浙江大学

课程名称：计算机动画

姓　　名：刘佳润

学　　院：计算机科学与技术学院

专　　业：数字媒体技术

学　　号：3180105640

指导教师：于金辉

2020　年　10　月　22　日

浙江大学实验报告

课程名称：计算机动画 实验类型：综合

实验项目名称：Free-Form-Deformation 的实现

学生姓名：刘佳润 专业：数字媒体技术 学号：3180105640

同组学生姓名：无 指导老师：于金辉

实验地点： 实验日期：2020 年 10 月 22 日

一、实验目的和要求

- 掌握 FFD 变形算法的思想与实现原理。
- 能够实现交互控制进行 FFD 三维变形。

二、实验内容和原理

根据 FFD 的原理，将某个三维模型嵌入一个可调整 box 内，通过交互操作嵌入空间的控制点，对嵌入其中的物体进行变形操作。要求控制点数量可以变化。

数学原理分析：

自由变形技术 Free-Form Deformation 是编辑几何模型的重要手段，它于 80 年代由 Siedberg 等人提出，目前许多三维建模软件中都有这种变形算法。自由变形方法在变形过程中并不是直接操作几何模型，而是把几何模型嵌入到变形空间，然后通过操作变形空间来使得

嵌入其中的几何模型发生变形。

首先创建一个平行六面体的变形空间框架（Box），并将目标几何体嵌入该框架，建立世界坐标与局部坐标（相对坐标）的联系。公式如下：

$$s=\frac{\mathbf{T}\times\mathbf{U}\cdot(\mathbf{X}-\mathbf{X}_0)}{\mathbf{T}\times\mathbf{U}\cdot\mathbf{S}},\ t=\frac{\mathbf{S}\times\mathbf{U}\cdot(\mathbf{X}-\mathbf{X}_0)}{\mathbf{S}\times\mathbf{U}\cdot\mathbf{T}},\ u=\frac{\mathbf{S}\times\mathbf{T}\cdot(\mathbf{X}-\mathbf{X}_0)}{\mathbf{S}\times\mathbf{T}\cdot\mathbf{U}}$$

其中，**S**、**T**、**U** 是变形框架的三个方向向量。我们得出的局部坐标都是归一化了的，即 0<s<1，0<t<1，0<u<1。在整个变形的过程中，几何模型的顶点的局部坐标是不变的，只有世界坐标相应变化。

下面引入框架控制点 Pijk：

$$\mathbf{P}_{ijk}=\mathbf{X}_0+\frac{i}{l}\mathbf{S}+\frac{j}{m}\mathbf{T}+\frac{k}{m}\mathbf{U}.$$

$$X=X0+sS+tT+uU$$

移动框架的控制点，利用几何模型顶点局部坐标、控制点世界坐标和 Bernstein 多项式重新计算新的顶点世界坐标：

$$\mathbf{X}_{ffd}=\sum_{i=0}^{l}\binom{l}{i}(1-s)^{l-i}s^{i}\Big[\sum_{j=0}^{m}\binom{m}{j}(1-t)^{m-j}t^{j}\Big[\sum_{k=0}^{n}\binom{n}{k}(1-u)^{n-k}u^{k}\mathbf{P}_{ijk}\Big]\Big]$$

