# Preface

# Algodoo Scripting Guide

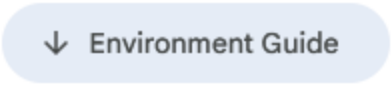
*by **Doc Desk** with help from **Lapse** and **Erikfassett**.*

This guide attempts to teach scripting in the 2D physics simulator Algodoo. The language Algodoo uses for scripting is **Thyme**, a language created specifically for scripting in Algodoo. Unlike most of the current resources on Thyme, this guide attempts to teach Thyme as a programming language. It goes through several programming concepts as well as providing lists of useful built-in properties, functions, operators, and detailing several Algodoo-specific techniques that will help you make the most of scripting in Algodoo.

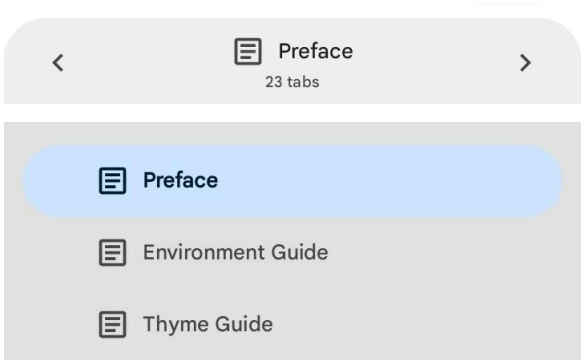
*Some of the information in italics is extra that you don't need to know, but may provide interesting information.*

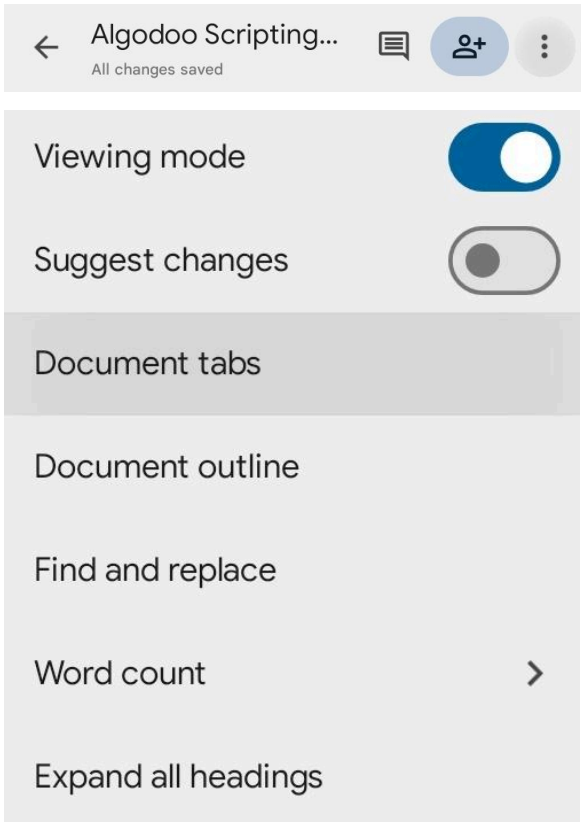
## Document tabs

Sections have been split into document tabs: you can navigate them using the left pane, or scroll down and mouse over the bottom of the page to show a button to the next section.




On the mobile app: you can tap the bottom menu that pops up; or tap the three buttons from the top menu, then Document tabs.




Algodoo Scripting...
All changes saved
Viewing mode
Suggest changes
Document tabs
Document outline
Find and replace
Word count
Expand all headings

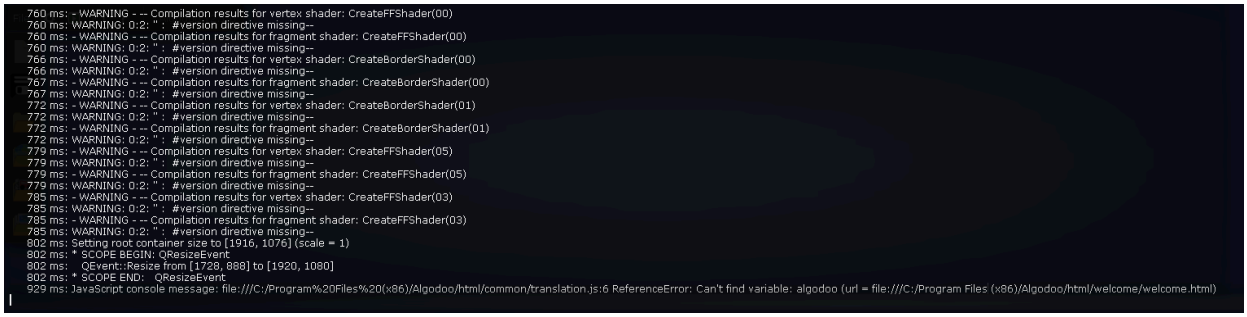
# Environment Guide

# Environment Guide

Before we start learning about the Thyme language itself, it's important to get used to the environment Algodoo provides us to write our Thyme code.

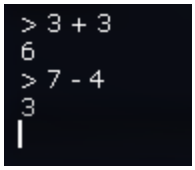
## Console

The console is a panel where you can enter Thyme code, find properties, and view error messages. To open the console, press F10, the tilde `~` key (Mac only), or the backtick key `` ` `` on your keyboard. You should see something like this:

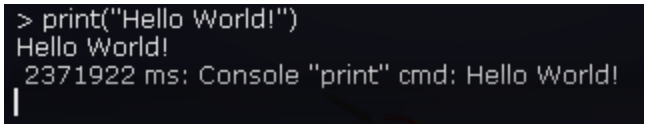



Don't worry about these error messages - they don't mean that anything will go wrong for you.

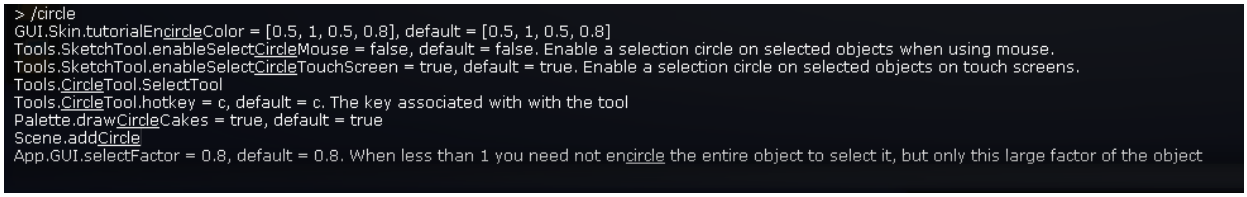
To start with, try entering some basic arithmetic expressions, such as `3 + 3` or `7 - 4`.



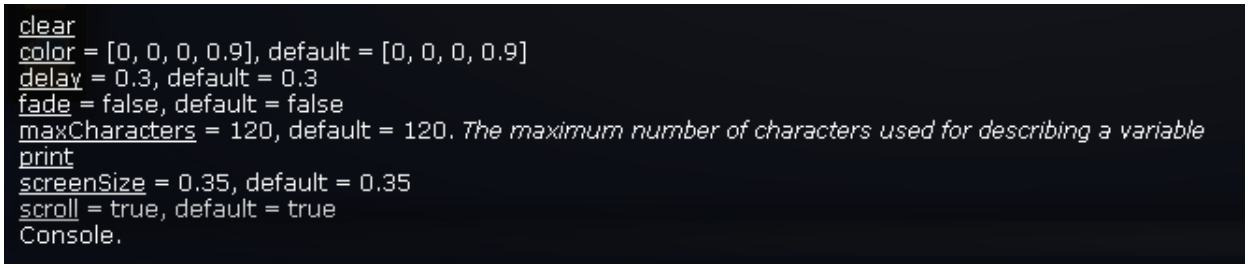

You can also use functions! The `print` function prints a message to the console. Try printing `"Hello World!"` to the console:



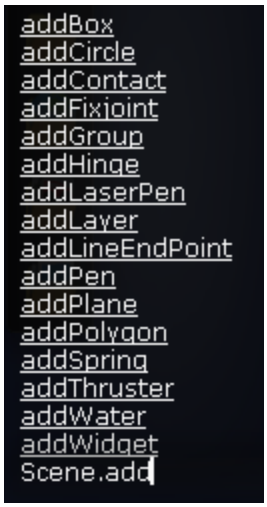
You can find properties using the console by preceding the search term with `/`. For example, if you wanted to see all the properties relating to circles, type `/circle`.




You can see the properties inside a specific object by typing the name of the object followed by `.`, then the tab key on your keyboard. For example, if you wanted to see all the console's properties, type `Console.` then tab:




If you press tab while typing an identifier (e.g. a variable name), Algodoo will attempt to autocomplete it. For example, typing `con` then pressing tab will change `con` to `Console.`. If there are multiple variables beginning with the characters you just typed, the console will show all the possibilities, e.g.:

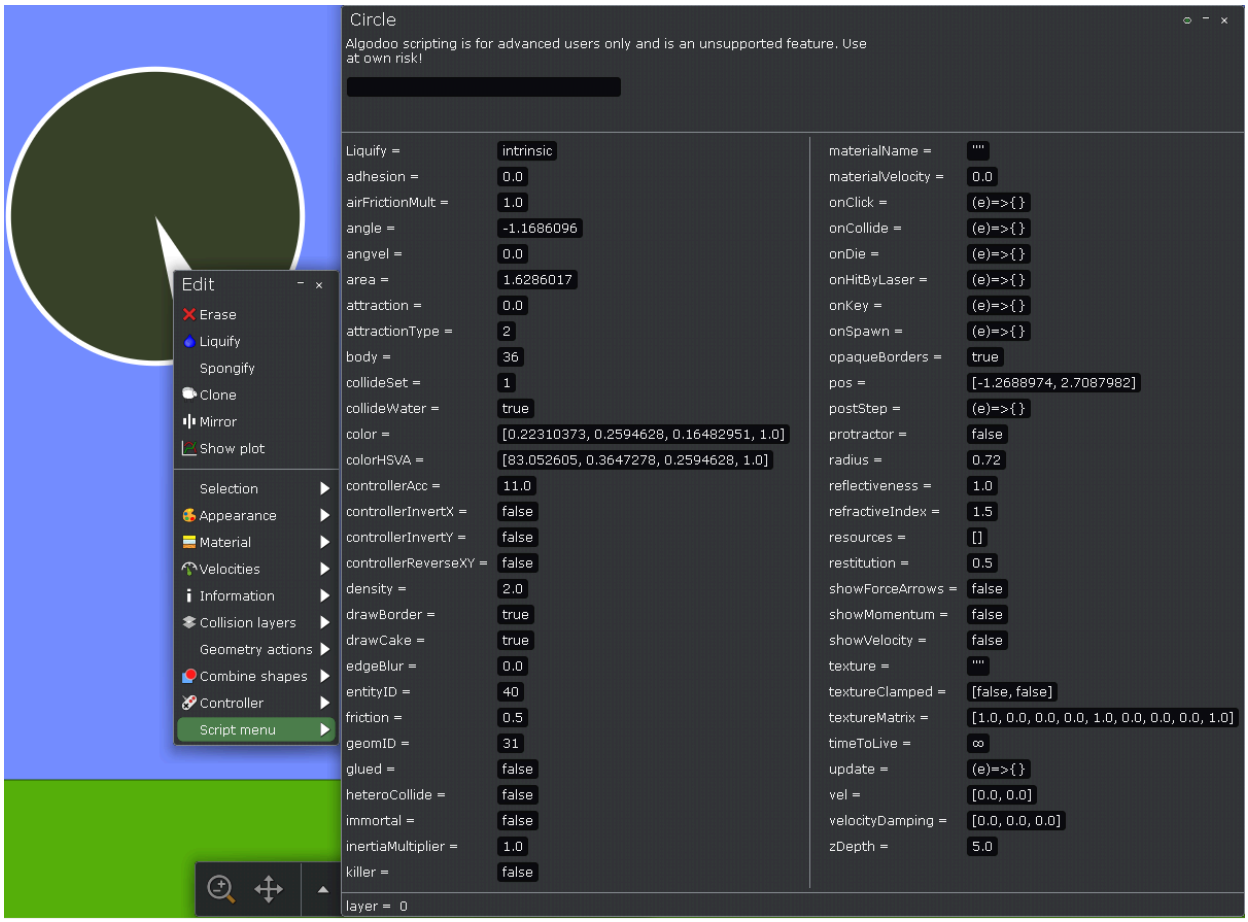



A few more tips:

- Scroll through the console using the Page Up and Page Down keys.
- Use previously entered commands by pressing the up and down arrow keys.

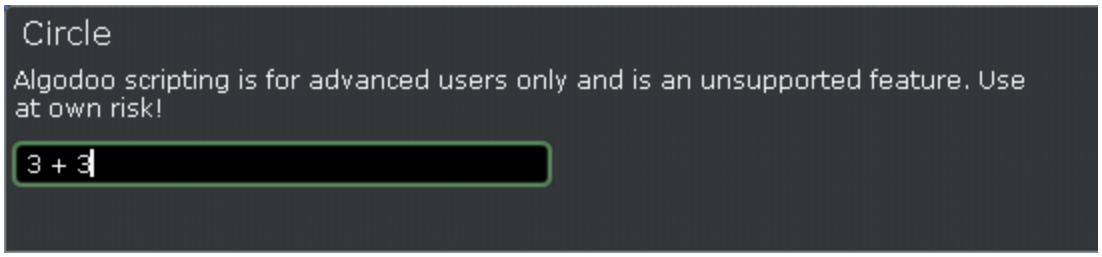
# Script menu

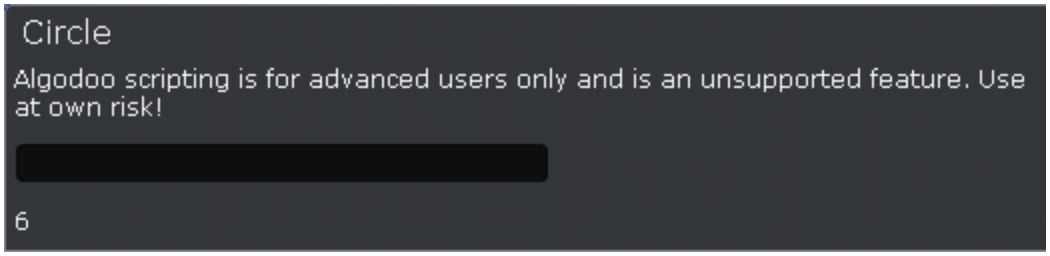
The script menu is found in the menu that appears when you right-click on an object. It looks like this:




## Object console

At the top of the script menu is a black input box with no official name. This will be called the **object console** for the rest of this document, as it functions similarly to the console. If you try putting in arithmetic operations again, you'll find that they still work:






Several features of the console apply in the object console, however:

- You cannot use / to find properties.
- Pressing tab will only autocomplete if one autocompletion is possible. Otherwise, it inserts a tab character.
- You cannot scroll using the Page Up and Page Down keys.
- You cannot use previously entered commands by pressing the up and down arrow keys.

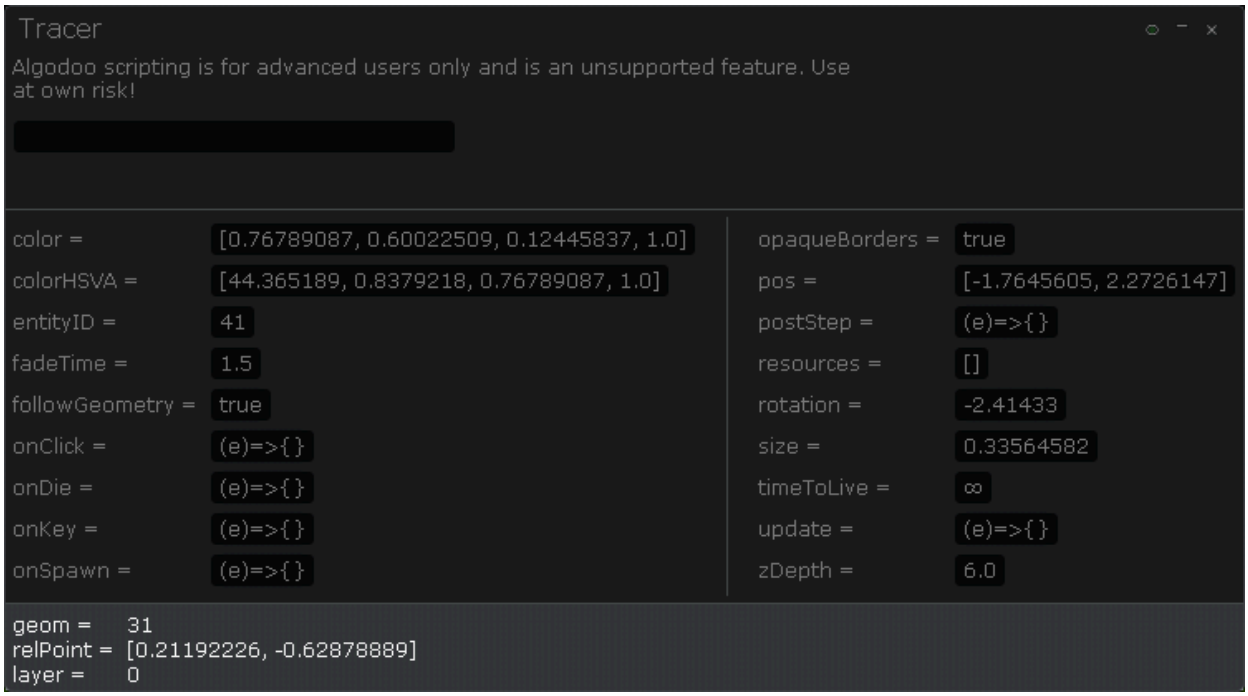
## Object properties

Below the object console, all the properties of the objects are listed. Some can be changed (e.g. you can change the radius of a circle by entering a different value for `radius`):


Some properties are read-only, e.g. `area`:

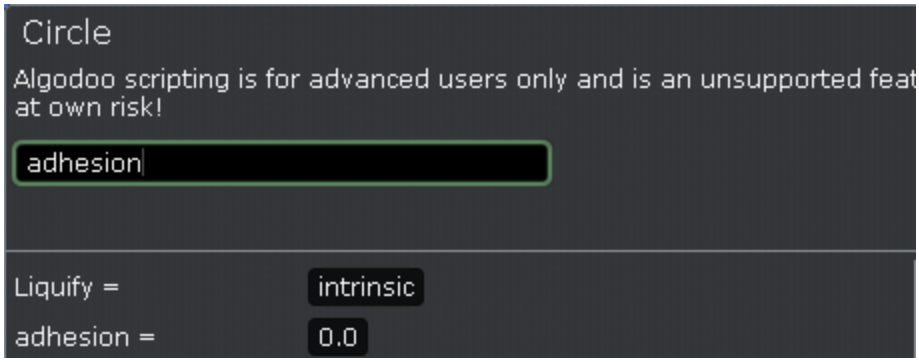

You cannot enter a value for a read-only variable, nor change it via a script.

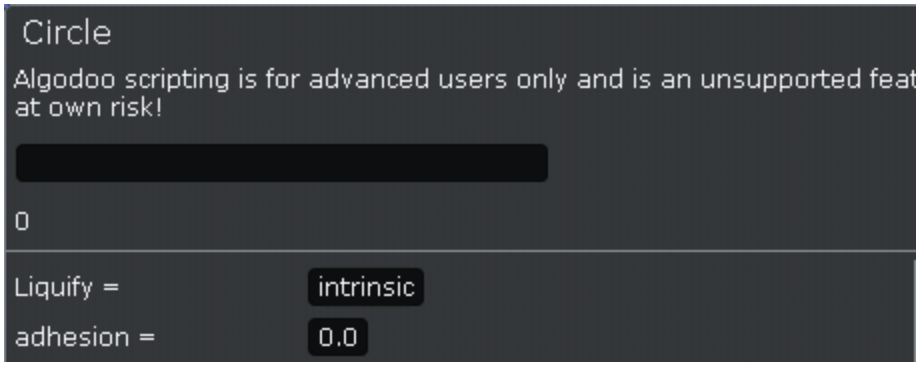
The variables listed below the normal variables are what could be considered private variables, for they are inaccessible to even read through normal means. However, they can be read by using the `readable` function on the object, which generates a read-only copy of the object where all variables, including these, can be read.




Different types of objects have different properties (e.g. each circle has a `radius` property).

All of these properties can be accessed (except the variables listed below the normal variables, for an unknown reason) through the object console:



When making changes in a property's text field, Algodoo will save a history of those changes for that field until the Script menu is closed.

You can then hit Ctrl+Z to undo, and Ctrl+Y to redo changes while the field is selected.
This allows you to recover scripts that Algodoo rejects and go back to fix any mistakes without having to rewrite the entire thing again.

# Thyme Guide

# Thyme Guide

Learn the basics of Thyme commonly used in Algodoo scripts from the following sections, navigate using the left pane or click on a link to a section.

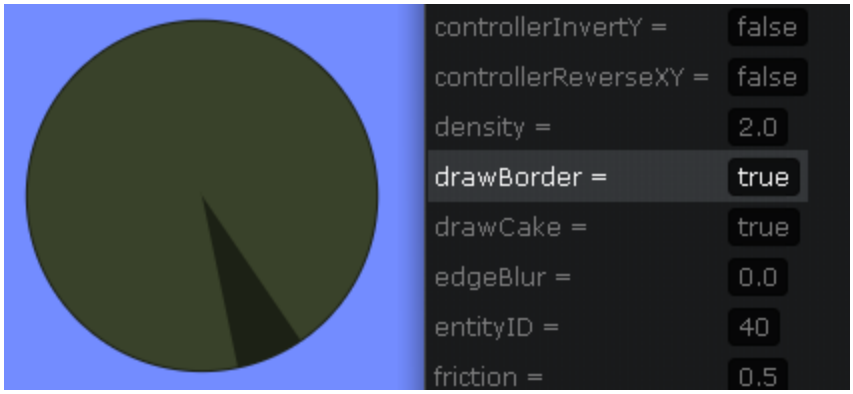
# Basic data types

# Basic data types

There are four primitive data types in Thyme - `bool`, `float`, `int` and `string`. There are also three aggregate (formed of several different elements) data types - `list`, `function` and `ClassObject`.

## bool

A boolean has only two possible values: `true` or `false`. An example of a boolean property is `drawBorder` - if `drawBorder` is `true`, the polygon's border is drawn; if `drawBorder` is `false`, the polygon's border is not drawn.




`drawBorder` is `true` here, so the circle's border is drawn.

## int

An integer is a whole number (i.e. a number with no decimal part). In Thyme, the int data type can store any integer between `-2147483648` and `2147483647`. An example of an integer property is `entityID` - each entity needs a unique ID number and the number of entities is an integer:


### Binary literals

A binary literal starts with `0b`, followed by any binary integer with a denary value between `-2147483648` and `2147483647` inclusive. The binary is interpreted as the exact data of a signed 32-bit integer. Any number larger than what 32 bit allows is truncated to 32 bits by cutting off the most significant bits, limiting you to 32 binary digits.
For example, `0b0101` returns `5`.

Due to the nature of how signed integers work, negative numbers are extremely unwieldy to write. For example, `0b11111111111111111111111111111001` returns `-7`. When the most significant digit of a 32 digit long binary number is set to `1`, then a negative number is returned, counting up from the minimum to `-1` at the highest possible value. If you don't intend to work with negative binary numbers, then you may treat this method as entering in a basic binary number, as long as you don't exceed the positive maximum.

### Hexadecimal literals

A hexadecimal literal starts with `0x`, followed by any hexadecimal integer with a denary value between `-2147483648` and `2147483647` inclusive. The hexadecimal is interpreted as a representation of a signed 32-bit integer where each digit represents four binary digits. Any number larger than what 32 bit allows is truncated to 32 bits by cutting off the most significant bits, limiting you to 8 hexadecimal digits.
For example, `0xFF` returns `255`.
Due to the nature of how signed integers work, negative numbers are interesting to write. For example, `0x7FFFFFFF` returns `2147483647`, but `0x80000000` returns `-2147483648`, and `0xFFFFFFFF` returns `-1`. Any 8-digit hexadecimal number where the most significant digit is equal to or greater than `8` will return a negative number, counting up from the minimum to `-1` at the highest possible value. If you don't intend to work with negative hexadecimal numbers, then you may treat this method as entering in a basic hexadecimal number, as long as you don't exceed the positive maximum.

*A note on literals, Algodoo will automatically resolve binary and hexadecimal literals into the familiar dernary, or base-10 numbers we use. This does significantly reduce the utility of these literals, as once they are input into a script, they are turned into regular integers.*

## float

A floating-point number represents a number with a decimal part. In Thyme, a float can represent a wide range of real numbers to 6-9 significant figures of accuracy (this is known as a single-precision float). The largest number that can be represented by a float is approximately `3.4028235e+038` (anything higher will be treated as infinity). An example of a float property is `radius` - the radius of a circle can be a decimal.


One important thing to keep in mind is that in any case where a function requires a float, an integer is also an acceptable type. Whenever this document lists that a function requires a float, you may also use an integer in place of the float.

*In the console, if a float value can be represented as an integer (e.g. `2.0`), Algodoo will return the result as an integer.*

## Angles

All angles in Algodoo are internally stored as radians (though most GUI displays angle in degrees). By convention, and for ease of mathematics, the direction o radians points to is right, in other words the positive X direction. An increase in radians rotates the direction counterclockwise. In all cases excluding angular motion, the range of angles Algodoo uses goes from -π rad (-180°) exclusive to π rad (180°) inclusive, where of course the edges of the range point left, the negative X direction. If an angle outside of this range is entered, Algodoo will automatically correct it.

## Infinity & NaN

Floats can be used to represent both infinity and negative infinity. Infinity is represented as `+inf` whereas negative infinity is represented as `-inf`. They can also be represented as `∞` and `-∞` respectively. If the dividend is positive, float division by `0` will result in `+inf`, and division by `-0` will result in `-inf` (`-0` can only be achieved using `math.negate(0.0)`). This reverses if the dividend is negative. Every number will be greater than `-inf` and less than `+inf`.

`NaN` is a special case value that means Not a Number, and is produced in various ways:

- Adding or subtracting infinities that would cancel each other (`+inf - +inf`, `+inf + -inf`, `-inf + +inf`, `-inf - -inf`)
- Multiplying infinity by `0`
- Dividing `0` by itself (`0.0 / 0.0`)
- Dividing infinity by infinity
- Getting the root of negative numbers (e.g. `(-1) ^ 0.5`) except `-inf` which results in `+inf` - imaginary numbers are simply imaginary...
- Performing any arithmetic operation with `NaN` (except for `NaN ^ 0` which results in `1`)
- Inputting a number in a math function that's outside their domain (e.g. `math.acos(2)`)

Any equality and comparison operation with `NaN` will never match itself and always return `false`, or `true` if using `!=`.

# string

A string is a sequence of characters surrounded by quotation marks `""`, e.g. `"Hello World!"`. An example of a string property is `texture` - it refers to a filename.

```
texture =          "gold.png"
```

*You can also input Unicode characters, either copy them from anywhere that contains it or run their alt codes.*
*Alt codes are number sequences inputted while holding and then releasing the Alt key to output a character (including ones that don't appear in standard keyboards).*

*When running alt codes in Algodoo, note that it's using the character set in HTML UTF-8 as a decimal value: e.g. • bullets use 8226 (2022 in hexadecimal) instead of 7 or 0149.*
*See UTF-8 General Punctuation for the full list.*

## String escapes & verbatim string literals

Thyme uses the backslash character `\` as an escape character. The character after the escape character is treated as a special character (an **escape sequence**). Here are all the escape sequences that Thyme supports.

| Escape sequence | Description |
|---|---|
| `\\` | Backslash |
| `\"` | Double quote |
| `\n` | New line |
| `\t` | Tab |

For example:

```
> "Hello\nWorld!\nLook at this backslash:\t\\"
Hello
World!
Look at this backslash:    \
```

To treat the backslash character literally, you need to precede the string with `@`, e.g.

```
@"I can use single backslashes! \ Look at my backslashes! \"
```

```
> @"I can use single backslashes! \ Look at my backslashes! \"
I can use single backslashes! \ Look at my backslashes! \
```

Strings preceded with `@` are called **verbatim string literals**.

Verbatim string literals can also be used to store multi-line strings, e.g.:

```
@"(e)=>{
    print(e.dt);
}"
```

```
> @"(e)=>{
>     print(e.dt);
> }"
(e)=>{
    print(e.dt);
}
```

You can also use double quotes using two double quotes in the string.
When using a verbatim string literal, Algodoo will automatically convert it to a normal string, adding appropriate backslashes as needed and replacing newline characters with `\n`.

## Markup language & entity references

Displayed text in Algodoo, such as in text boxes or in the console, can be modified using a markup language that consists of elements.

An **element** consists of a start tag `<tagname>` and an end tag `</tagname>`. The content of the element is placed in between.

All elements must be placed inside a markup element, e.g.:

```
<markup>Content here</markup>
```

If done correctly, `<markup>` and `</markup>` should be hidden in the string when it is evaluated:

```
> print("markup")
markup
   4018 ms: Console "print" cmd: markup
> print("markup<markup>")
<markup>markup<markup>
  10821 ms: Console "print" cmd: <markup>markup<markup>
```

Algodoo's markup language defines special elements for formatting. As with <markup>, each of these elements has a start tag and an end tag.

`<b>` bolds the string.

```
> print("markup")
markup
  505439 ms: Console "print" cmd: markup
print("<markup><b>markup</b></markup>")
```

`<i>` italicises the string.

```
> print("markup")
markup
  572952 ms: Console "print" cmd: markup
print("<markup><i>markup</i></markup>")
```

`<u>` underlines the string.

```
> print("markup")
markup
  600763 ms: Console "print" cmd: markup
print("<markup><u>markup</u></markup>")
```

`<s>` strikes through the string.

```
> print("markup")
markup
  634195 ms: Console "print" cmd: markup
print("<markup><s>markup</s></markup>")
```

`<tt>` makes the text use a monospace font.

```
> print("markup")
markup
  747889 ms: Console "print" cmd: markup
print("<markup><tt>markup</tt></markup>")
```

`<small>` makes the text smaller. **Only on v2.2.0 and up.**

```
> print("markup")
markup
  30592 ms: Console "print" cmd: markup
print("<markup><small>markup</small></markup>")
```

`<big>` makes the text bigger.

```
> print("markup")
markup
  874580 ms: Console "print" cmd: markup
print("<markup><big>markup</big></markup>")
```

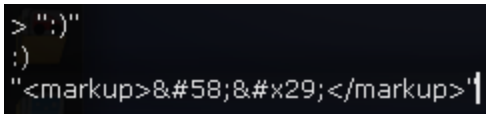
`<sub>` makes the text use subscript. **Only on v2.2.0 and up.**


`<sup>` makes the text use superscript. **Only on v2.2.0 and up.**


Using `<markup>`, you can also write entity references to render certain characters.


| Entity reference | Description |
|---|---|
| `&` | Ampersand `&` |
| `&apos;` | Apostrophe `'` |
| `>` | Greater than `>` |
| `<` | Less than `<` |
| `&quot;` | Double quote `"` |

Character references are also possible to render a specific Unicode character, this is done by `&#N;` or `&#xN;` for hexadecimal, replacing `N` with a number.




## The span element

`<span>` is a special element that doesn't do anything by itself, but can have attributes added to it. This can be useful to have different colors and fonts within a single text box, along with a couple of extra features not present within other elements.

`<span>` attributes come in the format of `<span attribute="value">`, where `attribute` is the name of the attribute, and `"value"` is the value of the attribute. The value must always be in quotations, even if it is a number. You can have multiple attributes per span tag, where each attribute is separated by a space like so:
`<span attribute1="value" attribute2="value">`
*Each attribute must be unique, you cannot have two of the same attribute in a single span tag.*

The following is a list of currently known attributes and their behavior:

`color` or `foreground` both modify the text color. It can take either an HTML color name (`"cyan"`) or a hexadecimal color code (`"#ff00ff"`). Some hexadecimal colors can be shortened, where `"#a3f"` is equivalent to `"#aa33ff"`.
This example uses `<span color="#ffff00">`


`background` sets a background color of the text. It takes the same values as `color`.
This example uses `<span background="#66f">`


`font` changes the text font. It takes a value with three optional arguments in this order: `"typeface style scale"`. `typeface` is the name of the font you want to use, such as Verdana, Arial, Georgia. `style` is if the text should be `normal`, `italic`, or `bold`. `scale` is how big the text should be as a percentage, with `100` being 100% scale.
Each argument is optional, so as long as they're still in the correct order then you can omit certain arguments. `"impact 75"` will set the text to be Impact at 75% scale. `"bold 250"` will make the text bold at 250% scale without changing the font. `"arial"` will only set the text to be Arial.
This example uses `<span font="georgia italic 150">`

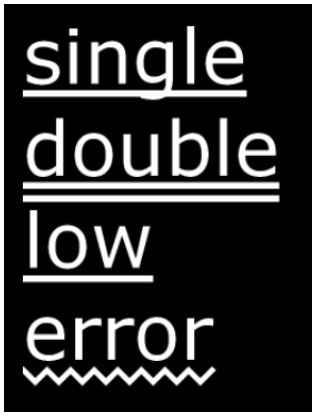

`size` or `font-size` both change the text size. It takes an integer, or either `"small"`, `"medium"`, or `"large"`. With an integer, you can set it from `0` up to `2097153` exclusive as long as the text can fit in the box or `textConstrained` is `false`.
A value of `"100000"` is normal size, and `"small"` is in fact actually slightly bigger than normal size. `"medium"` and `"large"` both are self-explanatory, being bigger than `"small"`.

