说明文档

V1.0 2016年7月1日

# 1.工程目录说明

## 1.1 主目录说明

App文件夹：项目用的是ExtJs，是一个MVC框架，因此网页被组织成了MVC结构，放在app文件夹中；

Js文件夹：大部分JavaScript逻辑代码都在这个文件夹中；

Lib文件夹：项目所引用的第三方Js库；

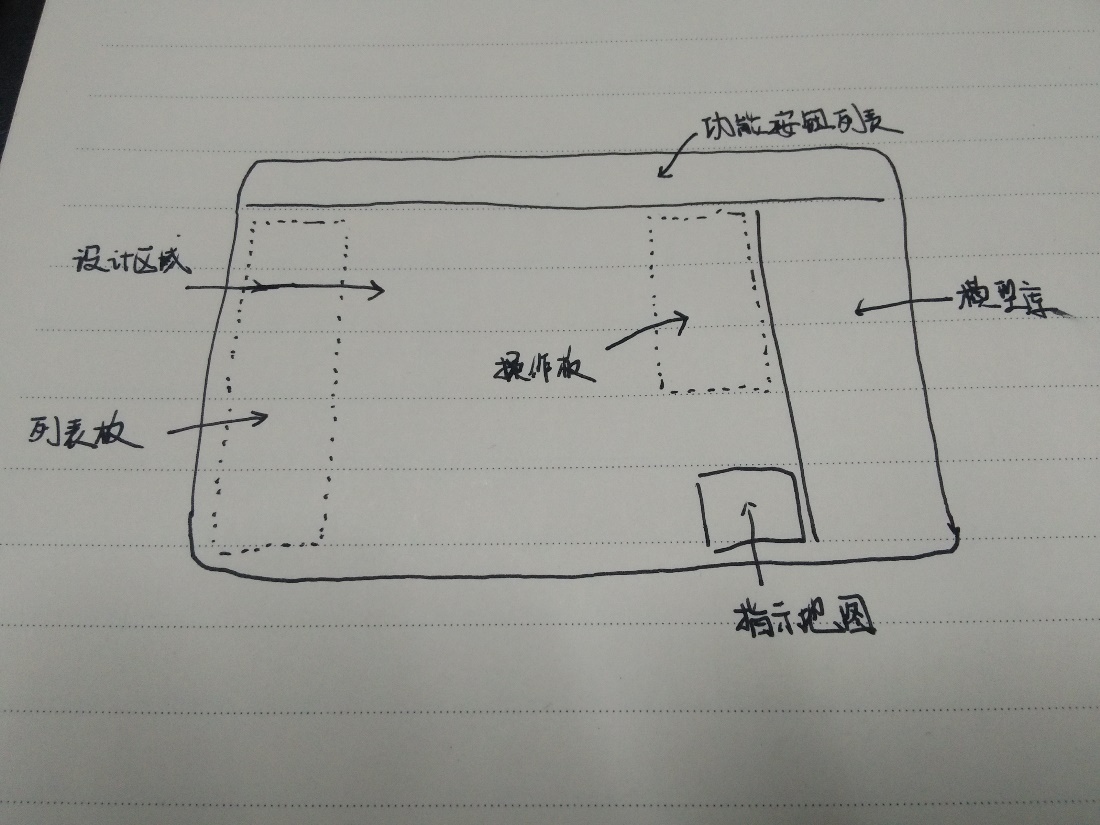
Page文件夹：注册及找回密码的html页面；

Resources文件夹：项目用到的所有图片、模型、视频等资源；

Server文件夹：服务端代码；

DesignPage.html：设计页面；

Index.html：登录引导页。



## 1.2 App文件夹说明

该文件夹由ExtJs自动生成，其中DesignApp.js作为MVC的入口，创建一个应用实例。该文件夹内的文件由于大部分是自动生成的，还是不要轻易改动为好。

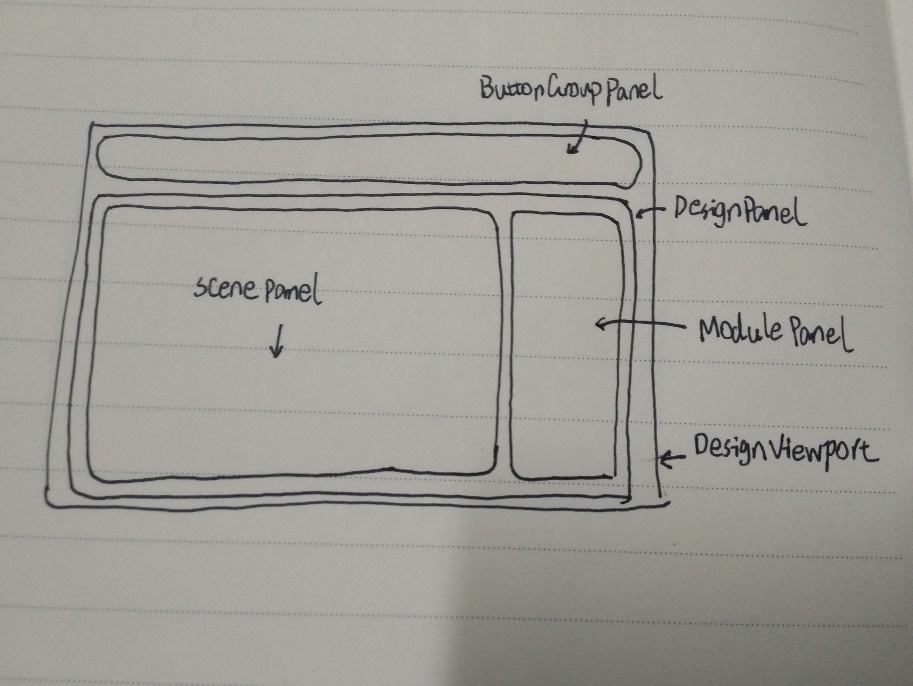
Controller文件夹：包含部分逻辑代码。

1. AdvancedController.js控制属性面板的响应；
2. ButtonController.js控制页面上面一排按钮的响应；
3. EnvironmentController.js这里面是整个页面逻辑的起点；
4. ModuleController.js对应右侧模型库的响应逻辑；

Model文件夹、Store文件夹：包含了模型库各类别模型的数据信息。

View文件夹：是整个页面各模块的视图层。

1. DesignViewport.js将页面分为上面一行按钮区域和下面一块区域，DesignPanel.js将页面下方区域又分为左侧设计场景和右侧模型库两个区域；ButtonGroupPanel.js是上面一行按钮；ScenePanel.js对应左侧设计区域；ModulePanel.js是右侧模型库区域，其中包含了BasicComponentGridPanel.js、ExpansionGridPanel.js分别对应基础模型列表和扩展列表。可以看到Ext若划分界面区域就得一层套一层
