

一、实验目的和要求

- 设计并实现一个关键帧变形动画系统（线性插值与矢量线性插值）。
- 了解关键帧动画系统的结构，变形算法的思想以及不同算法对应的不同性能。

二、实验内容和原理

- 输入数据：
 - 包括初始形状数据和终止形状数据, 一般为事先定义好的整型变量数据, 如简单的几何物体形状（苹果，凳 陶罐）以及简单的动物形状（大象，马）等。
 - 可以设计交互界面，用户通过界面交互输入数据。
- 插值算法，包括**线性插值**和**矢量线性插值**。
 - 线性插值：对于初始和终止形状上每个点的坐标 P_i 进行线性插值得到物体变形的中间形状；
 - 矢量线性插值：对初始形状和终止形状上每两个相邻点计算其对应的长 L_i 和角度 θ_i ，然后对 L_i 和 θ_i 进行线性插值得到中间长度和角度，顺序连接插值后定义的几个矢量得到中间变化形状。插值变量变化范围是 $[0, 1]$, 插值变量等于 0 时对应于初始形状，插值变量等于 1 时对应于终止形状；数据类型为 float。
- 插值结果输出
 - 在屏幕窗口直接显示中间插值结果
 - 将各个中间图形用图像保存并转换成 AVI 动画文件。

三、实验平台

HTML5 + CSS + Javascript @Chrome

四、实验步骤

1.背景

关键帧动画技术是计算机动画中的一类重要技术。对于一些对动画效果质量要求很高的对象，如头发、皮肤、衣服等，需要设计师精细把控每一帧。但对于一些不那么细致的对象，如背景中运动的汽车，天空中的云朵等，就不需要再手工绘制每一帧画面。使用关键帧动画技术，指定首尾图形，设定预期的参数，通过插值算法就能得到理想的中间帧和动画了。

2.线性插值

线性插值的核心思想是通过计算对应点的线性距离来得到中间图形，也就是两点连成线，插值点都分布在这条直线上。具体实现上，通过给定初始点集合和终止点集合，然后给定一个映射关系。对一一对应的点的位置，即对 x, y 坐标进行线性插值。

$$\begin{aligned}x &= x_0 * t + x_1 * (1 - t) \\y &= y_0 * t + y_1 * (1 - t)\end{aligned}$$

代码实现上：

```

//计算线性插值点
function getlinearPoint() {
  let num = 5;
  for (let i = 1; i <= num; i++) {
    t = i / num;
    // console.log(t);
    for (let j = 0; j < graphs[0].length; j++) {
      let x = t * graphs[1][j][0] + (1 - t) * graphs[0][j][0];
      let y = t * graphs[1][j][1] + (1 - t) * graphs[0][j][1];
      points.push([x, y]);
    }
    if (!gifMode) GraphGenerate();
    interPoints.push(points);
    points = [];
  }
  if (gifMode) gifMake();
}

```

3. 矢量线性插值

而矢量线性插值，是在给定初始点和终止点集合后，将n个点转换为n-1个按顺序首尾相连的向量。然后，再将向量转化为极坐标，并对长度和角度进行线性插值。相当于将其转换到另外一个空间，维护了其角度和长度上的连续性。

$$a = a_0 * t + a_1 * (1 - t)$$

$$p = p_0 * t + p_1 * (1 - t)$$

因此我们首先把直角坐标转换为向量再转换为极坐标：

```

//计算矢量线性插值
function calcVec() {
  //calc start Vec
  for (let i = 1; i < graphs[0].length; i++) {
    let dy = graphs[0][i][1] - graphs[0][i - 1][1];
    let dx = graphs[0][i][0] - graphs[0][i - 1][0];
    let r = calcLength(dy, dx); //计算x,y之间的距离r
    let a = Math.atan2(dy, dx); //反正切计算角度a
    startVec.push([r, a]);
  }
  //calc end Vec
  for (let i = 1; i < graphs[1].length; i++) {
    let dy = graphs[1][i][1] - graphs[1][i - 1][1];
    let dx = graphs[1][i][0] - graphs[1][i - 1][0];
    let r = calcLength(dy, dx);
    let a = Math.atan2(dy, dx);
    endVec.push([r, a]);
  }
}

