defined via textbox

```
{
    "mark": "sphere",
    "width": 500,
    "height": 500,
    "depth": 500,
    "data": {
      "url": "cars.json"
    },
    "encoding": {
      "x": {
        "field": "Horsepower",
        "type": "quantitative"
      },
      "y": {
        "field": "Miles_per_Gallon",
        "type": "quantitative"
      },
      "z": {
        "field": "Acceleration",
        "type": "quantitative"
      }
    }
}
```

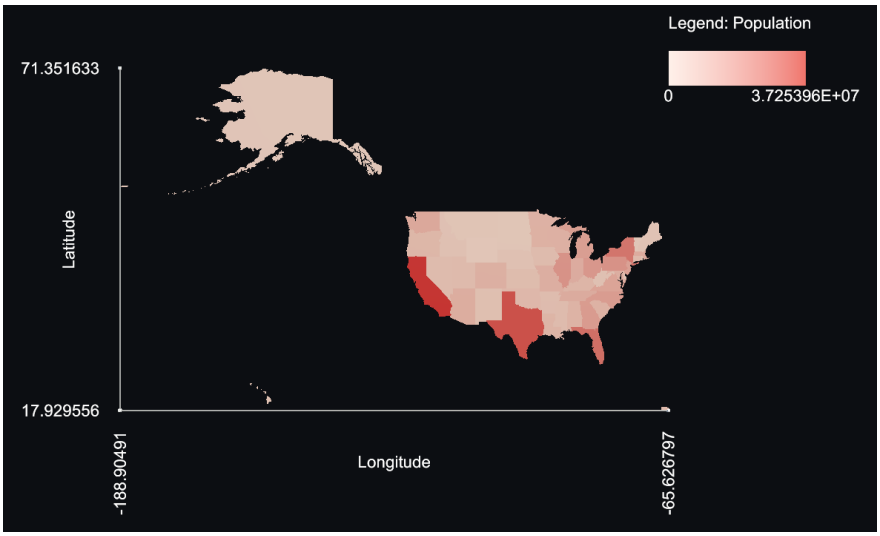
Sample DxR specification of a 3D scatterplot (shown in above inspector image)

Maps

Deimos includes (basic) support for maps. Data should be provided in GeoJSON format. Two datasets, *us-states.json* and *aus-states.json*, are included. To properly parse and visualise this data, the *geoshape* mark should be used.

geoshape exposes the *Longitude* and *Latitude* fields and the *spatial* channel type. Setting a spatial encoding (i.e., x, y, z) to *Longitude/Latitude* and *spatial* will position each mark at each polygon's centroid. Setting a size encoding (i.e., width, height, depth) *Longitude/Latitude* and *spatial* will draw the actual polygon. Note that non-spatial fields will still work with *geoshape*, meaning that you can set the position or size to other data fields, with the exception of the depth encoding which will produce artefacts.

```
{
    "width": 800,
    "height": 500,
    "mark": "geoshape",
    "data": {
        "url": "us-states.json"
    },
    "encoding": {
        "x": {
            "field": "Longitude",
            "type": "spatial"
        },
        "y": {
            "field": "Latitude",
            "type": "spatial",
            "scale": {
                "type": "linear",
                "domain": [ 15, 75 ]
            }
        },
        "width": {
            "field": "Longitude",
            "type": "spatial"
        },
        "height": {
            "field": "Latitude",
            "type": "spatial"
        },
        "color": {
            "field": "Population",
            "type": "quantitative"
        }
    }
}
```



Example DxR Choropleth Map Specification

Side-by-Side and Stacked Barcharts

Deimos includes support for side-by-side and stacked barcharts by extending DxR's offset encodings. Note that as DxR does not support data transformations (and neither does Deimos), manually formatting data suitable for these barcharts can be a pain. The *titanic_2d.csv* and *titanic_3d.csv* datasets are included for testing purposes.

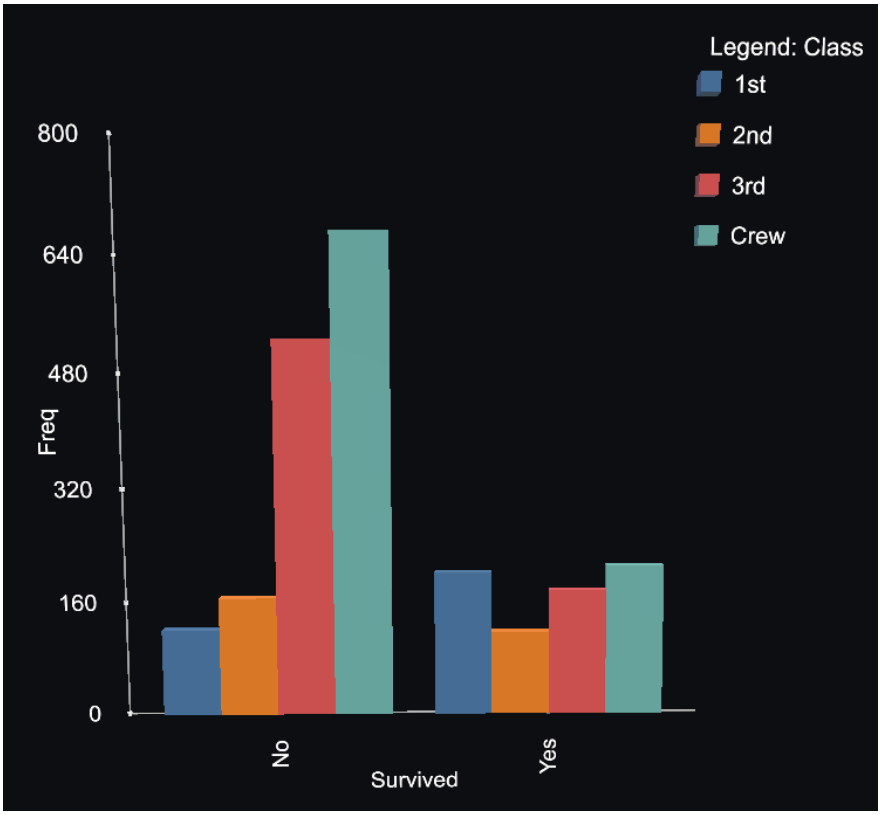
To create a side-by-side barchart, apply an *offset* encoding along the spatial dimension that the bars should be arranged along. For example, if bars are to be arranged from left to right, the *xoffset* encoding should be used. In this, the *field* should be set to the appropriate category (typically the same field as colour), and the *type* be set to either *nominal* or *ordinal.*

To create a stacked barchart, the same process applies, except the spatial dimension is the one which the bars are "stacked" along. For example, if the bars are to be stacked vertically, the *yoffset* encoding should be used.

These encodings can theoretically be applied to non-barcharts as well, although this has not been rigorously tested.

(see next page)

```
{
    "width": 600,
    "height": 600,
    "mark": "cube",
    "data": {
        "url": "titanic_2d.csv"
    },
    "encoding": {
        "x": {
            "field": "Survived",
            "type": "nominal"
        },
        "width": {
            "value": 60
        },
        "y": {
            "field": "Freq",
            "type": "quantitative",
            "scale": {
                "type": "linear",
                "domain": [0, 800]
            }
        },
        "height": {
            "field": "Freq",
            "type": "quantitative",
            "scale": {
                "type": "linear",
                "domain": [0, 800]
            }
        },
        "yoffsetpct": {
            "value": -0.5
        },
        "color": {
            "field": "Class",
            "type": "nominal"
        },
        "xoffset": {
            "field": "Class",
            "type": "nominal"
        },
        "xoffsetpct": {
            "value": -1.5
        }
    }
}
```



Example DxR Side-by-Side Barchart Specification

Faceted Charts

Lastly, Deimos includes support for faceted charts (small multiples), in both flat and curved layouts. Our implementation is loosely based on facet_wrap() in ggplot2. Note that facet_grid() is not implemented.

