An Interactive Web Gallery for Goal-based Caustics, Part II

Jad-Nicolas Khoury*
Supervisor: N. Thanikachalam
Professor: Dr. Mark Pauly

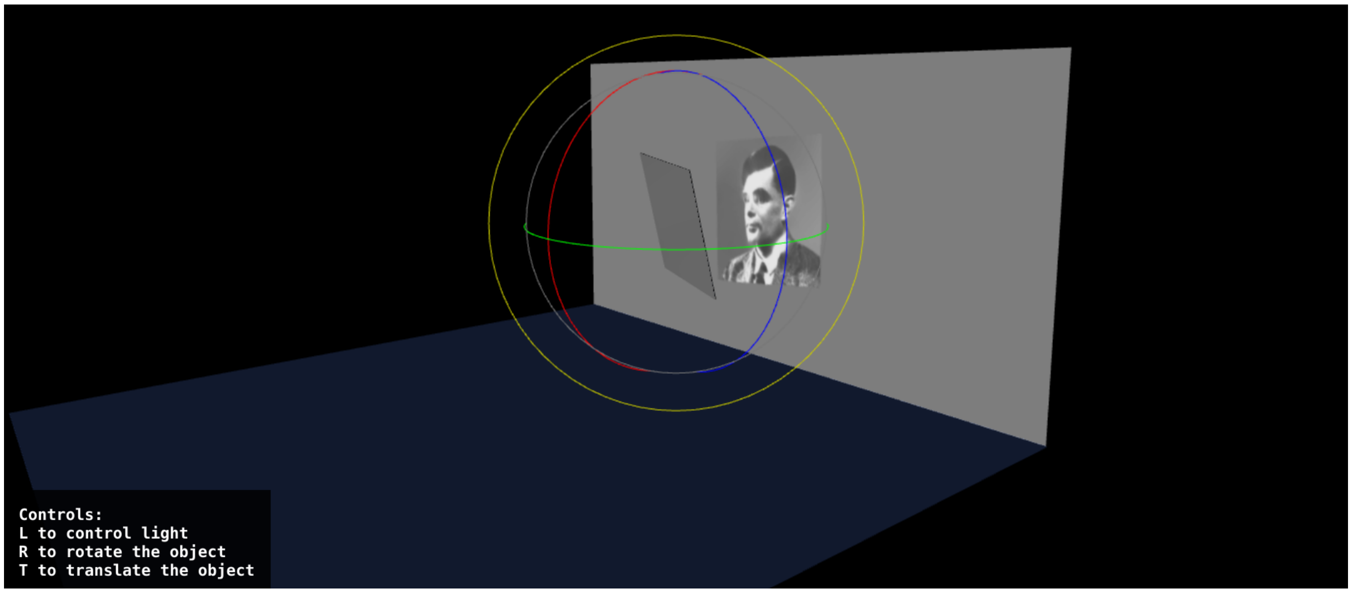


Figure 1: *Running WebGallery, in rotation mode, with lighting computed on server*

Abstract

This project follows the work done during last semester in the context of a Semester Project, with the LGG lab of the EPFL and in collaboration with the Rayform company, see [Khoury 2016] for the first report. The goal was to develop an interactive web-gallery displaying the real-world effect of Rayform products. These products are either reflective or refractive objects which, when in the right light conditions and position, project a pre-defined image as a caustic pattern. During the previous semester, an OpenGL/C++ Ray Tracing Engine was developed to render caustics in real time, and a Three.js program has been written to display the scene, and to allow interaction with the caustic object and the light source. What remained to be done was to distribute the OpenGL program on an AWS G2 instance, and implement everything needed to allow communication between the browser program and the Ray Tracing Engine.

Keywords: Distributed rendering, caustics, AWS

1 Introduction

As of the writing of this report, no available documentation or paper treat of partially distributing the rendering pipeline. As a result, a novel and complete system architecture had to be designed and developed for this project during this semester. A major part of the work accomplished has been to make all of the needed libraries and package run smoothly on an AWS G2 instance.
Most of the used technologies are fairly recent (G2 instances, Socket.io C++ client, etc ...), and the available ressources are extremely limited if not nonexistent in some cases, and the final implementation is the result of endless trial and errors. Nevertheless, the installations and implementations have been tested and yield working instances from empty Ubuntu images, and can be used as reference for anyone trying to run OpenGL or a socket.io C++ client on Amazon EC2 G2 instances. Detailed notes on implementation and installations are presented at the end of this report.

