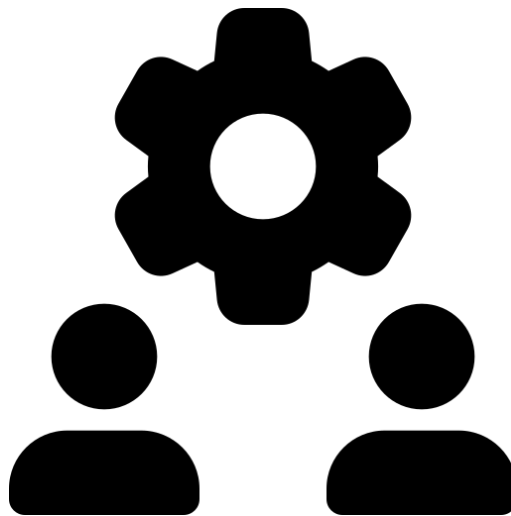


Adapting 3D multiuser capabilities
using open source projects and
standardized libraries.



119112856

Lucas Gonzalez de Alba

CS6105 Future and emerging technologies

Index

▪ Abstract.....	3
▪ Introduction and project overview	4
▪ 3D service provider	5
▪ Interaction management.....	5
▪ Client-server architecture.....	5
▪ Peer-to-peer architecture.....	6
▪ Client-server vs Peer-to-peer architecture.....	6
▪ Communications provider	7
▪ Network basics.....	7
▪ Communications delay.....	8
▪ Real-time systems (RTS).....	9
▪ The project structure.....	10
▪ Client and server class diagrams.....	11
▪ System's workflow.....	13
▪ Running the server on a Linux Debian distribution	13
▪ Simulations.....	16
▪ Conclusion.....	18
▪ References.....	18

Abstract

This paper introduces the results of my research on adapting a 3D rendering application to support multiuser experience over a browser using standard open source libraries.

Today there are multiple companies (Mozilla, Qbit, HTC...) with 3D applications that support multiuser experiences but the main problem is that the products they develop are based on proprietary software. Branded software is exclusive and thus less accessible but also less robust than standardized libraries. Features can be deprecated removed or updated and cause many compatibility issues. Open source projects tend to grow slowly but steadily and usually become much more stable because the community of developers participating in the project offer wide range of support. Likewise if it is standardized it is much easier to expand, develop for and rely on.

For such reasons I decided to investigate whether it is possible to develop a three-dimensional multiuser experience based only on free open source projects.

Introduction and project overview

In order to start my research first, I had to define what '3D Multiuser experience' is and what features presents. A three dimensional multiplayer application is a system that immerses two or more users in a 3D environment. Such system has to establish a shared space for users to interact remotely. Since users experience the environment in a unique way, the representation of the shared space is exclusive to each user. This of course creates three challenges, first establishing each individual perspective, second managing the user interactions with scene and users and third building a system that supports distributed communications.

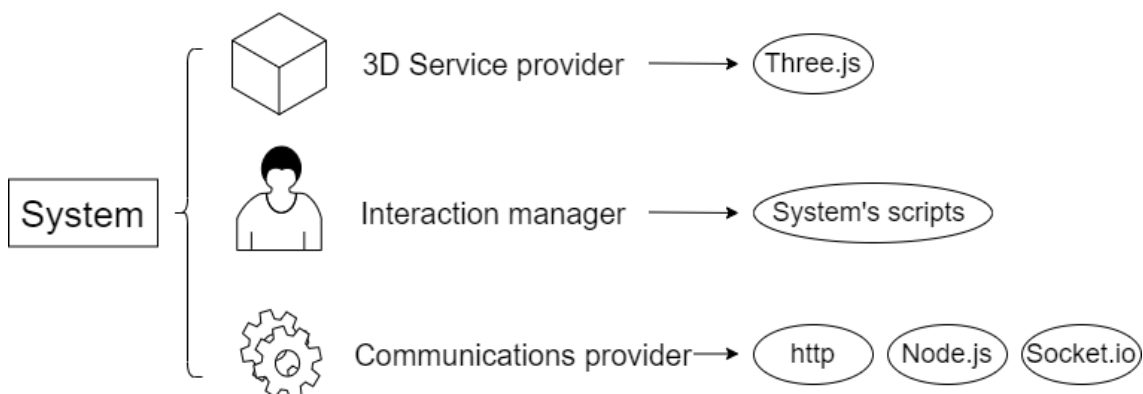
In other words, the system has three roles:

1. 3D service provider: In charge of rendering the environment for each user, that's scene, objects and users.
2. Interaction manager: In charge of tracking each user's actions and translating them into environment updates.
3. Communication provider: In charge of the distributed infrastructure and network management

While doing my research I realized that although there are multiple tools developed for each role, there are is no unified system that provides all the features under one library. This intrigued me because I expected multiuser services to be integrated into the standard tools, but it is not the case. I was surprised because the market demands multiuser services all over from 3D multiplayer games to online social virtual spaces or virtual reality conferences, but nonetheless there is no unifying open source tool.

When a development team builds a multiuser experience (using open source libraries), it must select the different providers for each of the system's roles.


For this project, it was convenient to use the following services:



3D Service provider

Three.js is a JavaScript library that serves as an application-programming interface for three-dimensional rendering. The library is an abstraction of the low-level implementation details that provides WebGL. Thus, it is often said that Three.js sits on top of WebGL.

