CEF in GTA:N / GTANET

Introduction

Goal of document

The aim of this document is clarify as much as possible regarding CEF in GTA Network (will be shortened from now on to "GTA:N") modification of GTA V. Seeing as the wiki pages provided by GTA:N for most things are at the moment (March 2017) slightly vague, it is up to community members to improve that by writing our own tutorials and documentations, perhaps edit the wiki-page ourselves. It is strongly recommended that we, at least during the beta, as community members of GTA:N share as much knowledge in-between each other and put silly "win" mentalities aside at least until GTA:N has grown bigger and can actually afford internal competition. As stated before, the goal with this is to explain as much as possible about CEF and how it can be used in GTA:N.

Furthermore, I created this tutorial because I believe it is something that I needed when I started with CEF (a few hours ago) which would have made me know things a lot faster. Also this guide was written for my own purpose of learning, mainly also expressing different program-terms as well as coding. It was fun and I hope you, whoever goes through reading 4500 words, gets to learn at least something!

Limitations

This guide, tutorial and documentation will be limited in terms of explanations due to too much having to be covered if everything was explained. Simple programming language syntax will therefore not be explained, but may sometimes be explained depending on situations (for instance when something unobvious is being scripted). This document will not only explain CEF but also how to get started with linking different files, but please keep in mind that CEF is the "end-product" that we want to have explained in this document. This document will not display how to install any editor neither how to download the GTA:N server package.

Requirements from your end

To completely follow this guide, you will most likely need Visual Studio. You can use other editors but it is highly recommended to use Visual Studio.

Sources

Nothing in this document will be referenced in terms of where the information was found or how it was acknowledged as that would be a waste of time for most readers. Instead I will place a few links or a few names that might be recognizable:

- GTA:N Wikipedia page
  - https://wiki.gtanet.work/index.php?title=Main_Page
- Jingles' *The Godfather* solution
  - https://forum.gtanet.work/index.php?threads/the-godfather.2137/
- TakeiT's *Simple CEF Tutorial*
  - (https://forum.gtanet.work/index.php?threads/simple-cef-tutorial.1808/)

While none of these projects, threads or documents were directly copied, they did have at some point some sort of influence on what has been written and therefore I want to give my sincere gratitude to the people behind the creation of these documents and projects.

Get started

My files

The very first thing to do is to setup all the files that will be needed for this tutorial and documentation. Do know that in reality you will not know what files you will always need from the beginning but in order to make you, the reader, understand as much as possible according to this guide, I have chosen to go down this path.

Step 1: First step is to open up Visual Studio and open up a new project. A new window will pop up and to the upper right side you can see a search bar, search for "solution". Choose the item that has "Visual Studio Solutions" text next to it. Go ahead and create a name for your new solution and save it down somewhere where you will have access to it. My path was "C:\GTANetwork\server\resources" since I have GTANetwork installed directly under C: directory. Below is an image of how it may look like.

[spoiler][img]http://i.imgur.com/N0qvriO.png[/img][/spoiler]

Step 2: On the right bottom hand-side on your Visual Studio you will see different tabs. Click on the one named "Solution Explorer" and right click on your newly created solution, select "Add" and then "New Project". Search for "Class library" in the search bar just as you did in step 1. Choose the one that has the text "Visual C#" on the right side of the item. The name of this library will be your resource, you will have to remember that a few steps ahead, other than that it's not too important of what to name it, but give it a fitting name. Because I lack creativity, I named mine *CEFResource*.

[spoiler][img]http://i.imgur.com/qgZdlY0.png[/img][/spoiler]

Step 3: You should be seeing a bunch of new stuff (only four new items) in your *Solution Explorer*: your resource (class library), properties, references and a *Class1.cs* file. Go ahead and remove that last one. Right click now on your resource (class library) and select "Manage NuGet Packages…". Go into the "Browse" tab and search for "GTANetwork". Two packages should appear (unless you are reading this in the future and GTA:N has gone worldwide and has now a lot of packages!), something similar to what the picture below shows. Go ahead and install "gtanetwork.api" and "types-gtnetwork". Once installed, they should have that green check-mark as displayed in the picture below as well.

[spoiler][img]http://i.imgur.com/tYRjimw.png[/img][/spoiler]

Step 4: Now the last step of file creating is initialized! All we have to do is create a few folders and a few files, bear with me. First off, go to your *Solution*

Explorer. Don't panic, there is supposed to be a folder called *types-gtanetwork* and there's supposed to be a *packages.config* now. We will ignore those, until forever probably.

