Setting up local workspace for Phaser 3 projects

In this document, I will guide you through the process of setting up for development with Phaser3 using VisualStudio code as IDE.

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

Foreword

You are going to get your hands dirty, a lot. We could theoretically jump directly to the most complex use-case of Phaser 3 we will be targeting, Phaser3 + TypeScript + RequireJS + AMD module system + local web server, but chances are you will miss something along the way. Web development is a nasty one - full of tools built around JavaScript language whose design was not meant to accommodate complex projects. So let's take it easy and take one step at a time.

Goals we will try to achieve here are:

- Being able to develop with Phaser3 JavaScript/WebGL framework
- Use Visual Studio Code as our IDE to code, compile, run and debug the game
- Use TypeScript in order to write typed code and having intellisense in VSCode available at the same time
- Being able to split the code of your game into multiple files, define file dependencies (imports) in order to keep things clean and tidy

Disclaimer - Windows 10

I'm a Windows 10 user, so this tutorial has been tried and confirmed working for Windows 10 only. However, all tools used are available in other platforms as well (Linux, MacOS), so it should work there as well.

Step 1: Install Chrome

In the end, we will be using Chrome as it has JavaScript (JS) debugging capabilities. Phaser3 game will run as JS game within this browser and through Chrome Debugger Extension for Visual Studio Code (VSCode), we will be having VSCode to attach to Chrome to allow for placing code breakpoints within VSCode and debug the code.

http://www.google.com/chrome

Step 2: Install NodeJS

Software development is complex and there are myriads of engines, frameworks, libraries even small snippets of code that are depending on each other. So if you want to take the development seriously, you always need to use some kind of package dependency management system (a.k.a. dependency hell).

In the realm of JavaScript, the widely used system is NodeJS, it offers more than dependency management (a.k.a. NPM), but we will use it mostly for that. Moreover, VSCode has NodeJS integration built-in, so we will also be able to run NodeJS tasks directly from our IDE (like "compile").

https://nodejs.org/en/download/

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce" kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

Step 3: Install HTTP server

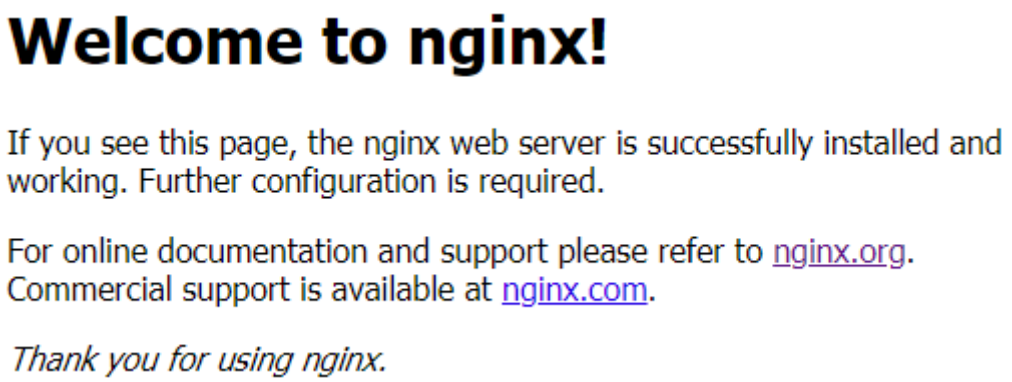
Phaser3 is a JavaScript/WebGL application, that means it will run in the context of a webpage. In order to access the webpage, we need to locally run HTTP server that will be serving the webpage as well as the rest of files (JavaScript files, assets, GLSL code…).

There are multitude of options to go for as this page is suggesting: https://phaser.io/tutorials/getting-started-phaser3/part2

Personally, I have Apache HTTP Server at my machine but that's a beast you probably do not want to fight with. If you are on Windows, I'm suggesting you to use Nginx. If you are on Linux, good chances are you know Python, so you can use Python HTTP module to run local server. Another option is to go with NPM package http-server.

If you are on Windows, here are the steps for Nginx:

1. Download Nginx
2. Upack it somewhere
3. Run nginx.exe
4. In chrome, navigate to http://localhost to confirm it is working for you
5. You should see



6. If you don't, than it means Nginx failed to start on port 80, to fix than, edit `conf/nginx.conf` and change the number on line with `listen 80;` to, e.g., `listen 8080;`
7. Restart nginx.exe and navigate to http://localhost:8080 (or supply any other number you have set Nginx to listen to / serve at).
8. Once you have confirmed Nginx is running, check it's `html` folder, this is the `webroot` folder that is being served by Nginx.

If you are going to work with other http servers, always confirm they are working as expected and you know where `webroot` folder is located as you will need to deploy your game into that folder and forge URLs relatively to that folder as you will want to access your games from Chrome.

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

In all the examples and localhost URLs, I will assume you are running your web server at port 80. If not, you will have to correct all such URLs to match your environment.

Step 4: Install Visual Studio Code

Now you may have different preferences on IDE. Go for that by all means. What I'm guaranteeing with VSCode is you will be able to code, compile, run and debug your game within VSCode (expect the game will be running in Chrome of course).

VSCode is available for Windows, Linux as well as MacOS.