This example uses both `<span size="small">` and `<span size="50000">`


`weight` or `font-weight` changes the text weight. It takes either an integer, `"light"`, `"bold"`, or `"ultrabold"`. Any integer less than or equal to 550 is equivalent to `"light"`, and any integer 551 and greater is equivalent to `"ultrabold"`. `"light"` is normal text, and despite the name, `"ultrabold"` is regular bold text.
This example shows each unique value. Values below 550 and above 551 are no different from 550 and 551 respectively.


`style` or `font-style` changes the text style. It takes either `"normal"`, `"oblique"`, or `"italic"`. `"normal"` is just normal text, and both `"oblique"` and `"italic"` are equivalent, creating italicized text.
This example shows each unique value.


`underline` adds an underline to the text. It takes either `"none"`, `"single"`, `"double"`, `"low"`, and `"error"`. `"none"` does nothing, `"single"` creates a single underline, `"double"` creates a double underline, `"low"` creates a single underline that will move downwards to be below the lowest stroke in the text, and `"error"` creates a zig-zag underline.
This example shows each unique value except `"none"`. `"low"` looks like `"single"` because no character extends downward in that line. If there were, for example, a "y" in the text, the line would move down to accommodate.




`underline_color` sets the color of the underline in text. It takes the same values as `color`.
To be visible, it requires either the `underline` attribute to be set, or for the text to use the `<u>` element.
This example uses `"red"` along with setting an error underline.


*Use of the span tag in the console can cause a rendering error that can make the console entirely blank, so it should only be used in text boxes. Errors in scripts that somehow involve the span tag can also cause the same rendering error in the console. If the console is blank, you will need to restart Algodoo.*

## Summary Table

| Type | Example | Description |
|---|---|---|
| `bool` | `true` | Short for boolean. Either true or false. |
| `float` | `3.51` | A (32-bit) floating-point number that stores any real number up to 6-9 significant figures of accuracy. |
| `int` | `7` | An (signed 32-bit) integer (whole number). |
| `string` | `"Hello!"` | A sequence of characters. |

# Lists

# Lists

A list is a collection of variables surrounded by square brackets `[]` separated by commas `,`. An example of a list property is `pos` (the position of a geometry) - the first value represents the geometry's horizontal (X) position and the second value represents its vertical (Y) position, so it makes sense that these two positions are grouped together as a single variable.

```
pos =                [-4.0, 2.0]
```

Another format that works for creating lists is using parentheses `()` instead of square brackets. However, doing so isn't recommended as it can reduce clarity regarding mathematical operations that also use parentheses to specify which operations to run first. Using parentheses also makes it impossible to create single element lists.

You can read the individual elements of a list by using parentheses, e.g. `pos(0)` would return the horizontal position (in this case `-4.0`) and `pos(1)` would return the vertical position (in this case `2.0`). You cannot write to individual elements of a list through this method.
You can also get a list of elements by using a list in the parentheses, e.g. `list([2, 3, 4])` returns a list containing the second, third, and fourth elements in order. You may have duplicates and change the order, e.g. `list([3, 1, 1])` returns a list containing the third, the first, and again the first elements in that order.
To make it easier to get a list of elements, you may use `..` to quickly create a range. So, `list(1..3)` is equivalent to `list([1, 2, 3])`.

The number you give when trying to access an element inside a list is called the **index**. Indices start at `0`, so if you want to access the first element, you must use the index `0`. If you give an invalid index, Algodoo will throw an error:

```
> exampleList = ["hello", 3, 12.5, false]
[hello, 3, 12.5, false]
> exampleList(0)
hello
> exampleList(3)
false
> exampleList(4)
  47925 ms: - WARNING - Failed to evaluate: exampleList(4), List index out of bounds
error
```

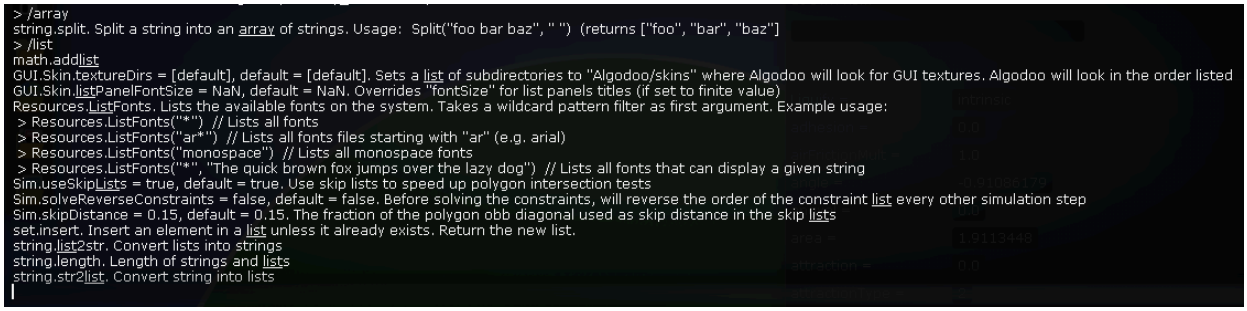
You can get multiple elements from a list by using a list of integers as an index:

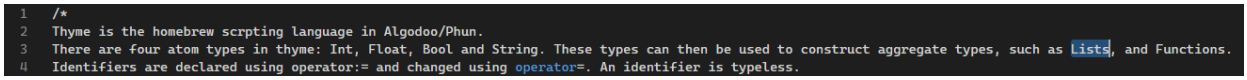
```
> exampleList := ["Hello", "World", "This", "Is", "An", "Example", "List"]
[Hello, World, This, Is, An, Example, List]
> exampleList([0, 2, 3, 4, 5])
[Hello, This, Is, An, Example]
```

You can get consecutive elements from a list by using the range operator `..`:

```
> exampleList := ["Hello", "World", "This", "Is", "An", "Example", "List"]
[Hello, World, This, Is, An, Example, List]
> exampleList(5..6)
[Example, List]
```

*Lists in Thyme are sometimes referred to as arrays - I chose to refer to them as lists as they are more often called that internally and the thyme.cfg file refers to them as such:*



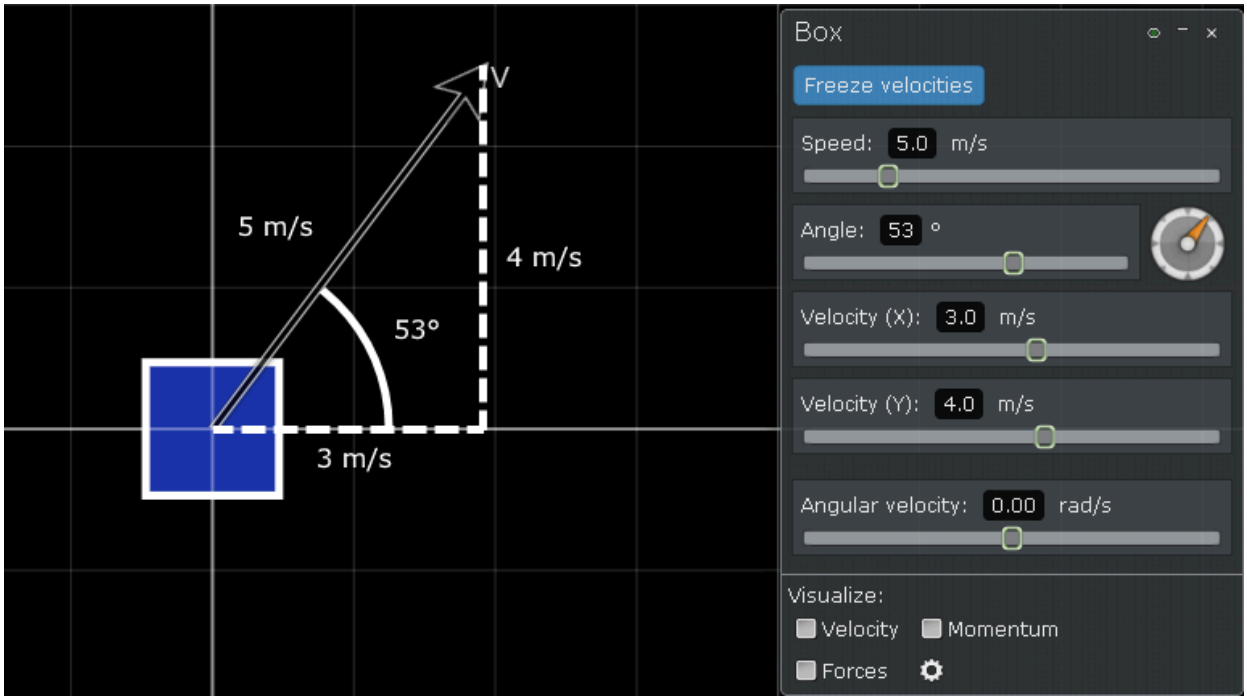




*In many programming languages (not including Thyme), lists and arrays are two separate collections - the difference is usually that a list's size can be changed without redeclaring the entire collection while arrays must be redeclared.*

# Vectors

A vector is a quantity with magnitude and direction (as opposed to a scalar, which only has magnitude), represented in Algodoo by a list of integers or floats. The number of elements in the list represents the number of dimensions the vector has. The object properties `pos` and `vel` are examples of vectors.

In a diagram, the length of a vector represents its magnitude and the arrowhead points in the vector's direction. This velocity would be represented in Thyme as `[3, 4]`.




The magnitude can be worked out using the built-in function `math.vec.len` (which uses Pythagoras' theorem) and the direction can be worked out using basic trigonometry.

*The magnitude of an object's velocity is its **speed**. Speed is a scalar, so has no direction and can never be negative.*

Algodoo allows you to visualise an object's velocity, momentum, and the forces acting on the object.

## Colors

A color is represented in Algodoo by a list of four ints or floats.

| Element | HSL | HSV | RGB |
|---|---|---|---|
| `color(0)` | Hue | | Red |
| `color(1)` | Saturation | | Green |
| `color(2)` | Lightness | Value | Blue |
| `color(3)` | Alpha (opacity) | | |

Each element in the list ranges from 0 to 1, except Hue, which ranges from 0 to 360 exclusive.

For example, the RGB color `[1.0, 0.5, 0.0, 1.0]` represents orange.

# Variables

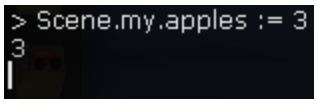
# Variables

A variable is a name given to a stored data value that can change.

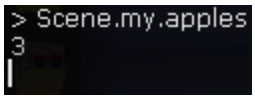
Variables are declared as follows:

```
variableName := initialValue
```

You can try declaring a variable in the console:

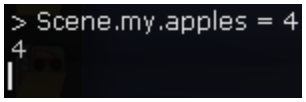



You can access the variable's value by stating its name:




You can change the variable's value as follows:

```
variableName = newValue
```




`:=` cannot change a variable's value that has already been declared**,** you may prefer to use this if you're working with multiple variables of the same name (such as creating objects with specific properties).

When you declare variables**,** you should **always give them meaningful names**. Only ever use a single letter as a variable name if the variable's purpose is abundantly clear.

*A constant is a name given to a stored data value that cannot change. Examples of constants include* `math.pi` *and* `math.e`*. There is no way to declare your own constants in Thyme.*

*Thyme is a* ***dynamically-typed language****, meaning that the data type of a variable can change. Some programming languages,* ***statically-typed languages****, do not allow this.*

# Variable scope

The variables that were shown have **scene** scope (depicted as variable names starting with `Scene.my.`), meaning they can only be accessed in the current scene.

Variables cannot be accessed from outside their scopes. Here is a table of all four scopes:

| Scope | Example | Description |
|---|---|---|
| **Block** | `{`<br>`  i := 0;`<br>`}` | The variable exists inside a block (a group of curly brackets `{}`).<br>It is deleted when the closing curly bracket `}` appears and cannot be accessed afterwards. |
| **Object** | `textureMatrix` | The variable exists inside an object and can be seen in the object's script menu.<br><br>**Object-scoped variables that you create are deleted when the scene is reloaded if they do not start with underscores.** |
| **Scene** | `Scene.my.apples` | The variable exists inside the `Scene.my` object. Every variable placed inside the Scene.my object is saved with the scene.<br>A scene variable from one scene cannot be accessed from a different scene.<br><br>*These variables do not need to start with underscores unlike object variables which is a common misconception.*<br>*There is also the* `Scene.temp` *object where instead variables inside it are* ***not*** *saved with the scene.* |
| **Global** | `Sim.running` | The variable exists in the console and can be accessed from anywhere.<br><br>**Avoid making custom global variables as it may conflict with variables in another scope or scripts that utilize the same variable name.**<br>**Global variables that you create will be deleted when Algodoo is reset.** |

Variables declared in the console will have global scope (unless they are declared inside an object or `Scene.my`), while variables declared in the object console will have object scope (unless they are declared inside `Scene.my`).

This shows several variable declarations with different variable scopes within an event function:

```
(e)=>{
    localVar := 3; // block scope
    eval("localVar2 := 4"); // block scope
    e.this.objectVar := 5; // object scope
    Scene.my.sceneVar := 6; // scene scope
    geval("globalVar := 7"); // global scope
}
```

Event functions, `eval`, and `geval` will be explained in later sections. You don't need to learn about them yet.

# Functions

# Functions

A function is a block of code used to perform a certain action.

Here is an example of a function:

```
add := (n1, n2)=>{
      n1 + n2;
};
```

As you can likely tell, this function adds two numbers together.
Here is an example of this function being **called** (used):

```
add(3, 4);
```

Breaking this into smaller parts:

- `n1` and `n2` are **parameters**. Parameters are placed in brackets and are separated by commas. When you use the function, you give Algodoo values (called **arguments**) for each parameter in the function. In this case, `3` is being passed into `n1` and `4` is being passed into `n2`.
- Everything inside the curly brackets `{}` is the **body** of the function - the action that the function performs. In this case, `n1 + n2` is the only operation being performed - the function takes the two given numbers and adds them together.
- `n1 + n2` is also the **return value**. In Thyme (like in Lisp), the last evaluated expression in the function is returned.

When we enter `add(3, 4)`, we give the function `3` as `n1` and `4` as `n2`. The function then adds `3` and `4` together and returns the result, `7`:

```
> add = (n1, n2)=>{ n1 + n2; };
(n1, n2)=>{n1 + n2}
> add(3, 4);
7
```

A function with N parameters is declared as follows:

```
functionName := (parameter1, parameter2, ..., parameterN)=>{

};
```

The function can have as many parameters as you need.
A function with N parameters is called as follows:

```
functionName(argument1, argument2, ..., argumentN);
```

Like any other variable, functions can have any of the four scopes mentioned before. For example, they can be placed in the Scene.my container:

```
> Scene.my.add = (n1, n2)=>{ n1 + n2; };
(n1, n2)=>{n1 + n2}
> Scene.my.add(5, 1);
6
```

Functions can be declared without parameters. You declare a parameter-less function as follows:

```
functionName := {};
```

Parameter-less functions can be called in two ways:

```
functionName;
functionName();
```

*You can also declare a parameter-less function like a normal function, but leaving the parameter list empty (i.e. `()=>{}`). This still works, but Algodoo will delete the parameter list and turn the function into the above format when you input it.*

# Event functions

You may have noticed by now that each object has several functions by default. Each of these functions is called when a specific event happens. For example, if one object touches another, each object will run its own collision function (`onCollide`) which you can customize with a script.

Here is a list of all the events in Algodoo:

| Event | Description |
|---|---|
| `onClick` | Called when the object is clicked. |
| `onCollide` | Called when the object collides with another object or with water. |
| `onDie` | Called when the object is destroyed. |

| | |
|---|---|
| `onHitByLaser` | Called every frame while the object is being hit by a laser. |
| `onKey` | Called when a key is pressed or released. |
| `onLaserHit` | Called every frame for every object the laser is currently hitting. |
| `onSpawn` | Called when the object is created. Objects are re-created when the scene is loaded or refreshed (e.g. when undone or redone). |
| `postStep` | Called after every physics step, a.k.a. tick (by default 60 times per second only while the scene is running) |
| `update` | Called every frame (by default 60 times per second). |

## Event arguments

By default, each event function is simply this:

```
(e)=>{}
```

They all have a single parameter called `e`. It's short for **event arguments** and is a `ClassObject` containing different properties about the event that has just been called. Every time an event function is called, Algodoo creates an object containing the event properties and calls the relevant event function, passing in the event arguments as `e`.

### e.this

All event arguments contain `e.this` It refers to the object that the event function belongs to. For example, the following code in the postStep event would make the object cycle through all the colors of the rainbow:

```
(e)=>{
    e.this.colorHSVA = e.this.colorHSVA + [1, 0, 0, 0]
}
```

`e.this` isn't actually needed to access the object's variables unless you're declaring a new variable within the object (e.g. `e.this._marble := true`) or you're passing `e` as an argument to a function outside the current object.

## Event properties table

The following is a table of all the properties passed into each event, excluding `e.this`. `onDie` and `onSpawn` have no other properties.

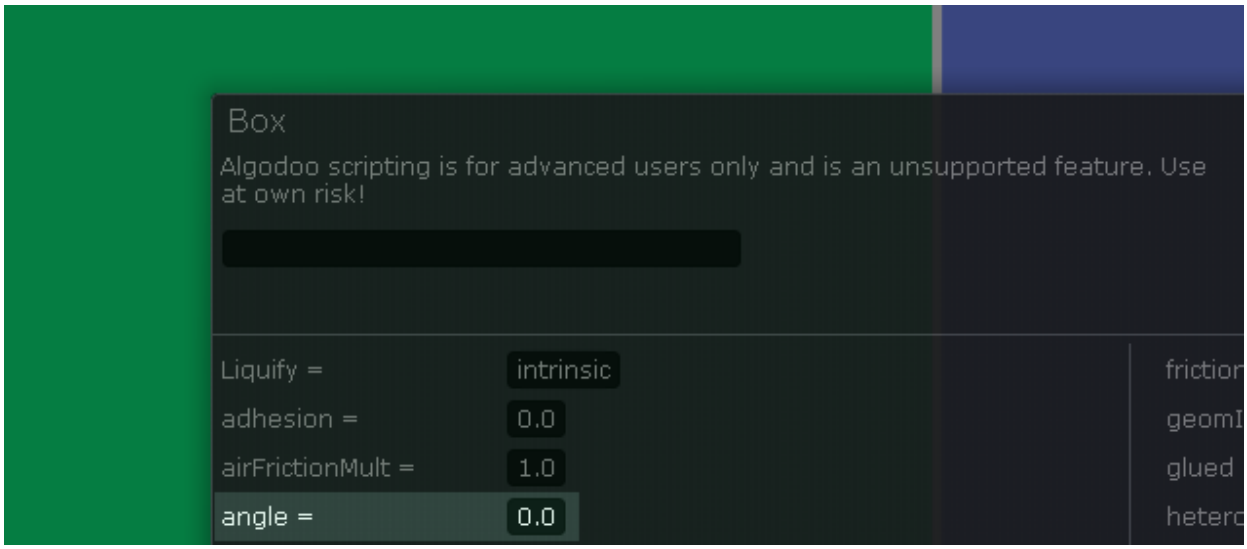
| Event | Property | Property Type | Property Description |
|---|---|---|---|
| `onClick` | `clickCount` | Int | How many clicks have happened in quick succession. |
| | `pos` | Float List: [x, y] position | The position of the cursor when it clicked. |
| | `handled` | Bool | Unknown. Always is false. |
| `onCollide` | `normal` | Float List: [x, y] vector | The normal of the collision (a normalised vector pointing from the current object to the other object). |
| | `other` | ClassObject: Entity | The other object that the current object has collided with. This does not exist if the collision was with water. |
| | `pos` | Float List: [x, y] position | The position where the collision happened. |
| | `soundGain` | Float | Scales with the impulse of the collision, the camera's zoom, and goes to zero if offscreen.<br><br>*Sound was never officially implemented in Algodoo, but it was planned that the higher this number was, the louder the collision sound played would be.* |
| `onHitByLaser` and `onLaserHit` | `geom` | ClassObject: Entity | The object that the laser is hitting. |
| | `hsv` | Float List: [h, s, v] color | The HSV color of the laser when it hits the object (not necessarily the set color of the laser pointer). **Only on v2.2.0 b4 and up.** |
| | `laser` | ClassObject: | The laser that is hitting the object. |

| | | | |
|---|---|---|---|
| | | Entity | |
| | `normal` | Float List: [x, y] vector | The normal of the laser hit (a normalised vector pointing from the object to the laser). |
| | `pos` | Float List: [x, y] position | The position of the laser hit. |
| `onKey` | `keyChar` | String: Character | The text symbol created by the key. If no symbol exists, (e.g. the arrow keys) this variable does not exist either. |
| | `keyCode` | String: Key code | The name of the key being held.<br><br>*Note that onKey is called when a key is pressed or released. If you only want code to run while a key is being pressed, use the `keys.isDown` function.* |
| | `pressed` | Bool | True if the key is pressed down, false if the key was released. |
| | `handled` | Bool | Unknown. Always is false. |
| `postStep` and `update` | `dt` | Float | Short for 'delta time'. Equal to `1.0 / Sim.frequency` for postStep, amount of time since last frame for update. |

## Object properties as functions

Object properties can be turned into parameterless functions, as long as the function returns the correct data type. This can be used to keep an object property's value constant or have it depend on another variable. One caveat though is that you cannot copy the function to another property. This is due to the fact that they're parameterless functions and are called by name, so attempting to copy just calls it and returns the value of the function, effectively getting the value at that point.

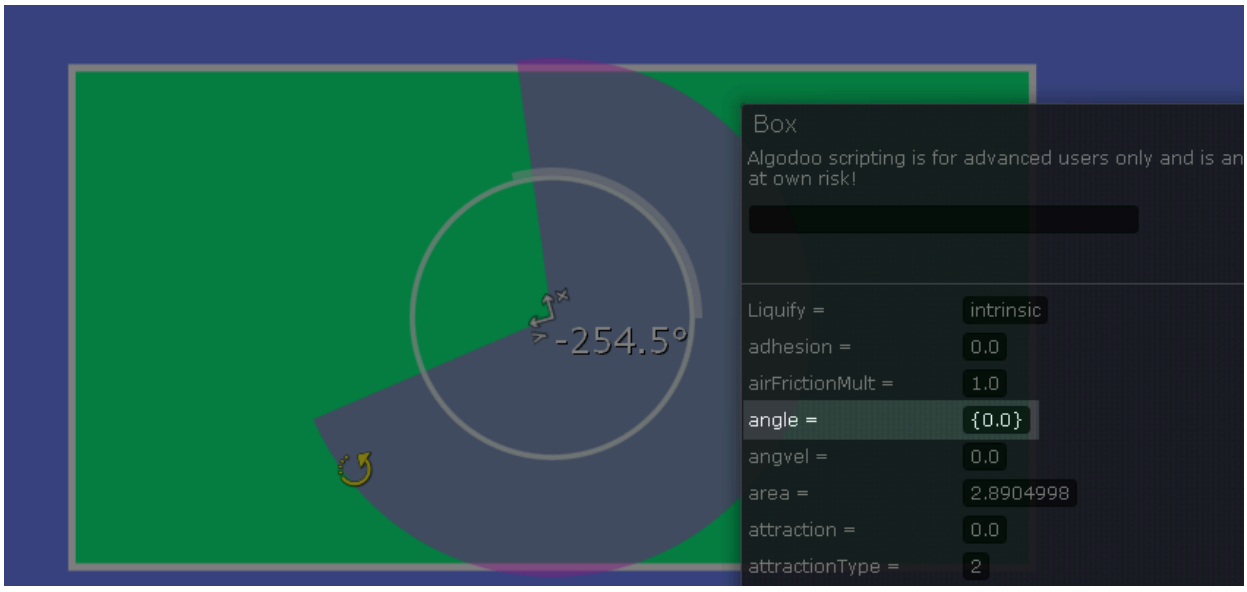
For example, the value of `angle` is usually a float:




If we change the value of `angle` to the following function:

```
{
	0.0;
}
```

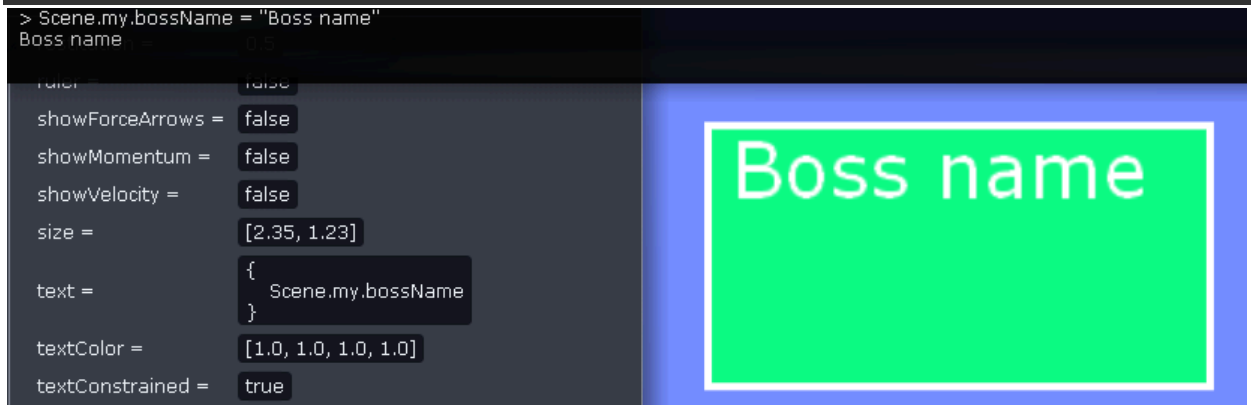
...the value of `angle` is kept constant at zero.




When you change the value of an object's property to a function, the function is executed every frame (even when the scene isn't running).

For example, if you wanted a text box to display the value of `Scene.my.bossName`, you would make the value of `text` the following:

```
{
	Scene.my.bossName;
}
```




As these are functions, you can have code run before the returned value to have more control over the set value. Here's an example where `colorHSVA` is set to this function:

```
{
	bg := math.RGB2HSV(App.background.skyColor);
	[bg(0), 1.0, 1.0, 1.0];
}
```

In this above example, we are ultimately forcing an object to always have the same hue as the sky, but maintain 100% saturation, value, and alpha. This is written in this way as sky color is only stored as an RGB color, but in order to preserve hue we need to convert it to HSV. The last line of this function is what is returned, so that is what `colorHSVA` is set to.

One more note, using this method to programmatically set properties to values has the least performance impact compared to any other method. If you think you can convert a `postStep` script into one of these sorts of functions, then do so.

*Again, make sure that the function returns the correct data type. For `text`, the value returned must be a string. If you want to display the boss' health (an integer), for example, you would need to convert it to a string (either by using `math.toString` - `math.toString(Scene.my.bossHealth)` or concatenating it with an empty string - `"" + Scene.my.bossHealth`).*

# Logic structures

# Logic structures

Using everything covered so far, it's important to learn what you can do with the data types using the three basic logic structures in programming: **sequence, selection,** and **recursion**.

- Sequence - Learn the order of how Thyme scripts are run.
- Selection - Learn how to run scripts under certain conditions.
- Iteration - Efficiently run scripts multiple times!

# Sequence

# Sequence

Sequence is one of the three basic logic structures in programming. In a sequence structure, each action happens one after another. In Thyme, a semicolon `;` is used to separate actions. For example:

```
Scene.my.apples := 3;
Scene.my.apples = Scene.my.apples * 2;
print(Scene.my.apples);
```

The above script declares a scene variable, `Scene.my.apples`, with an initial value of `3`. It then sets the value of `Scene.my.apples` to twice its original value, and finally prints its value to the console.

*Almost all programming languages that use semicolons in this way require you to end each statement with a semicolon. In Thyme, putting a semicolon on the last line is not needed (and Algodoo will delete it when you enter the script), but is a good practice to get into nevertheless as it lowers your chances of forgetting to include one.*

Some terminology:

An **expression** is a combination of values and/or functions to create a new value. Expressions include:

```
3; // returns 3
3 + 4; // returns 7
App.background.skyColor; // returns the RGB color of the sky
```

A **statement** is a standalone instruction. Statements include:

```
Scene.my.apples := 3; // declares Scene.my.apples with initial value
3
postStep(e); // calls the postStep function, passing in e as an
argument
print("Hi!"); // calls the print function, printing "Hi" to the
console
```

# Update order

Following in line with sequence, it is possible to expand that notion to beyond a simple script. When Algodoo is processing each tick, it runs all scripts in a specific order that can be controlled. Every object in the scene will run its code at the same time relative to every other object. To put it simply, object B will always run its code after object A, and in some cases this can be reversed if desired.

Update order is a useful thing to keep track of as it can make communication between two otherwise separate objects more seamless. A simple example is if you have two objects A and B. Object B must always be positioned 5 meters to the left of object A, so you could create a script so that Object A updates a scene variable with its current position that Object B then reads. However, in order for this to work properly, Object B must read that variable *after* it's written to, else it will lag behind its intended position. This is because if Object B is running its code first, the variable won't have updated to Object A's new position yet, so it instead reads the position Object A had in the previous tick.

In Algodoo, update order for the postStep, update, and onSpawn functions is determined by the object's zDepth. Objects that have a lower zDepth, in other words are visibly behind other objects, will have their code run first, and objects with a higher zDepth, in other words are visibly in front, will have their code run last.

Along with zDepth, what layer an object is on also controls update order. Every object on the top layer will run its script first, then on the first layer down, then the second layer, and so on. In essence, this is backwards from how zDepth is as in this case the objects on the higher layers, which appear in front, are first instead of being last.

Additionally, update order for onCollide functions is partially determined by the objects' body. When two objects collide, the onCollide code that runs first is the code for the object with the lower body value. Objects glued to the background always have a body of 0, and objects with infinite density are treated as having a body of 0 as well regardless of their actual body. However, it is currently unknown how the order of multiple collisions in a scene are processed.

To put it simply, in a single layer the objects that run their code first are the ones furthest behind other objects, and for objects of different layers it's the objects on the top layers that run first.

*At the moment, it is unknown how the call order of other functions is determined.*

One last note: When calling functions, Algodoo batches all alike functions to be called together. This means that all postStep functions run together, all onCollide functions run together, all update functions run together, and so on. So, even if an object is set to update first, its update function will only be called after every other postStep function in the scene is called.

# Selection

# Selection

Selection is one of the three basic logic structures in programming. In a selection structure, a question is asked. Depending on the answer, the program takes a specific course of action.

## If statements

The if statement is the most popular selection structure. There are several ways to form an if statement in Thyme.

### Boolean expressions

A **boolean expression** is an expression (a combination of values and/or functions) that always returns a boolean value (`true` or `false`). The question (or condition) that is asked in an if statement must be a boolean expression.

### if function

The if function is used as follows:

```
if(condition, {
     // code if condition is true
});
```

For example, if you wanted to change the background color if we had more than 100 apples, you could use this code:

```
if(Scene.my.apples > 100, {
     App.background.skyColor = [0.5, 0.9, 0.5, 1.0];
});
```

While the if function is the simplest to understand, it is also the least useful method of creating an if statement. It lacks the ability to run code for when the condition is false (an *else* statement). Additionally, due to its implementation, the if function is significantly worse for performance than the following two methods. Due to these performance concerns, it is highly recommended you use the following methods at all times.

### if_then_else function

The if_then_else function is called as follows:

```
if_then_else(condition, {
	// code if condition is true
}, {
	// code if condition is false
});
```

Given the performance concerns of the regular if statement, you may choose to use this method instead. Unlike the simple if, this is considered an intrinsic function which means that its code utilizes the much faster C++ Algodoo is built on instead of the much slower Thyme script.

If you do not wish to have an else statement, you can leave the function empty. As long as there is a set of braces, like so: `if_then_else(condition, { // code }, {});`, then it will work properly.

### Ternary operator

The ternary operator `?:`, sometimes referred to as the conditional operator, returns a different value based on the given condition. It is used as follows:

```
condition ? returnThisIfTrue : returnThisIfFalse;
```

For example, if you want to reset the number of apples to `0` every time it went over `100`, you could use the following script in postStep:

```
(e)=>{
	Scene.my.apples = Scene.my.apples > 100 ? 0 : Scene.my.apples;
}
```

You can use the ternary operator with parameter-less functions to form a ternary operator-based if statement:

```
condition ? {
	// code if condition is true
} : {
	// code if condition is false
};
```

Like with `if_then_else`, you may leave code blocks empty if you don't wish to use them. The ternary operator at the moment is the most popular method for creating if statements. Whether you use the ternary or `if_then_else` is up to personal preference.

