WebGL-based Image Processing through JavaScript Injection

Tim Meyer
Universität der Bundeswehr München
Neubiberg, Germany
tim.meyer@unibw.de

Gabi Dreo Rodosek
Universität der Bundeswehr München
Neubiberg, Germany
gabi.dreo@unibw.de

Daniel Haehn
University of Massachusetts Boston
Boston, USA
daniel.haehn@umb.edu

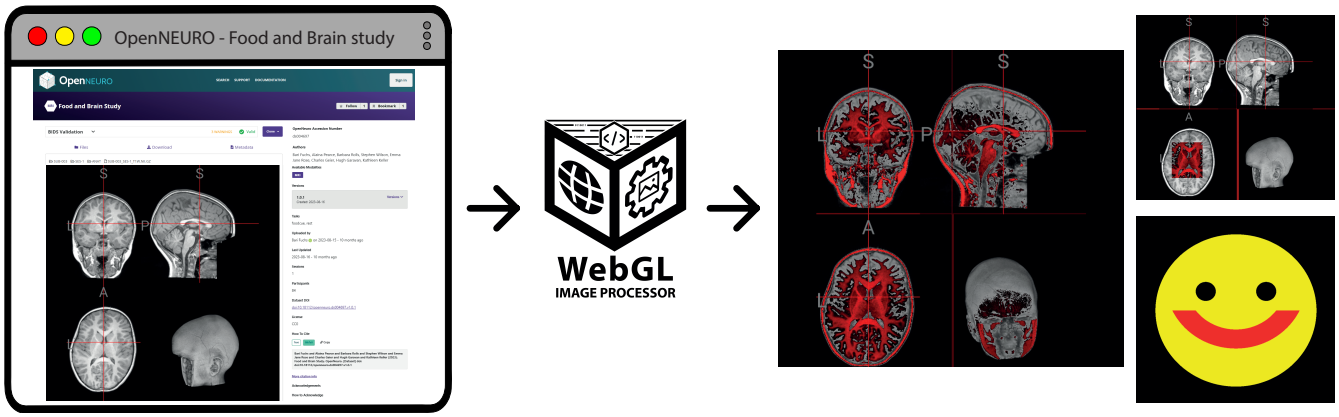


Figure 1: We hijack the WebGL context of any external website to perform GPU-accelerated image processing through JavaScript injection. This allows client-side modification of the rendered scene without access to the web server. Examples here show pixel-based color highlighting, full visual replacement (smiley), and region of interest selected processing. Further examples, a demo video and code can be accessed at: https://meyerstim.github.io/WebGL-Image-Processor/

ABSTRACT

Can we modify existing web-based computer graphics content through JavaScript injection? We study how to hijack the WebGL context of any external website to perform GPU-accelerated image processing and scene modification. This allows client-side modification of 2D and 3D content without access to the web server. We demonstrate how JavaScript can overload an existing WebGL context and present examples such as color replacement, edge detection, image filtering, and complete visual transformations of external websites, as well as vertex and geometry processing and manipulation. We discuss the potential of such an approach and present open-source software for real-time processing using a bookmarklet implementation.

CCS CONCEPTS

• **Computing methodologies** → **Image processing**; • **Security and privacy** → *Browser security*.

KEYWORDS

WebGL, WebGL2, web-based, Image Processing, Vertex Processing, JavaScript Injection

ACM Reference Format:
Tim Meyer, Gabi Dreo Rodosek, and Daniel Haehn. 2024. WebGL-based Image Processing through JavaScript Injection. In *The 29th International ACM Conference on 3D Web Technology (WEB3D '24), September 25–27, 2024, Guimarães, Portugal*. ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3665318.3677163

WEB3D '24, September 25–27, 2024, Guimarães, Portugal

ACM ISBN 979-8-4007-0689-9/24/09
https://doi.org/10.1145/3665318.3677163

1 INTRODUCTION

WebGL [Jackson and Gilbert 2017], a JavaScript Application Programming Interface (API) for rendering interactive 3D graphics within any compatible web browser has revolutionized web-based graphics for visualization and compute. Every WebGL application uses this API to create a context that allows access to OpenGL-style buffers to render and process content. Typically a web-server hosts HTML and JavaScript code for interactive enduser consumption in the client/web-browser.

