# Lua API

The Lua API documentation is currently a stub, more information will follow soon.

# Introduction

> ZeroBrane is no longer in development and therefore not recommended anymore for development with Lua in Pragma. Please use Visual Studio Code instead.

Pragma uses the programming language Lua 5.1 (with LuaJIT) for its scripting engine. If you're new to Lua, the PIL is a good starting point.

Any text editor suited for programming (Notepad++, Sublime, Visual Studio Code, etc.) can be used to write Lua scripts, however using the ZeroBrane IDE is highly recommended due to its powerful debugging capabilities specialized for Lua development (including step-by-step debugging, breakpoints, auto-completion, etc.).

There are many ways to execute Lua code, but the easiest one is via console commands:

- `lua_run <code>` / `lua_run_cl <code>`: Executes the specified Lua **code** serverside / clientside (if the game state is available)
- `lua_exec <code>` / `lua_exec_cl <code>`: Executes the specified Lua **file** serverside / clientside (if the game state is available)

Pragma has one Lua instance per game state. The game state is split into a serverside portion and a clientside portion. The client generally handles things like rendering and device inputs, while the server handles AI, game logic, etc. Some components (like physics) are simulated on both sides and may or may not be synchronized. The Pragma Filmmaker, for instance, is an entirely clientside Lua addon.

The game states (along with the Lua instances) are initialized when you're loading a map / starting a game, which means Lua commands and scripts cannot be executed when you're still in the main menu. Which game states are available depends on the situation:

- Starting a local game/server: Both game states are initialized.
- Connecting to a server: Only the clientside game state is initialized.
- Starting a dedicated server: Only the serverside game state is initialized.

These Lua instances are completely independent and cannot communicate directly with each other (except via net messages). Any variables or functions you define in the serverside state will not be accessible in the clientside state and vice versa. Here are a few examples using the `lua_run` and `lua_run_cl` console commands:

```
lua_run var = "This is a serverside variable" -- Defines the global serverside variable 'var'
lua_run_cl print(var) -- Prints 'nil' (indicating an undefined variable) to the console, because 'var' was not defined
```

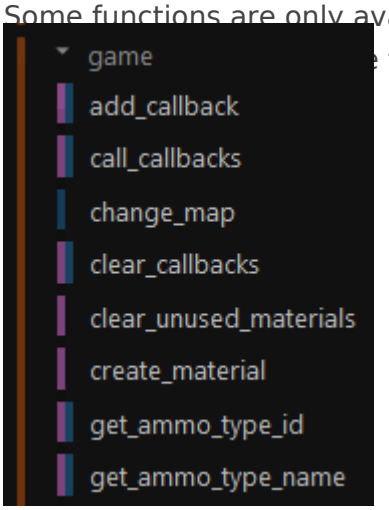
```
clientside
lua_run print(var) -- Prints 'This is a serverside variable' to the console, because we're back in the serverside instance
```

You can also place Lua-files in one of the following autorun directories, in which case they will automatically be executed when the game is started:

- "lua/autorun/<fileName>.lua": The Lua-file will be executed for **both** the serverside and clientside Lua instances.
- "lua/autorun/server/<fileName>.lua": The Lua-file will be executed serverside.
- "lua/autorun/client/<fileName>.lua": The Lua-file will be executed clientside.

