AnalyticalGraphicsInc / **3d-tiles**

Branch: master ▾ 3d-tiles / Styling /

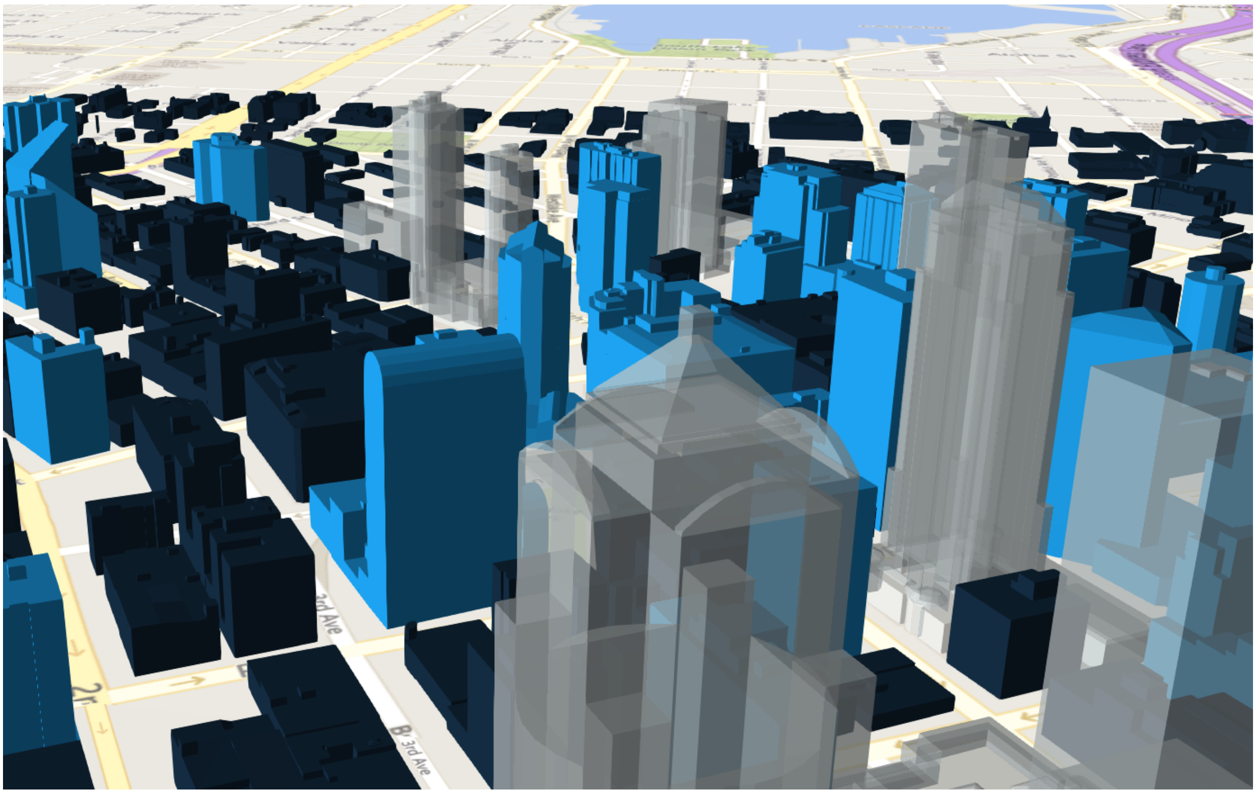
Create new file | Find file | History

**lilleyse** Small typo fix — Latest commit 913c7cb on 25 Jun

| | | |
|---|---|---|
| .. | | |
| figures | Add example figure | 2 years ago |
| schema | Add defines, remove expression | 6 months ago |
| README.md | Small typo fix | 6 months ago |

README.md

# Styling




```
{
    "show" : "${Area} > 0",
    "color" : {
        "conditions" : [
            ["${Height} < 60", "color('#13293D')"],
            ["${Height} < 120", "color('#1B98E0')"],
            ["true", "color('#E8F1F2', 0.5)"]
        ]
    }
}
```

Example: Creating a color ramp based on building height.

## Contributors

- Gabby Getz, @ggetz
- Matt Amato, @matt_amato
- Tom Fili, @CesiumFili

- Sean Lilley, @lilleyse
- Patrick Cozzi, @pjcozzi

Contents:



## Overview

3D Tiles styles provide concise declarative styling of tileset features. A style defines expressions to evaluate a feature's `color` (RGB and translucency) and `show` properties, often based on the feature's properties stored in the tile's batch table.

Styles are defined with JSON and expressions written in a small subset of JavaScript augmented for styling. Additionally the styling language provides a set of built-in functions to support common math operations.

## Examples

The following style assigns the default show and color properties to each feature:

```
{
    "show" : "true",
    "color" : "color('#ffffff')"
}
```

Instead of showing all features, `show` can be an expression dependent on a feature's properties, for example:

```
{
    "show" : "${ZipCode} === '19341'"
}
```

Here, only features in the 19341 zip code are shown.

```
{
    "show" : "(regExp('^Chest').test(${County})) && (${YearBuilt} >= 1970)"
}
```

Above, a compound condition and regular expression are used to show only features whose county starts with `'Chest'` and whose year built is greater than or equal to 1970.