Just download and install it: https://code.visualstudio.com/

And open it to confirm… it opens :D

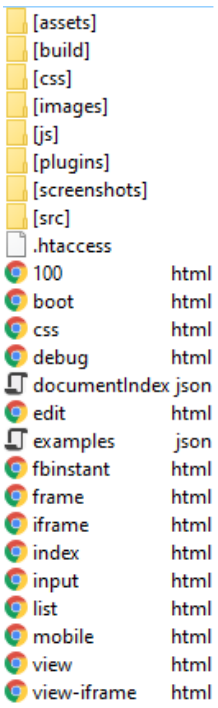
Step 5: Serving Phaser3-Examples locally

The reason Phaser has been chosen for the course is that there are a lot of examples available for you to see and play with. They are hosted here: https://phaser.io/examples
I encourage you to go there and click on a few of those! At least, play some games: https://phaser.io/examples/v3/category/games :-)

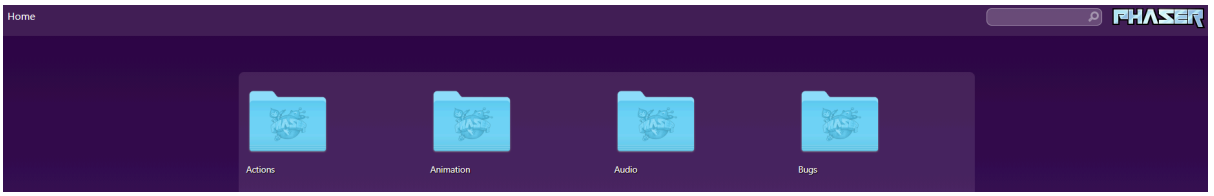
Now having those examples online is nice, but it is better to have them locally stored, so you can play with them. So in this step, we will use IDE (Visual Studio Code) to edit the examples and let our http server serve them.

1. Download Phaser3-Examples, just get the zip or clone the source: https://github.com/photonstorm/phaser3-examples
2. In the repo (unpacked zip), there is a folder `public`, locate it, it has more or less following content:

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

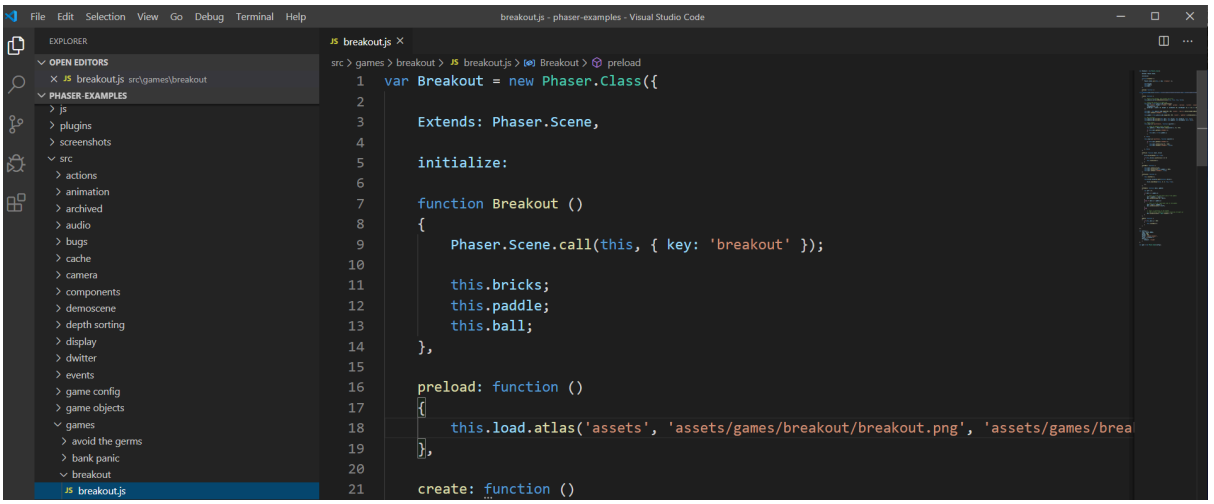


3. In your `webroot` folder, create `phaser3-examples` folder and copy the contents of the `public` folder into it.
4. Open http://localhost/phaser3-examples and confirm the examples are being served by your webserver.



...

5. Now, you can open VSCode and from Menu > File > Open Folder open the `phaser3-examples` folder in your `webroot`.
6. Navigate, e.g., to `src/games/breakout/breakout.js` to see the code for the Breakout game in VSCode



Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

Now you are able to start playing with any example from within VSCode. No TypeScript, no debugging yet...