To facet a chart, the *facetwrap* encoding should be used. The *field* should be set to the faceting variable, and the *type* set to either *nominal* or *ordinal*. This will default to sensible values, positioning small multiples along the x, y plane (i.e., position from left to right, then wrapping around from bottom to top).

Further properties can be specified for further customisation. *directions* can be specified to define the plane in which small multiples are positioned. This is an array of only two unique string values, each value being either “x”, “y”, or “z”. The first element is the direction of the ribbon, the second element is the direction which the ribbon wraps around. A *size* property can also be specified to denote the number of small multiples until the ribbon wraps around.

Additional properties are used for curved layouts. *angles* can be specified for each of the aforementioned directions to denote the angle between each small multiple. This is an array of two numbers in degrees. To use a flat layout for a given direction, an angle of 0 should be set (to allow for cylindrical layouts). A *radius* can also be set to specify the radius of the curve in millimetres. Lastly, the *orientation* property can be set to *centre* to have all small multiples in a curved layout face towards the centre. Note however that this last feature is somewhat buggy and is prone to breaking.

```
{
    "mark": "sphere",
    "data": {
        "url": "cars.json"
    },
    "encoding": {
        "x": {
            "field": "Miles_per_Gallon",
            "type": "quantitative"
        },
        "y": {
            "field": "Weight_in_lbs",
            "type": "quantitative"
        },
        "facetwrap": {
            "field": "Year",
            "type": "nominal",
            "directions": ["x", "y"],
            "radius": 1250,
            "angles": [30, 0],
            "orientation": "centre"
        }
    }
}
```

Example DxR Curved Faceted Scatterplot Specification

MRTK Simulator

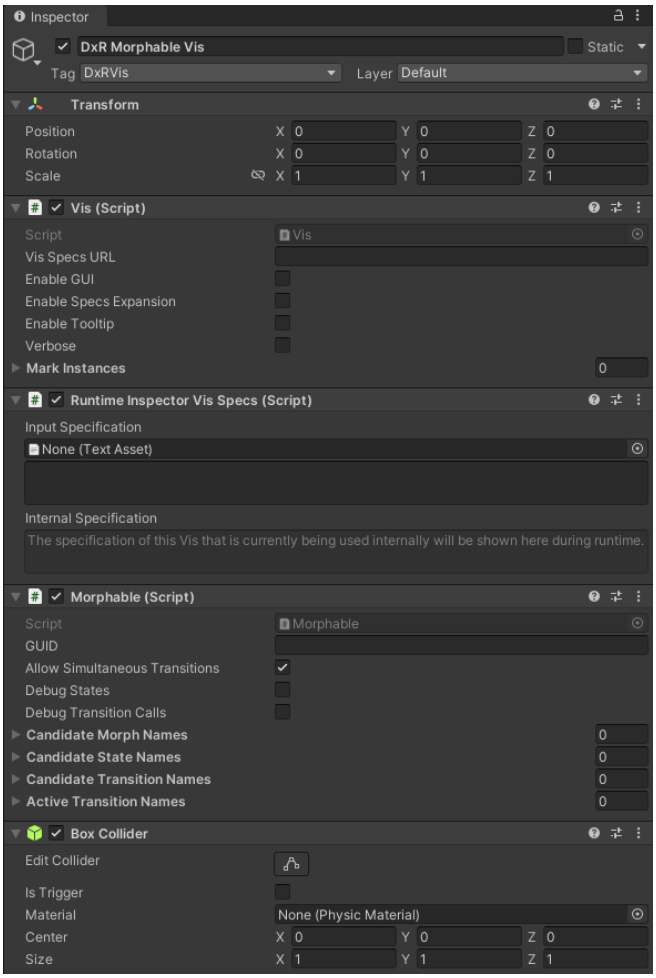
Deimos uses MRTK 2 for XR functionality. MRTK 2 provides an input simulator to test interactions without needing to don a VR/MR headset. For the purposes of this study, we do not expect you to use a headset and instead recommend using the input simulator for rapid prototyping. Microsoft provides

documentation for how to use the input simulator. In short, controls are the same as in the Unity Scene panel, with spacebar enabling the right hand and left shift enabling the left hand.

Basic Morphs

Let's now get started on actually creating some Morphs with Deimos! Let's create a very basic example: change the colour of all uncoloured marks to red when the left mouse button is pressed down.

Open the *UserStudyScene* scene in *Assets/Deimos/User Study*. In this, we need to create our Vis GameObject and JSON specification. Open the Hierarchy context menu and select *Morphs* > *DxR Morphable Vis*. This is the same as the *Runtime* version but has an additional component necessary for Morphs to function. Next, open the Project context menu and select *Create* > *Morphs* > *Morph Specification.* Open this new JSON file in a code editor, ideally VS Code. You should see three sections already populated for you. Hopefully your code editor supports JSON Schemas which enables autocompletion and tooltips, which will greatly help this process.



Base DxR Morphable Vis component

```
{
    "$schema":
"https://raw.githubusercontent.com/benjaminchlee/Deimos
/master/Assets/Deimos/Schema/morph-schema.json",
    "name": "New Morph Specification",

    "states": [
        {
            "name": "first state"
        },
        {
            "name": "second state"
        }
    ],

    "signals": [

    ],

    "transitions": [
        {
            "name": "transition",
            "states": ["first state", "second state"]
        }
    ]
}
```

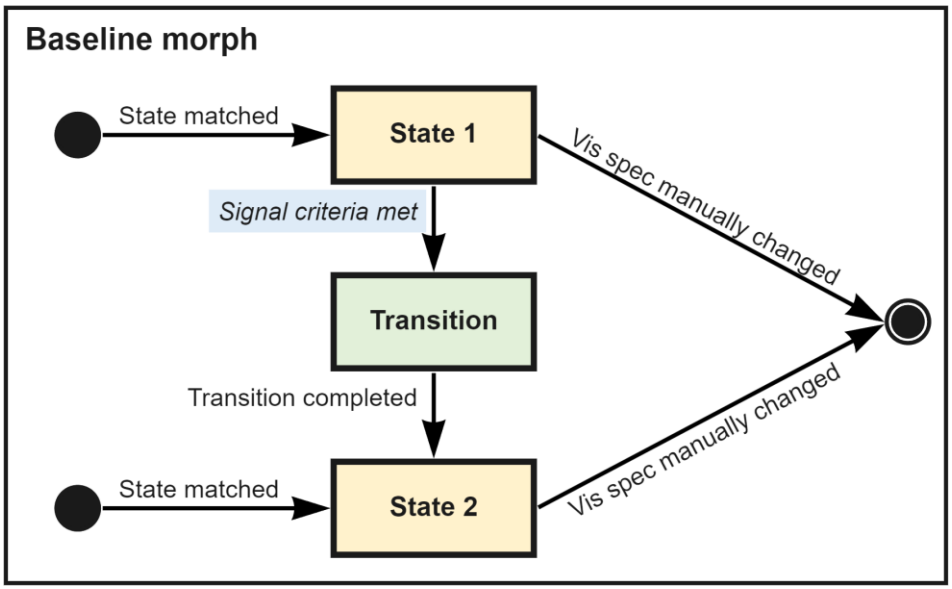
Default Morph Specification

General Concept

Deimos' grammar works off the concept of a state machine. Within each Morph, *States* are defined that act as animation keyframes. These States are only accessible when a visualisation, at runtime, matches some criteria. When a given State is accessed, it is eligible to have a *Transition* applied to it provided some other criteria is met. This is the actual animation. This process is controlled by *Signals* which act as both the aforementioned criteria as well as ways for the user to affect the Morph. We refer to the overall state machine as a *Morph*. A Morph contains one or more *Transitions* within it.

The concept of Morphs exists separately from any functional visualisation editor of the system it resides in. That is, the end user can access and use these Morphs while still being able to create and modify visualisations as per usual (e.g. in JSON for DxR). Whenever the visualisation specification is manually edited by the user, the state machine exits the Morph. It may then immediately re-enter the Morph via a State if it still matches any one of them.