Colors can also be defined by expressions dependent on a feature's properties, for example:

```
{
    "color" : "(${Temperature} > 90) ? color('red') : color('white')"
}
```

This colors features with a temperature above 90 as red and the others as white.

The color's alpha component defines the feature's opacity, for example:

```
{
   "color" : "rgba(${red}, ${green}, ${blue}, (${volume} > 100 ? 0.5 : 1.0))"
}
```

This sets the feature's RGB color components from the feature's properties and makes features with volume greater than 100 transparent.

In addition to a string containing an expression, `color` and `show` can be an array defining a series of conditions (think of them as `if...else` statements). Conditions can, for example, be used to make color maps and color ramps with any type of inclusive/exclusive intervals.

For example, here's a color map that maps an ID property to colors:

```
{
    "color" : {
        "conditions" : [
            ["${id} === '1'", "color('#FF0000')"],
            ["${id} === '2'", "color('#00FF00')"],
            ["true", "color('#FFFFFF')"]
        ]
    }
}
```

Conditions are evaluated in order so, above, if `${id}` is not `'1'` or `'2'`, the `"true"` condition returns white. If no conditions are met, the color of the feature will be `undefined`.

The next example shows how to use conditions to create a color ramp using intervals with an inclusive lower bound and exclusive upper bound.

```
"color" : {
    "conditions" : [
        ["(${Height} >= 1.0)  && (${Height} < 10.0)", "color('#FF00FF')"],
        ["(${Height} >= 10.0) && (${Height} < 30.0)", "color('#FF0000')"],
        ["(${Height} >= 30.0) && (${Height} < 50.0)", "color('#FFFF00')"],
        ["(${Height} >= 50.0) && (${Height} < 70.0)", "color('#00FF00')"],
        ["(${Height} >= 70.0) && (${Height} < 100.0)", "color('#00FFFF')"],
        ["(${Height} >= 100.0)", "color('#0000FF')"]
    ]
}
```

Since conditions are evaluated in order, the above can more concisely be written as:

```
"color" : {
    "conditions" : [
        ["(${Height} >= 100.0)", "color('#0000FF')"],
        ["(${Height} >= 70.0)", "color('#00FFFF')"],
        ["(${Height} >= 50.0)", "color('#00FF00')"],
        ["(${Height} >= 30.0)", "color('#FFFF00')"],
        ["(${Height} >= 10.0)", "color('#FF0000')"],
        ["(${Height} >= 1.0)", "color('#FF00FF')"]
```

```
    ]
}
```

Commonly used expressions may be stored in a `defines` object. If a variable references a define, it gets the result of the define's evaluated expression.

```
{
    "defines" : {
        "NewHeight" : "clamp((${Height} - 0.5) / 2.0, 1.0, 255.0)",
        "HeightColor" : "rgb(${Height}, ${Height}, ${Height})"
    },
    "color" : {
        "conditions" : [
            ["(${NewHeight} >= 100.0)", "color('#0000FF') * ${HeightColor}"],
            ["(${NewHeight} >= 50.0)", "color('#00FF00') * ${HeightColor}"],
            ["(${NewHeight} >= 1.0)", "color('#FF0000') * ${HeightColor}"]
        ]
    },
    "show" : "${NewHeight} < 200.0"
}
```

A define expression may not reference other defines, however it may reference feature properties with the same name. In the style below a feature of height 150 gets the color red.

```
{
    "defines" : {
        "Height" : "${Height}/2.0}",
    },
    "color" : {
        "conditions" : [
            ["(${Height} >= 100.0)", "color('#0000FF')"],
            ["(${Height} >= 1.0)", "color('#FF0000')"]
        ]
    }
}
```

Non-visual properties of a feature can be defined using the `meta` property.

For example, to set a `description` meta property to a string containing the feature name:

```
{
    "meta" : {
        "description" : "'Hello, ${featureName}.'"
    }
}
```

A meta property expression can evaluate to any type. For example:

```
{
    "meta" : {
        "featureColor" : "rgb(${red}, ${green}, ${blue})",
        "featureVolume" : "${height} * ${width} * ${depth}"
    }
}
```

## Schema Reference

TODO: generate reference doc from schema

Also, see the JSON schema.

## Expressions

The language for expressions is a small subset of JavaScript (EMCAScript 5), plus native vector and regular expression types and access to tileset feature properties in the form of readonly variables.

*Implementation tip: Cesium uses the jsep JavaScript expression parser library to parse style expressions.*

### Semantics

Dot notation is used to access properties by name, e.g., `building.name` .

Bracket notation ( `[]` ) is also used to access properties, e.g., `building['name']` , or arrays, e.g., `temperatures[1]` .

Functions are called with parenthesis ( `()` ) and comma-separated arguments, e.g., ( `isNaN(0.0)` , `color('cyan', 0.5)` ).

### Operators

The following operators are supported with the same semantics and precedence as JavaScript.