*All if statements can return values as they are all functions. Specifically, the accepted code blocks in the if functions are in fact parameterless functions, which leads to them returning values.*

*If you're curious, the reason the simple if function is so slow is because it piggybacks off of the ternary operator. The underlying script in the if function simply takes the condition and function passed into it and applies them to a ternary, essentially making the if function a middleman. Doing this is slow as it forces an extra step into the whole process. Now, the ternary technically itself piggybacks off of if_then_else, but the ternary is considered an infix operation, which means it functions as shorthand rather than passing from one place to another. There is an extremely negligible performance difference between infix operators and their underlying functions, as infix is a far more direct connection than simply a function calling another function. This means for practical purposes, there is no difference between using the ternary and if_then_else.*

# Iteration

# Iteration

Iteration is one of the three basic logic structures in programming. In an iteration structure, a block of code is repeated a specified number of times or until a condition is met.

## Built-in for function

A for loop repeats a block of code a specified number of times. The built-in for function is used as follows:

```
for(numberOfLoops, (i)=>{
      // code to run
});
```

The for loop works as follows:

1. It first makes a check whether `numberOfLoops` is `0` or below. If false, continue.
2. It then runs the `for()` function again where `numberOfLoops` is decremented by `1`, this repeats until `numberOfLoops` is `0`.
3. It then executes the function you've passed in (i.e. `(i)=>{ // code to run }`) for `numberOfLoops` starting with the function where `i` is declared as `0`.

*You can name `i` whatever you want. Giving `i`, like any other parameter, a meaningful name will improve your code's readability.*

For example, if you wanted to print the numbers 0 to 4, you could use this code:

```
for(5, (i)=>{
      print(i);
});
```

```
> for(5, (i)=>{ print(i); });
0
10512727 ms: Console "print" cmd: 0
1
10512727 ms: Console "print" cmd: 1
2
10512727 ms: Console "print" cmd: 2
3
10512728 ms: Console "print" cmd: 3
4
10512728 ms: Console "print" cmd: 4
```

The function does have some limitations:

- You can't change how the value of the iterator variable is first declared.
- You can't control how the value of the iterator variable is declared through iterations.
- You can only accurately run at most 66 iterations.
- You have to specify how many loops you want to run.

## Kilinich's xFor function

Kilinich created the `xFor` function which lets you set how the iterator variable is first declared.
http://www.algodoo.com/forum/viewtopic.php?f=13&t=5146&p=54364&hilit=xfor#p54364
(**Make sure HTTPS-Only mode is disabled on your browser when visiting this site**.)

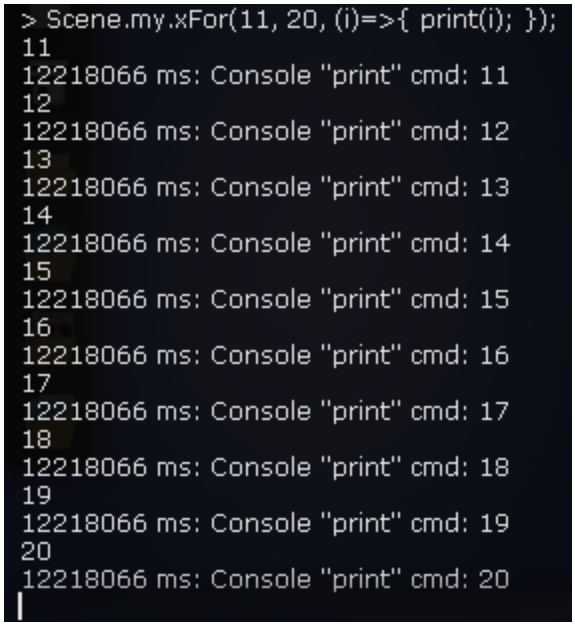
This function also serves to solve an issue known as the recursion limit, which is a limit imposed by Algodoo on how many times you can nest functions.
A regular for loop in Algodoo can only achieve around 65 iterations, it loses accuracy beyond this limit or fails entirely in some cases, whereas this function can achieve many more (which does so by utilizing a binary recursion tree).

```
Scene.my.xFor = (n1, n2, code) => {
    n2 > n1 ? {
        m := (n1 + n2) / 2;
        Scene.my.xFor(n1, m, code);
        Scene.my.xFor(m + 1, n2, code)
    } : {code(n1)}
};
```

If you want to use the `xFor` function in your own code, copy and paste the above code into the console. You do not need to understand how this function works.

Using xFor to print the numbers from 11 to 20:

```
Scene.my.xFor(11, 20, (i)=>{ print(i); });
```




One important thing to understand, the behavior of the upper limit of the for loop is different from the built-in for loop. `for(10, (i)=>{})` is equivalent to `Scene.my.xFor(0, 9, (i)=>{})`. The upper bound of `for` is exclusive, whereas the upper bound of `xFor` is inclusive. If you're used to `for`, be wary of the difference to avoid running into bugs involving this difference in behavior.

## The Real Thing's xWhile function

A while loop repeats a block of code while a given condition is true.

The Real Thing created the `xWhile` function, which functions as a while loop.
http://www.algodoo.com/forum/viewtopic.php?f=13&t=12135&p=86262&hilit=while&sid=9f50118cb532b2ebff1ae87d3794fbc7#
(**Make sure HTTPS-Only mode is disabled on your browser when visiting this site**.)

It is declared like this:

```
Scene.my.xWhile = (conditionFunc, mainFunc)=>{
    unfinished := math.toBool(conditionFunc);
    iteration := {
        unfinished ? {
            mainFunc;
            unfinished = math.toBool(conditionFunc)
        } : {}
    };
```

```
    str := "iteration; iteration; iteration; iteration; iteration; iteration;
iteration; iteration";
    exec := {
        unfinished ? {
            eval(str);
            str = str + "; " + str;
            exec
        } : {}
    };
    exec;
    // According to Steve Noonan, these two lines are to fix memory leaks when
using this function.
    iteration = "";
    exec = "";
};
```

`conditionFunc` must be a function that returns a boolean expression.

As with Kilinich's `xFor` loop, copy the contents of the above code block into the console if you want to use this in a scene. You do not have to understand how it works.

Using `xWhile` to increase an integer to the value 10:

```
Scene.my.xWhile({
    integer < 10;
}, {
    integer = integer + 1;
});
```

```
> integer = 0
0
> Scene.my.xWhile({integer < 10}, {integer = integer + 1});
undefined
> integer
10
```

## Recursion

As you can't get while, repeat…until and proper for loops without importing others' code or coding them yourself, it may be better to drop iteration entirely in favour of recursion.

A recursive function is a function that calls itself.

An example of a recursive function is one that checks if a given string is a palindrome:

```
Scene.my.isPalindrome = (word)=>{
    wordLength := string.length(word);
    if_then_else(wordLength <= 1, {
        true;
    }, {
        wordArray := string.str2list(word);
        if_then_else(wordArray(0) == wordArray(wordLength - 1), {
            Scene.my.isPalindrome(string.list2str(wordArray(1 .. wordLength -
2)));
        }, {
            false;
        });
    });
};
```

This code may look complicated at first glance. Let's think about how to solve the problem:

A string is a palindrome if:

- The first character and the last character are the same.
- The rest of the string is a palindrome.

The rest of the string is a palindrome if:

- The first character and the last character are the same.
- The rest of the string is a palindrome.

...and so on, until the rest of the string consists of one character (in which case the first character and the last character are obviously the same) or no characters at all.

Let's apply this thinking to the string `"MADAM"`.
`"MADAM"` is a palindrome if:

- **The first character and the last character are the same.**

The first character is `"M"` and the last character is also `"M"`. So far so good.

- **The rest of the string is a palindrome.**

The rest of the string, `"ADA"`, is a palindrome if:

- **The first character and the last character are the same.**

The first character is `"A"` and the last character is also `"A"`. Let's keep going.

- **The rest of the string is a palindrome.**

The rest of the string, `"D"`, is only one character long. Therefore, the string `"MADAM"` must be a palindrome.

Here is the same code with comments:

```
Scene.my.isPalindrome = (word)=>{
    wordLength := string.length(word);
    if_then_else(wordLength <= 1, {
        true; // if the string length is 0 or 1, return true
    }, {
        wordArray := string.str2list(word);
        if_then_else(wordArray(0) == wordArray(wordLength - 1), { // is the
first character the same as the last character?
            Scene.my.isPalindrome(string.list2str(wordArray(1 .. wordLength -
2))); // cuts the first and last character off the string and sends the rest of
the string back into the function
        }, {
            false;
        });
    });
};
```

*Kilinich's xFor loop was written using recursion.*

## Higher order functions

A higher-order function is a function that takes a function as an argument and/or returns a function. For example:

```
doTwice = (function, operand)=>{
      function(function(operand));
};

addThree = (number)=>{
      number + 3;
};

print(doTwice(addThree, 3));
```

The above code outputs 9 (3 added to 3 twice).

```
> doTwice = (function, operand)=>{
>     function(function(operand));
> };
>
> addThree = (number)=>{
>     number + 3;
> };
>
> print(doTwice(addThree, 3));
>
9
  7766 ms: Console "print" cmd: 9
```

## Filter

An example of a higher order function is **filter**. Filter takes a list and a condition, and returns a list of the elements that meet the condition. The condition itself is a function that takes a value and returns a boolean (also known as a **predicate**).

Filter can be implemented as follows:

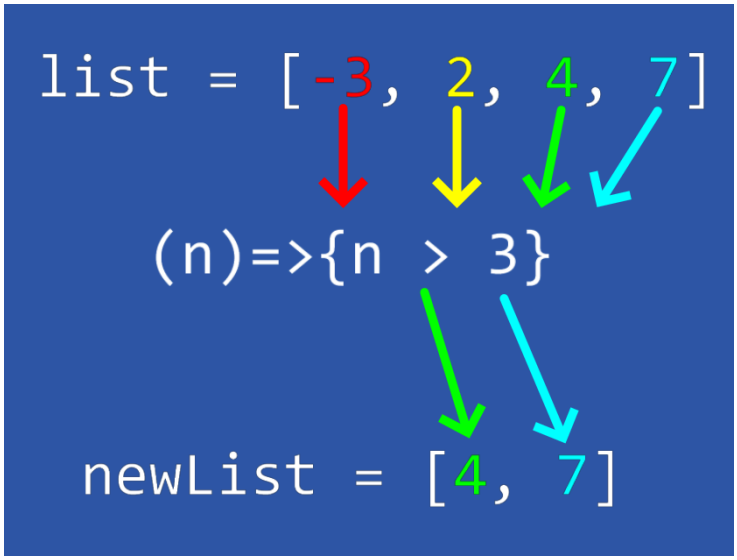
```
Scene.my.filter := (list, predicate) => {
    newList := [];
    for(string.length(list), (i)=>{
        if_then_else(predicate(list(i)), {
            newList = newList ++ list[i];
        }, {});
    });
    newList;
};
```

The above function creates a new list. It then goes through each element of the list, checking whether it meets the condition. If the element meets the condition, it's added to the new list. Finally, the new list is returned.

You can use filter to return numbers in a certain range. For example, all the numbers greater than 3:

```
> Scene.my.filter([-3, 2, 4, 7], (n)=>{n > 3})
[4, 7]
```

An illustration of what the above function call does:




## Map

Another higher order function is **map**. Map takes a list and a function, applies the function to all the elements of the list, and returns the new list.

The function must take one parameter and return a value.

Map can be implemented as follows:

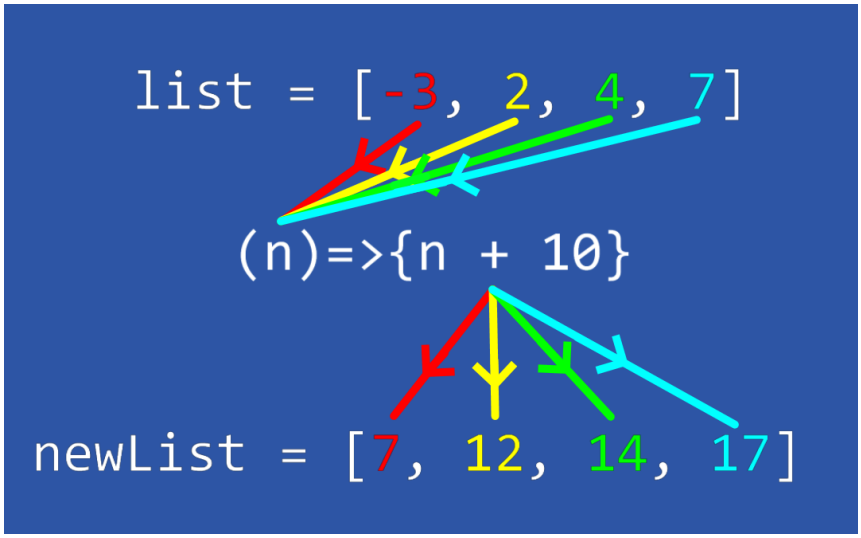
```
Scene.my.map := (list, func) => {
    newList := [];
    for(string.length(list), (i) => {
        newList = newList ++ [func(list(i))];
    });
    newList;
};
```

The above function creates a new list. It then goes through each element of the list, applies the function to it, and adds it to the list. Finally, the new list is returned.

You can use map to add 10 to every number in a list, for example:

```
> Scene.my.map([-3, 2, 4, 7], (n)=>{n + 10})
[7, 12, 14, 17]
```

An illustration of what the above function call does:




## Fold

Another common higher order function is **fold**. Fold takes an initial total, a list and a function. It runs the function over each element on the list, adding the result to the total. The total is then returned.

The function must take two parameters (the total and the element) and return the new total.

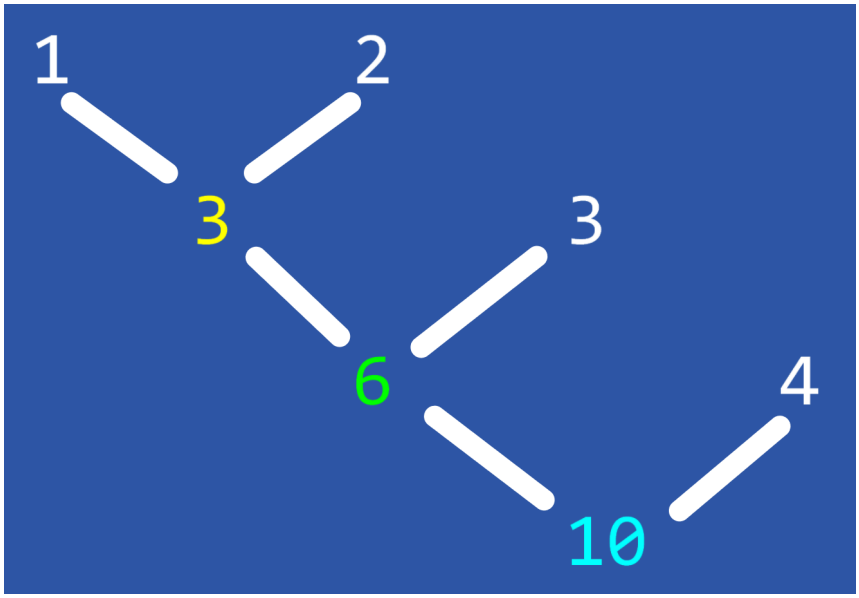
Fold can be implemented as follows:

```
Scene.my.fold := (initialValue, list, func) => {
    accumulator := initialValue;
    for(string.length(list), (i)=>{
        accumulator = func(accumulator, list(i))
    });
    accumulator
};
```

Fold can be used to add numbers in a list together, for example:

```
> Scene.my.fold(0, [1, 2, 3, 4], (total, n)=>{total + n})
10
```

An illustration of what the above function call does:




## Custom functions

The following document contains several helpful functions (including xFor, xWhile, filter, map and fold, mentioned earlier).

Thyme Additional Functions

# Objects

# Objects

A `ClassObject` (or just object) is a variable with labelled properties (in contrast with a list, where each element isn't named and is instead referred to by index). For example, a circle has named properties such as its radius and its area.

An empty `ClassObject` is declared by setting a variable's value to `alloc`:

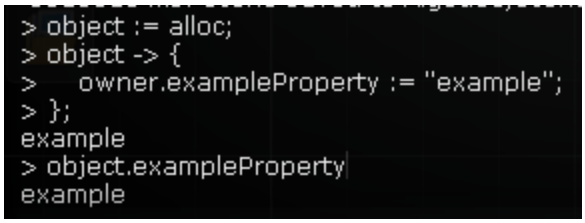
```
object := alloc;
```

You can then give a `ClassObject` properties in two ways:

```
// 1
object.exampleProperty := "example";

// 2
object -> {
    owner.exampleProperty := "example";
};
```

The above code creates a `ClassObject` named `object` with a single property, `exampleProperty`. The value of `exampleProperty` is `"example"`.

The connection between a `ClassObject` and any one of its properties is indicated by a dot (`.`) written between them:




`ClassObject`s can store any data type as a property, including functions. A function inside a `ClassObject` is often called a **method**. To access the parent object's properties within a method, you use the `owner` keyword:

```
object := alloc;
object -> {
    owner.exampleProperty := 1;
```

```
    owner.addOne = {
        owner.exampleProperty = owner.exampleProperty + 1;
    };
};
```

*The `owner` keyword is equivalent to the `this` keyword or the `self` parameter in other languages.*

In the above example, `addOne` adds one to the value of `exampleProperty`.

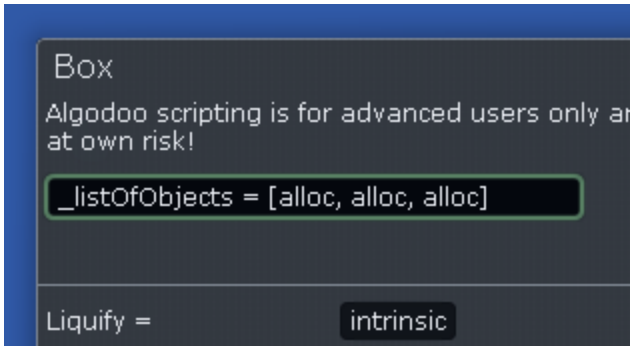
`ClassObject`s are volatile, meaning that they disappear whenever the scene is undone or reloaded. To counteract that, you will need to declare the object with its initial values in an `onSpawn` script.

```
onSpawn = (e)=>{
   object := alloc;
   object -> {
      owner.exampleProperty := 1;
      owner.addOne = {
         owner.exampleProperty = owner.exampleProperty + 1
      }
   }
}
```

*In a traditional object-oriented language, classes and objects are different. A class is effectively a blueprint for a related group of objects - it defines what properties and methods each object of the group should have. An object is a specific instance of a class. For example, all circles have a radius property, so we would have a circle class with a radius property. A big marble and a small marble would both be instances of the circle class - both marbles have a radius property, but the values of their radii may be different.*

**NOTE**: As `ClassObject`s can themselves store `ClassObject`s, and due to how `ClassObject`s are referential in nature, it is possible for a `ClassObject` to store itself. This is known as a self-reference or cyclic reference, and this is **extremely unstable**. Attempting to perform most actions while a self-referential object exists **will crash Algodoo**. As references can be chained, you can end up with a situation such as A -> B -> C -> A or a circular reference. Ultimately, through that chain A references itself, which does cause this unstable state even if the self-reference isn't within A itself.

**NOTE**: Setting an object variable to be a list of empty `ClassObject`s (e.g. `[alloc, alloc, alloc]`) **will crash Algodoo** if the object console with the list is open.



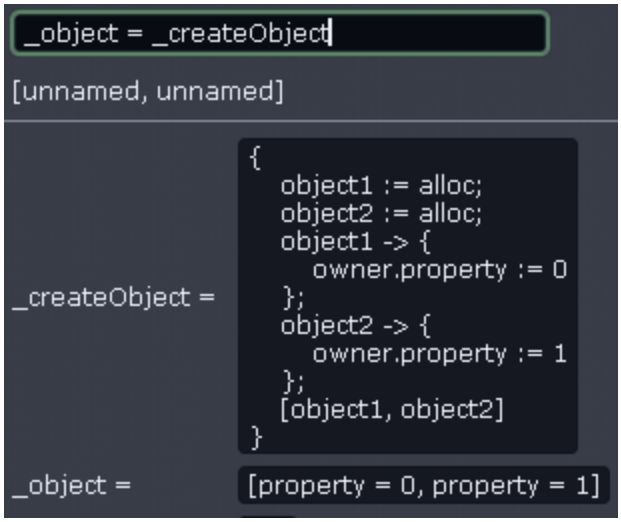

Since Algodoo doesn't know how to display an empty `ClassObject` in the script menu, it instead outputs `[BAD]`. The crash may be due to Algodoo not knowing how to display a list of `[BAD]` values and how it shares its nature with lists, thus Algodoo gets confused that `[BAD]` is in a list and crashes. When creating `ClassObject`s in object variables, there should at least be a property assigned to them.

**NOTE**: Object variables as ClassObjects are saved as functions, they are set to the return value when you exit out the input box of the object variable, make sure you have an `onSpawn` script dedicated for this.
This is not the case with **scene variables as ClassObjects**, as their properties are improperly saved as **global variables**, this may cause **Algodoo** to **call these variables over other properties** such as from objects.
You should remove these declarations or use `Scene.temp`, then **reset Algodoo** in Options if necessary.

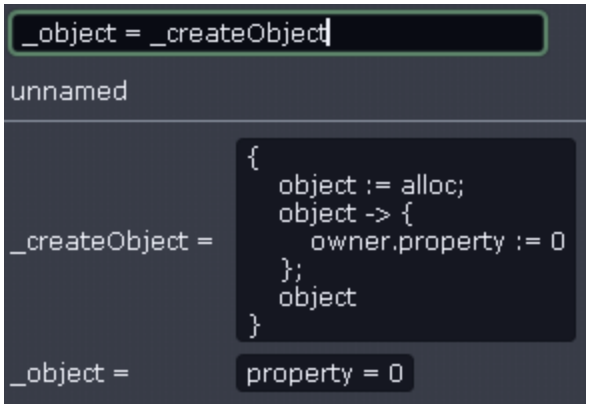
A workaround would be to create a function that returns a list of ClassObjects.

```
{
    object1 := alloc;
    object2 := alloc;
    object1 -> {
        // first object properties
    };
    object2 -> {
        // second object properties
    };
    [object1, object2]
}
```

Then, assign the function's return value into an object variable.




If you want to assign an object variable to only one ClassObject, just make the return value of the function return one.




One final note about syntax, some care is required when a function returns a `ClassObject`. If the function has no parameters, then you can access the variables within the returned `ClassObject` as if the function were the `ClassObject`. So, `function.value` is valid to access `value` from the `ClassObject` that `function` returns. However, the same does not hold true for functions with parameters. Instead, you need to surround the whole function in a set of parentheses, so `(function(5)).value` is the proper method of getting `value` from the returned `ClassObject`. A common example of this is `Scene.entityByGeomID`, where getting the radius of a circle that has the `geomID` of `5` requires doing this: `(Scene.entityByGeomID(5)).radius`.

## Owner and Entity

There are two important variables when dealing with objects: `owner` and `entity`. You should have already seen `owner` in the above section, but let's dive deeper into what it does.

`owner` gets the object that owns the current function. In something like `postStep`, this is of course the parent entity of the function. However, it only looks one level up. If you use any sort of if-statement, then using `owner` inside the statement won't work as now it's inside a function within a function. Since you're now in a function whose parent is a function, `owner` will instead return something other than the parent entity. Instead, `owner` will return the hidden object that contains the parameters of the parent function. In the case of a single if-statement in `postStep`, you can do something like `owner.e.dt`. However, you cannot chain `owner` to keep going up layers, as it appears it only works within functions. There's also some other weird behavior regarding custom functions in objects, where the owner of the custom function is an object separate from the entity that only contains the function.

Ultimately, `owner` isn't that useful outside of the redirection operator `->` when creating class objects. Most of the time, you may actually want to use `entity` instead.

`entity` works similarly to `owner`, but with a significant difference: `entity` always gets the parent entity of a function. There's no ambiguity on what it'll get. While generally its behavior is replicated with `e.this` in built-in functions, `entity` is useful in custom functions to get the object itself in case it is needed.

## Encapsulation

In object-oriented programming, **encapsulation** is the bundling of data with methods that operate on the data.

The general rule is that a property cannot be accessed directly, and must instead be accessed through **getter** and **setter** methods.

- Getter (or accessor) methods return the property's value.
- Setter (or mutator) methods allow you to change the property's value.

For example:

```
object := alloc;
object -> {
    owner._exampleProperty := "example";
    owner.getExampleProperty = {
        owner._exampleProperty;
    };
```

```
    owner.setExampleProperty = (value)=>{
        owner._exampleProperty = value;
    };
};
```

In the above example, `getExampleProperty` returns the value of `exampleProperty` and `setExampleProperty` changes the value of `exampleProperty` to the parameter `value`.

By convention, getter methods start with 'get', followed by the property's name. Similarly, setter methods start with 'set', followed by the property's name.

Most object-oriented languages have some way to make the data itself private (hidden from anything outside the object), forcing programmers to use the methods. Thyme does not provide any way of doing this. In some languages it is convention to begin private fields with an underscore. You can use this convention in Thyme to indicate that you do not want the data to be modified.

*Creating a getter method for a property but no setter method implies that the property is **read-only**.*

Setter methods can also be used as input validation - to make sure that the data never goes outside a certain range. For example, a health property should not go below zero and should not go above maximum:

```
Scene.my.enemy := alloc;
Scene.my.enemy -> {
    owner._maxHealth := 500;
    owner.getMaxHealth := {
        owner._maxHealth;
    };

    owner._health := 500;
    owner.getHealth := {
        owner._health;
    };
    owner.setHealth := (value)=>{
        owner._health = clamp(value, 0, owner._maxHealth);
    };
};
```

```
> Scene.my.enemy.setHealth(300)
300
> Scene.my.enemy.setHealth(-4)
0
> Scene.my.enemy.setHealth(600)
500
```

The benefit of encapsulation is that programmers do not have to worry about how the data is stored internally, how the getter and setter methods are coded, what data type is being used internally, etc.

## References

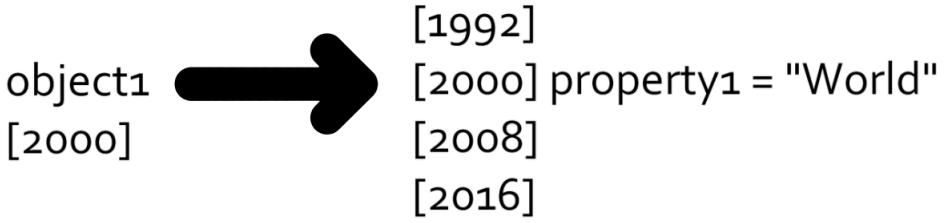
ClassObject variables store a reference to the contents of the object, rather than storing everything directly. This makes ClassObject a **reference type**.

This becomes clear if you attempt to assign an object to another variable:

```
> object1 := alloc;
unnamed
> object1.property1 := "Hello"
Hello
> object2 := object1;
unnamed
> object2.property1 = "World"
World
> object1.property1
World
```

Here, the `object1` variable is storing a *reference* to its data. The line `object2 := object1` declares a new variable, `object2`, which stores another reference *to the same data*.

References (also known as pointers) are memory addresses. For example, the `object1` variable could be storing the memory address 2000:

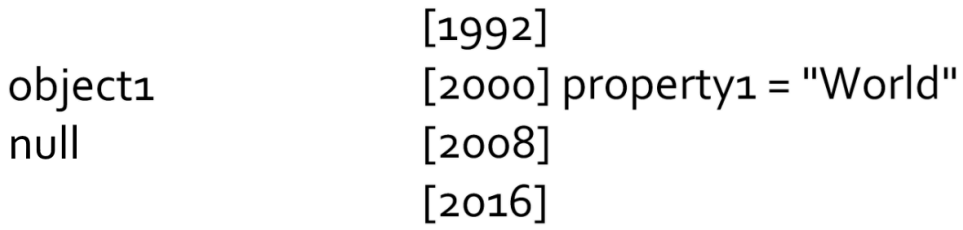


When you tell Algodoo to set a variable equal to `object1`, the new variable simply takes the memory address rather than copying the contents of everything at that address.

*Strings, lists and functions are also reference types, however, since they are immutable (you cannot change one without redeclaring it) and you cannot modify parameter values in functions, this does not make a difference.*

## Null

When a variable is storing `null`, it is a reference type that is not actually storing a reference to anything. Setting `object1` from the above example to `null` would break the link between `object1` and its properties:



# Operators

# Operators

An operator is a symbol used to do certain operations on operands to get a result. There are three types of operators:

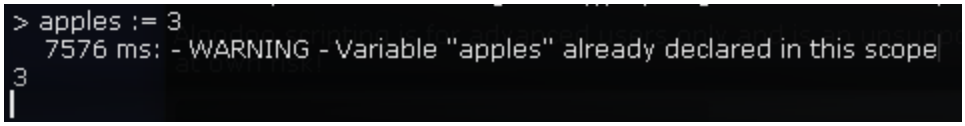
| Operator | Example | Description |
|---|---|---|
| Prefix | `-(3)` returns `-3` | The operator is placed before the operand. |
| Infix | `3 + 4` returns `7` | The operator is placed between the operands. |
| Postfix | N/A | The operator is placed after the operand. |

The vast majority of Thyme's operators are infix. There are no postfix operators in Thyme.

## Assignment operators :=, =

The `thyme.cfg` file in Algodoo's installation directory (possibly the closest thing we have to official documentation) states that `:=` is supposed to be used for declaring a variable and `=` for changing a variable. Both operators, however, can be used interchangeably in some situations.

Using `:=` to change a variable will cause a warning to be displayed, but still change the variable as intended.




Incorrect use of `:=` is likely to cause bugs. If you try to access a variable with it, you may end up creating a new variable without changing the existing variable if the existing variable isn't part of the current scope. Bugs with incorrect use of `=` are less common, though can still occur and most often occur with object creation functions (`Scene.addBox`, `Scene.addCircle`, etc.). It is recommended to always use the correct assignment operator. `:=` creates variables, `=` changes them.

Here's a common example of one of the bugs caused by using the wrong operator.
The following code will change the position of the **original** object (along with setting the new object's position):

```
Scene.addCircle({
    pos = [10, 10];
```

```
});
```

The following code will only set the position of the newly created object (which is likely the intention):

```
Scene.addCircle({
    pos := [10, 10];
});
```

The assignment operators also return a value. For example, `apples = 3` both sets `apples` to `3` and returns the value `3`. You can use this to set the values of multiple variables at the same time:

```
apples = pears = oranges = 3;
```

## Arithmetic operators +, -, *, /, %, ^

The arithmetic operators only work when the data types of the operands are `int`, `float`, or `list` (more information on lists below the list of operators)

The addition operator `x + y` adds `x` and `y` together, returning the result.

The subtraction operator `x - y` subtracts `y` from `x`, returning the result.

The multiplication operator `x * y` multiplies `x` and `y` together, returning the result.

The division operator `x / y` divides `x` by `y`, returning the result. Note that if the data types of `x` and `y` are both `int`, the data type of the result will also be `int` (so `3 / 2` returns `1`). This is known as integer division and is useful for certain applications.

The modulo operator `x % y` divides `x` by `y`, returning the remainder. The result will be negative when `x` is negative, `y` being positive or negative has no effect. Be careful when doing math with this as some programming languages and calculators reverse this, where it's `y` instead of `x` that determines if the result is negative.

The exponentiation operator `x ^ y` raises `x` to the power of `y`, returning the result.

Regarding lists, there are two modes of operation:
Both operands are lists. In this case, the lists must be of equal length. Each element of the first list is operated on with the corresponding element in the second list. For example: `[3, 4] +`

`[5, 1]` returns `[8, 5]`. In other words, `[3, 4] + [5, 1]` is equivalent to `[3 + 5, 4 + 1]`.

One operand is a list and the other is a number. In this case, the single number will operate on every element of the list. For example: `[3, 4] * 2` returns `[6, 8]`. In other words, `[3, 4] * 2` is equivalent to `[3 * 2, 4 * 2]` This doesn't work for every operation, however. You cannot do this for addition and subtraction at all, you may only add or subtract two lists. And, you can only use a list as the first operand for division, modulo, and exponentiation. Only multiplication allows this case in any order.

One final thing, you may perform math on nested lists. The best way to explain how this works is with examples:
`[[1, 2], [3, 4]] * 2` returns `[[2, 4], [6, 8]]`. You can imagine it like this:
`[[1, 2], [3, 4]] * 2` is equivalent to `[[1, 2] * 2, [3, 4] * 2]`.

`[[1, 2], 3] + [[3, 4], 5]` returns `[[4, 6], 8]`. You can imagine it like this:
`[[1, 2], 3] + [[3, 4], 5]` is equivalent to `[[1, 2] + [3, 4], 3 + 5]`.