Our framework, however, introduces an approach by hijacking the WebGL context of such existing external website: we can intercept WebGL commands issued by any web application and modify the rendering output in real-time. We inject JavaScript into the target webpage to capture the WebGL context and redirect the rendering output to off-screen buffers for generic processing. By reading out the framebuffers and temporarily storing them in a texture, we can apply various image processing algorithms before

drawing the results back onto the original scene to be visible to the user.

By leveraging this technique, we can achieve real-time and GPU-accelerated image processing and scene modification of the website's content even without access to the original code. Scene modification also includes the manipulation, transformation and modification of vertex objects in the scene. These are the objects and geometries from which the respective content is constructed. In a further step, these objects are then processed and colored by the fragment shader. At this stage, the manipulation allows the objects to be scaled in the X and Y directions, so the size of these elements can be changed in real time, again without having access to the original data or rendering pipeline Our framework is open-source and available for further development and customization, providing a powerful tool for developers and researchers to explore the capabilities of WebGL for real-time image processing within the browser environment.

2 RELATED WORK

WebGL is a JavaScript API that allows for rendering interactive 2D and 3D graphics within any compatible web browser without the need for plug-ins. It leverages the power of the GPU to execute code directly on the hardware, enabling high-performance graphics rendering. WebGL operates through the HTML5 canvas element, providing a programmable environment for rendering shapes, images, and complicated 3D scenes [Ghayour and Cantor 2018] and many abstraction libraries exist to simplify development (Three.js [Three.js 2024], Babylon.js [Catuhe and Babylon.js 2024], XTK [Haehn et al. 2014], Niivue [Niivue 2021]).

Besides pure visualization, WebGL is also used in various ways to perform image processing and GPU computations directly within the browser. For example, TensorFlow.js [Smilkov et al. 2019] and ONNX.js [Microsoft Corporation 2024] enable developers to execute machine learning models and perform GPU-accelerated computations directly in the browser using WebGL bindings. A noteworthy end-user application in the medical imaging domain is Brainchop.org [Masoud et al. 2023] that allows the real-time segmentation of Magnetic resonance imaging (MRI) and computed tomography (CT) scans within a self-contained environment.

However, our framework can dynamically expand such functionalities by modifying the rendering output of any WebGL-based application without any server-side access. This allows client-side image processing and modification of existing scenes.

Most similar to our work is Spector.js [Spector.js 2017], a debugging and profiling tool designed for WebGL developers. This tool allows capturing and analyzing WebGL commands to aid in code optimization and debugging. Unlike other tools, Spector.js can interact with existing WebGL contexts by injecting itself into the rendering pipeline. However, its primary focus is WebGL development, and it does not modify the rendered WebGL content.

3 CONCEPT

Our framework can intercept and manipulate WebGL draw calls on external websites. By injecting JavaScript code into a browser session, we gain access to the WebGL context. We can then redirect the rendering output by reading the original framebuffers and re-writing their content to off-screen buffers or textures for processing. This allows for real-time image modifications, such as applying fragment shaders to alter colors, perform edge detection, or enhance visuals with various other processing algorithms. The processed content is then rendered back onto the canvas, making the modifications visible to the user without changing the original website.

In addition, we can also manipulate the vertex data and objects, i.e. the structures, arrays and values that construct the scene. These can also be changed and manipulated. Among other things, this enables scaling and other transformations in the scene.

3.1 Framework Overview

The framework consists of several components designed to interact seamlessly with the WebGL rendering pipeline. The process begins with JavaScript injection into a web page, which sets the stage for capturing and manipulating WebGL contexts. This multi-stage processing can also be seen in Figure 2 and is carried out as follows:

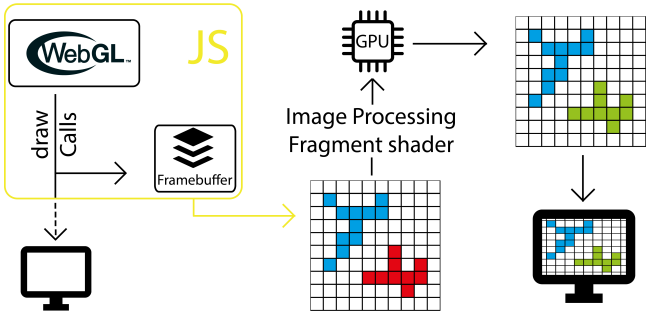


Figure 2: Processing pipeline to hijack and manipulate existing WebGL context on an external website to perform client-side image processing and update contents in real-time without web server access.