- Unary: `+` , `-` , `!`
    - Not supported: `~`
- Binary: `||` , `&&` , `===` , `!==` , `<` , `>` , `<=` , `>=` , `+` , `-` , `*` , `/` , `%` , `=~` , `!~`
    - Not supported: `|` , `^` , `&` , `<<` , `>>` , and `>>>`
- Ternary: `? :`

`(` and `)` are also supported for grouping expressions for clarity and precedence.

Logical `||` and `&&` implement short-circuiting; `true || expression` does not evaluate the right expression, and `false && expression` does not evaluate the right expression.

Similarly, `true ? leftExpression : rightExpression` only executes the left expression, and `false ? leftExpression : rightExpression` only executes the right expression.

### Types

The following types are supported:

- `Boolean`
- `Null`
- `Undefined`
- `Number`
- `String`
- `Array`
- `vec2`
- `vec3`
- `vec4`
- `RegExp`

All of the types except `vec2` , `vec3` , `vec4` , and `RegExp` have the same syntax and runtime behavior as JavaScript. `vec2` , `vec3` , and `vec4` are derived from GLSL vectors and behave similarly to JavaScript `Object` (see the Vector section). Colors derive from CSS3 Colors and are implemented as `vec4` . `RegExp` is derived from JavaScript and described in the RegExp section.

Example expressions for different types include the following:

- `true` , `false`
- `null`
- `undefined`
- `1.0` , `NaN` , `Infinity`
- `'Cesium'` , `"Cesium"`
- `[0, 1, 2]`
- `vec2(1.0, 2.0)`
- `vec3(1.0, 2.0, 3.0)`
- `vec4(1.0, 2.0, 3.0, 4.0)`
- `color('#00FFFF')`
- `regExp('^Chest'))`

#### Number

As in JavaScript, numbers can be `NaN` or `Infinity` . The following test functions are supported:

- `isNaN(testValue : Number) : Boolean`
- `isFinite(testValue : Number) : Boolean`

### Vector

The styling language includes 2, 3, and 4 component floating-point vector types: `vec2`, `vec3`, and `vec4`. Vector constructors share the same rules as GLSL:

#### vec2

- `vec2(xy : Number)` - initialize each component with the number
- `vec2(x : Number, y : Number)` - initialize with two numbers
- `vec2(xy : vec2)` - initialize with another `vec2`
- `vec2(xyz : vec3)` - drops the third component of a `vec3`
- `vec2(xyzw : vec4)` - drops the third and fourth component of a `vec4`

#### vec3

- `vec3(xyz : Number)` - initialize each component with the number
- `vec3(x : Number, y : Number, z : Number)` - initialize with three numbers
- `vec3(xyz : vec3)` - initialize with another `vec3`
- `vec3(xyzw : vec4)` - drops the fourth component of a `vec4`
- `vec3(xy : vec2, z : Number)` - initialize with a `vec2` and number
- `vec3(x : Number, yz : vec2)` - initialize with a `vec2` and number

#### vec4

- `vec4(xyzw : Number)` - initialize each component with the number
- `vec4(x : Number, y : Number, z : Number, w : Number)` - initialize with four numbers
- `vec4(xyzw : vec4)` - initialize with another `vec4`
- `vec4(xy : vec2, z : Number, w : Number)` - initialize with a `vec2` and two numbers
- `vec4(x : Number, yz : vec2, w : Number)` - initialize with a `vec2` and two numbers
- `vec4(x : Number, y : Number, zw : vec2)` - initialize with a `vec2` and two numbers
- `vec4(xyz : vec3, w : Number)` - initialize with a `vec3` and number
- `vec4(x : Number, yzw : vec3)` - initialize with a `vec3` and number

#### Vector usage

`vec2` components may be accessed with

- `.x`, `.y`
- `.r`, `.g`
- `[0]`, `[1]`

`vec3` components may be accessed with

- `.x`, `.y`, `.z`
- `.r`, `.g`, `.b`
- `[0]`, `[1]`, `[2]`

`vec4` components may be accessed with

- `.x`, `.y`, `.z`, `.w`
- `.r`, `.g`, `.b`, `.a`
- `[0]`, `[1]`, `[2]`, `[3]`

Unlike GLSL, the styling language does not support swizzling. For example `vec3(1.0).xy` is not supported.

Vectors support the following unary operators: `-`, `+`.

Vectors support the following binary operators by performing component-wise operations: `===`, `!==`, `+`, `-`, `*`, `/`, and `%`. For example `vec4(1.0) === vec4(1.0)` is true since the x, y, z, and w components are equal. Operators are essentially overloaded for `vec2`, `vec3`, and `vec4`.

`vec2`, `vec3`, and `vec4` have a `toString` function for explicit (and implicit) conversion to strings in the format `'(x, y)'`, `'(x, y, z)'`, and `'(x, y, z, w)'`.

- `toString() : String`

`vec2`, `vec3`, and `vec4` do not expose any other functions or a `prototype` object.