THREE allows the developer to add objects, animations and several other elements inside a scene. The scene is the 3D canvas on which the elements stand. The elementary components of a scene are:

	<u>Scene:</u>	<code>scene = new THREE.Scene();</code>
	• Camera:	<code>camera = new THREE.PerspectiveCamera(...)</code>
	• Renderer:	<code>renderer = new THREE.WebGLRenderer()</code>
	• Light:	<code>light = new THREE.PointLight(...)</code>
	• Object/Mesh:	composed of material and geometry

These features provide a unique user experience. The application sets up the environment and object variables, initializes shaders and starts the rendering loop that will last as long as the scene is operable. The user perceives the scene through the camera's perspective. This in term means that if we want to build a multiuser experience, we will have to differentiate each of the user's viewpoints. In addition, the multi-view approach has to be distributed across the internet.

Interaction management

We have to define an architecture to model the interactions between users and system. The two most common architectures are client server or peer to peer, each with a number of advantages and disadvantages.

Client-server architecture.

The client server architectures is a hierarchical model in which the resources are hosted (stored) in one or multiple locations and are managed by one or multiple entities (the servers). The server architecture is very useful because it simplifies the problem of distributed content across the network and creates a single input-output stream of data. In this case, the clients connect to the server and requests the resources to start the app's execution. The server then is in charge of sending the requested data and managing the client's execution flow.

The main disadvantage of the client-server design is scalability and load distribution. If the number of clients linked the server is big enough so that the rate of incoming requests is above the server's processing threshold (due to bandwidth "bits per second" limitations, poor incoming/outgoing messages ratio) then the system is overwhelmed and ceases its activity.

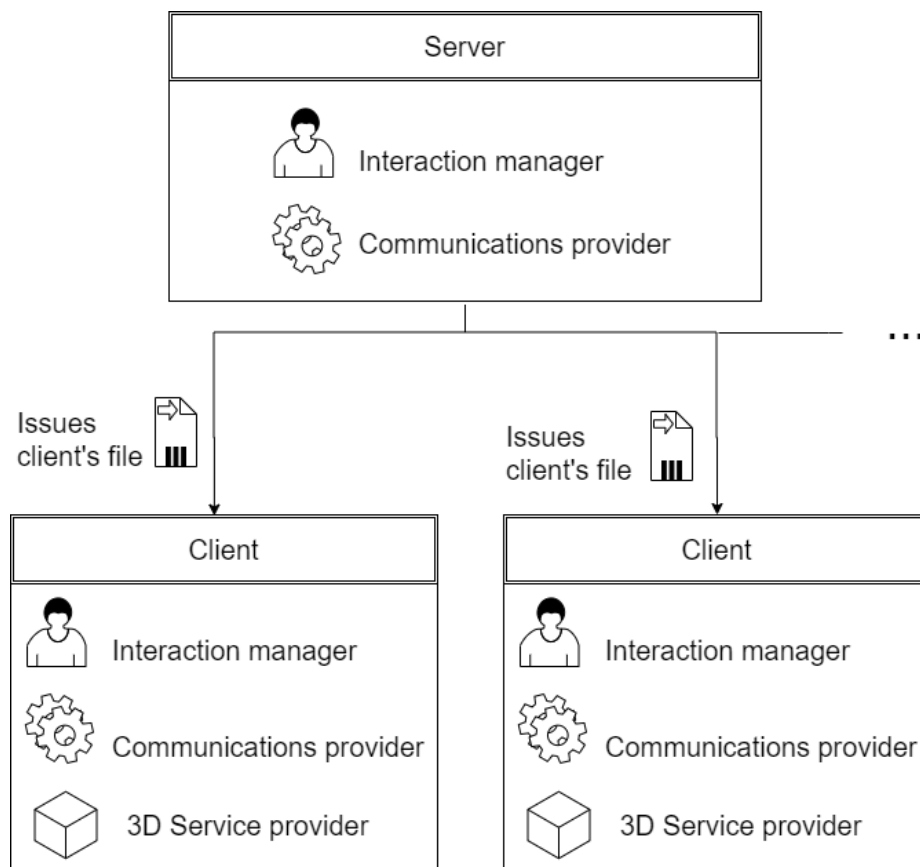
Peer-to-peer architecture (P2P).

The P2P architecture is a hierarchical model for which each user is both client and server meaning it both hosts and serves content. The advantages of such design are robustness, improvement in greater levels of reliability since it eliminates single points of failure.

The main concern with decentralized networks has to do with consistency and security. Since there is no authority figure ensuring that all peers share the same state of the global process at any point in time, it becomes a challenge to organize the network. Furthermore, communications between all users complicate updates propagation but in contrast, it disperses the load the server would have had to handle across peers. It is possible to synchronize actions but it may come at the expense of execution time dynamism and most importantly real time response.

Client-server vs Peer-to-peer architecture

Client-server	Peer-to-peer
Centralized	Decentralized
Great modularity	Highly scalable
Single point of failure	Robust and reliable
Lower levels of complexity	Higher levels of complexity
Better security management	Intricate security management



Because this project serves as a client server demo, a centralized approach has been favored over peer's network. This in term means that the server will organize all connected clients.