2 Overview

The global architecture of the system, depicted on fig.2, is composed of three parts:

- A Virtual Private Server (VPS, currently Digital Ocean) that hosts the HomePage,
- A Render Server, concretely implemented as an Amazon Web Services (AWS) G2 instance, in charge of serving the front-end application, hosting the Node.js server, and running the Ray Tracing Engine developed previously,
- The Front End Application that runs on the users browser and that encapsulates both the Three.js program and the socket.io client.

This architecture has been designed with this usage scenario in mind:
First, the user opens the homepage of the WebGallery, where he can find concise information about the project, and choose one of the existing demo meshes to launch the simulation with. As of the writing of this report, three meshes are available: a caustic mesh for the Turing portrait, a reflective mesh for the Turing portrait, and a refractive mesh for the Einstein picture and signature. When choosing a mesh, the homepage automatically launches the AWS instance, and notifies which mesh was asked. At launch, the AWS

*e-mail:jad-nicolas.khoury@epfl.ch

instance runs both its Node server and the Ray Tracing Engine, and the users browser downloads and runs the Front End Application. He can now interact with the position and rotation of the object or the light, and the Render Server automatically computes the new lightmap and transmits it to the Three.js program which then updates the displayed lightmap.

3 Render Server

Amazon Web Services Elastic Cloud 2 (AWS EC2) is a platform that offers on demand cloud computing by proposing a virtual cluster of computers, available with different hardware configuration to suit different computing needs. The user selects a hardware configuration that will correspond to the available ressources on the instance once its launched. The configuration used in this project is the G2.2 instance, enabling access to NVidia GRID GPUs on the cloud. More specifically, it offers:

- 1 Intel Xeon E5-2670 CPU
- 1 NVIDIA GRID GPU with 1,536 CUDA cores and 4 GB of VRAM
- 15GB of memory

At termination, all the content stored on the instance is deleted. To palliate this constraint, a system of AMIs (Amazon Machine Images) is available. It allows the user to create an image of the current state of any given instance, which will then be available when launching another instance. For example, one could create an AMI with everything prepared to run an OpenGL program, and then launch multiple AWS instances with this AMI to run different rendering programs.

3.1 AWS Instance

The first difficulty encountered when trying to implement the Render Server has been to prepare an Ubuntu image on a G2 instance to run OpenGL without any display attached.
Even if some AMIs and tutorials aimed at running OpenGL on AWS instances are available, many of them are outdated, and the NVidia AMIs that are supposed to be used in application like ours are not maintained anymore. For this reason, the AMI created for this project has been developed from scratch, with an empty Ubuntu AMI as starting point. Preparing a Linux instance to run headless OpenGL is more complex than one could expect, and needs some extra precautions and scripts to allow OpenGL rendering at boot time. Section 7.1 details all the installation and commands necessary to obtain an OpenGL ready AWS instance.

3.2 The OpenGL program

The OpenGL program itself did not change a lot from what was detailed in the previous report. The only part modified has been the implementation of the lighting of the screen that allows the use of the transmitted texture directly, and not to superpose it to the lighting done by Three.js. Figure 3 shows the lighting computed on the Render Server in the directional light and spotilght cases.

3.3 Socket.io C++ client

From the developers description, socket.io is a library that enables real-time bidirectional event-based communication. More concretely, this package allows to send objects and events from a client to a server, or vice-versa. In this project, the socket.io library is used communicate between the Render Server and the Front End Application. This entails transmitting events and sending parameters from one party to the other.
Originally, socket.io is a javascript library, but the developers recently made available a C++ client, that offers the same functionalities as the javascript client, but much less intuitively. The use of this package in a C++ program can be problematic, but the working solution is detailed in section 7.2.
In our Ray Tracing Engine, the use of this library implied the addition of a new class, called SocketHander, that takes the place of the actual socet.io client in a javascript application. In particular, this class is in charge of treating the incoming packets and relaying their content, should it be event or parameters, and also to wrap the scene parameters in a special data structure (required by the C++ client) and send them through the socket at some key events.