2. AdvancedControlWindow.js是属性控制面板的界面；
3. StructListWindow.js对应模型列表版；
4. VideoDemoWindow.js对应视频播放窗口；



## 1.3 Js文件夹说明

1. CooperationManage.js协同管理类，包含协同相关的逻辑；
2. Language.js对应中英文切换功能，里面包含中英文字符串；
3. Matrixs.js矩阵工具类，提供各种运算矩阵；
4. SceneManage.js场景管理类，包含与场景相关联的逻辑；
5. SceneMap.js对应右下角的小地图指示；
6. StorageManage.js存储管理类，包含场景整体导入导出逻辑；
7. Struct.js最基础的类，是整个场景运行的核心；
8. StructManage.js结构管理类，包含组件操作逻辑；
9. Structure.js结构类，是一个实体类，每一个组件对应一个对象；
10. Tool.js工具类，包含一些工具函数；
11. WindowManage.js窗口管理类，包含与弹窗相关联的逻辑；

## 1.4 Lib文件夹说明

Ext中包含ext框架的类库，不修改

Threejs库中，three.js是核心，BoxHelper.js提供组件高亮框，GridHelper.js提供场景辅助网格，MTLLoader.js与OBJMTLLoader.js提供obj模型文件的加载，OrbitControls.js提供三维场景的平移旋转缩放，TransformControls.js提供组件的平移旋转缩放。tween.min.js提供动画。

## 1.5 Resources文件夹说明

1. Css文件夹包含网页css样式文件；
2. Data文件夹包含模型库中模型的索引信息；
3. Iconsinuse文件夹中是页面上用到的小图标；
4. Images文件夹中是整个项目用到的图片；
5. Movie文件夹中存放有帮助视频；
6. Objmodel文件夹中放有obj&mtl格式的三维组件；