Communications provider

Distributing the processes across the network is one of the main requisites and challenges of the project. Since the application runs over a browser, the communication protocol used is HTTP. The server uses a socket-programming interface called Socket.io in conjunction with node.js. We will study this on detail but first a brief overview of some networking aspects.

Networking basics

All end-to-end communications are established using physical links with several properties such as propagation speed, transmission rate, degree of mobility (static or portable), etc... Different technologies allow for very different throughputs and bandwidths but regardless of the physical layer, all communications have to get routed from source to destination (Also known as link layer).

At the network core, there exist two architectures:

1. Circuit switched: connection-oriented networks as used for some telecommunications
2. Packet switched: connectionless-oriented networks as used for Internet Protocol (IP)

Since the scope of this project is to study the viability of a multiuser 3D application distributed over the internet we will only focus on packet switched networks because the IP protocol is a non-dedicated data transfer. Over this layer we have the transport protocol, based on TCP's reliable mechanisms, and HTTP internet communication protocol.

The server provides:

- Standard libraries
 - Three.js, the 3D renderer & Socket.io, the network communications manager
- Application scripts and assets
 - JavaScript individualized files



fpscontrols.js



index.js



scene.js

- CSS stylesheet, the web design library.

To send and receive messages between server and client it is necessary to establish a communications route. To do so we use socket programming, which consists of a set of networking techniques and tools to provide end-to-end interconnectivity. A socket is a software abstraction that encapsulates an IP address and a port number, plus additional data structures and functions. In this project, we create a socket on both server and client side. The client opens a TCP connection with the server's socket, which listens for upcoming TCP messages on port 3000. It is bidirectional because when a client's scene changes (i.e. user moves), the client informs the server and the server propagates the changes. Therefore, the client's socket also listens for upcoming messages over the connection.

Socket.IO is a JavaScript library for real-time web applications. It enables real-time, bi-directional communication between web clients and servers. It has two parts: a client-side library that runs in the browser and a server-side library for Node.js. Both components have a nearly identical API. Like Node.js, it is event-driven.

Socket.io

Communication delays

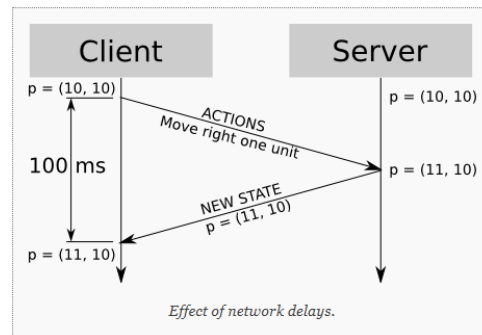
Packet switched networks are highly modular and scalable but at the expense of absolute reliability. The communication can suffer bit errors, packet loss and packet drops (Buffer overflow). The internet protocol is a statistical model in which the performance metrics depend on the rate at which packets enter the net, the number of simultaneous transmissions, and the level of router congestion ...etc.

Time delays change depending on the end-to-end distance (source to destination path including intermediate routers and proxy servers) or retransmissions due to high levels of congestion within the network but in general, the cumulative delay is equal to the sum of the delays at each node (router)



$$\text{TimeDelay} = \sum d_{\text{nodal}} ; d_{\text{nodal}} = d_{\text{proc}} + d_{\text{queue}} + d_{\text{trans}} + d_{\text{prop}}$$

In our multiplayer online experience, there is a client server architecture. When a client's browser sends an HTTP GET request to the server, the server sends the client's the files and assets required to display the scene and manage the interconnectivity. Once the files are fully retrieved and downloaded, the client initiates the communications (as seen in the right figure).



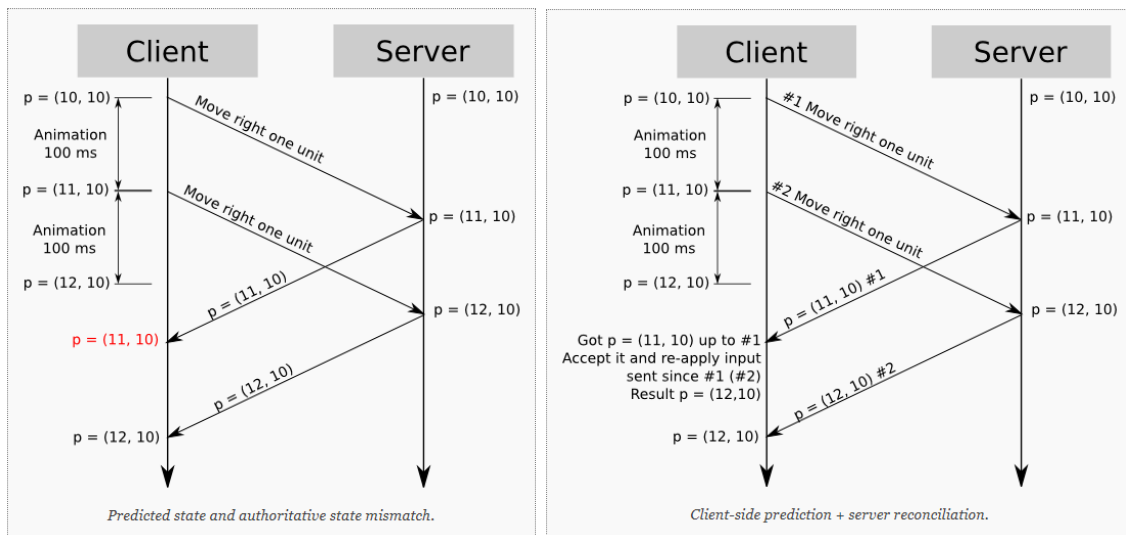
There is a fundamental issue we have to address, and that is how the communications provider in conjunction with the interaction manager affect the 3D service provider. In our project, we have JavaScript files linked to an HTML webpage. When an action is performed the client informs the server and when the server receives updates, it propagates them to all clients. The fundamental question we have to answer is, how are updates applied?

