如何编写FaceUnity道具脚本

基本结构

每个FaceUnity道具包的核心是文件名为`script.js`的Javascript脚本。`script.js`会在加载道具时得到执行，然后返回一个道具对象。而不同的视觉效果需要通过重载这个道具对象的不同方法进行实现。

`script.js`的基本结构是这样的：

```
(function(){
 //初始化代码
 return {
  SetParam:function(name,value){/*...*/},
  GetParam:function(name){/*...*/},
  FilterImage:function(params){/*...*/},
  Render:function(params){/*...*/},
  name:"道具名称",
 };
})()
```

可以重载的方法一共有四个：

- `SetParam`用于提供`fuItemSetParam`系列函数的内部实现，外面每次调用`fuItemSetParam`都会传到道具里面的`SetParam`。这个接口是可选的，通常只是在道具编辑器的模板中用来支持用户界面，不需要进行改动。有特殊需要的时候可以用来实现触控之类的实时交互。
- `GetParam`和`SetParam`类似，用于提供`fuItemGetParam`系列函数的内部实现，一般不需要改。
- `FilterImage`是可选的，用于实现作用在摄像头图像上的滤镜效果。
- `Render`是必须重载的，用于实现叠加在摄像头图像上面的所有其他绘制效果。

可以重载的成员只有一个，就是道具名称`name`。请务必为每一个道具设置一个**独有**的名称，并将公司名加在前面，以方便日后管理。比方说，如果Tiki公司做了一个猫耳道具，可以叫做`"tiki_maoer"`。日后又做了另一个颜色不同的猫耳道具的话，就要改成`"tiki_maoer_v2"`之类。

我们使用的Javascript解释器duktape符合ECMAScript 5标准。在语法、标准库和编程技巧方面建议查查做网页用的教程。调试的时候想要输出信息可以跟浏览器里一样用`console.log`，但是其余的HTML接口不会提供。参考链接：

- http://duktape.org/
- http://www.w3schools.com/js/

Javascript中未处理的异常会导致`libnama`崩溃，所以重载的函数中请务必加上`try...catch`，像这样：

```
(function(){
 //……
 return {
  //……
  Render:function(params){
   try{
    //实际代码
   }catch(err){
    console.log(err.stack);
   }
  },
  //……
 };
})()
```

初始化

初始化代码是用来载入道具包中其他文件的唯一地点。有三个内部接口用来载入不同类型的资源：

- `FaceUnity.ReadFromCurrentItem(file_name)`用于读取文本文件，会将整个文件的内容作为字符串返回
- `FaceUnity.LoadBlendshape("avatar.json","avatar.bin")`用于读取三维模型，会解析两个输入文件返回一个模型对象
- `FaceUnity.LoadTexture(file_name)`用于读取贴图文件，会返回一个贴图对象。图片格式支持png、jpg、tif、tga、webp。

这里FaceUnity是一个提供了各种libnama内部接口的全局对象。后面的接口基本也都是在这里提供的。资源载入的具体案例会在后面具体效果的地方具体介绍。

获取人脸数据

FilterImage和Render两个接口接收的params对象是一样的，都是由核心库跟踪得到的各种人脸数据以及一些相关的辅助信息。具体的参数会在具体用到的时候具体介绍。想要研究具体数值的话，可以在Render接口里写个console.log(JSON.stringify(params, null, 2));查看。现在版本中params主要有以下几类成员：

人脸数据：

- translation：3个元素的数组，内容是表示人脸平移变换的三维向量
- rotation：4个元素的数组，内容是表示人脸三维旋转变换的四元数
- expression：46个元素的数组，内容是各个表情的blendshape系数。每个表情具体是什么请参照blendshape制作教程：https://beta.modelo.io/projects/57d223f4df427650465ad6cb/folder/57d223f5df427650465ad6cc
- identity：75个元素的数组，内部使用，用来重建用户的人脸模型
- focal_length：1个元素的数组，内容是摄像头的焦距（单位是像素）
- landmarks：75×2=150个元素的数组，内容是所有人脸特征点在图像上的二维坐标。特征点编号参见landmarks.jpg。
- landmarks_ar：75×3=225个元素的数组，内容是所有人脸特征点的三维空间坐标。
- pupil_pos：2个元素的数组，内容是两眼的平均瞳孔位置，单位没有物理意义，一般用来定性地控制三维眼球旋转

辅助信息：

- frame_id：当前帧号，是一个从0开始不断增加的整数
- w：摄像头图像宽度
- h：摄像头图像高度
- mat_face_camera_rotation：4个元素的数组，内容是用来把摄像头图像转成用户头顶朝上的二维旋转矩阵
- tracker_space_w：把摄像头图像转成用户头顶朝上之后的图像宽度
- tracker_space_h：把摄像头图像转成用户头顶朝上之后的图像高度

多人模式的识别信息：

- face_identifier：在多人模式下用于识别哪个脸是上一帧的哪个脸的字符串。同一个脸的字符串相等
- face_ord：多人模式下的人脸编号

面具效果

FaceUnity.RenderAR(texture)接口用于渲染面具。只要传入一个符合标准UV布局的贴图对象，就可以渲染一个贴图贴在脸上。贴合效果要高于二维贴纸和三维道具。