其中 P(i, j, k)为框架控制点的新坐标，l、m、n 分别为在 S、T、U 坐标轴上划分的格子数目。上式也可以简单地写作：

$$\sum_{i=0}^{3}\sum_{j=0}^{3}\sum_{k=0}^{3}p_{ijk}B_i(u)B_j(v)B_k(w)$$

其中 Bi(u)是 Bernstein 基函数，定义为：

$$B_i^n(u)=C_n^i\,u^i(1-u)^{n-i}\quad(i=0,1,\ldots n)$$

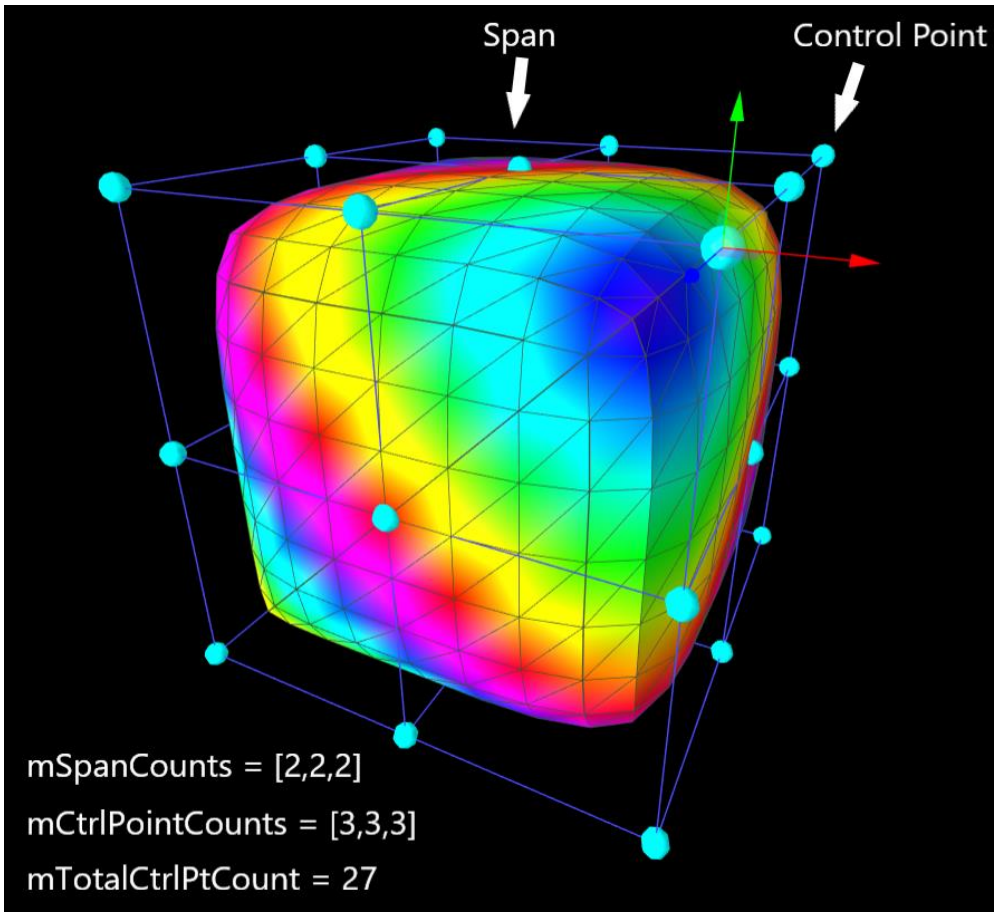
三、实验器材

HTML5 + CSS+ JavaScript、 Three.js 三维库

工程依托 Visual Studio Code 完成

四、实验步骤

0. 基本数据结构定义




```
// 嵌入空间
var mBBox = new THREE.Box3();

// 每个方向的分割点个数（尺寸）
var mSpanCounts = [0, 0, 0];

// 每个方向控制点个数
var mCtrlPtCounts = [0, 0, 0];

// 控制点总个数
var mTotalCtrlPtCount = 0;

// 每条边上的矢量
var mEdge = [new THREE.Vector3(0, 0, 0), new THREE.Vector3(0, 0, 0), new THREE.Vector3(0, 0, 0)];

// 控制点的坐标
var mCtrlPts = [];
```

1. 构建嵌入空间，建立世界坐标与局部坐标的联系

在 Three.js 里，构建嵌入空间非常简单，调用 box 对象的 setFromPoints 方法就可以实现：setFromPoints(points: Vector3[])，设置此包围盒的上边界和下边界，以包含数组 points 中的所有点。根据公式：

$$s=\frac{\mathbf{T}\times\mathbf{U}\cdot(\mathbf{X}-\mathbf{X}_0)}{\mathbf{T}\times\mathbf{U}\cdot\mathbf{S}},\ t=\frac{\mathbf{S}\times\mathbf{U}\cdot(\mathbf{X}-\mathbf{X}_0)}{\mathbf{S}\times\mathbf{U}\cdot\mathbf{T}},\ u=\frac{\mathbf{S}\times\mathbf{T}\cdot(\mathbf{X}-\mathbf{X}_0)}{\mathbf{S}\times\mathbf{T}\cdot\mathbf{U}}$$