Multiple Morphs (and Transitions) can be applied to any given visualisation at the same time, meaning that there are multiple state machines existing concurrently. Each Morph can only have one active State at any time. however.



General state machine for Deimos Morphs and Transitions in a simplified case. More States and Transitions can be added, with additional rules for which States can be accessed and how

States

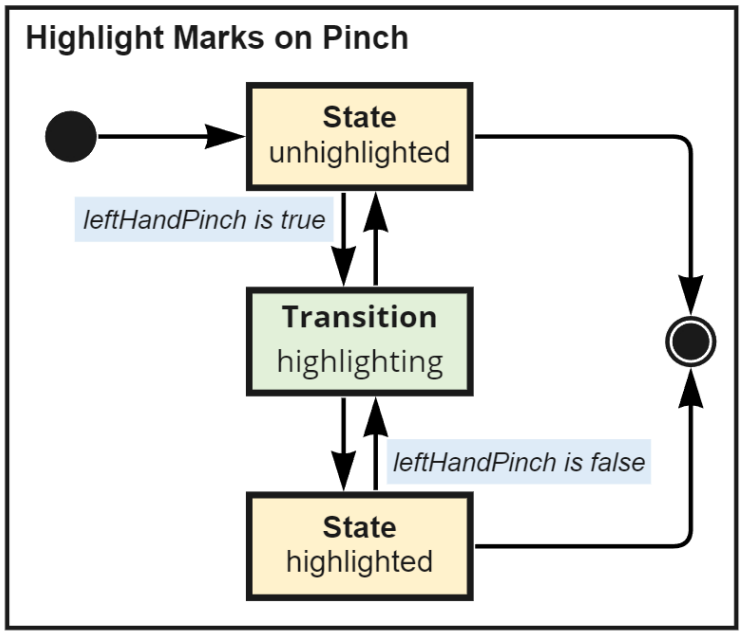
We first need to define the States in our Morph. States are essentially keyframes in our animated transitions. However, the States we are defining here are not complete visualisation specifications but are instead *partial* states. Our visualisations (defined by DxR vis specs) will, at runtime, pattern match against these partial states, and will have Morphs applied to them should they actually match. You can think of these as the minimum set of visualisation encodings (à la visualisation idioms) that still allows the Morph to be valid.

We specify visualisation properties and encodings for these partial states using the same grammar as a DxR visualisation. Any property (and its sub-properties) defined in a State will need to match the visualisation's specification for the Morph to be applied. Any property that has *null* as a value means that the visualisation's specification should not have that property. We can either be as specific (i.e., define many properties in the State) or as vague (i.e., define little properties in the State) as we want. Obviously the more vague we are, the wider range of visualisation configurations it can apply to at runtime.

In our example we have two States: a State where the marks are uncoloured, and a second State where the marks are coloured red. Therefore, in our first State we add a *colour* encoding with its value set to *null*. This means that the first State is not accessible in the state machine unless the visualisation does not have any colour encoding. In our second State we add a *colour* encoding as well, but with its *value* set to *red*. You can also give these a more descriptive *name* as well. We also add a *restrict* property in the second state and set it to true. What this functionally does is prevent that state from being directly accessed by the state machine except through a Transition. This can be useful if we want a Morph to only start from certain States and not from others.

```
"states": [
    {
        "name": "unhighlighted",
        "mark": "sphere",
        "encoding": {
            "color": null
        }
    },
    {
        "name": "highlighted",
        "restrict": true,
        "encoding": {
            "color": {
                "value": "red"
            }
        }
    }
]
```

Specification for States



Effect of the *restrict* flag on the state machine. It prevents the given state (highlighted) from being an entry point to the Morph state machine.

Deimos uses these states to generate the keyframes to perform the Transition later on. It should hopefully be clear that the initial keyframe will have the uncoloured visualisation, and the final keyframe be a red coloured visualisation. And that should be about it! Notice how we did not need to specify any other encodings, but we could if we want the Morph to be applied only to more specific situations (e.g. only on scatterplots with *x* and *y* encodings, or only on maps with the *geoshape* mark type).

Signals

Signals form the basis for the control and manipulation of Morphs by end-users. Signals can more easily be thought of as events which generate some sort of data (e.g. boolean, float, string). In our example, we want our Morph to respond to a user left mouse button press, therefore we need to create a Signal which emits data to this effect.

In the Signals section of our Morph specification, create a new object and give it a descriptive *name* property (e.g. leftMouseDown). We then need to provide it a *source*, which is as the name suggests the source of the data we want. Set its value to *mouse* (autocomplete will show you the other options available but we'll skip these). Since we want to specify a left mouse button press, we can also set a *handedness* property of *left*. Lastly, we want to get a boolean value which denotes whether the user is pressing the button down or not. Add a *value* property and set it to *select*.

```
"signals": [{
        "name": "mousedown",
        "source": "mouse",
        "handedness": "left",
        "value": "select"
    }
]
```

Specification for Signals

This is the simplest form of Signal available, but it will do for our purposes. This Signal can be referenced elsewhere in the Morph specification by *name*, usually as a variable in mathematical expressions. As such, **Signal names should not have spaces in them**, otherwise these expressions will not work as expected. All other names can have spaces.

Transitions

Transitions actually define the animated transition itself. These bridge two States together and are controlled by Signals, while also providing some parameters to control the Transition's behaviour. In our example we'll be creating a Transition to connect the two States we already have in the state machine.

In the Transition section of the specification, create a new object and give it a descriptive *name* (e.g. highlighting). Next, we define our *states* property. This requires an array of two strings. The first value is the initial State in our Transition, and the second value is the final State. The strings here correspond to the *name* properties of our two States we had created earlier (case sensitive). The order here matters as by default these Transitions are unidirectional! These two properties are the bare minimum we need for a Transition to be enabled. Without specifying anything else, our visualisation which matches the first uncoloured state will instantly transition to the second coloured state. We of course want our Morph to be more sophisticated than that!

Let's now define our *trigger* property. As the name suggests, this is the condition which needs to be met for the Transition to actually initiate. This property needs a Signal which emits Boolean values, so we enter the *name* of the Signal we had created for the left mouse button press. The Transition will now only take place when the mouse button is actually pressed, rather than instantaneously.

Next, we define a set of *control* parameters as a JSON object (curly braces). This *control* object supports several properties, but we'll use *timing* and *interrupted*. *timing* takes in either the name of a Signal which emits float values between 0 and 1 (i.e., a tweening value), or a number which signifies the duration of the Transition. We'll keep things simple and set the value to *0.25*. *interrupted* sets the rule for what happens when the *trigger* is no longer *true* whilst the Transition is being applied. We will set it to *ignore* so that it keeps progressing even if the user releases the mouse button.

Lastly, we set the *bidirectional* property to *true*. This indicates to Deimos that the Transition is valid in both directions, creating a bidirectional connection in the state machine. The reverse direction will use the same parameters we have already defined, but instead requires a *false* value from the *trigger* instead. For more control over how the reverse Transition behaves, you can instead create two unidirectional Transitions instead, each one with their own parameters.

```
"transitions": [
    {
        "name": "highlighting",
        "states": ["unhighlighted", "highlighted"],
        "trigger": "mousedown",
        "control": {
            "timing": 0.25,
            "interrupted": "ignore"
        },
        "bidirectional": true
    }
]
```

Specification for Transitions

And that should be all we need to define in our Morph! Of course, the specification alone will not enable the Morph. We need to do some things in Unity.

Putting it all together

Back in Unity, find the MorphManager GameObject in the scene (or create a new one in the Inspector context menu *Morphs* > *Morph Manager*). The MorphManager component stores the references of all Morph specifications you have created and whether or not they are enabled. Any specification that has *Enabled* unchecked will not be loaded and will therefore be inactive during the session. Add a new element to the *Morph Json Specifications* list in the inspector, and drag and drop your Morph specification file into the new slot. Make sure that *Enabled* is also ticked.

