

# Development of **3D Visualizations**

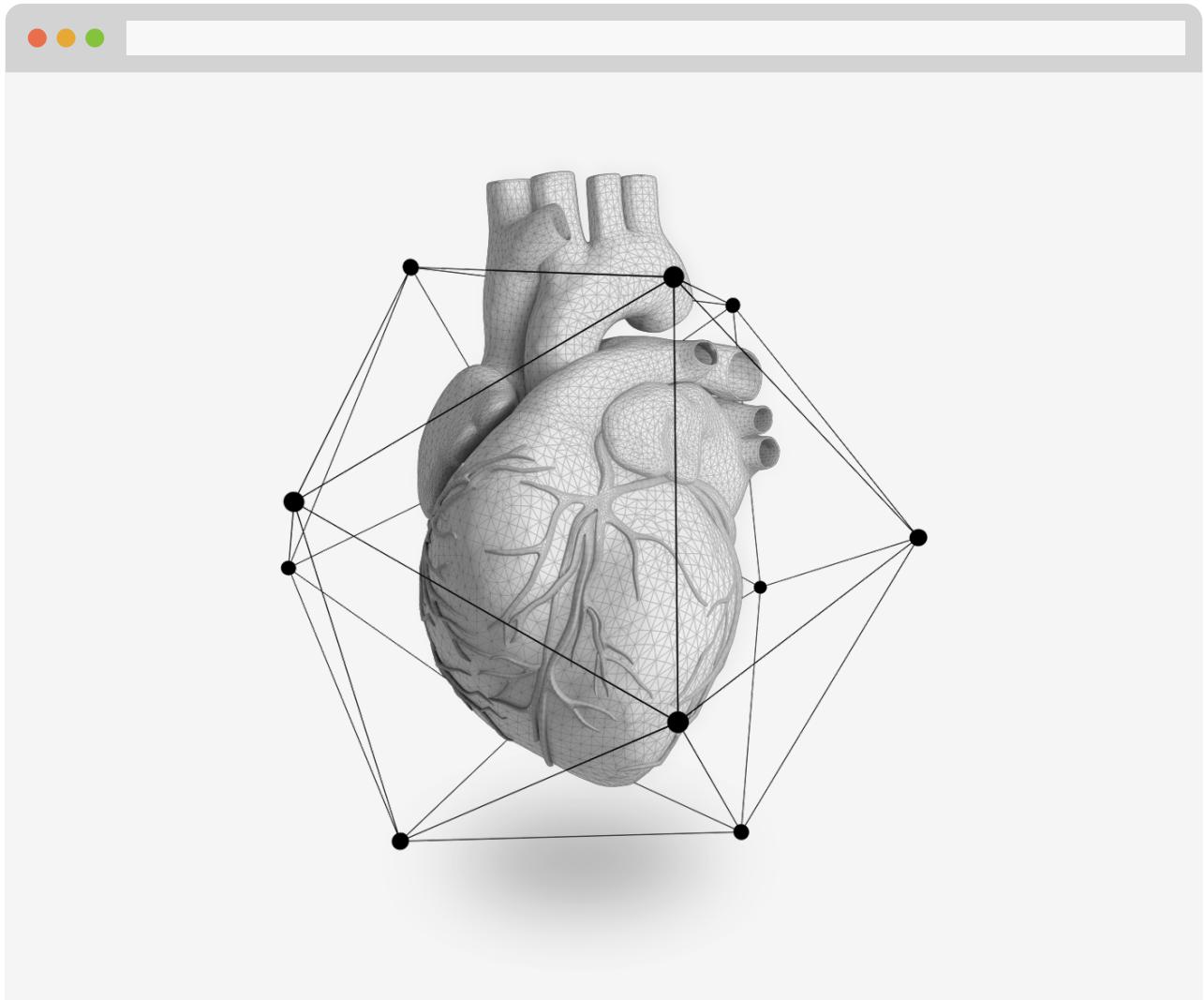
Node.js, React.js and  
Three.js for web  
visualization



1<sup>st</sup> Edition  
Lucas Cassiano  
2017

# Desenvolvimento de **Visualizações 3D**

Node.js, React.js and Three.js para visualização de  
projetos na web



1<sup>a</sup> Edição  
Lucas Cassiano  
2017

# Desenvolvimento de Visualizações 3D

Node.js, React.js e Three.js para visualização de  
projetos

Lucas Cassiano - [cassiano@mit.edu](mailto:cassiano@mit.edu) - Setembro, 2017

Este material foi criado como parte de um curso de extensão da Universidade Federal do Rio Grande do Norte, ministrado parte presencial e parte em forma de Educação a Distância, no ano de 2017, por Lucas Cassiano. Parte do conteúdo foi traduzido de diversas fontes, outra parte criada pelo autor. As devidas referências, links e citações estão organizados como notas de rodapé ao decorrer do material.

Material de suporte e códigos de exemplos podem ser encontrados em:

[https://github.com/lucascassiano/Development\\_of\\_3D\\_Visualizations\\_examples](https://github.com/lucascassiano/Development_of_3D_Visualizations_examples)



# Conteúdo

<b>1 Node.js</b>	2
1.1 Sobre o Node	
1.2 NPM - Node Package Manager	
1.3 Instalação	
<b>2 WebApp e WebApp Progressivo</b>	5
2.1 WebApp vs App Nativo	
2.2 Definição (da Google) de Webapp Progressivo	
<b>3 React.js</b>	7
3.1 Sobre o React.js	
3.2 Programação Declarativa	
3.3 Componentes Encapsulados	
3.4 Parâmetros, Estado e Ciclo de Vida de um Componente	
3.5 Design de Interfaces com Componentes React	
<b>4 Criando um App</b>	15
4.1 Create-react-app	
4.2 Organização de Diretórios	
4.3. Electron: WebApp Desktop	
<b>5 Three.js</b>	18
5.1 OpenGL e WebGL	
5.2 Renderizadores, Cenas e Câmeras	
5.3 Geometrias, Materiais e Malhas	
5.4 Iluminação	
5.5 Importando Objetos	
5.6 Interações com Objetos	
<b>6 Encapsulamento do Three.js como Componente React</b>	28
6.1 Componente Container3	
6.2 Importando Objetos - Componente ObjectViewer	
<b>7 React-Three-Renderer</b>	
<b>8 Material de Referência</b>	30

# 1.Node.js

## 1.1. Sobre o Node.js

Node.js é um JavaScript Runtime, ou seja, uma plataforma JavaScript para programação de propósito geral, que funciona administrando eventos assíncronos e fora de navegadores web padrão. Node.js foi projetado para aplicações escaláveis. Dessa forma o Node.js auxilia na construção de aplicações, trazendo ainda consistência por permitir a construção de back-end e front-end utilizando a mesma plataforma e linguagem de programação. Abaixo temos um exemplo básico, “hello world” utilizando node.js - abra o navegador web de sua preferencia na página - <http://localhost:3000>.

```
const http = require('http');
const hostname = '127.0.0.1';
const port = 3000;

const server = http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.listen(port, hostname, () => {
  console.log(`Server running at http://${hostname}:${port}/`);
});
```



Node.js vem crescendo em uso a cada dia, de acordo com o

W3Techs<sup>1</sup>, mais de 30% dos websites estão rodando com um web-server node.

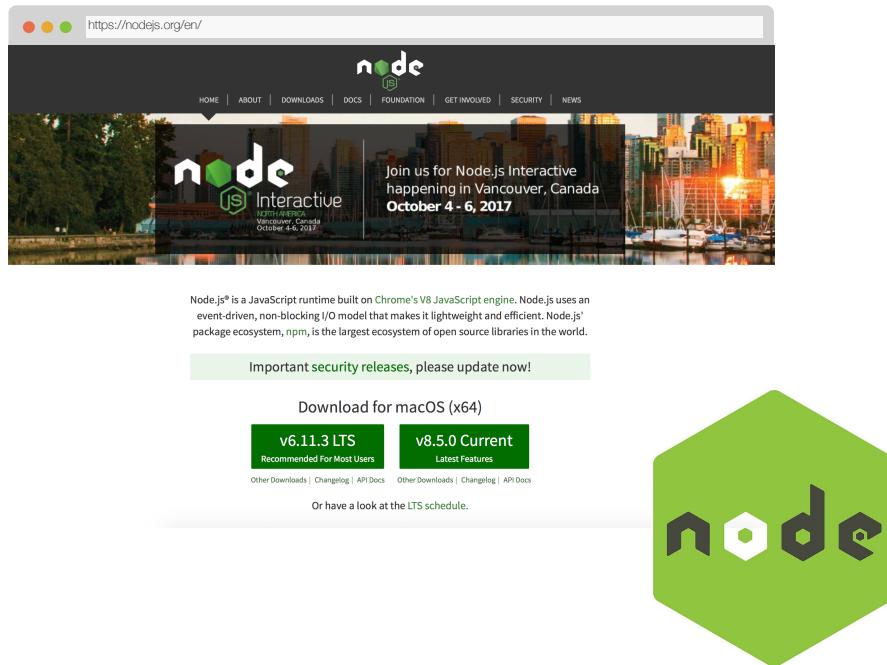
Alguns aplicativos e sites que você possivelmente usa diariamente rodam em instâncias do Node.js, para citar alguns como exemplo temos: Uber, Netflix e Ebay.

## 1.2. NPM - Node Package Manager

Como o próprio nome já diz, Node Package Manager é um administrador de Pacotes<sup>2</sup> JavaScript, e o maior registro de código do mundo<sup>3</sup>. NPM torna simples o compartilhamento e o re-uso de códigos em JS criado por outros desenvolvedores. NPM ainda atualiza automaticamente pacotes já instalados no seu projeto.

## 1.3. Instalação

*Node.js* tem um instalador para Windows, MacOS e Linux, para baixa-lo  
acesse <http://nodejs.org>



<http://nodejs.org>

<sup>2</sup> Módulos de códigos JavaScript que podem ser usados em diversas aplicações

<sup>3</sup> <https://www.npmjs.com>

Se o Sistema Operacional utilizado for Linux, É possível ainda instalar o Node.js utilizando o apt-get, como pode ser visto em mais detalhes e com passo-a-passo neste link<sup>4</sup>.

Mas, de forma simples, seguindo o conjunto de comandos abaixo é possível instalar tanto o node.js quanto o npm (1.2), usando um OS Linux.

```
$ sudo apt-get update  
$ sudo apt-get install nodejs  
$ sudo apt-get install npm
```

---

<sup>4</sup> <https://www.digitalocean.com/community/tutorials/como-instalar-o-node-js-no-ubuntu-16-04-pt>

## 2. WebApp e WebApp Progressivo

### 2.1. WebApp vs App Nativo<sup>5</sup>

No ramo de desenvolvimento de aplicativos mobile, uma discussão recorrente é a comparação entre aplicativos web e aplicativos nativos. Cada um tem suas vantagens e desvantagens. Aqui vamos focar em comparar ambos e no decorrer desta apostila iremos elaborar mais sobre o conceito de aplicativos web.

**Aplicativo Nativo** é um aplicativo desenvolvido essencialmente para um dispositivo em particular e instalado diretamente no dispositivo. Usuários podem instalar tal aplicativo pelas lojas, por exemplo a *Apple App Store*, ou *Google Play store* no caso de dispositivos Android. Um exemplo de aplicativo nativo é o aplicativo da câmera para dispositivos Apple (iOS).

Um **Aplicativo Web** é basicamente um aplicativo baseado na internet, que para ser acessado precisa ser executado por um navegador web, por exemplo o Google Chrome.

Em termos de **Eficiência**, aplicativos nativos são mais caros<sup>6</sup>, porém funcionam de forma mais rápida e otimizada para o sistema operacional em questão, indo além, os usuários acabam por confiar mais em aplicativos nativos, pois estes estão disponíveis apenas em lojas online das desenvolvedores do OS. De todo modo um aplicativo web pode ser otimizado a ponto de rodar de forma rápida e otimizada, mas não é a regra geral.