Go ahead and right click on your class library/resource name and click on "Add". Pick to create a new folder and name it something like *Javascript*. In this folder we will have all our javascript files, even though we will just create one file, I am intending to use folder directories because of a purpose I will explain later on. In this newly created folder, create a .js file by right clicking on the folder and selecting "Add". Select "New item" and search for "javascript" in the search bar. Pick "JavaScript File" and name it something fancy; I named mine *CEF.js* because it will handle all the CEF things, acting like some sort of "controller".

Now create a new folder under your class library and name it *Designs*. Inside this folder, create a HTML file (by search like above for javascript but this time search for HTML) and name it *blackbox.html*. While you are inside *Designs*-folder, also add a CSS file (use same method to find the correct extension type file as for HTML and javascript, but search for "style" only and pick "Style Sheet"). Name this file *main.css*.

Now we will add the file that will hold the necessary in-game commands that we will use to test out the CEF functions. Under your class library, create a new C# file (.cs) and name it *CEFCommands.cs*.

The absolutely last thing you will have to create, in terms of files, will be the meta.xml file. Under your class library, add a new file (search for XML) and name it *meta.xml*. Do know that this must have that name in order to work properly.

With all this being done, your *Solution Explorer* should look something like this:

[spoiler][img]http://i.imgur.com/XNE4kw6.png[/img][/spoiler]

Step 5: Right click on your class library and go down to properties. Click on the "Application" tab and find "Target framework". Select the ".NET Framework 4.5.2". If you read this in the future, then a newer framework target might be needed, there should be a warning coming up once you try to build your project of which target is required.

[spoiler][img]http://i.imgur.com/ovRMK0U.png[/img][/spoiler]

Connecting the files with one another

Before we can start coding, it is important to know how to connect the files with one another. The connection between the files will look something similar to this:
HTML -> CSS
JS -> HTML
CS <-> JS
XML -> CS, JS, HTML, CSS

To structure this according to the files, of how we named them:
blackbox.html will call/include *main.css*
CEF.js will call/include *blackbox.html*
CEFCommands.cs will call/include *CEF.js*
CEF.js will also call/include *CEFCommands.cs*
meta.xml will call/include all the above

By a simple google search, we can find out how to actually link a CSS file to an HTML file, but let's explain it (not the syntax) here anyways. Open up your blackbox.html file and under the head-tags, create a link-tag:

```
<link rel="stylesheet" type="text/css" href="main.css">
```

Since they are under the same folder (/Designs), you don't have to specify exact location of the file, otherwise you'd have to give the whole directory.

The only other file we will connect to another file is in the *meta.xml* file for the time being. When time comes, we will link the rest as well, but keep in mind of how the linking structure works for future references. Open up your *meta.xml* file. Delete whatever was in there and open up new meta-tags. There are three types of "important includes" that you will have to do: script, file and assembly.

Script: This type will include any server or client files, such as C# or javascript files. Javascript files are always client-sided and C# files are always server sided. There is also another type of script include you can make which has the language *compiled* but I won't go into that. The syntax looks like this:

```
<script src="DIRECTORY" type="TYPE" lang="PROGRAM LANGUAGE" />
```

File: This type will include any HTML or CSS files that you need the client to have. Note that all client-types (including script-type as stated above) will force the player to download them when joining the server, but this type ("File") is for HTML and CSS. The syntax is the following:

```
<file src="DIRECTORY" />
```

Assembly: This type will include any .dll files. Whether they are made by you (by compiling for instance your class library) or someone else, they will have to be included for certain functions to work. In this guide, we do not need these, but you might need them in the future when for instance storing data. The syntax is the following:

```
<assembly ref="DIRECTORY" />
```

So now that we know that, let's go ahead and add stuff to our *meta.xml* file. Knowing the information above, you are eligible to simply copy the code below since you know what it means now (more or less at least).

[code]

```
<meta>
  <info name="CEFResource" author="Hansrutger" />

  <script src="Javascript/CEF.js" type="client" lang="javascript" />
  <script src="CEFCommands.cs" type="server" lang="csharp" />

  <file src="Designs/blackbox.html" />
  <file src="Designs/main.css" />
</meta>
```

[/code]

I said before that I wanted to point out something regarding folders and this is an excellent example. In your *meta.xml* file it is absolutely crucial that you include the whole directory from the point where your resource/class library is located. As you can see, the *CEFCommands.cs* doesn't need any other source since it is placed literally under/in the class library, but *CEF.js* does need "Javascript/" directory because it's inside another folder. Keep this in mind, because it will occur multiple times in this guide and also when you are working alone. Last but least, the first line *<info … />* is added to place information about the resource (because every resource has one meta.xml file). These are fully optional!