# 2.配置与部署

## 2.1 客户端配置

用webstorm打开即可。推荐使用谷歌浏览器访问。

## 2.2 服务端配置

1．安装java1.7；

2．使用idea打开server项目并如下配置；

Project中Libraries添加lib中的json，以及整个tomcat\lib下的文件

Artifacts的output directory输出到tomcat的webapps中（单独建立一个文件夹）

3．将项目放至Tomcat的webapps下，便同时启动客户端与服务端；

## 2.3 项目部署

修改客户端Struct.js文件中服务器地址为部署服务器地址；

将客户端与服务端输出放在tomcat的webapps下，启动tomcat即可；

将客户端文件夹名改为ROOT，访问即可不用输入工程名；

修改tomcat的配置文件端口为80，访问即可不用输入端口名。

# 3.功能实现原理

## 3.1 首页中英文切换

我们使用了一个js全局变量，放在Language.js中，点击中英文切换按钮就会替换这个全局变量，而项目中所有字符串都和这个全局变量关联了，于是实现中英文切换。

## 3.2 首页用户登录

登录模块使用的是LeanCloud的登录接口，代码逻辑在index.html中，具体细节参照LeanCloud的文档。

## 3.3 设计场景的平移、旋转与缩放

逻辑代码位于SceneManage.js中，调用了threejs库的OrbitControl.js。方法为使用如下代码：

orbitControl = new THREE.OrbitControls(camera, container);

将场景绑定，便实现了鼠标的平移旋转缩放功能。

OrbitControl提供了一个'change'监听事件，当鼠标移动了场景视角时就会触发，可以在里面实现一些刷新事件等。

## 3.4 模型库中模型的添加与替换

先说view层。模型库是一个容器，为ModulePanel.js。每一种类别的模型都是一个独立的view，集中放在modulepanel中。

对于每一种类别的模型，以BasicComponentGridPanel.js为例，这个view会关联模型图片的位置resources/images/basicLoad/以及view的store即BasicLoadStore，而store又会与model相对应。

BasicLoadStore会查询resources/data/BasicLoad.json中的数据，关联索引和名字。因此若想添加或替换模型，只要修改这个json文件即可。另外图片、obj和mtl文件应该与模型同名并放于resources中正确的文件夹。

## 3.5 模型库模型加载

双击模型库中一张图片，逻辑被ModuleController.js中响应。创建一个Structure对象并调用其初始化方法。注意模型加载完成是需要时间的，需要模型加载完成才能进行的逻辑要用回调函数的方法进行实现。

## 3.6 鼠标拾取模型

鼠标拾取模型是一个把鼠标二维坐标转换为场景三维坐标的过程。原理是鼠标点击页面上一点时，会从这一点向三维场景里发出一条射线，被射线击中的对象都会被捕捉到。

代码逻辑位于StructManage.js中的addMouseControl里。具体如下：

//射线法选中物体，注意各种传入的参数

var raycaster = new THREE.Raycaster();

var projector = new THREE.Projector();

var directionVector = new THREE.Vector3(((event.clientX - Struct.container.getBoundingClientRect().left) / Struct.container.width ) \* 2 - 1, -((event.clientY - Struct.container.getBoundingClientRect().top) / Struct.container.height ) \* 2 + 1, 1);

projector.unprojectVector(directionVector, Struct.camera);

raycaster.set(Struct.camera.position,directionVector.sub(Struct.camera.position).normalize());

然后调用下面这条语句，就能获得一个射线击中对象的列表。

raycaster.intersectObjects(Struct.scene.children, true));

遍历列表寻找被击中的模型。

## 3.7 组件功能总原理

场景里面每一个组件都是Structure类的一个实例对象，场景里的组件都存在Struct.js中的全局变量structures数组中。对组件的操作很多都是在修改Structure对象的属性，具体可以查看该类的属性。

注意Structure对象有一个object属性是保存模型实体的，我们的方法是将obj模型导入的实体作为孩子添加到object上。

组件又分独立个体与组合体两种。我们的方法是将每一个Structure对象的object统统挂到一个Structure对象的object上去作为组合体。

所有按钮的响应事件都在ButtonController.js中。

一定要熟悉Structures类的全部属性与方法。

## 3.8 新建场景

代码逻辑位于SceneManage.js中的resetScene函数中，共四步：

1. 重置视角；
2. 调用每个组件对象的删除函数；
3. 清空structures
4. 刷新渲染（每次场景有更改都要刷新场景）

## 3.9 导出场景

为了弹出等待窗口而不让页面卡住设置延迟。代码逻辑位于ButtonController.js中的btnExportClick函数中，导出的数据使用storageManage.exportObj获得。

为了了解exportObj的工作原理需要熟悉obj文件格式。导出数据时先把所有的组合体解散变成单一的组件，然后获得每一个组件的数据，完成后恢复组合体。而为了将所有数据合在一起需要设置偏移量来保证索引的顺序正确。