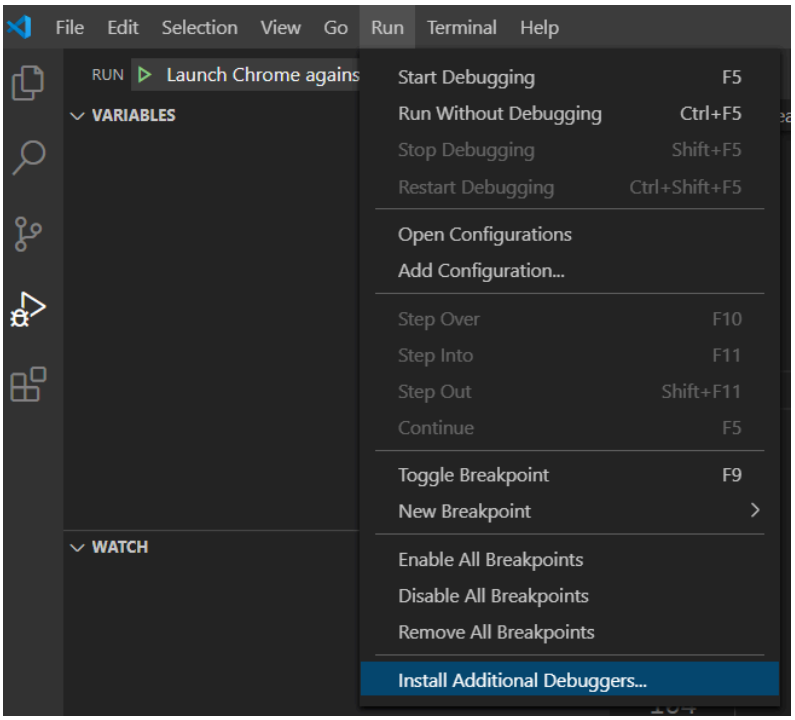
Step 6: Debug Phaser3-Examples from within VSCode

Let's add debugging support into VSCode now. For that, we will need to perform two steps:

1. Install Debugger for Chrome
2. Configure Chrome launch from within VSCode
3. Place breakpoints and see if it works

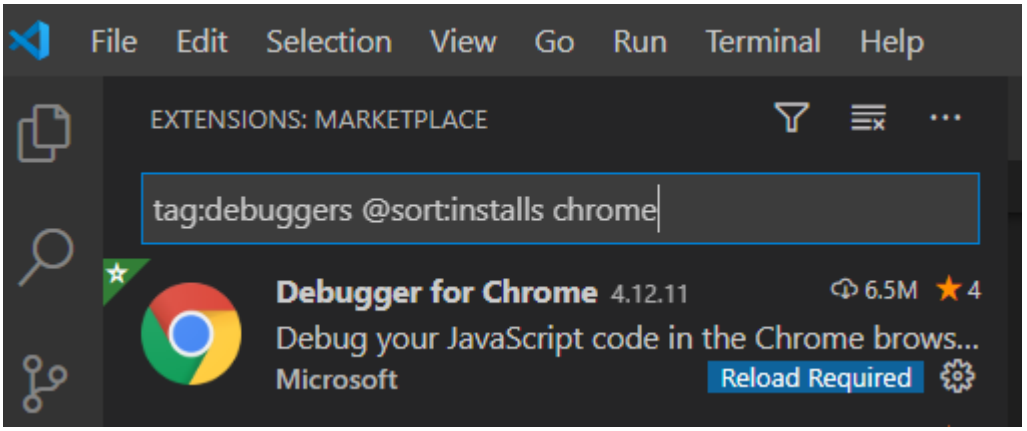
6.1 Install Debugger for Chrome into VSCode

In VSCode, navigate to Menu > Run> Install Additional Debuggers



In the Marketplace, add "Chrome" into the search line.

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce" kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

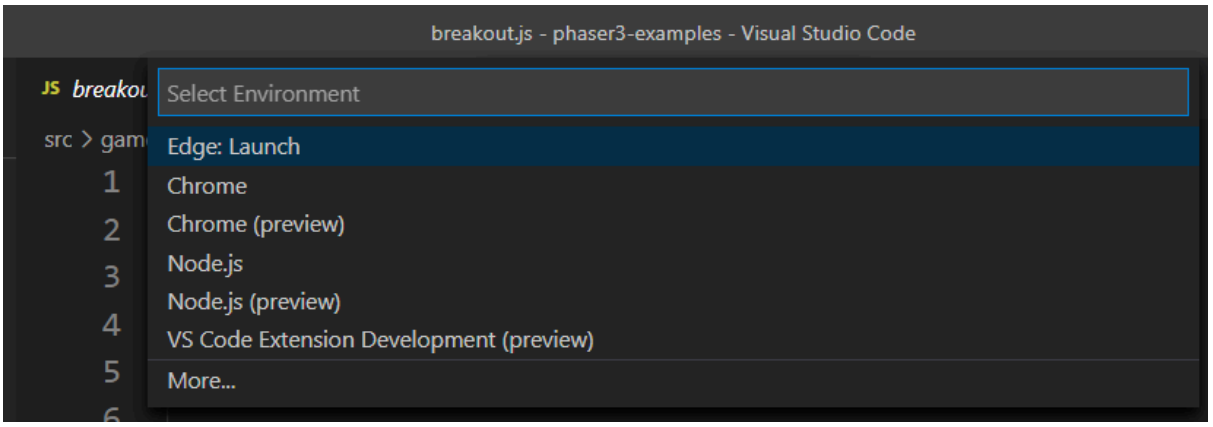


Click on it and install it.
Restart VSCode afterwards.