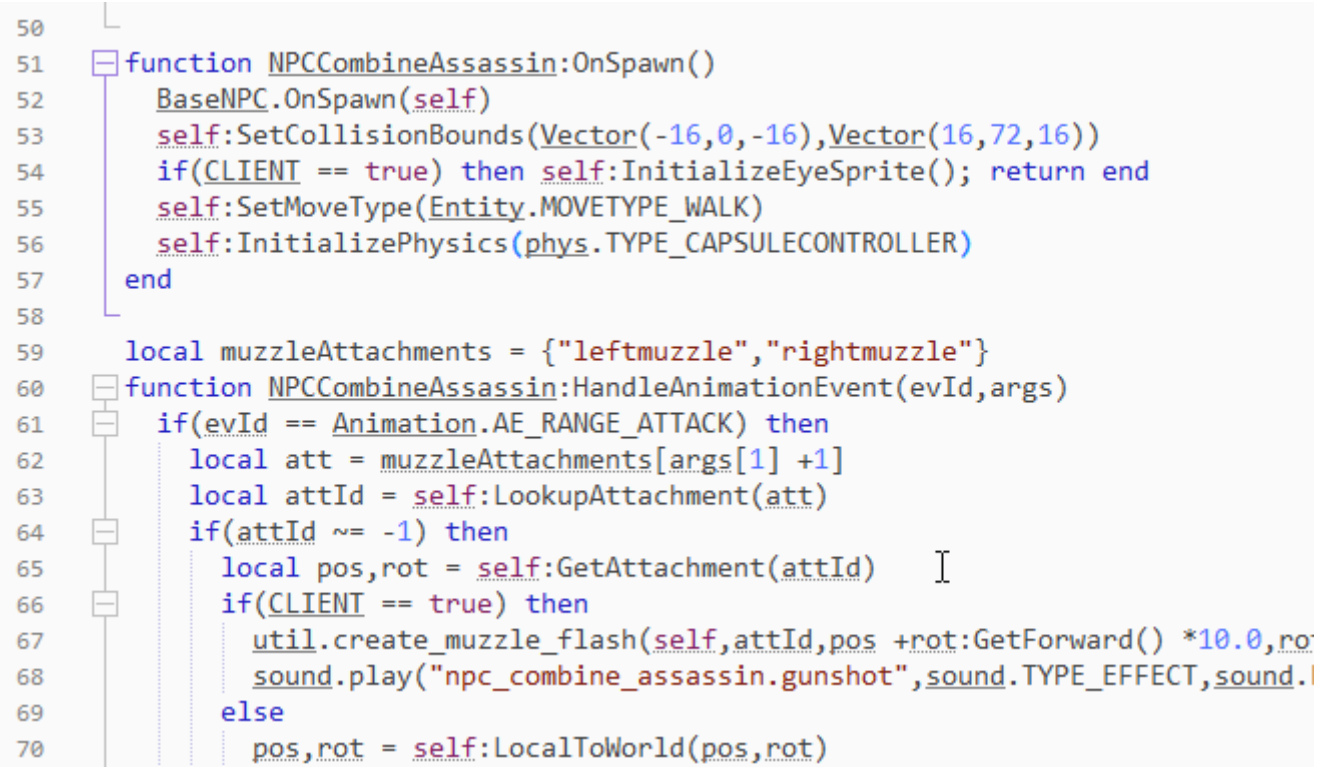
# API Documentation

You can find a list of all available Lua classes, libraries and functions in the API documentation. Some functions are only available serverside, some only clientside, which is indicated by the function names in the navigation:




Anything with a blue bar is available serverside, and anything with a purple bar clientside. For instance, the client cannot trigger a map change, so `game.change_map()` can only be executed via the serverside Lua instance. If you want to know how a particular function is implemented, you can click the -logo, which will direct you to the function implementation on the Pragma GitHub repository.

If you have configured ZeroBrane, all of the functions should also appear in the auto-complete list when typing the name of a library or class:

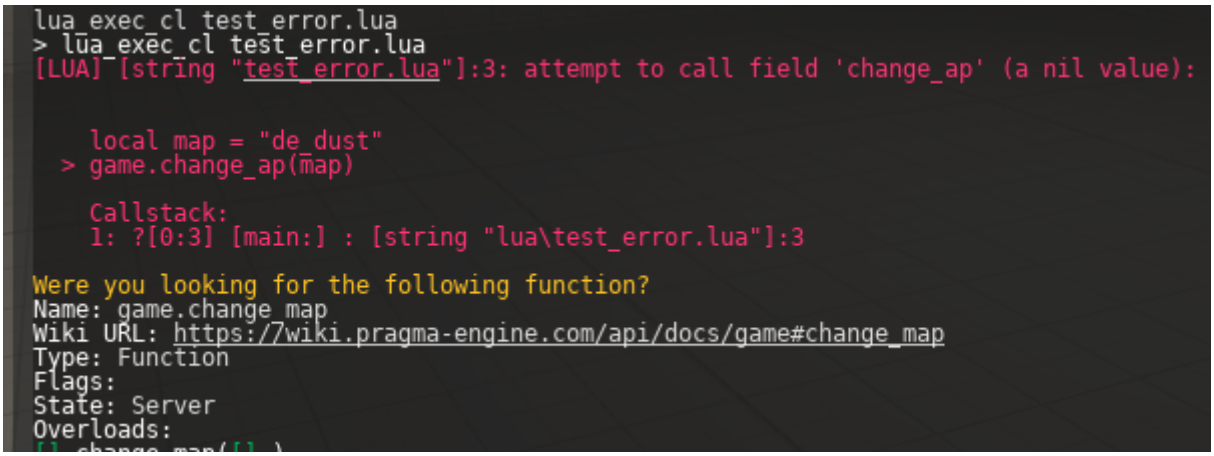


```
50
51  function NPCCombineAssassin:OnSpawn()
52    BaseNPC.OnSpawn(self)
53    self:SetCollisionBounds(Vector(-16,0,-16),Vector(16,72,16))
54    if(CLIENT == true) then self:InitializeEyeSprite(); return end
55    self:SetMoveType(Entity.MOVETYPE_WALK)
56    self:InitializePhysics(phys.TYPE_CAPSULECONTROLLER)
57  end
58
59  local muzzleAttachments = {"leftmuzzle","rightmuzzle"}
60  function NPCCombineAssassin:HandleAnimationEvent(evId,args)
61    if(evId == Animation.AE_RANGE_ATTACK) then
62      local att = muzzleAttachments[args[1] +1]
63      local attId = self:LookupAttachment(att)
64      if(attId ~= -1) then
65        local pos,rot = self:GetAttachment(attId)
66        if(CLIENT == true) then
67          util.create_muzzle_flash(self,attId,pos +rot:GetForward() *10.0,ro
68          sound.play("npc_combine_assassin.gunshot",sound.TYPE_EFFECT,sound.
69        else
70          pos,rot = self:LocalToWorld(pos,rot)
```

## In-Engine Documentation

In addition to the online API documentation, you can also use the `lua_help` console command to retrieve the information for a function/library/class ingame:


Lua errors will also use this documentation to provide you with suggestions on how to correct the error:




# Hello World

Creating a Lua script is very straight forward. Simply create a new text-file and add the following line:

```
print("Hello World")
```

Then save that file as "hello_world.lua" to "Pragma/lua/". To execute the script, simply start a new game and run the console command `lua_exec_cl hello_world.lua` and you should see the message "Hello World" printed in the console.

Any Lua files that have been executed at least once are cached by the Engine and automatically reloaded when changed in the future (until you close the game). This means if you make any changes to the file, all you have to do is save it, and it will automatically be executed again (i.e. you don't have to run `lua_exec_cl` anymore). You can test this by changing the print message in your Lua-script and then saving it, and it should automatically print your new message to the console.

## Addons

In general, putting Lua scripts in the main Lua directory ("Pragma/lua/") should be avoided to avoid clutter and naming conflicts and an addon should be used instead:

- Create a new directory called "hello_world" in "Pragma/addons/"
- Create a new "lua" directory inside it
- Move your "hello_world.lua" to "Pragma/addons/hello_world/lua/hello_world.lua"

Pragma will automatically mount the addon, and will treat all asset files within the addon as if they were in the main Pragma directory. This means that despite the file being in a different location, you can still use `lua_exec_cl hello_world.lua` to execute the file.