## 3.10 操作板与列表板

操作板的意思是使用键盘输入数据来操作组件。代码逻辑位于AdvancedController与windowManage文件中。

列表板的意思是将场景中所有组件信息都排列出来。代码逻辑位于WindowManage.js中的updateLW函数中，每次打开列表板都会调用。

## 3.11 平移与旋转与缩放

组件的平移旋转缩放使用的是第三方库TransformControl，代码位于StructManage中的addTransformControl函数，创建对象并绑定相机等就可以实现操作轴效果。

这里有两个设置，一是为了防止组件移动越界，设置了监听函数保证越界后的纠正。

二是拖动轴会和旋转场景这个动作重合，为了在拖动轴时不旋转场景，设置了一个标记transformFlag，并修改了OrbitControls库，旋转场景动作时先检查标记，若此刻在拖动轴则屏蔽旋转动作。

## 3.12 组合与分离

组合与分离的逻辑位于StructManage.js中的combinaOperate与combineByID函数中，利用Structure对象的initByCombina方法。会使用到全局坐标与局部坐标的相互转换。

## 3.13 对齐

对齐功能的逻辑位于StructManage.js中的alignmentX、alignmentY、alignmentZ三个函数中，原理是统计所有组件的位置，计算中心位置，然后移动所有组件到中心位置上去。

## 3.14 选框

代码逻辑位于StructManage.js中的鼠标事件以及SceneManage.js中的selectBoxControl函数中。选框是一个浮现在页面上的div，带有边框。选框的原理与拾取的原理正好相反，将场景中每个组件的三维位置换成二维坐标，然后查找这个二维坐标是否在选框中。

## 3.15 复制

复制逻辑位于StructManage.js的structCopy函数中。注意复制组合体时需要先复制出组合体中的每一个个体，然后再把他们组合。组合需要等待模型全部加载完成才能进行，因此需要使用回调函数。

凡是涉及模型加载的都要注意异步的问题。

## 3.16 初始

初始功能用于将场景恢复成初始视角，代码位于SceneManage.js的initView函数中。除了调整相机视角外还要调整orbitControl对象，不然场景的平移旋转会出现问题。

## 3.17 演示

演示功能弹出一个视频窗口播放演示视频。代码位于VideoDemoWindow.js中，注意不同的浏览器支持不同的视频播放格式。

# 4.特色功能介绍

## 4.1 吸附

吸附功能的操作是先后点击两个组件的两个面，然后这两个面会粘贴到一起。我们重新定义问题，即如何通过旋转移动组件，使得两个组件的面吸附到一起，重点在于求解旋转移动组件的矩阵。

通过Threejs，我们能够获得先后点击的两个面的两个点击点的信息，包括位置与法向量，最终当面重合的时候这两个点应该是重合的，而且法向量反向平行。

重合点容易，问题在于解决法向量反向平行。于是问题转换为现在有两个向量，求一个矩阵，使得一个向量通过矩阵变换后与另外一个向量反向平行。

已知旋转前后的向量的变化，如何求这个旋转矩阵的方法如下：

**1.旋转角度**

已知旋转前向量为P, 旋转后变为Q。由点积定义可知：

[image](http://images.cnitblog.com/blog/477176/201302/15164310-79970af70dc0424ea1b7385196b07875.png)

可推出P，Q之间的夹角为：

[image](http://images.cnitblog.com/blog/477176/201302/15164314-d6a2f58389254552b4c5c94132a3d610.png)

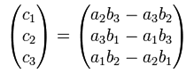
**2. 旋转轴**

由1中可知，旋转角所在的平面为有P和Q所构成的平面，那么旋转轴必垂直该平面。

假定旋转前向量为**a**(a1, a2, a3)， 旋转后向量为**b**（b1, b2, b3)。由叉乘定义得：

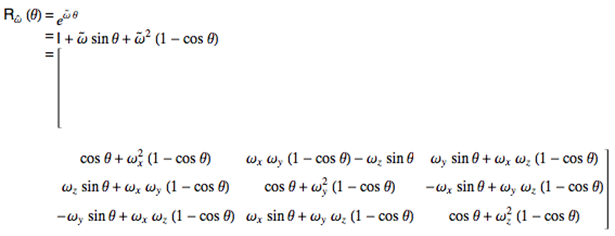
[image](http://images.cnitblog.com/blog/477176/201302/15164319-2ab7fcc6ad2d45a798e3fac73be1d0a1.png)