4 Front End

4.1 Node.js and Socket.io server

From the developper description, "Node.js is a JavaScript runtime [that] uses an event-driven, non-blocking I/O model."
In this project we use one Node server to host the homepage on the VPS, and another one to host the Front End Application on AWS and encapsulate the socket.io server. For the C++ client to send packets to the Front End Application like detailed before, it actually needs to connect to the socket.io server, and sent the packets to it. The server is then in charge of maintaining the connection to the different clients, and to treat the incoming packets. As a result, if the javascript client wants to send a position to the C++ client, it actually sends it to the socket.io server, which in turn relays it to the C++ client. Connection and disconnection events are also the responsibilities of the socket.io server.

4.2 Three.js

Three.js is a high-level javascript library that relies on WebGL and that greatly simplifies the implementation of real-time rendering on browser. Since the last report, the Three.js scene did not change a lot appart from a few optimisation concerning parameter checks and other similar functions. The Three.js program itself is also hosted on AWS, but is served by the Node.js server and run by the users browser.

4.3 Socket.io Javascript client

The Front End Application running in the browser also needs a socket.io client to be able to communicate with the C++ socket.io client implemented in the Ray Tracing Engine, through the socket.io server running on the AWS instance. Thankfully, installing and running a socket.io client in a javascript application is natively supported and requires almost no prior installation and no scripting.

4.4 HomePage

The homepage has been developed in HTML using a Bootstrap template and fig. 7 shows a part of it. It is hosted on a DigitalOcean droplet and served, as previously mentioned, using a Node server. The Node server is also in charge of launching the AWS instance using the AWS SDK, and redirecting the users browser toward the AWS instance IP address.

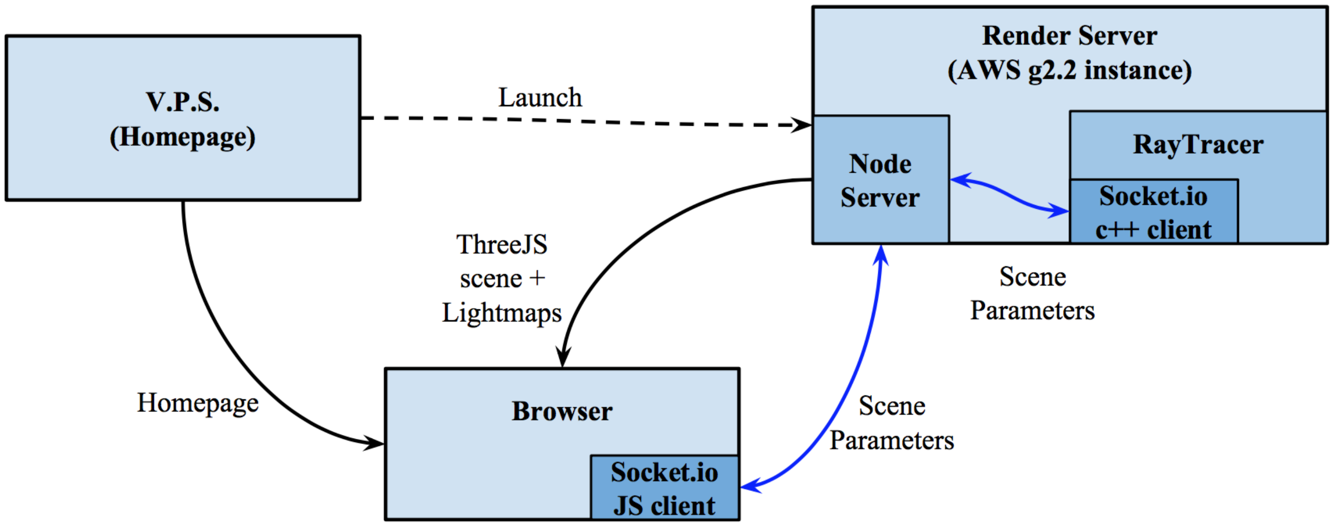


Figure 2: *System Architecture*

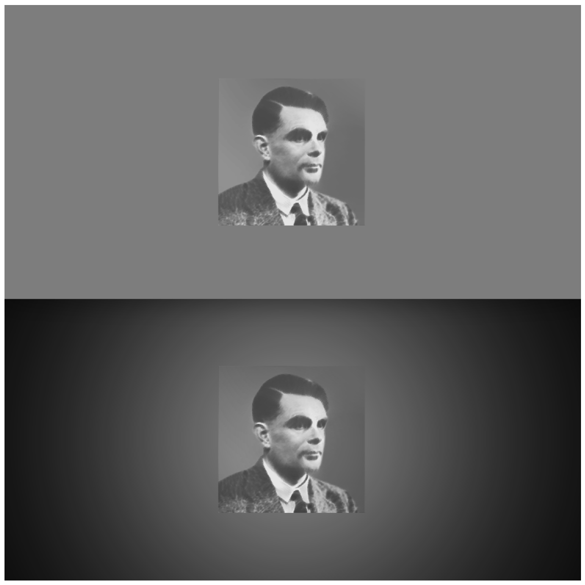


Figure 3: *On-server rendering of the screen lighting*
Top: Directional, bottom: Spotlight