(1) JavaScript Injection: This step involves dynamically inserting JavaScript code into web pages using a browser bookmarklet (wrapped JavaScript code). This injected code enables the capture and manipulation of WebGL contexts.
(2) WebGL Context Wrapper / Context hijacking: The injected code is essential for hijacking and manipulating WebGL contexts. This process involves wrapping WebGL functions to monitor and alter their execution. We intercept methods such as *drawElements* by creating a class that encapsulates WebGL functionality. This allows us to capture and redirect WebGL commands, giving us control over the rendering process.
The wrapper replaces the original WebGL functions with custom ones that include additional logic. For example, when the *drawElements* function is called, our wrapper function is executed instead. This wrapper function can log the call, modify its parameters, or completely override its behavior. Doing so allows manipulation of the rendering output by interception before the GPU processes it.

(3) Framebuffer Extraction and Texture Mapping: Framebuffer extraction and texture mapping are crucial steps for isolating and preparing rendering content for further processing. Before the content is rendered to the visible canvas, the framebuffer content is captured and copied into a texture. This involves extracting the framebuffer data used in the WebGL pipeline and mapping it to a texture for intermediate storage.
The process begins by creating and binding a framebuffer object to the current WebGL context. Once bound, the framebuffer data, which includes the rendered scene, is copied into a texture. This step isolates the rendering content from the original webpage and stores it in a format ready for further manipulation. By storing the framebuffer data in a texture, we ensure that it is available for subsequent operations without interfering with the ongoing rendering pipeline.

(4) Image Processing with Fragment Shaders: Utilizing the captured texture as input, various image processing algorithms can be executed using fragment shaders. These shaders are small WebGL programs that run on the GPU and process individual pixels, enabling real-time, parallelized image manipulations. Fragment shaders can perform tasks such as dynamic highlighting, edge detection, color correction, and more. Examples of image processing tasks include full processing, region of interest selection, and content overwriting, as seen in Figure 1.
- Full processing: Complete Canvas / WebGL content is processed. Different shaders can be used to achieve various processing results.
- Region of Interest Selection: Processes only a specified part of the canvas, focusing computational resources on areas of interest.
- Content Override: Replaces the original content on the canvas with the processed output, such as rendering a smiley face onto the canvas.

(5) Rendering Processed Data: After processing, the original geometry is used to render the processed texture back onto the browser's canvas. This step involves re-executing the WebGL commands with the modified fragment shader, ensuring the processed content is drawn onto the original drawing area. This displays the modified image to the user.
The processed texture is then rendered back onto the original canvas using the original geometry, ensuring the modifications are visible. This step involves rendering the processed data onto the canvas, replacing the original content with the modified version. Doing so does not interrupt the original interaction methods or animation frames.

Our framework supports real-time updates by continuously monitoring interactions with the canvas. The entire WebGL rendering pipeline is re-executed whenever there is user interaction, such as modifying the camera.

With JavaScript injection and by using the WebGL API and bypassing any potential abstraction framework, this approach works independent of any chosen client-side or server-side software stack and across different websites and applications.

3.2 Image Processing with Fragment Shaders

Fragment shaders play a crucial role in the WebGL Image Processor framework by enabling pixel-wise processing of images directly on the GPU. This allows for highly efficient and parallelized image manipulations, essential for real-time applications. A fragment shader is a small program executed on the GPU for each pixel, allowing various image processing algorithms to be applied.

When a texture containing the image data is passed to a fragment shader, the shader processes each pixel individually. The processing can involve color adjustments, edge detection, dynamic highlighting, and more. By leveraging the GPU's parallel processing power, fragment shaders can efficiently handle large image data and perform complex real-time calculations.

More complex shaders can implement sophisticated algorithms for various image processing applications, enabling a generic real-time image processing and scene manipulation tool.

3.3 Vertex Transformation

We can also manipulate and change geometries using a vertex shader. Our experiments include the translation and scaling in X- and Y-direction but any type of processing is generally possible. This can be combined with the image processing using fragment shaders.

4 VISUAL EXAMPLES

The visual examples in this paper demonstrate various image processing techniques applied to external websites. Each example demonstrates the framework's capability to hijack WebGL contexts using JavaScript injection and perform real-time image manipulations. Examples include dynamic highlighting, region of interest selection, and content overwriting.