Basically, when the operations involve lists, Algodoo simply performs the operation across all elements of the list. Of course, since it's the same operation, this can happen again if Algodoo encounters further lists. As long as it doesn't run into an invalid operation going through these steps (such as mismatched list lengths, incorrect variable type, or an operation between a list and a number not supported by the specific operation), then it will function properly.

## Prefix plus and minus operators +, -

The prefix `+` operator returns the value of its operand (e.g. `+3` returns `3`). The prefix `-` operator returns the numeric negation of its operand (i.e. multiplies its operand by `-1` and returns it, e.g. `-(4 + 4)` returns `-8`). The data types of the operands must be `int`, `float`, or `list`. In the case of lists, all values within must ultimately be `int` or `float`.

## String concatenation operator +

The `+` operator can also be used between two strings as the string concatenation operator. String concatenation is the process of joining two strings end-to-end. The data type of one of the operands must be `string`. The data type of the other operand cannot be `function` or `ClassObject`.

For example, `"Hello" + " World!"` returns `"Hello World!"`:

```
> "Hello" + " World!"
Hello World!
```

Like the arithmetic operators, this operator can work on two lists, but only when the data type of one of the operands is `string` and the other is not `function` or `ClassObject`. When used on two lists, each corresponding element of the lists is operated on and a new list of the results is returned. For example: `["Hello", "Alg"] + [" World!", "odoo"]` would return `["Hello World!", "Algodoo"]`.

One interesting quirk with string concatenation is that because it shares the same operator as addition, you can add together lists of both strings and numbers like so:
`["Hello", 5] + [" World!", 10]` returns `["Hello World!", 15]`.
This is due to the fact that these operations are indeed identical. All operators (except assignment operators) are simply shortcuts to functions, and the `math.add` function that `+` uses works on both strings and numbers.

## List concatenation operator ++

Lists can also be concatenated using the list concatenation operator `++`. The data type of the operands must be `list`.

```
> list1 = [1, 2, 3]
[1, 2, 3]
> list2 = [4, 5, 6]
[4, 5, 6]
> list1 ++ list2
[1, 2, 3, 4, 5, 6]
```

## Boolean logical operators !, &&, ||

The prefix logical NOT operator `!` returns the logical negation of its operand (i.e. it returns `false` if the operand is `true` and returns `true` if the operand is `false`).

The logical AND operator `&&` returns `true` if both operands are `true`, `false` otherwise.

*In some languages, the logical AND operator 'short-circuits'. When evaluating `x && y`, if `x` is `false`, `false` is returned immediately - it does not bother with `y` at all.* ***Thyme does not 'short-circuit'*** *- the second operand is always evaluated.*

The logical OR operator `||` returns `true` if either of its operands are `true`, `false` otherwise.

*Similarly to the logical AND operator, the logical OR operator also short-circuits in some languages. When evaluating `x || y`, if x is `true`, `true` is returned immediately - it does not bother with `y` at all.* ***Thyme does not do this*** *- the second operand is always evaluated.*

### Reference table of logic gates

| Gate | Example | Description |
|---|---|---|
| AND | `x && y` | Outputs true if both inputs are true. |
| OR | `x \|\| y` | Outputs true if either input is true. |
| NOT | `!x` | Outputs true if input is false. |
| NAND | `!(x && y)` | Outputs true if either input is false. |
| NOR | `!(x \|\| y)` | Outputs true if both inputs are false. |
| XOR | `(x \|\| y) && !(x && y)` | Outputs true if both inputs are different (one is true and the other is false). |
| XNOR | `!((x \|\| y) && !(x && y))` | Outputs true if both inputs are the same (both are true or both are false). |

Due to a quirk in Thyme that prevents `bool` variables from being compared with equality operators, logic gates are the only way to compare `bool` variables directly. For reference, `x XOR y` is equivalent to `x != y`, and `x XNOR y` is equivalent to `x == y`. However, a simpler method may be to use `math.toInt`, so `math.toInt(bool) == math.toInt(bool)`. This function converts `bool` values to `0` and `1` for `false` and `true` respectively, allowing them to be compared.

## Equality operators ==, !=

The equality operator `==` returns `true` if the operands are equal, `false` otherwise. If the data types of the two operands are different, an error is thrown **unless** the data types are `int` and `float` (order does not matter).

An error is thrown if either data type is `bool`, `function` or `ClassObject` (unless the two variables reference the same `ClassObject`):

```
> (i)=>{ print(i); } == (i)=>{ print(i); }
  31651 ms: - WARNING - Failed to evaluate: (i)=>{print(i)} == (i)=>{print(i)}, Incompatible types
error
> (Scene.entityByGeomId(20)) == (Scene.entityByGeomId(21))
  34319 ms: - WARNING - Failed to evaluate: (Scene.entityByGeomId(20)) == (Scene.entityByGeomId(21)), Incompatible types
error
```

The inequality operator `!=` returns `true` if its operands are not equal, `false` otherwise. `x != y` is equivalent to `!(x == y)`, so will throw the same errors as the equality operator described above.

## Comparison operators <, >, <=, >=

The comparison operators only work if the data types of the operands are `int`, `float`, or are both `string`s.

The less than operator `<` returns `true` if the first operand is less than the second operand (e.g. `3 < 4` returns `true`).

The greater than operator `>` returns `true` if the first operand is greater than the second operand (e.g. `3 > 4` returns `false`).

The less than or equal to operator `<=` returns `true` if the first operand is less than or equal to the second operand. `x <= y` is the same as `x < y || x == y`.

The greater than or equal to operator `>=` returns `true` if the first operand is greater than or equal to the second operand. `x >= y` is the same as `x > y || x == y`.

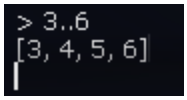
Comparing strings seems to compare the left-most character of both strings by ASCII code, and iterate to the right until the end of the string.

```
> "A" > "a"
false
> "a" > "A"
true
> "ca" > "bz"
true
> "ff " > "ff"
true
```

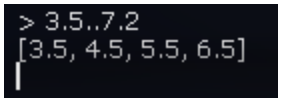
## Range operator ..

The range operator `..` returns a list of numbers between the two operands. It only works when the data types of the operands are `int` or `float`.

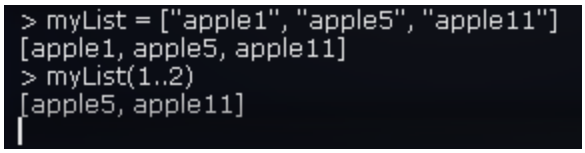
For example, 3..6 returns a list with the integers between 3 and 6 inclusive:




When using floats, the list will include the float from the first operand and start counting up by one until the floats are larger than the second operand.




The range operator can also be used with list indices, returning a list containing the elements between the element specified by the first operand and the element specified by the second operand inclusive.




## Ternary operator ?:

The ternary operator `?:`, sometimes referred to as the conditional operator, is the only operator that takes three operands. It is used as follows:

```
condition ? returnThisIfTrue : returnThisIfFalse;
```

`condition` must be a boolean value or boolean expression.

## Member access operator .

The member access operator `.` takes the name of a `ClassObject` as its left operand and the name of one of the ClassObject's properties as its right operand. It returns the value of the ClassObject's property, if it exists.

For example, Sim.timeFactor would return the simulation speed:

```
> Sim.timeFactor
value = 1
default = 1
```

## Indirection operator ->

The indirection operator `->` takes the name of a `ClassObject` as its left operand and a function as its right operand. It sets the given `ClassObject`'s properties using the given function. It returns the return value of the function.

The following script would move the camera to position `[0, 8]`, turn it upside down and change its zoom to `30` (0.2x).

```
Scene.Camera -> {
    pan = [0, 8];
    rotation = math.pi;
    zoom = 30;
};
```

## Summary table

| Operator | Type | Example expression | Example returns |
|---|---|---|---|
| := | Infix | radius := 5 | 5 |
| = | Infix | apples = 5 | 5 |
| + | Infix | 3.5 + 2.5 | 6.0 |
| - | Infix | 7 - 4 | 3 |
| * | Infix | 2 * 4.5 | 9.0 |
| / | Infix | 15 / 5 | 3 |
| % | Infix | 3 % 2 | 1 |
| ^ | Infix | 5 ^ 3 | 125 |
| + | Prefix | +3 | 3 |
| - | Prefix | -6 | -6 |

| | | | |
|---|---|---|---|
| + | Infix | `"Algodoo " + "Thyme"` | `"Algodoo Thyme"` |
| ++ | Infix | `[1, 2] ++ [3, 4]` | `[1, 2, 3, 4]` |
| ! | Prefix | `!(4 < 5)` | `false` |
| && | Infix | `4 > 3 && 3 > 2` | `true` |
| \|\| | Infix | `2 > 3 \|\| 3 > 2` | `true` |
| == | Infix | `8 == 8.0` | `true` |
| != | Infix | `2 != 2` | `false` |
| < | Infix | `6 < 5` | `false` |
| > | Infix | `9 > 2` | `true` |
| <= | Infix | `5 <= 4` | `false` |
| >= | Infix | `3 >= 3` | `true` |
| .. | Infix | `3 .. 6` | `[3, 4, 5, 6]` |
| ?: | Infix | `true ? 1 : 0` | `1` |
| . | Infix | `math.pi` | `3.1415927` |
| -> | Infix | `Scene.camera -> { zoom = 30; }` | `30` |

## Custom operators

Most operators seem to have been created using the `infix` keyword.

The syntax resembles a function where parameters take any number of _ underscores separated by any set of characters as operators.
Then it accepts a function with the same number of parameters, which the operator will run using its given arguments.

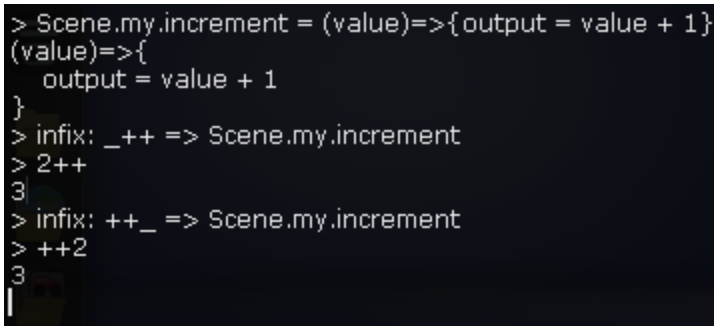
Here is an example of the range operator (found in the thyme.cfg file of the default Algodoo directory):

The two arguments after `infix` are unknown but are optional, a `:` colon must separate `infix` and the parameters.

```
infix 5 left: _ .. _ => inclusive_range
```

It is recommended not to make custom operators as they get removed after Algodoo reloads, and lacks clarity in what it does.

Despite the name, you can also set prefix and postfix operators.
*This increments the given value by 1 like in other programming languages.*



# Readability Guide

# Readability Guide

When writing code, especially code you want to show others, you should aim to make it readable. This is a good practice to get into because:

- It helps other people understand your code, which is important if you want them to help you.
- It helps you understand your own code in case you need to come back to it later.
- It's easier to change and add to your code.

## Meaningful identifiers

The name of a variable or parameter should always describe what the variable or parameter is doing. For example:

```
// Bad
(e)=>{
    d := math.vec.len(vel);
    e.other._HP = e.other._HP - d;
    print(d + " damage dealt");
}

// Good
(e)=>{
    damageDealt := math.vec.len(vel);
    e.other._HP = e.other._HP - damageDealt;
    print(damageDealt + " damage dealt");
}
```

In the good example, it is clear what `damageDealt` represents - you only have to look at the name to work out what it's for. In the bad example, it takes more time to work out what the variable `d` is being used for - the line where `d` is declared does not make it clear.

The impact of bad variable names worsens as your code gets longer. If `d` were referenced towards the end of a long block of code, you would have to scroll up to the beginning to work out what `d` actually represents. This wastes unnecessary time and would easily be prevented if `d` were given a meaningful name to begin with.

Avoid single-letter identifiers where possible. There are two exceptions to this:

`i` (which stands for **index** or **iterator**) in a for loop:

```
for(5, (i)=>{
    print(i);
});
```

`e` (which stands for **event arguments**) in an event function:

```
onCollide = (e)=>{
    e.other.color = [0.0, 0.0, 0.0, 0.0];
};
```

Both of these are acceptable as they are common programming conventions. Of course, if you want to use a more descriptive name, you are perfectly welcome to.

## Avoiding 'magic numbers'

A 'magic number' is an unnamed value with an unexplained meaning in code. For example, take the following simplified anti-gravity script:

```
postStep = (e)=>{
    vel = vel + [0, 0.32666668];
}
```

It is not immediately clear what the value `0.32666668` is supposed to be used for. Someone looking at the code may be able to work out that the script gives the object acceleration upwards, but the significance of the value `0.32666668` is not obvious at first glance. They may think: *"Why 0.32666668? Why not 0.33 or 0.3? What does this value actually represent?"*.

Compare the above code to the following:

```
postStep = (e)=>{
    vel = vel + [0, Sim.gravityStrength * 2 * e.dt];
}
```

In this example, it is clear that the script's acceleration is used to counter the object's acceleration due to gravity. Instead of using an unclear value, this script uses named values to clarify the purpose of the script.

# Using indentation correctly

Correct usage of indentation makes the intent of your code clear especially when that code contains many functions, indentations can also make it easier to spot errors in your code involving missing curly brackets.
While Algodoo does automatically correct the indentation of your code, it's still important to practice indentation when writing scripts in a text editor.

The general rule of thumb is to add indentation on all lines inside a block (curly brackets) {}, and increase per block.
Indents in Algodoo contain four spaces which can also be done using the Tab key, note that this would insert a Tab character in some text editors.

The first example shows the script without indentation making it hard to read and work with.

```
// Bad
Scene.addBox({
_count := 10;
postStep := (e)=>{
if_then_else(_count > 0, {
_count = _count - e.dt
}, {
timeToLive = 0
})
}
});

// Good
Scene.addBox({
    _count := 10;
    postStep := (e)=>{
        if_then_else(_count > 0, {
            _count = _count - e.dt
        }, {
            timeToLive = 0
        })
    }
})
```

# Comments

A comment is an annotation inside a script that is not run as code. Comments in Thyme work exactly as they do in C and its derived languages.

A **single-line comment** starts with `//` and continues until the end of the line. For example:

```
Scene.my.apples := 3; // stores the number of apples
```

A **multi-line comment** starts with `/*` and ends with `*/`. For example:

```
/* I don't know what I did here...
   Someone fix this... please... */
```

*Try to avoid making comments about the obvious like I did above. As a general rule of thumb, the more readable your code is, the less you'll have to use comments.*

## Using comments effectively

Comments are deleted by Algodoo when you enter code containing them. They can still be useful if you use an external text editor to edit Thyme scripts (explained in its own section) or if you need to show your code to others.

Comments should be used to summarise code or explain the intent of the code. If you use comments to state the obvious or state exactly what the code is doing, it may be a sign that your code is too hard to understand and should be re-written. For example:

Bad:

```
apples := 3 // stores the number of apples, initially 3
```

This comment is just restating the code. The code makes it obvious that the `apples` variable stores the number of apples and that there are initially 3 apples - the comment is completely unnecessary.

Better:

```
// _objects should not contain duplicates
_objects = set.insert(_objects, object);
```

In this example, the comment is explaining why `set.insert` is being used instead of the list concatenation operator.

# Property Guide

# Property Guide

There are far too many properties and variables that exist in Algodoo to exhaustively list in this document. If you wish to learn more about a specific property that isn't already listed below, check out the companion document:
Thyme Property List

The following document contains several helpful functions (including xFor, xWhile, filter, map and fold, mentioned earlier).
Thyme Additional Functions

## Mathematical constants

| Property | Value | Type | Description |
|---|---|---|---|
| `math.e` | `2.7182817` | Float | Mathematical constant e. |
| `math.pi` | `3.1415927` | Float | Mathematical constant π. |

## Common mathematical functions

This following table lists most of the math functions used. It does not list all of them, as many are simply helper functions to enable infix operations to exist.

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `clamp` | `t` | Float: Input value | Clamps `t` within the provided `mn` and `mx` bounds:<br>• If `t` is less than `mn`, returns `mn`<br>• If `t` is greater than `mx`, returns `mx`<br>• Otherwise, returns `t` |
| | `mn` | Float: Lower bound | |
| | `mx` | Float: Upper bound | |
| `lerp` | `a` | Float/List: Lower bound | Linearly interpolates between `a` and `b` based on `t`, where `a` is when `t` is `0` and `b` is when `t` is `1`. `a` and `b` must be of the same type, and if they are lists they must be lists of floats and of equal length. (Technically can |
| | `b` | Float/List: Upper bound | |
| | `t` | Float | |

| | | | |
|---|---|---|---|
| | | | work on nested lists, follows the same rules as adding nested lists) |
| `math.acos` | `n` | Float: Domain in range [-1, 1]. | Returns the arccosine of `n` in radians if within the domain, otherwise returns `NaN`. |
| `math.asin` | `n` | Float: Domain in range [-1, 1] | Returns the arcsin of `n` in radians if within the domain, otherwise returns `NaN`. |
| `math.atan` | `n` | Float | Returns the arctangent of `n` in radians. |
| `math.atan2` | `y` | Float: Y value of vector | Alternate function for arctangent which takes components of a vector and returns the angle of the vector in radians. Use this function to get the angle of a vector. |
| | `x` | Float: X value of vector | |
| `math.cos` | `n` | Float: Radians | Returns the cosine of `n` in radians. |
| `math.log` | `n` | Float: Domain in range [0, ∞] | Returns the natural log of `n` if within the domain, otherwise returns `NaN`. |
| `math.log10` | `n` | Float: Domain in range [0, ∞] | Returns log base 10 of `n` if within the domain, otherwise returns `NaN`. |
| `math.max` | `a` | Float | Returns the highest of `a` and `b`. |
| | `b` | Float | |
| `math.min` | `a` | Float | Returns the lowest of `a` and `b`. |
| | `b` | Float | |
| `math.sin` | `n` | Float | Returns the sine of `n` in radians. |
| `math.sqrt` | `n` | Float: Domain in range [0, ∞] | Returns the square root of `n` if within the domain, otherwise returns `NaN`. |
| `math.tan` | `n` | Float | Returns the tangent of `n` in radians. |

## Generating random numbers

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `rand.boolean` | `None` | N/A | Returns either `true` or `false`. |
| `rand.direction2D` | `None` | N/A | Returns a vector of length 1 with a random direction. |
| `rand.normal` | `None` | N/A | Returns a random float value with normal distribution with standard deviation of 1 |
| `rand.normal2D` | `None` | N/A | Returns a random vector with normal distribution with a standard deviation 1 (Precise mechanics unknown) |
| `rand.uniform01` | `None` | N/A | Returns a random float value with range [0, 1] |

Even though there is no function for generating a set range of numbers, you can use `rand.uniform01` with some math to accomplish that goal. The basic math works out to be `rand.uniform01 * (max - min) + min`. To generate integers rather than floats, you instead do `math.toInt(rand.uniform01 * (max - min)) + min`. In this case, the max is exclusive, so a min of 4 and a max of 10 will generate numbers 4 through 9.

## Converting between types

*Excluding `math.toString`, all conversion functions will act upon each element individually, returning a list of converted values.

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `math.toBool` | `var` | Bool, Float, Int, String, List* | Converts variables to booleans. Numeric types are false when zero, otherwise true. Strings must only be `"true"` or `"false"`, otherwise throws an error (is case insensitive). |
| `math.toFloat` | `var` | Bool, Float, Int, String, List* | Converts variables to floats. `true` returns `1.0`, `false` returns `0.0`. |

| | | | |
|---|---|---|---|
| | | | Strings must only contain numbers, otherwise throws an error. |
| `math.toInt` | `var` | Bool, Float, Int, String, List* | Converts variables to ints. `true` returns `1`, `false` returns `0`. Strings must only contain integers, otherwise throws an error. |
| `math.toString` | `var` | Bool, Float, Int, String, List | Converts variables to strings. |

## Converting between color formats

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `math.HSL2RGB` | `color` | Float List: HSLA format | Converts an HSLA color to RGBA (not to be confused with HSVA). |
| `math.HSV2RGB` | `color` | Float List: HSVA format | Converts an HSVA color to RGBA. |
| `math.RGB2HSL` | `color` | Float List: RGBA format | Converts an RGBA color to HSLA (not to be confused with HSVA). |
| `math.RGB2HSV` | `color` | Float List: RGBA format | Converts an RGBA color to HSVA. |

## Manipulating vectors

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `math.vec.dist` | `a` | Float List: `[x, y]` position | Returns the distance between points `a` and `b` |
| | `b` | Float List: `[x, y]` position | |
| `math.vec.distSq` | `a` | Float List: `[x, y]` position | Returns the square of the distance between points `a` and `b` |
| | `b` | Float List: `[x, y]` position | |
| `math.vec.len` | `vec` | Float List: `[x, y]` vector | Returns the length of `vec` |

| | | | |
|---|---|---|---|
| `math.vec.lenSq` | `vec` | Float List: `[x, y]` vector | Returns the square of the length of `vec` |

`math.vec.len` and `math.vec.dist` both use an additional square root operation. If you don't need the exact length/distance and want to compare them, `math.vec.lenSq` and `math.vec.distSq` are faster as they don't use a square root, cutting down the number of operations.

So:

```
math.vec.len([a, b]) < math.vec.len([c, d]) // slower
math.vec.lenSq([a, b]) < math.vec.lenSq([c, d]) // faster - no square root calculated
```

## Manipulating strings and lists

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `string.length` | `var` | String/List | Returns the number of characters if `var` is a string, or the number of elements if `var` is a list. |
| `string.list2str` | `list` | List | Converts `list` into a string by converting all elements into strings and concatenating them. |
| `string.split` | `str` | String | Splits `str` into a list of strings using `delim` as a delimiter. Throws an error if `delim` is longer than one character. |
| | `delim` | String: Single char | |
| `string.str2list` | `str` | String | Splits `str` into a list of single character strings. |

## Manipulating sets

A mathematical set is represented in Algodoo as a list. Mathematical sets cannot contain duplicates, so these functions assume that you do not want duplicates in your lists.

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `set.insert` | `set` | Float/String List | Appends `value` to the end of `set` only if `value` does not already |

| | | | |
|---|---|---|---|
| | `value` | Float, String | exist. `set` can only be a list of floats/ints, or a list of strings, not both. |
| `set.merge` | `a` | Float/String List | Merges `a` and `b` by appending `b` onto `a`, removing duplicate values from `b` doing so (does not remove duplicates that exist in `a`). Both sets may only contain either floats/ints, or strings, not both. |
| | `b` | Float/String List | |

For example, `set.merge([3, 4], [3, 5])` returns `[3, 4, 5]`.

## Keyboard

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `Keys.bind` | `key` | String: Key code | Binds `key` to `func`, where pressing the bound key will run `func`. This binding persists across scene loads. |
| | `func` | Parameterless Inline Function | |
| `Keys.isDown` | `key` | String: Key code | Returns a bool based on whether or not `key` is being held down. |
| `Keys.unbind` | `key` | String: Key code | Unbinds `key` from all functions bound to it. |

Key codes are names for keys on the keyboard; mouse; or gamepad, commonly they are the same as what it outputs when you type, others are assigned special names.
The following list commonly used keys, seek Thyme Built-In Properties for all possible key codes.

| Name | Output |
|---|---|
| `"return"` | Returns true if the Enter key is pressed. |
| `"enter"` | Returns true if the Enter key on the numeric keypad is pressed. **Only supported on keyboards with numeric keypads**. |
| `"space"` | Returns true if the Spacebar key is pressed. |

| | |
|---|---|
| | **Not recommended as it's intrinsically binded to pause/play the simulation.** |
| `"mouse_left"` | Returns true if the Left Mouse Button is pressed. |
| `"mouse_middle"` | Returns true if the Middle Mouse Button is pressed. |
| `"mouse_right"` | Returns true if the Right Mouse Button is pressed.<br>**Not recommended as it's intrinsically binded to open the properties menu or rotate objects when held**. |
| `"ctrl"` | Returns true if the Ctrl key is pressed.<br>*Adding left_ or right_ at the start does not correlate to the left or right Ctrl keys.* |
| `"alt"` | Returns true if the Alt key is pressed.<br>*Adding left_ or right_ at the start does not correlate to the left or right Alt keys.* |
| `"shift"` | Returns true if the Shift key is pressed.<br>*Adding left_ or right_ at the start does not correlate to the left or right Shift keys.* |
| `"backspace"` | Returns true if the Backspace key is pressed.<br>**Not recommended as it's intrinsically binded to delete selected objects**. |
| `"tab"` | Returns true if the Tab key is pressed.<br>**Not recommended as it's intrinsically binded to toggle the visibility of the Algodoo's GUI**. |
| `"escape"` | Returns true if the Escape key is pressed. |
| `"delete"` | Returns true if the Delete key is pressed.<br>**Not recommended as it's intrinsically binded to delete selected objects**. |

## Mouse

| Property | Value | Type | Description |
|---|---|---|---|
| `App.mousePos` | `[x, y]` | Float List | Position of the cursor in the scene. |

## Time

| Property | Default Value | Type | Description |
|---|---|---|---|
| `Sim.frequency` | `60` | Int | Number of simulation steps (ticks) per second |
| `Sim.tick` | `0` | Int | Number of ticks that have elapsed since the scene started. |
| `Sim.time` | `0.0` | Float | Number of simulation seconds that have elapsed since the scene started. Equal to `Sim.tick / Sim.frequency` unless frequency has been changed after the scene started. |
| `Sim.timeFactor` | `1.0` | Float | Multiplier for how fast the scene runs. |
| `System.time` | `0.0` | Float | Number of seconds that have elapsed since Algodoo was opened. Not affected by how fast the scene is running. |

## Manipulating the camera

| Property | Default Value | Type | Description |
|---|---|---|---|
| `Scene.Camera.pan` | `[0.0, 0.0]` | Float List | Position of the camera in the scene. |
| `Scene.Camera.zoom` | `150.0` | Float | Zoom of the camera. Lower values mean more zoomed out. |
| `Scene.Camera.rotation` | `0.0` | Float | Rotation of the camera in radians. |

## Manipulating gravity

| Property | Default Value | Type | Description |
|---|---|---|---|
| `Sim.gravityAngleOffset` | `0.0` | Float | Direction of gravity in radians. Unlike other rotations, `0` is downwards. |
| `Sim.gravityStrength` | `9.8` | Float | Strength of gravity in meters per second. |

## Manipulating wind

| Property | Default Value | Type | Description |
|---|---|---|---|
| `Sim.windAngle` | `0.0` | Float | Direction of wind in radians. |
| `Sim.windStrength` | `0.0` | Float | Speed of wind in meters per second. |
| `Sim.airFriction Linear` | `0.01` | Float | Linear component of air friction. |
| `Sim.airFriction Quadratic` | `0.001` | Float | Quadratic component of air friction. |
| `Sim.airFriction Multiplier` | `1.0` | Float | Multiplier applied to air friction. |

## Creating objects

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `Scene.addBox` | `props` | Parameterless Inline Function | Creates and returns a box with properties declared within `props`. |
| `Scene.addCircle` | `props` | Parameterless Inline Function | Creates and returns a circle with properties declared within `props`. |
| `Scene.addFixjoint` | `props` | Parameterless Inline Function | Creates and returns a fixate with properties declared within `props`. Requires `pos`. Only seems to work well when created in conjunction with other newly created objects. |
| `Scene.addGroup` | `props` | Parameterless Inline Function | Groups objects together. Technically doesn't require anything, but requires `entityIDs` in `props` to be useful. Best used to change what objects the camera follows, see Changing followed objects for more info. |
| `Scene.addHinge` | `props` | Parameterless Inline Function | Creates and returns an axle with properties declared within `props`. |

| | | | |
|---|---|---|---|
| | | | Requires `pos`. Only seems to work well when created in conjunction with other newly created objects. |
| `Scene.addLaserPen` | `props` | Parameterless Inline Function | Creates and returns a laser with properties declared within `props`. Requires `pos`. |
| `Scene.addLayer` | `props` | Parameterless Inline Function | Creates and returns a layer with properties declared within `props`. Requires `id`. |
| `Scene.addPen` | `props` | Parameterless Inline Function | Creates and returns a tracer with properties declared within `props`. Requires `pos`, only persists if it can attach to a geometry. |
| `Scene.addPlane` | `props` | Parameterless Inline Function | Creates and returns a plane with properties declared within `props`. |
| `Scene.addPolygon` | `props` | Parameterless Inline Function | Creates and returns a layer with properties declared within `props`. Requires `surfaces`. |
| `Scene.addLineEndPoint` | `props` | Parameterless Inline Function | Creates and returns a line end point with properties declared within `props`, used in conjunction with creating springs. Requires `pos`, will disappear without a spring. |
| `Scene.addSpring` | `props` | Parameterless Inline Function | Creates and returns a spring with properties declared within `props`. Requires `lineEndPoint1` and `lineEndPoint2` to persist. |
| `Scene.addWater` | `props` | Parameterless Inline Function | Creates water, only uses and requires `vecs` and `vels` in `props`. |
| `Scene.cloneEntityTo` | `entity` | ClassObject: Entity | Clones `entity`, placing it at `pos`. Returns the cloned entity. |
| | `pos` | List: [x, y] position | |
| `Scene.removeEntity` | `entity` | ClassObject: Entity | Removes `entity` from the scene. |

To use the object creation functions, you declare the object's properties inside a parameter-less function, e.g.:

```
Scene.addCircle({
    color := [1.0, 0.5, 0.0, 1.0];
    pos := [5.0, 5.0];
});
```

This would create an orange circle which would spawn at the position `[5.0, 5.0]`. Any properties that aren't specified will have their default values.

An easy way to spawn a tracer on an object is to do the following:

```
Scene.addPen({
    attachedObject := Scene.addBox({
        // box properties here
    });
    geom := attachedObject.geomID;
    pos := attachedObject.pos;
    // other tracer properties here
});
```

This script creates a variable `attachedObject`, which is the object that will have the tracer attached to it. The script then sets the tracer's `geom` property to the `geomID` of the new box.

A similar principle can be applied to fixjoints, lasers, etc.

To add water, you need to give the positions and velocities of each water particle. For example:

```
Scene.addWater({
    vecs := [[0, 0], [1, 0], [2, 0]];
    vels := [[0, 0], [-1, 0], [-2, 0]];
});
```

This will create a water particle at (0, 0) with no speed, a water particle at (1, 0) moving to the left and a water particle at (2, 0) moving faster to the left.

To create a group using scripting, simply add the `entityID`s of each object you want to group, you can also add a name for your group but it isn't mandatory.
Note that `"tracked"` and `"selected"` exist by default and have unique functionality, so they shouldn't be used, unless of course you intend to replace those groups with a new one.

```
Scene.addGroup({
```

```
    name := "";
    entityIDs := [5, 17, 73]
})
```

This will group all objects with an `entityID` of `5`, `17`, and `73`.

One last thing to know, if you regularly work with `ClassObject`s, you may also work with objects having references to other objects. Due to an odd quirk with how `ClassObject`s are handled, you cannot declare a `ClassObject` in a create object function, instead any objects are converted to functions. Instead, you must declare the `ClassObject` after the create object function. Simplest way to do it is like so:

```
circle := Scene.addCircle({
    // circle properties
});
circle._object := alloc;
```

Doing it this way will properly create a `ClassObject` stored in the newly created object. The same thing is true for references, which is useful if you want the object to still be able to access the object that created it:

```
circle := Scene.addCircle({
    // circle properties
});
circle._parent := e.this;
```

## The 'body' variable

`body` is a read-only property that can be found in all geometries. It controls what physics body the geometry is a part of. Geometries that are glued together will have the same `body`, and geometries not glued together will have different `body` values. Geometries glued to the background will have a `body` of `0`. Geometries of inactive layers will have a `body` of `-1`.

Because `body` is a read-only property, you can only set it upon creating new geometries, and it cannot be changed after the geometry has been created.

## Finding objects

| Function | Params | Param Description | Function Description |
|---|---|---|---|

| | | | |
|---|---|---|---|
| `Scene.entityByGeomID` | `id` | Int: Geom ID | Returns the entity in the scene if it exists, throws warning otherwise. |
| `Scene.entityByID` | `id` | Int: Entity ID | Returns the entity in the scene if it exists, throws warning otherwise. |

Before you attempt to use these find object functions, you need to be aware that objects will often have different IDs upon reloading the scene. You should not hardcode an ID, instead you need to have the object provide its current ID upon scene load (such as through `onSpawn`) or find some other way of accessing the ID (within certain entities, you can find the IDs of attached geometries within their property list, though generally only accessible through `readable`).