To not forget: add the resource!

Perhaps something that is easily missed: you need to add your resource into a file called *settings.xml*. Go to your GTANetwork directory and open your server folder. In there you will find a file called *settings.xml*. Open it in any editor (right click and click on edit). By default you will see a few resources down almost at the bottom. Create a new resource-tag and add your resource name there.

```
<resource src="CEFResource" />
```

This was the necessary part of remembering your resource name. Now you can forget the name completely if you have that superpower to forget things.

Almost at CEF!

Creating test commands

We are now at the stage that we will start having fun! Open up your c-sharp file (*CEFCommands.cs*) and let's start programming.

Step 1: A perhaps quite silly thing I like to always do in my class files is to declare them public. I am not sure if they are by default public but I always assume everything by default is private and false (bools) so that is why. So instead of it being called "class CEFCommands" it is called "public class CEFCommands" if you didn't quite follow what I meant.

Step 2: First thing we have to do is to include a bunch of stuff. Remember those nuget's we downloaded and installed? They are being used here! Add the following libraries on the top:

```
using GTANetworkServer;
using GTANetworkShared;
```

This will mean that we will inherit classes and functions from those packages, more or less. While we have done this, also make sure that your *CEFCommands* class inherits the *Script* class by adding ": Script" in this manner:

```
public class CEFCommands : Script
```

Step 3: Once this is done, we will start creating our first test command. Every command has a function and an attribute attached to them. Long story short, an attribute is that weird thing you add before a function in C#. The function will look like this:

```
[Command("test")]
public void TestFunction(Client player)
{

}
```

This indicates that when a player, in-game, types in "/test" this function will be run. This is possible because we added the *CEFCommands.cs* file to the *meta.xml* before if you remember! The function name itself can be literally anything but it's recommended to name them similar to the command name

itself. For more information about commands specifically, please visit this link here:
https://wiki.gtanet.work/index.php?title=Getting_Started_with_Commands

Step 4: Alright, so what do we actually want to do in this command function, in order to actually create CEF stuff? In our .cs file, it doesn't take much effort to be honest. All we have to do is add one more line and we don't have to visit this file (for a while).

```
API.triggerClientEvent(player, "testjs");
```

But what does that mean? This means almost the same as the function name says: API will trigger a client event for player "player" called "testjs". Note that I have used silly names just to specify that you don't need to name them the same thing as the command name or command function. Either way, that was quite easy once you actually read it, wasn't it? So what this means is that somewhere, far, far away in a .js file, there will be a method called *onServerEventTrigger* which will be called and will handle this. This is the awesome link between the server and the client: study it and treasure it, because it's very important for ***your*** future projects, features and systems that you will most likely create.

CEF.js taking the order

Head over to your *CEF.js* file now and we will start coding a bit javascript.

Step 1: As I said above, "onServerEventTrigger" will be called, but there is a function in there that actually will be called, "connect".

```
API.onServerEventTrigger.connect(function (eventName, args) {
});
```

This is basically the only function you will have to have in your javascript file, that will handle requests from the server to the client, but just to organize it a bit better and make it look better we will create a few more functions later on. Inside the current function, add a switch that will handle *eventName* (aka "testjs" that we sent from the server).

```
API.onServerEventTrigger.connect(function (eventName, args) {
    switch (eventName) {
        case 'testjs':
            // do something
            break;
    }
});
```

So now, when the server calls the client-javascript function "testjs" it will be directed to this .js file and you can do what you want here (in our case we will start CEF scripting soon). I highly recommend only having one of these files because it becomes a lot more organized, unless you organize it in another way: do not create multiple .js files with *onServerEventTrigger.connect*. But if you do, suit yourself, it's just a recommendation.

Step 2: Create another function, in the same .js file, called "TestHandler". In this function we will be dealing with CEF stuff, in the next chapter. Once this function is created, we will call it from the switch so *API.onServerEventTrigger.connect* isn't too populated. Again I am doing it this way to keep the latest function mentioned as clean as possible because having a lot of cases in a switch, along with unnecessary code, can be annoying to read as a programmer. The whole *CEF.js* file should now look similar to this:
[code]
```
API.onServerEventTrigger.connect(function (eventName, args) {
    switch (eventName) {
        case 'testjs':
            TestHandler();
            break;
    }
});

function TestHandler() {

}
```
[/code]

CEF will show boring box!

If you have come this far in this guide, then you probably have skipped a few steps but hey, congratulations nevertheless. We are finally where we are supposed to be, kind of.