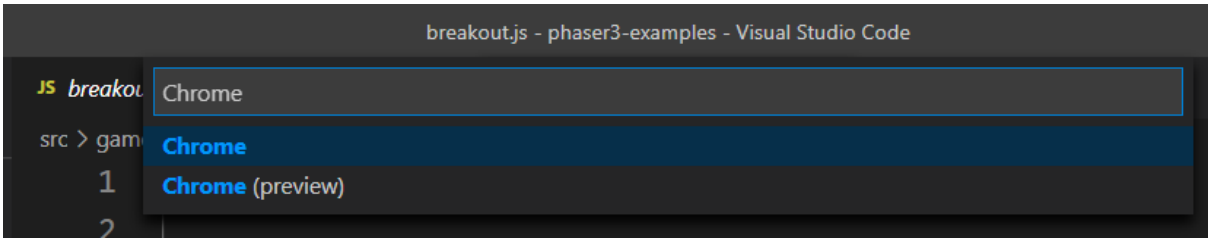
6.2 Prepare launch.json to start Chrome from within VSCode

To “Run” anything within VSCode, one usually uses the F5 function key as a shortcut, so press that. Alternatively do Menu > Run > Start debugging.

It will not do much as there is no launch configuration present in the folder we have opened, but VSCode will offer us to create one.



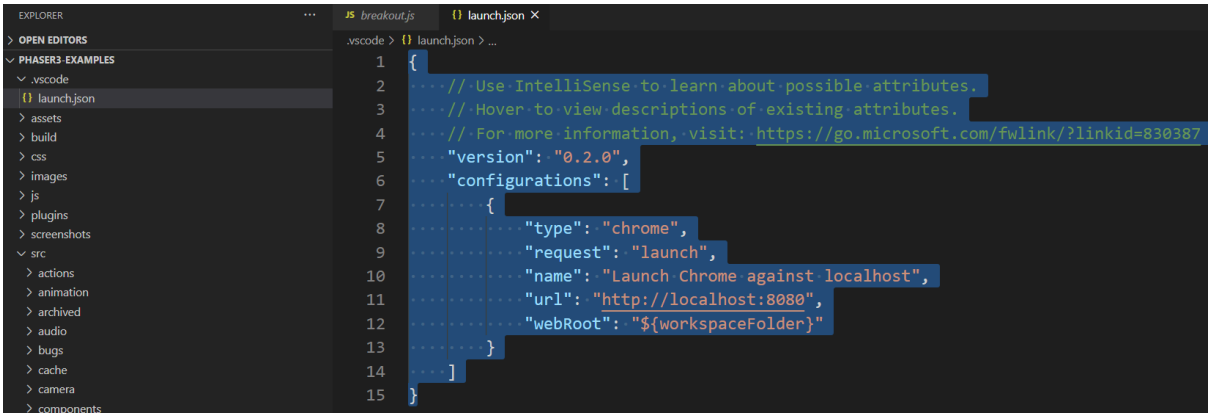
If you do not see “Chrome” in the list, just type Chrome into the search box.



And select it.

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

This will not launch Chrome yet, but it will create a new launch configuration that it will put into the `.vscode` folder within the folder we have opened in VSCode.



Now we need to adapt that. If you are running web server at port 80 and you are named all the folder as in this guide, following configuration will do the trick (copy-paste):