#### Color

Colors are implemented as `vec4` and are created with one of the following functions:

- `color() : Color`
- `color(keyword : String, [alpha : Number]) : Color`
- `color(6-digit-hex : String, [alpha : Number]) : Color`
- `color(3-digit-hex : String, [alpha : Number]) : Color`
- `rgb(red : Number, green : Number, blue : number) : Color`
- `rgba(red : Number, green : Number, blue : number, alpha : Number) : Color`
- `hsl(hue : Number, saturation : Number, lightness : Number) : Color`
- `hsla(hue : Number, saturation : Number, lightness : Number, alpha : Number) : Color`

Calling `color()` with no arguments is the same as calling `color('#FFFFFF')`.

Colors defined by a case-insensitive keyword (e.g., `'cyan'`) or hex rgb are passed as strings to the `color` function. For example:

- `color('cyan')`
- `color('#00FFFF')`
- `color('#0FF')`

These `color` functions have an optional second argument that is an alpha component to define opacity, where `0.0` is fully transparent and `1.0` is fully opaque. For example:

- `color('cyan', 0.5)`

Colors defined with decimal rgb or hsl are created with `rgb` and `hsl` functions, respectively, just as in CSS (but with percentage ranges from `0.0` to `1.0` for `0%` to `100%`, respectively). For example:

- `rgb(100, 255, 190)`
- `hsl(1.0, 0.6, 0.7)`

The range for rgb components is `0` to `255`, inclusive. For `hsl`, the range for hue, saturation, and lightness is `0.0` to `1.0`, inclusive.

Colors defined with `rgba` or `hsla` have a fourth argument that is an alpha component to define opacity, where `0.0` is fully transparent and `1.0` is fully opaque. For example:

- `rgba(100, 255, 190, 0.25)`
- `hsla(1.0, 0.6, 0.7, 0.75)`

Colors are equivalent to the `vec4` type and share the same functions, operators, and component accessors. Color components are stored in the range `0.0` to `1.0`.

For example:

- `color('red').x`, `color('red').r`, and `color('red')[0]` all evaluate to `1.0`.
- `color('red').toString()` evaluates to `(1.0, 0.0, 0.0, 1.0)`
- `color('red') * vec4(0.5)` is equivalent to `vec4(0.5, 0.0, 0.0, 1.0)`

#### RegExp

Regular expressions are created with the following functions, which behave like the JavaScript `RegExp` constructor:

- `regExp() : RegExp`
- `regExp(pattern : String, [flags : String]) : RegExp`

Calling `regExp()` with no arguments is the same as calling `regExp('(?:)')`.

If specified, `flags` can have any combination of the following values:

- `g` - global match
- `i` - ignore case
- `m` - multiline
- `u` - unicode
- `y` - sticky

Regular expressions support these functions:

- `test(string : String) : Boolean` - Tests the specified string for a match.
- `exec(string : String) : String` - Executes a search for a match in the specified string. If the search succeeds, it returns the first instance of a captured `String`. If the search fails, it returns `null`

For example:

```
{
    "Name" : "Building 1"
}
```

```
regExp('a').test('abc') === true
regExp('a(.)', 'i').exec('Abc') === 'b'
regExp('Building\s(\d)').exec(${Name}) === '1'
```

Regular expressions have a `toString` function for explicit (and implicit) conversion to strings in the format `'pattern'`.

- `toString() : String`

Regular expressions do not expose any other functions or a `prototype` object.

The operators `=~` and `!~` are overloaded for regular expressions. The `=~` operator matches the behavior of the `test` function, and tests the specified string for a match. It returns `true` if one is found, and `false` if not found. The `!~` operator is the inverse of the `=~` operator. It returns `true` if no matches are found, and `false` if a match is found. Both operators are commutative.

For example, the following expressions all evaluate to true:

```
regExp('a') =~ 'abc'
'abc' =~ regExp('a')

regExp('a') !~ 'bcd'
'bcd' !~ regExp('a')
```

### Operator Rules

- Unary operators `+` and `-` operate only on number and vector expressions.
- Unary operator `!` operates only on boolean expressions.
- Binary operators `<`, `<=`, `>`, and `>=` operate only on number expressions.
- Binary operators `||` and `&&` operate only on boolean expressions.
- Binary operator `+` operates on the following expressions:
    - Number expressions
    - Vector expressions of the same type
    - If at least one expressions is a string, the other expressions is converted to a string following String Conversions and the operation returns a concatenated string. E.g. `"name" + 10` evaluates to `"name10"`.
- Binary operator `-` operates on the following expressions
    - Number expressions
    - Vector expressions of the same type
- Binary operator `*` operates on the following expressions
    - Number expressions
    - Vector expressions of the same type
    - Mix of number expression and vector expression. E.g. `3 * vec3(1.0)` and `vec2(1.0) * 3`.

- Binary operator `/` operates on the following expressions
  - Number expressions
  - Vector expressions of the same type
  - Vector expression followed by number expression. E.g. `vec3(1.0) / 3`.
- Binary operator `%` operates on the following expressions
  - Number expressions
  - Vector expressions of the same type
- Binary equality operators `===` and `!==` operate on any expressions. The operation returns `false` if the expression types do not match.
- Binary regexp operators `=~` and `!~` requires one argument to be a string expression and the other to be a RegExp expression.
- Ternary operator `? :` conditional argument must be a boolean expression.

