Implementing Multiplayer Virtual Reality For any Application in Three.js COMP 3490

Wynand Badenhorst

December 16, 2017

Due on:	December 18, 2017
Developed by:	Wynand Badenhorst (And everyone who made the libraries that this project depends on)
Professor:	Neil Bruce

1 Objective

Before starting this project, I was curious about the limits of mobile VR. I sought out to implement a very basic multiplayer VR environment as a proof of concept. This was achieved, and now you can watch an arcade machine with your friends, as you all move the claw up down and around in Virtual Reality.

2 Challenges in Implementation

a Implementing the stereoscopic view was very difficult at first, but after thoroughly studying the documentation of Three.js, I found a that you could add effect to the rendering of your scene, and one of those is ‘Three.StereoEffect’. This works much more efficiently than making your own two cameras, and is much simpler to implement. The command is simply:

```
effect = new THREE.StereoEffect( renderer );
effect.setSize( window.innerWidth, window.innerHeight );
```

b Porting the game to mobile proved to be challenging as well, but socket.io and express.js proved very helpful in getting this done. I learned about this from an incomplete video tutorial (https://www.youtube.com/watch?v=LBj4XlySZLU).
Upon further research and trial and error, I succesfully managed to get the server and clients communicating.

c Sharing the world between clients, and keeping them synchronized, proved to be much easier than anticipated. My first idea was to stream video to the clients and make the application server heavy, but then I realized if only geometry were maintained on the server side, then the rendering sould be done by the clients.

How I achieved the shared updatable world was by creating the Geometries of the game in the server, but never adding them to a scene or rendering them. Whenever a client affects the world around it, it informs the server, the server updates it’s data structures, if it chooses to, and then the server broadcasts the results to all the clients.

The clients then do their own rendering and their own (strictly local) geometry updates.

d In my first iteration, I was using more bandwidth than available, so messages to the server were being cached until I ran out of memory. What I realized was that the updates to the server were being sent once every time interval instead of once every time interval when there was an actual update. This was corrected by refactoring my code to only send updates when the user has interacted with the scene and manipulated it.

e I did not implement any physics or world-world interaction, because this would be continuous, or would need to be done almost entirely client-side with ‘corrections’ from the server on some interval.

3 Techniques used to implement Multiplayer VR

a Implementing VR:

Implementing the virtual reality component of the game proved simpler than expected. I started work on building my own camera object, but Three.js already has stereo vision implemented.

With a regular camera with orbit controls, we saw that it is implemented as follows:

```
cameraControls = new THREE.OrbitControls(camera, renderer.domElement);
```

While the render function does this:

```
renderer.render(scene, camera);
```

Stereo vision is implemented like this:

```
effect = new THREE.StereoEffect( renderer );
effect.setSize( window.innerWidth, window.innerHeight );
cameraControls = new THREE.DeviceOrientationControls( camera );
```

While the render function does this:

```
effect.render(scene, camera);
```

b Implementing Multiplayer:

Let's assume the client already has code, somehow.

The client's freedom should be restricted, for safety. For instance, a command which asks the server to lower the hanging arm might look like:

```
$.ajax({
        type: 'POST',
        url: '/update',
        data: JSON.stringify({
                name: 'hangingArm',
                direction: 'decr'
        }),
        contentType: 'application/json',
        cache: false
});
```

Where the only unique variable are 'hangingArm' and 'decr', both decipherable by the server. The 'url' can also be unique, as you'll see in a bit.

The server's handlers for different requests look something like this:

```
app.post('/update',
  function(req, res){
    if(updatable.has(req.body.name) && updatable.get(req.body.name).has(req.
      console.log(req.body.name + " " + req.body.direction);
      updatable.get(req.body.name).get(req.body.direction)();
    }
    res.send('');
  }
)
```

In this case 'updatable' is a mapping of component names to mapping of action names to the respective actions. These actions update the model server-side and broadcast a request for all the clients to update, as follows:

```
function(){
  if(crane.position.y < 700){
    crane.position.y += 1;
    io.sockets.emit('crane', {data: crane.position.y});
  }
}
```

This specific instance updates the position of the crane, unless it's a a maximum already, and then broadcasts the new location to all the clients connected to the server. Notice it does not tell the clients to increase the height of the crane, it simply informs them that this is the new height. This ensures that lost packets have a minimal effect of client model synchronization.

The client handles the broadcast message like so:

```
socket.on('hangingArm', function(data){
  hangingArm.position.x = data.data;
});
```

And finally the arm's position is updated.

Note: This communication is implemented in socket.io and express, but may be similiarly connected using other libraries.

Also worth noting is that the implementation of Virtual Reality is not directly tied to the implementation of the multiplayer game. Although the dependancy on online communication has implications for the user experience in VR, it does not affect how VR is implimented.

4 Concerns with VR user experience

As we learned in class, virtual reality can make people nautious. Delayed feedback on your environment, cognitive dissonance on the nature of your environment as well as weird physics you're not used to can make people uncomfortable. A large concern I had going into this project was latency, because it's core to the nature of how we get feedback from our environment.

On an online server the latency is very noticeable, while on a local server (on the same Local Area Network) the latency is rarely noticable, and infrequently affects the experience if at all.

An issue I had with the implementation of this specific application is that there are few common tools people can use for input when their phone is inside a VR cardboard style headset. The multiplayer aspect of the resulting application almost fixes this by allowing you to simply have the app open on desktop as well as on your phone, but this doesn't work as well when

5 Results and Conclusions

It worked. One of the main things I was looking for in this project was if it would even be possible for VR to be part of a multiplayer mobile game, and now we know it is (at least for very simple games). The frame rate stayed high, the latency was not ideal online but acceptable on a LAN. With a controller for your phone, or control based on direction, you could make a fully fledged mobile game.

I've implemented this with a very simple game, which we saw in assignment 1, but I will conclude this can be done with most games in Three.js, and most of my peers who I've shown my code to tend to agree, though few grasp the networking side.

6 What's next

I know now that it is possible to make a simple multiplayer VR application, and as a result to share a virtual space with someone without special hardware being necessary. What I would like to see is more VR applications become democratized, and for them to possibly open up a world of new experience.

Part of this process is peripherals. Many gaming controllers already have excellent tactile feedback (for when you can't look at your controller) but as any first person shooter gamer with a pc can tell you, controllers give a distinct disadvantage to a player in terms of reaction time and precision. Most fps gamers prefer a mouse and keyboard for this reason. An alternative of the mouse and keyboard combo with the same effectiveness.

In my application I didn't implement any physics, but handling physics is somewhat solved in many games, such as World of Warcraft, Overwatch, Call of Duty, or many other examples of massively multiplayer games with physics implemented. A future verison of this project might look at what level we can implement real looking (and feeling) physics.

Finally, on a personal note, I would like to implement a simple game akin to what you might have seen 'in the old days' on a gamecube, XBOX or a Playstation 2. The reason why I would like to implement a game from that era is because the multiplayer aspect of those games was almost entirely focussed on couch-coop, or couch-pvp. At the moment this mean you could have a local game over LAN (to reduce latency) and have an experience similiar to the old days, but in VR. Specifically, the multiplayer mode from 'Goldeneye: Rogue Agent' would be an excellent candidate for this. The environments are fairly limited, the physics is straightforward, and the concept of the game has proven to be very fun. I will however note that screenpeeking would not be possible.