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "type": "chrome",
            "request": "launch",
            "name": "Launch Chrome against localhost",
            "url": "http://localhost/phaser3-examples",
            "webRoot": "${workspaceFolder}/.."
        }
    ]
}
```

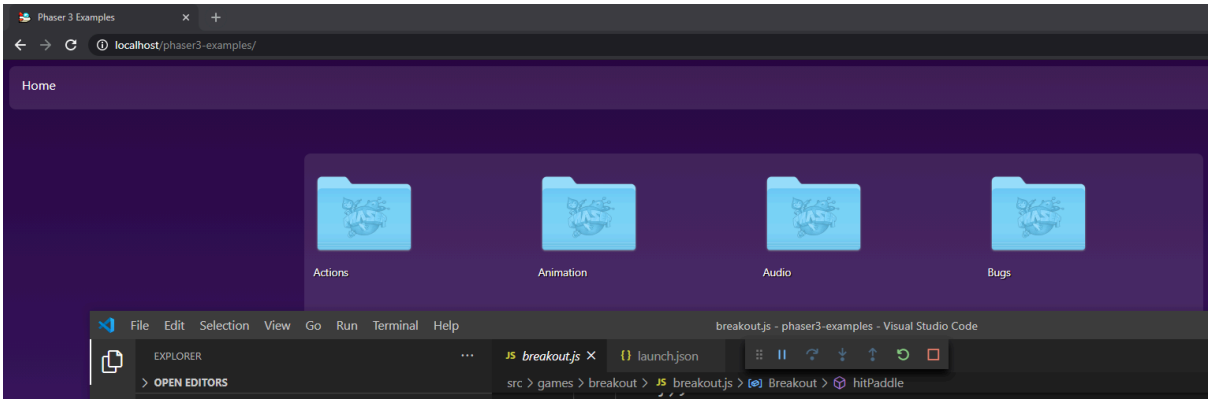
The important configuration lines are `"url"` and `"webRoot"`.

`"url"` says what URL Chrome should open when launched, so we change it to open the `phaser3-examples` folder right away.

`"webRoot"` must then point to your server `webroot` folder. As our VSCode workspace folder is `phaser3-examples`, which resides in the `webroot` folder, we just need to get one directory above and so we are using `/..` Suffix.

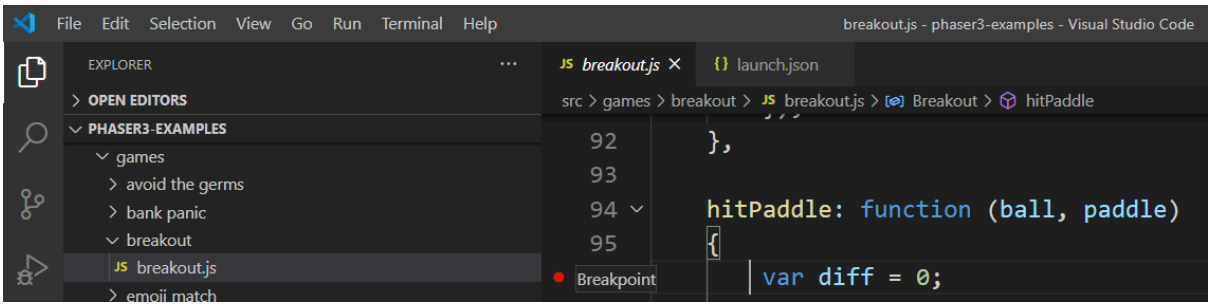
Press F5 again to confirm VSCode will launch Chrome for you.

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.



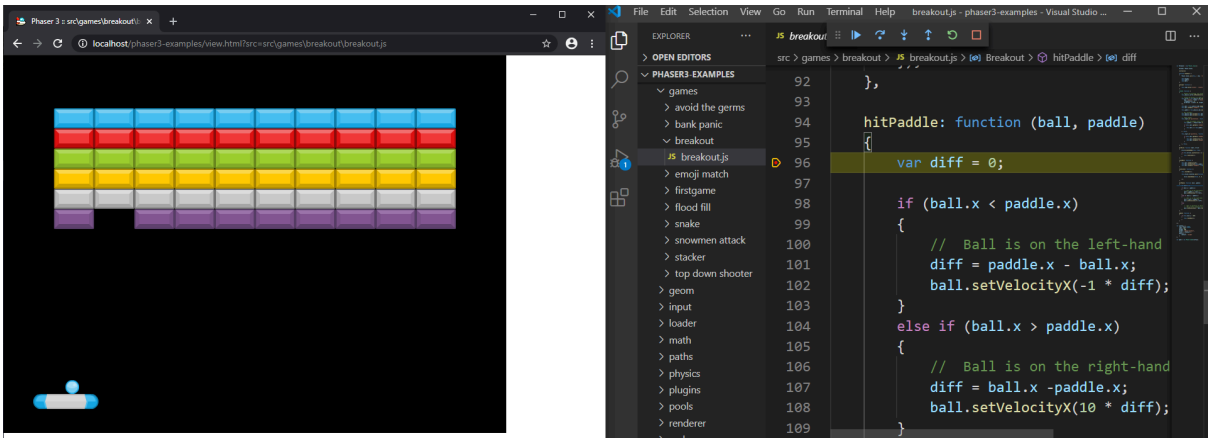
6.3 Try debugging from within VSCode

Let's breakpoint Paddle game, in VSCode navigate to `src/games/breakout/breakout.js`. And put breakpoint at the beginning of `hitPaddle` function by clicking left of a line you want to break point like this:



That's, now start examples again (F5), open Breakout game, and let the ball hit the paddle.

Observe, the game in Chrome will pause, as our breakpoint gets hit and we are able to step the code within VSCode.



Voila! Very convenient!

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

WARNING: You might not be able to hit breakpoints in your code due to the following bug https://github.com/microsoft/vscode-chrome-debug/issues/961
If your breakpoints are not hit, try to edit `launch.json` file and replace `${workspaceFolder}` with the actual full path.
(kudos to Arthaka who reported that!)

Step 7: Shooter example in TypeScript

As a next step, we will install TypeScript compiler and test it on an example game, that is not modularized (it seems like it is, but it isn't as we are not using any import statements in there).

Now it is going to be quite complex:

1. We will use NPM to globally install TypeScript compiler.
2. We will need to understand how VSCode is running the compiler and where it is putting its output.
3. We will need to understand, how the code is run from JavaScript.

As the result, we will be able to start coding in TypeScript (instead of JavaScript) while being able to breakpoint TypeScript code still (the magic we will cherish).

7.1 Install TypeScript Compiler via NPM

This one should be easy, though quite magical. We will have NPM to install TypeScript compiler globally, meaning it will be put on path so over programs (like our VSCode) will be able to use it.
Open the command line (in Windows 10 as Administrator) and execute following (provided you have NodeJS and its NPM installed and put on path):

```
npm install -g typescript
```

Then type in the command line tsc, which is TypeScript compiler program to see if it is working...

```
tsc
```

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

```
C:\Users\Jimmy>tsc
Version 4.0.3
Syntax:   tsc [options] [file...]