If every time the server receives an update the system had to reset and resend all files again through the overhead would increase significantly and the client would experience a dramatic augment of the network time delay. Our application is a real-time distributed system, so we cannot afford to reload each 3D scene every time there is a user update because it would be too costly in terms of time and performance efficiency. Instead of a reset strategy, the clients adapt their running scenes to match the server's updates. With this design it is important to stress that the files are loaded only once. The client side broadcasts the user interactions (i.e. change of positions) to the server, and the server brings up-to-date information to all clients. To maintain all 3D world instances updated any interaction from any user has to get propagated across the network. This, of course, creates a new challenge and that is how to synchronize scenes without running into inconsistencies while minimizing time delays.

Real Time Systems (RTS)

Some multiplayer experiences are resilient against time delays (turn based strategy games for example) but other are very sensitive to time delays. When a system fails to keep itself updated with others, lag appears. Our application is time sensitive so we will try to reduce overhead, time costly operations (such as reloading and resending files) and redundant transmissions.

One solution for systems that depend on high-latencies is client-side prediction. A mathematical model governs most interactive systems and therefore it is possible to predict most of the system states in a pure deterministic fashion. If the user or the application changes its state it will propagate the modifications but instead of waiting for the server acknowledgement it will assume the change of state was valid and proceed to display it. There is a slight trick with this and it has to do with delays. If the update and acknowledgment are not synchronized, the user will experience inconsistencies. This can be addressed with a stack of last performed actions. This way you can keep track of what actions have been displayed and which have been confirmed by the server (authority).



The left figure illustrates how the client state suffers from status inconsistency due to network delays. The right figure solves this issue with server reconciliation.

In the case of this project, since the application works as a demo, the server doesn't validate any of the clients states. The clients simply provide the user's status (position coordinates, rotation, angle ...) to the host and the host propagates the updates to the rest of the players connected. In some sense, this approach presents many vulnerability issues such as game cheating due to excessive responsibility on the client side or security violations because the server lacks client's validation. I find important to point out that such weaknesses are design contingencies. It possible to eliminate most of these with a layered architecture (scene engine, security, and network) or a decentralized responsibility management. However, it is not always possible to solve all weaknesses without decrementing performance. The designer must maintain a balance and trade off strengths to cope with some weaknesses.

The project structure

As mentioned before the server application combines Three.js (3D Graphics), with Node.js (Server networkings). Since it is built above Node.js so it has to follow the defined conventions in Node's workflow. One of these are scripting which operations Node.js has to follow and establishing what are the dependencies on a file named Package.json.

Package.json

```
{
  "name": "MultiplayerThreeScene",
  "version": "0.0.1",
  "description": "Introduce multiuser experience on three.js",
  "main": "src/index.js",
  "scripts": {
    "start": "concurrently \"nodemon server.js\" \"watchify src/index.js -o public/js/bundle.js -v\"",
    "build": "browserify src/index.js | uglifyjs -mc warnings=false > public/js/bundle.min.js",
    "test": "npm run build"
  },
  "author": "Lucas Gonzalez based on Or Fleisher <contact@orfleisher.com>",
  "license": "MIT",
  "dependencies": {
    "ejs": "^2.5.7",
    "event-emitter-es6": "^1.1.5",
    "express": "^4.16.2",
    "gsap": "^1.20.3",
    "http": "0.0.0",
    "socket.io": "^2.0.4",
    "three": "^0.89.0"
  },
  "devDependencies": {
    "babel-core": "^6.26.0",
    "babel-preset-env": "^1.6.1",
    "babelify": "^8.0.0",
    "browserify": "^15.2.0",
    "concurrently": "^3.5.1",
    "glslify": "^6.1.0",
    "nodemon": "^1.14.12",
    "uglify-js": "^3.3.9",
    "watchify": "^3.10.0"
  },
  "browserify": {
    "transform": [
      [
        "babelify",
        {
          "presets": [
            "env"
          ]
        }
      ],
      "glslify"
    ]
  }
}
```

In this json package file we define:



- Application details
 - **Name**
 - **Version**
 - **Description**
 - **Main javascript file**
- Scripts "workflow"
 - **Start** : Concurrently execute:
 - `nodemon` Which restarts the server on every change (port: 3000)
 - `watchify` Which bundles the client code from `src/` on every change to `./public/js/bundle.js`

`nodemon server_file watchify src_file -o dest_file -v`
 - **Build** : Execute
 - `browserify` Which is setup to transform both ES6 Javascript and glslify for GLSL shader bundling.
 - `uglifyjs` Which parses input files in sequence and apply any compression options.