### Type Conversions

Explicit conversions between primitive types are handled with `Boolean`, `Number`, and `String` functions.

- `Boolean(value : Any) : Boolean`
- `Number(value : Any) : Number`
- `String(value : Any) : String`

For example:

```
Boolean(1) === true
Number('1') === 1
String(1) === '1'
```

`Boolean` and `Number` follow JavaScript conventions. `String` follows String Conversions.

These are essentially casts, not constructor functions.

The styling language does not allow for implicit type conversions, unless stated above. Expressions like `vec3(1.0) === vec4(1.0)` and `"5" < 6` are not valid.

### String Conversions

`vec2`, `vec3`, `vec4` and `RegExp` expressions are converted to strings using their `toString` methods. All other types follow JavaScript conventions.

- `true - "true"`
- `false - "false"`
- `null - "null"`
- `undefined - "undefined"`
- `5.0 - "5"`
- `NaN - "NaN"`
- `Infinity - "Infinity"`
- `"name" - "name"`
- `[0, 1, 2] - "[0, 1, 2]"`
- `vec2(1, 2) - "(1, 2)"`
- `vec3(1, 2, 3) - "(1, 2, 3)"`
- `vec4(1, 2, 3, 4) - "(1, 2, 3, 4)"`
- `RegExp('a') - "/a/"`

### Constants

The following constants are supported by the styling language:

- `Math.PI`
- `Math.E`

#### PI

The mathematical constant PI, which represents a circle's circumference divided by its diameter, approximately `3.14159`.

```
{
    "show" : "cos(${Angle} + Math.PI) < 0"
}
```

#### E

Euler's constant and the base of the natural logarithm, approximately `2.71828`.

```
{
    "color" : "color() * pow(Math.E / 2.0, ${Temperature})"
}
```

### Variables

Variables are used to retrieve the property values of individual features in a tileset. Variables are identified using the ES 6 (ECMAScript 2015) Template Literal syntax, i.e., `${feature.identifier}` or `${feature['identifier']}`, where the identifier is the case-sensitive property name. `feature` is implicit and can be omitted in most cases.

Variables can be used anywhere a valid expression is accepted, except inside other variable identifiers. For example, the following is not allowed:

```
${foo[${bar}]}
```

If a feature does not have a property with specified name, the variable evaluates to `undefined`. Note that the property may also be `null` if `null` was explicitly stored for a property.

Variables may be any of the supported native JavaScript types:

- `Boolean`
- `Null`
- `Undefined`
- `Number`
- `String`
- `Array`

For example:

```
{
    "enabled" : true,
    "description" : null,
    "order" : 1,
    "name" : "Feature name"
}
```

```
${enabled} === true
${description} === null
${order} === 1
${name} === 'Feature name'
```

Additionally, variables originating from vector properties stored in the Batch Table Binary are treated as vector types:

| `componentType` | variable type |
|---|---|
| `"VEC2"` | `vec2` |
| `"VEC3"` | `vec3` |
| `"VEC4"` | `vec4` |

Variables can be used to construct colors or vectors, for example:

```
rgba(${red}, ${green}, ${blue}, ${alpha})
vec4(${temperature})
```

Dot or bracket notation is used to access feature subproperties. For example:

```
{
    "address" : {
        "street" : "Example street",
        "city" : "Example city"
    }
}
```

```
${address.street} === `Example street`
${address['street']} === `Example street`

${address.city} === `Example city`
${address['city']} === `Example city`
```

Bracket notation supports only string literals.

Top-level properties can be accessed with bracket notation by explicitly using the `feature` keyword. For example:

```
{
    "address.street" : "Maple Street",
    "address" : {
        "street" : "Oak Street"
    }
}
```

```
${address.street} === `Oak Street`
${feature.address.street} === `Oak Street`
${feature['address'].street} === `Oak Street`
${feature['address.street']} === `Maple Street`
```

To access a feature named `feature` , use the variable `${feature}` . This is equivalent to accessing `${feature.feature}`

```
{
    "feature" : "building"
}
```

```
${feature} === `building`
${feature.feature} === `building`
```

Variables can also be substituted inside strings defined with backticks, for example:

```
{
    "order" : 1,
    "name" : "Feature name"
}
```

```
`Name is ${name}, order is ${order}`
```

Bracket notation is used to access feature subproperties or arrays. For example:

```
{
    "temperatures" : {
        "scale" : "fahrenheit",
        "values" : [70, 80, 90]
    }
}
```

```
${temperatures['scale']} === 'fahrenheit'
${temperatures.values[0]} === 70
${temperatures['values'][0]} === 70 // Same as (temperatures[values])[0] and temperatures.values[0]
```

## Built-in Variables

The prefix `tiles3d_` is reserved for built-in variables. The following built-in variables are supported by the styling language:

- `tiles3d_tileset_time`

### tiles3d_tileset_time

Gets the time, in milliseconds, since the tileset was first loaded. This is useful for creating dynamic styles that change with time.

```
{
    "color" : "color() * abs(cos(${Temperature} + ${tiles3d_tileset_time}))"
}
```

## Built-in Functions

The following built-in functions are supported by the styling language:

- `abs`
- `sqrt`
- `cos`
- `sin`
- `tan`
- `acos`
- `asin`
- `atan`
- `atan2`
- `radians`
- `degrees`
- `sign`
- `floor`
- `ceil`
- `round`
- `exp`
- `log`
- `exp2`
- `log2`
- `fract`
- `pow`
- `min`
- `max`
- `clamp`
- `mix`
- `length`
- `distance`
- `normalize`
- `dot`
- `cross`

Many of the built-in functions take either scalars or vectors as arguments. For vector arguments the function is applied component-wise and the resulting vector is returned.