Examples: tsc hello.ts
          tsc --outFile file.js file.ts
          tsc @args.txt
          tsc --build tsconfig.json

Options:
 -h, --help                                          Print this message.
 -w, --watch                                         Watch input files.
 --pretty                                            Stylize errors and messages u
 --all                                               Show all compiler options.
 -v, --version                                       Print the compiler's version.
```

If it does not, you will need to check whether installation was successful and whether tsc has been put on path.

More on the topic of TypeScript in VSCode.

7.2 Understanding the Shooter example setup

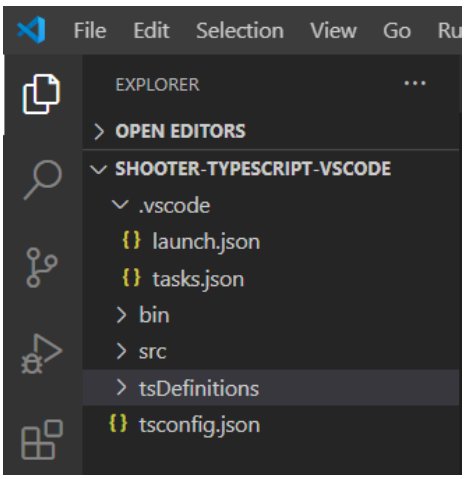
Having TypeScript compiler installed, we can download and open the Shooter example project.

Download the example here: shooter-typescript-vscode

Unpack the zip and put the folder `shooter-typescript-vscode` into your `webroot` folder.

And open `shooter-typescript-vscode` folder within VSCode.

Looking at the folder structure, there are several things we will need to comment on.



Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

7.2.1 launch.json

We have seen that previously, it contains configuration how to run our example. Adapt it if required.

```
{
  "version": "0.2.0",
  "configurations": [
  {
    "type": "chrome",
    "request": "launch",
    "name": "Launch Chrome against localhost",
    "url": "http://localhost/shooter-typescript-vscode/bin/",
    "webRoot": "${workspaceFolder}/.."
  }
  ]
}
```

Note the `"url"` property, we are navigating to a folder, into which we will be compiling our game, TypeScript code.

7.2.2 tasks.json

This file defines a custom task VSCode is able to run via shortcut Ctrl+Shift+B. If you press the shortcuts, VSCode should offer you “tsc” as per below.

Here is the file contents:

```
// Compiles a TypeScript program
{
    "version": "2.0.0",
    // The command is tsc. Assumes that tsc has been installed using npm
install -g typescript
    "command": "tsc",
    // The command is a shell script
    "type": "shell",
    // Show the output window only if unrecognized errors occur.
    "presentation": {
        "echo": true,
        "reveal": "silent",
```

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

```
        "focus": false,
        "panel": "shared",
        "showReuseMessage": true,
        "clear": false
    },
    // The args to pass to the typescript compiler
    "args": ["--out","bin/js/game.js"],
    // use the standard tsc problem matcher to find compile problems in
the output.
    "problemMatcher": "$tsc"
}
```

The important property is `"args"`, which is telling the compiler to spill out only a single file. I.e., it will look for all `*.ts` files in your folder and compile them (transpill them) into JavaScript concatenating the output into a single `game.js` file.

However, this is not the only place where we store the configuration for the TypeScript compiler.

WARNING: At some Windows 10 systems, you might run into an issue that VSCode cannot run tsc because of execution policies. See https://docs.microsoft.com/en-us/powershell/module/microsoft.powershell.core/about/about_execution_policies?view=powershell-7 and use PowerShell to set the execution policy to RemoteSigned.
(kudos to Patrik Křepinský who reported that!)

WARNING: Windows 10 are having actually two powershells (x64 and x86) and depending on your VSCode you might need to change execution policy in both of them, see: https://stackoverflow.com/questions/4037939/powershell-says-execution-of-scripts-is-disabled-on-this-system
(kudos to Arthaka who reported that!)

7.2.3 tsconfig.json

Another place where `tsc` loads configuration from (by VSCode) is `tsconfig.json`.

```
{
    "compilerOptions": {
        "target": "ES6",
        "module": "AMD",
        "sourceMap": true
    }
}
```

There are two important stuff here. `"module"` is telling TypeScript to utilize AMD JavaScript module layout, but we will ignore that for now. And `"sourceMap"`, which is pretty important

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

as that will create a mapping between TypeScript code and transpilled JavaScript code, which is required to make the debugging in TypeScript work.

7.2.4 tsDefinitions folder

Now if you are new to TypeScript you might be wondering how it works. In short, you have JavaScript code for which you provide “type definitions” via `.d.ts` files. These are stored within `tsDefintions` folder. These definitions allow you to create TypeScript code (our game) that interface plain JavaScript code (Phaser3 framework). TypeScript compiler does not check whether .d.ts definitions are correct, it just takes them as “interface” to whatever JavaScript code you plan to use with that. In the end, TypeScript gets transpilled to JavaScript and all those types does not matter as long as all methods and properties matche during calls.

7.3 Compiling and running the example

7.3.1 src folder

Here we have the code of the game. Notice ugly naming ZGame.ts, the problem is, that this code is not modularized correctly. We are putting everything into single module called Shooter that gets compiled by TSC sequentially and as ZGame (main Phaser configuration) is referencing Play and Boot scenes, it must be the last file. The same logic applies to Play scene that is referencing stuff from Bullet and Enemy.

For now, let’s ignore the code in there. It’s not target to learn Phaser3 yet.

7.3.2 bin folder

Last but not least, the bin folder where we have the output of tsc stored within `js/game.js` and `js/game.js.map`.

More importantly it contains `assets` as a subfolder with images and `index.html`, which we need to inspect.

Follows the contents of index.html file.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Shooter example</title>
        <script src="./js/phaser.js"></script> <-- 1. -->
        <script src="./js/game.js"></script>   <-- 2. -->
    </head>
    <body>
        <h1>Shooter example</h1>
        <div id="content"></div>
    </body>
```