Once installed, our proof-of-concept bookmarklet allows the user to select different processing shaders using a combobox when visiting an external WebGL-driven website. The user can enable or disable processing with the *INSERT* button. Figure 3 shows this user interface injected in OpenNeuro.org and the application of Sobel Edge detection. Figure 1 demonstrates *Dynamic Highlighting* to color thresholded pixels within a selected region-of-interest.

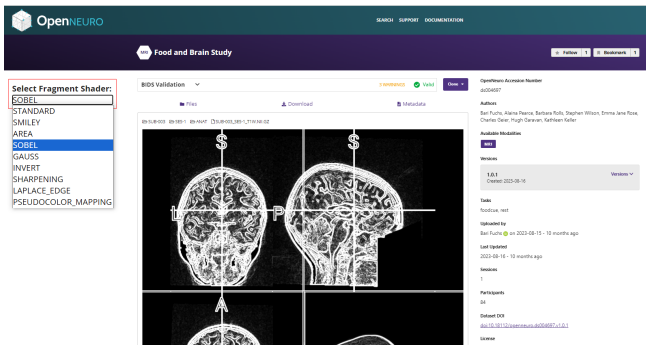
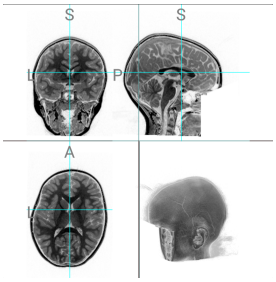
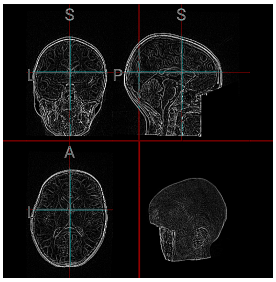


Figure 3: Sobel Edge detection filter applied to a sample dataset on the external OpenNeuro.org website that uses the WebGL-abstraction layer NiiVue.js without access to the server infrastructure or data.[1]



Color Inverter



Laplace Edge Detection

Figure 4: Fragment shaders for sample image processing (Color inverter and Edge detection algorithm) applied to existing WebGL context, without access to the server infrastructure or data, on the same external website as with Figure 3.[1]

Additional injected processing functionality is shown in Figure 4: the *Color Inverter* exchanges the RGB values for each pixel, and the *Laplace Edge Detection*.

And Figure 5 displays a grayscale to RGB conversion on a different external website.

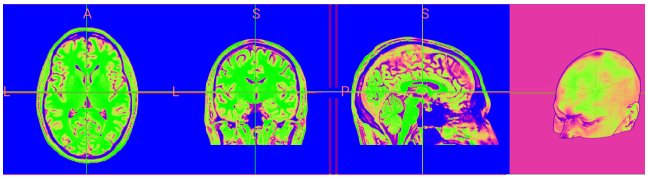


Figure 5: Pseudocolor filter to convert grayscale images to RGB-like images, applied on an existing WebGL context, without access to the server infrastructure or data, on a external website.[2]

Finally, Figure 6 shows the the transformation and scaling of the vertex data. There can be see a reduction of the vertex data in the Y direction by a factor of 0.5, i.e. its size is reduced by half. At the same time, a fragment shader is also used to apply the SOBEL edge detection algorithm. Both of the manipulations will be reapplied to the content in real-time each time any interaction or change occurs in the scene. As soon as the WebGL context is self-updating with a certain framerate, scaling and processing will be performed for each of these frames.

Further sample images of vertex manipulation and processing algorithms as well as a demo video showing the real-time performance can be found on our website.

5 LIMITATIONS AND FUTURE WORK

Hijacking WebGL contexts for real-time image processing can be slow or lead to the loss of WebGL context, especially on embedded systems with less memory. This is a significant issue for dynamic