# Examples

## Creating a timer

Lua state: Any

```
-- Print "Hello" every 2 seconds
local numberOfRepetitions = -1 -- -1 = infinite
local timer = time.create_timer(2.0,numberOfRepetitions,function()
    print("Current game time:",time.cur_time())
end)
timer:Start()

-- Remove the timer after 15 seconds using a second timer
time.create_simple_timer(15,function()
    util.remove(timer)
end)
```

## Registering a custom console command

Lua state: Any

```
console.register_command("custom_command",function(pl,...)
    if(pl ~= nil) then
□□print("Command was initiated by player ",pl:GetEntity():GetName())
    end
    print("Command arguments: ",...)

    local arg1,arg2,arg3 = ...
    -- Do something with arguments
end)
```

## Creating a GUI element

Lua state: Client

```
local rect = gui.create("WIRect")
rect:SetColor(Color.Red)
rect:SetPos(Vector(32,32))
rect:SetSize(256,65)

local text = gui.create("WIText",rect)
text:SetColor(Color(0,255,64,255))
text:SetPos(5,5)
text:SetText("Hello World")
text:SizeToContents()
```

## Classes and class inheritance

Lua state: Any

```
-- Define class 'Car' in 'util' library
util.register_class("util.Car")
-- The constructor for the 'Car' class, which allows us to create an instance of the class like so:
-- local carInstance = util.Car("Name of the car")
function util.Car:__init(name)
	self.m_name = name
	self.m_color = Color.Red
end

function util.Car:SetName(name) self.m_name = name end
function util.Car:GetName() return self.m_name end

function util.Car:SetColor(color)
	self.m_color = color
	self:OnColorChanged(color)
end
function util.Car:GetColor() return self.m_color end

-- These can be overwritten by derived classes
function util.Car:GetWheelCount() return 4 end
function util.Car:OnColorChanged(newColor)
	print("Color of car '" .. self:GetName() .. "' has been changed to " .. tostring(newColor) .. "!")
```

```
end

-------

-- Define class 'Truck' and derive from base class 'Car'
util.register_class("util.Truck",util.Car)
function util.Truck:__init(name)
	util.Car.__init(self,name) -- Base constructor has to be called, note the use of '.' instead of ':' for calling base functions
end

function util.Truck:GetWheelCount() return 6 end -- Overwrites the base function when called on a truck instance

function util.Truck:OnColorChanged(color)
	util.Car.OnColorChanged(self,color) -- Call base function
	debug.beep() -- Play a beep sound after the message has been printed
end

-------

-- Create car object
local car = util.Car("Car Name")
car:SetColor(Color.Lime)
print("Number of car wheels: " .. car:GetWheelCount())

-- Create truck object
local truck = util.Truck("Truck Name")
truck:SetColor(Color(0,255,128,255))
print("Number of truck wheels: " .. truck:GetWheelCount())
```

## Respawn all players

Lua state: Server

```
for ent in ents.iterator({ents.IteratorFilterComponent(ents.COMPONENT_PLAYER)}) do
	local playerC = ent:GetComponent(ents.COMPONENT_PLAYER)
	playerC:Respawn()
end
```