实现方法如下：

```
// 计算相对坐标（局部坐标）
this.world2local = function(world_pt) {
    // 构建向量：从框定空间的最小点指向目标世界坐标
    var vec = new THREE.Vector3(world_pt.x, world_pt.y, world_pt.z);
    vec.sub(mBBox.min);
    // 叉积
    var cross = [new THREE.Vector3(), new THREE.Vector3(), new THREE.Vector3()];
    cross[0].crossVectors(mEdge[1], mEdge[2]);
    cross[1].crossVectors(mEdge[0], mEdge[2]);
    cross[2].crossVectors(mEdge[0], mEdge[1]);
    // 局部坐标
    var local_pt = new THREE.Vector3();
    for (var i = 0; i < 3; i++) {
        local_pt.setComponent(i, cross[i].dot(vec) / cross[i].dot(mEdge[i]));
    }
    return local_pt;
};
```

2. 构造控制点、估计点

预期效果是在包围盒上画出线框与控制点。所以写了如下绘制线框的代码：

```
// 重建
mBBox = bbox;
mSpanCounts = span_counts;
mCtrlPtCounts = [mSpanCounts[0] + 1, mSpanCounts[1] + 1, mSpanCounts[2] + 1];
mTotalCtrlPtCount = mCtrlPtCounts[0] * mCtrlPtCounts[1] * mCtrlPtCounts[2];

// 设定s,t,u坐标空间
mEdge[0].x = mBBox.max.x - mBBox.min.x;
mEdge[1].y = mBBox.max.y - mBBox.min.y;
mEdge[2].z = mBBox.max.z - mBBox.min.z;

// 重设控制点
mCtrlPts = new Array(mTotalCtrlPtCount);

// 设置每个控制点的位置
for (var i = 0; i < mCtrlPtCounts[0]; i++) {
    for (var j = 0; j < mCtrlPtCounts[1]; j++) {
        for (var k = 0; k < mCtrlPtCounts[2]; k++) {
            var position = new THREE.Vector3(
                mBBox.min.x + (i / mSpanCounts[0]) * mEdge[0].x,
                mBBox.min.y + (j / mSpanCounts[1]) * mEdge[1].y,
                mBBox.min.z + (k / mSpanCounts[2]) * mEdge[2].z
            );
            this.setPositionTernary(i, j, k, position);
        }
    }
}
```

其中 mEdge 变量是在每个方向上建立的单位向量。setPositionTernary 函数通过三元坐标设置控制点。

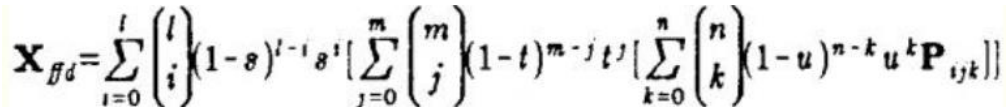
3. 变形后几何体坐标的得到

Berstein 基函数的实现：

```
// 计算阶乘
function fact(n) {
    var res = 1;
    for (var i = n; i > 0; i--)
        res *= i;
    return res;
};

// 生成Bernstein基函数Bi,n(u)
function bernstein(n, i, u) {
    var coeff = fact(n) / (fact(i) * fact(n - i));
    return coeff * Math.pow(u, i) * Math.pow(1 - u, n - i);
};
```