### abs

```
abs(x : Number) : Number
abs(x : vec2) : vec2
abs(x : vec3) : vec3
abs(x : vec4) : vec4
```

Returns the absolute value of `x` .

```
{
    "show" : "abs(${temperature}) > 20.0"
}
```

#### sqrt

```
sqrt(x : Number) : Number
sqrt(x : vec2) : vec2
sqrt(x : vec3) : vec3
sqrt(x : vec4) : vec4
```

Returns the square root of `x` when `x >= 0` . Returns `NaN` when `x < 0` .

```
{
    "color" : {
        "conditions" : [
            ["${temperature} >= 0.5", "color('#00FFFF')"],
            ["${temperature} >= 0.0", "color('#FF00FF')"]
        ]
    }
}
```

#### cos

```
cos(angle : Number) : Number
cos(angle : vec2) : vec2
cos(angle : vec3) : vec3
cos(angle : vec4) : vec4
```

Returns the cosine of `angle` in radians.

```
{
    "show" : "cos(${Angle}) > 0.0"
}
```

#### sin

```
sin(angle : Number) : Number
sin(angle : vec2) : vec2
sin(angle : vec3) : vec3
sin(angle : vec4) : vec4
```

Returns the sine of `angle` in radians.

```
{
    "show" : "sin(${Angle}) > 0.0"
}
```

#### tan

```
tan(angle : Number) : Number
tan(angle : vec2) : vec2
tan(angle : vec3) : vec3
tan(angle : vec4) : vec4
```

Returns the tangent of `angle` in radians.

```
{
    "show" : "tan(${Angle}) > 0.0"
}
```

#### acos

```
acos(angle : Number) : Number
acos(angle : vec2) : vec2
acos(angle : vec3) : vec3
acos(angle : vec4) : vec4
```

Returns the arccosine of `angle` in radians.

```
{
    "show" : "acos(${Angle}) > 0.0"
}
```

#### asin

```
asin(angle : Number) : Number
asin(angle : vec2) : vec2
asin(angle : vec3) : vec3
asin(angle : vec4) : vec4
```

Returns the arcsine of `angle` in radians.

```
{
    "show" : "asin(${Angle}) > 0.0"
}
```

#### atan

```
atan(angle : Number) : Number
atan(angle : vec2) : vec2
atan(angle : vec3) : vec3
atan(angle : vec4) : vec4
```

Returns the arctangent of `angle` in radians.

```
{
    "show" : "atan(${Angle}) > 0.0"
}
```

#### atan2

```
atan2(y : Number, x : Number) : Number
atan2(y : vec2, x : vec2) : vec2
atan2(y : vec3, x : vec3) : vec3
atan2(y : vec4, x : vec4) : vec4
```

Returns the arctangent of the quotient of `y` and `x`.

```
{
    "show" : "atan2(${GridY}, ${GridX}) > 0.0"
}
```

#### radians

```
radians(angle : Number) : Number
radians(angle : vec2) : vec2
radians(angle : vec3) : vec3
radians(angle : vec4) : vec4
```

Converts `angle` from degrees to radians.

```
{
    "show" : "radians(${Angle}) > 0.5"
}
```

#### degrees

```
degrees(angle : Number) : Number
degrees(angle : vec2) : vec2
degrees(angle : vec3) : vec3
degrees(angle : vec4) : vec4
```

Converts `angle` from radians to degrees.

```
{
    "show" : "degrees(${Angle}) > 45.0"
}
```

#### sign

```
sign(x : Number) : Number
sign(x : vec2) : vec2
sign(x : vec3) : vec3
sign(x : vec4) : vec4
```

Returns 1.0 when `x` is positive, 0.0 when `x` is zero, and -1.0 when `x` is negative.

```
{
    "show" : "sign(${Temperature}) * sign(${Velocity}) === 1.0"
}
```

#### floor

```
floor(x : Number) : Number
floor(x : vec2) : vec2
floor(x : vec3) : vec3
floor(x : vec4) : vec4
```

Returns the nearest integer less than or equal to `x` .

```
{
    "show" : "floor(${Position}) === 0"
}
```

#### ceil

```
ceil(x : Number) : Number
ceil(x : vec2) : vec2
ceil(x : vec3) : vec3
ceil(x : vec4) : vec4
```

Returns the nearest integer greater than or equal to `x` .

```
{
    "show" : "ceil(${Position}) === 1"
```

```
}
```