`browserify src_file 'flags' | uglifyjs 'flags' > dest_file`
 - **Test** : Execute
 - `npm run build` Which runs the build script as described in json.

Once the Json package file is defined we can execute the Node Package Manager (npm)

npm install is a npm cli-command which does the predefined thing i.e, as written by Churro, to install dependencies specified inside package.json

npm run command-name or **npm run-script command-name** (*ex. npm run build*) is also a cli-command predefined to run the custom scripts with the name specified in place of "command-name". So, in this case **npm run build** is a custom script command with the name "build" and will do anything specified inside it.

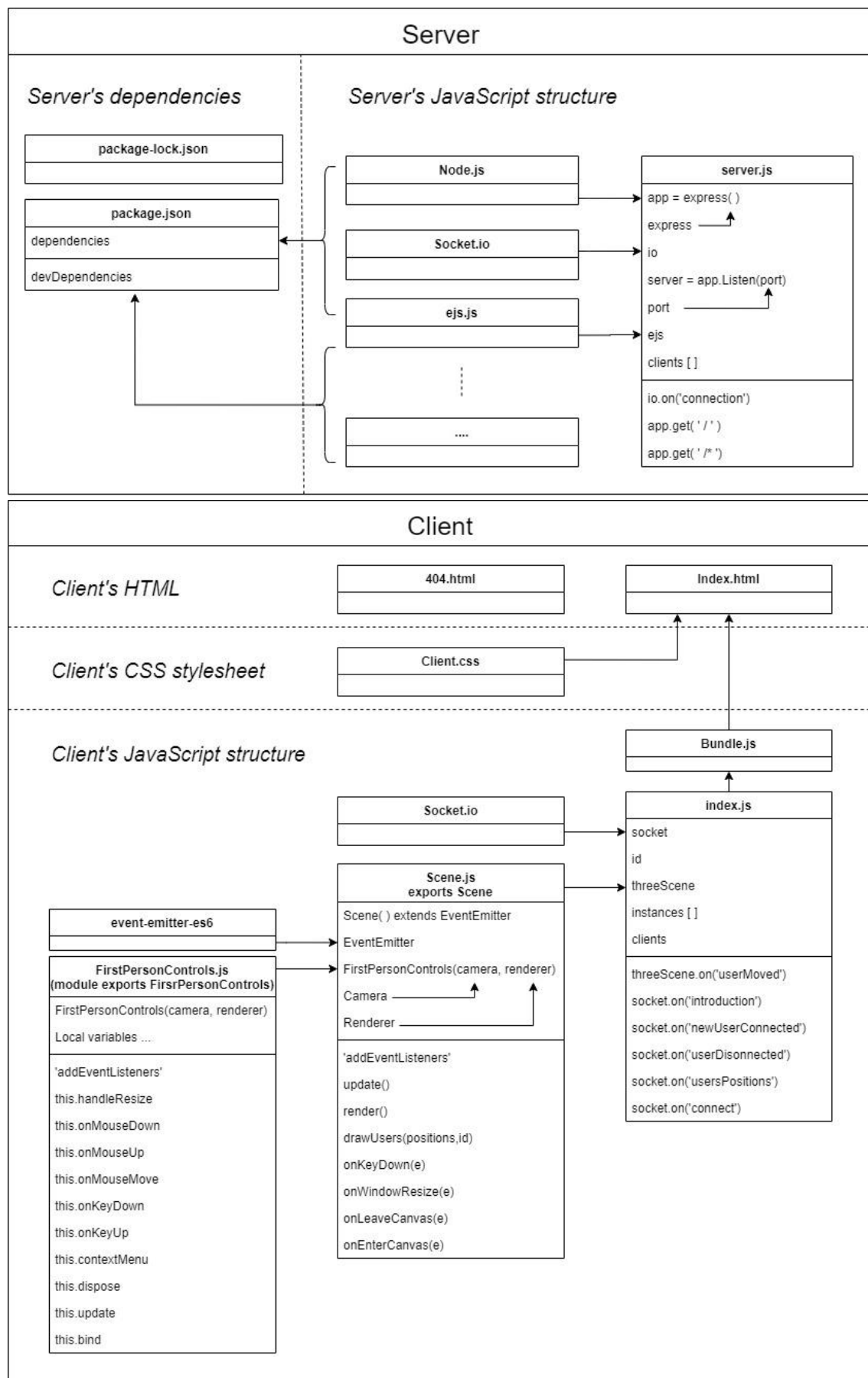
nodejs.org

What happens when we run our script?

First Browserify parses the main JavaScript file (index.js) and creates a single file (bundle.js) that contains all the requirements. To collect all the requirements into a unique .js file the browserify has to access recursively to solve all dependencies. Since Bundle.js concatenates all the subscripts it is the only script tag present in our HTML.

Then using watchify we can monitor the main (index.js) for changes (the changes may also be in the files included from index.js) and automatically generate the resulting bundle.js so that the server hosts the updated version. As a consequences updates are refresh instantly clients always request the new versions.

Client and server class diagrams



System's workflow

Running the server on a Linux Debian distribution

For this installation it is required to use curl. This command is a tool to download or transfer files/data from or to a server using FTP, HTTP, HTTPS, SCP, SFTP, SMB and other supported protocols on Linux or Unix-like system.