## File I/O

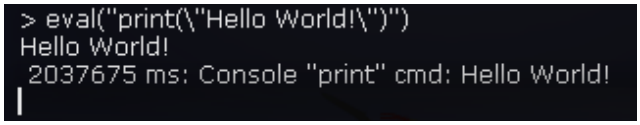
File I/O (input/output) refers to transferring data to or from a file. Thyme offers two functions for working with files

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `System.ReadWholeFile` | `file` | String: File path | Returns the contents of `file` as a string. If it fails, it returns `null`. `file` is a file path where the root is the user Algodoo directory. |
| `System.WriteToFile` | `file` | String: File path | Appends text to `file`. Returns `true` on success. |
| | `text` | String | |

## Evaluating strings as code

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `eval` | `str` | String: Script | Evaluates `str` as a script in the current scope. |
| `geval` | `str` | String: Script | Evaluates `str` as a script in the global scope. |
| `Reflection.ExecuteCode` | `str` | String: Script | Same as `geval`. |
| `Reflection.ExecuteFile` | `file` | String: File path | Reads `file` and executes it as code in the global scope. |

For example, `eval("print(\"Hello World!\")")` would print `"Hello World!"` to the console:




This code shows the difference between `eval` and `geval`:

```
(e)=>{
    eval("localVar := 4"); // localVar is declared with block scope, so will be deleted when the closing curly bracket } appears

    geval("globalVar := 7"); // globalVar is declared with global scope and will be deleted when Algodoo is reset
}
```

## UI

| Function | Params | Param Description | Function Description |
|---|---|---|---|
| `Scene.addWidget` | `props` | Parameterless Inline Function | Creates and returns a widget with properties declared within `props`. Requires valid `widgetID`. |
| `App.GUI.ShowMessage` | `message` | String | Creates a window displaying `message`. |
| `God.AddCheckbox` | `cvar` | String: Bool Variable | Creates a checkbox that toggles the variable in `cvar`. |
| `God.AddLabeledButton` | `text` | String | Creates a button labeled with `text` that evaluates `func` as a script when pressed. |
| | `func` | String: Script | |
| `God.AddSlider` | `cvar` | String: Float Variable | Creates a slider that changes `cvar` going from `min` to `max`. Uniquely, this function has an optional parameter. `logarithmic` by default is `false`, but it can be set to `true` to make the slider logarithmic. |
| | `min` | Float | |
| | `max` | Float | |
| | `logarithmic` | Bool (OPTIONAL) | |

| | | | |
|---|---|---|---|
| `God.AddTexturedButton` | `texture` | String: Skin path | Creates a button with a texture that evaluates `func` as a script when pressed. This texture is not a regular texture, but points to an image file in Algodoo's currently selected skin. |
| | `func` | String: Script | |

## God

| Property | Default Value | Type | Description |
|---|---|---|---|
| `God.godMode` | `false` | Bool | Whether or not God Mode is active. God Mode enables various features, such as extra options for axles and chains, more digits in the info tab of objects, and an increase in how far you can zoom in. |

# Techniques

# Techniques

Learn about other techniques you can do with Algodoo.

- Reducing lag
- List techniques (Testing for an item in a list, Removing items from lists)
- Simulating anti-gravity
- Creating trigger hitboxes
- Object techniques (Destroying objects, changing followed objects, object constructors)
- Creating polygons
- Texture manipulation
- Color manipulation
- Editing scene files
- Writing add-ons

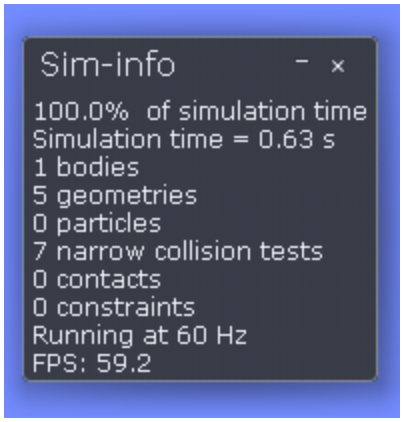
# Reducing lag

# Reducing lag

Performance is a crucial part of simulations and can result in slow and inconsistent progression if not taken into account.
It is important to look into efficient solutions to ensure that performance isn't impacted when the simulation is running, this is especially useful for people you share the scene with in case they don't have a powerful device to run the scene.

## Sim-info

Sim-info logs information about the simulation, you can access it in the File menu and click the Show sim-info option.
You can leave it open as the simulation is running to see how your scene is performing.




Sim-info lists the following information.

- **n% of simulation time / Paused** - How much of the simulation is running based on the simulation speed. e.g. 50.0% of simulation time at double speed means it's running at normal speed (1x).
- **Simulation time = n s** - The number of simulation seconds has passed since the simulation first started. In other words, this is the number of simulation steps that have passed divided by the simulation frequency. Equivalent to `Sim.time`.
- **n bodies** - The number of physics bodies that are used in the scene. A physics body is a group of geometries that are glued together, the background has a physics body in which geometries glued to the background are static.
- **n geometries** - The number of geometries that exist in the scene.
- **n particles** - The number of water particles that exist in the scene.
- **n narrow collision tests** - The number of tests Algodoo is performing to check whether geometries are colliding with one another, Algodoo checks for collisions if two objects'

bounding boxes overlap. A bounding box is the smallest rectangle that can be drawn around the object without any rotation, a good way to view it is by using the Scale tool on a geometry.

- **n contacts** - The number of points where geometries are touching each other. *Separate from collisions. During the tick an object first collides with another, it does not count as a contact*.
- **n constraints** - The number of axles and springs. *The drag tool also technically counts as a constraint.*
- **Running at n Hz** - The simulation frequency where it controls the number of physics steps performed per second, this can be changed in the Options menu under the Simulation tab. The simulation frequency should match Algodoo's frame rate (60 FPS by default, 90 in older versions), lower values make physics choppy, whereas higher values make physics accurate but can impact performance. Any changes may also break some scripts that rely on it being the default value (60 Hz). Equivalent to `Sim.frequency`.
- **FPS: n** - The framerate of Algodoo. This is separate from the simulation, it only affects visual fidelity, lasers and anything to do with them, and the `update` function. Framerate should ideally be 60.

You can improve performance by doing the following.

1. The number of bodies should ideally be near the number of geometries, as bodies containing a large number of objects can cause lag when reloading the scene, pressing undo/redo, and when moving an object that's part of a larger body using `pos` changing scripts. The best thing you can do is avoid gluing anything to the background and instead give them infinite density. *A section further along describes why this is a good idea in further detail.*
2. Reduce the number of geometries in the scene, fewer geometries mean less work Algodoo has to do in the scene. *This also accounts for killer planes, which can be deactivated in the Options menu under the Simulation tab.*
3. Remove any water particles that are not in use, water is notoriously laggy for numerous reasons. You can remove all the water particles in the File menu and click on the Erase all water option.

## Optimizing scripts

It is also important to optimize scripts in the scene to prevent unnecessary code from causing lag spikes and frame drops in Algodoo.

The general rule of thumb is to have the program run as few operations as possible.

In this example, this object starts as red but turns green for 5 seconds when it collides with another object.

```
_cooldown := 0;

// Bad
onCollide = (e)=>{
    _cooldown <= 0 ? {
        _cooldown = 5
    } : {}
};

postStep = (e)=>{
    _cooldown > 0 ? {
        _cooldown = _cooldown - e.dt;
        color = [0, 1, 0, 1]
    } : {
        color = [1, 0, 0, 1]
    }
};

// Good
onCollide = (e)=>{
    _cooldown <= 0 ? {
        _cooldown = 5;
        color = [0, 1, 0, 1];
        postStep = (e)=>{
            _cooldown > 0 ? {
                _cooldown = _cooldown - e.dt;
            } : {
                color = [1, 0, 0, 1];
                postStep = (e)=>{};
            }
        }
    } : {}
}
```

The first example sets `_cooldown` to `5` when an object collides with it, then while `_cooldown` is above `0`, it turns green and `_cooldown` ticks down until it hits `0` where it turns red.

The problem with this script is that the `postStep` script makes this check for every simulation step causing unnecessary calls which may affect performance, this is easily resolved if the script is first initialized when it is needed and is removed afterwards.

Of course, the above example script only serves as an example. The "bad" version is a perfectly acceptable script in most cases, and it is more readable than the "good" method. Though, if you have lots of objects running the script, it may be worth switching to the more performant method.

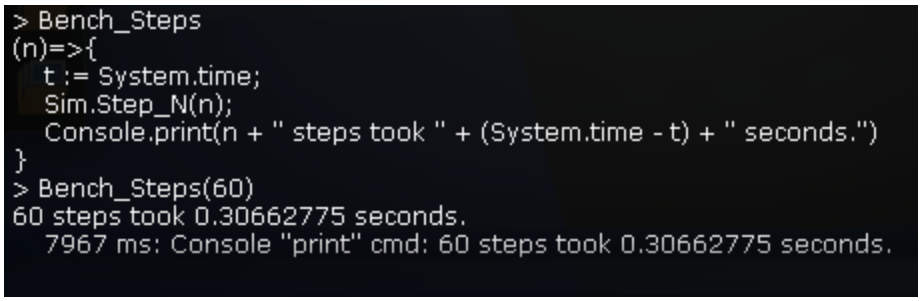
Here are some ways to reduce lag on your scene. Ordering of the list is based on importance, ease of optimization, and overall impact.

1. Read your scripts diligently and think about how Algodoo runs them, focus on finding redundant code and reduce the number of things (variables, functions, etc) used in the script as much as possible. *Algodoo runs them in a specific order, further information can be found in the Update Order section*
2. When comparing values with `<` or `>` operators, make sure both operands have defined values - `null` and `undefined` have been observed to impact performance.
3. Avoid `sin()`, `cos()`, `if()`, and `time`. Opt for `math.sin()`, `math.cos()`, `Sim.time`, `if_then_else()`, or the ternary operator `? :`. The former functions work by simply calling the latter, acting as essentially middlemen that add extra unnecessary overhead.
4. Avoid scene variables (`Scene.my.`) unless they absolutely need to be accessed globally in the scene. Opt for local variables whenever possible.
5. Use function properties instead of setting values through other functions when possible. For example, `vel` being set to `{ [0.0, 0.0] }` is better than a `postStep` script that sets `vel` to that value at all times.
6. Reduce the number of math operations you perform. If you can, try precomputing multiple operations, so instead of `var / 5.0 * 3.0`, try `var * 0.6`. Be wary though as this can lead to magic numbers in your script, so only do this if you need the performance boost.
7. If you're utilizing `math.vec.len` or `math.vec.dist`, you may want to try using `math.vec.lenSq` or `math.vec.distSq`. These alternate functions don't perform the square root necessary for the correct answer. But, if you are comparing two lengths or distances and don't need the correct values, they are useful.
8. Avoid super long variable names, the fewer characters the better. *The effect of long variable names is often insignificant. As long as variable names aren't longer than 20 characters, you will be fine. Please choose readability over performance in this specific case.*

One last thing, Algodoo is known to have memory leaks and generally decreases in performance over time. It's a good idea to regularly restart Algodoo when working on larger projects to have a better idea on the actual performance.

## Benchmarking

Benchmarking helps you to assess the performance of scripts. In Algodoo, you can use the built-in `Bench_Steps` function for benchmarking. It takes in a number of steps (ticks), runs the scene for that many ticks and prints how many seconds it took to run those ticks.




You can use benchmarking to compare the performance of multiple different solutions to the same problem.

## Scene.temp

`Scene.temp` is an alternative to `Scene.my`.

While both objects are used to store variables intended to be accessed anywhere in the scene, `Scene.temp` is explicitly meant for temporary variables that should not be saved with the scene.
If you enter in `Scene.temp.var := 1` into the console, then you can access `Scene.temp.var` up until you reload the scene, or even undo the scene where the variable will be undefined.

This may initially seem to be not useful, however there are cases where you may not actually want variables to save with the scene, such as...

- Variables that are only defined when the scene starts.
    - This can help clean up the `Scene.my` object to only have more important variables.
- Object references as they don't persist through scene reloads anyway.

    - This allows for a super easy check to see if you need to recreate an object reference, where just doing `Scene.temp.obj == undefined` will return `true` once the object reference gets deleted.

Another benefit of storing object references in `Scene.temp` is that it will prevent variable leak.

An issue with storing object references in `Scene.my` is that when the scene is saved, the definitions of the object's properties get erroneously saved in the scene.
When the scene is reloaded, those definitions are interpreted as creating new variables within the global scope.

This is undesirable as it can lead to unintended behavior due to how Algodoo handles scope. Instead of something like `radius` being undefined in boxes, the value returned will be the `radius` in global scope.

# A better way to glue to background

To glue a geometry to the background, you may prefer setting the geometry's `density` to `+inf`, this will act as if it's being glued to the background.

Some benefits with infinite density include...

- Freezing and unfreezing a geometry in place with scripts, which is impossible for the other built-in methods.
- No performance issues when moved via scripts.
- Reloading the scene won't slow down.
- Applying velocity will only affect the collision physics of that single geometry.

These issues are due to how physics bodies work.

## The 'body' variable

Every geometry has a special read-only `body` variable that controls which "physics body" the geometry is a part of.
A physics body is simply a collection of geometries that all act as a singular object.

By default, each geometry gets its own physics body, so each object has a unique `body` value. Two geometries that are either glued together or connected with fixates are part of the same physics body, so they will share the same `body` value.

When you glue or fixate a geometry to the background, its `body` is set to `0` - the background can be thought of as being part of the oth body.
This means that every geometry glued to the background is technically glued to each other as well.

*Additionally, if you're utilizing the layer feature, making a layer inactive sets all body values on that layer to `-1`.*
*When you make the layer active again, none of the geometries remember being glued to the background or to each other, and instead they all fall freely.*

It also explains why if you unglue multiple geometries from the background at once, the geometries still end up glued together, which is a problem.

In order for Algodoo to properly process geometries being glued together, it needs to calculate internal values for their shared body whenever the scene is reloaded, or whenever a single geometry is changed relative to the rest of the body.
This causes increased load times for the scene, and also means that moving, rotating, and resizing glued objects that are glued causes lag.

If you've attempted to move glued geometries via script, you may have noticed that the script may be much worse for performance than other scripts that affect physical properties (such as velocity).
Considering that many scenes can easily have over a hundred geometries glued to the background, you can see how this might be a significant cause for lag.

*If you're interested in the numbers, on Erikfassett's computer, 192 geometries were given a position changing script in their postStep functions.*
*Using the sim-info window to check performance, the scene ran at 17.7% speed when the geometries were glued to the background.*
*However, with the infinite density method, the scene ran without any slowdown at all.*

Also if you apply a velocity to an object glued to the background, you will notice that suddenly every glued geometry in the scene gains this velocity.
This is because geometries glued together will share the same velocity values at all times (as long as angular velocity is 0, at least), and geometries glued to the background are considered glued to each other.

This leads to odd and seemingly unexplainable buggy behavior that can be frustrating.

# List techniques

# List techniques

## Testing for an item in a list

Testing if an item exists in a list is not a particularly difficult task. In fact, here's a naive (but still useful in certain circumstances) method of doing so:

```
hasItem := (list, item)=>{
    ret := false;
    for(string.length(list), (i)=>{
        list(i) == item ? {
            ret = true
        } : {}
    });
    ret
}
```

This method is considered naive as it is limited in some ways. Most notably, due to the recursion limit, this method only works on lists that aren't more than about fifty elements long (exact number is unclear). Additionally, this method is slow, and it worsens with longer lists. However, there is a far better method of handling this: `set.insert`.

Algodoo provides a couple of functions for handling sets. The rules for sets mean that you cannot have duplicate items in a single set. `set.insert` helps with this by appending an element to the end of a list only if the element does not already exist. Otherwise, the list is unchanged. If you can't already tell, this means we can compare the old list and new list to see if the list had changed with the function. If it has changed, that means the item couldn't have already been in the list. If it hasn't changed, then the item must have already been in the list. So, to facilitate this, here's a function using `set.insert`:

```
hasItem := (list, item)=>{
    string.length(list) == string.length(set.insert(list, item))
}
```

This function will work on lists of any size, and is a vast performance improvement compared to the for loop method. A simple test on Erikfassett's machine revealed the for loop method to take more than 3 times longer on a list of 4 elements, and almost 10 times longer on a list of 16 elements.

You may think it would be best to always use the `set.insert` method, but there is one caveat it has: it only works on lists of either numbers or strings. No other variable type works, whereas with the for loop any type that can be compared with `==` works (and it can be modified to suit other types of variables and nested lists). And, the list needs to be homogenous, a mixed list of strings and numbers does not work with `set.insert`. However, as lists of types other than strings and numbers are relatively rare, and heterogenous lists even rarer, this should cover most cases.

## Removing items from lists

One of the more annoying things to work with in Thyme are lists. Due to how they work, they are rather static. At best, you can easily do some math on whole lists, and combine two lists into one. You can't change singular values at an index (`list(i) = value` is illegal), and you can't easily remove an item from the list. Though, when it comes to removing items from a list, we do have more options.

The only real way we can remove items from a list is to create a smaller sublist from the original list. The easiest items to remove are the items at the beginning and end of the list. To remove the item at the beginning, you can simply do this: `list = list(1 .. (string.length(list) - 1))`. This uses the range operator `..` with the length of `list` to create a list of values like so: `[1, 2, 3,...]`, where the last number in this new list is equal to the last index of `list` (as the last index is equal to one less the length of the list). Because using a list of values as the index of a list produces a new list containing all of the values in the index list, you will get a list of numbers without the first item.

The same logic above applies to the end of a list, with the command slightly modified to this instead: `list = list(0 .. (string.length(list) - 2))`. The index list instead starts at `0` to keep the first item and ends one less than the last index (two less than the length).

Of course, the above methods don't work as well for items in the middle of a list. Though, if you already know the current length of a list, you can do it manually: `list = list([0, 1, 2, 4, 5, 6])`. This creates a copy of `list` without the item at index `3` and puts it back into `list`. In other words, you're removing the item at index `3`. But, of course, this only works for lists that are seven items long. A longer list will lead to the end being chopped off, and a shorter list will produce an error. Though, if you want the end to be chopped off as well, then it can work for you too.

But, there is a general way to remove items from anywhere in a list:

```
list = list(0 .. (i - 1)) ++ list(i + 1 .. (string.length(list) - 1))
```

In this script, `i` is the index you want to remove. What it's doing is basically creating two sublists: the first sublist contains every item up to (but not including) `i`, and the second sublist contains every item after `i`. It then combines the two lists together to create a new list without the item at `i`. This script can work pretty well, but it has a couple of caveats: It's relatively slow (the `..` operator is relatively slow and worsens with larger ranges), it doesn't work for lists much longer than 50 items (again thanks to the `..` operator), and it's not helpful if you want to remove a specific value from the list without knowing its index.

If you wish to remove a specific value from a list, then using a for loop to produce a new list that excludes the specific item works well. Here's an example:

```
list := [1, 2, 3, 4, 5];
newList := [];
for(string.length(list), (i)=>{
    // include any list element that is NOT 3
    list(i) != 3 ? {
        newList = newList ++ [list(i)]
    } : {}
});
list = newList; // returns [1, 2, 4, 5]
```

This is a rather slow method, but it's reliable. It runs into similar issues to the above method to remove an item at an index, so you still need to avoid lists longer than 50 items and consider how often you remove items. If you absolutely need speed, then this isn't a super good method.

Luckily, there is one more method that is a good bit faster, but it only works well if the target list follows some rules. Specifically, the rules are that you should not have duplicate values in your list (you can, but you'll run into some likely undesired behavior), and the list must either contain only numbers or contain only strings. This is because this method utilizes a built-in `set` method. Here's an example:

```
list := [1, 2, 3, 4, 5];
list = set.merge([3], list);
list = list(1 .. (string.length(list) - 1));
list; // returns [1, 2, 4, 5]
```

So, what's going on here? Well, the trick is how `set.merge` works. As explained earlier in the document, a set is a mathematical construct that contains a bunch of items, but none of the items can be duplicates of each other. In Algodoo, this is represented using a list. All of the

methods in `set` are built assuming you want to avoid duplicate items. There's `set.insert` which appends an item to the end of a list unless the item already exists, and then `set.merge` which combines two lists in the same way `a ++ b` does, but removes duplicate items.

However, with `set.merge`, the first list isn't actually touched at all. With `set.merge(a, b)`, all items in `a` will remain intact (regardless of if there are duplicates in `a` or not). When combining the lists, the function only removes items from `b` if they already exist in `a`. So, `set.merge([0, 2, 3], [1, 2, 3, 4, 5])` produces `[0, 2, 3, 1, 4, 5]`. Now, if `a` were to only contain items that already exist in `b`, you may notice that the effect of `set.merge` is equivalent to just moving items to the front of the list. So, with `set.merge`, we can move an item to the front of the list and then make a simple operation to remove that item.

And, there's more, we can remove a specific index as well:

```
list := [1, 2, 3, 4, 5];
list = set.merge([list(2)], list);
list = list(1 .. (string.length(list) - 1));
list; // returns [1, 2, 4, 5]
```

Here, we've gotten the item at index `2` and moved it to the front to be easily removed. And, unlike the earlier method, this allows for any index to be removed.

Of course, as mentioned earlier, this only works on lists of either strings or numbers. And, importantly, the list shouldn't contain duplicate values. Both of these are due to how `set.merge` (and `set.insert`) work, with the former limitation being a restriction of the `set` methods, and the latter being due to a side-effect of using the `set` methods.

The main reason to avoid using this method on lists of duplicate values is that the duplicate values get deleted. It doesn't matter which item you're targeting, if an item has duplicates in the list, those duplicates are removed. So, outside the unlikely case you want duplicate entries to be removed as well, you should avoid this method if you know if you will be working with lists containing duplicate entries.

Ultimately, there isn't a single method for removing items from lists that's easy to use without any downsides. However, at the very least one of the above methods should hopefully work for you if you need to remove something from a list.

# Color manipulation

# Color manipulation

## Using layers to multiply colors

You can use the `Scene.addLayer` function to give layers a color (in the RGB format). Every object in the layer will appear as if its color has been multiplied by the layer's color.

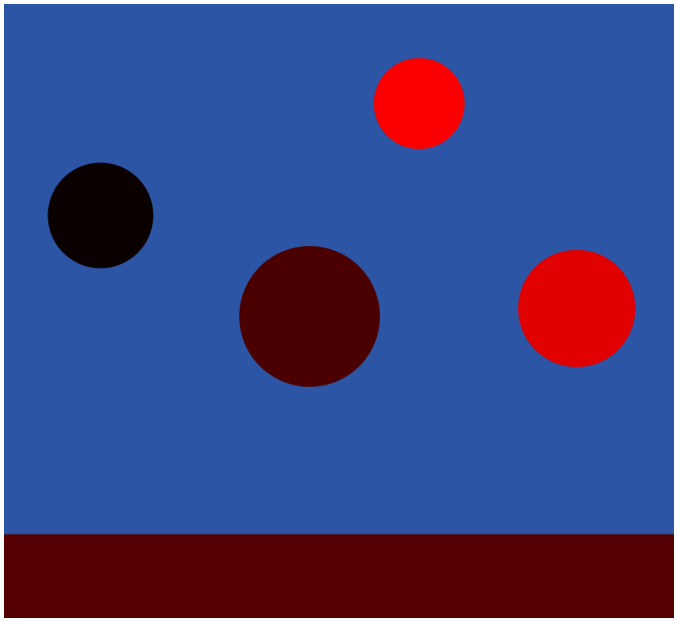
To use the `Scene.addLayer` function, you must provide an ID (the layer number). To update a pre-existing layer, just give the ID of the pre-existing layer.

For example, to turn layer o red, you would use this script:

```
Scene.addLayer({
    id := 0;
    color := [1, 0, 0, 1];
});
```

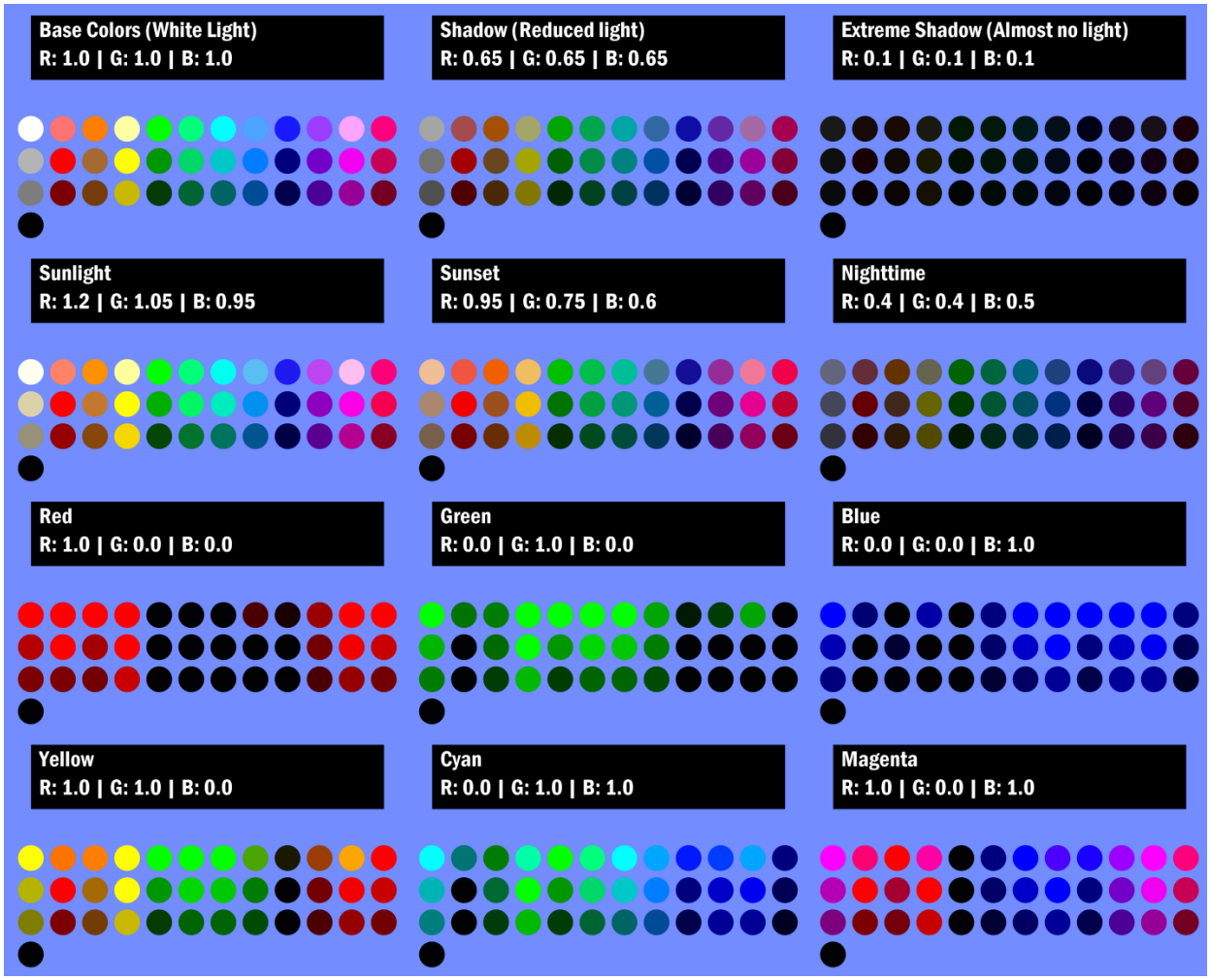
Example scene before the above script was applied:

The same scene after the script was applied:



The pink objects appear brighter as they have high red values. The green object, however, has a very low red value, so appears black.

Here is a chart displaying a set of colored marbles under various colored layers:

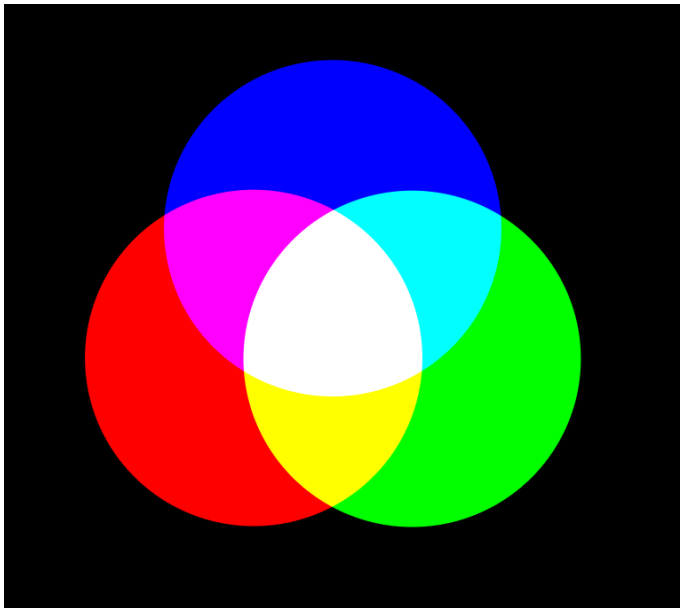



## Using value and transparency to add colors

If the value of an object's color is higher than `1` (100%) and transparency is low, then Algodoo's renderer will start adding the object's RGB color to objects behind it. This effect improves the larger value is and smaller transparency is.

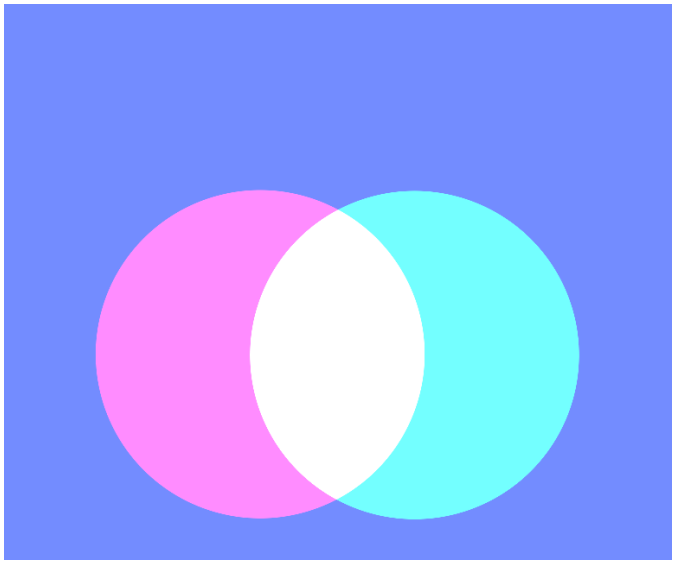
The numbers for value and transparency can be found by setting the object's color to whatever you want with 100% transparency, and then multiplying value by a large number while dividing transparency by the same large value. A good minimum value to multiply and divide by is `10000`. For a color with 100% value, this should lead to a value of `10000` (1,000,000%) and a transparency of `0.0001` (0.01%). With a 50% color value, then value would change to `5000`.

An example of how additive color works can be seen here:



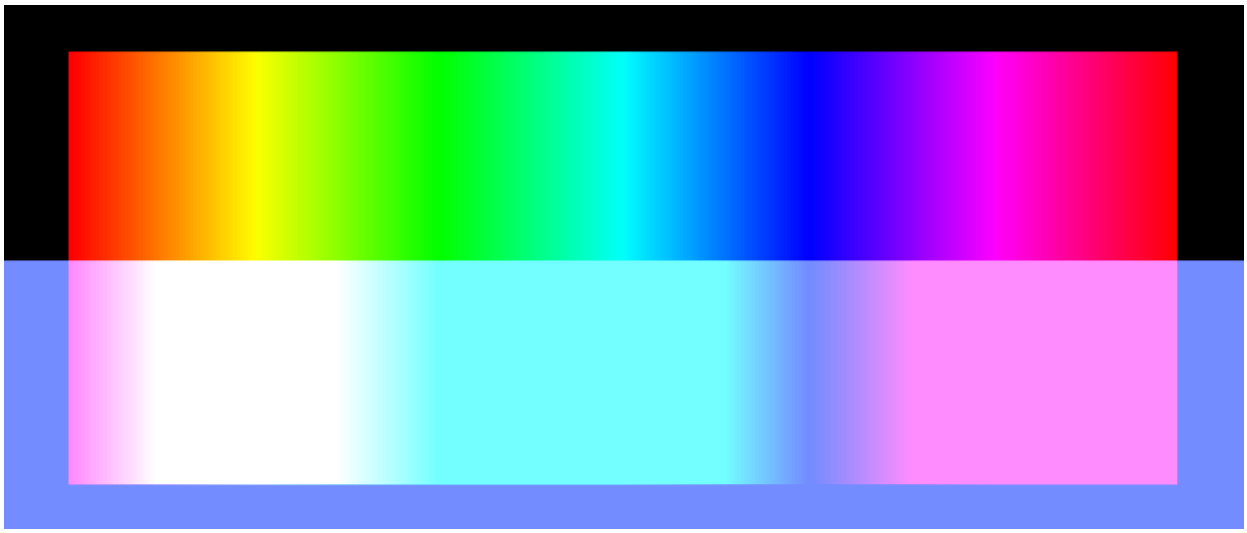
All three circles have a value of `10000` and transparency of `0.001`. The magenta, cyan, yellow, and white colors come from the circles adding their colors together.