首先根据公式：



得到在局部坐标（即相对坐标）系下估计的点的坐标。

```
// 获取(s,t,u)空间的估计点—变化后坐标
this.evalLocal = function(s, t, u) {
    var eval_pt = new THREE.Vector3(0, 0, 0);
    for (var i = 0; i < mCtrlPtCounts[0]; i++) {
        var point1 = new THREE.Vector3(0, 0, 0);
        for (var j = 0; j < mCtrlPtCounts[1]; j++) {
            var point2 = new THREE.Vector3(0, 0, 0);
            for (var k = 0; k < mCtrlPtCounts[2]; k++) {
                // 计算矢量和berstein基函数
                var position = this.getPositionTernary(i, j, k);
                var poly_u = bernstein(mSpanCounts[2], k, u);
                // 将所传入的矢量与标量相乘所得的乘积和这个向量相加
                point2.addScaledVector(position, poly_u);
            }
            var poly_t = bernstein(mSpanCounts[1], j, t);
            point1.addScaledVector(point2, poly_t);
        }
        var poly_s = bernstein(mSpanCounts[0], i, s);
        eval_pt.addScaledVector(point1, poly_s);
    }
    return eval_pt;
};
```

然后可以转换坐标系到世界坐标：

```
// FFD得到变形后的世界坐标（绝对坐标）
FFD.prototype.evalWorld = function(world_pt) {
    var local = this.world2local(world_pt);
    return this.evalLocal(local.x, local.y, local.z);
}
```

eval_pt（估计点）穿插在几何形体顶点之间，定义如下：

```
var eval_pt_spans = new THREE.Vector3(16, 16, 16);
var eval_pt_counts = new THREE.Vector3(
                                    eval_pt_spans.x + 1,
                                    eval_pt_spans.y + 1,
                                    eval_pt_spans.z + 1);
var eval_pts_geom = new THREE.Geometry();
var eval_pts_mesh;
```

在变形过程中的代码如下：

```
// 更新几何体顶点
for (i = 0; i < geom.vertices.length; i++) {
    // 计算变形后顶点
    var eval_pt = ffd.evalWorld(vert[i]);
    if (eval_pt.equals(geom.vertices[i]))
        continue;
    geom.vertices[i].copy(eval_pt);
}
geom.verticesNeedUpdate = true;
```

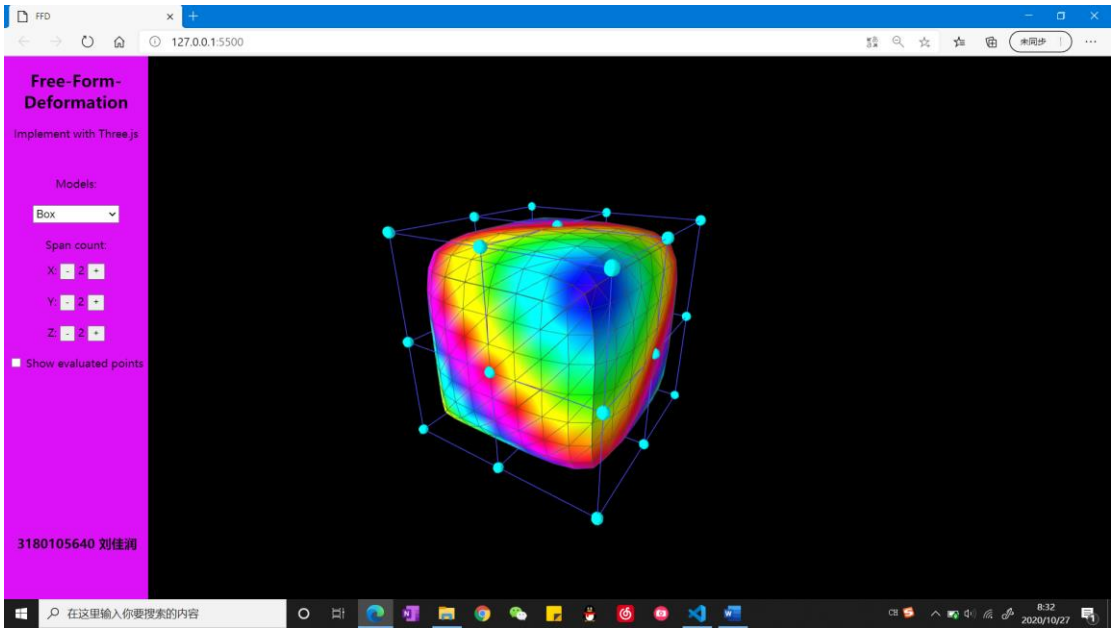
如果要绘制中间估计点（插值点），那么只需要调用我们上面的 evalLocal 函数就可以得到插值点的位置。