#### round

```
round(x : Number) : Number
round(x : vec2) : vec2
round(x : vec3) : vec3
round(x : vec4) : vec4
```

Returns the nearest integer to `x`. A number with a fraction of 0.5 will round in an implementation-defined direction.

```
{
    "show" : "round(${Position}) === 1"
}
```

#### exp

```
exp(x : Number) : Number
exp(x : vec2) : vec2
exp(x : vec3) : vec3
exp(x : vec4) : vec4
```

Returns `e` to the power of `x`, where `e` is Euler's constant, approximately `2.71828`.

```
{
    "show" : "exp(${Density}) > 1.0"
}
```

#### log

```
log(x : Number) : Number
log(x : vec2) : vec2
log(x : vec3) : vec3
log(x : vec4) : vec4
```

Returns the natural logarithm (base `e`) of `x`.

```
{
    "show" : "log(${Density}) > 1.0"
}
```

#### exp2

```
exp2(x : Number) : Number
exp2(x : vec2) : vec2
exp2(x : vec3) : vec3
exp2(x : vec4) : vec4
```

Returns 2 to the power of `x`.

```
{
    "show" : "exp2(${Density}) > 1.0"
}
```

#### log2

```
log2(x : Number) : Number
log2(x : vec2) : vec2
log2(x : vec3) : vec3
log2(x : vec4) : vec4
```

Returns the base 2 logarithm of `x` .

```
{
    "show" : "log2(${Density}) > 1.0"
}
```

### fract

```
fract(x : Number) : Number
fract(x : vec2) : vec2
fract(x : vec3) : vec3
fract(x : vec4) : vec4
```

Returns the fractional part of `x` . Equivalent to `x - floor(x)` .

```
{
    "color" : "color() * fract(${Density})"
}
```

### pow

```
pow(base : Number, exponent : Number) : Number
pow(base : vec2, exponent : vec2) : vec2
pow(base : vec3, exponent : vec3) : vec3
pow(base : vec4, exponent : vec4) : vec4
```

Returns `base` raised to the power of `exponent` .

```
{
    "show" : "pow(${Density}, ${Temperature}) > 1.0"
}
```

### min

```
min(x : Number, y : Number) : Number
min(x : vec2, y : vec2) : vec2
min(x : vec3, y : vec3) : vec3
min(x : vec4, y : vec4) : vec4
```

```
min(x : Number, y : Number) : Number
min(x : vec2, y : Number) : vec2
min(x : vec3, y : Number) : vec3
min(x : vec4, y : Number) : vec4
```

Returns the smaller of `x` and `y` .

```
{
    "show" : "min(${Width}, ${Height}) > 10.0"
}
```

### max

```
max(x : Number, y : Number) : Number
max(x : vec2, y : vec2) : vec2
max(x : vec3, y : vec3) : vec3
max(x : vec4, y : vec4) : vec4
```

```
max(x : Number, y : Number) : Number
max(x : vec2, y : Number) : vec2
max(x : vec3, y : Number) : vec3
max(x : vec4, y : Number) : vec4
```

Returns the larger of `x` and `y`.

```
{
    "show" : "max(${Width}, ${Height}) > 10.0"
}
```

### clamp

```
clamp(x : Number,  min : Number, max : Number) : Number
clamp(x : vec2,  min : vec2, max : vec2) : vec2
clamp(x : vec3,  min : vec3, max : vec3) : vec3
clamp(x : vec4,  min : vec4, max : vec4) : vec4
```

```
clamp(x : Number,  min : Number, max : Number) : Number
clamp(x : vec2,  min : Number, max : Number) : vec2
clamp(x : vec3,  min : Number, max : Number) : vec3
clamp(x : vec4,  min : Number, max : Number) : vec4
```

Constrains `x` to lie between `min` and `max`.

```
{
    "color" : "color() * clamp(${temperature}, 0.1, 0.2)"
}
```

### mix

```
mix(x : Number,  y : Number, a : Number) : Number
mix(x : vec2,  y : vec2, a : vec2) : vec2
mix(x : vec3,  y : vec3, a : vec3) : vec3
mix(x : vec4,  y : vec4, a : vec4) : vec4
```

```
mix(x : Number,  y : Number, a : Number) : Number
mix(x : vec2,  y : vec2, a : Number) : vec2
mix(x : vec3,  y : vec3, a : Number) : vec3
mix(x : vec4,  y : vec4, a : Number) : vec4
```

Computes the linear interpolation of `x` and `y`.

```
{
    "show" : "mix(20.0, ${Angle}, 0.5) > 25.0"
}
```

### length

```
length(x : Number) : Number
length(x : vec2) : vec2
length(x : vec3) : vec3
length(x : vec4) : vec4
```

Computes the length of vector `x`, i.e. the square root of the sum of the squared components. If `x` is a number, `length` returns `x`.

```
{
    "show" : "length(${Dimensions}) > 10.0"
}
```

### distance

```
distance(x : Number, y : Number) : Number
distance(x : vec2, y : vec2) : vec2
```

```
distance(x : vec3, y : vec3) : vec3
distance(x : vec4, y : vec4) : vec4
```