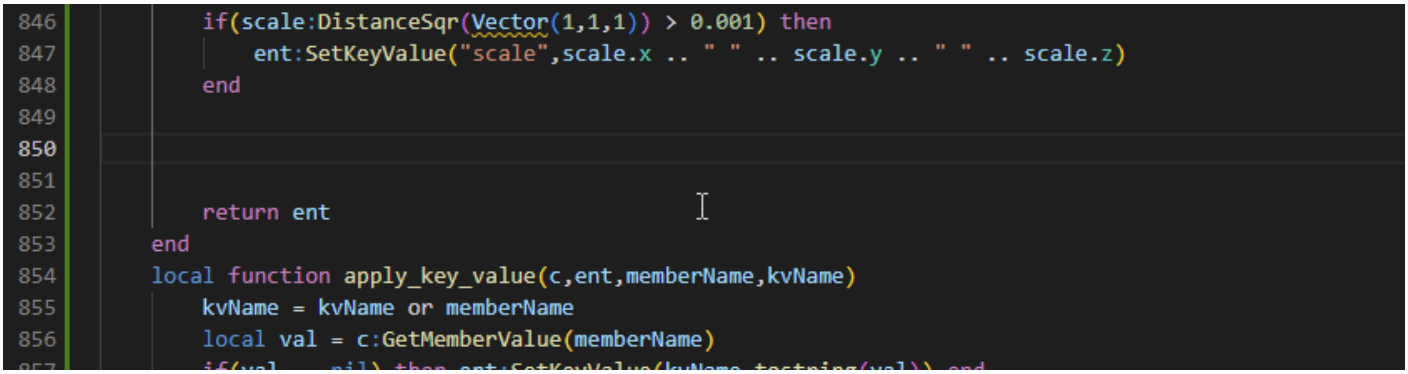
# Pragma Filmmaker

The Pragma Filmmaker is implemented as a Pragma addon written in Lua, you can find the source code on GitHub: https://github.com/Silverlan/pfm

# Visual Studio Code

> You can use any script editor of your choice to create Lua scripts for Pragma. The information on the page is the recommended approach, but it is not required.

Pragma supports Lua development and debugging with Visual Studio Code. This includes code auto-completion and suggestions:

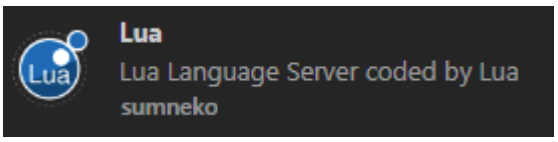



as well as debugging using breakpoints, step-by-step code execution and immediate code execution:

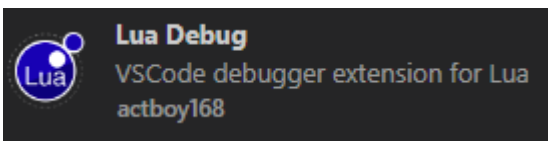
## Setup

To enable it, install these extensions:

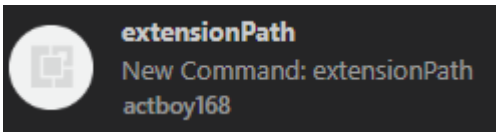
- Lua by sumneko:




- Lua Debug by actboy168:




- extensionPath by actboy168:

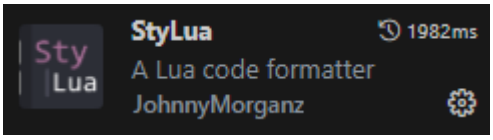



- Optional: GitHub Copilot

### StyLua

For automated code-formatting, install the following extension:

- StyLua by JohnnyMorganz:




After installing StyLua, press Ctrl +LShift +P, and run the command "Preferences: Open User Settings (JSON)". In the JSON file, add the following line below `editor.defaultFormatter`:

```
"editor.formatOnSave": true
```

Once all of these extensions are installed, open the root directory of Pragma in Visual Studio Code.

# Debugging

To be able to use the Lua debugger, you have to start the debugger server in Pragma first.

> LuaJIT will be disabled when debugging, which means script execution will be slower.

1. Add `-console -luaext` to the launch options of Pragma. This will enable additional Lua libraries required by the debugger, as well as the developer console, which will print Lua errors and can be used to execute Lua code directly.
2. Open the main Pragma directory in Visual Studio Code (`File > Open Folder...`).
3. Run the console command `debug_start_lua_debugger_server_cl` in Pragma to launch the debugger server.
4. Press F5 in Visual Studio Code to start debugging.

You can now use the debugging tools in Visual Studio Code.