附：调用的 Threejs 库包括：

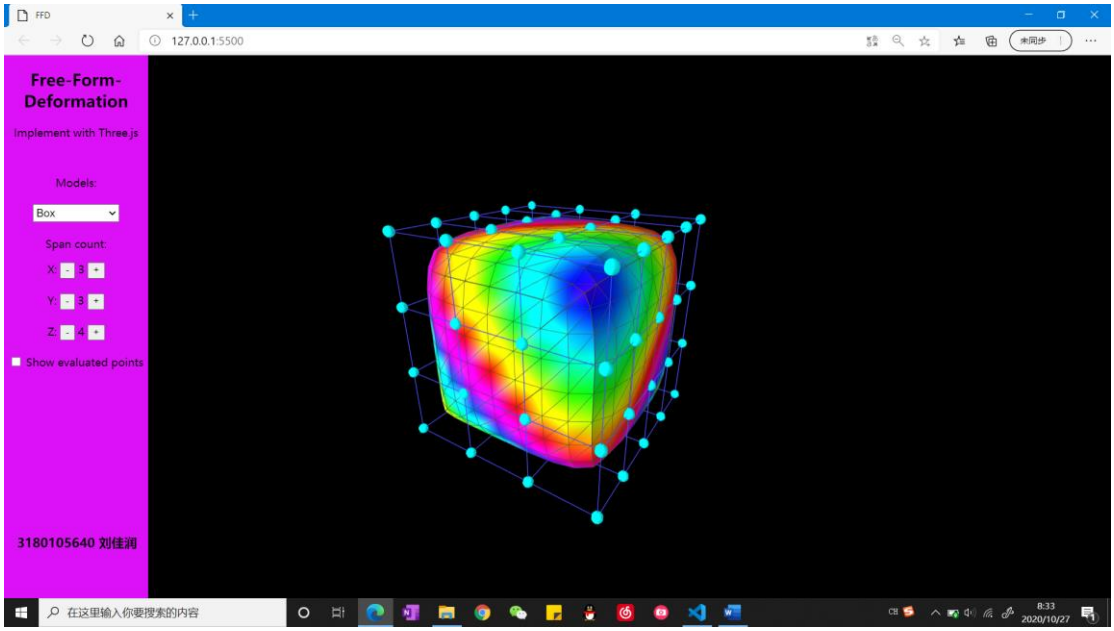
- OrbitControl 库：视角轨道控制器，建立鼠标与屏幕的交互，可以拖动、缩放等等。
- TransformControl 库：交互变形控制器，自带类似于 Maya 操作界面的移动 UI 等。
- SubdivisionModifier 库：对几何体进行指定等级的细分。