Since I did not have curl command installed, I also had set it up.

```
# Install curl tool
$ sudo apt update
$ sudo apt upgrade
$ sudo apt-get install -y curl
```

Install node.js library.

```
# Using Debian, as root
$ curl -sL https://deb.nodesource.com/setup_13.x | bash -
apt-get install -y nodejs
```

The next step is to obtain the project's source code. You can do it by manually accessing and downloading the code from the source [ref1] or by cloning the repository using git's clone tool.

If you lack git's command tools you can install them by running the following command

```
# Install git's tools
$ sudo apt-get install git-all
```

Clone repository

```
# Clone the source repository
$ git clone https://github.com/juniorxsound/THREE-Multiplayer.git
```

Now the application is ready to execute. We launch the server, that is executing the node.js application.

`npm run start` "Executes the 'start' commands scripted inside JSON"

```
> MultiplayerThreeScene@0.0.1 start
/home/groongra/ucc/cs6105_future_and_emerging_technologies/final_assignment
> concurrently "nodemon server.js" "watchify src/index.js -o
public/js/bundle.js -v"[0] [mon] 1.14.12
[0] [nodemon] to restart at any time, enter `rs`
[0] [nodemon] watching: *.*
[0] [nodemon] starting `node server.js`
[0] Server is running localhost on port: 3000
[0] [nodemon] restarting due to changes...
[0] [nodemon] starting `node server.js`
[1] 1096535 bytes written to public/js/bundle.js (4.20 seconds) at 3:35:07 PM
[0] Server is running localhost on port: 3000
```

Here's an example of the application's workflow with three client connections (A,B and C).

When **client A** connects, the server accepts and provides the client A with ID (C1GjydsakwLOITOAAAA)

Server

```
[0] User C1GjydsakwLOITOAAAA connected, there are 1 clients connected
```

Client A

```
My ID is: C1GjydsakwLOITOAAAA bundle.js:46588
1 clients connected bundle.js:46592
```

When **client B** connects, the server accepts and provides the client with ID (VnB17t2AUoNFtaMvAAAB) and a list of all connected clients {...}

Server

```
[0] User VnB17t2AUoNFtaMvAAAB connected, there are 2 clients connected
```

Client B

```
> {C1GjydsakwLOITOAAAA: {...}} bundle.js:46585
My ID is: VnB17t2AUoNFtaMvAAAB bundle.js:46588
2 clients connected bundle.js:46592
```

Client A receives updates

```
2 clients connected bundle.js:46592
A new user connected with the id: VnB17t2AUoNFtaMvAAAB bundle.js:46601
```

When **client C** connects, the server accepts and provides the client with ID (reTRYJ9IjT7ps3RIAAAC) and a list of all connected clients {...}

Server

```
[0] User VnB17t2AUoNFtaMvAAAB connected, there are 3 clients connected
```

Client C

```
> {C1GjydsakwLOITOAAAA:{...}, VnB17t2AUoNFtaMvAAAB:{...}} bundle.js:46585
My ID is: reTRYJ9IjT7ps3RIAAAC bundle.js:46588
3 clients connected bundle.js:46592
```

Client A receives updates

```
3 clients connected bundle.js:46592
A new user connected with the id: reTRYJ9IjT7ps3RIAAAC bundle.js:46601
```

Client B receives updates

```
3 clients connected bundle.js:46592
A new user connected with the id: reTRYJ9IjT7ps3RIAAAC bundle.js:46601
```

When any user interacts with the scene (i.e. Press down a key to move mesh), the event Listener inside the client script detects the action, updates the scene and informs the server. The server then broadcasts the updates to all user. When the news reach the clients each of the clients apply the received changes, except if they were the source. How is this checked? The message contains a field with the ID of the user that originated the updates and therefore the clients can validate if the information received belonged to them.

For example, suppose a server with two clients, *b6bFBQ5qexMH3Xaaam* and *Mm11b1s_NCpZu30VAAAN*. Every movement is identified by:

```
'some-user-id': { position: [0.4, 1.4, -0.5], rotation: [0, 0, 0] }
```

If *b6bFBQ5qexMH3Xaaam* moves then:

Client *b6bFBQ5qexMH3Xaaam* receives updates but doesn't apply changes to scene because `Object.keys(_clientProps)[0] == id` -> true

Positions of all users are

```
{6_b6bFBQ5qexMH3xAAAM: {...}, Mm11b1s_NCpZu30VAAAN: {...}}

6_b6bFBQ5qexMH3xAAAM:
  position: (3) [0, 0, 0]
  rotation: (3) [0, 0, 0]
  __proto__: Object{...}
Mm11b1s_NCpZu30VAAAN:
  position: (3) [0.1519916506600505, 0, -0.002114987034348504]
  rotation: (3) [0, 0, 0]
  __proto__: Object{...}
__proto__: Object{...}
```

bundle.js:46624

bundle.js:46625

true

Client *Mm11b1s_NCpZu30VAAAN* receives updates and apply changes to scene because `Object.keys(_clientProps)[0] == id` -> false