**User Interface** é um ponto importante quando se pensa no design de um aplicativo. Do ponto de vista do usuário de um dispositivo, alguns aplicativos nativos e web têm funcionalidades e visual similares, com poucas diferenças entre eles.

---

<sup>5</sup> <https://www.lifewire.com/native-apps-vs-web-apps-2373133>

<sup>6</sup> <https://www.lifewire.com/mobile-app-development-the-cost-factor-2373475>

**O Processo de desenvolvimento** é o que diverge bastante entre os dois tipos de aplicativos. Cada plataforma móvel<sup>7</sup> usa uma linguagem de programação diferente. Enquanto iOS Usa Objective-C e/ou Swift, Android Usa Java, Windows Mobile usa C++ e ai por diante. Para um aplicativo web as ferramentas, no geral, são JavaScript, HTML5, CSS3 e outros frameworks web, como React.

## 2.2. Definição (da Google) de Webapp Progressivo<sup>8</sup>

**Progressive Web Apps (PWA)** são experiências que combinam o melhor da web e o melhor dos aplicativos<sup>9</sup>. Por serem aplicativos Web não exigem instalação, porém conforme o usuário acessa funções do aplicativo, este vai se tornando mais eficaz, pois vai progressivamente realizando o download e salvando partes importantes do código/aplicativo no dispositivo do usuário, de forma automática.

Pelo fato de ter partes pré-salvas no navegador, um PWA é carregado de forma rápida, mesmo em redes instáveis e pode até mesmo funcionar offline. Alguns browsers como o Chrome ainda exibem uma caixa de diálogo perguntando ao usuário se ele deseja criar um atalho para o webapp na tela inicial do dispositivo.

Progressive WebApps são mais **rápidos** para ser carregado quando comparado a webapps simples, são ainda mais **confiáveis** por funcionar até em redes instáveis. Permitem ainda uma melhor imersão e **experiência do usuário** por permitir modos *fullscreen*, notificações e etc. São ainda **responsivos**, ou seja, podem funcionar em qualquer tamanho de tela e ainda oferecer uma experiência consistente de navegação.

---

<sup>7</sup> <https://www.lifewire.com/what-is-a-mobile-operating-system-2373340>

<sup>8</sup> <https://developers.google.com/web/progressive-web-apps/?hl=pt-br>

<sup>9</sup> <https://developers.google.com/web/fundamentals/getting-started/codelabs/your-first-pwapp/?hl=pt-br>

## 3. React.js<sup>10</sup>

### 3.1. Sobre o React.js<sup>11</sup>

React é uma biblioteca JavaScript para construção de Interfaces gráficas (UI). React atualiza e renderiza de forma eficiente apenas os componentes gráficos que tiveram os dados alterados. Esta biblioteca utiliza *views* Declarativas, o que faz com que o código se torne mais fácil de ler e seja mais previsível e mais facilmente ‘debugável’. No próximo tópico iremos dar uma pausa rápida, ao invés de focar em React explicaremos um pouco mais sobre programação declarativa.

React permite ainda a construção de componentes encapsulados que administram seus próprios estados, e utilizando estes construir UIs complexas. De forma mais simples, um componente que serviria de barra de notificações “saberia” aonde, quando e como atualizar as informações necessárias e ainda saber como deve se comportar a ações do usuário. Imagine que os componentes gráficos de uma UI passam a saber como devem reagir (sim, é daí que vem o nome ‘react’) às alterações de dados.

### 3.2. Programação Declarativa

Programação declarativa é um paradigma da programação baseada em programação funcional, lógica ou restritiva<sup>12</sup>. A ideia de programação declarativa foi iniciada nos anos de 1950<sup>13</sup>, com a criação das primeiras linguagens de programação de alto nível.

---

<sup>10</sup> <http://reactjs.com>

<sup>11</sup> <https://facebook.github.io/react/>

<sup>12</sup> [https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o\\_declarativa](https://pt.wikipedia.org/wiki/Programa%C3%A7%C3%A3o_declarativa)

<sup>13</sup> <https://www.netguru.co/blog/imperative-vs-declarative>

Basicamente um app declarativo é um programa que descreve o que ele faz e não como seus procedimentos, ou tarefas, são ordenados e/ou funcionam. Programação declarativa foca em como a informação é tratada e não como as tarefas são executadas - você deve estar imaginando agora que é algo muito complexo, mas não, são apenas funções que chamam outras funções de retorno, ‘callbacks’ que funcionam de forma não síncrona).

Como exemplo iremos usar o básico de uma interface gráfica: um botão. Ao clicar no botão, é adicionado mais um nome de usuário em um array e exibir o numero total de usuários. Algo bem simples. Vejamos como seria com **programação Imperativa**<sup>14</sup> e para efeito de comparação, utilizaremos JavaScript :

Imperativo	Descritivo
<pre>var users = [];  function addUser(newUser){     users.push(newUser);     return users.length; }  \$("#button").click(     function(){         var count = addUser("João");         \$(             this         ).toggleClass("highlight");         \$("#number").text(             count+" usuários"         );     } )  &lt;Button id="btn"&gt;     &lt;p id="number"&gt;0 usuários&lt;/p&gt; &lt;/Button&gt;</pre>	<pre>var users = [];  function addUser(newUser){     users.push(newUser);     return users.length; }  &lt;Button     onClick={addUser("João")}     &gt;     &lt;p&gt;{users.length} usuários&lt;/p&gt; &lt;/Button&gt;</pre>

---

<sup>14</sup> <https://tylermcginnis.com/imperative-vs-declarative-programming/>

Como pode ser visto no exemplo acima, o código a direta é mais simples de ser compreendido. Não atente apenas para o número de linhas que os códigos têm, atente para o tempo que você levou para compreender o que cada um dos exemplos executa.

### 3.3. Componentes Encapsulados

Para tornar o código ainda mais simples de ser lido, podemos encapsular algumas funções e ações de um componente. Estes componentes poderão ser ordenados de forma hierárquica, com componentes tendo sub-componentes internos. Para tornar a escrita desses componentes mais simples iremos usar JSX<sup>15</sup>. JSX é uma linguagem de programação baseada em JavaScript que oferece um sistema de classes muito mais próximo de Java, por exemplo.

Abaixo um exemplo de um componente escrito em JSX e outro apenas em JavaScript:

#### Utilizando JSX

```
class HelloMessage extends React.Component {
  render() {
    return
      <div>
        Hello {this.props.name}
      </div>;
  }
}

ReactDOM.render(
  <HelloMessage name="Jane" />,
  mountNode);
```

#### Utilizando JavaScript

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
}

ReactDOM.render(React.createElement(HelloMessage, { name: "Jane" }), mountNode);
```

---

<sup>15</sup> <https://jsx.github.io>

Mas **não se preocupe**: JSX pode e é utilizada em conjunto com JavaScript, de forma a tornar os scripts mais simples e melhor comprehensíveis.

No exemplo acima, encapsulamos o Componente HelloMessage<sup>16</sup> como uma tag <HelloMessage/>, então na próxima vez que usarmos esse componente e quisermos mudar o nome, apenas passaremos o nome como um atributo JSX - ou parâmetro (“props” - próxima unidade).

### 3.4. Parâmetros, estado e Ciclo de vida de um Componente

Cada componente recebe parâmetros, que são passados dentro de um atributo JSX como um único objeto, chamado de **props**<sup>17</sup>.

**props** podem ser acessados utilizando:

```
this.props
```

Por exemplo, se você tem um componente que recebe uma cor (no exemplo utilizando código de cor Hexadecimal<sup>18</sup>) como parâmetro:

```
class Example extends Component {  
  
  render() {  
    return (  
      <div>  
        {this.props.color}  
      </div>  
    );  
  }  
}
```

---

<sup>16</sup> Sempre comece nomes de componentes customizados com letras maiúsculas, para evitar confundir com tags do DOM, como <div>

<sup>17</sup> <https://facebook.github.io/react/docs/components-and-props.html>

<sup>18</sup> <http://www.color-hex.com>

```
//Como seria usado:  
<Example color="#00b0b2"/>
```

Estados (**states**) contêm informações específicas para o componente que podem ser alteradas. O estado é definido pelo usuário e deve ser um objeto JavaScript. As alterações desses estados são o que definem se um componente deve ser renderizando novamente ou não. Por tanto nunca altere um estado diretamente (this.state), utilize a api:

```
setState()
```

Os “métodos de ciclo de vida” são diversos para cada componente, podendo ser sobreescritos. O **Ciclo de vida** de um componente é baseado em 5 métodos:

#### **constructor()**

É executado antes do componente ser montado, é o ponto ideal para definir estados iniciais. Se houver necessidade de ler algum parâmetro nesse momento, use:

```
super(props)
```

#### **componentWillMount()**

Chamado imediatamente antes do componente ser montado, é chamado antes de *render*. De forma que as alterações em Estados nesse momento não irão acionar um *re-rendering*.

#### **render()<sup>19</sup>**

Este é um método obrigatório, todo componente react deve ter uma implementação deste. Este deve examinar os parâmetros e estado (props e state) e deve retornar um único elemento React. Podendo ser um elemento nativo do DOM, como <div> ou um componente composto.

#### **componentDidMount()**

---

<sup>19</sup> <https://facebook.github.io/react/docs/react-component.html#render>

É invocado imediatamente após o componente ser montado. Utilize para inicializações que requerem acesso a elementos do DOM.

Outros métodos como “componentWillReceiveProps” ou “shouldComponentUpdate” são úteis para atualizar um componente, ou verificar parâmetros antes ou depois dessa atualização. veja mais sobre<sup>20</sup>.

### 3.5. Design de Interfaces com Componentes React

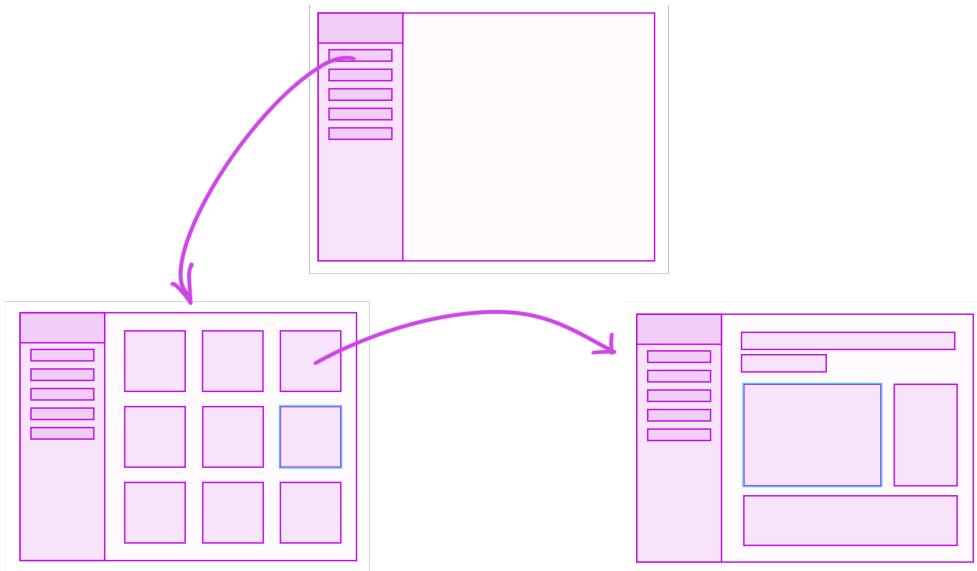
Após entendermos como criar componentes React e encapsular métodos nesses componentes, iremos agora explorar um pouco como projetar uma interface gráfica utilizando os conceitos de componentes.

Uma das principais diferenças, quando se fala de design gráfico, entre um aplicativo web a uma página web é que para tornar a experiência do usuário mais fluída o aplicativo ao invés de carregar várias páginas para cada *view*, carrega as páginas como um componente dentro da página, e só vai trocando este componente a partir das ações do usuário.

De forma mais prática, vamos imaginar um página web de compra de livros. Teremos então 3 *views*: página inicial (home), lista de livros e detalhes de livros. A navegação ficaria como na imagem abaixo

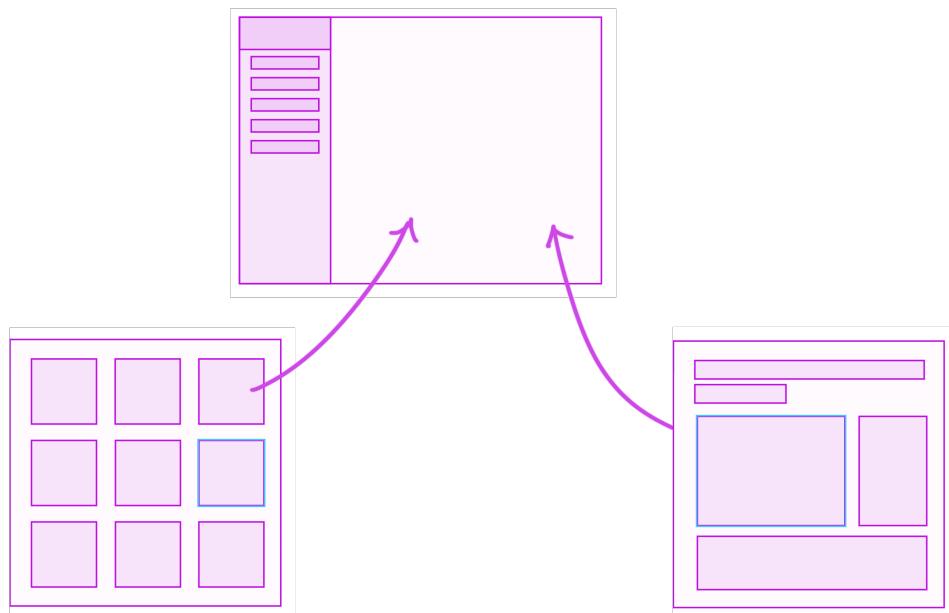
---

<sup>20</sup> <https://facebook.github.io/react/docs/react-component.html#componentwillreceiveprops>



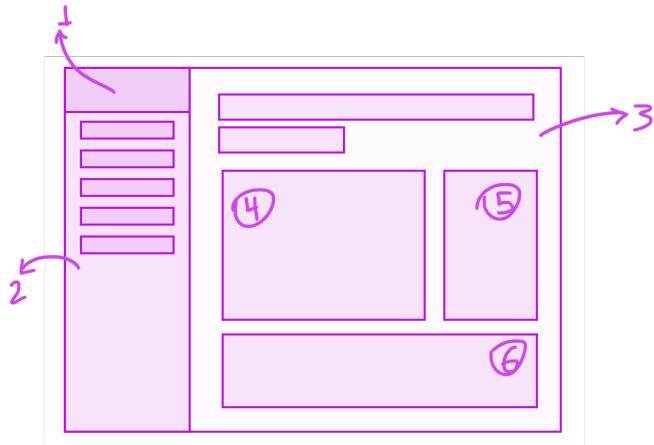
Exemplo de navegação em um website

No caso de uma página similar criada com React, teríamos 3 Componentes principais - que seriam compostos por diversos outros componentes, vamos chamar estes grandes componentes de **Containers** - e dentro do componente *Index*, teríamos instâncias dos componentes *Home*, *ListaDeLivros* e *DetalhesDoLivro*, de forma que iríamos apenas trocar o que seria visualizado pelo usuário baseado nas escolhas do menu lateral.



Exemplo de navegação de um webapp com componentes React

Como visto no exemplo acima temos que um dos componentes possui vários outros sub-componentes. Focando na janela de Detalhes do livro, podemos subdividir essa janela em componentes menores, estes por sua vez teriam métodos encapsulados que facilitariam a criação, manutenção e reuso da interface como um todo.



Exemplo de como subdividir um container em componentes menores

Perceba que no exemplo acima, defini o título e subtítulo do livro (3) como um só componente, porque isso? Este componente recebe os parâmetros da mesma fonte e estes têm o mesmo, meta-dados do livro - *Strings*. Dessa forma é mais simples tratá-los como um só componente. Além disso, eles estão visualmente próximos, podendo ser encapsulados em uma só tag `<div>`.

```
class Titulo extends React.Component {  
  render() {  
    return (  
      <div>  
        <h1> {this.props.book.title}</h1>  
        <h2> {this.props.book.author}</h2>  
      </div>  
    )  
  }  
}
```

```
//A chamada desse componente ficaria assim:  
<Titulo book={selectedBook} />
```

Se fossemos chamar o componente *Titulo* dentro do container *DetalhesDoLivro* teríamos algo como:

```
import Titulo from "./Titulo";  
import BookPhoto from "./BookPhoto";  
import InfoLivro from "./InfoLivro";  
import ComentariosLivro from "./ComentariosLivro";  
  
class DetalhesDoLivro extends React.Component {  
    render() {  
        var selectedBook = this.props.selectedBook;  
        return  
        (  
            <div>  
                <Titulo book={selectedBook} />  
                <BookPhoto book={selectedBook} />  
                <InfoLivro book={selectedBook} />  
                <ComentariosLivro bookNumber={selectedBook.id} />  
            </div>  
        )  
    }  
}
```

Ao início do script temos os imports dos demais arquivos contendo as outras classes, de forma que neste mesmo diretório teríamos os arquivos: *Titulo.js*, *BookPhoto.js*, *InfoLivro.js* e *ComentariosLivro.js*.

## 4. Criando App React

### 4.1. Create-react-app

Uma das formas mais simples de criar um App React é utilizando o pacote npm: **create-react-app**<sup>21</sup>. Este pacote foi criado pelos mesmos desenvolvedores do React (Facebook) para facilitar na criação de aplicativos utilizando o framework node + react. Este pacote cria um projeto node.js completo, com todos os carregadores, compiladores e etc para rodar em um servidor.

Para instalar o pacote (de forma Global, ou seja, disponível em qualquer ambiente do sistema), iremos utilizar os seguintes comandos no terminal:

```
$ npm install -g create-react-app
```

Após instalado, iremos para o diretório principal do nosso projeto e iremos criar um app chamado “myapp”:

```
$ create-react-app myapp
```

No diretório /myapp/ se executarmos o script start teremos:

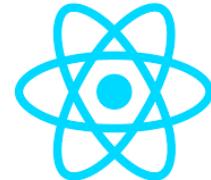
```
$ cd myapp
```

```
$ npm start
```

O navegador web principal do seu OS deve ser aberto automaticamente com uma nova página, no endereço <http://localhost:3000>. Isso quer dizer que seu navegador está acessando um servidor local (localhost) que está “ouvindo” pedidos na porta 3000. Se tudo deu certo, você deve ver a tela abaixo:

---

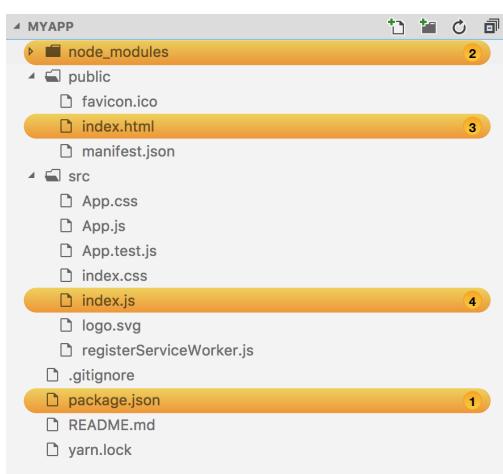
<sup>21</sup> <https://github.com/facebookincubator/create-react-app>



Tela inicial do app criado com *create-react-app*

## 4.2. Organização de Diretórios

Se olharmos o conteúdo do diretório /myapp/, iremos ver algo como:



Estrutura de diretórios e arquivos de app criado com *create-react-app*

O arquivo **package.json** [1] é um arquivo de configuração para o npm. Nele estão contidos informações importantes para o projeto como principais dependências, nome do app, nome do autor, etc.

A pasta **node\_modules** [2] é aonde são salvas todas as dependências locais do seu projeto, basicamente toda vez que você executa algo como :

```
$ npm install --save nomeDoPacote
```

Esse novo pacote será baixado e salvo nessa pasta, seguindo o exemplo acima teríamos algo como: /node\_modules/nomeDoPacote/index.js. Mas não se preocupe,

o npm é quem irá administrar o conteúdo desse diretório, raros são os momentos em que você terá que abrir algum arquivo manualmente deste diretório.

Dentro do diretório `/public/` - que é onde os arquivos públicos disponíveis para serem acessados pelo usuário diretamente - está o arquivo **index.html**[3]. Este é o html básico que receberá todos os componentes criados pelos nossos códigos JS e JSX.

O diretório `/src/` é onde irão ser salvos os nossos scripts que guiarão a execução do(s) App(s) - Sim, você pode ter Apps rodando dentro de Apps, abstraia como encapsulamento, explicado na sessão anterior - aonde o ponto inicial que o navegador deve ler será **index.js** [4].

### 4.3. Electron: WebApp desktop

Este tópico será apenas um extra, um passo que não é necessário para os demais exemplos, porém é importante ser ilustrado. Em alguns casos pode ser que seja necessário o acesso a recursos nativos do sistema, por exemplo: ler uma port Serial proveniente de um Arduino<sup>22</sup> conectado ao computador, nesse caso é aconselhável a criação de um aplicativo desktop. Para isso usaremos o Electron<sup>23</sup>, um framework para criação de aplicativos desktop Cross Platform (ou seja, roda em diversos OS). Algumas das ferramentas utilizadas para a criação deste material foram feitas usando Electron, como o Visual Studio Code, Github Desktop e Kap.

Electron faz com que webapps sejam “empacotados” como apps desktop. Uma versão modificada do navegador Chromium é utilizada como ambiente de execução para os scripts.

A forma mais simples de configurar e rodar um ambiente para com os frameworks descritos é utilizando um boilerplate, algo como um projeto “template” que serve como base para o restante do projeto.

---

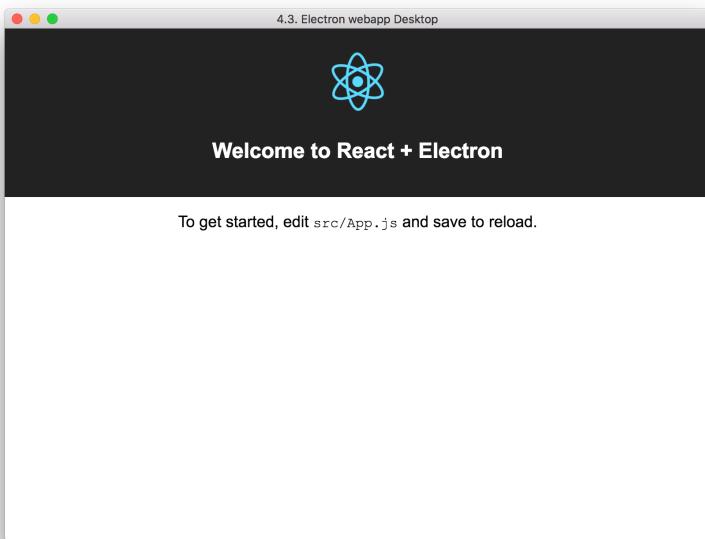
<sup>22</sup> <http://johnny-five.io>

<sup>23</sup> <https://electron.atom.io>

No caso atual iremos utilizar um repositório GitHub como boilerplate:  
<https://github.com/b52/electron-es6-react>. Este projeto traz Electron, com React e um loader -babel - ES2015 (ES6). Perfeito para o que precisamos.

Seguindo esses passos, no terminal, é possível baixar e executar esse projeto:

```
$ git clone https://github.com/b52/electron-es6-react.git  
$ cd electron-es6-react  
$ npm install  
$ npm start
```



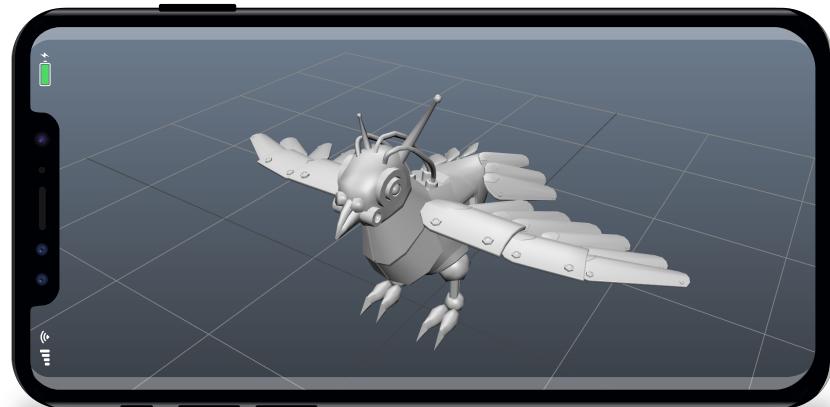
No diretório `./examples/4.3. Electron webapp desktop/` é possível encontrar um app já configurado com ambos os frameworks.

## 5. Three.js<sup>24</sup>

### 5.1. WebGL: OpenGL para web



**OpenGL<sup>25</sup>** é uma API para renderização avançada de gráficos 2D e 3D, a versão OpenGL ES (Embedded Systems - sistemas embarcados) é uma versão cross-platform desta API. Sim, é o OpenGL que auxilia no desenvolvimento de Jogos para celulares modernos e atualmente é base para a revolução em visualização de dados trazida pelos avanços em Realidade Aumentada (AR - ARkit<sup>26</sup> e ArCore<sup>27</sup>)



Exemplo: OpenGL rodando em um dispositivo mobile (iPhone X)

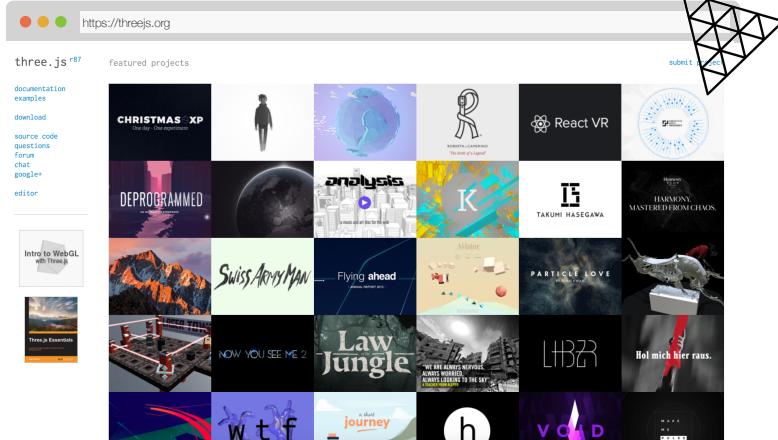
<sup>24</sup> <https://medium.com/@necsoft/three-js-101-hello-world-part-1-443207b1ebe1>

<sup>25</sup> [https://www.khronos.org/webgl/wiki/WebGL\\_and\\_OpenGL\\_Differences](https://www.khronos.org/webgl/wiki/WebGL_and_OpenGL_Differences)

<sup>26</sup> <https://developer.apple.com/arkit/>

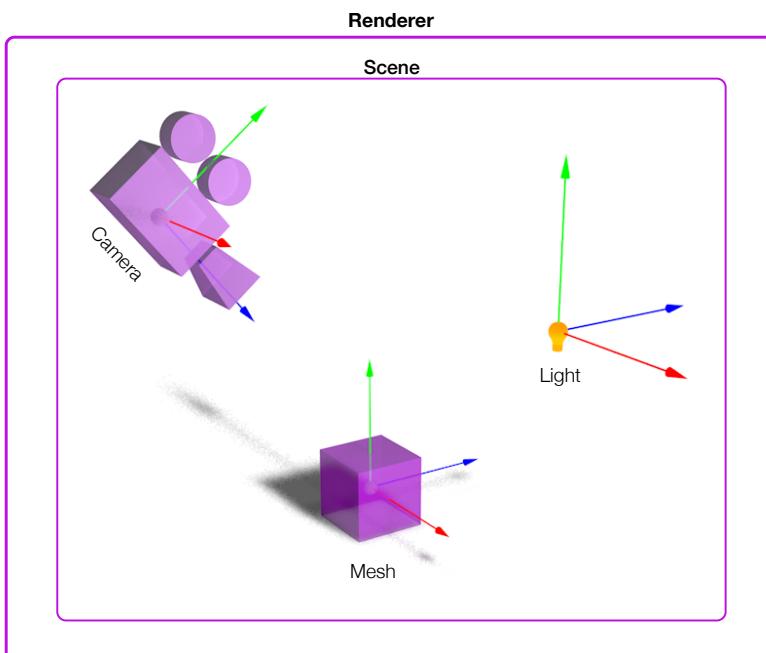
<sup>27</sup> <https://developers.google.com/ar/>

**WebGL**<sup>28</sup> é um motor de rasterização baseado no OpenGL ES 2.0, ele desenha pontos, linhas e triângulos baseado em um determinado código. WebGL roda na GPU do computador



tuitiva. A documentação no site da biblioteca é muito bem feita, contando com diversos exemplos e referências.

## 5.2. Renderizadores, Cenas e Camera



**Three.js** é um biblioteca JavaScript desenvolvida por Ricardo Cabello em 2010<sup>29</sup>. Pensada para ser leve e fácil de usar, esta ferramenta permite a utilização de gráficos 3D direto do navegador web, utilizando WebGL de forma mais intuitiva.

Three.js segue uma lógica simples quanto à estrutura e criação de renderizações, cada visualização (vamos chamar de “janela” para facilitar) possui um renderizado (*Renderer*), este contém uma ou mais cenas, estas contém uma ou mais câmeras. Seguindo a hierarquia ao lado.

<sup>28</sup> <https://webgl2fundamentals.org>

<sup>29</sup> <https://github.com/mrdoob/three.js/>

O **Renderer** é o responsável por calcular os pixels de cada câmera da cena e transformá-los em uma imagem 2D, com pode ser vista ao lado.

A **Scene** é o conjunto de objetos existentes, um deles sendo a **Camera**, esta pode ser entendida como o ponto de vista da cena, sua posição seria análoga a posição dos olhos do usuário.

Estes objetos de cena podem ser malhas, luzes, skyboxes.. etc. Estes serão melhor explorados nas próximas sessões.

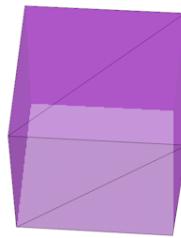
Exemplo básico de uma cena, utilizando **WebGLRenderer**:

```
var camera, scene, renderer;
var width=600;
var height=300;

init();

function init() {
    camera = new THREE.PerspectiveCamera(70, width/height, 1,
1000);
    camera.position.z = 300;
    scene = new THREE.Scene();

    renderer = new THREE.WebGLRenderer();
    renderer.setPixelRatio(window.devicePixelRatio);
    renderer.setSize(width,height);
    renderer.render(scene, camera);
    document.body.appendChild(renderer.domElement);
}
```

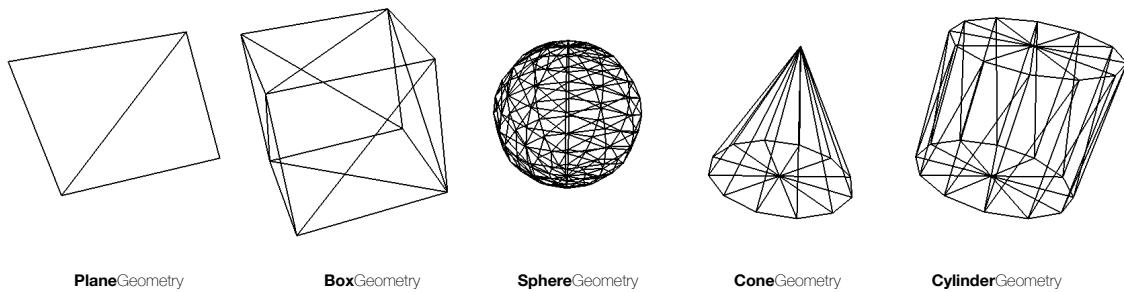


}

### 5.3. Geometrias, Materiais e Malhas

Uma Malha é basicamente uma união de Geometria com um Material.

**Geometria<sup>3º</sup>** (geometries) é um objeto definido por um conjunto de vértices e arestas que formam faces. Three.js já possui geometrias pré prontas, como cubos, esferas e planos.



#### Geometrias básicas do THREE.js

Porém, se necessário pode-se criar a definição de posição de vertices programaticamente, ou ainda carregar uma geometria de um modelo 3D criado em outro programa como o Blender. Iremos explorar isso nas próximas sessões.

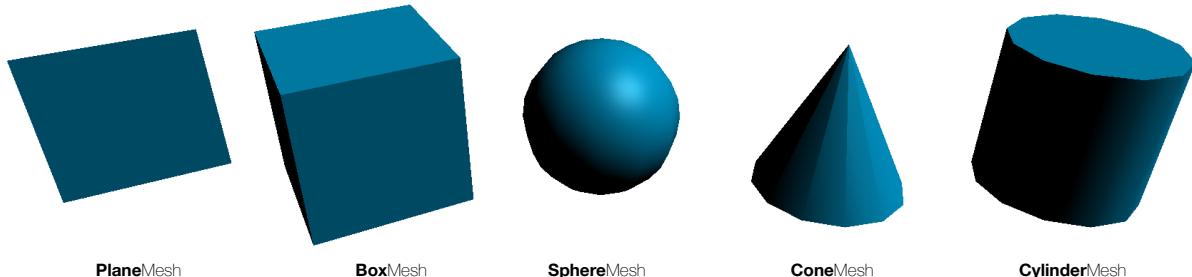
Quando se trata de **Materiais** (materials) o three.js também disponibiliza um conjunto de materiais, os mais simples são os listados abaixo:



#### Principais Materiais do THREE.js

<sup>3º</sup> <https://threejs.org/docs/#api/geometries/>

Como dito anteriormente, em conjunto, uma geometria e um material formam uma **malha** (mesh) , esta que será renderizada na cena principal.



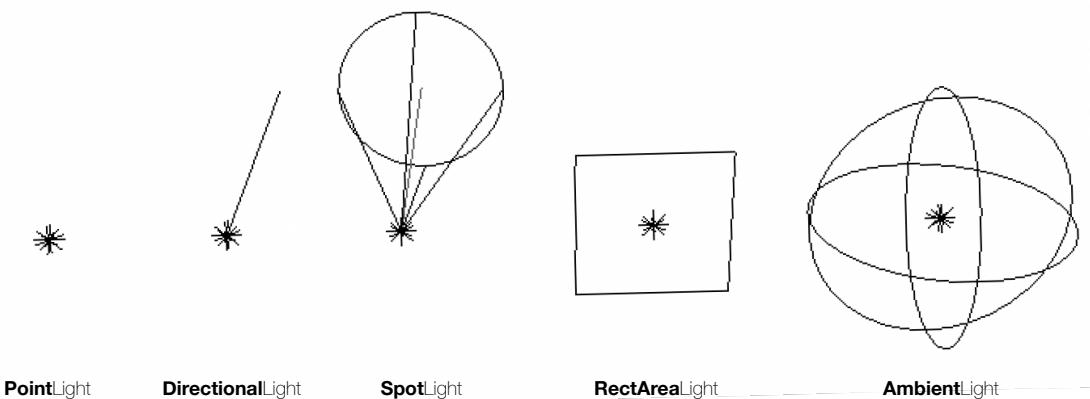
Principais Geometrias com *PhongMaterial*

No exemplo acima, o script para criar o cubo visualizado é:

```
var geometry = new THREE.BoxGeometry( 100, 100, 100 );
var material = new THREE.MeshPhongMaterial( {color:
"#0099cc"} );
var boxMesh = new THREE.Mesh( geometry, material );
scene.add( boxMesh );
```

## 5.4. Iluminação

Existem 6 tipos básicos de luzes, Ponto, Direcional, Spot, Área Retangular, Hemisferica e Ambiente, como podem ser vistos nas representações abaixo:



Tipos básicos de luzes do Three.js

As luzes são adicionadas a cenas como um objeto qualquer, por exemplo:

```
var light = new THREE.DirectionalLight(0xffffffff, 1);
light.position.set(1, 1, 1).normalize();
scene.add(light);
```

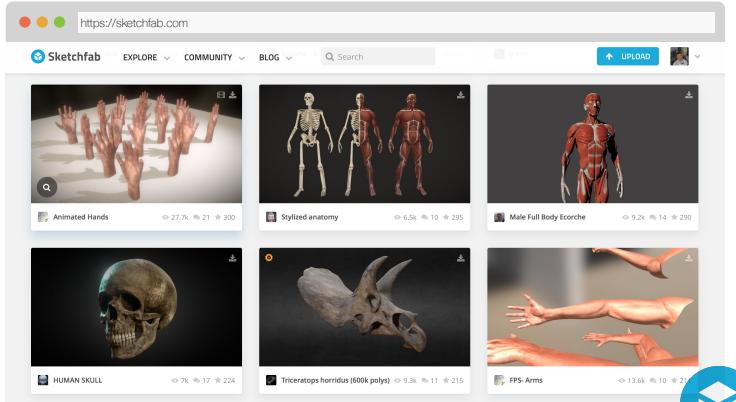
Como dito no primeiro tópico sobre Three.js, a página de exemplos é um recurso muito rico que deve ser explorado, abaixo um dos exemplos com efeitos avançados de luz e sombra.



## 5.5. Importando Objetos

Criar Geometrias complexas utilizando apenas código é algo inviável em muitos momentos. Normalmente, quando se quer visualizar um modelo 3D na web, modela-se esse(s) objeto(s) utilizando algum software de modelagem, como Blender ou Fusion360 - no caso de modelagem paramétrica - e exporta-se esse modelo em um formato compatível com Three.js.

Atualmente Three.js já conta com suporte a diversos formatos padrão de arquivos descrevendo objetos 3D. Dentre eles temos formatos como .3ds (Autodesk



3DS Max<sup>31</sup> - software de modelagem, animação e renderização) e .obj (formato para definição de geometrias).



Uma curiosidade:

Three.js possui suporte a

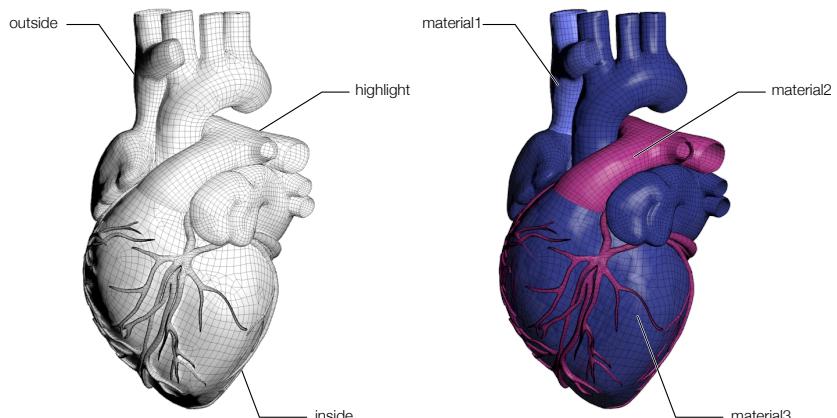
VRML - Virtual Reality Modeling Language - que, de forma simplificada e antiquada, faz o que desejamos alcançar ao fim deste módulo: criar ambientes 3D utilizando “tags” - de forma descritiva - ao invés de criar ambientes/modelos de forma imperativa e tudo isso com foco para aplicações web.

Neste material iremos focar em utilizar apenas .obj e .mtl (formato que define os materiais das geometrias, pois .obj define apenas vértices e ordem de conexão dessas arestas).

Todos os formatos compatíveis com Three.js estão listados na página de exemplos: <https://threejs.org/examples/?q=loader>

## 5.6. Organização de Modelos Importados

Para importarmos um objeto precisamos primeiro ter esse objeto modelado. Nesse ponto temos duas opções, modelar um objeto ou baixar algo já pronto.

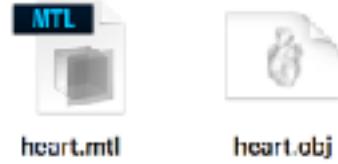


No caso de baixar algo pronto sugiro o site sketchfab<sup>32</sup>

<sup>31</sup> <https://www.autodesk.com/products/3ds-max/overview>

<sup>32</sup> <https://sketchfab.com>

Utilizaremos, a fim de exemplificação, o modelo de um coração (sim o mesmo modelo utilizado na capa desta apostila). O arquivo deste modelo ficará disponível no repositório github deste material.



O modelo em questão é separado em três partes, cada uma com um material diferente.

Na pasta /examples/5.5 Importing Objects/models temos dois arquivos: heart.obj e heart.mtl. O nosso script de exemplo irá carregar primeiro a geometria (.obj) e depois os materiais (.mtl).

Utilizando o mesmo esqueleto de script dos exemplos anteriores, iremos importar os seguintes arquivos: ObjLoader.js e MTLLoader.js, ambos na pasta ..//js/.

Nosso arquivo *index.html*, após as importações ficará:

```
<html>
  <head>
    <title>Example 5.5 Importing Objects</title>
    <!--Import three.js-->
    <script src="js/three.js"></script>
    <script src="js/MTLLoader.js"></script>
    <script src="js/OBJLoader.js"></script>
  </head>

  <body>
    <h1>Loaded Obj</h1>
    <script src="js/index.js"></script>
  </body>

</html>
```

O arquivo *js/index.js* será o script principal desse exemplo. Utilizando *basic.js* do exemplo anterior, adicionaremos a os dois *loaders*, para .mtl e .obj respectivamente:

```
var camera, scene, renderer;
var width = 600;
var height = 300;

init();
animate();

function init() {
    camera = new THREE.PerspectiveCamera(70, width / height,
1, 1000);
    camera.position.z = 10;
    scene = new THREE.Scene();
    scene.background = new THREE.Color(0xffffffff);

    var ambient = new THREE.AmbientLight(0x444444);
    scene.add(ambient);

    var directionalLight = new THREE.DirectionalLight(0xf-
eedd);
    directionalLight.position.set(0, 0, 1).normalize();
    scene.add(directionalLight);

//Loading Obj-----
var onError = function (xhr) { };
//THREE.Loader.Handlers.add(/\dds$/i, new THREE-
.DDSLoader());
var onProgress = function (xhr) {
    if (xhr.lengthComputable) {
        var percentComplete = xhr.loaded / xhr.total * 100;
```

```

        console.log(Math.round(percentComplete, 2) + '%
downloaded');
    }
};

//Carregando Material
var mtlLoader = new THREE.MTLLoader();
mtlLoader.setPath('models/');
mtlLoader.load('heart.mtl', function (materials) {
    materials.preload();
    //Quando o material for carregado, carregamos a geometria (.obj)
    var objLoader = new THREE.OBJLoader();
    objLoader.setMaterials(materials);
    objLoader.setPath('models/');
    objLoader.load('heart.obj', function (object) {
        object.position.y = - 2; //<-- Ponto aonde podemos
        fazer alteracoes no objeto
        scene.add(object); //Adicionando o objeto a cena
    }, onProgress, onError);
});

//-----
renderer = new THREE.WebGLRenderer();
renderer.setPixelRatio(window.devicePixelRatio);
renderer.setSize(width, height);
document.body.appendChild(renderer.domElement);
}

function animate() {
    requestAnimationFrame(animate);
    renderer.render(scene, camera);
}

```

Esse é o resultado final do index.html executando index.js. O modelo 3D é exibido na tela.

Pode-se ainda alterar as propriedades tanto do material quanto da geometria após carregadas.

No exemplo abaixo alteramos as propriedades de posição, rotação e escala de cada instância do modelo.

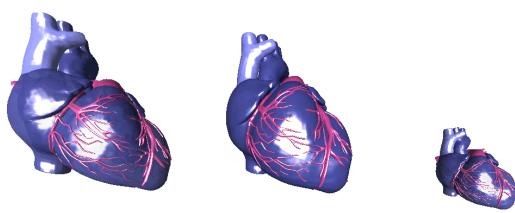
Por exemplo, o coração ao lado esquerdo foi alterado da seguinte forma:

## 5.5. Importing Objects

### Simple Imported Object



### Multiple Imported Object



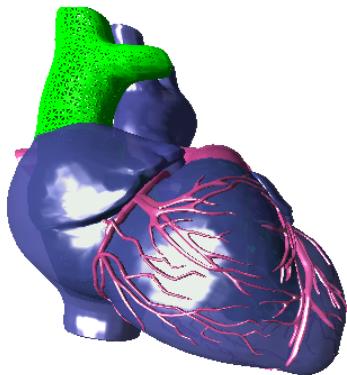
```
var obj2 = object.clone();
obj2.position.x = -8;
obj2.rotation.y = 0.8;
scene.add(obj2);
```

Podemos ainda alterar o material após carregar o .mtl. Se quisermos que ao invés de roxo, nosso modelo seja verde (iremos refazer isso quando adicionarmos interações, quando o mouse passar por cima do modelo, ele irá mudar de cor), podemos alterar no momento em que carregamos o arquivo .mtl:

```
materials.materials.material1 = new THREE.MeshPhongMaterial(
  {color: "#00FF00", wireframe:true}
);
```

Para visualizar o objeto em 360 graus, iremos rotacioná-lo no eixo Y. Para isso iremos salvar a instância do objeto de forma global e dentro do método *animate()* teremos como alterar o parâmetro de rotação do objeto (*object.rotation.y+=0.01*).

O método `init()` é executado uma vez, ao início do programa e é nele em que devem ser feitos todos os carregamentos de arquivos e configurações de cena e de renderizador.



Já a função `animate()` é invocada a cada quadro - idealmente seu projeto irá rodar a 60 quadros por segundo (FPS, frames per second), porém até 30FPS ainda são valores aceitáveis para experiência com minima fluidez para o

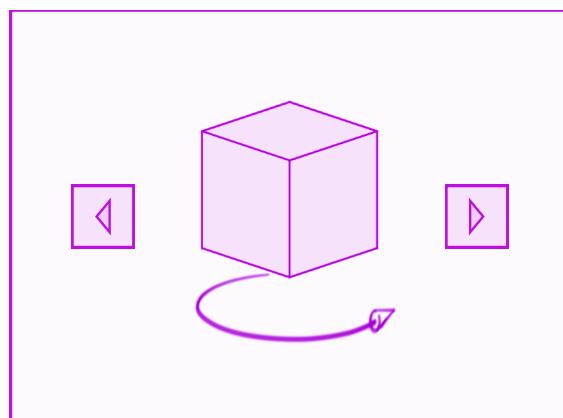
usuário. Nesse método é aonde devemos fazer as alterações desejadas durante a execução do app

## 5.6. Interação com Objetos

Apenas carregar um objeto 3D não é o suficiente para termos uma boa visualização, é necessário permitir que o usuário interaja de alguma forma com o objeto em questão, para que possa escolher o melhor ângulo de visão ou ate mesmo selecionar partes específicas para ver mais detalhes de um objeto.

Uma interação simples seria alterar a rotação do objeto de acordo com *inputs* do usuário. No primeiro momento iremos fazer isso utilizando teclado e botões na tela.

Em um próximo exemplo iremos abordar o mesmo conceito de interação, porém utilizando mouse.



Precisamos então criar dois botões, um para rotacional positivamente e outro para rotacional negativamente, ambos em relação ao eixo Y.

No arquivo *index.html* serão adicionados dois botões (também adicionando o arquivo *css/style.css* para melhor layout):

```
<div class="left" onclick="RotationLeft()">
    
</div>

<div class="right" onclick="RotationRight()">
    
</div>
```

Esses dois botões estão chamando, ao serem clicados, os métodos: *RotationLeft()* e *RotationRight()* respectivamente. Estes métodos estão definidos em *js/index.js*.