The same three circles above will look different depending on different backgrounds. An example on the default Algodoo sky background can be seen here:



The blue circle disappears entirely because the background already has the maximum blue color, so adding more blue leads to no visual difference. This blue is also added to the red and green circles, leading to their pink and cyan colorations.

This effect also works on textures. This rainbow texture has both a black and sky background behind it, directly showing how each color is affected by different backgrounds.



Additionally, all objects in a layer can be made to have additive color blending by setting the layer color to an RGB value of `[10000, 10000, 10000, 0.0001]`. This can be done easily through running this function:

```
Scene.addLayer({
    id := 0;
    color := [10000, 10000, 10000, 0.0001];
});
```

This will force every object on layer 0 to additively blend its color with objects in their own layer and lower layers. You can additionally combine this with color multiplication by multiplying the above list with the RGB color you want to multiply by.

*If you wish to, you can multiply and divide the value and transparency by a smaller number to create a partial additive blend. This could be useful for things like fire, where you may want them to not completely disappear on bright backgrounds, but still appear brighter than they would with normal transparency blending. As a start, numbers 10 or less create a pretty good effect, but feel free to experiment. You cannot use negative numbers to recreate subtractive blending.*

# Texture manipulation

# Texture manipulation

A geometry's `textureMatrix` property controls how its texture is displayed. It is a 3x3 matrix, represented in Thyme as a list of 9 floats.

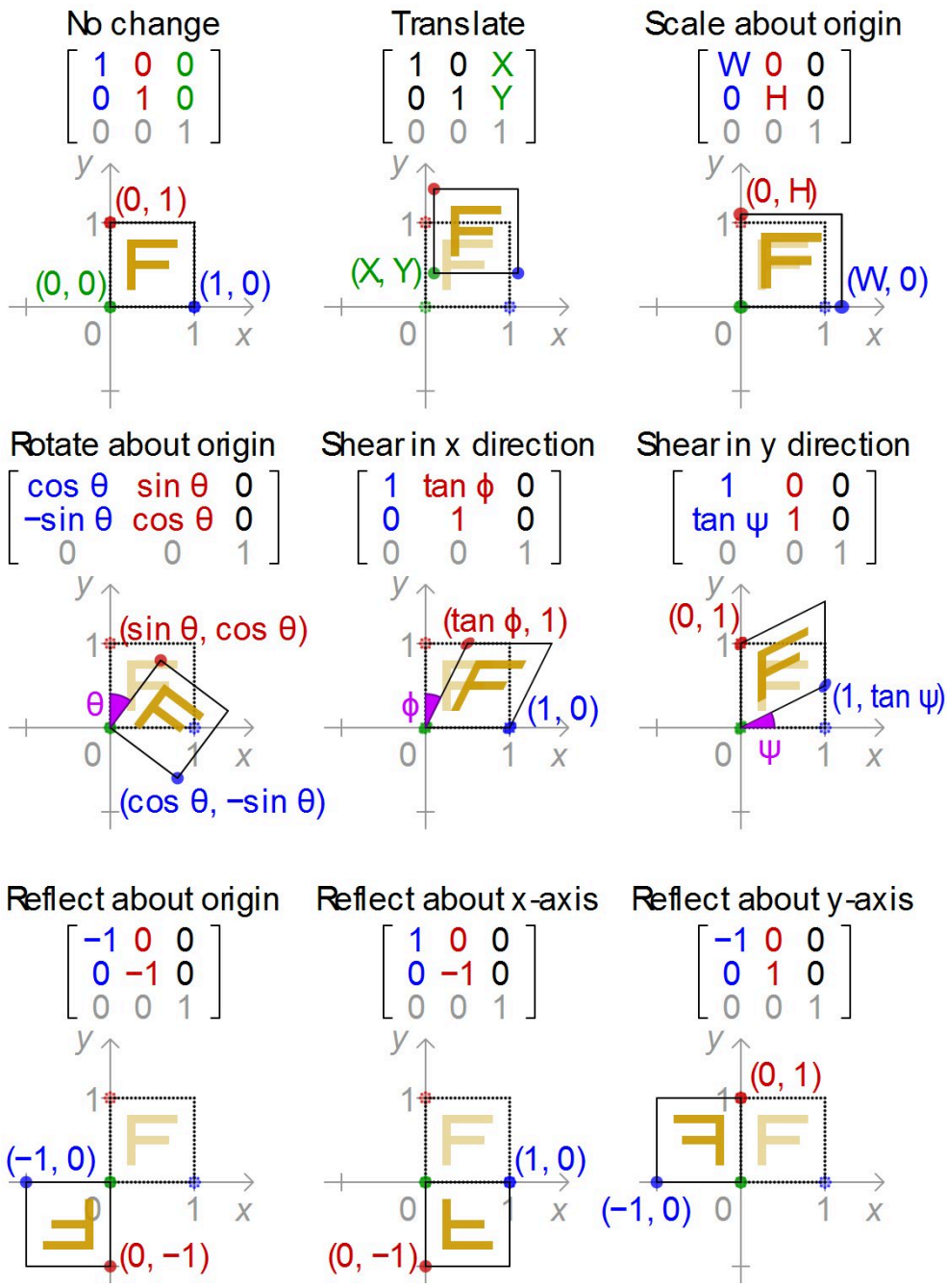
The default value of `textureMatrix` is the identity matrix:

```
[1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
```

In matrix notation:

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

The following chart shows several transformation matrices you can use to manipulate `textureMatrix`:




One important thing to keep in mind though is that Algodoo handles its `textureMatrix` a bit differently than what the above chart implies. Here's how to imagine how displaying textures in Algodoo works:

Imagine an infinite repeating grid of the texture you've selected. The texture is sized to be exactly a 1m x 1m square (with rectangular textures being squished to fit in the square). The

center of the grid, (0, 0), is at the corner of four adjacent grid squares. The center of the grid is also placed at the geometry's center of mass - or simply the center of the geometry.

What `textureMatrix` defines is the shape, size, and rotation of a "view" or "camera" that is looking at this infinite repeating texture grid. The displayed texture will be what's seen under the view, where both the view and the infinite grid are both warped and transformed equally so that the view becomes a 1x1 square centered at the origin. The resulting infinite texture grid after this transformation is then what is displayed.

How the `textureMatrix` defines the view is as such: The 3rd and 6th numbers define the x and y coordinate of the center of the view (i.e., the general position of the view on the infinite texture grid). The 1st and 4th numbers define the x and y positions of the top left corner of the view *relative* to the bottom left corner, and the 2nd and 5th numbers do the same but with the bottom right corner relative to the bottom left corner. The top right corner is simply the top left and bottom right corners combined. The last three numbers of the matrix should be left untouched. Though, the very last number can be seen as a multiplier for all other numbers, where all numbers are multiplied by the last number.

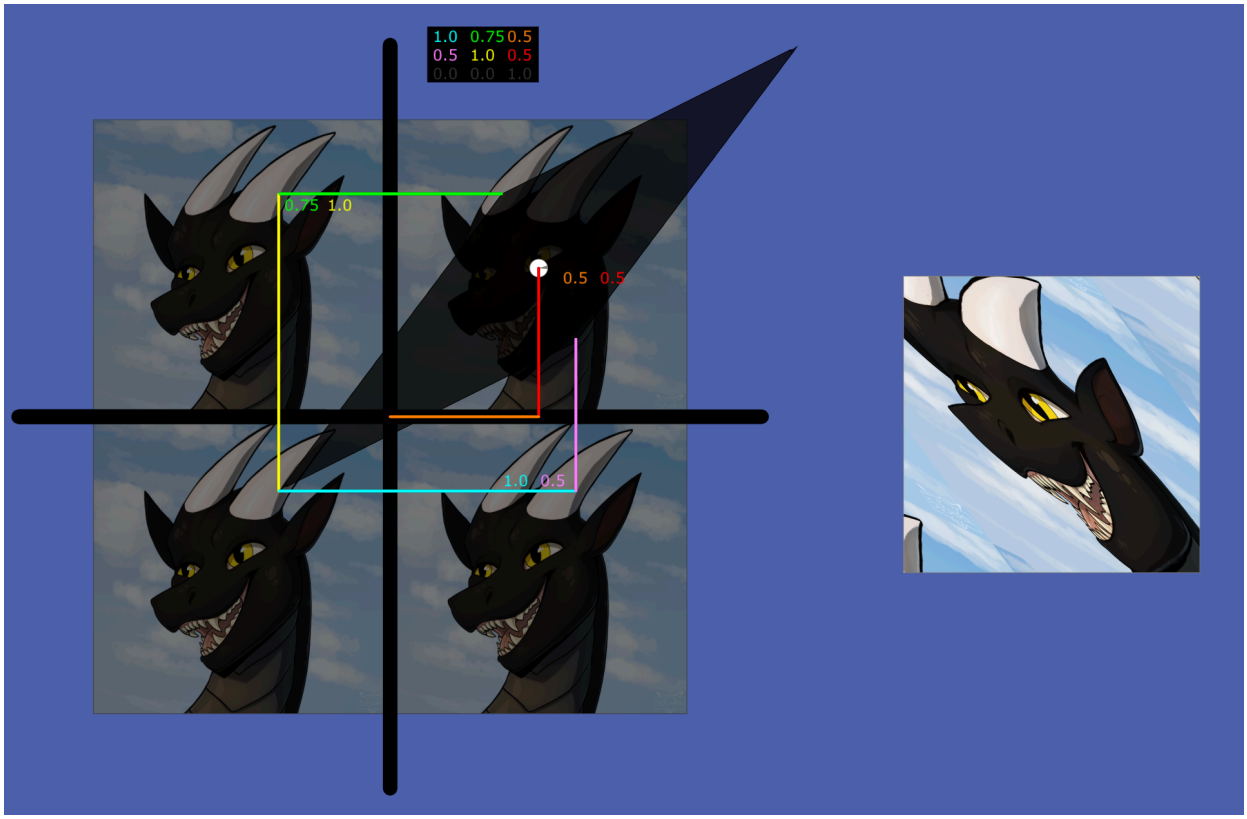
It is important to remember that the positions of the corners of the view are *relative* to the bottom left corner, and the position of the bottom left corner is such that 3rd and 6th numbers will always describe the position of the center of the view.

*The reason the corners aren't defined based on the origin is likely to make texture rotation easier*

If the above was hard to follow, here are some example images of what is happening. The following examples will use Erikfassett's old Discord avatar, which is an image created by artist 5ushiRoll.

This first image helps show how the view and textures warp with it. The top row shows what the view looks like on an unmodified version of the texture grid, the middle row shows the part of the texture the view actually sees, and the bottom row shows the texture being warped back to be square and centered.

This second image shows how the view is affected by each number in the `textureMatrix`. With this image, you can see how the positions of the top left and bottom right corners are defined relative to the bottom left corner rather than the origin.



One final note about `textureMatrix` is that rectangular textures are first warped to be square, which then requires the view to warp in such a way so that the final displayed texture is once again properly rectangular. Additionally, the view isn't precisely what's actually displayed on the final texture. It's just used as a reference for how to warp the texture plane, where the view must return to being a 1m x 1m square. The displayed texture is the final warped texture plane cut out by the geometry, where the center of the plane is the center of the geometry.

# Object techniques

# Object techniques

## Destroying objects

There are two official ways to destroy entities:

1. Setting or `timeToLive` to `0`.
2. Using the `Scene.RemoveEntity` function (e.g. `Scene.RemoveEntity(e.this)` in any event function; `Scene.RemoveEntity(e.other)` in onColllide).

`Scene.RemoveEntity` and setting `timeToLive` to `0` are the recommended methods to delete objects. In some cases, setting `timeToLive` to `0` is actually preferable because in some specific cases `Scene.RemoveEntity` for an unknown reason has a single tick delay.

*For older versions of Algodoo, setting specific physics properties to `0` or `NaN` is the only available method - this is not recommended for current versions.*

## Changing followed objects

If you look at a scene file, you might notice two groups that exist by default:

```
Scene.addGroup({
    name := "tracked"
});
Scene.addGroup({
    name := "selected"
})
```

The "tracked" group stores the entityIDs of the objects that are being followed by the camera. Similarly, the "selected" group stores the entityIDs of the selected objects in the scene.

You can modify the contents of either group by using the `Scene.addGroup` function. For example:

```
Scene.addGroup({
    name := "tracked";
    entityIDs := [34, 35]
})
```

This causes the camera to follow the objects with entityIDs 34 and 35.

## Object constructors

Creating objects via script can become unwieldy as you need to specify all of the properties of the objects individually. This especially becomes a problem when you need to spawn multiple different objects. In some cases, just using a for-loop may suffice, but in others that isn't necessarily the case. If many objects being spawned have similar or identical properties, then you may want to use an object constructor.

The add object functions provided by Algodoo take an inline function as an argument. It is possible to have a function return an inline function, which means we can replace the inline function in the argument with a "constructor" function. A constructor function would be set up like so:

```
constructor := {
    {
        propertyA := value;
        propertyB := value;
        propertyC := value
    }
};
```

After setting up the constructor, you can then just run something like this: `Scene.addCircle(constructor)`. This will spawn a circle with the properties specified in the constructor.

Now, it is very important to note how the constructor is formatted. The properties of the object are nested within two pairs of braces, not one like typical functions are. This is because the inner pair of braces is an inline function that's being returned. The add object function then runs the returned inline function in order to set the spawned object's properties. Attempting to create a constructor that only has one pair of braces will lead to an error being thrown in the console and no object being spawned.

Right now, the constructor isn't super useful. While it works well for spawning a specific object, it only works for that specific object being spawned. Luckily, we can solve this pretty easily using parameters. Here's an example:

```
constructor := (position, velocity)=>{
    {
```

```
        pos := position;
        vel := velocity;
        color := [0, 0, 1, 1];
        radius := 0.5;
    }
};

Scene.addCircle(constructor([0, 3], [5, 10]))
```

In the above example, a constructor is set up to create a blue circle with a half meter radius. But, the constructor allows you to set your own position and velocity for the circle. And, in the above example, the spawned blue circle is given a position of `[0, 3]` and a velocity of `[5, 10]`.

It is also possible to add additional code before the returned inline function. In that case, you can use extra logic or conditionals to determine some values in the returned function. You could potentially even have multiple different possible inline functions be returned based on conditionals.

Constructor functions shouldn't always be used though. They are best used in situations where similar object spawn scripts are found in multiple different places in code. And, there is the alternative method of just calling a parameter function that itself contains the object spawn code, instead of spawning an object with a constructor function. Which method you use (function spawning object or spawning object with constructor) is up to you.

# Creating polygons

# Creating polygons

Creating circles or boxes using the object creation functions is fairly simple.

To create a circle with a specific size, you only need to specify its radius (a float or integer), e.g.:

```
Scene.addCircle({
    radius := 0.5;
});
```

To create a box with a specific size and orientation, you must specify its angle (a float or integer in radians) and its size (a list containing its width and its height - both floats or integers), e.g.:

```
Scene.addBox({
    angle := math.pi / 2;
    size := [1.0, 1.0];
});
```

You can add more variables in order to create the exact object you want. Any required variable that's left unspecified will be set to default values for the scene.

There are a few exceptions. Non-geometric entities like tracers, axles, and so on require certain properties in order to be spawned. Generally the required properties are simply the IDs of the geometries the entities will be attached to.

However, there is one object that requires a property much more complex: Polygons.

# The ‘surfaces’ variable

If you look at a polygon’s script menu, you will see a read-only variable named `surfaces`. It is a list of surfaces, where each surface itself is a list of vertices.


*Note that `surfaces` can only be accessed by using the `readable` function on a reference to the polygon. An example is `(readable(e.this)).surfaces` in an event function. This behavior is indicated by the variable being in the bottom section of the script menu.*

Algodoo uses each surface to draw a line, where it draws straight lines between vertices in order, as if connecting numbered dots where each dot’s number is its index in a list. Once Algodoo has all of the surfaces drawn, it uses two rules to fill in the polygon:

1. Every point along a surface must be adjacent to both filled and empty areas (i.e., every surface must be a *surface*)
2. The shape must be contained within a finite area.

These two rules don’t consider surfaces individually, but rather the combined structure of all surfaces together. An easy way to tell whether or not a point is within the polygon is to draw a straight line from that point towards any direction you want. If that line intersects surfaces an odd number of times, the point is within the polygon. If the line intersects surfaces an even number of times, then the point is outside the polygon. (If the line intersects a point where two surfaces intersect, or a point where a surface intersects itself, that counts as intersecting two surfaces).

There are some additional currently unknown rules regarding how multiple surfaces are allowed to be placed. More testing needs to be done, but it appears as if the combination of surfaces cannot allow for a discontinuous polygon. In other words, all parts of a polygon must be connected to each other.

Ultimately, how precisely Algodoo fills in a shape is often unnecessary for polygon generation. Only if you're trying to procedurally generate complex shapes may this information be more important.
All that's needed right now is that `surfaces` is a list of surfaces, and each surface is a list of vertices.

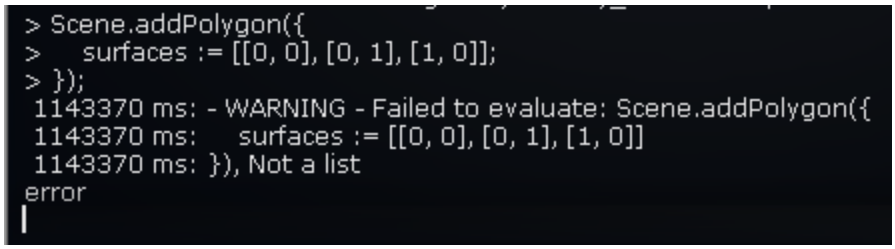
A simple right-angled triangle is created like this:

```
Scene.addPolygon({
    surfaces := [[[0, 0], [0, 1], [1, 0]]];
});
```

Each vertex is represented as a position vector (so a list of two floats or integers). The position points to where the vertex would be if `pos` were `[0, 0]` and `polyTrans` were `[1, 0, 0, 1]` (more on that variable later).

Notice that each vertex is stored in a list, and that list itself is also nested within a list. This is important. The three vertices in a list constitute a single surface, and that surface itself is in a list of surfaces. In this example, the list of surfaces only contains a single surface. The number of brackets you need can get confusing at times, especially with more complex scripts that procedurally generate surfaces.

If you're having trouble formatting the whole `surfaces` variable, the errors Algodoo throws should provide some hints.
In the first case, the error thrown states "Not a List". This means that you only have a single surface not within a list of surfaces.



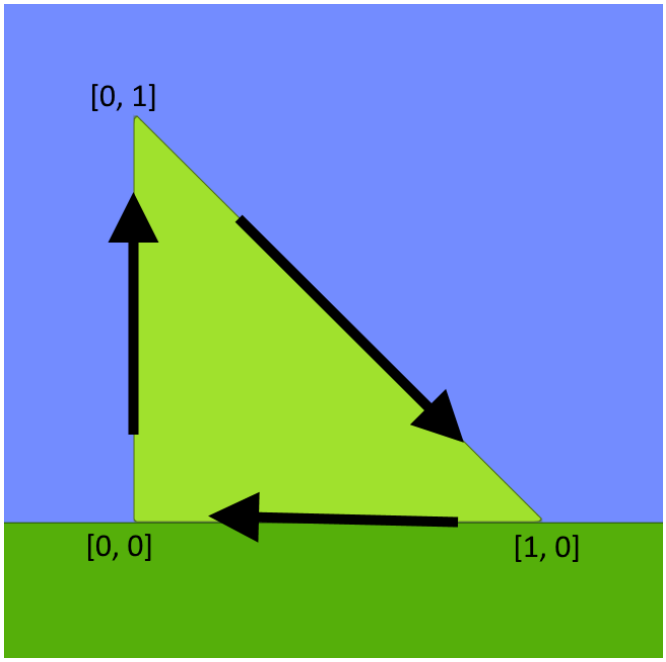

In this example, Algodoo was expecting to find a list, but instead found `0`.

In the second case, the error thrown states that it got a list that was too long. This means that the whole list of surfaces was accidentally inserted inside another list.

```
> Scene.addPolygon({
>     surfaces := [[[[0, 0], [0, 1], [1, 0]]]];
> });
 1220318 ms: - WARNING - Failed to evaluate: Scene.addPolygon({
 1220318 ms:     surfaces := [[[[0, 0], [0, 1], [1, 0]]]]
 1220318 ms: }), Expected vector of length 2, got list of length 3
error
```

In this example, Algodoo was expecting to receive a list of two floats/integers, but got a list of three lists instead (`[[0, 0], [0, 1], [1, 0]]`).

Analysing the triangle:

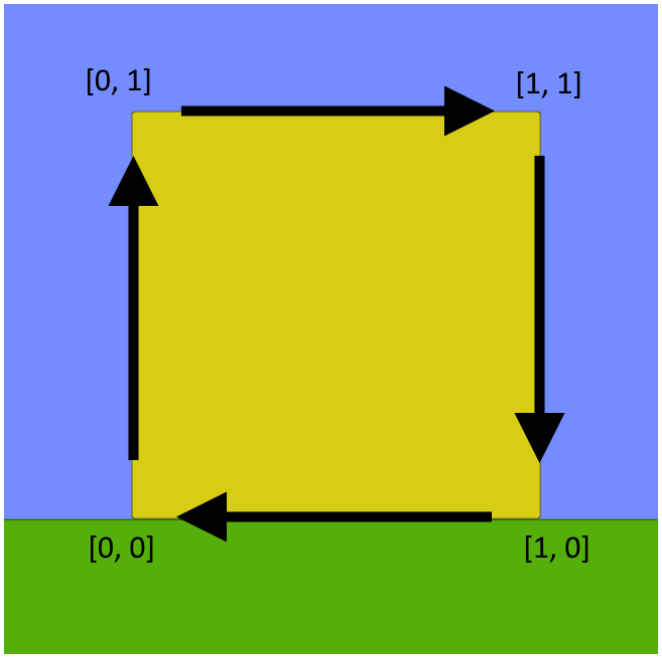



Algodoo takes the first vertex (`[0, 0]`), draws a line to the second vertex (`[0, 1]`), draws another line from the second vertex to the third vertex (`[1, 0]`) then, seeing that the third vertex is the last one, draws a line back to the first vertex.

You can extend the script above to create a square by simply adding the vertex `[1, 1]`:

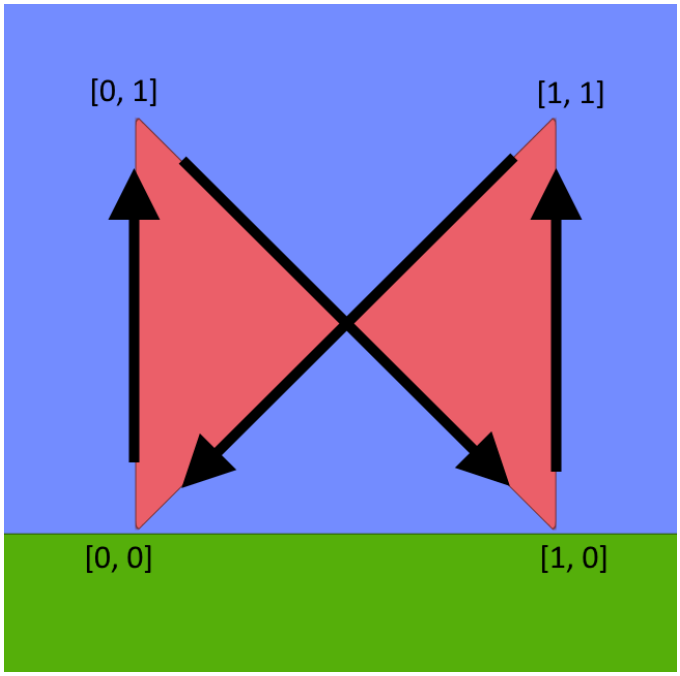
```
Scene.addPolygon({
    surfaces := [[[0, 0], [0, 1], [1, 1], [1, 0]]];
});
```

The order of the vertices matters. This diagram below shows Algodoo's drawing process for the above square:




If you change the order of the vertices to the following, it will not produce a box:

```
Scene.addPolygon({
    surfaces := [[[0, 0], [0, 1], [1, 0], [1, 1]]];
});
```



Now, because these vertices themselves are position variables, you don't actually need to define a `pos` variable to place the polygon in the correct place, assuming the vertices have been set up to put those points in the correct positions as well.
However, once the polygon is created, Algodoo does modify the `surfaces` variable to center it at the origin. The polygon will still be in the position you set it via the vertices, as Algodoo figures out the corresponding `pos` as well, but it does mean that a pre-generated polygon cannot have its surfaces used to set the position of a resulting polygon. Only manually placed or programmatically generated points in `surfaces` can be used to replace `pos`.

## Generate polygon vertices

The thyme.cfg file contains a function that simplifies the creation of regular polygons (which has been altered to make more readable):

```
Scene.my.generatePolygonVertices = (radius, sides)=>{
    outputList := [];
    for(sides, (i)=>{
        angle = (2 * math.pi * i) / sides;
        outputList = outputList ++ [radius * [math.cos(angle),
math.sin(angle)]];
    });
    outputList;
};
```

The function takes a 'radius' (the distance between the centre of the polygon and each vertex) and the number of sides of the polygon.

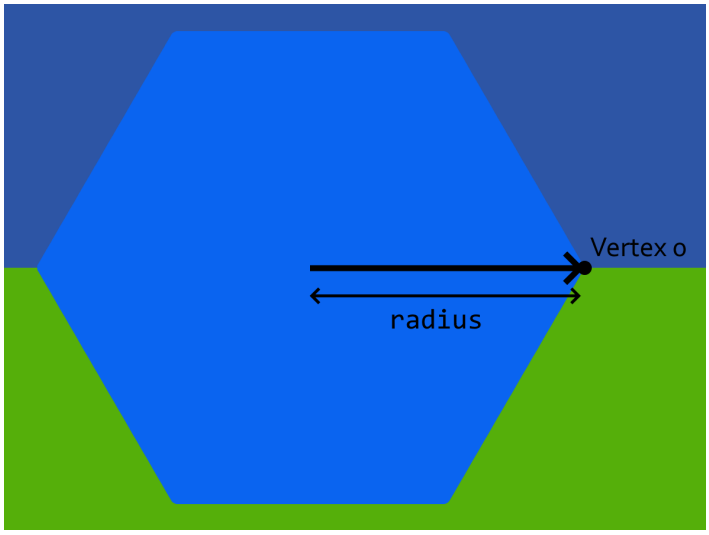
Here is code that creates a regular hexagon using the above function:

```
Scene.addPolygon({
    surfaces := [Scene.my.generatePolygonVertices(1, 6)];
});
```

This is the resulting hexagon:

An explanation of what the function does:

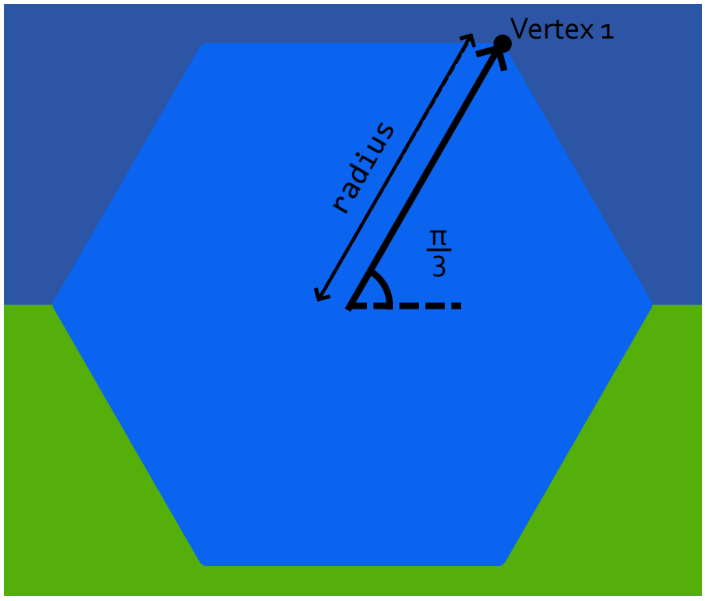
- In the first iteration of the for loop, `i = 0`.
- The code calculates the angle between the positive x axis and a line to the current vertex. Since `i` is `0`, `2 * math.pi * i / sides` is also `0`.
- The code then works out the vertex's cartesian (x and y) coordinates using the radius and the angle (the x coordinate is `radius * math.cos(angle)` and the y coordinate is `radius * math.sin(angle)`). The vertex (in this case, `[1, 0]`) is then added to the output list.



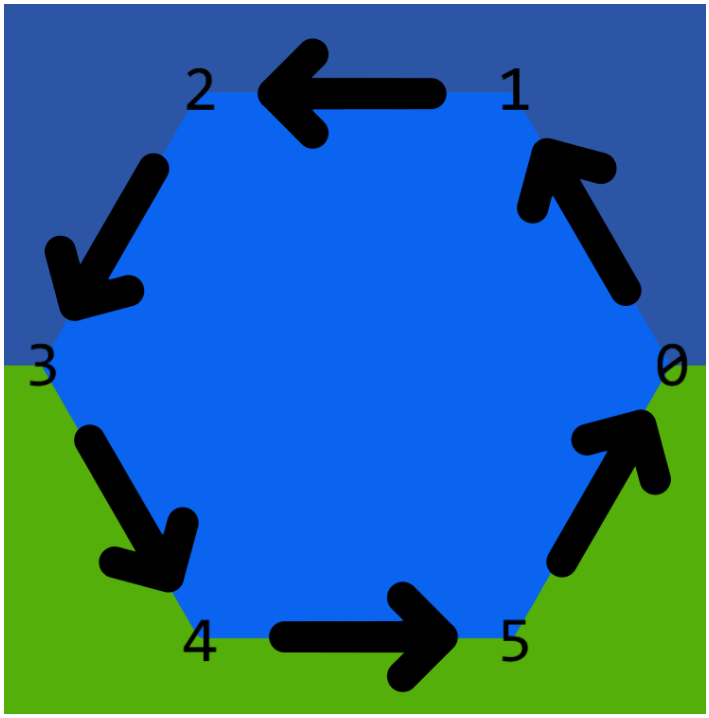

*The vertex in the diagram below is named 'Vertex 0' as it is the first element in the list of vertices.*

- In the second iteration of the for loop, i = 1.
- The code calculates the new angle, now π/3 (60°).

- The code then uses the radius and the angle to calculate the vertex coordinates and adds the new vertex to the list (`[0.5, 0.86602545]`).




This process is repeated until each vertex has been created. Algodoo then draws the polygon with the generated vertices.



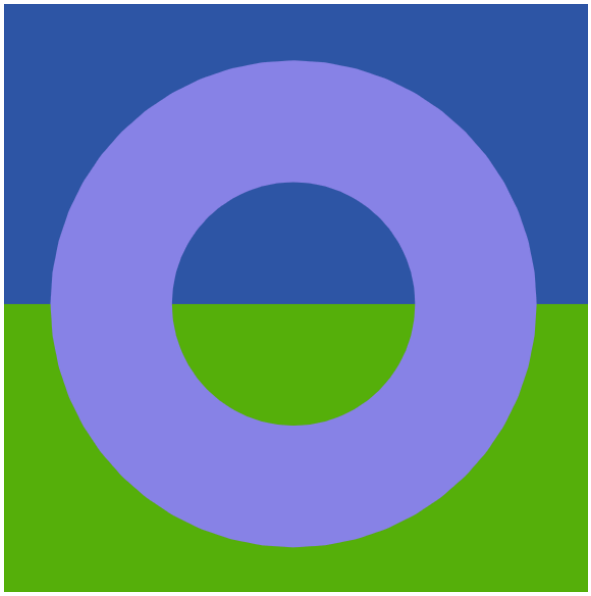

*Creating a regular polygon with 48 or more sides will create a polygon effectively indistinguishable from a circle.*

## Rings

You can also use the `generatePolygonVertices` function to create composite shapes. An example is a ring. The following code creates a circular ring by generating the vertices for a circle and the vertices for another smaller circle. While the function generates regular polygons rather than circles, a regular polygon with many sides is a very good approximation for a circle (which is the best you can get for a polygon in Algodoo as only the circle tool can generate perfect circles anyway).

```
Scene.addPolygon({
    surfaces := [Scene.my.generatePolygonVertices(1, 48),
Scene.my.generatePolygonVertices(0.5, 48)],
});
```

The ring should look like this:



## The 'polyTrans' variable

The `polyTrans` variable is a list of four floats/integers that represents a 2x2 transformation matrix. Like `surfaces`, it is read-only and, like `surfaces`, can only be accessed through the `readable` function.

