SYSTEM EXPRESSIONS

View online: https://www.construct.net/en/make-games/manuals/construct-3/system-reference/system-expressions

This section outlines the expressions in the built-in System object in Construct. Many are common mathematical operations, and they can be listed with descriptions in the Expressions dictionary, but they are included here for completeness.

This section does not list the operators or syntax that can be used in expressions - just the expressions specific to the System object. For more general information on how to use expressions in Construct, see Expressions.

Display

OriginalViewportWidth

OriginalViewportHeight

Get the original values of the *Viewport size* project property. The value of these expressions does not change when using fullscreen modes like *Scale inner* and *Scale outer* that change the viewport size depending on the available screen area. Note however that the system *Set canvas size* action does modify the values returned by these expressions when in any fullscreen mode other than *Off*, as in this case it has the effect of changing the *Viewport size* project property.

Layers

In expressions where a layer is required, either its name (as a string) or index (as a number, zero-based) can be entered.

CanvasToLayerX(layer, x, y)

CanvasToLayerY(layer, x, y)

Calculate the layout co-ordinates underneath a position in canvas CSS co-ordinates for a given layer.

The 3D Camera plugin also provides expressions with the same name that work in 3D.

LayerToCanvasX(layer, x, y)

LayerToCanvasY(layer, x, y)

Calculate the canvas CSS co-ordinates above a position in layout co-ordinates for a given layer.

> *The 3D Camera plugin also provides expressions with the same name that work in 3D.*

LayerToLayerX(fromLayer, toLayer, x, y)

LayerToLayerY(fromLayer, toLayer, x, y)

Calculate the position on a second layer (*toLayer*) that corresponds to a position given on a first layer (*fromLayer*). This is a shorthand for converting a layer position to canvas CSS co-ordinates, and then back to a position on a different layer.

> *The 3D Camera plugin also provides expressions with the same name that work in 3D.*

LayerAngle(layer)

Get the angle, in degrees, of a layer.

LayerIndex(layer)

Get the zero-based index of a layer from its name.

LayerOpacity(layer)

Get the opacity (or semitransparency) of a layer, from 0 (transparent) to 100 (opaque).

LayerParallaxX(layer)

LayerParallaxY(layer)

Get the current parallax X and Y components of a layer.

LayerScale(layer)

Get the current scale of the layer, not including the overall layout scale.

LayerScaleRate(layer)

Get the current scale rate of the layer, which defines how quickly it scales (if at all).

LayerScrollX(layer)

LayerScrollY(layer)

Get the current scroll position of a specific layer. Note this is always the same as the layout scroll position (given by the *ScrollX* and *ScrollY* system expressions) unless a layer was independently scrolled using the *Set layer scroll* system action.

LayerZElevation(layer)

Get the current elevation of the layer on the Z axis.

ViewportBottom(layer)

ViewportLeft(layer)

ViewportRight(layer)

ViewportTop(layer)

Return the viewport boundaries in layout co-ordinates of a given layer. Not all layers have the same viewport if they are parallaxed, scaled or rotated separately.

> *The 3D Camera plugin also provides viewport expressions that work in 3D.*

ViewportMidX(layer)

ViewportMidY(layer)

Return the mid-point of the viewport area in layout co-ordinates for a given layer.

ViewportWidth(layer)

ViewportHeight(layer)

Return the size of the viewport in layout co-ordinates for a given layer.

Layout

CanvasSnapshot

Contains the resulting image from a *Snapshot canvas* action after *On canvas snapshot* has triggered. (Note this expression is not available immediately after the *Snapshot canvas* action - you can only use it after *On canvas snapshot* triggers.) The expression returns a data URI of the image file. This can be loaded in to a Sprite or Tiled Background object via *Load image from URL*, sent to a server or stored locally, or opened with the Browser object in a new tab to save to disk.

LayoutAngle

Get the angle, in degrees, of the current layout. This does not include the rotation of individual layers.

LayoutScale

Get the current scale of the entire layout set by the *Set layout scale* action. This does not include the scaling of individual layers.

LayoutWidth

LayoutHeight

Get the size of the current layout in pixels.

LayoutName

Get the name of the current layout.

ScrollX

ScrollY

Get the current position the view is centered on.

VanishingPointX

VanishingPointY

Get the current vanishing point in the layout, where the range 0-100 represents the viewport. For more information refer to the *Vanishing point* layout property.