Lastly, we need to create a DxR specification for the *DxR Morphable Vis* GameObject which we created at the start. Create a simple visualisation without a colour encoding using either of the two methods as explained earlier, such as a scatterplot.

And now all you should need to do is press play, and left mouse click anywhere in the Game panel. You should now see your visualisation changing colour! Feel free to mess around with the Morph specification you have just created as well, such as adding new *encodings* to the States or changing the *timing*. Any changes which you make to the Morph specification during runtime can be updated by clicking the *Refresh Morphs* button in the *MorphManager* component (remember to save the specification file!).

Advanced Morphs

Let’s now look at a more complex example of a Morph which will reveal more complex features of Deimos and its grammar: partitioning a 3D barchart into 2D facets when it collides with a flat surface.

Go ahead and create a new *Morph Specification* and *DxR Vis Spec*. You may use the same scene as the previous walkthrough, although you might also want to create a new *DxR Morphable Vis* as well. One thing to make sure of is that you give your new specification a unique *name* (prefilled as “New Morph Specification” at the top).

States - Special Operators

Let’s now create our States. We clearly have two States: a 3D barchart and a 2D faceted barchart. We’ll need to create these two States in the specification and fill in their relevant properties and encodings. However, in this case it is easier if we work backwards — we will begin with the visualisation specifications and identify what we should then transfer over to our Morph specification. Note that in this case we are using *stacked* versions of these two barcharts, which are even more specific than just a regular barchart!

(see next page)

```
{
    "width": 300,
    "height": 400,
    "depth": 600,
    "mark": "cube",
    "data": {
        "url": "titanic_3d.csv"
    },
    "encoding": {
        "x": {
            "field": "Sex",
            "type": "nominal"
        },
        "width": {
            "value": 130
        },
        "y": {
            "field": "Freq",
            "type": "quantitative",
            "scale": {
                "type": "linear",
                "domain": [0, 900]
            }
        },
        "height": {
            "field": "Freq",
            "type": "quantitative",
            "scale": {
                "type": "linear",
                "domain": [0, 900]
            }
        },
        "yoffset": {
            "field": "Survived",
            "type": "nominal"
        },
        "yoffsetpct": {
            "value": -0.5
        },
        "z": {
            "field": "Class",
            "type": "nominal"
        },
        "depth": {
            "value": 130
        },
        "color": {
            "field": "Survived",
            "type": "nominal",
            "legend": "none"
        }
    }
}
```

3D Stacked Barchart Visualisation Specification

```
{
    "width": 300,
    "height": 400,
    "mark": "cube",
    "data": {
        "url": "titanic_3d.csv"
    },
    "encoding": {
        "x": {
            "field": "Sex",
            "type": "nominal"
        },
        "width": {
            "value": 130
        },
        "y": {
            "field": "Freq",
            "type": "quantitative",
            "scale": {
                "type": "linear",
                "domain": [0, 900]
            }
        },
        "height": {
            "field": "Freq",
            "type": "quantitative",
            "scale": {
                "type": "linear",
                "domain": [0, 900]
            }
        },
        "yoffset": {
            "field": "Survived",
            "type": "nominal"
        },
        "yoffsetpct": {
            "value": -0.5
        },
        "depth": {
            "value": 10
        },
        "color": {
            "field": "Survived",
            "type": "nominal",
            "legend": "none"
        },
        "facetwrap": {
            "field": "Class",
            "type": "nominal",
            "directions": ["x", "y"]
        }
    }
}
```

2D Faceted Stacked Barchart Visualisation Specification

Feel free to give these visualisations a try to see what they look like.

We could technically just copy and paste these specifications directly into our two States and be done with it. However, obviously this means our Morph would only function with these two exact visualisations and nothing else. Therefore, let's reduce these specifications down to their minimum set of properties that still adequately describe the visualisation idioms of a 3D barchart and a 2D faceted barchart (without any stacking). This allows us to have this partitioning Morph be applied to any visualisations of these two idioms, regardless of their fields and data values.

```
{
    "name": "3DBarchart",
    "encoding": {
        "x": {
            "type": "nominal"
        },
        "y": {
            "type": "quantitative"
        },
        "height": {
            "type": "quantitative"
        },
        "yoffsetpct": {
            "value": -0.5
        },
        "z": {
            "type": "nominal"
        },
        "depth": {
            "value": 130
        }
    }
},
```

Reduced 3D Barchart Specification

```
{
    "name": "facetedBarchart",
    "encoding": {
        "x": {
            "type": "nominal"
        },
        "y": {
            "type": "quantitative"
        },
        "height": {
            "type": "quantitative"
        },
        "yoffsetpct": {
            "value": -0.5
        },
        "z": null,
        "depth": {
            "value": 10
        },
        "facetwrap": {
            "type": "nominal",
            "directions": ["x", "y"]
        }
    }
```

Reduced 2D Faceted Barchart Specification

A lot shorter now, huh!? Notice how many of the *field* properties and some *encodings* have been removed. These States will now match any of the two visualisation idioms regardless of their fields, so long as their types (as well as size values) match.

However, this poses two problems for our Morph:

1. A depth of 130 is still far too specific
2. The second State (faceted barchart) introduces new *encodings*, but our keyframe generator does not know what *field* these *encodings* should be set to

To solve the first problem, we can use the *wildcard* operator. By replacing the *130* with a "*" (as a string), we signify to Deimos that it doesn't matter what value the visualisation has for this property, as long as it actually has it. This wildcard can be used in any property, even for entire encodings (e.g. "x": "*"). Alternatively, we can give it an *inequality expression* to signify that so long as the visualisation matches the inequality for that property, then it can match. We do this by replacing the *130* with ">= 130" (as a string), or something similar. The variable is always on the left hand side, and this only works on numeric properties.

To solve the second problem, we can use *JSON path accessor* operators. We can set any JSON value in our State specification with a JSON path (separated by full stops (".")). Doing so will replace said value with whichever value is found at that JSON path when the keyframes are generated. However, JSON paths need to be prefixed with either "this." or "other." which will indicate which of the two keyframes (i.e.,

State) to access this value from: the same State that is declaring that *accessor*, or the other State in the Transition. In our example, we have two of these we need to add. Add a *field* property to the *z* encoding in the 3D barchart State and set its value to “other.encoding.facetwrap.field”, and do the same thing for the *facewrap* encoding in the 2D faceted barchart State with “other.encoding.z.field”. Through this, Deimos will know what *field* value to set to the new target keyframe, meaning we don’t need to hard code anything!

Another approach we could also do is to use a Signal which emits a string in place of the *field* value here, and Deimos will automatically set whatever value that was emitted as the *field*. We leave this for you to try out!

And that about does it! You can see the final version of our State specifications below. You may notice that there are two additional properties in the faceted barchart State however, *position* and *rotation*. These are used to position and rotate our barchart such that when the Transition is applied, it moves towards and attaches itself to the surface it touches. The *values* that are currently set are the names of Signals that we have yet to create, so leave them as-is for now.