所以旋转轴**c**(c1, c2, c3)为：

[](http://images.cnitblog.com/blog/477176/201302/15164322-c8ef412c78614a42bb1bedbdb64bb89b.png)

**3.  罗德里格旋转公式(Rodrigues' rotation formula)**

已知单位向量[Inline3](http://images.cnitblog.com/blog/477176/201302/15164324-7a2cd186fda44852bff992caed574b74.gif) ， 将它旋转θ角。由罗德里格旋转公式，可知对应的旋转矩阵[Inline5](http://images.cnitblog.com/blog/477176/201302/15164326-1211440157be4633918d2e17d8a571f6.gif) ：

[](http://images.cnitblog.com/blog/477176/201302/15164342-44d9364dcbf740b0bd2605307ce3a3ad.png)

其中I是3x3的单位矩阵，

omega^~ 是叉乘中的反对称矩阵r：

[NumberedEquation1](http://images.cnitblog.com/blog/477176/201302/15164357-8ed8312819634b6a93b1ca8fbbda2d52.gif)

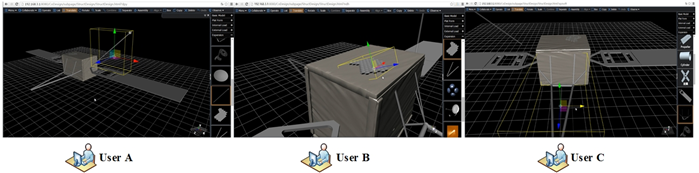
**4. 求旋转矩阵**

根据旋转前后的两个向量值，使用上面的方法，先求出旋转角度和旋转轴，然后用罗德里格旋转公式即可求出对应的旋转矩阵。

旋转矩阵的代码位于Matrixs.js中，吸附逻辑代码位于StructManage.js的assemblyAction函数中，使用状态标记来对应吸附的不同状态。

## 4.2 协同

协同功能是多人能够同时进行拼装，而且实时看到相同的画面。如下图所示：



为了实现多人设计工作的实时同步，让每个用户工作空间中的内容始终保持一致，我们采用如下的方法实现数据同步：当某一用户进行了某一操作后，其他用户端都“模仿”相同的操作。具体可表述为如下过程：第一步，浏览器端对用户的操作进行立即响应，修改本地数据与更新画面显示，并将这一操作转化成指令发送给服务器；第二步，服务器接收指令进行处理，然后发送给其他用户的浏览器端；第三步，其他浏览器端接收到指令后使用指令中的数据进行“模仿”，在本地执行同样的操作。任一用户的任一操作都会在所有用户的工作空间中得到响应，如此便可达到实时同步的目的，同时用户的操作会得到即时的反馈，保证了用户体验。

在上述同步过程中，需要将设计过程中的各种用户操作和操作参数封装成操作指令，称之为“指令化”。我们用字符串拼接的方式把各参数字段连接成为一个长字符串，由发送方序列化为json格式进行传输，由接收端负责解析处理。除了能直接支持基本的增删改等三维组件操作之外，通过组合使用这些基础指令，也能对复杂的三维操作进行同步。例如某用户对一个多组件的组合体进行复制时，可以先使用多条ADD指令在其他用户端创建出子组件，然后通过COM指令将子组件进行组合，最后利用MOD指令对组合完成的组合体进行位置转向等三维参数的同步。

系统支持用户独立进行设计工作，也允许用户在设计的任何阶段开放协同环境以邀请其他用户加入到自己的工作中。开放协同环境的用户我们称之为Host，其他参与用户称之为Participant。开放的用户生成一个id，然后其他用户输入id参加进去。

虽然所有用户的工作空间始终保持一致，但每个用户是拥有自己独立的任意的观察视角的，画面的显示由各浏览器利用工作空间数据自行绘制，并不需要服务器进行计算。服务器主要作为“信使”，与所有浏览器端频繁地通信来保证同步过程。我们借助于HTML5中的WebSocket特性，能提供更加稳定有效的通信支持。在协同过程中我们使用简单的对三维组件加锁的方法来避免多用户的操作冲突。但是要注意处理中途用户的加入、中途用户的离开、锁的先后冲突等问题。

与协同直接相关的逻辑代码位于CooperationManage.js中，其他代码分散在各种操作之中，每次完成一个操作都会发送协同消息，这从可维护性角度来说是不好的。

另外WebSocket能发送的数据量有限制，当数据量太大时需要分片发送。