Still in CEF.js

Step 1: What is, when it comes to web design, probably the most important thing for a fellow programmer to do first? Well usually it's not this but it should be kept in mind at least: the resolution. It is absolutely crucial thing nowadays with so many different platforms: mobile phones, laptops, 4k screens which no one but rich people can afford, and so on forth. So when dealing with CEF, this will be a very important thing to get because when you know the resolution, you can start scaling your design according to the user, without

having too different sizes from client to client. In the function we start declaring this by doing the following code:

```
var res = API.getScreenResolution();
```

Now *res* will hold our resolution. To get the height and width, use the properties *Height* (*res.Height*) and *Width* (*res.Width*).

Step 2: The next thing we are going to do is to define the browser itself by adding a height and a width to that we will use. Before we do this, we want to make absolutely sure that there is a declaration **outside** the function, of the browser. We therefore add the following code, before the function:

```
var browser = null;
```

We then go back inside the function and we add a heigh and a width by simply performing the following:

```
browser = API.createCefBrowser(500, 499);
```

In this case we have intentionally used up 500 of the width and 499 of the height (just to identify which is which). Do know that this will not make a square unless your screen resolution is absolutely a square. In most cases, using 499x500 will make the height (499) a lot bigger than the width (500) even though the difference is only 1.

Theory: Because it's actually using 499 pixels out of let's say 1080 means that it will take up almost half of the screen on the height. But while 1920 (the width) is just a quarter (500).

Step 3: Waiting and setting position. First we must wait for the CEF browser to fully initialize by using the function *waitUntilCefBrowserInit* which is quite self-explained of what it does. After that, the positioning of the browser has to be done. By a simple calculation, we will place the the box in the middle of the screen. For math nerds:
[img]https://latex.codecogs.com/gif.latex?%28%28res.Height%20/%202%29%20-%20%28box.Height%20/%202%29%29[/img]

More or less the same thing is being done for width and by this we will get a centered box. To the code part of what to add next:

```
API.waitUntilCefBrowserInit(browser);
    API.setCefBrowserPosition(browser, (res.Width / 2) - (500 / 2), (res.Height / 2) - (499 / 2));
```

Step 3: Here comes the connection between files! Time to look at the connection-structure that we made earlier. It states that *CEF.js* will include or call stuff from *blackbox.html* and here it comes:

```
API.loadPageCefBrowser(browser, "Designs/blackbox.html");
```

Please do note again that we have to define the directory and not just the file! Just because you mentioned something in *meta.xml* it doesn't make you invincible against these sort of mistyping's (I sat myself trying to figure out what I did wrong for at least one hour, only to find out that I had to include the whole directory).

Step 4: Additional "fun stuff" to be added. One function will make the cursor show and the other will disable the chat. The latter one is optional, but for our guide to fully function please implement the first one!

```
API.showCursor(true);
API.setCanOpenChat(false);
```

The whole code in *CEF.js* should look similar to this:
[code]

```
API.onServerEventTrigger.connect(function (eventName, args) {
    switch (eventName) {
        case 'testjs':
            TestHandler();
            break;
    }
});

function TestHandler() {
    var res = API.getScreenResolution();
    browser = API.createCefBrowser(500, 499);
    API.waitUntilCefBrowserInit(browser);
    API.setCefBrowserPosition(browser, (res.Width / 2) - (500 / 2),
(res.Height / 2) - (499 / 2));
    API.loadPageCefBrowser(browser, "Designs/blackbox.html");
    API.showCursor(true);
    API.setCanOpenChat(false);
}
```

[/code]

HTML and CSS!

Step 1: Let's start designing some stuff! First off, open your html file, *blackbox.html* in your Design folder. Inside the body we will add something very easy: a text and a button, as well as a form. Create a simple paragraph with the text "Hello, World!" (yes… we've all heard that one before) and then under that add form-tags and a button:

```
<p>Hello, World!</p>
<form>
    <button id="button">Click me!</button>
</form>
```

Now, anyone with experience of web designing will want to make this a lot fancier and by all means, feel free to spend a few hours on perfecting a button and a paragraph but for the sake of this long tutorial: let's move on.