5 Synchronous pipeline implementation

The implementation of the Three.js program and the Ray Tracing Engine implied some constraints that had to be respected in order to achieve a stable implementation:

- The initial parameters of the scene have to correspond to the parameters computed by the C++ program during the preprocessing step detailed in the previous report (allowing the implementation to be agnostic to the coordinate system as long as the mesh has the caustic surface along on the X-Y plane).
- When interacting with the simulation, the new parameters should be sent to the Ray Tracing Engine, which should in turn update its parameters, recompute the new lightmap and transfer the new image.
- The browser connection should be persistant, i.e. if the browser disconnect and reconnect, the simulation should pick up the scene as it was left.

For these reasons, implementing some kind of handshakes between the three parties was required. In this implementation, the NodeJS / socket.io server orchestrates the handshakes.

5.1 Initialisation Handshake

The first handshake happens when the two clients connect to the Node server. This handshake has to be agnostic to the order of connection, so the browser connecting before or after the C++ client should have no effect on the execution. For this reason, the Node server on the AWS instance has to maintain information about what is currently connected, and inform the other parties. To do so, the Initialisation Handshake detailed on fig. 4 has been implemented. At the end of this handshake, the three parties are synchronised and the simulation can begin.

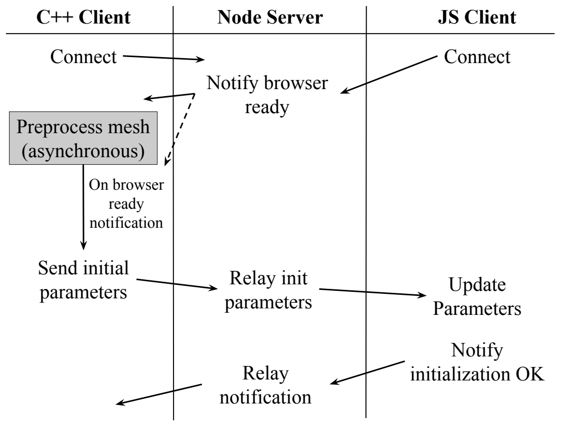


Figure 4: *Initialization Handshake*

5.2 Parameter Exchange

During the simulation, the javascript client has to send to the C++ client the parameters that are updated by the user (e.g. by translating the mesh). Then, the Ray Tracing Engine has to update the lightmap, sends the path of the image to the Three.js program, which in turn loads the image and updates the texture, as pictured in fig. 5. This is possible because the Three.js program and the Ray Tracing Engine (and therefore its socket.io client) are hosted on the same AWS instance. Hence, updating the texture with the new image breaks down to reading an image file on the server that hosts the Front End Application.

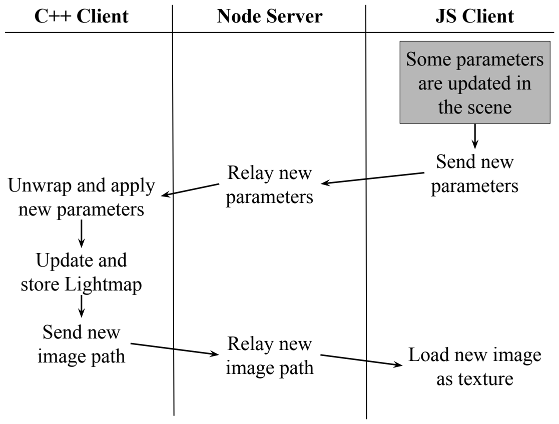

C++ Client
Node Server
JS Client
Some parameters are updated in the scene
Send new parameters
Relay new parameters
Unwrap and apply new parameters
Update and store Lightmap
Send new image path
Relay new image path
Load new image as texture

Figure 5: *Parameter Exchange*

5.3 Disconnection handling

If the users connection is unstable and disconnects his browser from the server, the socket.io pipe is broken and without particular attention, the simulation behaviour becomes unpredictable. For this reason, a special handshake implementation has been necessary to be able to reset the scene at reconnection exactly like it was before disconnection. The implementation is schematized in fig. 6