```
var rotation = 0;
...
function RotationLeft() {
    rotation -= 0.1;
}

function RotationRight() {
    rotation += 0.1;
}
```

Dentro do método *animate* iremos alterar a rotação do objeto carregado:

```
function animate() {
    ...
    loadedObj.rotation.y = rotation;
    ...
}
```

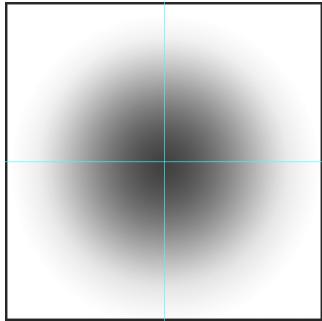
Nesse Momento temos a rotação Básica do objeto, porém para fins didáticos iremos adicionar outros objetos nessa cena: um cubo definindo o tamanho máximo do modelo, uma sombra projetada e um indicador de rotação.

O primeiro objeto extra, o cubo, é bem simples de ser adicionado, no método *init()*:

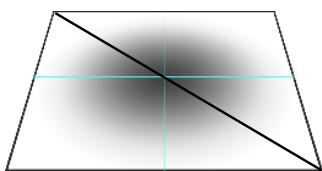
```
var group;
var cube;
...
function init() {
...
group = new THREE.Object3D();
...
group.add(loaderObj);
...
var material = new THREE.MeshBasicMaterial({
    color: 0xBD10E0,
    wireframe: true,
    opacity: 0.2,
    transparent: true
});
var geometry = new THREE.BoxGeometry(6, 6, 6);
cube = new THREE.Mesh(geometry, material);
cube.position.y = 1.5;
group.add(cube);
...
}
```

No código acima criamos um objeto chamado THREE.Object3D. Basicamente Object3D é uma definição genérica de um objeto 3D que pode conter diversas malhas, de forma a agrupar objetos, criando formas mais complexas que podem ser alteradas em conjunto.

No caso do exemplo, iremos rotacionar todos os objetos de cena de uma vez, mantendo as posições e rotações relativas entre eles constantes.



**texture:** shadow.png



**PlaneMesh**

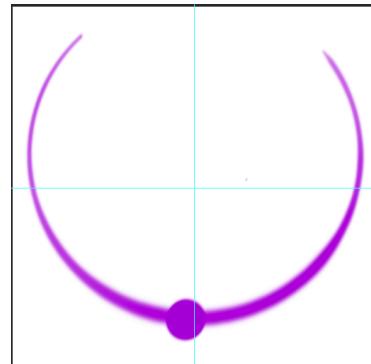
Para criarmos a sombra, de forma simples, iremos utilizar uma textura com um gradiente circular centralizado. Esta textura será aplicada a um material de um PlaneMesh:

```
//Shadow
var shadowtexture = new
THREE.TextureLoader().load("tex/shadow.png");
shadowtexture.wrapS = THREE.ClampToEdgeWrapping;
shadowtexture.wrapT = THREE.ClampToEdgeWrapping;

var shadowMaterial = new THREE.MeshBasicMaterial
( {
color: 0xff0000,
map: shadowtexture,
transparent: true,
opacity: 0.25 });

var shadowGeometry = new THREE.PlaneGeometry(5, 5, 1);
var shadow = new THREE.Mesh(shadowGeometry, shadowMaterial);
shadow.position.y = -3;
shadow.rotation.x = -Math.PI * 0.5;
group.add(shadow);
```

O método `THREE.TextureLoader().load("textura.png")` carregamos a textura, que é mapeada para o material que será aplicado em um Plano (`planeGeometry`), esse conjunto forma o `PlaneMesh` da sombra que é adicionado ao grupo da cena.



**texture:** circle.png

Para criarmos o Indicador de rotação iremos utilizar o mesmo método descrito acima, porém trocando a textura para a textura ao lado.

Pelo fato de todos os objetos descritos estarem contidos no mesmo Object3D, as alterações em posição e rotação feitas no group irão afetar todos os demais objetos.

A segunda forma de interagir com esse exemplo é via teclado, utilizando as setas para a direita e esquerda ao invés dos botões na tela. Para isso iremos alterar: *document.onkeydown*;

```
document.onkeydown = checkKey;

function checkKey(e) {
    e = e || window.event;

    if (e.keyCode == '37') {
        // left arrow
        rotation -= 0.1;
    }
    else if (e.keyCode == '39') {
        // right arrow
        rotation += 0.1;
    }
}
```

Como a rotação do grupo já está sendo alterada a cada frame, nós só precisamos alterar a variável *rotation* quando uma tecla for pressionada. Atente ao fato de que as setas são definidas como 37 e 39, esse método usa keyCode<sup>33</sup>, aonde cada teclado do teclado é mapeada para um número.

Certo, temos um objeto 3D rotacionando na tela utilizando inputs discretos do usuário, porém temos também um cubo, que mesmo “transparente” pode atrapalhar a visualização do objeto. Vamos então permitir que o usuário defina quando o cubo

---

<sup>33</sup> <http://keycode.info>

deve ser renderizando ou não na cena. Começaremos criando um botão no *index.html*:

```
<button onclick="ShowHideCube()">show/hide Cube</button>
```

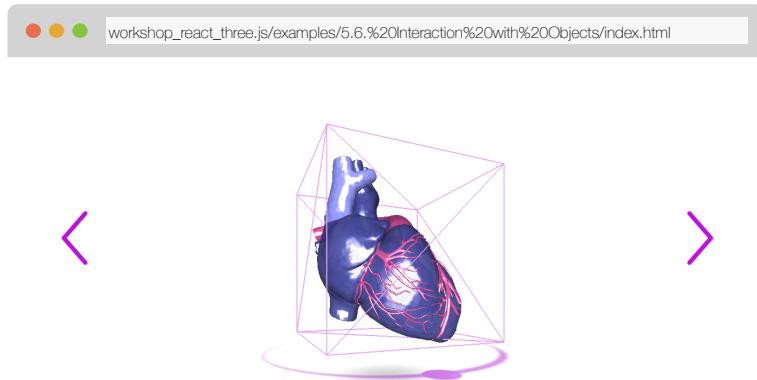
Quando este botão for clicado irá chamar a função:

```
function ShowHideCube() {  
    cube.visible = !cube.visible;  
}
```

Simples e pode ser aplicado a todos os outros objetos de cena individualmente (ou agrupados).

## 5.7. Controles de Cena

Neste tópico iremos criar uma interação similar anterior, porém ao invés de clicar com o teclado iremos utilizar inputs contínuos, como a posição do mouse.



Para utilizar a posição do mouse como input devemos primeiro registrar os eventos de clicar com o botão esquerdo e mover o mouse:

```
function init(){  
    ...  
  
    renderer.domElement.addEventListener(  
        "mousedown",
```

```

function(e){
    onDocumentMouseMove(e);
    this.addEventListener("mousemove", onMouseMove);
}

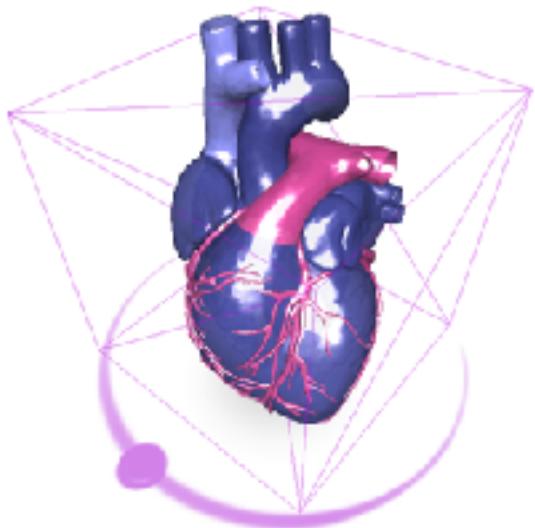
renderer.domElement.addEventListener(
    "mouseup",
    function(e){
        this.removeEventListener("mousemove", onMouseMove);
    });
}

function onMouseMove( event ) {
    mouseX = ( event.clientX - width/2 ) / 2;
    mouseY = ( event.clientY - width/2 ) / 2;
}

function animate() {
    ...
    group.rotation.y = mouseX * 0.02;
    ...
}

```

Utilizou-se o `renderer.domElement` ao invés de `document` pois estamos interessados apenas no movimento do mouse quando este estiver dentro das bordas do nosso renderizador.



Explicando ainda melhor o script anterior: apenas quando o botão esquerdo do mouse estiver pressionado a variável `MouseX` será atualizada, rotacionando assim o objeto. `MouseX` foi multiplicada por

0.02 para tornar a rotação mais lenta.

Podemos ainda rotacionar no eixo X utilizando o Input do *mouseY*:

```
group.rotation.x = mouseY * 0.02;
```

Outra forma, mais simples e mais eficiente de rotacionar um objeto em tela é usar o controle de órbita: *OrbitControls*.

No *index.html* iremos adicionar a tag para importar o script *OrbitControls.js*:

```
<script src="js/OrbitControls.js"></script>
```

Em *index.js* deve-se adicionar o seguinte script:

```
var controls;

...

function init(){
    ...

    controls = new THREE.OrbitControls(
        camera,
        renderer.domElement
    );
    controls.addEventListener( 'change', render );
    controls.enableZoom = true;

    ...
}

function animate(){
    ...
}
```

```

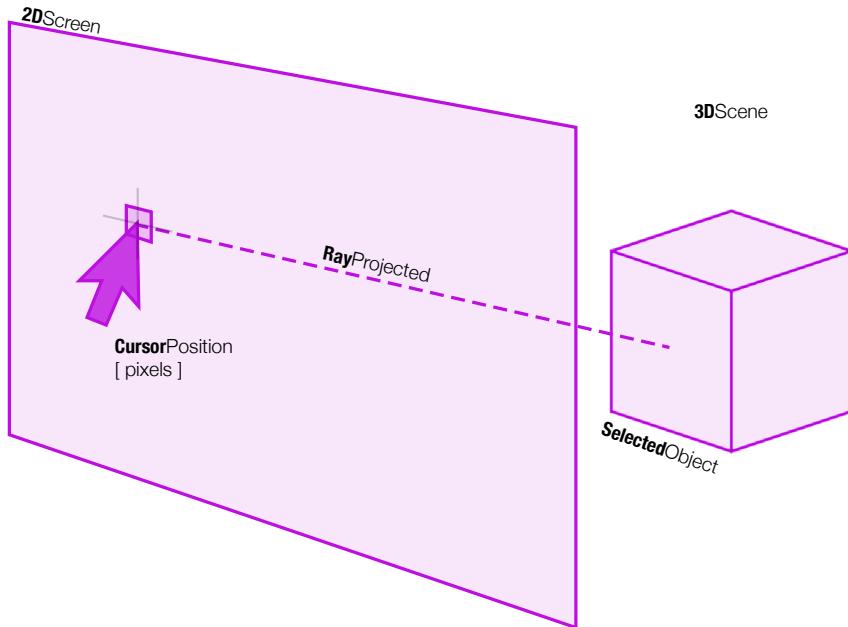
    controls.update();
    ...
}

function render() {
    renderer.render( scene, camera );
}

```

## 5.8. Selecionando Objetos - Raycasting

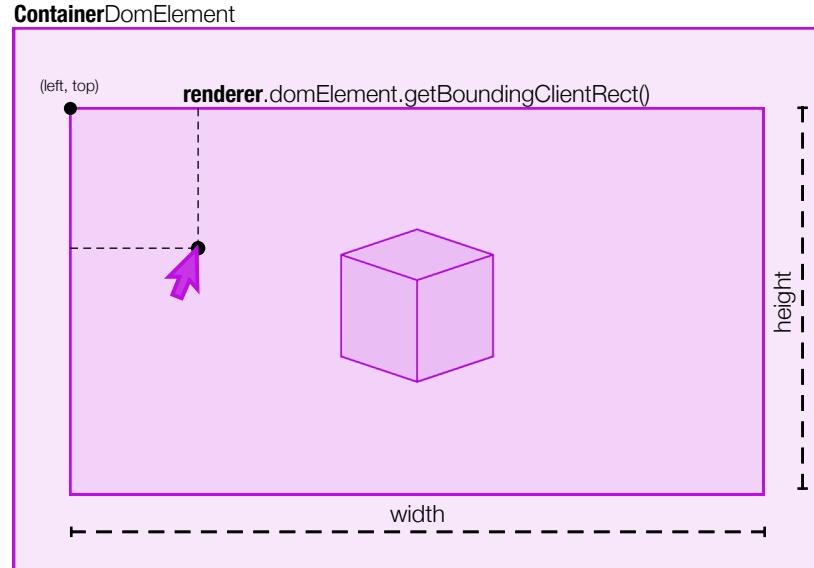
Para selecionar um objeto 3D em cena precisamos levar várias coisas em consideração, como posição dos objetos em cena, posição da câmera, tipos de projeção, etc. Para facilitar esse processo, existe uma técnica chamada *RayCasting*.



Basicamente projeta-se um “raio” virtual, do ponto em que está o mouse até o ‘infinito’, perpendicularmente a tela. O primeiro objeto 3D que for ‘tocado’ por este raio será o objeto tido como selecionado.

Na realidade esse processo é bem mais complicado que isso, envolvendo conversões de matrizes e etc. Para nossa conveniência Three.js possui um conjunto de métodos justamente para a implementação rápida e simples de RayCasting.

Para encontrar a posição do mouse relativo ao tamanho do renderer iremos utilizar o método `onMouseMove`, utilizando o conceito ao lado para localizar o ponteiro do mouse.