Math

These expressions are simply ordinary math functions like you find on calculators. However, note that all functions using an angle take it in **degrees**, not radians. Angles start with 0 degrees facing right and increment clockwise.

sin(x), **cos(x)**, **tan(x)**, **asin(x)**, **acos(x)**, **atan(x)** Trigonometric functions using angles in *degrees*.

abs(x) Absolute value of x e.g. `abs(-5)` = 5

angle(x1, y1, x2, y2) Calculate angle between two points

anglelerp(a, b, x) Linearly interpolate the angle *a* to *b* by *x*. Unlike the standard *lerp*, this takes in to account the cyclical nature of angles.

anglediff(a1, a2) Return the smallest difference between two angles

anglerotate(start, end, step) Rotate angle *start* towards *end* by the angle *step*, all in degrees. If *start* is less than *step* degrees away from *end*, it returns *end*.

ceil(x) Round up x e.g. `ceil(5.1)` = 6

cosp(a, b, x) Cosine interpolation of *a* to *b* by *x*. Calculates *(a + b + (a - b) * cos(x * 180°)) / 2*.

cubic(a, b, c, d, x) Cubic interpolation through *a*, *b*, *c* and *d* by *x*. Calculates *lerp(qarp(a, b, c, x), qarp(b, c, d, x), x)*.

distance(x1, y1, x2, y2) Calculate distance between two points

exp(x) Calculate e^x

floor(x) Round down x e.g. `floor(5.9)` = 5

infinity A floating point number value representing infinity.

lerp(a, b, x) Linear interpolation of *a* to *b* by *x*. Calculates *a + x * (b - a)*.

unlerp(a, b, y) Reverse linear interpolation: if lerp(a, b, x) = y, then unlerp(a, b, y) = x. Calculates *(y - a) / (b - a)*.

ln(x) Log to base e of x.

log10(x) Log to base 10 of x.

max(a, b [, c...]), **min(a, b [, c...])** Calculate maximum or minimum of the given numbers. Any number of parameters can be used as long as there are at least two.

pi The mathematical constant pi (3.14159...)

qarp(a, b, c, x) Quadratic interpolation through *a*, *b* and *c* by *x*. Calculates *lerp(lerp(a, b, x), lerp(b, c, x), x)*.

round(x) Round x to the nearest whole number e.g. `round(5.6)` = 6

roundToDp(x, digits) Round x to a given number of decimal places, e.g. `roundToDp(1.666666, 2)` = 1.67

sign(x) Retrieve the sign of *x*: -1 for any negative number, 1 for any positive number, or 0 if *x* is zero.

sqrt(x) Calculate square root of x e.g. `sqrt(25)` = 5

---

getbit(x, n)

Get the nth bit of x represented as a 32-bit integer. For example `getbit(7, 0)` will get the least significant bit of the number 7 when represented as a 32-bit integer. Returns either 0 or 1.

---

setbit(x, n, b)

Set the nth bit of x represented as a 32-bit integer to b (either 0 or 1). The resulting 32-bit integer is returned.

---

togglebit(x, n)

Toggle the nth bit of x represented as a 32-bit integer. If that bit is 0, it is set to 1; if it is 1, it is set to 0. The resulting 32-bit integer is returned.

Memory management

ImageLoadingProgress

Return the current loading progress of any memory management *Load* actions that are currently busy, on a scale of 0-1.

Save & Load

SaveStateJSON

In *On save complete* or *On load complete*, returns a string of JSON data representing the savegame data. This can later be loaded using the *Load from JSON* action. For more information see How to make savegames.

System

CurrentEventNumber

CurrentEventSheetName

Return the number of the current event being run, and the name of the event sheet it belongs to. These are useful for testing purposes, such as logging the current event number to the browser console.

ImageMemoryUsage

Returns the estimated total memory usage, in megabytes, of all the currently-loaded images. Note image memory is sometimes also referred to as "VRAM", but this is not strictly correct since not all devices have video-specific memory. Also remember this expression does not include the memory use of sounds, code, or other non-image resources.

LoadingProgress

Return the current load progress on a loader layout. The progress is returned as a number from 0 to 1, e.g. 0.5 for half complete. For more information, see the tutorial how to make a custom loading screen.

LoopIndex

Get the index (number of repeats so far) in any currently running loop.

LoopIndex(name)

Get the index (number of repeats so far) of the loop with the given name. Useful for getting indices in nested loops.

ObjectCount

The total number of objects currently created.

ProjectFileCount

ProjectFileNameAt(index)

Retrieve the list of project files in this project at the time of preview or export. Files in the Sounds, Music, Videos, Fonts, Icons & Screenshots and Files folders are included. Each entry is a string with the relative path to the file, e.g. `subfolder/mydata.json`.