Computes the distance between two points `x` and `y`, i.e. `length(x - y)`.

```
{
    "show" : "distance(${BottomRight}, ${UpperLeft}) > 50.0"
}
```

#### normalize

```
normalize(x : Number) : Number
normalize(x : vec2) : vec2
normalize(x : vec3) : vec3
normalize(x : vec4) : vec4
```

Returns a vector with length 1.0 that is parallel to `x`. When `x` is a number, `normalize` returns 1.0.

```
{
    "show" : "normalize(${RightVector}, ${UpVector}) > 0.5"
}
```

#### dot

```
dot(x : Number, y : Number) : Number
dot(x : vec2, y : vec2) : vec2
dot(x : vec3, y : vec3) : vec3
dot(x : vec4, y : vec4) : vec4
```

Computes the dot product of `x` and `y`.

```
{
    "show" : "dot(${RightVector}, ${UpVector}) > 0.5"
}
```

#### cross

```
cross(x : vec3, y : vec3) : vec3
```

Computes the cross product of `x` and `y`. This function only accepts `vec3` arguments.

```
{
    "color" : "vec4(cross(${RightVector}, ${UpVector}), 1.0)"
}
```

### Notes

Comments are not supported.

## Batch Table Hierarchy

The styling language provides the following built-in functions intended for use with the Batch Table Hierarchy:

- `getExactClassName`
- `isExactClass`
- `isClass`

#### getExactClassName

```
getExactClassName() : String
```

Returns the feature's class name, or `undefined` if the feature is not a class instance.

For example, the following style will color all doorknobs yellow, all doors green, and all other features gray.

```
{
    "defines" : {
        "suffix" : "regExp('door(.*)').exec(getExactClassName())"
    },
    "color" : {
        "conditions" : [
            ["${suffix} === 'knob'", "color('yellow')"],
            ["${suffix} === ''", "color('green')"],
            ["${suffix} === null", "color('gray')"],
            ["true", "color('blue'"]
        ]
    }
}
```

#### isExactClass

```
isExactClass(name : String) : Boolean
```

Returns `true` is the feature's class is equal to `name`, otherwise `false`.

For example, the following style will color all doors, but not features that are children of doors (like doorknobs).

```
"color" : {
    "conditions" : [
        ["isExactClass('door')", "color('red')"],
        ["true", "color('white')"]
    ]
}
```

#### isClass

```
isClass(name : String) : Boolean
```

Returns `true` is the feature's class, or any of its ancestors' classes, are equal to `name`.

For example, the style below will color all doors and doorknobs.

```
"color" : {
    "conditions" : [
        ["isClass('door')", "color('blue')"],
        ["true", "color('white')"]
    ]
}
```

## Point Cloud

A Point Cloud is a collection of points that may be styled like other features. In addition to evaluating a point's `color` and `show` properties, a point cloud style may evaluate `pointSize`, or the size of each point in pixels. The default `pointSize` is `1.0`.

```
{
    "color" : "color('red')",
    "pointSize" : "${Temperature} * 0.5"
}
```

Implementations may clamp the evaluated `pointSize` to the system's supported point size range. For example, WebGL renderers may query `ALIASED_POINT_SIZE_RANGE` to get the system limits when rendering with `POINTS`. A `pointSize` of `1.0` must be supported.

Point cloud styles may also reference semantics from the Feature Table including position, color, and normal to allow for more flexible styling of the source data.

- `${POSITION}` is a `vec3` storing the xyz Cartesian coordinates of the point before the `RTC_CENTER` and tile transform are applied. When the positions are quantized, `${POSITION}` refers to the position after the `QUANTIZED_VOLUME_SCALE` is applied, but before `QUANTIZED_VOLUME_OFFSET` is applied.
- `${POSITION_ABSOLUTE}` is a `vec3` storing the xyz Cartesian coordinates of the point after the `RTC_CENTER` and tile transform are applied. When the positions are quantized, `${POSITION_ABSOLUTE}` refers to the position after the `QUANTIZED_VOLUME_SCALE`, `QUANTIZED_VOLUME_OFFSET`, and tile transform are applied.
- `${COLOR}` evaluates to a `Color` storing the rgba color of the point. When the feature table's color semantic is `RGB` or `RGB565`, `${COLOR}.alpha` is `1.0`. If no color semantic is defined, `${COLOR}` evaluates to the application-specific default color.
- `${NORMAL}` is a `vec3` storing the normal, in Cartesian coordinates, of the point before the tile transform is applied. When normals are oct-encoded `${NORMAL}` refers to the decoded normal. If no normal semantic is defined in the feature table, `${NORMAL}` evaluates to `undefined`.

For example:

```
{
    "color" : "${COLOR} * color('red')'",
    "show" : "${POSITION}.x > 0.5",
    "pointSize" : "${NORMAL}.x > 0 ? 2 : 1"
}
```

#### Point Cloud Shader Styling

**TODO : add note about GLSL implementations requires strict type comparisons among other things: https://github.com/AnalyticalGraphicsInc/3d-tiles/issues/140**

## File Extension

TBA

## MIME Type

*TBA, #60*

`application/json`

## Acknowledgments

- Piero Toffanin, @pierotofy