Positions of all users are

```
{6_b6bFBQ5qexMH3xAAAM: {...}, Mm11b1s_NCpZu30VAAAN: {...}}

6_b6bFBQ5qexMH3xAAAM:
  position: (3) [0, 0, 0]
  rotation: (3) [0, 0, 0]
  __proto__: Object
Mm11b1s_NCpZu30VAAAN:
  position: (3) [0.1519916506600505, 0, -0.002114987034348504]
  rotation: (3) [0, 0, 0]
  __proto__: Object
```

bundle.js:46624

bundle.js:46625

false

When any user disconnects, the server eliminates the client searching for its ID in the list of all connected clients {...} and deletes the instance.

```
[0] User C1Gjydsakw10IT0AAAA disconnected, there are 2 clients connected
[0] User reTRYJ9IjT7ps3RIAAAC disconnected, there are 1 clients connected
[0] User VnB17t2AUoNFtaMvAAAB disconnected, there are 0 clients connected
```

Simulations

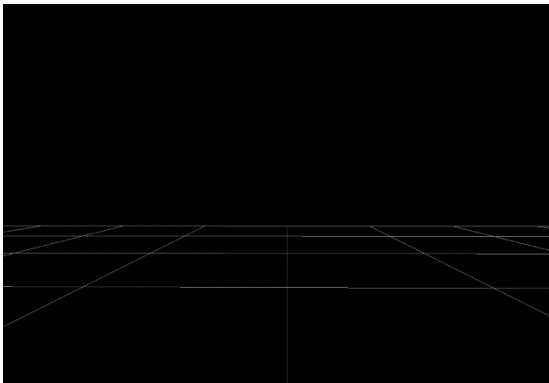
To better illustrate the system's performance I've included a number of figures to compare the different user's perspective plus the server's perception.

Server is launched.

```
> MultiplayerThreeScene@0.0.1 start /home/groongra/ucc/cs6105_future_and_emerging_technologies/final_assignment
> concurrently "nodemon server.js" "watchify src/index.js -o public/js/bundle.js -v"

[0] [nodemon] 1.14.12
[0] [nodemon] to restart at any time, enter `rs`
[0] [nodemon] watching: *.*
[0] [nodemon] starting `node server.js`
[0] Server is running localhost on port: 3000
[0] [nodemon] restarting due to changes...
[0] [nodemon] starting `node server.js`
[1] 1096535 bytes written to public/js/bundle.js (4.94 seconds) at 12:40:05 PM
[0] Server is running localhost on port: 3000
```

Client A connects <http://localhost:3000/>

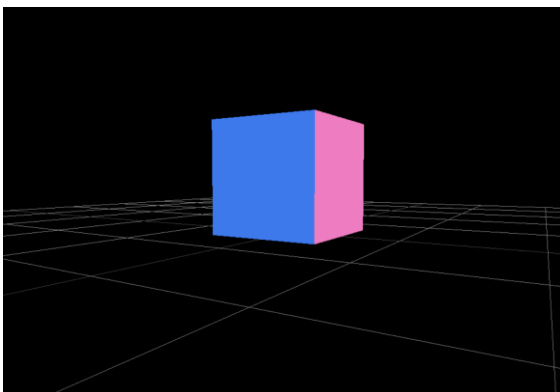


Client A perspective

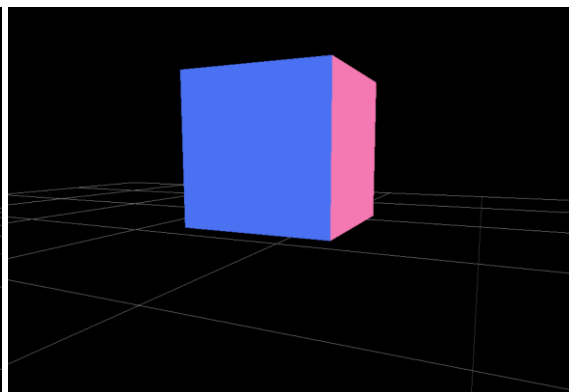
A is the first user so scene is empty

```
[0] User HETfctSPq1WC8gEwAAAA connected, there are 1 clients connected
```

Client B connects <http://localhost:3000/>



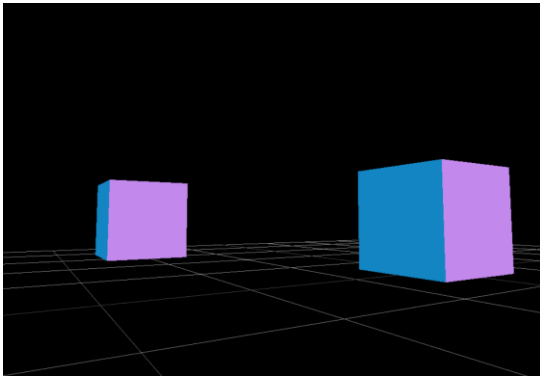
Client A perspective



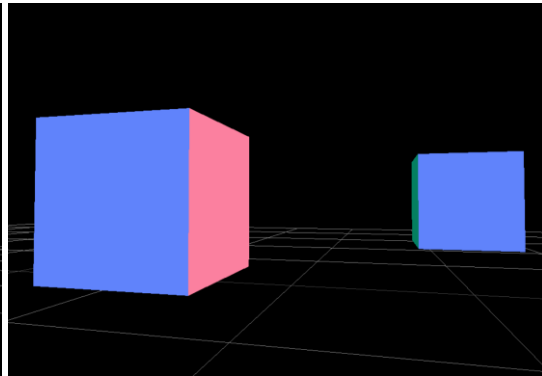
Client B perspective

```
[0] User Ejn7IYNPAZrOPVIZAAAB connected, there are 2 clients connected
```