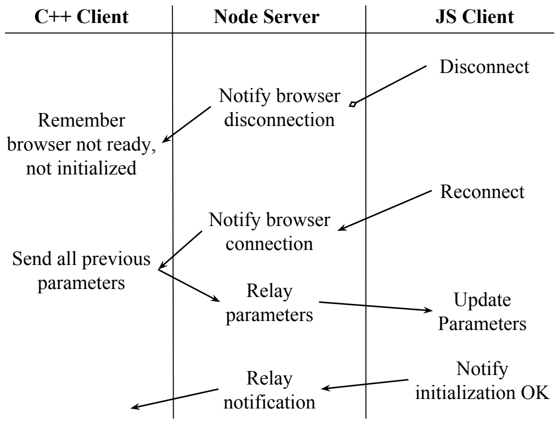

C++ Client
Node Server
JS Client
Disconnect
Notify browser disconnection
Remember browser not ready, not initialized
Reconnect
Notify browser connection
Send all previous parameters
Relay parameters
Update Parameters
Notify initialization OK
Relay notification

Figure 6: *Disconnection Handling*

6 Conclusion, Future Work

This project has been extremely slowed down by the lack of proper documentation on how to run OpenGL on Amazon G2 instance with Linux AMIs. Tutorials are available, some even written by NVidia, but none of them worked as they were written. At the time of the writing of this report, the Nvidia Capture SDK doesnt work at all on Linux instances. The use of the socket.io C++ client as built and static libraries was also problematic, even when running everything as described in the official documentation. All of this combined implied that the instance setup presented in this report is the result of months of trial and errors, but have been successfully tested again on an empty Ubuntu image, and can be used as reference when aiming at running OpenGL programs and/or socket.io C++ client on similar instances.
Distributing part of the rendering pipeline is an idea that we believe will become more exploited in the future, but for now technologies are too young and not integrated well enough to allow easy implementation. Hopefully, in the future, Amazon and NVidia will propose instances that let the developper upload an OpenGL program, and that will natively be able to run it and live stream the framebuffer to a client. Implementing the asynchrone pipeline described in the previous report ([Khoury 2016]) would then have been a matter of minutes, but it is not the case yet.
For these reasons, this part of the project has been more about networks and server than rendering and computer graphics, but the necessary has been done to have a working case of the usage scenario we aimed at.
This project could be improved by implementing a better HDR algorithm on the Ray Tracing Engine, and by making the NVidia Capture SDK work with our project.

7 Implementation Details

This section will detail all the installations and commands necessary to run the libraries used in this project. First, we detail how to install the packages necessary to run Opengl. Next, we list the commands necessary to run headless OpenGL applications on an AWS G2 instance from an Ubuntu image. We then do the same for the socket.io C++ client and the Node / Socket.io server. All these commands and scripts are available at [Khoury 2017], easing the implementation of a similar instance for the interested reader.

7.1 Running OpenGL on a raw Ubuntu AMI

This sequence of commands has to be executed in this exact order for headless OpenGL rendering to work. Also note that because of the pdf formatting from Latex, it is possible that some tabs and/or spaces have been affected and that some command might not work by copy-pasting this PDF (in particular the sed command on the xorg.conf file). Please refer to [Khoury 2017] if interested in running similar libraries.

Driver installation
At the time of this writing, G2 instance have a some trouble running NVidia Drivers past the 367 version. As Ubuntu added the NVidia driver to the default APT ppa, it can simply be installed by running:

```
sudo apt-get update
sudo apt-get upgrade
sudo apt-get install nvidia-367
sudo reboot now
```

After rebooting the instance and connecting to it again through SSH, we can check if the driver has been well installed and is running with:

```
nvidia-smi
```

Disable automatic updates
The APT automatic updates must be turned off to avoid the NVidia driver to be automatically updated (as newer versions seem to be incompatible with the G2 instances). This can be done by editing the configuration file located at

```
/etc/apt/apt.conf.d/10periodic.
```

Simply change the 1 by a 0 in the line:

```
APT::Periodic::Update-Package-Lists "1";
```

Install OpenGL libraries
Different libraries are available to run OpenGL on UNIX systems, the ones used for this project are:

```
libglfw3-dev, libglew-dev, mesa-utils,
    ↪ cmake, make
```

They are all available on the default ppa of APT and can be installed like any other standard packages.