Taking our right-angled triangle example from above:

```
Scene.addPolygon({
```

```
    surfaces := [[[0, 0], [0, 1], [1, 0]]];
});
```

We can scale a polygon by x times using the following transformation matrix:

$$\begin{pmatrix} x & 0 \\ 0 & x \end{pmatrix}$$

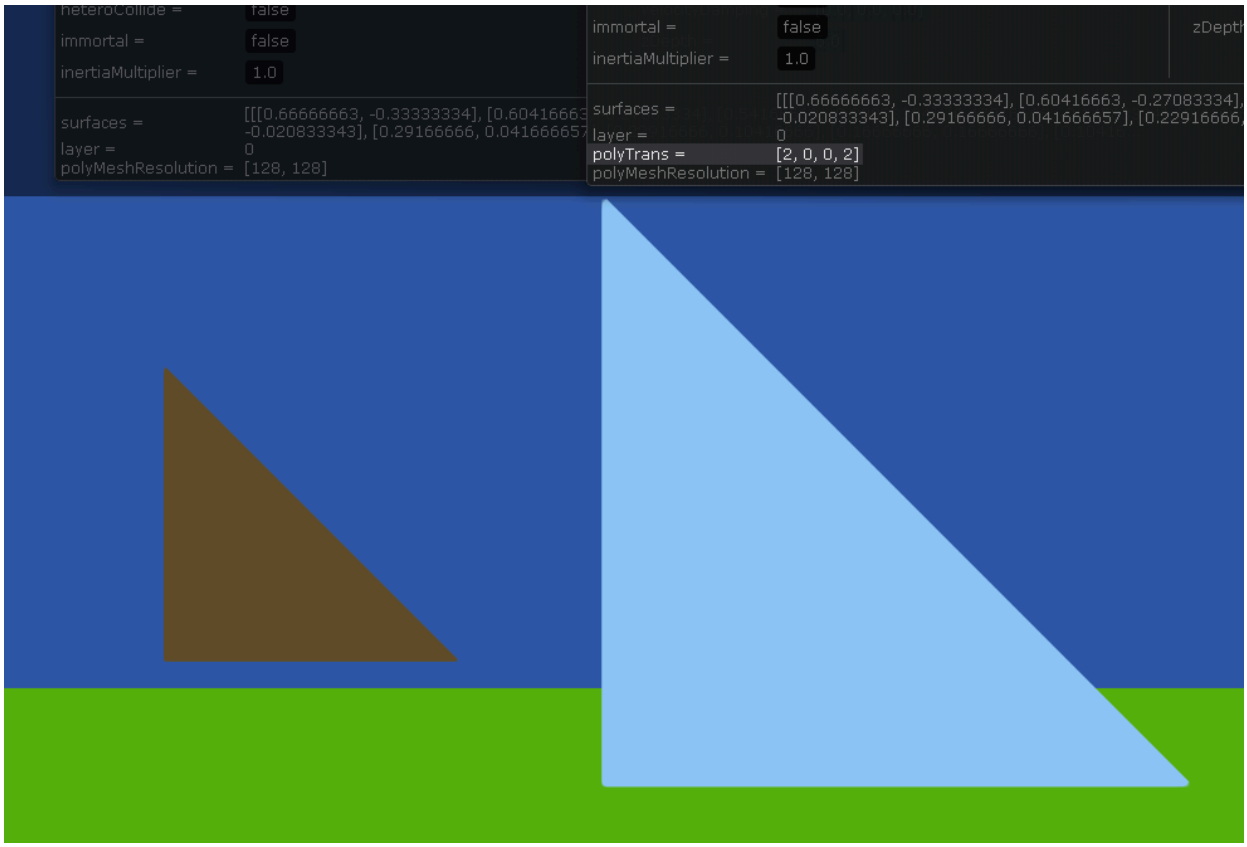
To scale a polygon by 2, for example, we would use the following matrix:

$$\begin{pmatrix} 2 & 0 \\ 0 & 2 \end{pmatrix}$$

In code:

```
Scene.addPolygon({
    polyTrans := [2, 0, 0, 2];
    surfaces := [[[0, 0], [0, 1], [1, 0]]];
});
```

Comparing the new polygon (right) to the old polygon (left), we can see that the new polygon is twice as large:




We can rotate a polygon by θ radians clockwise using the following transformation matrix:

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

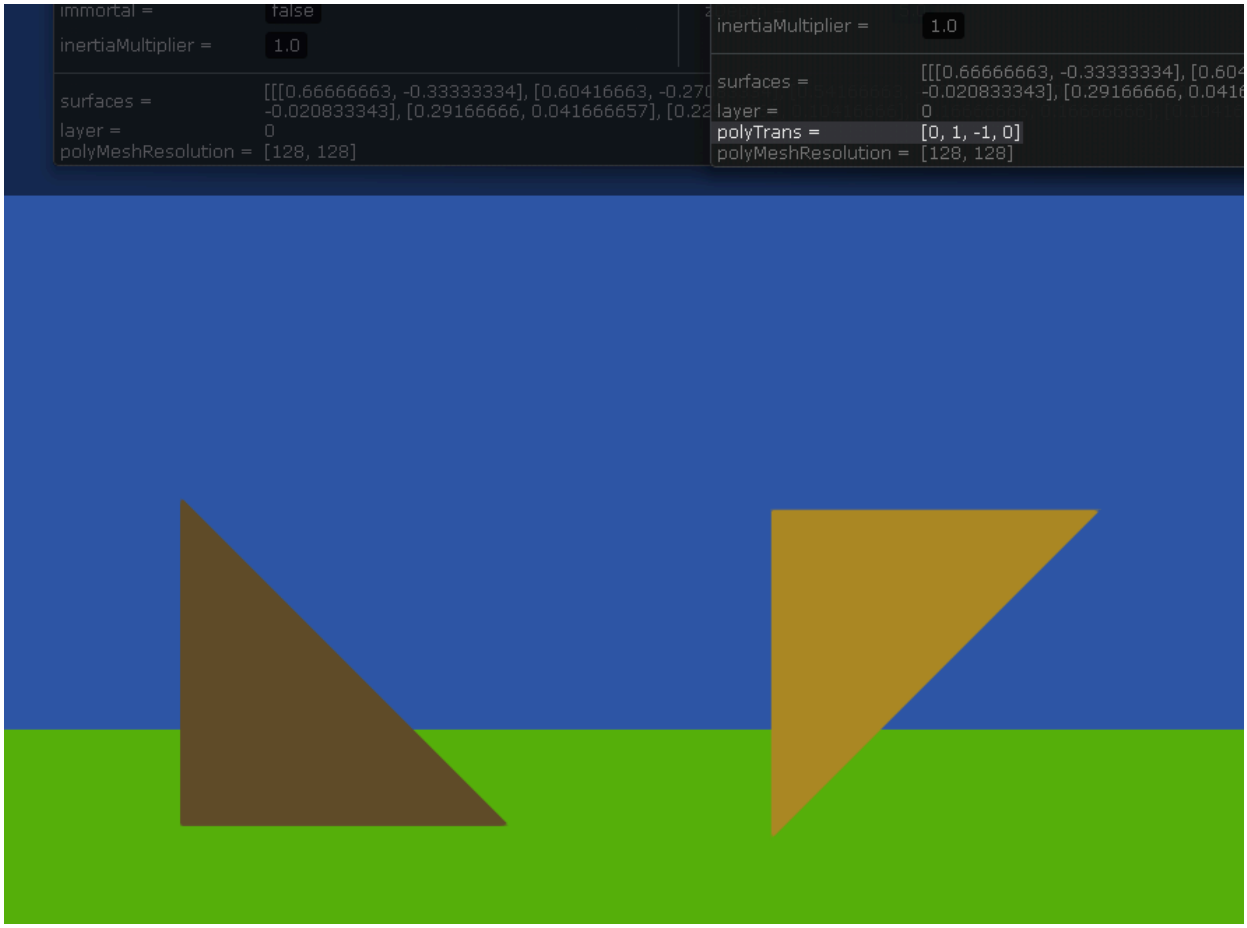
To rotate a polygon by π/2 (90°) clockwise, for example, we would use the following matrix:

$$\begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

In code:

```
Scene.addPolygon({
    polyTrans := [0, 1, -1, 0];
    surfaces := [[[0, 0], [0, 1], [1, 0]]];
});
```

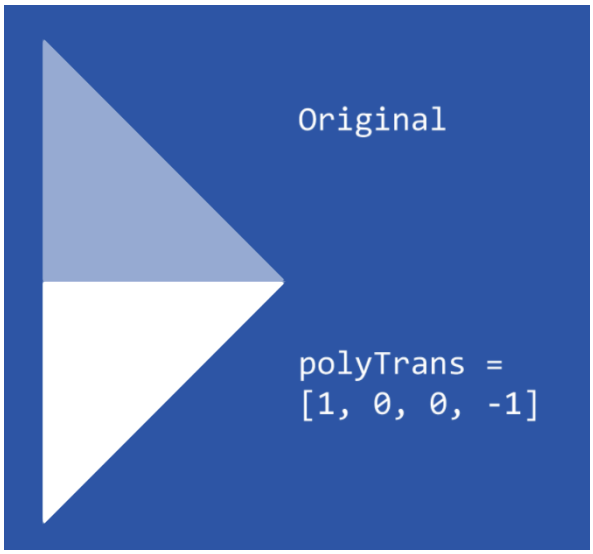
Comparing the new polygon (right) to the old polygon (left), we can see that the new polygon is the same as the old polygon but rotated π/2 (90˚) clockwise:




Here are the matrices for reflections:

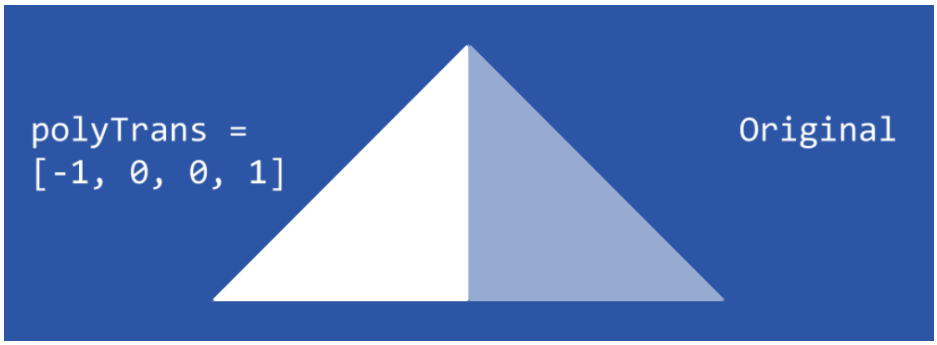
Reflection in the x axis:

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix}$$



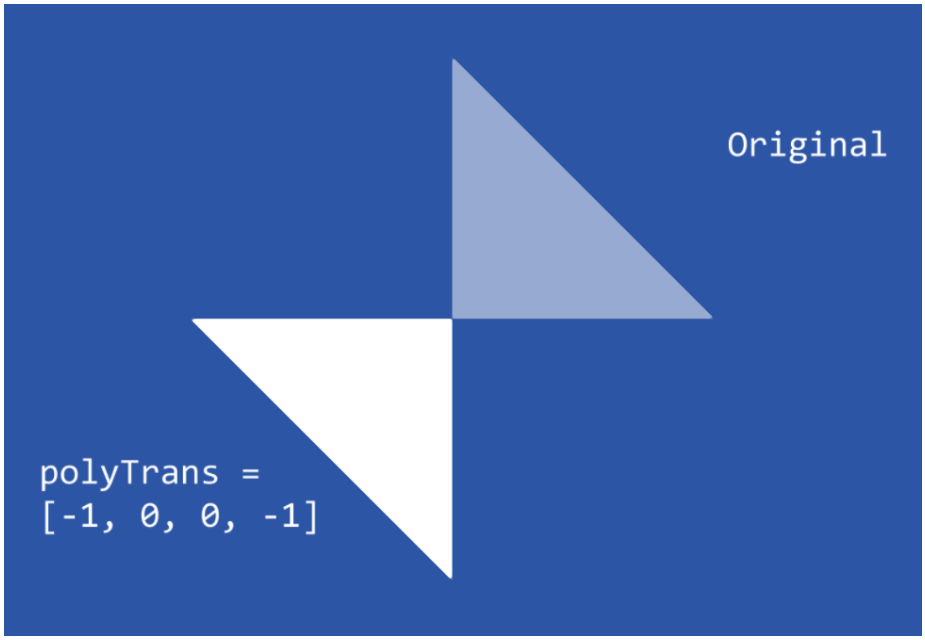

Reflection in the y axis:

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \end{pmatrix}$$




Reflection about the origin (i.e. in the line y = -x):

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix}$$




## More complicated shapes

If you want to spawn a polygon that is too complicated for the methods described above, you can create the polygon in Algodoo then copy and paste it into a script. Copying a polygon will copy a `Scene.addPolygon` script to your clipboard that will create the exact polygon you copied. Bear in mind that the `surfaces` variable tends to get very long for complicated polygons.

# Simulating anti-gravity

# Simulating anti-gravity

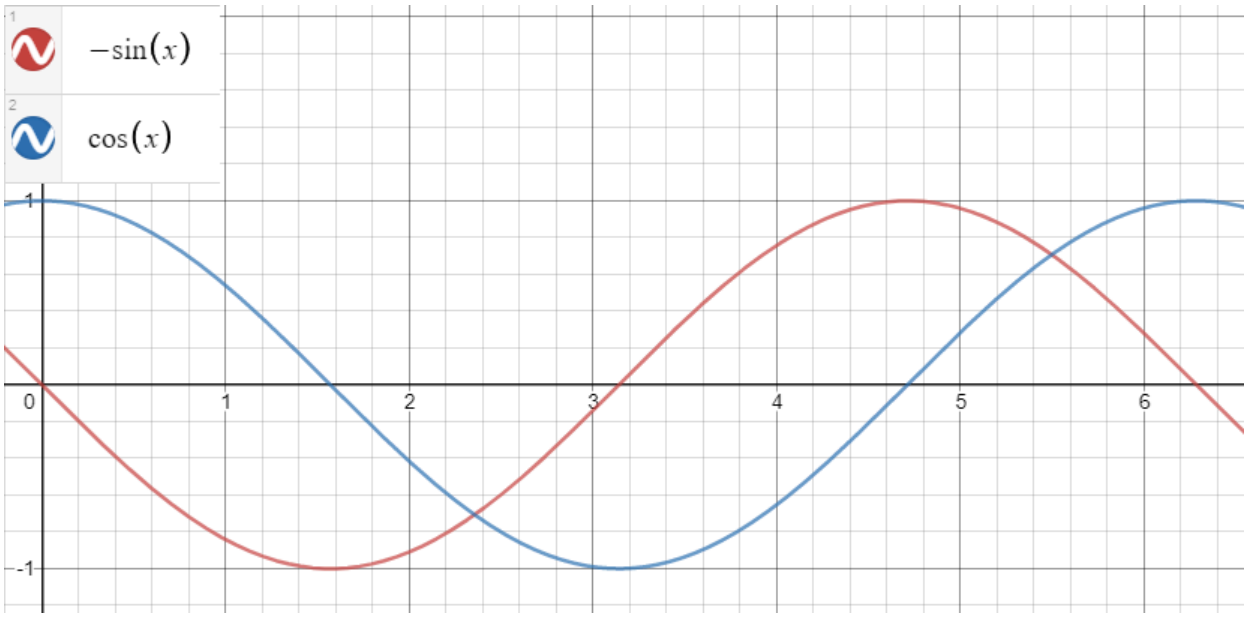
As stated in a previous section, to change the gravity of the entire scene, you would set `Sim.gravityAngleOffset` to the desired value. Giving individual objects an anti-gravity effect is more difficult, however.

First of all, we have to negate the current effects of gravity. The force of weight due to gravity causes an acceleration (by default 9.8 $ms^{-2}$ downwards), so we'll have to counteract that with an equal acceleration in the opposite direction.

To give an object constant acceleration, we can add to its velocity at a given time interval in `postStep`. Our time interval is `e.dt`. This value is the length of time in between `postStep` calls assuming no lag. It's equivalent to `1 / Sim.frequency`, where `Sim.frequency` is the number of simulation steps, or ticks, per second. The velocity we need to add is `Sim.gravityStrength`, which is in meters per **second** squared (when representing acceleration). Because the value for gravity strength is a representation of acceleration over one second, we need to correct it so it represents acceleration over a single tick. This is easily done with `Sim.gravityStrength * e.dt`.

```
postStep = (e)=>{
    vel = vel + [ - math.sin(Sim.gravityAngleOffset),
math.cos(Sim.gravityAngleOffset)] * Sim.gravityStrength * e.dt;
}
```

A little explanation of the trigonometry:

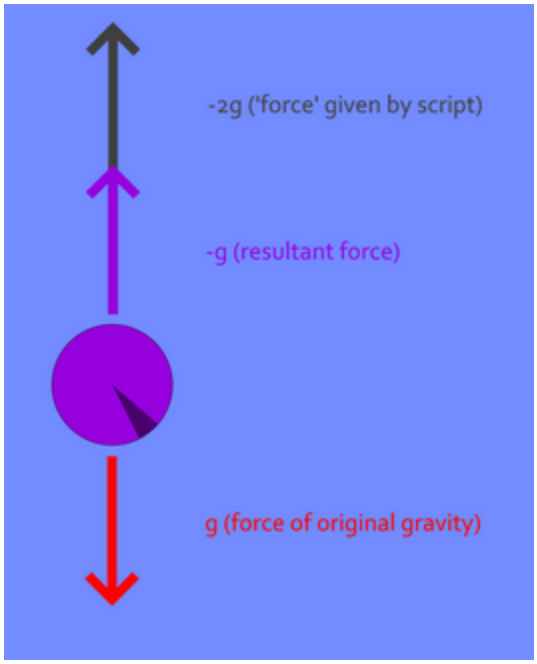



As you might be able to tell, $x$ represents `Sim.gravityAngleOffset`, $-\ sin(x)$ represents the horizontal velocity we need to counteract gravity and $cos(x)$ represents the vertical velocity we need to counteract gravity.

- At $x\ =\ 0$, gravity points downwards, so the counteracting acceleration should have no horizontal component ($-\ sin(x)\ =\ 0$) and a vertical component pointing upwards ( $cos(x)\ =\ 1$).
- At $x\ =\ \frac{\pi}{2}$, gravity points to the right, so the counteracting acceleration should have a horizontal component pointing to the left ($-\ sin(x)\ =\ -\ 1$) and no vertical component ($cos(x)\ =\ 0$).
- At $x\ =\ \pi$, gravity points upwards, so the counteracting acceleration should have no horizontal component ($-\ sin(x)\ =\ 0$) and a vertical component pointing downwards ($cos(x)\ =\ -\ 1$).
- At $x\ =\ \frac{3\pi}{2}$, gravity points to the left, so the counteracting acceleration should have a horizontal component pointing to the right ($-\ sin(x)\ =\ 1$) and no vertical component ($cos(x)\ =\ 0$).

*You may notice that the object continues to move a bit after entering the gravity script. This happens as Algodoo performs physics calculations before running scripts. Algodoo uses the object's resultant force to calculate the new velocity of the object, then uses that velocity to calculate the object's new position. Air resistance also comes into play here, which makes things far more complicated. To get a perfect lack of movement, you should turn off air resistance and set the velocity so that the object would be moving away from the direction of gravity with a speed equivalent to gravity's acceleration over a single tick. With default values, this would be a speed of 9.8/60 m/s.*

To give an individual object an anti-gravity effect (i.e. the object's gravity is in the opposite direction to the scene's gravity), we simply multiply by 2:




```
postStep = (e)=>{
    vel = vel + [ - math.sin(Sim.gravityAngleOffset),
math.cos(Sim.gravityAngleOffset)] * Sim.gravityStrength * 2 * e.dt;
}
```

Our original script counteracted gravity by adding an acceleration that was equal and opposite to the scene's gravity, making the total acceleration zero. By adding that acceleration again, the total acceleration on the object will be equal and opposite to gravity.

To give an individual object a gravity effect in any direction, we need to counteract gravity and add the appropriate acceleration of the new gravity effect (where `newGravityAngle` is the direction of your gravity effect):

```
postStep = (e)=>{
    vel = vel + ([ - math.sin(Sim.gravityAngleOffset),
math.cos(Sim.gravityAngleOffset)] + [math.cos(newGravityAngle),
math.sin(newGravityAngle)]) * Sim.gravityStrength * e.dt;
}
```

The above functions can be simplified if the direction of gravity in the scene is known to not change. In the case of gravity pointing downwards (default direction), the simple anti-gravity script can be simplified to this:

```
postStep = (e)=>{
    vel = vel + [0, Sim.gravityStrength * 2 * e.dt];
}
```

Likewise, the function for any direction of gravity can be simplified to this:

```
postStep = (e)=>{
    vel = vel + ([0, 1] + [math.cos(newGravityAngle),
math.sin(newGravityAngle)]) * Sim.gravityStrength * e.dt;
}
```

These simplified functions should only be used if the direction of gravity is known to be both unchanging and pointed straight down.

*You may find that other people use Sim.frequency instead of e.dt in their gravity scripts. Sim.frequency and e.dt are the inverse of each other, so dividing by Sim.frequency leads to the same outcome as multiplying by e.dt. Sim.frequency is used as that was and is still more well-known than e.dt. Using e.dt is recommended though as Sim.frequency has a greater impact on performance than e.dt. However, forewarning, you cannot use e.dt outside of postStep, as e.dt is intrinsic to postStep. Technically it exists in update as well, but its behavior there is different. Use e.dt for any script within postStep (or in any function called by postStep that takes e as an argument), and use Sim.frequency everywhere else.*

# Creating trigger hitboxes

# Creating trigger hitboxes

By default, the `onCollide` function is rather limited. It's only called upon an actual initial collision, and there's no built-in way to actually test if an object enters another object's hitbox without colliding. However, through the use of negative restitution, it is possible to almost perfectly create what is known as a trigger.

Triggers are commonly used in video games to activate things and begin events whenever a player or some other entity enters an area. To put it simply, a trigger is an area that if entered will cause something to happen. In the vast majority of situations, entities never directly interact with triggers.

To create a trigger, simply set its restitution to a negative value. Ideally this negative value should be large, something like `-9999` would work well. Then, the trigger needs its `onCollide` function set so that it either changes any colliding object's collision layers, or it changes its own collision layers, so that the collision between the trigger and an object can only happen once. This is needed because Algodoo handles simple collisions and objects overlapping differently, and restitution stops being the only factor when objects are overlapping rather than simply colliding.

Here's a simple example: You create a trigger and put it on collision layer A. Then you can put this code in its `onCollide` function so that it changes any other object that collides with it to layer B.

```
(e)=>{
    e.other.collideSet = 2;
}
```

If done correctly, objects should pass through the trigger as if they never collided, but any additional code you may stick into `onCollide` will have been run once.

There are some limitations to this method. The most obvious limitation is that the trigger does have to change the collision layers of the object, which may be undesirable in some circumstances. This collision layer change also means that the trigger can only be activated once, but good placement of triggers can help fix that issue.

Another issue that you might run into is that triggers don't work for objects with a restitution of `0`. In fact, restitutions close to `0` in general don't work as well with triggers. As an object's

restitution gets close to `0`, the negative restitution needed for the trigger to function gets greater and greater. This is why it's recommended that the negative restitution is set to be large. At `0`, the needed negative restitution would theoretically be `-inf`, but this doesn't work as in this case, objects with `0` restitution just get deleted due to collision force calculations breaking and setting a velocity value of `[NaN, NaN]`, an invalid value. Non-zero restitution values actually do work with negative infinite restitution, but as to not accidentally delete zero restitution objects it's safer to not use negative infinite restitution.

One last issue that may be encountered is that collisions have different behavior if the two objects start off overlapping. This could be achieved by an object starting off inside the trigger, or the object's collision layer is changed to be back to the trigger's collision layer while still inside the trigger. In both of these situations, the object's velocity may be affected as restitution is no longer the only factor in how the object's velocity is affected by the collision, as Algodoo wants to prevent two colliding objects from overlapping.

However, these downsides are far outweighed by the benefits of using triggers. The two other alternatives for triggers are simply testing if the position is within a certain area, and lasers. The former method is very inefficient, inflexible, only easily allows for simple shapes, and causes lag if used too much. The latter isn't great either as objects can block lasers, lasers can be difficult to hide, only work well as single lines, and they run on frame rate rather than tick rate, which can lead to being unreliable regardless of implementation. Triggers on the other hand are very performance friendly, can be any shape, and are extremely reliable if implemented well.

# Editing scene files

# Editing scene files

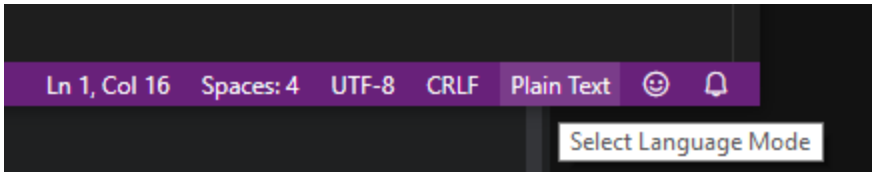
## Text editors

If you enter an incorrect script into a variable, Algodoo will replace it with its last known value. This can get rather annoying as your incorrect script (which you may have been working on for a while) is removed completely.

This problem can be avoided if you edit your scripts in an external program. I personally use Visual Studio Code (https://code.visualstudio.com/), although you are free to use other editors if you prefer.

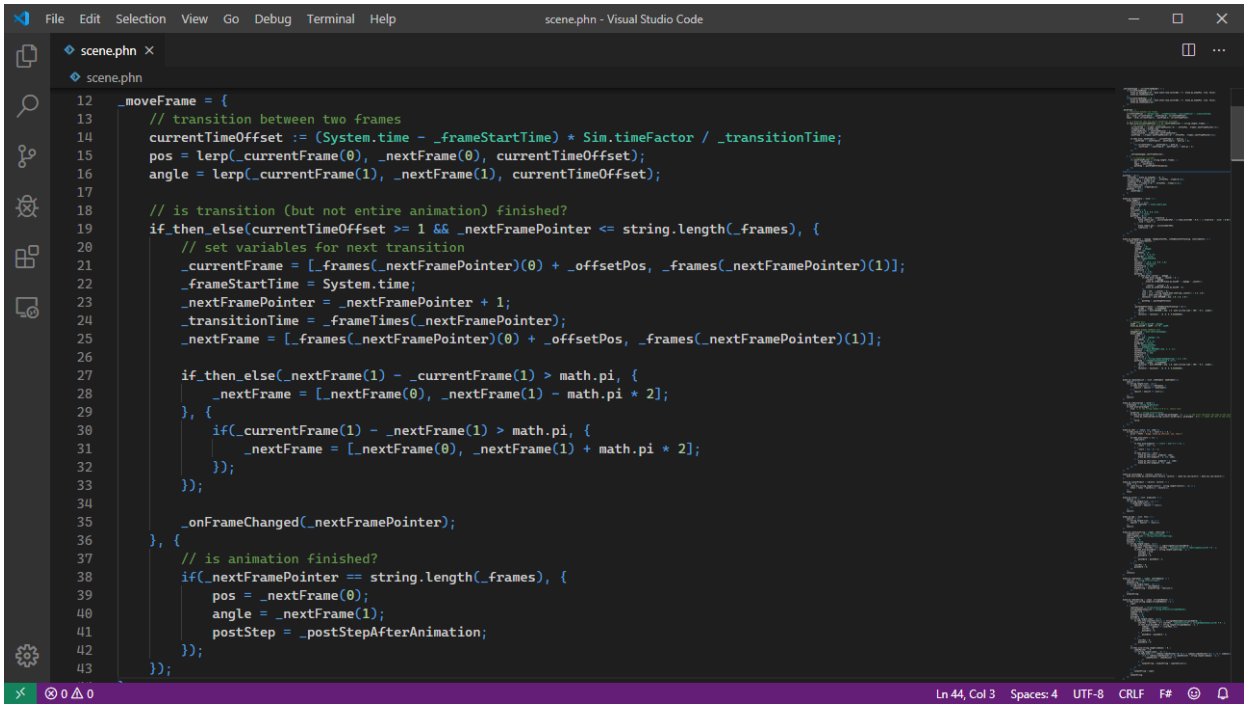
Some editors provide syntax highlighting (highlighting different parts of the code with different colors). This makes the code more readable and easier to follow.

If using Visual Studio Code, you can set the language's syntax highlighting by clicking the third option from the right.



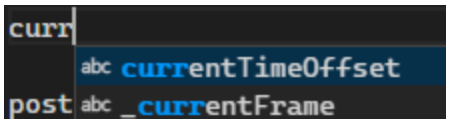

This will bring up a list of all the languages you can use for syntax highlighting. Thyme is not on the list, so feel free to use a language with syntax highlighting rules that you think are appropriate. I personally use F#'s rules.
*Notepad++ allows you to create your own syntax highlighting rules. It is only available on Windows, however.*

This is an example of Thyme code highlighted with F#'s rules in Visual Studio Code:
*Note that F# is not Thyme. The two programming languages have little in common.*




*Don't worry about what any of this code actually means! The Thyme guide will help you understand it.*

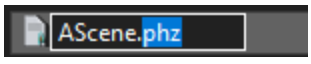
Some editors even provide tab-completion (just as the console does) - start typing an identifier and you can press tab to finish it.

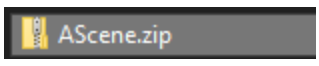



*When saving a Thyme file, I personally use the .phn extension as Algodoo scene files (**not** packages) are written in Thyme and have the .phn extension. There is more information on this in the 'Editing .phz and .phn files' section.*

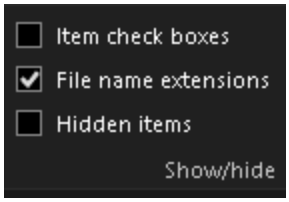
# Editing .phz and .phn files

Algodoo scene package files (.phz files) are actually compressed archives. They can be converted into .zip files by changing the file extension (i.e. from .phz to .zip):



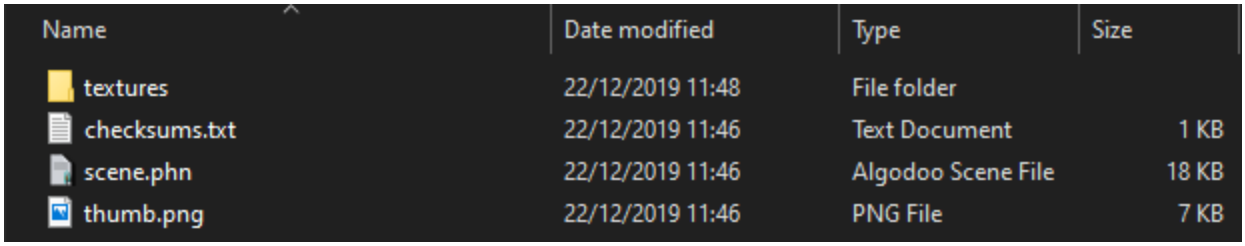




Make sure you have File name extensions shown, this can be done in the View tab of File Explorer.



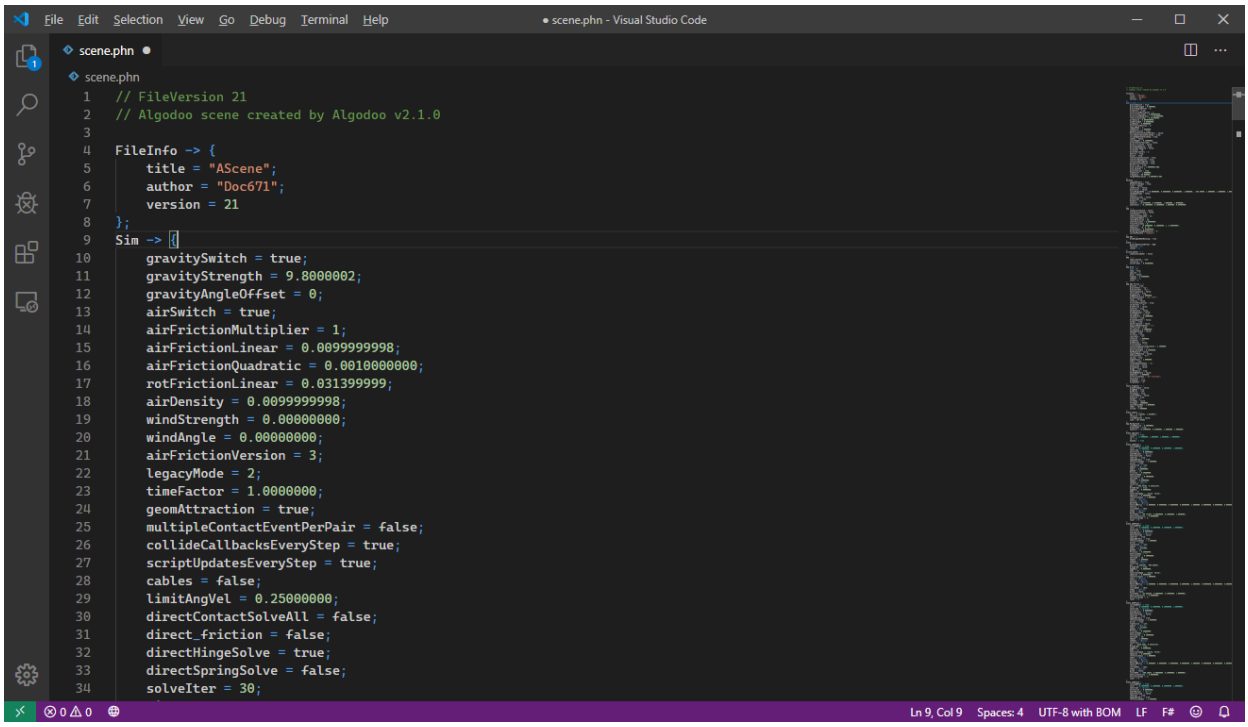

*The screenshots shown are taken on Windows 10, but the same principle applies to all operating systems.*

When you extract the .zip file into a new folder and go into the folder, you'll see three files and a folder:




- **textures** - a folder storing all of the scene's textures.
- **checksums.txt** - a text file containing the name of every other file alongside a hexadecimal checksum. Algodoo uses this to check whether the scene has been tampered with - if it has, an error will appear in the console when you load the scene and you won't be able to upload the scene to Algobox. The scene will work normally otherwise.
- **scene.phn** - a Thyme file containing information about the scene - including information about every object.
- **thumb.png** - the scene's thumbnail.