One last thing to mention is that Deimos saves these generated keyframes during the lifespan of a Morph. This means that if a visualisation transitions in reverse from the faceted barchart State back to the 3D barchart State, the original starting properties of the 3D barchart will be reused again.

```
"name": "3DBarchart",
"encoding": {
    "x": {
        "type": "nominal"
    },
    "y": {
        "type": "quantitative"
    },
    "height": {
        "type": "quantitative"
    },
    "yoffsetpct": {
        "value": -0.5
    },
    "z": {
        "field": "other.encoding.facetwrap.field",
        "type": "nominal"
    },
    "depth": {
        "value": "*"
    }
}
```

Complete 3D Barchart State Specification

```
{
    "name": "facetedBarchart",
    "restrict": true,
    "encoding": {
        "x": {
            "type": "nominal"
        },
        "y": {
            "type": "quantitative"
        },
        "height": {
            "type": "quantitative"
        },
        "yoffsetpct": {
            "value": -0.5
        },
        "z": null,
        "depth": {
            "value": 10
        },
        "facetwrap": {
            "field": "other.encoding.z.field",
            "type": "nominal",
            "directions": ["x", "y"]
        }
    },
    "position": {
        "value": "touchedSurfaceIntersectionPoint"
    },
    "rotation": {
        "value": "touchedSurfaceRotation"
    }
}
```

Complete 2D Faceted Barchart State Specification

Targeted Signals

Now let's handle the Signals. The primary concept here is that Signals can be *targeted*. A simple analogy is with a mouse click event. In our previous example, we only listened for whether the mouse was clicked. With *targeted* Signals, we can listen for what object the mouse had clicked, and derive any further values either from that target, or based on some sort of comparison function between the target and the source.

In our case, we need Signals that let us know three things:

1. What surface did the visualisation touch
2. What is the point of intersection when the visualisation touched the surface
3. What is the normal direction of the surface

The following is the complete version of the specification that we need. Let's go through it.

```
"signals": [{
        "name": "isVisTouchingSurface",
        "source": "vis",
        "target": "surface",
        "criteria": "touch",
        "value": "boolean"
    }, {
        "name": "touchedSurfaceIntersectionPoint",
        "source": "vis",
        "target": "surface",
        "criteria": "touch",
        "value": "intersection"
    }, {
        "name": "touchedSurfaceRotation",
        "source": "vis",
        "target": "surface",
        "criteria": "touch",
        "value": "rotation"
    }
]
```

Complete Signal Specification

In addition to specifying a *name*, we set the *source* to *vis.* This tells Deimos that the Signal(s) should emit values that stem from the visualisation itself. Next, we set *target* to *surface*. This indicates that we want this Signal to then select some *surface* object to be used for our Signal. For this however, we need to set a *criteria* property as well, which indicates the manner in which this target object is selected. We'll set it to *touch*, meaning the *surface* that is targeted is whichever one that the visualisation is touching. Lastly, we need to derive some sort of *value* from the surface. There are three that we use: *boolean* emits true or false depending on whether there is an object actually targeted (i.e., is the visualisation touching a surface); *intersection* gets the point of intersection between the source and target (i.e., where did the visualisation contact the surface); and *rotation* gets the rotation value of the target (i.e., the surface's normal vector).

Remember how we had the *position* and *rotation* encodings in our faceted barchart State? We've now just created two Signals which generate a Vector3 and Quaternion value. In the same way that Signals can be used to pipe string values into State specifications, so can they pipe these Unity types!

What this example does not cover is the use of *expression* Signals. By using the *expression* property instead of *source*, *target*, etc., we can use a mathematical expression to calculate a new Signal value from existing ones. For example, if we wanted to rotate the direction of the surface normal, we can create an *expression* Signal with the value “touchedSurfaceRotation * euler(0, 90, 0)”.

Transition Controls

Now let’s define our Transition. Since we only have two States, we only need a single Transition to be defined. The following is the complete specification.

```
"transitions": [
    {
        "name": "partitioning",
        "states": ["3DBarchart", "facetedBarchart"],
        "trigger": "isVisTouchingSurface",
        "control": {
            "timing": 1.5,
            "easing": "easeoutcirc",
            "interrupted": "ignore",
            "staging": {
                "encoding": [0, 0.65],
                "position": [0.65, 1],
                "rotation": [0.65, 1]
            }
        },
        "bidirectional": true
    }
]
```

Complete Transition Specification

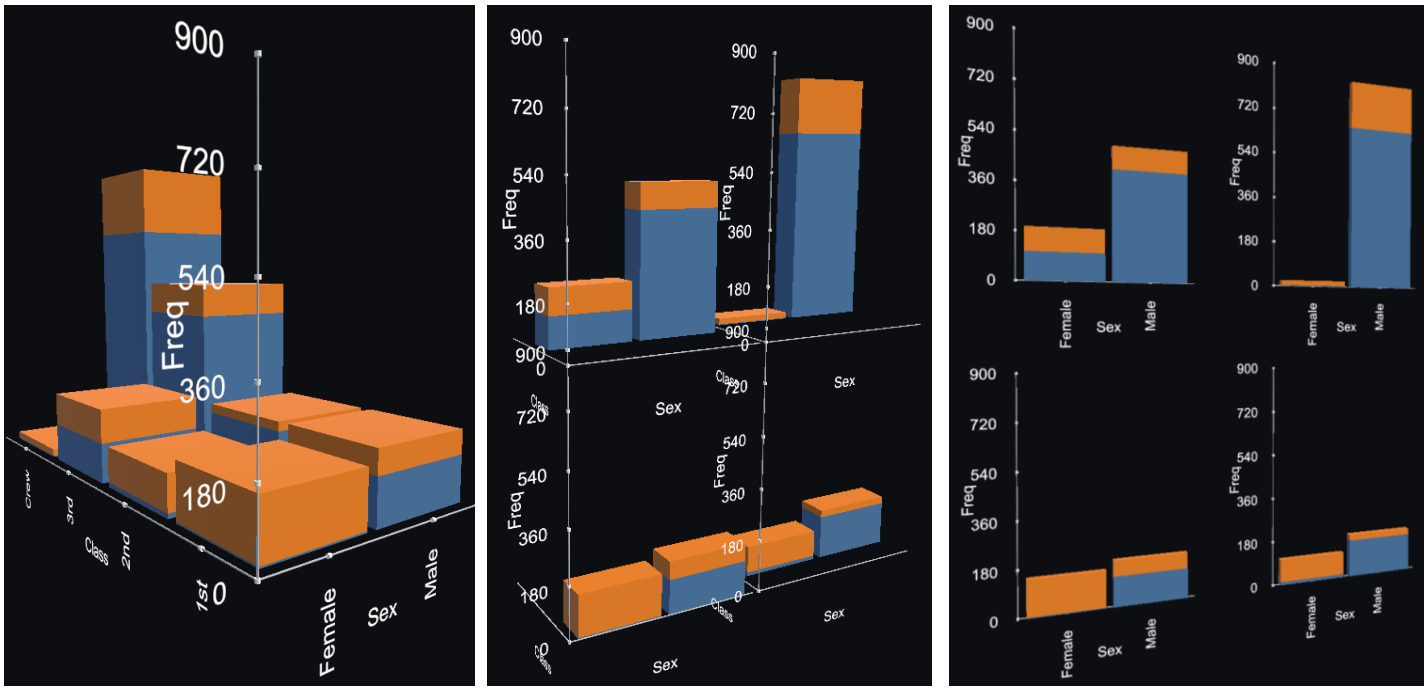
This is mostly the same as our previous example, except for two things. We set an *easing* function in the *control* property in order to smooth the animation a little bit. Autocomplete should show you all of these functions available to you, so feel free to give them all a try! The *staging* property is used to stage different visualisation properties at different times throughout the Transition. For each visualisation property that you want to stage, set it as the property in the *staging* JSON object. The value is an array of two numbers between 0 and 1: the start and end point of the Transition which that property should be animated between. In this example, *position* and *rotation* will only be changed when the Transition is between 65% and 100% completed. Individual encodings such as *x*, *y*, and *color* can be set as well, however note that all *offset* encodings are bundled together as part of their parent dimension (i.e., *x*, *xoffset*, and *xoffset* are all affected by *x* staging). If you want to stage all encodings at once, use *encoding* as the property name. Note that Deimos does not have any support for staggering.

Putting it all together

We're nearly done! We now need to do some things inside of Unity to put it all together.

First, save and set your new Morph Specification in the MorphManager component like last time. Then, you'll need to create a new GameObject which will act as our surface. The best way to do this is by creating a quad. On this quad, you will need to change its tag to *Surface* such that Deimos actually recognises it as a surface. You will also need to make sure that its Mesh Collider has *Convex* set to true, as this makes the collider a bit thicker and therefore will allow the faceted barchart to remain in contact with the surface easier (feel free to try it without this enabled to see what happens). Then you'll need to create the visualisation itself if you haven't done so already, using the 3D barchart specification provided from earlier.