## Breakpoints

### Conditions

# Immediate Code Execution

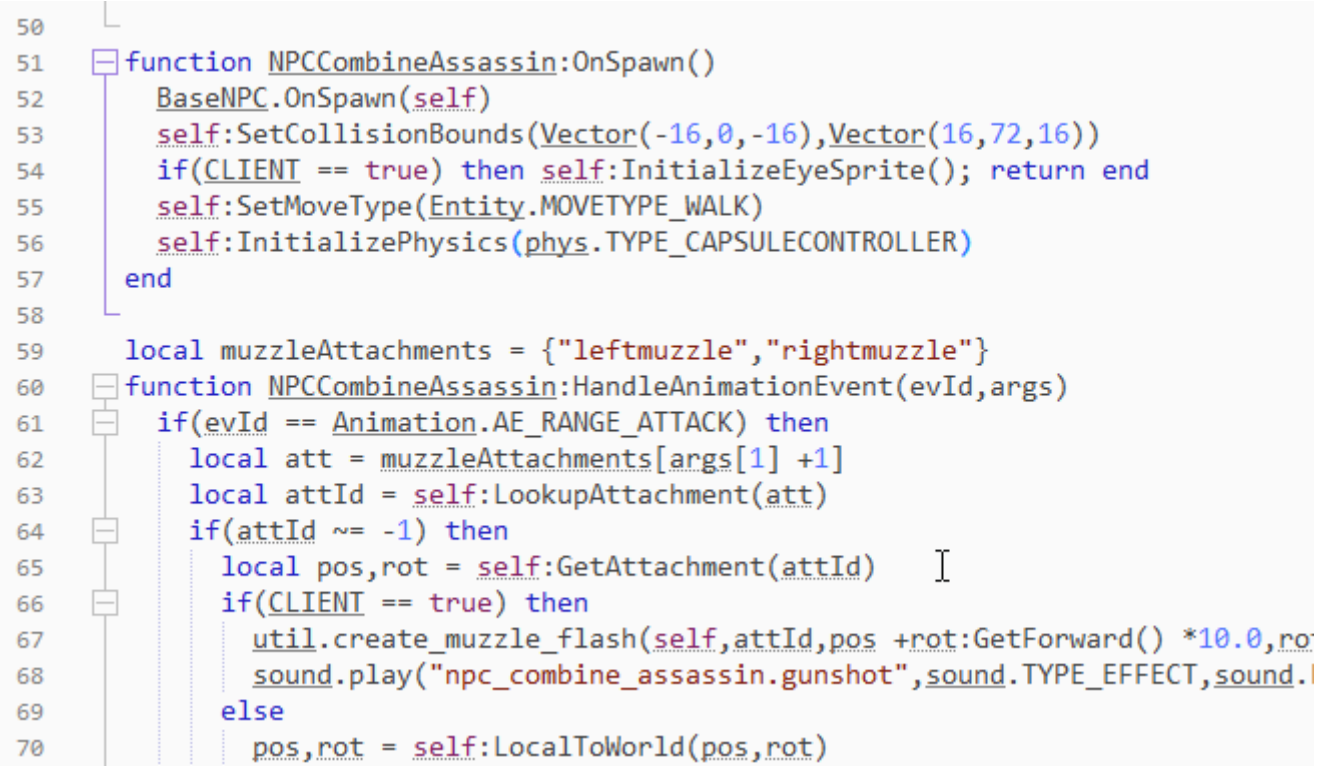
# ZeroBrane IDE

## Overview

All you need to create Lua-scripts for Pragma is a basic text-editor, however to get access to advanced debugging capabilities, it's recommended to use the ZeroBrane Studio Lua IDE. ZeroBrane is free and allows you to debug Lua code with features like step-by-step code execution and breakpoints, which you can see in action here:

https://www.youtube.com/embed/pm73zJIQwUo

Pragma also ships with an auto-complete script for ZeroBrane to maximize productivity:



```
50
51  function NPCCombineAssassin:OnSpawn()
52    BaseNPC.OnSpawn(self)
53    self:SetCollisionBounds(Vector(-16,0,-16),Vector(16,72,16))
54    if(CLIENT == true) then self:InitializeEyeSprite(); return end
55    self:SetMoveType(Entity.MOVETYPE_WALK)
56    self:InitializePhysics(phys.TYPE_CAPSULECONTROLLER)
57  end
58
59  local muzzleAttachments = {"leftmuzzle","rightmuzzle"}
60  function NPCCombineAssassin:HandleAnimationEvent(evId,args)
61    if(evId == Animation.AE_RANGE_ATTACK) then
62      local att = muzzleAttachments[args[1] +1]
63      local attId = self:LookupAttachment(att)
64      if(attId ~= -1) then
65        local pos,rot = self:GetAttachment(attId)
66        if(CLIENT == true) then
67          util.create_muzzle_flash(self,attId,pos +rot:GetForward() *10.0,ro
68          sound.play("npc_combine_assassin.gunshot",sound.TYPE_EFFECT,sound.
69        else
70          pos,rot = self:LocalToWorld(pos,rot)
```

You are of course free to use any other editor, in which case you can ignore the rest of this tutorial.

# Setup

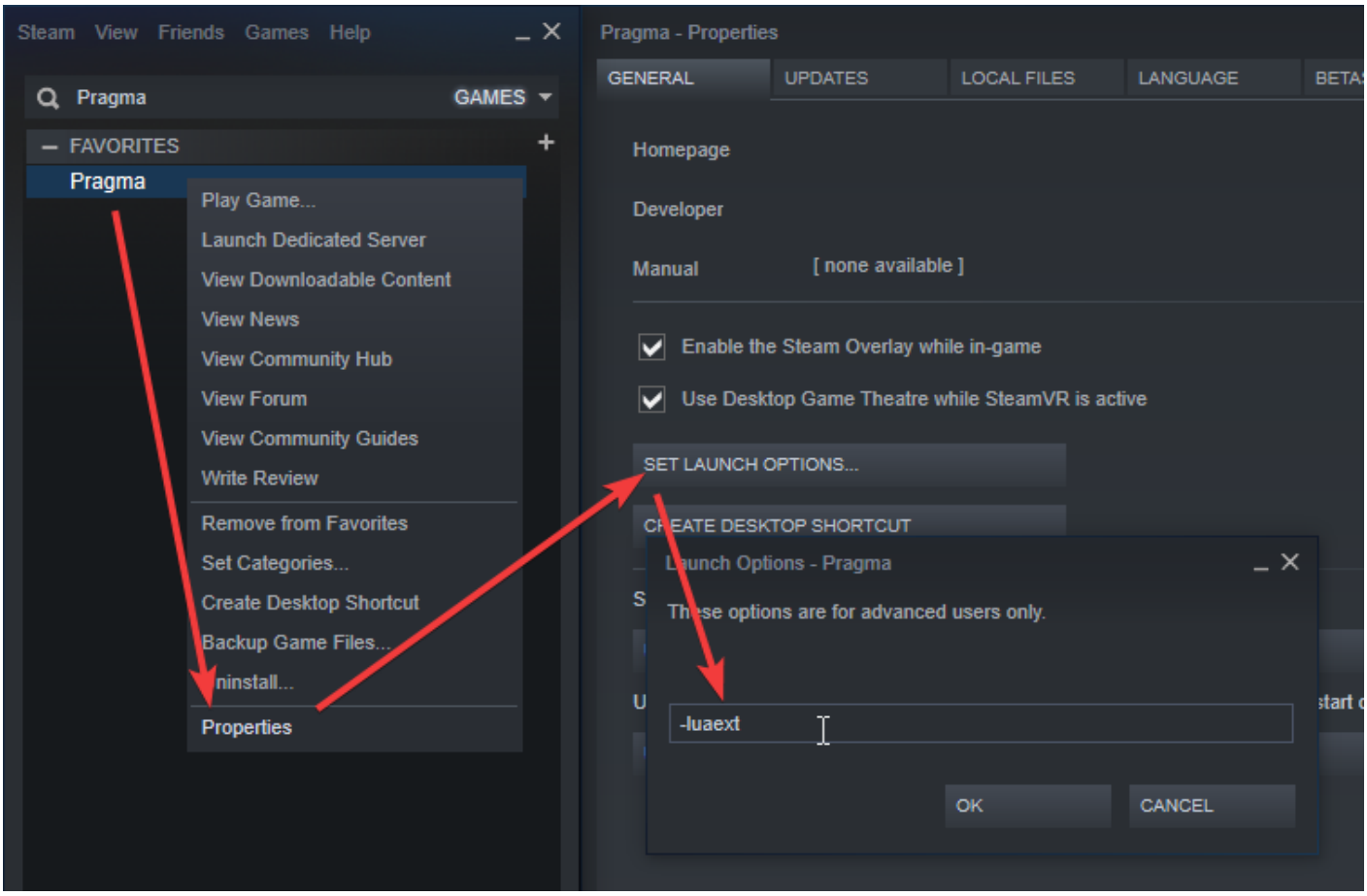
After downloading and installing ZeroBrane, a few more manual steps are required to set it up for Pragma:

1. Start ZeroBrane, Select "Project -> Project Directory -> Choose..." from the menu bar, navigate to your Pragma installation (Default: "C:/Program Files (x86)/Steam/steamapps/common/Pragma/") and select it. Without this step, the debugger may not be able to locate the Lua files during debugging.
2. Navigate to "Pragma/doc/ZeroBrane" and copy its contents to your ZeroBrane installation (Default: "C:/Program Files (x86)/ZeroBraneStudio").
3. Select "Edit -> Preferences -> System" from the menu bar and append the following lines to the end of the file:

   ```
   include "pragma.lua"
   editor.autotabs = true
   ```

4. Save the file and restart ZeroBrane.
5. Select "Project -> Lua Interpreter -> Pragma" from the menu bar. This step is required for the auto-complete feature.

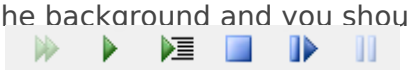
This completes the basic setup for ZeroBrane, however to use the debugging capabilities a few more steps are required every time you want to use the debugger. The debugger requires that you add `-luaext` to the launch parameters in Pragma:




If set up correctly, you should see a red "[D]" next to the version number in the main menu:
**v0.9.7 Win64 [D]**

This will enable several Lua modules which are required for the debugger. These modules could potentially be used to cause damage to your system by a malicious Lua-script, so it's recommended to remove the launch option again when playing on multiplayer servers, or if you've downloaded custom addons from sources you don't completely trust.

> The following steps have to be executed *every* time you want to use the debugger, *after* Pragma has been started, but *before* a map has been loaded (i.e. while you're in the main menu, or before you've run the "map" command on a dedicated server).

1. Start ZeroBrane and select "Project -> Start Debugger Server" from the menu bar.
2. Run the console command `sh_lua_remote_debugging 1` if you want to debug a serverside script, or `sh_lua_remote_debugging 2` if you want to debug a clientside script.
3. Open a Lua-file from your Pragma installation in ZeroBrane. Any Lua-file will do, but without this step, ZeroBrane may not be able to locate any Lua-files at all.
4. Load a map. The "output" window of ZeroBrane should say something along the lines of "debugging session started", which means it connected successfully to Pragma. In this case Pragma should appear frozen and unresponsive in the background and you should see the following controls in the menu bar of ZeroBrane: 
5. Click the green arrow (alternatively press F5) to resume Pragma.

The debugger is now fully set up and initialized. You can find an overview of its features and how to use them here.

# API Documentation

Lua API documentation of all available classes, libraries and functions.