下面是一个渲染面具的示例脚本：

```
(function(){
 var tex_mask=FaceUnity.LoadTexture("mask.png");
 return {
  Render:function(params){
   try{
    FaceUnity.RenderAR(tex_mask);
   }catch(err){
    console.log(err.stack);
   }
  },
  name:"mask_example",
 };
})()
```

todo: attach the reference texture

二维贴纸

FaceUnity.RenderBillboard接口用于渲染二维贴纸。这个函数的参数比较多，用起来比较复杂，一般都是和二维贴纸的打包工

具配合使用。具体的调用方法是：

```
FaceUnity.RenderBillboard(texture, board_description, frame_description);
```

其中texture是存放二维贴纸内容的贴图对象。board_description是一个存放贴纸全局信息的对象，其一般形式为：

```
var board_description={
 type: "billboard_ex2",
 base_vertex: 贴纸绑定在标准人脸模型上顶点号，-1代表不绑定，
 blendfunc_src: gl.blendFunc的src参数（可选），
 blendfunc_tar: gl.blendFunc的tar参数（可选），
 enable_depth_test: 设为1代表对二维贴纸启用深度检测，处理和三维道具之间的遮挡（可选），
 shader: 用来实现特效的shader函数（可选），
 uniforms: 给shader提供的常量值（可选），
};
```

frame_description是一个存放贴纸动画帧信息的对象，其一般形式为：

```
var frame_description={
 v: [4个顶点的三维位置，一共12个数],
 vt [4个顶点的二维纹理坐标，一共8个数],
};
```

一般来说，二维贴纸道具都是通过工具打包的。工具会把贴纸的动画和贴图数据分别放在desc.json和bigtex.png里面。这种情况下Render函数一般会这么写：

```
(function(){
 var boards=JSON.parse(FaceUnity.ReadFromCurrentItem("desc.json"));
 var tex=FaceUnity.LoadTexture("bigtex.png");
 //……
 return {
  //……
  Render:function(params){
   try{
    var frame_id=Math.floor(params.frame_id*播放速度倍数);
    for(var i=0;i<boards.length;i++){
     FaceUnity.RenderBillboard(tex,boards[i],boards[i].texture_frames[(frame_id)%boards[i].texture_frames.length]);
    }
   }catch(err){
    console.log(err.stack);
   }
  },
  //……
 };
})()
```

其中播放速度倍数越大越快，越小越慢，对于一般的手绘动画可以用0.5。

FaceUnity.RenderBillboard的另一个用途是绘制全屏背景图或者全屏特效。这种情况下一般这么写：

```
(function(){
 //……
 var tex_bg=FaceUnity.LoadTexture("bg.jpg");
 var bg_board={
  type:"billboard_ex2",
  base_vertex:-1,
  //对于全屏特效，可以在这里加上shader、uniforms等其他参数
 };
 //……
 return {
  //……
  Render:function(params){
   try{
    //……
    //计算一个刚好覆盖整个屏幕的三维贴纸位置，并且从贴图上抠出来一块合适的部位
    var scale=(20000/params.focal_length);
```

```
    var scale_tex=Math.min(tex_bg.w/params.tracker_space_w,tex_bg.h/params.tracker_space_h);
    var w_tex=(scale_tex*params.tracker_space_w)/tex_bg.w;
    var h_tex=(scale_tex*params.tracker_space_h)/tex_bg.h;
    var x_tex=(1.0-w_tex)*0.5;
    var y_tex=(1.0-h_tex)*0.5;
    FaceUnity.RenderBillboard(tex_bg,bg_board,{
     v:[scale*params.tracker_space_w/2,-scale*params.tracker_space_h/2,scale*params.focal_length,
      -scale*params.tracker_space_w/2,-scale*params.tracker_space_h/2,scale*params.focal_length,
      -scale*params.tracker_space_w/2,scale*params.tracker_space_h/2,scale*params.focal_length,
      scale*params.tracker_space_w/2,scale*params.tracker_space_h/2,scale*params.focal_length],
     vt:[
      x_tex+w_tex,y_tex,
      x_tex,y_tex,
      x_tex,y_tex+h_tex,
      x_tex+w_tex,y_tex+h_tex],
    });
    //……
   }catch(err){
    console.log(err.stack);
   }
  },
  //……
 };
})()
```

如果希望自己写shader的话，就需要提供`shader`和`uniforms`两个成员。`board_shader_example/`下面提供了一个例子，可以用编辑器中的"Load template"功能加载查看。例子中的原特效来自https://www.shadertoy.com/view/Xtf3zn，根据原网站要求，请勿用作商用。

这里注意到，shader中的`uniform`变量不需要手工定义。在设置了`uniforms`的成员之后系统会自动生成那部分的定义代码。另外，系统对shader的接口进行了封装，用户需要提供的是一个`shader_main`函数：

```
vec4 shader_main(vec4 C0){
 vec4 fragColor;
 mainImage( fragColor, vec2(st_frag.x,1.0-st_frag.y)*iResolution);
 return fragColor;
}
```