Once done, run the scene and try moving the visualisation into the surface! If all goes well, you should see your barchart be partitioned into facets!



The barchart partitioning Morph in action

And that's about all for this walkthrough! There are of course some concepts and functions which we did not cover in these walkthroughs, such as those found in the Grammar Reference and examples. We leave these for you to try out, either on your own or as part of the examples provided in the project (see next section).

SELF-EXPLORATION

From here on out you are free to explore Deimos' existing Morphs or create your own ones from scratch—perhaps even import your own data! Examples can be found in the *Assets/Deimos/Examples* folder. The scene contained within it has all of the examples already loaded and ready to test via the MRTK simulator (or on your VR headset)

There are already datasets located in the *Assets/StreamingAssets/DxRData* folder for you to use. The majority of these are those that came with DxR originally. If you want to use your own dataset, you will need to place it in this folder. Only .txt (using comma separated values), .csv, .json, and GeoJSON files are supported.

Here are several ideas for you to create and implement (ordered from easiest to hardest):

- A standardised Morph which allows for surface snapping of *any* visualisation (similar to the partitioning barchart example in the *Advanced Morphs* walkthrough)
- A standardised Morph that changes the colour of *any* visualisation when the user is touching its *marks* with their hand/controller (similar to the mark highlighting example in the *Basic Morphs* walkthrough)
- A Morph with hard-coded *fields* that automatically transitions between three or more States as the user presses pre-defined forward/backwards buttons, mimicking a data-driven story slideshow
- A Morph with hard-coded *fields* that automatically transitions between three or more States as the user moves towards it, creating an immersive data-driven story
- A Morph that changes a visualisation between different States as the user points to different predefined objects with their hand/controller

You can also try modifying the existing examples by changing the types of Signals they use. For example, turning timer-based Morphs to ones that use a tweener, or changing their input modality altogether (e.g. from touching a surface to a UI button press).

And a final reminder to please save any creations that you have made, even if they didn't end up working! Thank you once again!

FREQUENTLY ASKED QUESTIONS (FAQs)

General

There are a lot of warning messages "DynamicExpresso.Exceptions.UnknownIdentifierException: Unknown identifier.." in the console. Should I be worried?
You can ignore these warnings if they were ones which were thrown on start up, since it's a side-effect of how initialisation works. If these errors are thrown during runtime, like when you're interacting with a visualisation, then that's a problem.

Data Visualisations (DxR)

How do I do data transformations like in Vega-Lite?
You can't. Deimos is built on top of DxR which also doesn't support data transformations. Any data you use will need to be preprocessed.

My framerate is really low, but I have a powerful computer!?
As with base DxR, large data sets (>1000 rows) will cause framerate drops due to the large number of GameObjects (marks) that are instantiated. This slowdown will increase if both your Game and Scene panels are looking at the marks, or if the Inspector panel has all of the mark GameObjects open. Point one of the cameras away and collapse any DxR visualisation hierarchies if this is the case, especially if you are using tethered VR.

Virtual Reality (MRTK)

How do I set up Deimos with VR/MR?
It should already be set up with MRTK 2 and good to go for Windows Mixed Reality or Oculus Quest (2). We have only tested Deimos with a tethered Oculus Quest 2 via Oculus Link, and to a lesser extent a WMR headset. Standalone VR/MR headsets will likely require some tweaking of the project settings to work properly.

Can I use hand tracking instead of controllers?
Yes you can! Assuming your device is supported, it should be able to hook right into MRTK 2 and Deimos. Note that while Deimos distinguishes between controller and hand Signals, under the hood they are functionally identical for now. Further distinction could be had when things like hand gestures are added.

Can I use gaze and/or voice input?
These forms of input aren't supported in Deimos, although they should be relatively straightforward to implement.

I created a DxR visualisation. How do I make it grabbable?
Please see here. The easiest way to make any GameObject interactive is to add a collider and the *ObjectManipulator* script to it. If you want to make it grabbable when the user touches it, add the *NearInteractionGrabbable* script as well. If you want to disable grabbing while at a distance, uncheck *Allow Far Manipulation* in the *ObjectManipulator* component.

How do I teleport in MRTK?

Please see here.

How do I create MRTK UI elements?
The easiest way is to open the MRTK Toolbox window (*Mixed Reality* > *Toolkit* > *Toolbox*). Note that this window tends to bug and not show anything inside of it when Unity is restarted. In this case just close and reopen the window.

Morphs

Can I create Morphs that affect multiple coordinated linked views?
Unfortunately not. Deimos (and the grammar to some extent) is not built to handle multiple views that exist as separate independent visualisations, with the exception of faceted visualisations.

Can I create Morphs which transition between different mark types (e.g. sphere to cube)?
Due to the way that DxR is set up, as well as the nature of interpolating between an arbitrary number of vertices in 3D graphics, this is unfortunately not possible at this time. The closest you can get is interpolating between polygons and cubes/rectangles using the *geoshape* mark (see the geovisualisation scatterplot example).

Can I create Morphs that animate based on a temporal variable (like in Gapminder)?
No. This process is sort of akin to filtering which DxR doesn't support, hence it wasn't a priority for us to include it in Deimos (even if through some other custom directive).

Can I create Morphs that continuously update based on a changing variable (e.g. have a visualisation constantly rotate towards the user's head)?
This is unfortunately not possible. While the ability to pipe Signal values into State keyframes is implemented in Deimos, the ability to pipe Signals into base DxR visualisation specs is not implemented.

Can I have multiple Morphs and Transitions to the same visualisation simultaneously?
Yes you can! So long as the *Allow Simultaneous Transitions* flag in the *Morphable* component is ticked, a visualisation can have multiple Morphs and Transitions applied to it simultaneously. Note that this only works if each concurrent Transition affects different visualisation properties and encodings (i.e., simultaneous Transitions must be mutually exclusive). For example, you can have a Transition that is changing the *z* encoding and another Transition that changes the *colour* encoding being applied at the same time.

States

My State with a *size* encoding doesn't seem to be able to have Morphs applied to it. I also receive NaN errors when trying to apply a *size* transition.
The State matching process only checks against the visualisation specification that you yourself had entered. Even though DxR automatically adds (infers) a *size* encoding even though you did not specify one in your visualisation specification, Deimos intentionally doesn't detect this *size* encoding for now. For Morphs involving *size* encoding changes you should ensure that your visualisation(s) have a *size* encoding explicitly defined.

Signals

What combinations of Signals are available? It's hard to figure out what can be used with each other.
Please see Signal Combinations for a diagrammatic view.

How do I know what functions are supported in expression Signals? What do I do if the function that I need isn't supported?
Please see Expression Functions. You can also add new functions if need be at the bottom of the *MorphManager.cs* script, following a similar syntax as the other functions defined within.

My Morphs with *handedness* of type *any* seem to behave erratically when I try to use both hands simultaneously. What's going on?
This is a side effect of the way that we handle this type of Signal. As both hands share the same stream of data, if one of them emits a "complete" value the entire Signal will end, even if the other hand has not actually completed yet. The most robust way around this is just to use two separate Signals, one for each hand, particularly when implementing bimanual operations.

Transitions

I want the reverse direction in bidirectional Transitions to use different *triggers*/*states*/*staging*/etc. How do I do it?
You will need to create another Transition with the order of the States reversed. Instead of creating a single *bidirectional* Transition, you need to create two *unidirectional* Transitions that mimic a bidirectional one, but each with their own distinct parameters.

Where can I visually see all of the easing functions available to me?
This website has some very nice visualisations of the common easing functions.