> *This file list only reflects the state of the project at the time it was previewed or exported. If you export the project and then change some of the files, this list will not update to reflect the change. If you need a list that updates to post-export changes, consider another approach, such as using File System to list folder contents on desktop, or a separate data file to list files in which can also be updated after export.*

ProjectID

Return the project ID as it appears in Project Properties.

ProjectName

Return the name of the project as it appears in Project Properties.

ProjectUniqueID

Return a string of random characters that Construct automatically generated to uniquely identify this specific project.

ProjectVersion

Return the version entered in to Project Properties. Note that this is always returned as a string, not a number.

Text

find(src, text)

findCase(src, text)

Find the first index within *src* that *text* occurs, else returns -1. *find* is case-insensitive, and *findCase* is case-sensitive.

left(text, count)

Return the first *count* characters of *text*.

---

len(text)

Return the number of characters in *text*.

---

lowercase(text)

Convert the given text to all lowercase.

---

mid(text, index, count)

Return the *count* characters starting from *index* in *text*. Note *count* can be set to -1 to return characters from *index* to the end of the string.

---

newline

A string containing a line break. Use to insert line breaks in to strings, e.g. `"Hello" & newline & "World"`

RegexMatchAt(String, Regex, Flags, Index)

Process the regular expression *Regex* on *String* with *Flags*, and in the list of results, return the entry at *Index*.

RegexMatchCount(String, Regex, Flags)

Process the regular expression *Regex* on *String* with *Flags*, and return the number of entries in the list of results.

RegexReplace(String, Regex, Flags, Replace)

In *String* substitute matches for the regular expression *Regex* (with *Flags*) with the string *Replace*. The replacement string can contain the following special characters: **$$** (inserts a $), **$&** (inserts the matched substring), **$`** (inserts the portion of the string that precedes the matched substring), or **$'** (inserts the portion of the string that follows the matched substring).

RegexSearch(String, Regex, Flags)

Return the index of the first character in *String* where a match for *Regex* with *Flags* could be found.

---

replace(src, find, rep)

Find all occurrences of *find* in *src* and replace them with *rep*.

right(text, count)

Return the last *count* characters of *text*.

StringSub(text, sub1 [, sub2...])

Substitute placeholders of the form `{n}` in the given string. This expression accepts a variable number of parameters. For example `StringSub("Hello {0}!", "Sam")` returns *Hello Sam!*, as the placeholder `{0}` is replaced with the first provided additional parameter. Further parameters can be provided by increasing the number in the placeholder, e.g. `StringSub("Hi {0}, your score is {1}!", "Sam", 100)` returns *Hi Sam, your score is 100!*. Note the substitutions can be either strings or numbers. If there are multiple occurrences of the same placeholder, they are all replaced. If a placeholder is used for which a parameter is not provided, then it is left as-is, e.g. `StringSub("Hi {0}, your score is {1}!", "Sam")` returns *Hi Sam, your score is {1}!* as there is only one substitution provided.

tokenat(src, index, separator)

Return the Nth token from *src*, splitting the string by *separator*. For example, `tokenat("apples|oranges|bananas", 1, "|")` returns *oranges*.

Use the Array object's Split string action for more flexibility. To better handle more complex data, use a more robust data format like JSON.

tokencount(src, separator)

Count how many tokens occur in *src* using *separator*. For example, `tokencount("apples|oranges|bananas", "|")` returns 3.

trim(src)

Return *src* with all whitespace (spaces, tabs etc.) removed from the beginning and end of the string.

uppercase(text)

Convert the given text to all uppercase.

URLEncode(str)

URLDecode(str)

Convert to and from a string in a format suitable for including in a URL or POST data.

zeropad(number, digits)

Pad *number* out to a certain number of *digits* by adding zeroes in front of the number, then returning the result as a string. For example, `zeropad(45, 5)` returns the string "00045".

Time

CPUUtilisation

The percentage of the last second that was spent in logic, such as running events or processing behaviors. This is for performance measurements. Note on most devices the rendering happens on the separate GPU and so is not counted by this measurement; for that *fps* or *GPUUtilisation* is a better measure. Also note this measurement is based on timers so should be treated as an approximation, and it only measures time on the main thread.