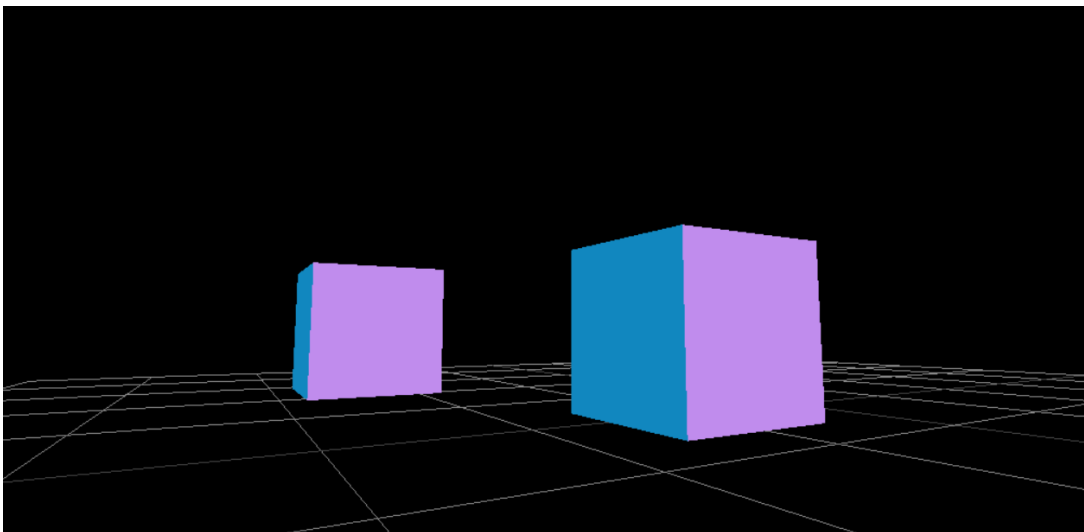

Client C connects <http://localhost:3000/>



Client A perspective



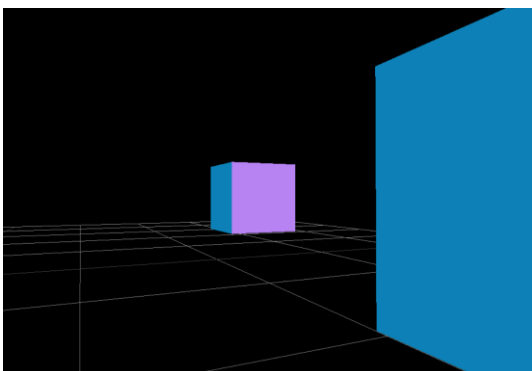
Client B perspective



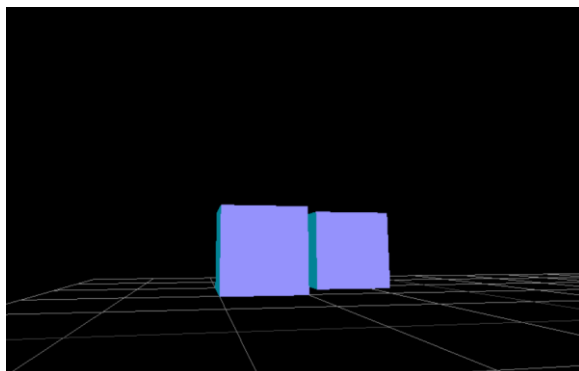
Client C perspective

```
[0] User cCQcKHu4wjKYj3HEAAAC connected, there are 3 clients connected
```

Client A moves



Client A perspective



Client C perspective

Conclusion

Virtual reality (VR) is a fast growing field with constant leaps in development and standardization (WebGL and Three.js for example). Today it is clear that 3D applications form part of internet's ecosystem and more and more websites include 3D tools for visualization and exploration. Thanks to open source projects, it is possible to expand content and develop new applications based on the standardized libraries. In addition, it is quite clear that we as users tend to expect greater and greater levels of connectivity and interaction and thus the demand for multiplayer VR is increasing. This research made me realize we lack a non-proprietary 3D multiuser platform. Although some companies do offer 3D software for multiuser applications, I believe it is only a matter of time until new standards introduce these services into already existing libraries. The tools we need already exist (The three system roles explained in this paper) but they aren't unified under one same system. Since the demand is out there, I consider this state of affairs as an opportunity to adapt and enhance many of open source virtual reality tools we have today.

Finally, I would like to give special thanks to Or Fleisher for providing a study case for my research (The link to the repository can be found in the references section) and David Murphy for his help and orientation during the project.

References

Assets:

Icon made by <https://www.flaticon.com/authors/pixel-perfect>

Resources:

[ref1] Multiplayer boilerplate <https://github.com/juniorxsound/THREE.Multiplayer#installation>

GitHub <https://github.com/>

Node.js <https://nodejs.org/en/>

Node.js Source code (Binary distributions)

<https://github.com/nodesource/distributions/blob/master/README.md>

Client-Server time diagrams: Gabriel Gambetta <https://www.gabrielgambetta.com/>