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

```
    <script>
        <-- 3. -->
        window.onload = () => {
          var game = new Phaser.Game(Shooter.gameConfig);
          window.focus();
        }
    </script>
</html>
```

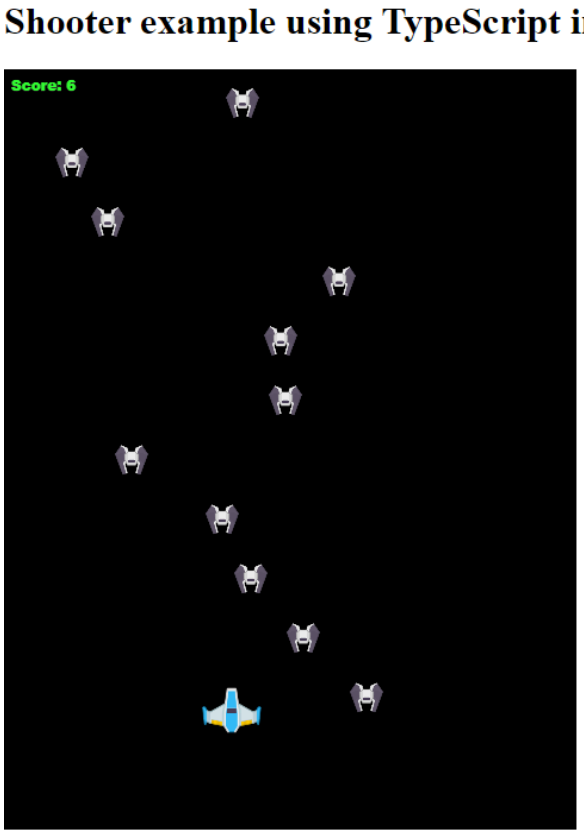
Lines in bold are important.

Ad 1., we import `phaser.js` that contains the Phaser3 framework.
Ad 2., here we import the code of our game as compiled by `tsc`.
Ad 3., we are creating new Phaser instance using `Shooter.gameConfig` that is defined within `ZGame.ts`.

7.3.3 Running the game

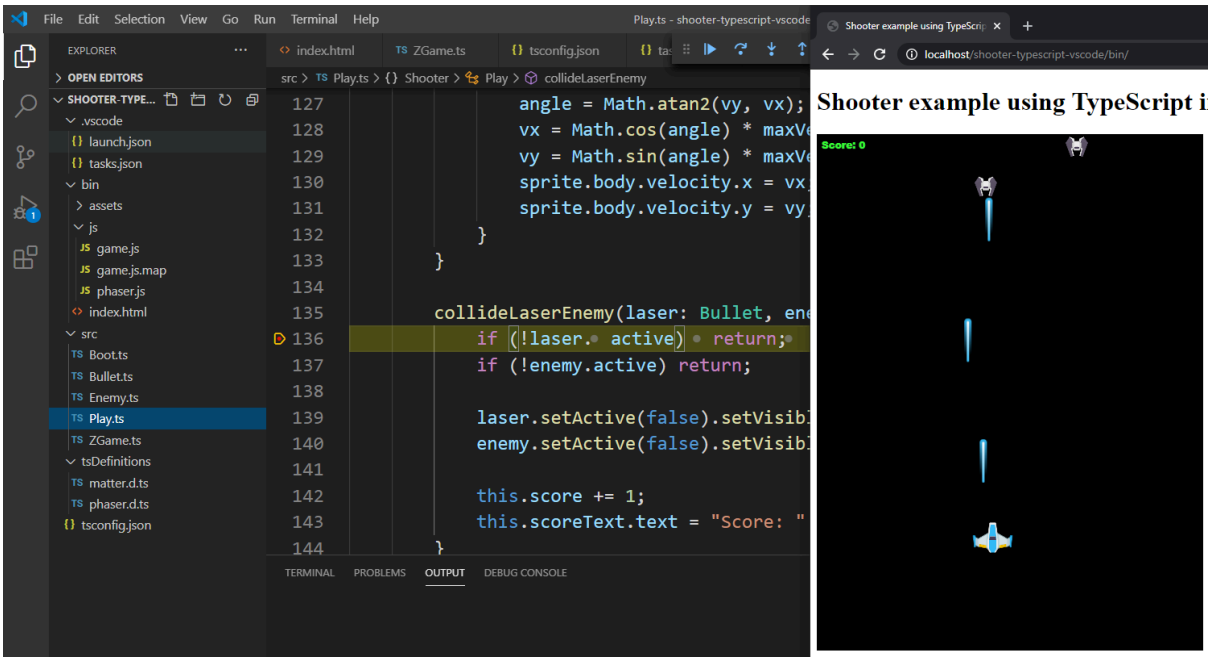
Now putting it all together, if you change the .ts sources, you can invoke `tsc` via Ctrl+Shift+B and then run it via F5.