Virtual Display Script
Once the NVidia Drivers are working and the OpenGL-related packages are installed, this instance has to be setup to be able to run rendering program without any display connected. Unlike the previous instructions and commands that are unaffected by reboots, the virtual display setup has to be run every time the instance starts, and therefore can be placed in a script if the instance will be reused.

The first thing to do is to stop the ligthdm service to be able to disable the X server and modify it, by running

```
sudo service lightdm stop
```

As we want to modify some parameters of the X server, the next command is to kill this process. It sometimes takes a little while for the process to stop completely, and since it's a non blocking operation it often causes problems in the rest of the script. As a "wait" command doesn't solve this issue, the safest way to do this is to loop the kill command until the process is stopped, for example:

```
sudo pkill X
res="$(echo␣$?)"
while [ $res -eq 0 ]
do
   sudo pkill X
   res="$(echo␣$?)"
done
```

Then, we need to tell the NVidia Driver that we want to use a virtual display with a given resolution. It can be done running the command

```
sudo nvidia-xconfig -a --use-display-
    ↪ device=None virtual=1920x1080
```

Now that the driver is aware of the display virtualisation, we need to modify the X server configuration to be compatible with the hardware setup and the virtual display:

```
sudo sed -i 's/    BoardName      "GRID␣
    ↪ K520"/    BoardName      "GRID␣K520"
    ↪ \n BusID            "0:3:0"/g' /etc/
    ↪ X11/xorg.conf
```

To be able to run OpenGL programs on this configuration, the DISPLAY environment variable has to be set to 0, or can be stored using:

```
export DISPLAY=:0
```

The last thing that needs to be done is to relaunch the X server:

```
sudo /usr/bin/X &
```

The glxgears program can be used to test the configuration, and should run at approximately 20000 frames per second.

Launch script at boot time
If the instance is supposed to be ready to run OpenGL at each boot, all of these commands can be put in a script, then this script needs to be made executable and finally copied in the correct directory:

```
chmod 755 display_script.sh
cp display_script.sh /etc/profile.d/
```

7.2 Running the socket.io C++ client

The first thing to do to be able to run the C++ client is to install the full boost.io library, available directly on apt-get:

```
sudo apt-get install libboost1.58-all-dev
```

Once thats done, the socket.io C++ client can be downloaded at: `https://github.com/socketio/socket.io-client-cpp.git` Building and using the socket.io as release libraries fails and doesn't seem to be really ready for use. A good work-around is to copy the complete socket.io directory in the directory where other external library for the project are stored (in our case: Caustics_RayTracer/external) and keep the CMake files provided as they are.
In the Cmake files of the project:

```
#Add in the linker flags
-pthread -lssl -lcrypto -glfw3
#Add as include:
include(external/sio/CMakeLists.txt)
#Add as targer link library:
sioclient
```

Once everything is ready for use, the only thing needed in the C++ project is to include in the `sio_client.h` header.

7.3 Running a socket.io server

The socket.io javascript client and server rely on Node.js, so it must be installed first. Node itself relies on some packages available on apt-get:

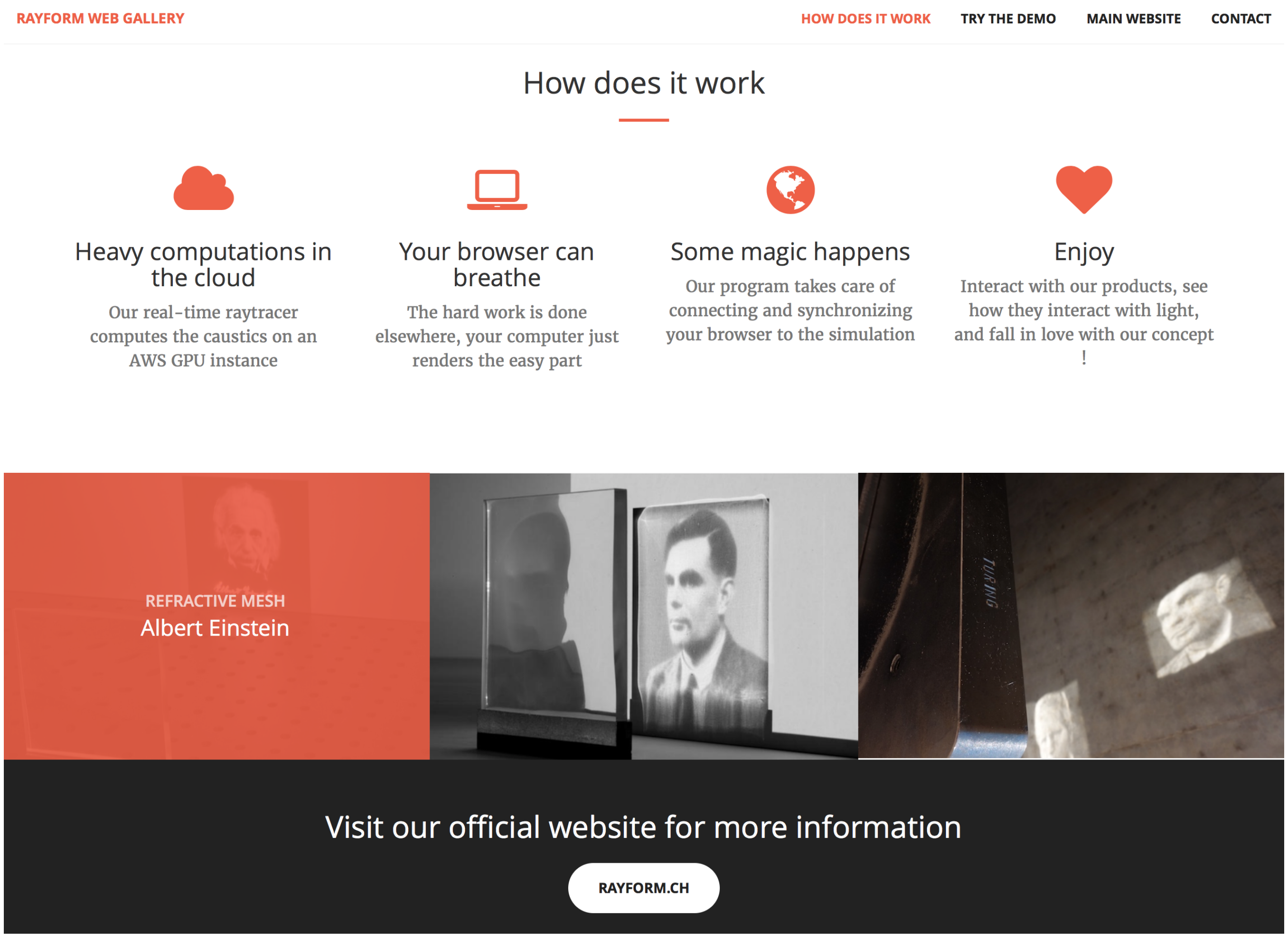
```
sudo apt-get install build-essential
sudo apt-get install checkinstall
sudo apt-get install libssl-dev
```

Once their installation is complete, the Node Packet Manager required to install Node can easily be installed with curl:

```
curl -o- https://raw.githubusercontent.com
    ↪ /creationix/nvm/v0.31.0/install.sh |
    ↪  bash
```

Finally, after exiting and launching the terminal again, run:

```
nvm install 7.7.4
```


RAYFORM WEB GALLERY

HOW DOES IT WORK TRY THE DEMO MAIN WEBSITE CONTACT

How does it work

Heavy computations in the cloud

Our real-time raytracer computes the caustics on an AWS GPU instance

Your browser can breathe

The hard work is done elsewhere, your computer just renders the easy part

Some magic happens

Our program takes care of connecting and synchronizing your browser to the simulation

Enjoy

Interact with our products, see how they interact with light, and fall in love with our concept !

REFRACTIVE MESH

Albert Einstein

Visit our official website for more information

RAYFORM.CH

Figure 7: *WebGallery Homepage*

References

AWS, 2006. Amazon web services: reliable, scalable, and inexpensive cloud computing services. `https://aws.amazon.com/`.

KHOURY, J.-N., 2016. An interactive web gallery for goal-based caustics (part 1). `https://jadkhoury.github.io/files/WebGallery_Report_V2.1.pdf`.

KHOURY, J.-N., 2017. A set of scripts and command to run opengl and socket.io on an aws ubuntu instance. `https://github.com/jadkhoury/AWSscripts`.

SOCKET.IO, 2013. Socket.io: the cross-browser websocket for realtime apps. `https://socket.io/`.

THREE.JS, 2015. Three.js, a javascript 3d library. `http://threejs.org/`.