You can open scene.phn in a text editor. You should see something like this:




The `FileInfo` object contains information about the scene:

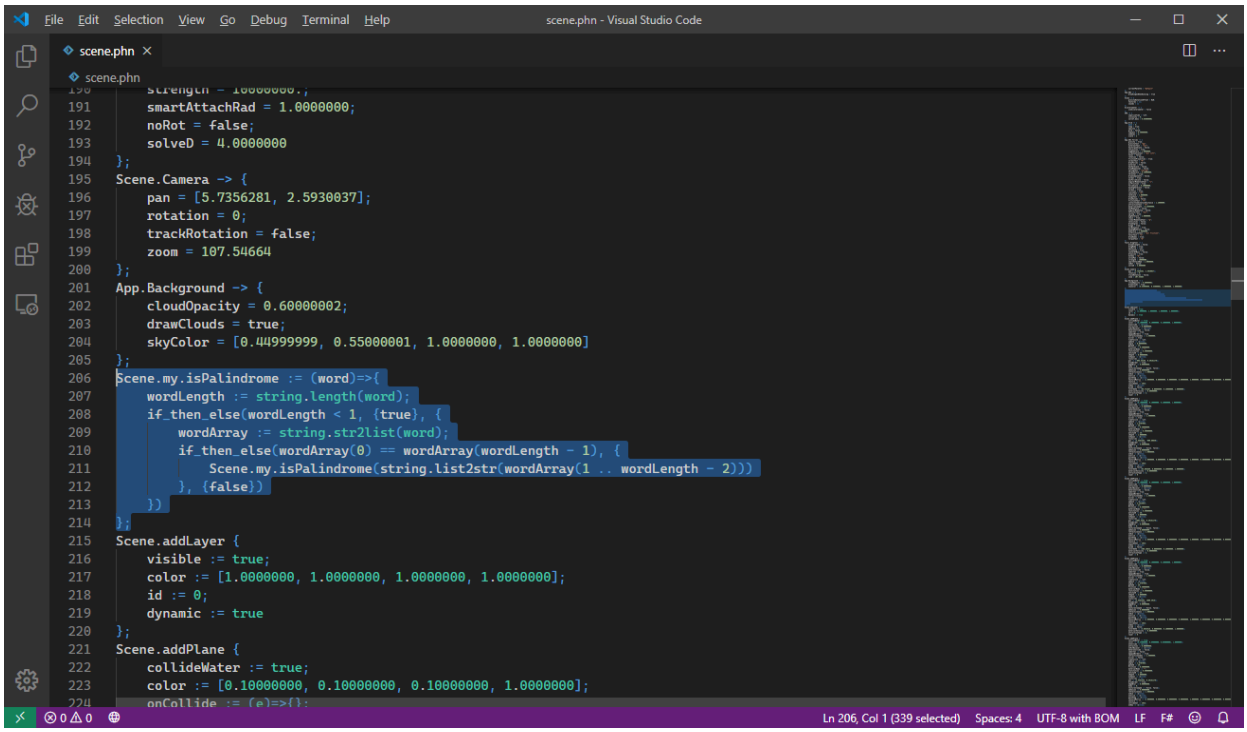
| Property | Data type | Description |
|---|---|---|
| `FileInfo.algoboxID` | `int` | The scene's ID on Algodoo's file-sharing platform Algobox. |
| `FileInfo.author` | `string` | The author of the scene. |
| `FileInfo.description` | `string` | The description of the scene. |
| `FileInfo.standardAuthor` | `string` | The default value of `FileInfo.author` for new scenes.<br><br>*This is never saved with the scene, but can be accessed from the console.* |
| `FileInfo.title` | `string` | The title of the scene. |
| `FileInfo.version` | `int` | The version of Algodoo the scene is created in (e.g. if the scene were created in version 2.1, the value of this property would be `21`). |

All of these properties can be accessed from the console.

Several other objects are also saved with the scene:

- `App` - contains information about the features of the application Algodoo.
- `App.Background` - contains basic properties for the scene's background, including the background's color, whether clouds are shown and their opacity.
- `App.Grid` - contains properties for the scene's grid, including the number of axes, the scale of the grid and how transparent it is.
- `App.GUI` - contains properties about Algodoo's UI, such as whether to draw axles.
- `App.GUI.Forces` - contains properties about visualising forces.
- `Accelerometer` - contains a single property - whether to display the accelerometer or not. This does not seem to have an effect.
- `Palette` - contains properties about new geometries that are drawn with the circle, box and polygon tools.
- `Scene` - contains information included in the scene such as textures, sounds and its gravity rotation offset.
- `Scene.Camera` - contains general properties for the scene's 'view', such as how much the scene is zoomed in, or where the 'camera' is placed in the scene.
- `Sim` - contains general information about how to simulate the scene, including the simulation speed and gravity properties.
- `SPH` - short for 'smoothed-particle hydrodynamics'. Contains properties for the scene's water, such as the size of each particle, the amount of time particles die after no contact with its neighbours and the viscosity of each particle.
- `Tools.DragTool` - contains information about the scene's drag tool, including the strength of the drag, whether it must drag the center of an object and whether the drag force is shown.

If you've declared any variables in the `Scene.my` object, you'll notice that they are saved too:

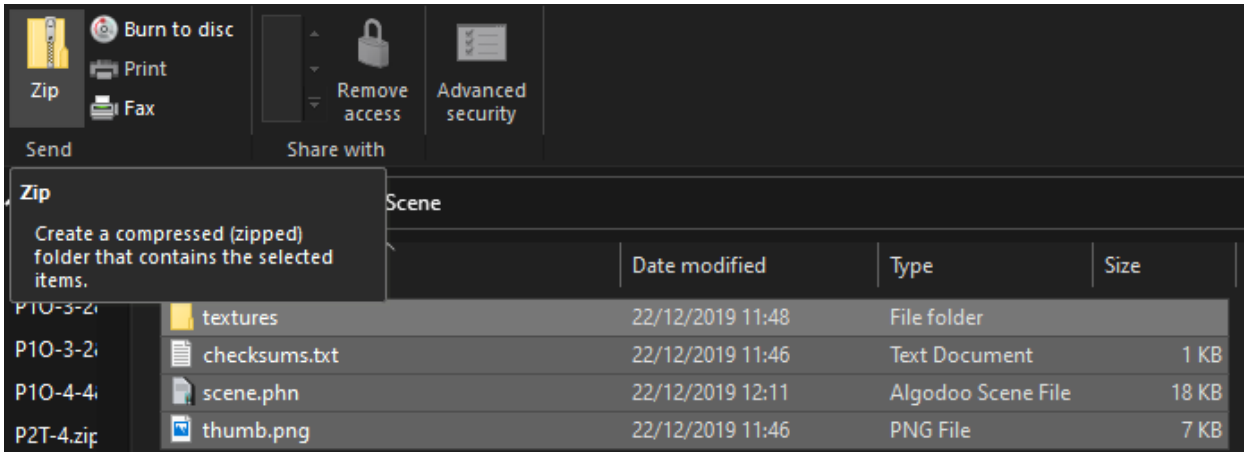



Finally, you'll notice a lot of object creation functions (`Scene.addBox`, `Scene.addCircle`, etc.). This is how objects are stored in Algodoo scenes. You'll also notice that every single property is saved here - when you use the object creation functions in scripts, you only specify a few properties - the rest are set to their default values.

*When you cut or copy any object in Algodoo, it cuts/copies the relevant object creation function with all of the object's properties to your clipboard. You can use this to edit objects individually without needing to unpack the entire scene.*

You can edit the scene in an external text editor. Make sure that all your code is valid Thyme (otherwise the scene may not open properly) before saving.

To package your scene back together, put all of the scene's components into a .zip file:




Change the new .zip file's extension to .phz, and you will be able to open it in Algodoo again!

*Note that .phn files can be opened in Algodoo, but any textures the scene requires that aren't in your textures folder won't load.*

# Writing add-ons

# Writing add-ons

Algodoo scenes can communicate with external programs via file I/O. An example of an add-on is AlgoSound, created by Steve Noonan.
http://www.algodoo.com/forum/viewtopic.php?f=13&t=11738&p=85506#p85506.
(**Make sure HTTPS-Only mode is disabled on your browser when visiting this site**.)

AlgoSound works like this:

- When you want to play a sound, you call Scene.my.playSound with the appropriate arguments.
- Scene.my.playSound writes the appropriate information to a file.
- The external program reads the file, seeing the information. It then uses the information to play the requested sound.

This concept could be used to do other things with Algodoo that are outside Thyme's capabilities.

For example, a multiplayer game in Algodoo could work like this:

- Player 1 moves their character.
- A Thyme script writes Player 1's position to a file.
- An external program on Player 1's computer sees that the file has been changed and reads the file.
- The external program sends Player 1's position to a web server.
- The external program on Player 2's computer then receives Player 1's position from the web server.
- The external program writes Player 1's position to a file.
- A Thyme script on Player 2's computer reads the file and sets Player 1's position to the position stated in the file.

An automatic fan marble race signup system could work like this:

- A YouTube bot is hosted on the host's computer (the host being the one doing the FMR livestream).
- When a YouTube user wants to sign up, they enter a bot command in the livestream chat.
- The bot detects the command and checks whether the signup is valid. *Has this user already signed up? Has this marble been taken?*
- If the signup is valid, the bot writes the appropriate data to a file on the host's computer.

- A Thyme script sees that the file has been changed and reads the file.
- The Thyme script updates the scene to show that the user has signed up.

Writing an add-on of your own requires you to know another programming language with file I/O capabilities.

*Add-ons are often incorrectly called 'mods'. They are not mods as they do not modify Algodoo itself. It is very, very difficult to create a true Algodoo mod as Algodoo was written in C++, which is compiled to assembly code (very close to machine code that the CPU runs). Decompiling assembly code will result in code that is very hard to understand - assembly has no concept of variable names, meaning that the decompiler must come up with them itself. Mods are easier to create with games such as Minecraft (Java edition only) or Terraria (desktop version only - written in C#) as Java and C# compilers preserve a significant amount of the information in the original code. The resulting decompiled code is much easier to understand.*

## Example add-on: Text Relay

This add-on will take a message from the console (of the external program) and display that message on a box in Algodoo.

This is what will happen:

- When you write a message into the console, the program will write the message to a file.
- A script inside Algodoo will periodically check the file for any changes. Once it sees a new message, the script will then change the box's text property to the message.

This tutorial is done in three languages: C#, Python and JavaScript (with Node.js). Each language section assumes that you are already familiar with the basics of programming in each of these languages.

First, create a plain text file (this tutorial will call it 'communication.txt') in Algodoo's home directory. Algodoo's home directory should be:

- Documents\Algodoo on Windows.
- Library/Application Support/Algodoo on MacOS.
- ~/Algodoo on Linux.

This file will be used to allow your program to communicate with Algodoo.

Move to the external program section of your preferred programming language. After finishing the external program, move to the Thyme script section.

## C# program

Create a new **Console Application** project.

- Using Visual Studio on Windows: https://docs.microsoft.com/en-us/dotnet/core/tutorials/with-visual-studio?tabs=csharp
- Using Visual Studio for Mac: https://docs.microsoft.com/en-us/dotnet/core/tutorials/using-on-mac-vs
- Using the .NET Core CLI: https://docs.microsoft.com/en-us/dotnet/core/tutorials/cli-create-console-app

Declare a constant string storing the path to the communication file.

```
// Windows example
const string targetFileDirectory =
@"C:\Users\Username\Documents\Algodoo\communication.txt";
```

*On Windows, the directory separator character is a backslash* `\` *rather than a forward slash* `/` *as it is on other operating systems though it's accepted when you manually type it. The backslash is an escape character, so the code above makes use of a* ***verbatim string literal****, which works the same in C# as it would in Thyme.*

Declare a variable that will store the message:

```
string message;
```

We will repeatedly ask the user to enter a message until they enter an empty string. To do this, we can use a do...while loop:

```
do
{
}
while (message != "");
```

Inside the do...while loop, ask the user to enter input and set the message variable to contain the contents of the input:

```
Console.Write("Enter message: ");
message = Console.ReadLine();
```

You will now need to write the message to a file. At the top of the code file, import the System.IO namespace (which contains classes for file I/O):

```
using System.IO;
```

Declare a new StreamWriter for the communication file:

```
using (var fileWriter = new StreamWriter(targetFileDirectory))
{
}
```

*Note that once a file stream is opened, it must be closed in order to allow other programs to access it. A using statement automatically closes the file stream with the closing curly bracket }.*

Inside the using statement, write the message to the file:

```
fileWriter.Write(message);
```

Finally, give the user confirmation that the message has been successfully written to the file:

```
Console.WriteLine($"The message '{message}' has been written! Please
wait for the message to appear!");
```

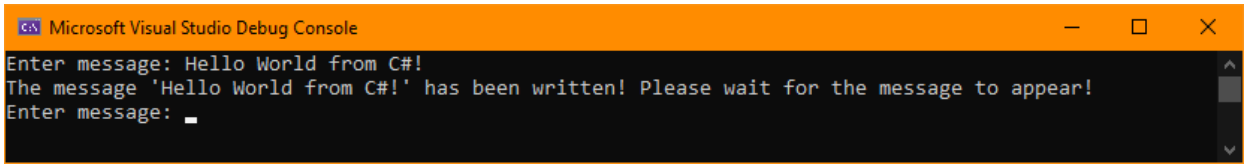
The full code should look like this:

```
using System;
using System.IO;

namespace TextRelay
{
    class Program
    {
        static void Main(string[] args)
        {
            const string targetFileDirectory =
@"C:\Users\Username\Documents\Algodoo\communication.txt";

            string message;
            do
            {
                Console.Write("Enter message: ");
                message = Console.ReadLine();

                using (var fileWriter = new StreamWriter(targetFileDirectory))
```

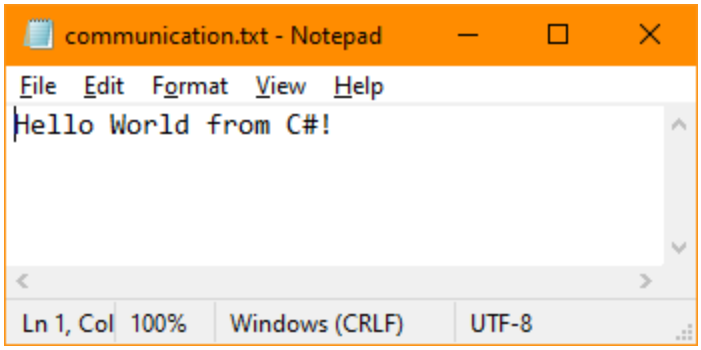
```
                {
                    fileWriter.Write(message);
                }

                Console.WriteLine($"The message '{message}' has been written! Please wait
for the message to appear!");
            }
            while (message != "");
        }
    }
}
```

When running the program, the console should ask you for input:


Enter a message and press enter. The console should tell you that the message has been written:




If you open the file, you should see that the message has been written:




If your program works, you can move onto the Thyme code.

## Python program

Create a new Python file in Algodoo's home directory:


Declare a variable that will store the message:

```
message = " "
```

We will repeatedly ask the user to enter a message until they enter an empty string (which is why the initial value of the message variable is not an empty string). To do this, use a while loop:

```
while message != "":
```

Inside the while loop, ask the user to enter input and set the message variable to contain the contents of the input:

```
message = input("Enter message: ")
```

Open the communication file for writing:

```
with open("communication.txt", "w") as communication_file:
```

*Note that once a file stream is opened, it must be closed in order to allow other programs to access it. A with statement automatically will automatically close the file stream.*

Inside the with statement, write the message to the communication file:

```
communication_file.write(message)
```

Finally, give the user confirmation that the message has been successfully written to the file:

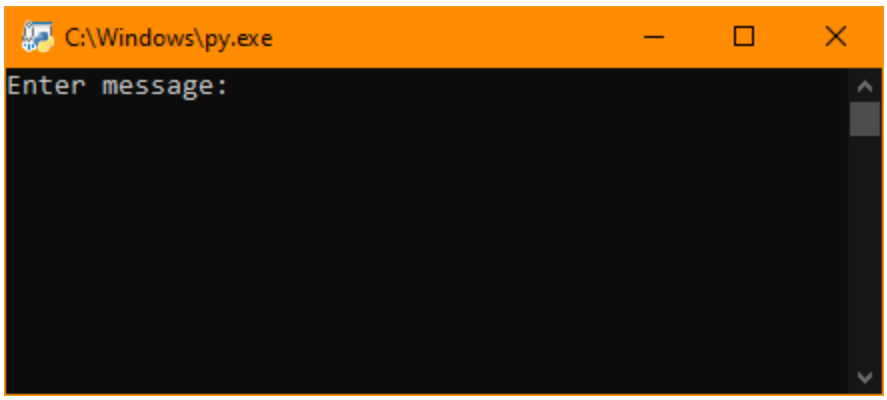
```
print(f"The message '{message}' has been written! Please wait for the message to appear!")
```

The full code should look like this:

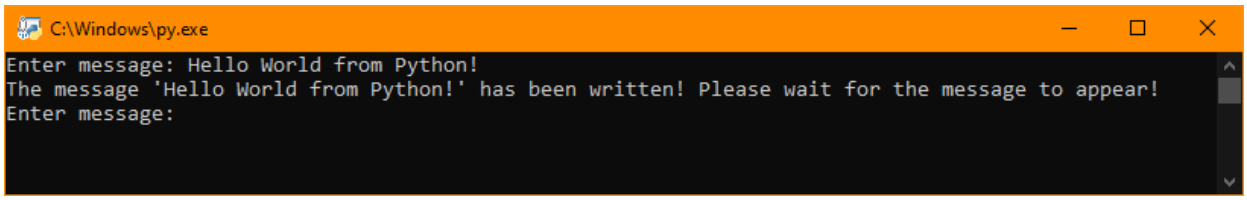
```
message = " "

while message != "":
```

```
    message = input("Enter message: ")

    with open("communication.txt", "w") as communication_file:
        communication_file.write(message)

    print(f"The message '{message}' has been written! Please wait for the message to
appear!")
```

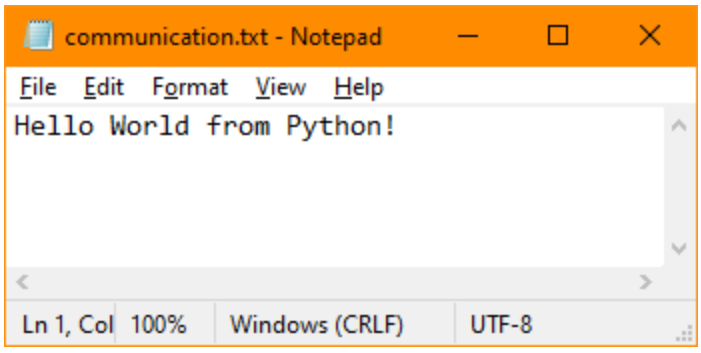
When running the program, the console should ask for input:




Enter a message and press enter. The console should tell you that the message has been written:




If you open the file, you should see that the message has been written:




If your program works, you can move onto the Thyme code.

## Node.js program

Create a new JavaScript file in Algodoo's home directory:


We need to import both the filesystem and readline modules:

```
const fs = require("fs");
const readline = require("readline");
```

To use the readline module, we need to create a readline interface:

```
const readlineInterface = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});
```

We will repeatedly ask the user to enter a message until they enter an empty string using a recursive function.

*As Node.js is single-threaded, we cannot use a synchronous loop to get console input. Instead, we must use a recursive function.*

Declare a function called `askForMessage`:

```
function askForMessage() {
}
```

Inside the `askForMessage` function, ask for the user to enter a message:

```
readlineInterface.question("Enter message: ", (message) => {
});
```

Inside the callback function, if the message isn't an empty string, write the message to the communications file:

```
if (message != "") {
    fs.writeFileSync("communication.txt", message);
}
```

Inside the if statement, give the user confirmation that the message has been successfully written to the file:

```
console.log(`The message '${message}' has been written! Please wait for the message to appear!`);
```

At the end of the if statement, call the `askForMessage` function:

```
askForMessage();
```

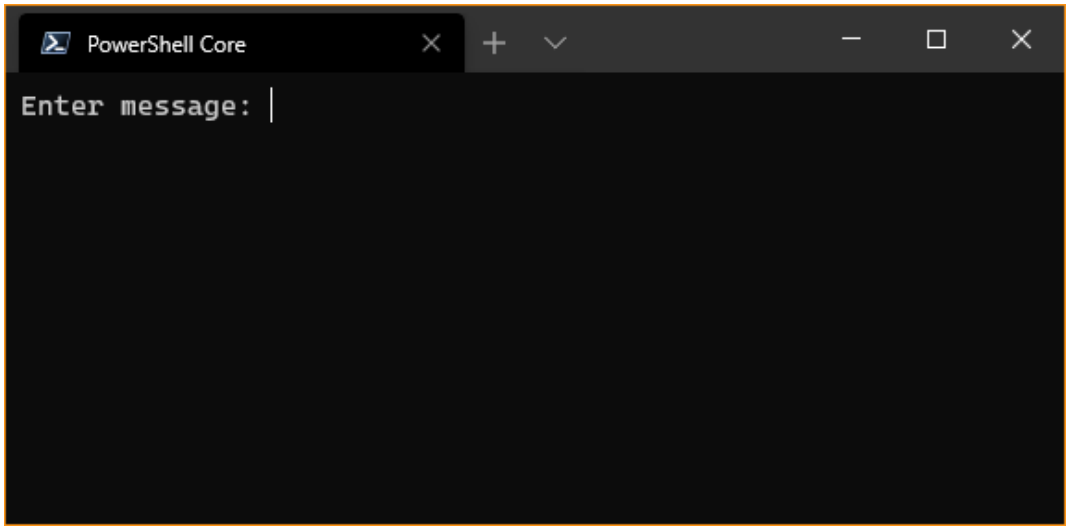
Finally, at the bottom of the code file, call `askForMessage`:

```
askForMessage();
```

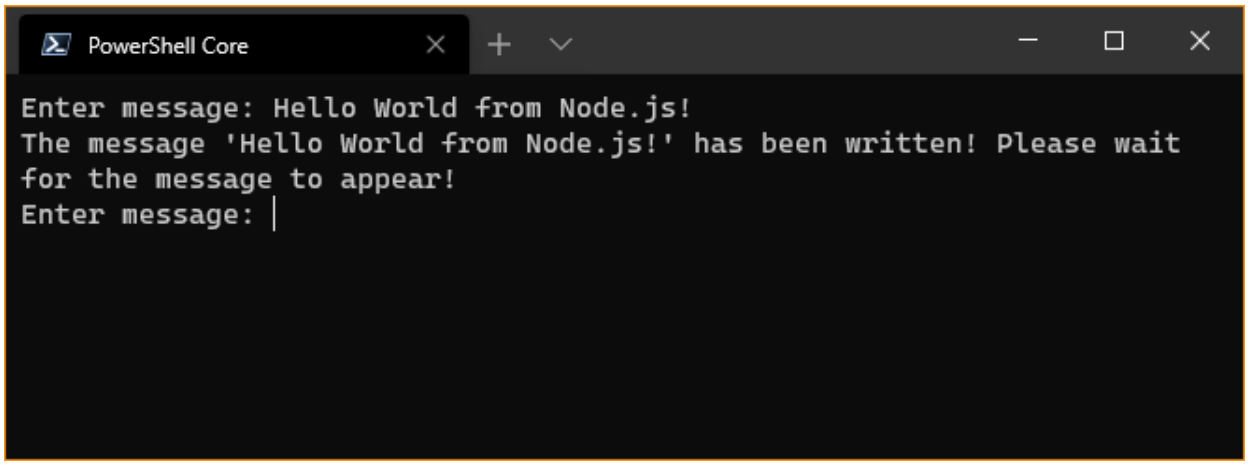
The full code should look like this:

```
const fs = require("fs");
const readline = require("readline");

const readlineInterface = readline.createInterface({
    input: process.stdin,
    output: process.stdout
});

function askForMessage() {
    readlineInterface.question("Enter message: ", (message) => {
        if (message != "")
        {
            fs.writeFileSync("communication.txt", message);

            console.log(`The message '${message}' has been written! Please wait for the message to appear!`);

            askForMessage();
        }
    });
}

askForMessage();
```

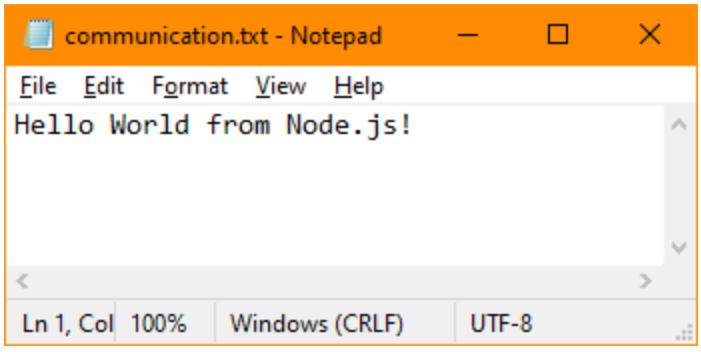
When running the program, the console should ask you for input:




Enter a message and press enter. The console should tell you that the message has been written:




If you open the file, you should see that the message has been written:

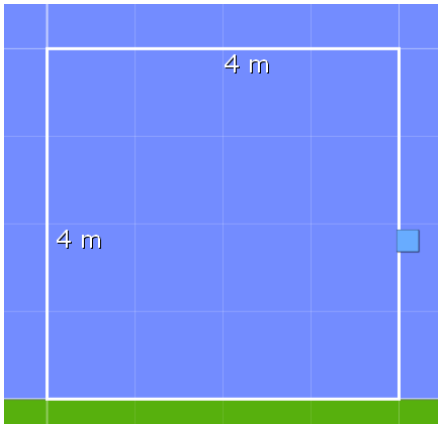



If your program works, you can move onto the Thyme code.

## Thyme script

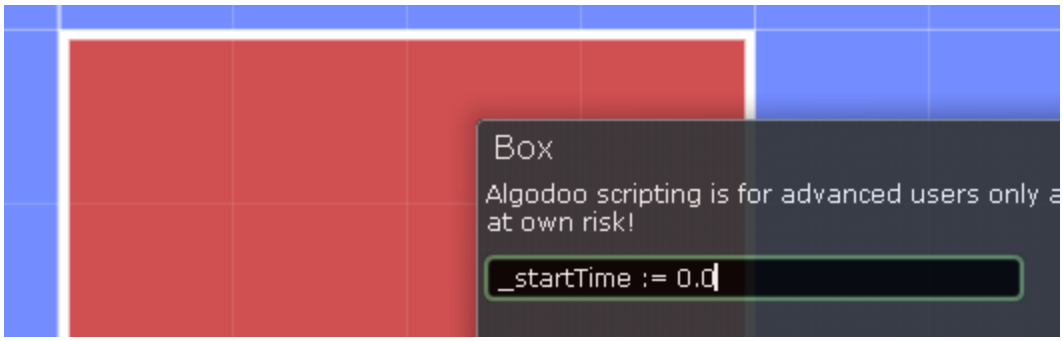
In this section, we will script a box in Algodoo to display the contents of the communication file.

Draw the box that will display the text:




In this example, we will set a timer script. Checking the file every time `postStep` is called may cause lag, so in this example we will check the file every second.

Declare an object variable on the box named `_startTime`:



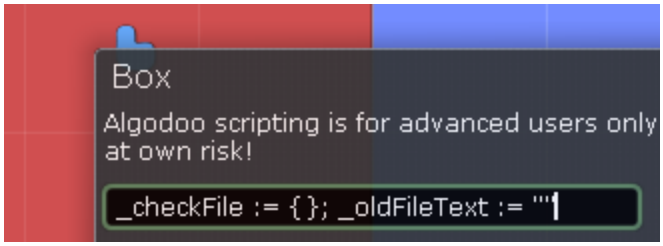

Enter the following script in `postStep`:

```
(e)=>{
    if_then_else(System.time - _startTime > 1, {
        _checkFile();
        _startTime = System.time;
    }, {})
}
```

This is the timer script. Every second, the script will run the `_checkFile` function and reset `_startTime`.

Declare two more object variables:

- `_checkFile` - a function that will read all of the text from the communication file, check whether the text has changed and if it has, change the displayed text to the text from the file.
- `_oldFileText` - used to determine whether the file's text has changed.



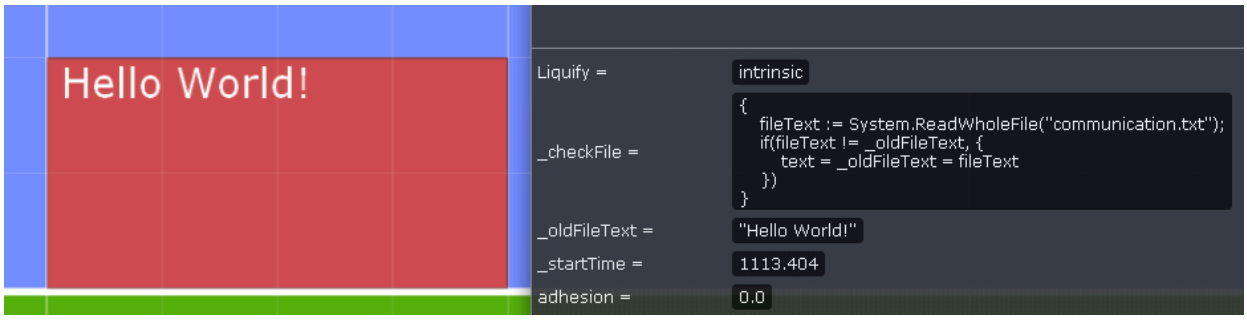

Inside the `_checkFile` function, declare a local variable named `fileText` and set its value to the contents of the communication file:

```
fileText := System.ReadWholeFile("communication.txt");
```

If `fileText` and `_oldFileText` are different, change `_oldFileText` and `text` to `fileText`:

```
if_then_else(fileText != _oldFileText, {
    text = _oldFileText = fileText
}, {})
```

If everything works, the box should display the contents of the file:




While the scene is running, run your program. Enter a new message and the box should display the message.

# Glossary

# Glossary

- **Argument** - a value passed into a function.
- **Boolean** - a data type that can have one of two values - true or false.
- **Comment** - an annotation or explanation in code that is not executed. Used to make the code easier for humans to understand.
- **Concatenation** - joining things together in a series.
- **Encapsulation** - the bundling of data with the methods that operate on that data or the restricting of direct access to some of an object's components.
- **Expression** - a combination of values and/or functions to create a new value.
- **Event** - an action that occurs as a result of the user or another source, such as a mouse click.
- **Float** - a data type representing a number with a decimal part.
- **Function** - a block of code that performs a specific task.
- **Identifier** - a user-defined name of a program element.
- **Integer** - a data type representing a number with no decimal part (i.e. a whole number).
- **Iteration** - where a set of instructions is repeated a given number of times or until a given condition is met.
- **List** - a data type that represents a collection of ordered values.
- **Literal** - a notation that represents a fixed value in code.
- **Operand** - a value that is manipulated by an operator.
- **Operator** - a symbol that tells the language to perform a mathematical, relational or logical operation to produce a result.
- **Parameter** - identify values that are passed into a function. They allow functions to perform tasks without knowing the specific input values ahead of time.
- **Primitive** - basic, not derived from anything else.
- **Property** - a named member of an object.
- **Recursion** - a method of solving a problem where the solution depends on solutions to smaller instances of the problem.
- **Return** - where the execution of a function stops and a value is given out.
- **Scope** - the region where a variable can be used.
- **Selection** - where a program takes a course of action depending on a condition.
- **Sequence** - where defined actions happen in order. A program is formed by sequences of statements.
- **Statement** - an instruction telling a computer to perform a specified action. A program is formed by sequences of statements.
- **String** - a sequence of characters used to represent text.
- **Variable** - a name given to a stored data value that can change.

- **Vector** - a quantity with both direction and magnitude.