This measurement can be unreliable, especially when the system is largely idle. Most modern devices deliberately slow down the CPU if not fully loaded in order to save power. This means work takes longer to get done, and this expression will misleadingly return a higher measurement, since it's based on timing how long the work takes. It will generally only be reliable in the device's maximum performance mode, i.e. under full load.

dt

Delta-time in seconds. See Delta-time and framerate independence.

fps

How many frames per second (FPS) the project is rendering. The most common display refresh rate is 60 Hz, so typically an efficiently designed project will render at 60 FPS. Note however if nothing is changing on-screen, then nothing is rendered, and so the FPS measurement may fall to 0 or display a lower result; this does not indicate poor performance, only that fewer frames are necessary to render. The *TicksPerSecond* expression of the Platform Info object indicates how frequently the engine is stepping, which may be different to the frames rendered per second.

GPUUtilisation

The percentage of the last second that was spent rendering graphics. This represents how busy the graphics processing unit (GPU) is, which is useful for performance measurements. This measurement is based on timers so should be treated as an approximation. The GPU utilisation is only affected by the amount of rendering work to be done, such as the number of objects visible on-screen, and also increases if the window size is larger. Note this measurement is only available on certain systems. If it is not supported, it will return NaN (a special value representing Not A Number) to indicate there is no value available.

> *See the note under CPUUtilisation about possibly unreliable measurements in some circumstances. This also applies to the GPU and can affect this measurement too.*

tickcount

The number of ticks that have run since the project started.

time

The number of seconds since the project started, taking in to account the time scale.

timescale

The current time scale.

wallclockdt

This is the same as the *dt* expression, but is not affected by the time scale. This can be useful to still obtain the real-world delta-time value when the project is paused by setting the time scale to 0, which causes *dt* to also become 0.

wallclocktime

The number of seconds since the project started, not taking in to account the time scale (i.e. the real-world time).

> *Unlike other time expressions, wallclocktime can update during the same tick, such as within a long loop. This means it can be used for things like performance measurements, or doing work for a fixed period of time.*

Values

choose(a, b [, c...])

Choose one of the given parameters at random. E.g. `choose(1, 3, 9, 20)` randomly picks one of the four numbers and returns that. This also works with strings, e.g. `choose("Hello", "Goodbye")` returns either *Hello* or *Goodbye*. Any number of parameters can be used as long as there are at least two.

chooseindex(index, value0[, ...])

Choose one of the given parameters by a zero-based index. For example:

- `chooseindex(0, "foo", "bar", "baz")` returns *"foo"*

- `chooseindex(1, "foo", "bar", "baz")` returns *"bar"*
- `chooseindex(2, "foo", "bar", "baz")` returns *"baz"*

Any number of parameters can be included after the index (but there must be at least one). If the index is out of range, it returns either the first or last value, e.g. in the above example an index of -1 will still return *"foo"*.

clamp(x, lower, upper)

Return *lower* if *x* is less than *lower*, *upper* if *x* is greater than *upper*, else return *x*.

float(x)

Convert the integer or text *x* to a float (fractional number). If *x* is text, non-numeric characters are allowed after the number, but not before. For example `float("3.1xx")` returns *3.1*, but `float("xx3.1")` returns 0.

int(x)

Convert the float or text *x* to an integer (whole number). If *x* is text, non-numeric characters are allowed after the number, but not before. For example `int("33xx")` returns 33, but `int("xx33")` returns 0.

random(x)

Generate a random float from 0 to *x*, not including *x*. E.g. `random(4)` can generate 0, 2.5, 3.29293, but not 4. Use `floor(random(4))` to generate just the whole numbers 0, 1, 2, 3.

random(a, b)

Generate a random float between *a* and *b*, including a but not including *b*.

rgbEx(r, g, b)

rgbEx255(r, g, b)

rgba(r, g, b, a)

rgba255(r, g, b, a)

Generate a single number containing a color with the given red, green, blue and optionally alpha components. *rgbEx* and *rgba* use components in the range 0-100, whereas *rgbEx255* and *rgba255* use components in the range 0-255. When an alpha is not provided, the resulting color is opaque. These are useful for conditions or actions taking a color parameter.

HexColor(string)

Convert a string with a color in hexadecimal notation to a numerical color value. The hexadecimal notation is case insensitive and can optionally start with `#`, and can use three or six character forms for opaque colors, or four or eight character forms to include alpha. For example `"F00"`, `"#f00"`, `"#FF0000"` and `"ff0000"` are all interpreted as opaque red.

ColorToHexString(value)

Convert a numerical color value back in to a string with hexadecimal notation. The string will always be lowercase and begin with a `#`, and use six characters for opaque colors, or eight characters to include alpha. For example this will return `"#ff0000"` for opaque red.

---

str(x)

Convert the integer or float *x* to a string. Generally this is not necessary since strings can be built using the & operator, e.g. `"Your score is " & score`