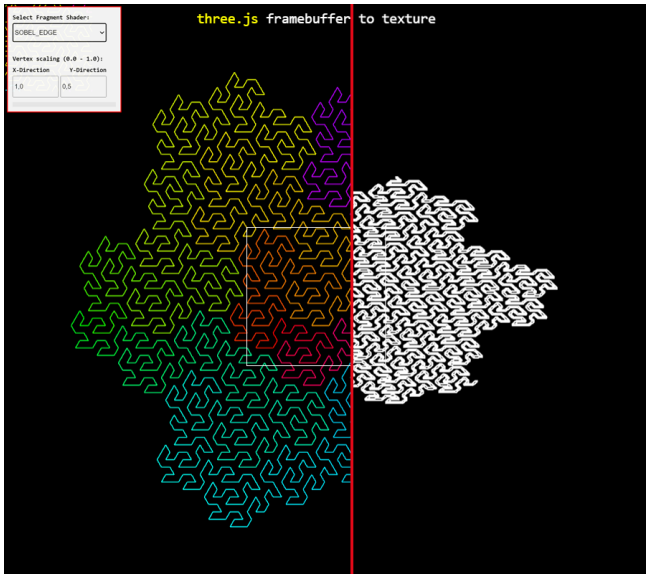


Figure 6: Splitscreen of original (left) and processed (right) WebGL content. Vertex manipulation of existing WebGL Objects to scale and transform in Y-Direction by the factor of 0.5. Additionally application of the SOBEL Edge detection filter. Everything is performed on an existing, interactive 3D WebGL context, without access to the server infrastructure or data, on a external website.[3]

content, where frames update multiple times per second, requiring frequent processing.

In addition to the aspect that the basic possibility of manipulating and changing the vertex elements has been shown by means of object scaling, there is of course further potential here to carry out more complex processing on the elements and geometries, based on the existing processes and results.

We also plan to explore potential security risks associated with dynamic content manipulation, especially in connection with WebGL's sandboxed GPU-access.

6 CONCLUSION

We demonstrate the capability to hijack external websites' WebGL context and dynamically interact with it through JavaScript injection. This approach allows for modifying the rendering output in real-time, enabling various image processing tasks such as pixel-wise color modification, edge detection, or full frame replacement directly within the client's browser. It also allows you to manipulate, transform and change the vertex and geometry objects from which the scene is constructed. We tested scaling and other transformations by injecting an additional vertex shader.

REFERENCES

David Catuhe and Babylon.js. 2024. *Babylon.js: a powerful, beautiful, simple, and open game and rendering engine packed into a friendly JavaScript framework.* https://www.babylonjs.com/

Farhad Ghayour and Diego Cantor. 2018. *Real-Time 3D Graphics with WebGL 2: Build interactive 3D applications with JavaScript and WebGL 2 (OpenGL ES 3.0), 2nd Edition.* Packt Publishing.

[1]External website: https://openneuro.org/datasets/ds004697/versions/1.0.1/file-display/sub-003:ses-1:anat:sub-003_ses-1_T1w.nii.gz
[2]External website: https://brainchop.org/
[3]External website: https://threejs.org/examples/webgl_framebuffer_texture.html

Daniel Haehn, Nicolas Rannou, Banu Ahtam, Ellen Grant, and Rudolph Pienaar. 2014. Neuroimaging in the Browser using the X Toolkit. *Frontiers in Neuroinformatics* 8 (2014). https://doi.org/10.3389/conf.fninf.2014.08.00101

Dean Jackson and Jeff Gilbert. 2017. WebGL 2 Specification. https://registry.khronos.org/webgl/specs/2.0.0/ retrived on 2024-05-29.

Mohamed Masoud, Pratyush Reddy, Farfalla Hu, and Sergey Plis. 2023. Brainchop: Next Generation Web-Based Neuroimaging Application. arXiv:2310.16162

Microsoft Corporation. 2024. *ONNX.js: run ONNX models using JavaScript.* https://github.com/microsoft/onnxjs

Niivue. 2021. *Niivue: a WebGL2 based medical image viewer. Supports over 30 formats of volumes and meshes.* https://github.com/niivue/niivue

Daniel Smilkov, Nikhil Thorat, Yannick Assogba, Ann Yuan, Nick Kreeger, Ping Yu, Kangyi Zhang, Shanqing Cai, Eric Nielsen, David Soergel, Stan Bileschi, Michael Terry, Charles Nicholson, Sandeep N. Gupta, Sarah Sirajuddin, D. Sculley, Rajat Monga, Greg Corrado, Fernanda B. Viégas, and Martin Wattenberg. 2019. TensorFlow.js: Machine Learning for the Web and Beyond. https://arxiv.org/abs/1901.05350

Spector.js. 2017. *Spector.js: Explore and Troubleshoot your WebGL scenes with ease.* https://github.com/BabylonJS/Spector.js

Three.js. 2024. *Three.js: JavaScript 3D Library.* https://threejs.org/

Received 16 June 2024; revised XXX; accepted XXX