Hopefully, after hitting F5, the game will show up.



And you should be able to breakpoint TypeScript code!

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.



This is neat :)

Step 8: Fully modularized Shooter that uses NPM for dependency hell

Last but not least, the modules. In Step 7, we might be completely happy with the setup and dive into coding. However, we will very quickly run into problems with having the game in multiple files; try it yourself, in the project in the previous step try renaming `ZGame.ts` to `Game.ts` and try to compile it. It will fail saying it does not recognize the “Play” scene and that’s because `tsc` will start compiling Game.ts before `Play.ts`.

How to solve this situation? Using modules, which TypeScript supports as described here. Sounds easy in theory but there is a caveat. JavaScript does support modules out of the box now but I could not make them work with TypeScript and Phaser. Therefore, we need to use older approach via custom module systems. There are multiple such systems and we will be using AMD via requirejs.

If you opened some of the pages above you might become perplexed, but you do not need to be. TypeScript can automatically translate import/export/module declarations into AMD, which will be then gracefully consumed by requirejs, it just bits of (correct) boilerplate code.

Additionally, we will include NPM dependency management (so you will be able to automatically download all required dependencies like Phaser3 itself).

There is only one small caveat … you need to manually patch `phaser.d.ts` (TypeScript definition file) as it contains the wrong case of a “phaser” module that needs to be “Phaser”. But it’s easy to do once you know where to do the path.

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

So we need to go through the following steps to make it all work:

1. Download the last example and unpack it
2. Download dependencies through npm
3. Patch `phaser.d.ts`
4. Profit!

8.1 Downloading last example

Download the template from here:
https://drive.google.com/file/d/1uDiHBs_vJBDuWNCbQ3pWeIAWVQ_ZjB8C/view?usp=sharing
And unpack it into your `webroot` folder.

8.2 Downloading dependencies through npm

Now open your console and navigate to the folder `shooter-typescript-vscode-module` you unpacked into your `webroot`.

And then simply execute:

```
npm install
```

This will create folder `node_modules` and fill it with required dependencies as requested (described) in `package.json` file.

WARNING: For some reason, npm did not install .d.ts file for requirejs for me, so I had to manually execute also:

```
npm install --save @types/requirejs
```

Which is required by `tsc` to be able to compile one of .ts file that interfaces requirejs.

8.3 Patching phaser.d.ts

Finally, in order to match cases during imports/exports we need to patch TypeScript definition of `phaser.d.ts`.

Locate `phaser.d.ts` file inside:
`shooter-typescript-vscode-module\node_modules\phaser\types\`

Open the file and scroll to the end of it.

You will find following definition here:

```
declare module 'phaser' {
```

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

```
    export = Phaser;

}
```

All you have to do is change ‘phaser’ to ‘Phaser’ like this:

```
declare module 'Phaser' {
    export = Phaser;

}
```

8.4 Profit!

Now you are ready to compile it all! Open the folder `shooter-typescript-vscode-module` within VSCode.

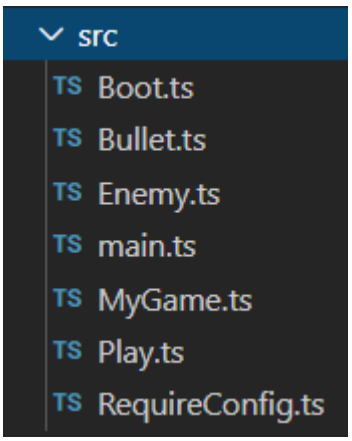
You can hit Ctrl+Shift+B and invoke `tsc` and then F5 to run it (provided you do not need to adapt url in `launch.json`), go ahead (fingers crossed). Note that you have to invoke **`debug-build`** NPM script, which is using AMD module packaging.

But the real beauty lies in the code :-)

The Modules and import statements

What is the biggest difference from the example in Step 7 are the modules and imports within .ts code.

Here is the sources listing:



And for instance in `Play.ts`, we can see this difference:

```
import * as Phaser from "Phaser";
import Bullet from "./Bullet";
import Enemy from "./Enemy";
```

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

```
export default class Play extends Phaser.Scene {
```

Imports! Export! We are now able to split the code into separate files and classes (one class per file is advised) and use import statements to define file dependencies, which are then handled automatically by requirejs. Roughly requirejs will a) load all files at once noting their requirements (imports), b) when invoking something from some file, it satisfies those requirements.

To quickly comment on the files:

`Boot.ts, Bullet.ts, Enemy.ts, Play.ts` … Shooter game classes

`MyGame.ts` … just a wrapper for default Phaser Game class, might get handy

`Main.ts` … sort of a main method, we fire up the Phaser framework in there upon loading

`RequireConfig.ts` … requirejs configuration (description is beyond this tutorial).

And as usual, breakpoints still work!

Profit :-)

Acknowledgement

Material has been produced within and supported by the project „Zvýšení kvality vzdělávání na UK a jeho relevance pro potřeby trhu práce“ kept under number CZ.02.2.69/0.0/0.0/16_015/0002362.

EVROPSKÁ UNIE
Evropské strukturální a investiční fondy
Operační program Výzkum, vývoj a vzdělávání