I'm trying to apply staging to a curved facetwrap that has its small multiples facing the centrepoint visualisation but it's behaving erratically?
Using the *"orientation": "centre"* property in curved facets will break any staged transition that affects the *x*, *y*, or *z* encodings. We recommend generally avoiding staging when using this type of curved facetwrap.

GRAMMAR REFERENCE

Deimos

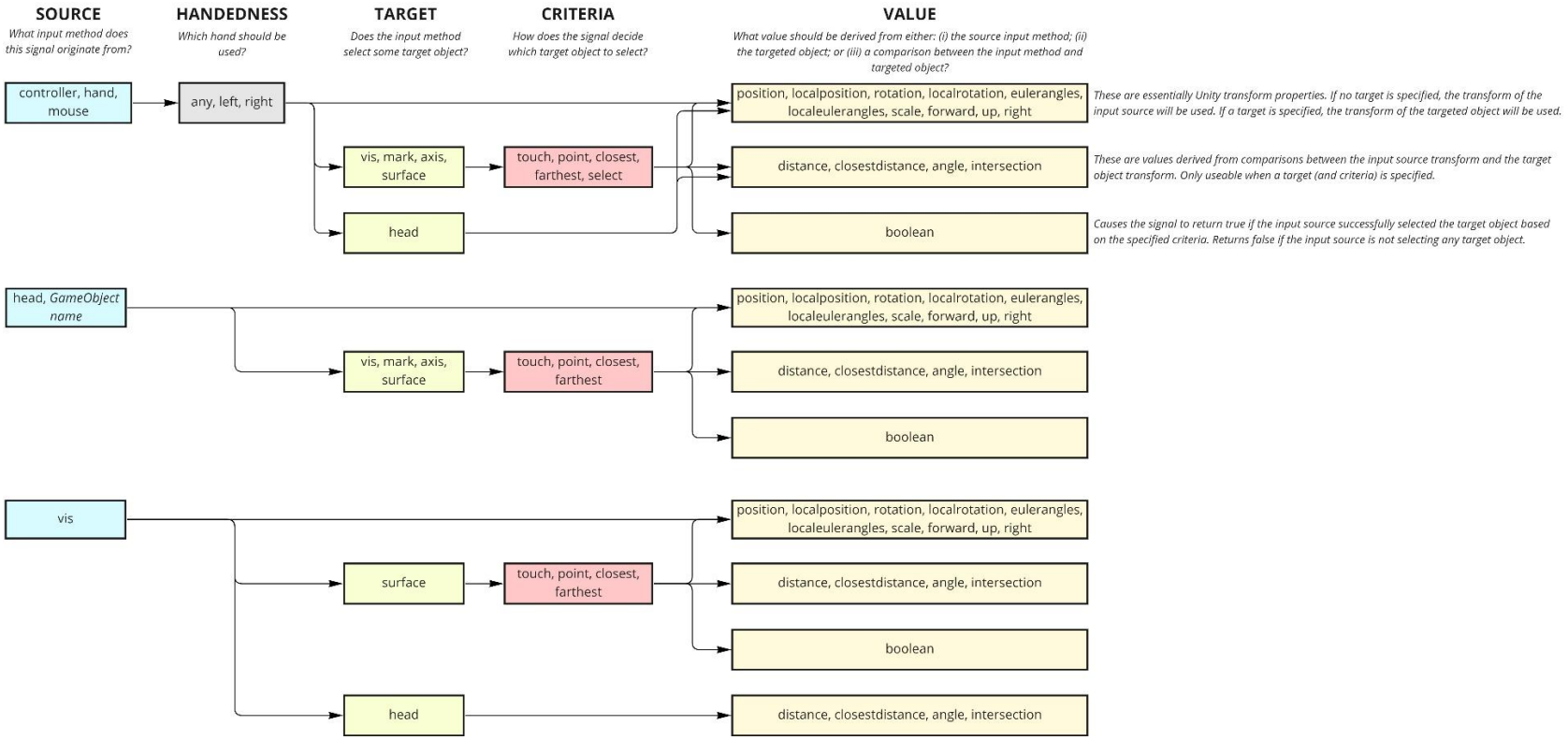
Property (Level 1)	Property (Level 2)	Property (Level 3)	Property (Level 4)	Type	Required	Supported Values	Description
states				object array	Yes	-	Describes the states that are part of this morph.
	name			string	Yes	-	The name of this state. Must be unique across all states. Case sensitive.
	restrict			boolean	No	-	If true, this state cannot be reached by a visualisation except through a transition. Defaults to false.
	DxR properties			-	-	-	One or more DxR properties and accompanying values. Any property specified here will be used to determine if a visualisation matches this state. Any value (with some exceptions) can be replaced with any one of the following operators. They can be placed both at the root DxR property level (e.g. “x”: …) and at any leaf property (e.g. “type”: …, “domain”: [..., …]): ● "*" ○ The DxR property should be included in the visualisation, but its value does not matter ● *null* ○ The DxR property should **not** be included in the visualisation ● *an inequality expression* ○ e.g. "value": ">= 100" ○ The DxR property should be included in the visualisation, and its value should satisfy the inequality. Only applicable to numeric DxR properties. ● *JSON path accessor* ○ e.g. “field”: “this.encoding.x.field”, “color”: “other.encoding.color” ○ The DxR property should be included in the visualisation, but its value does not matter. When keyframes are generated at the start of a transition, the value residing at that JSON path will be substituted into this DxR property. Must be prefixed with either *this.* or *other. this.* accesses the JSON path of this state's keyframe, *other.* accesses the path of the other state's keyframe. ● *any signal name* ○ The DxR property should be included in the visualisation, but its value does not matter. When keyframes are generated at the start of a transition, the referenced signal's value will be substituted into this DxR property. ● *an expression* ○ e.g. "value": "other.encoding.size.value * 10", “value”: “this.encoding.width.value + 20” ○ The DxR property should be included in the visualisation, but its value does not matter. When keyframes are generated at the start of a transition, the expression will be evaluated and its result substituted into this DxR property. Expression can contain JSON path accessors and/or signal names which behave as variables. Only applicable to numeric DxR properties.

signals				object array	No	-	Describes the signals that are part of this morph.
	name			string	Yes	-	The name of this signal. Must be unique across all signals. Case sensitive. No spaces or mathematical characters (e.g. +, -) are allowed.
	source			string	Yes*	"mouse", "controller", "hand", "ui", "vis", "head"	The source of this signal. One of this source property or the expression property is required.
	handedness			string	No	"any", "left", "right"	The handedness of the source signal to use. Defaults to "any". Only applicable to hand and controller sources.
	target			string	No	"mark", "axis", "surface", "vis", "head", *any GameObject name, any UI element name*	The target of this signal. If specified, the signal will target the type of object described here. Alternatively, the name of any Unity GameObject can be used instead. For UI sources, this should be the name of the GameObject holding the UI script.
	criteria			string	Yes*	"touch", "point", "select", "click", "closest", "farthest"	The criteria that this signal uses to select the target. Required when the target property is set to "mark", "axis", or "surface".
	value			string	Yes*	"position", "localposition", "rotation", "localrotation", "eulerangles", "localeulerangles", "scale", "forward", "up", "right", "boolean", "distance", "closestdistance", "angle", "intersection", "select", *dot notation on a GameObject*	The value to extract from the source or from the target that the source had selected. Required when a source property is used. Not all source and target configurations support all values. For example, any value which does a comparison will only work with a target. See Signal Combinations for further details. Dot notation can also be used to access GameObject properties via reflection (e.g. "transform.localPosition.x"). This is case sensitive, using the exact Unity syntax.
	expression			string	Yes*	-	A mathematical expression. One of this expression property or the source property is required. Standard mathematical operators and some mathematical functions are supported. Signal names can be used as variables (case sensitive).
transitions				object array	Yes	-	Describes the transitions that are part of this morph.
	name			string	Yes	-	The name of this transition. Must be unique across all signals. Case sensitive.
	states			string array	Yes	-	The names of the initial and final states in this transition, in that order. Must only contain two values. Values must be the names of two states (case sensitive).
	trigger			string	No	-	The name of a signal or expression. Signal or expression used must return a boolean value. The transition is triggered when this returns true (assuming a state has been reached). If not defined, the transition will begin as soon as a state has been reached.
	control			object	No	-	Describes rules for how the transition is controlled and behaves.