Step 2: We will add some javascript inside the HTML file. If you didn't know that, then you know now. You can do that, but please don't get any silly ideas of actually using HTML files to try to call methods such as *onServerEventTrigger.connect* because, just don't. I will strangle you, it will look bad structure-wise. Basically what we want to do is to know when someone clicks that button and then do something. Simple enough? Let's go:

```
<script>
    document.getElementById("button").onclick = function () {
        resourceCall("BtnClicked");
    }
</script>
```

There are multiple ways of checking this, but I just took the shortest path here. What is the interesting part in this snippet is however *resourceCall*. What does that actually do? Well it looks for a javascript function called "BtnClicked" in this case. This is basically a way to connect from HTML to Javascript, but it is actually Javascript to Javascript if you think about it (hence it's not in the connection structure). So what do we want to do next then? Well simple, we want to add a function that handles something. But before we do that, we need to put at least a bit of design into this HTML file. There was connection that I mentioned between HTML and CSS, which we already made. If you look in the head-tags of the HTML file, you should see the following code (unless you didn't do as I asked you to):

```
<link rel="stylesheet" type="text/css" href="main.css">
```

What this actually does is include any CSS code from *main.css* to be applied on this HTML file.

Step 3: With that being said, let's move on to *main.css*. By default there will be a "body" there already. All we have to do is add a simple line of code:

```
background-color:rgba(0, 0, 0, 0.3);
```

Again, in reality you will probably spend at least one hour getting the design in good shape, but for the purpose of this guide, we keep things ugly! The full *main.css* should look like this:

[code]

```
body
{
    background-color:rgba(0, 0, 0, 0.6);
```

```
}
```
[/code]

The full *blackbox.html* should look like this:
[code]
```
<head>
    <meta charset="utf-8" />
    <title></title>
    <link rel="stylesheet" type="text/css" href="main.css">
</head>
<body>
    <p>Hello, World!</p>
    <form>
      <button id="button">Click me!</button>
    </form>
</body>
</html>

<script>
    document.getElementById("button").onclick = function () {
        resourceCall("BtnClicked");
    }
</script>
```
[/code]

CEF.js again

Now that the html and css part is fully done, we only have a few steps left before we are completely done. What do we actually want to do when the player clicks the button? We want to hide the CEF design because it's there without much use. After that we also want to go back to the server (*CEFCommands.cs*) just to prove a point that we can move however we want along sides our files, because we own them.

Button clicked event

We want to create that "BtnClicked" event and by creating a function called exactly the same, that's how we accomplish that. Inside this function all we want to do is to close the browser, stop the use of the browser cursor and also enable the chat possibility for the player (if you turned that off before). We're adding the following code in *CEF.js*:

```
function BtnClicked() {
    API.showCursor(false);
    API.setCanOpenChat(true);
    API.destroyCefBrowser(browser);
}
```

Initialize client-server connection

To just show, as a last thing in this guide, that we are able to move back again toward the server-sided files we will create the connection by using the opposite function of what we used before (*triggerClientEvent*) if you remember it. The one you use on the client side, to call the server, is instead called *triggerServerEvent*, yes a surprise. This one triggers a method on the server side called *onClientEventTrigger*.

Step 1: Still inside *BtnClicked*, add the following:

```
API.triggerServerEvent("BackToTheFuture");
```

After this, we will never again return to *CEF.js*. Therefore, the full code of this file should look like this:

[code]

```
API.onServerEventTrigger.connect(function (eventName, args) {
    switch (eventName) {
        case 'testjs':
            TestHandler();
            break;
    }
});

var browser = null;

function TestHandler() {
    var res = API.getScreenResolution();
    browser = API.createCefBrowser(500, 499);
    API.waitUntilCefBrowserInit(browser);
    API.setCefBrowserPosition(browser, (res.Width / 2) - (500 / 2),
(res.Height / 2) - (499 / 2));
    API.loadPageCefBrowser(browser, "Designs/blackbox.html");
    API.showCursor(true);
    API.setCanOpenChat(false);
}

function BtnClicked() {
    API.showCursor(false);
    API.setCanOpenChat(true);
    API.destroyCefBrowser(browser);

    API.triggerServerEvent("BackToTheFuture");
}
```

[/code]

Step 2: Going back to the file we were at long time ago, *CEFCommands.cs*, we need to only do a few steps before being completely done. First thing we

have to add is a constructor to *CEFCommands.cs*. This is done by simply creating a public function that doesn't return anything (not void either) and it has the same name as the class name:

```
public CEFCommands()
{
      API.onClientEventTrigger += OnClientTriggered;
}
```

Inside this constructor you can see that we are calling an event to happen. Whenever *onClientEventTrigger* is called, we trigger the function called *OnClientTriggered*.

Step 3: Which leads us to the very last step of this guide, adding this function. This will look very similar to what we did in the *CEF.js* file before, with the switch, but this time in C#.

```
public void OnClientTriggered(Client player, string eventName, params
object[] arguments)
{
      switch (eventName)
      {
            case "BackToTheFuture":
                  API.sendNotificationToPlayer(player, "We're back where
                  we started.");
                  break;
      }
}
```

This will of course trigger a notification being sent to the player.