本次实验重点在与 FFD 算法的复现，关于界面的实现，不是本次实验的重点，因此不在报告中赘述。

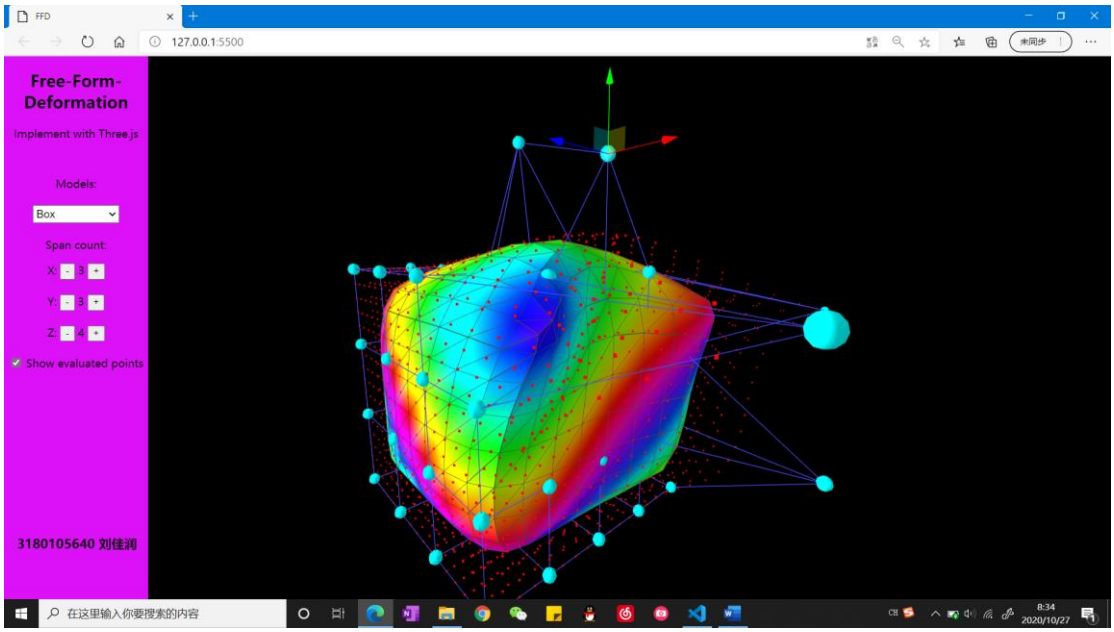
五、实验结果分析



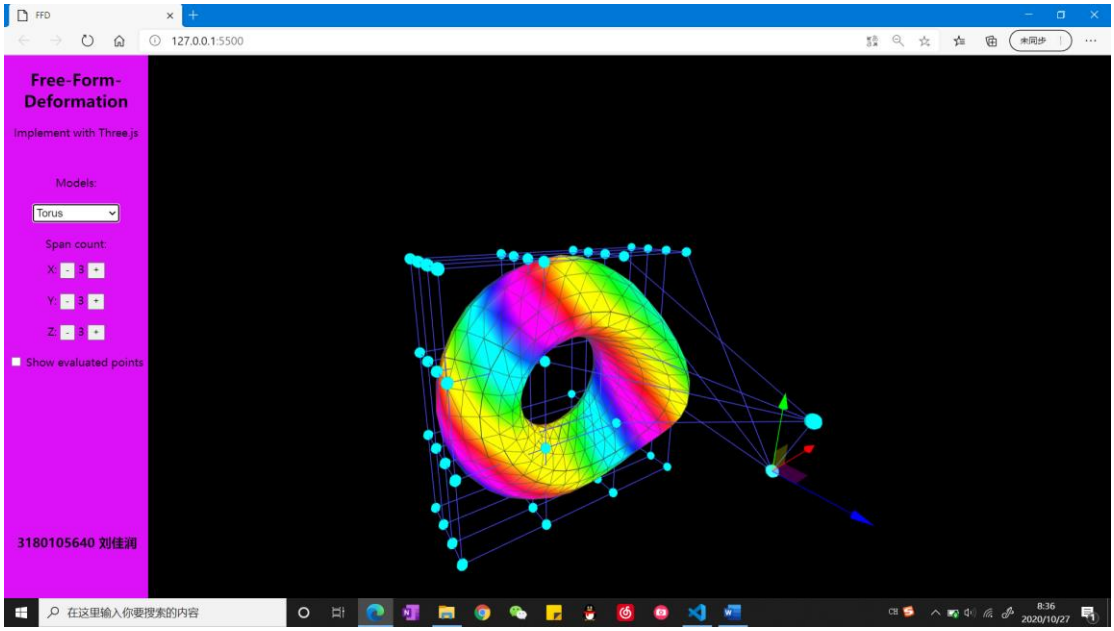
更改 Span Count

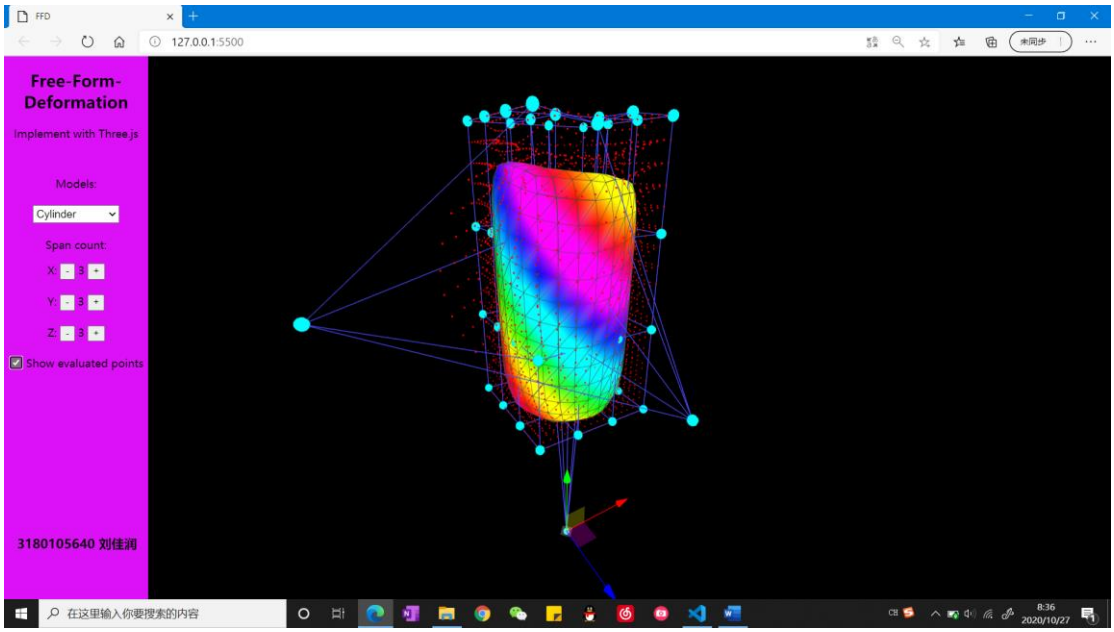


变形与插值点的显示


FFD
127.0.0.1:5500
Free-Form-Deformation
Implement with Three.js
Models:
Box
Span count:
X: - 3 +
Y: - 3 +
Z: - 4 +
Show evaluated points
3180105640 刘佳润
在这里输入你要搜索的内容
8:34
2020/10/27

其他模型


FFD
127.0.0.1:5500
Free-Form-Deformation
Implement with Three.js
Models:
Torus
Span count:
X: - 3 +
Y: - 3 +
Z: - 3 +
Show evaluated points
3180105640 刘佳润
在这里输入你要搜索的内容
8:36
2020/10/27



更多效果请看录屏演示

六、反思与总结

通过对 Siedberg 的论文的复现，理解了 FFD 的各个公式的意义，参数的意义。通过网页编程的框架，对网页交互的实现也更加娴熟。WebGL 框架下的图形学渲染编程有点像 OpenGL，比较繁琐，但是效果很好。

对于变形方法，我发现在控制点变多的情况下，变形的效果会有所打折，这应该是受于公式所限。对于算法的改进，还可以再搜索一些资料，了解 FFD 的改进方法。

在很多 3D 建模软件里对于 FFD 变形都有了封装，基本上采用的都是 4*4*4 的控制网格。从现在实验的效果来看，当分段数大的时候，变形会比较细微。

七、参考资料

Three.js 的交互拾取 https://zhuanlan.zhihu.com/p/143642146

Three.js 高级材质：MeshLambertMaterial

https://blog.csdn.net/qq_30100043/article/details/78047104

THREE.js 几何体 Geometry

https://blog.csdn.net/yangnianbing110/article/details/51306653

Three.js 几何变换操作库 TransformControl

https://www.jianshu.com/p/7858db822b62

Three.js Vector3 类源码说明

https://github.com/omni360/three.js.sourcecode/blob/master/Three.js