```
...
var mouse = new THREE.Vector2();
...

function onMouseMove(event) {

    var rect = renderer.domElement.getBoundingClientRect();
    mouse.x = (( event.clientX - rect.left) / rect.width )*2-1;
    mouse.y = - (( event.clientY - rect.top) / rect.height )*2+1;

}
```

Com a posição do mouse relativa à tela, podemos agora começar a “traçar” o raio - o processo de raycasting propriamente dito começa agora - vamos começar criando e inicializando duas variáveis, `raycaster` e `INTERSECTED`, a segunda irá receber os objetos “atingidos” pelo raio:

```
var raycaster;
var INTERSECTED;
```

```

...
function init() {
    ...
    raycaster = new THREE.Raycaster();
    ...
}

```

A cada frame iremos então verificar, baseado na posição atual do mouse, quais objetos dentro de um grupo específico de objetos, quais destes estão em contato com o raio. O primeiro objeto desta lista é o mais próximo à tela, logo o objeto que queremos selecionar.

```

function render() {

    raycaster.setFromCamera(mouse, camera);

    var intersects = raycaster.intersectObjects(loaderObj.children);

    if (intersects.length > 0) {

        if (INTERSECTED != intersects[0].object) {
            console.log("Selected", INTERSECTED);
        }
    } else {

        INTERSECTED = null;
    }

    renderer.render(scene, camera);
}

```

Se a lista de objetos Intersecionados estiver vazia, logo nenhum objeto está em evidência, pode-se então limpar o valor de *INTERSECTED*.

Uma parte do script que deve ser enfatizada é o fato de usarmos *loadedObj.children* ao invés de *scene.children* (tente alternar os dois grupos de objetos código seguinte e veja como os objetos se comportam).

No exemplo acima a intenção é descobrir qual dos objetos contidos em *loadedObj* - o *.obj* - está "sob" o ponteiro do mouse, logo deve-se procurar se o raio toca algum dos objetos importados.

Agora que o objeto em evidência é conhecido, pode-se então fazer alterações no mesmo. Como alterar a cor do material do mesmo. Iremos alterar para verde (hex color code - *0x00ff00*).

Alterando o código anterior teremos:

```
function render() {  
    // find intersections  
    //camera.updateMatrixWorld();  
  
    raycaster.setFromCamera(mouse, camera);  
    var intersects = raycaster.intersectObjects(loadedObj.children);  
  
    if (intersects.length > 0) {  
        //console.log("intersected", intersects);  
  
        if (INTERSECTED != intersects[0].object) {  
            if (INTERSECTED)  
                INTERSECTED.material.color.setHex(INTERSECTED.currentHex);  
            INTERSECTED = intersects[0].object;  
        }  
    }  
}
```

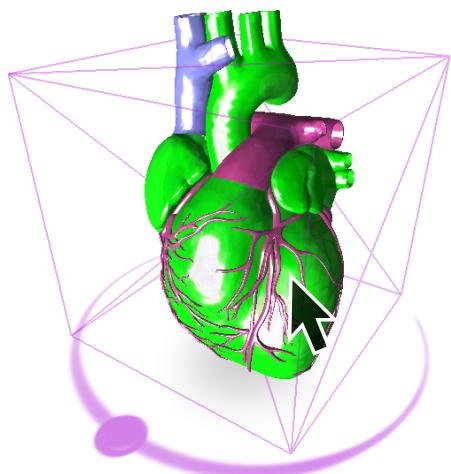
```

INTERSECTED.currentHex = INTERSECTED.material.color.get-
Hex();
INTERSECTED.material.color.setHex(0x00FF00);

}

} else {
    if (INTERSECTED){
        INTERSECTED.material.color.setHex(INTERSECTED.current-
Hex);
    }
    INTERSECTED = null;
}

```



Este é o resultado que alcançamos até o momento. Outras propriedades do modelo 3D podem ser exploradas, como nome do objeto que está sendo selecionado, posição (x,y,z), escala, etc. Iremos explorar algumas delas nos próximos exemplos (após integrarmos three.js com React e node.js). Perceba que tanto o cubo em wireframe quanto o círculo (indicador de rotação) não afetam como as interseções com o raio são calculadas, pois ambos objetos não estão contidos no grupo *loadedObj.children*, estão no grupo *scene.children*.

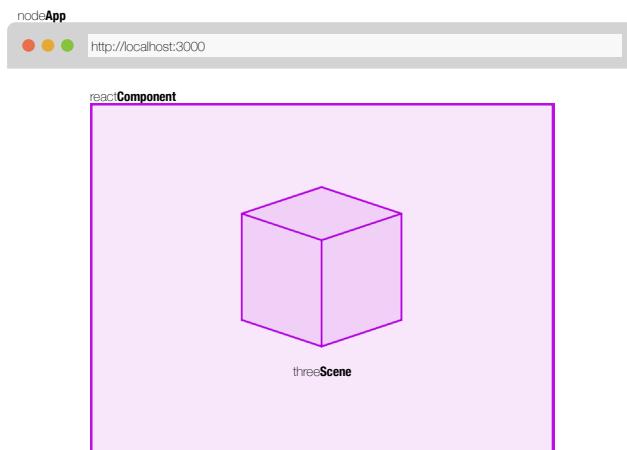
## 6. Encapsulamento do Three.js como Componente React<sup>34</sup>

Certo, temos um ambiente 3D implementado.. porém como unir este ambiente a todas as vantagens do node.js e react.js mencionadas nos capítulos iniciais? - autores com pouca experiência se utilizam de perguntas retóricas para introduzir novos assuntos - Encapsulando o ambiente 3D em um componente React.

Neste ponto poderíamos pular diretamente para o capítulo 7, porém se você leitor deseja ter mais controle sobre seu ambiente 3D, este capítulo pode lhe ser útil.

Se você deseja apenas utilizar JSX e não se preocupar com o que acontece por trás do seu app, pule para o capítulo 7.. Ou se você estiver com muita pressa para entregar um projeto funcionando, dê uma olhada nos componentes do Exemplo 8 (diretório GitHub deste material).

### 6.1. Componente Container3



Iremos agora encapsular, aos poucos, o que já implementamos utilizando Three em um componente React.

A cena do Three.js estará contida em um componente React.js, este contido em um App node.js. Parece que estamos complicando algo que era tão simples, não? - spoiler: não. Esse passo é importante para deixar seu código mais fácil de ler,

<sup>34</sup> <https://codepen.io/jacoboakley/pen/xRxqdO>

de se fazer manutenção<sup>35</sup> e até reutilizar partes do código em outros projetos ou views do app. Ou seja, você estará ganhando tempo e simplificando seu projeto.

Vamos ao script. Criaremos um componente react simples, só com dois detalhes: importa o three.js como um pacote nem e retorna um “canvas” html5 no render. Ficando assim:

```
import React, { Component } from 'react';
import * as THREE from 'three'; //Importando Three

class ContainerThree extends Component {
  constructor(props) {
    super(props);
  }

  componentDidMount() {
    //Quando o componente foi montado
  }

  render() {
    //Retornando Canvas Html
    return (
      <div>
        <canvas ref="threeCanvas">
        </canvas>
      </div>
    );
  }
}

export default ContainerThree;
```

---

<sup>35</sup> “O único software que não precisa de manutenção é aquele que nunca é usado” - algum professor da minha graduação, 2014.

Agora iremos implementar o renderer para a cena 3D, a diferença para os exemplos anteriores é que esse renderizador ao invés de apontar para um novo elemento em *body*, irá apontar para nosso canvas (utilizando refs - referências do react). Como precisamos que o canvas esteja montado para poder-mos utilizá-lo como referência, implementaremos nosso ambiente 3D no método *componentDidMount()*.

```
import React, { Component } from 'react';
import * as THREE from 'three';
//Variaveis globais da cena
let renderer, scene, camera, mainSphere;

class Container3 extends Component {
  constructor(props) {
    super(props);
  }

  componentDidMount() {

    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(50, 500 / 400, 0.1, 1000);

    //this.refs.threeCanvas é o nosso canvas
    //alpha:true faz com que o fundo da cena fique transparente
    renderer = new THREE.WebGLRenderer({
      canvas: this.refs.threeCanvas,
      alpha:true
    });

    renderer.setSize(500, 400);

    var geometry = new THREE.SphereGeometry(3, 24, 24);
    var material = new THREE.MeshBasicMaterial({
      color:0x00aaff,
      wireframe:true,
```

```

        transparent:true,
        opacity:0.4
    });
    mainSphere = new THREE.Mesh(geometry,material);

    scene.add(mainSphere);

    camera.position.z = 10;

    this._render();
}

_render=()=>{

    //Este método será chamado a cada frame
    requestAnimationFrame(this._render);
    renderer.render(scene, camera);
    mainSphere.rotation.y+=0.01;

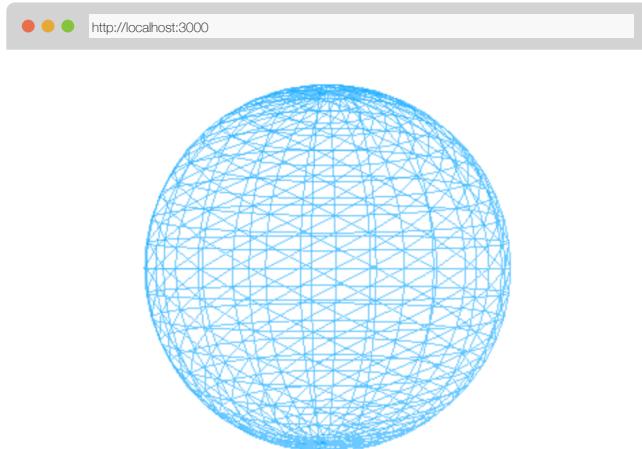
}

render() {
    return (
        <div>
            <canvas ref="threeCanvas">
            </canvas>
        </div>
    );
}
}

export default Container3;

```

Utilizou-se `_render()` ao invés do nome sem o underline no inicio para não haver confusões futuras com os nomes das funções. Uma observação importante é



que utilizamos em `_render()` uma estrutura chamada “fat arrow function”<sup>36</sup>, presente no ES6.

Desta forma todas as alterações que desejamos fazer durante a execução da cena devem ser feitas em `_render()`.

Pode-se ainda alterar os parâmetros do renderer ou elementos da cena alterando-se os `props` passados ao componente. Por exemplo: se ao invés de uma esfera azul quiséssemos um cubo verde rotacionando na cena, poderíamos passar isso como parâmetros:

```
import React, { Component } from 'react';
import * as THREE from 'three';
let renderer, scene, camera, mainObject;

class Container3 extends Component {
  constructor(props) {
    super(props);
  }

  componentDidMount() {
    scene = new THREE.Scene();
    camera = new THREE.PerspectiveCamera(50, 500 / 400, 0.1, 1000);

    renderer = new THREE.WebGLRenderer({ canvas: this.refs.threeCanvas, alpha: true });
    renderer.setSize(500, 400);

    //definindo um prop booleano
```

---

<sup>36</sup> <https://www.sitepoint.com/es6-arrow-functions-new-fat-concise-syntax-javascript/>

```

var wireframe = false;
if (this.props.wireframe)
    wireframe = true;

//alterando a cor do material
var material = new THREE.MeshBasicMaterial({ color: this.props.objectColor, wireframe: wireframe, transparent: true, opacity: 0.4 });
var geometry;

//checando o tipo de geometria a ser renderizado
if (this.props.object == "sphere") {
    geometry = new THREE.SphereGeometry(3, 24, 24);
}

else if(this.props.object == "cube"){
    geometry = new THREE.BoxGeometry(3,3,3);
}

mainObject = new THREE.Mesh(geometry, material);
scene.add(mainObject);
camera.position.z = 10;

this._render();
}

_render = () => {

requestAnimationFrame(this._render);
renderer.render(scene, camera);
mainObject.rotation.y += 0.01;

}

render() {

```

```

        return (
      <div>
        <canvas ref="threeCanvas">
      </canvas>
    </div>
  );
}

}

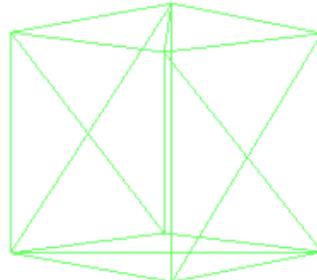
```



```
export default Container3;
```

Na chamada do componente  
usa-se:

```
<Container3
  object="cube"
  objectColor={0x00FF00}
  wireframe />
```



## 6.2. Importando Objetos - Componente ObjViewer

Com o three.js já encapsulado podemos re-implementar a importação e visualização de objetos 3D - assim como fizemos no capítulo 5.5. Como agora temos node.js e npm como base do nosso app podemos utilizar um pacote npm para importar o objeto 3D. Utilizaremos *three-react-obj-loader*<sup>37</sup> e *three-react-mtl-loader*<sup>38</sup>, para carregar as geometrias e as definições de materiais, respectivamente. Iremos utilizar esses pacotes por dois motivos:

1. O Three.js não possui os scripts para carregar modelos contido no script principal.
2. Os pacotes mencionados são versões dos scripts OBJLoader.js e MTL-Loader.js rescritos utilizando classes ES6 - compatível com React.

---

<sup>37</sup> <https://www.npmjs.com/package/three-react-obj-loader>

<sup>38</sup> <https://www.npmjs.com/package/three-react-mtl-loader>

No diretório principal, utilizado o terminal iremos importar o pacote:

```
$ npm install --save three-react-obj-loader  
$ npm install --save three-react-mtl-loader
```

No diretório `/src/components/` iremos criar um componente chamado *ObjViewer*, usaremos como base o componente anterior, *Container3*, adicionando apenas o import para o pacote mencionado acima.

*ObjViewer.js*

```
import React, { Component } from 'react';  
import * as THREE from 'three';  
  
import OBJLoader from "three-react-obj-loader";  
import MTLLoader from 'three-react-mtl-loader';  
  
let renderer, scene, camera, mainObject;  
  
class ObjViewer extends Component {  
  constructor(props) {  
    super(props);  
  }  
  
  componentDidMount() {  
  
    scene = new THREE.Scene();  
    camera = new THREE.PerspectiveCamera(50, 500 / 400, 0.1, 1000);  
  
    renderer = new THREE.WebGLRenderer({ canvas: this.refs.threeCanvas, alpha: true, antialias: true});  
    renderer.setSize(500, 400);  
  
    camera.position.z = 10;
```

```

var mtlLoader = new MTLLoader();
mtlLoader.load("./models/heart.mtl", function (materials) {

    materials.preload();

    var objLoader = new OBJLoader();
    objLoader.setMaterials(materials);
    objLoader.load("./models/heart.obj", function (object) {
        mainObject = object;
        mainObject.position.y = -5;
        scene.add(mainObject);
    });
});

//luzes
var light = new THREE.DirectionalLight(0xffffffff, 1);
light.position.set(1, 1, 1).normalize();
scene.add(light);

this._render();
}

_render = () => {

    requestAnimationFrame(this._render);
    renderer.render(scene, camera);
    if(mainObject)
        mainObject.rotation.y += 0.01;

}

render() {
    return (
        <div>
            <canvas ref="threeCanvas">

```

```

        </canvas>
    </div>
);
}

}

export default ObjViewer;

```

O próximo passo é tornar esse objeto mais facilmente alterável por meio de props. Ou seja, se quisermos importar diferente objetos não será preciso alterar o script, apenas altera-se o parâmetro a ser passado, tendo uma chama similar a esta:

```

<ObjViewer
    objPath="./models/file.obj"
    mtlPath="./models/file.mtl"
    width={800}
    height={600}
/>

```

Ou de forma ainda mais simplificada, se passarmos um parâmetro apenas, como *obj*, o componente deve ser capaz de entender que tanto o .obj quanto o .mtl possuem mesmo nome e estão no mesmo diretório.

Para tal iremos alterar o script ObjViewer.js para inserir a leitura dos *props*:

```

import React, { Component } from 'react';
import * as THREE from 'three';

import OBJLoader from "three-react-obj-loader";
import MTLLoader from 'three-react-mtl-loader';

let renderer, scene, camera, mainObject;

class ObjViewer extends Component {
    constructor(props) {
        super(props);
    }
}

```

```

}

componentDidMount() {
  var { objPath, mtlPath, width, height, obj } = this.props;

  if (obj) {
    objPath = obj + ".obj";
    mtlPath = obj + ".mtl";
  }

  scene = new THREE.Scene();
  camera = new THREE.PerspectiveCamera(50, width / height, 0.1,
1000);

  renderer = new THREE.WebGLRenderer({ canvas: this.refs.threeCanvas, alpha: true, antialias: true });
  renderer.setSize(width, height);

  camera.position.z = 10;

  var mtlLoader = new MTLLoader();
  mtlLoader.load(mtlPath, function (materials) {

    materials.preload();

    var objLoader = new OBJLoader();
    objLoader.setMaterials(materials);
    objLoader.load(objPath, function (object) {
      mainObject = object;
      mainObject.position.y = -5;
      scene.add(mainObject);
    });
  });

  //luzes
}

```

```

var light = new THREE.DirectionalLight(0xffffff, 1);
light.position.set(1, 1, 1).normalize();
scene.add(light);

this._render();

}

_render = () => {

requestAnimationFrame(this._render);
renderer.render(scene, camera);
if (mainObject)
    mainObject.rotation.y += 0.01;

}

render() {
    return (
        <div>
            <canvas ref="threeCanvas">
            </canvas>
        </div>
    );
}

export default ObjViewer;

```

Utilizando este componente como base podemos implementar todas as interações apresentadas nos capítulos anteriores (anteriormente feitas utilizando three.js apenas) com three.js e react, utilizando módulos ES6.

## 7. React-Three-Renderer

Como dito anteriormente, NPM nos permite utilizar pacotes – códigos prontos – JavaScript. Ao decorrer deste material exploramos métodos de controlar cenas 3D com three.js utilizando renderização em canvas encapsulada em um Component React ao invés de utilizar diretamente um React Wrapper para Three.js. Fizemos isso por motivos didáticos, com maior conhecimento de como um renderizador funciona têm-se também mais controle dos objetos renderizados e de como interagir com estes. Neste capítulo final iremos explorar o React-Three-Renderer<sup>39</sup>, um React Wrapper do three.js.

React-Three-Renderer é opensource e pode ser instalado com o npm:

```
$ npm install --save react-three-renderer
```

Iremos então criar um componente chamado *RendererViewer.js* e nele importar o pacote mencionado.

*/example/7.React-three-renderer/app/RendererViewer.js*

```
import React3 from 'react-three-renderer';
```

Este pacote funciona de uma forma muito simples, teremos um *render* e uma cena declarados em tags - JSX - e os componentes declarados hierarquicamente abaixo (dentro) destas tags estarão contidos na cena.

Por exemplo, se quisermos desenhar um cubo na tela:

```
<React3>
  <scene>
    <camera/>
    <cube/>
  </scene>
</React3>
```

---

<sup>39</sup> <https://github.com/toxicFork/react-three-renderer>

React-three-renderer possui wraps para as geometrias, luzes e etc. do three.js. Mais exemplos podem ser encontrados no GitHub do projeto, na página de exemplos<sup>40</sup>.

No script abaixo temos uma implementação simples de uma cena. Para mais

```
import React, { Component } from 'react';
import React3 from 'react-three-renderer';
import * as THREE from 'three';

class RendererViewer extends Component {
  constructor(props) {
    super(props);

    this.cameraPosition = new THREE.Vector3(0, 0, 5);

  }

  render() {
    const width = window.innerWidth; // canvas width
    const height = window.innerHeight; // canvas height

    return (
      <React3
        mainCamera="camera" // this points to the perspectiveCamera
        which has the name set to "camera" below
        width={width}
        height={height}
      >

      <scene>
        <perspectiveCamera
          name="camera"
          fov={75}
          aspect={width / height}
          near={0.1}
        >
      
```

---

<sup>40</sup> <http://toxicfork.github.io/react-three-renderer-example/#/>

```

        far={1000}
        position={this.cameraPosition}
      />
      <mesh>
        <boxGeometry
          width={1}
          height={1}
          depth={1}
        />
        <meshBasicMaterial
          color={0x00ff00}
        />
      </mesh>
    </scene>
  </React3>);

}

export default RendererViewer;

```

React-three-renderer apresenta conceitos fáceis de compreendermos, por ser um wrapper do Three.js e termos explorado como essa biblioteca funciona, da instalação a interações avançadas. Projetos simples, como apenas rotacionar um objeto podem ser feitos utilizando o Wrapper em conjunto com pacotes como <sup>41</sup>(*OrbitControls.js* do three.js convertido para um modulo ES6) ou *three-transform-controls*<sup>42</sup> que é também uma classe ES6 de um script original do three.js.

---

<sup>41</sup> <https://github.com/mattdesl/three-orbit-controls>

<sup>42</sup> <https://www.npmjs.com/package/three-transform-controls>

Por fim, esperamos que este material – em conjunto com os conceitos apresentados em sala – possam auxiliar no desenvolvimento de visualizações de projetos, aumentando assim a capacidade do aluno de comunicação, alcance e análise de seus trabalhos. O repositório github vinculado a este material será mantido aberto e todos os alunos são convidados a participar e contribuir dest repositório<sup>43</sup>.

---

<sup>43</sup> [https://github.com/lucascassiano/Development\\_of\\_3D\\_Visualizations\\_examples](https://github.com/lucascassiano/Development_of_3D_Visualizations_examples)

## Referências

STEFANOV, S. **React: Up & Running: Building Web Applications.** Primeira Edição. 2016

BANKS, A. **Learning React: Functional Web Development with React and Redux.** Primeira Edição. 2017

MATSUDA, K. **WebGL Programming Guide: Interactive 3D Graphics Programming with WebGL (OpenGL).** Primeira Edição. 2013

PARISI, T. **Programming 3D Applications with HTML5 and WebGL: 3D Animation and Visualization for Web Pages.** 2014

DIRKSEN, Jos. **Learning Three.js: The JavaScript 3D Library for WebGL.** Segunda Edição.

FEDOSEJEV, A. **React.js Essentials.** Primeira Edição. 2015

Dirksen, J. **Three.js Cookbook.** 2015

# **Desenvolvimento de Visualizações 3D**

Node.js, React.js e Three.js para visualização de projetos

© Lucas Cassiano. Cambridge. Massachusetts. U.S - 2017