		timing		string or number	No	-	Controls the timing of the transition. If a string, it must be the name of a signal which returns a number between 0 and 1. If a number, must be a positive number in seconds. If not defined, the transition will start and end instantaneously.
		easing		string	No	“linear”, “spring”, *for rest see* *https://easings.net/*	Applies an easing function to the transition. Defaults to linear.
		interrupted		string	No	"initial", "final", "ignore"	What state should the visualisation revert to when the transition is interrupted (i.e., when the trigger returns false mid-transition). If set to ignore, the transition will keep progressing until terminated by the timer. Defaults to final.
		completed		string	No	"initial", "final"	What state should the visualisation rest at when the transition is completed. Defaults to final.
		staging		object array	No	-	Describes the DxR properties that should be staged and when.
			DxR properties	number array	No	"x", "y", "z", "width", "height", "depth", "color", "size", "position", "rotation", "encoding"	DxR properties to apply staging to. Name should be the property that is to be staged (see supported values). Value is a number array of only two values between 0 and 1, indicating the percentage of the transition to start and stop the stage at for that property. If the "encoding" property is used, all properties not explicitly defined will fall back to its staging values.
	bidirectional			boolean	No	-	If true, the transition is bidirectional and can operate in reverse. Defaults to false.
	disablegrab			boolean	No	-	If true, any MRTK grab operation that is currently in progress on the visualisation when the transition starts will be disabled. Defaults to false.
	priority			number	No	-	The priority of this transition when multiple transitions are triggered at the same time. Transitions with higher priority will activate before those of lower priority. Defaults to 0.

Signal Combinations

The following diagram shows the different combinations of Signals available across the different properties. Only combinations that make sense are shown here.



- *GameObject name*
  - This refers to the literal name of the GameObject, accessed using GameObject.Find(). This obviously means that the name should be unique
  - For MRTK UI GameObjects, this should be the root GameObject of the MRTK UI element
- Targets
  - For *vis, mark,* and *axis* targets, these only target objects that belong to the visualisation that is being morphed
    - For example, if a barchart is undergoing a morph, a *mark* target will only select the bar marks that belong to that specific barchart, and not marks from any other visualisation
  - The *vis* target refers to **any** part of the visualisation
- Criteria
  - For *hand* and *controller* sources, *touch* and *select* criteria will perform an OverlapSphere operation of a radius of 0.125f in order to find colliding targets
  - For *point* criteria, a Raycast will be performed using the *source*'s transform.forward

Expression Functions

The following functions are supported in Deimos expressions. This includes expressions used in Signals and States. To add more functions, please see line 1066 of *MorphManager.cs*.

- Data types
  - vector3(x, y, z)
  - quaternion(x, y, z, w)
  - euler(x, y, z)
- Mathematical functions
  - clamp(input, min, max)
  - normalise(input, minInputRange, maxInputRange)
  - normalise(input, minInputRange, maxInputtRange, minOutputRange, maxOutputRange)
- Unity functions
  - distance(position1, position2)
    - Same as https://docs.unity3d.com/ScriptReference/Vector3.Distance.html
  - angle(fromDirection, toDirection)
    - Same as https://docs.unity3d.com/ScriptReference/Vector3.Angle.html
  - signedangle(fromDirection, toDirection, axis)
    - Same as https://docs.unity3d.com/ScriptReference/Vector3.SignedAngle.html
  - closestpoint(collider, position)
    - Similar to https://docs.unity3d.com/ScriptReference/Collider.ClosestPoint.html
  - lookrotation(forward)
    - Same as https://docs.unity3d.com/ScriptReference/Quaternion.LookRotation.html
  - lookrotation(forward, up)
    - Same as https://docs.unity3d.com/ScriptReference/Quaternion.LookRotation.html

Expanded DxR Grammar

The extended functions of DxR that Deimos introduces are shown below. Properties that overlap with existing DxR properties will have expanded options displayed in **bold**. Please see the official DxR Grammar Docs for more information.

Property (Level 1)	Property (Level 2)	Property (Level 3)	Type	Supported Values	Description
mark			string	**“geoshape”**, “point”, “bar”, “arrow”, “text”, ...	Specifies which graphical mark to use. For geoshape, the data should be in GeoJSON format. geoshapes also provide access to the Longitude/Latitude fields and spatial types (see below).
encoding			object	-	Describes how data fields are mapped to visual channels of the graphical mark.
	x, y, z, width, height		object		
		field	string	**“Longitude”, “Latitude”**, …	Name of the data attribute or data field that should be mapped to this channel. If set to Longitude or Latitude, centroid positions will be used for spatial channels, and geometric polygons will be used for size channels. If used, the type should also be set to spatial.
		type	string	**“spatial”,** “quantitative”, “nominal”, “ordinal”	The data type of this channel. If set to spatial, the data field will be scaled based on the overall longitude/latitude of the GeoJSON dataset. Generally speaking, use the spatial data type whenever the field is set to either Longitude or Latitude.
	xoffset, yoffset, zoffset		object	-	
		field	string	-	The data attribute or data field to apply an offset based on. Marks will be offset along this spatial dimension using this field as the category and the associated size encoding (width, height, depth).
		type	string	"nominal", "ordinal"	The data type of the offsetting data attribute or field. Must be either nominal or ordinal.
	facetwrap	field	string	-	The data attribute or data field to facet by.

		type	string	"quantitative", "nominal", "ordinal"	The data type of the faceting data attribute or field.
		directions	string array	"x", "y", "z"	A string array of spatial directions that should be faceted by. Facets will proceed in the first direction and wrap around in the second direction. Array must contain only two elements and have unique values. Defaults to ["x", "y"].
		spacing	number array	-	A number array of spacing values to have between each facet. Spacing amounts will be applied corresponding to the directions property. Array must contain only two elements. Defaults to the size of the visualisation along its respective directions.
		padding	number array	-	A number array of padding values to have between each facet. Padding amounts will be applied corresponding to the directions property, in addition to the spacing amounts. Array must contain only two elements. Defaults to [200, 200].
		radius	number	-	The radius of the sphere which facets will be placed along in curved layouts. Defined in millimetres. Defaults to 1000.
		angles	number array	-	A number array of angles (in degrees) of the two curves wherein facets will be positioned at angular intervals. For example, an angle of 15 will position facets 15 degrees apart from one another. If set, directions can only be "x" and "y". Angles of 0 will not have any curvature applied. For cylindrical layouts, use one angle value. For spherical layouts, use two angle values. For flat layouts, use no angle values (or set these to 0). Defaults to [0, 0].
		orientation	string	"forward", "centre"	If set to centre, facets will turn towards the centre of the sphere. If set to forward, facets will remain facing forwards. Has no effect on flat layouts. Defaults to centre.
position			object	-	Describes what the position of the visualisation should be set to in world space. The visualisation's position will only be set whenever the visualisation is created or updated.
	value		number array	-	An array of 3 numeric values that are parsed as a complete Vector3.
	x, y, z		number	-	The spatial dimension to override the visualisation's position of and the value to set it to. Any spatial dimensions not explicitly defined will be unchanged.
rotation			object	-	Describes what the rotation of the visualisation should be set to in world space. The visualisation's rotation will only be set whenever the visualisation is created or updated.
	value		number array	-	An array of 3 or 4 numeric values. If 3 values are set, they will be parsed as an euler angle. If 4 values are set, they will be parsed as a quaternion.
	x, y, z		number	-	The axis to override the visualisation's rotation of and the value to set it to. Any axes not explicitly defined will be unchanged. This only modifies euler angles and not quaternions.