传入一个`vec4`的贴图颜色，传出一个`vec4`的最终颜色。shader里面可以通过`st_frag`来访问纹理坐标。

todo: attach the tool

三维道具

三维道具的绘制有三个步骤，首先在初始化时载入编辑器打包好的模型信息：

```
var blendshape=FaceUnity.LoadBlendshape("avatar.json","avatar.bin");
```

注意虽然函数名字里有`Blendshape`字样，但是普通的静态道具其实走的也是这条路线。载入好的`blendshape`是一个JS对象，基本就是`avatar.json`的内容解析之后的结果。在程序中需要访问的主要是其中的`blendshape.drawcalls`成员。该成员是一个数组，其中的每个元素代表一个材质。受限于OpenGL ES2的特性，每个材质需要单独调用一次绘制函数。

在实际绘制模型之前，需要先处理一下blendshape变形并把几何数据上传到GPU：

```
FaceUnity.ComputeBlendshapeGeometry(blendshape,params);
```

还是注意，虽然函数名字里有`Blendshape`字样，但是普通的静态道具也要调用一下这个函数完成数据上传。参数中的`blendshape`是刚才加载的`blendshape`，而`params`就是`Render`函数参数中的那个`params`。

最后就是实际绘制每个材质：

```
FaceUnity.RenderBlendshapeComponent(blendshape,drawcall,s_vert_shader,s_frag_shader,uniforms);
```

`blendshape`就不再多加解释。`drawcall`是`blendshape.drawcalls`的元素。`s_vert_shader`是vertex shader的代

码，s_frag_shader是fragment shader的代码，uniforms还是shader里的uniform变量值。

RenderBlendshapeComponent和shader之间传递模型数据的接口是这样的：

```
attribute vec3 P; //顶点坐标
attribute vec3 N; //顶点法向
attribute vec2 st; //顶点纹理坐标
attribute vec3 dPds; //顶点坐标P关于纹理坐标s的导数，是做normal map用的第一个切向
attribute vec3 dPdt; //顶点坐标P关于纹理坐标t的导数，是做normal map用的第二个切向
```

系统另外提供四个函数计算一些必要的矩阵：

- FaceUnity.CreateProjectionMatrix()用来计算人脸数据所在坐标系向摄像头图像空间的投影矩阵
- FaceUnity.CreateViewMatrix(rotation,translation)用来计算把重建的用户人脸模型正确放置到三维空间中的刚性变换矩阵
- FaceUnity.CreateEyeMatrix(center,pupil_pos)用来计算眼球专用的旋转矩阵。center是一个有3个元素的数组，里面是旋转中心的三维坐标。pupil_pos就是params.pupil_pos。可以通过在pupil_pos上乘一个倍率来改变眼球的旋转幅度。
- FaceUnity.MatrixMul(a,b)是普通的矩阵乘法。

最后script.js可以访问一个叫做gl的全局对象，里面暴露了一套完整的WebGL 1.0接口。可以借助WebGL来改变调用FaceUnity内置函数时的渲染流水线状态。

关于三维道具绘制的具体例子可以参考编辑器自带的einstein模板中的js和glsl文件。为了符合ECMAScript标准，注释都是UTF-8的，要用合适的软件打开。没有的话，我们也可以提供一个。

滤镜效果

滤镜效果主要是通过在FilterImage接口中调用函数FaceUnity.InsertImageFilter(type,shader,uniforms)实现的。其中type有"warp"和"color"两种选择，分别对应于变形滤镜和颜色滤镜。和2D贴纸类似，shader是shader代码，uniforms是shader中的常量值。根据type的不同，shader的接口也会不同：

```
// warp类型的变形滤镜
vec2 shader_main(vec2 st){
 /*
 输入原始纹理坐标st。
 返回一个vec2，表示变形后的纹理坐标。
 后续的图像操作都会按变形后坐标来读取摄像头图像。
 */
}
```

```
// color类型的颜色滤镜
vec4 shader_main(vec2 st,vec4 C){
 /*
 输入原始纹理坐标st和原始颜色C。
 返回一个vec4，表示处理之后的颜色。
 st可以用来读取其他的自定义纹理。
 */
}
```

滤镜的例子可以参照提供的green_pig道具。

美颜和道具的相互作用

整个系统绘制的先后顺序是：

- 输入摄像头图像
- 磨皮
- 调用所有道具的FilterImage函数，将InsertImageFilter插入的所有滤镜和系统自带的美颜滤镜整合到一个shader里面
- 执行美颜的变形滤镜或者道具的变形滤镜
- 执行美颜的颜色滤镜
- 执行道具的颜色滤镜
- 调用所有道具的Render函数，绘制贴纸和三维模型

这里要注意几点：

- 如果设置了变形滤镜。美颜中的瘦脸等变形效果将会自动禁用。
- 出于效率上的考虑，变形滤镜和颜色滤镜之间的作用顺序永远是变形在前，颜色在后，不管`InsertImageFilter`的时候顺序如何。
- 同类滤镜的作用顺序以`InsertImageFilter`的调用顺序为准。