```

接下来是对长度和角度进行插值。由于矢量的旋转具有二义性，如果我们仍然照搬线性插值的方法，会出现参差不齐的现象：一些向量按照预期的方向旋转，一些向量按照相反的方向旋转，虽然都能旋转到最终图形，但这样的效果显然不是我们想要的。

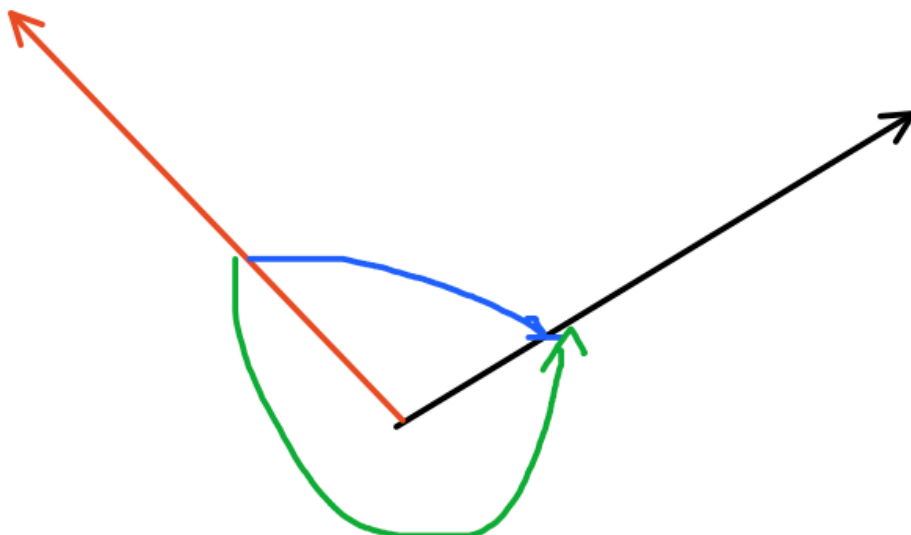


图1 矢量旋转的两种方向

因此我们不妨让用户告诉我们希望顺时针还是逆时针旋转，而我们在插值时中对每一次旋转强制保证末角度大于初角度（顺时针）或小于（逆时针）。

顺时针：

```
//顺时针矢量线性插值
function clockwise() {
  //calc starVecs and endVecs
  calcVec();
  //calc interVecs
  let num = 6;

  for (let i = 1; i <= num; i++) {
    let t = i / num;
    let interVec = [];
    for (let j = 0; j < startVec.length; j++) {
      let r = t * endVec[j][0] + (1 - t) * startVec[j][0];
      let a = t * endVec[j][1] + (1 - t) * startVec[j][1];
      if (endVec[j][1] - startVec[j][1] < 0) {
        a += t * 2 * Math.PI; //保证旋转后的角度始终大于初始角度
      }
      interVec.push([r, a]);
    }
    interVecs.push(interVec);
  }
  connectVecs(); //连接插值后的向量
}
```

逆时针：

```

//逆时针矢量插值
function anticlockwise() {
  //calc starVecs and endVecs
  calcVec();
  //calc interVecs
  let num = 6;
  for (let i = 1; i <= num; i++) {
    let t = i / num;
    let interVec = [];
    for (let j = 0; j < startVec.length; j++) {
      let r = t * endVec[j][0] + (1 - t) * startVec[j][0];
      let a = t * endVec[j][1] + (1 - t) * startVec[j][1];
      if (endVec[j][1] - startVec[j][1] > 0) {
        a += (1 - t) * 2 * Math.PI;
      }
      interVec.push([r, a]);
    }
    interVecs.push(interVec);
  }
  connectVecs();
}

```

由于矢量没有位置，只有大小和方向，因此我们还需要对第一个点线性插值，将矢量首尾连接起来得到最终的图形：

```

//首尾连接向量
function connectVecs() {
  //connect the vecs and draw
  let num = interVecs.length;
  for (let i = 1; i <= num; i++) {
    let t = i / num;
    points = [];
    let x = t * graphs[1][0][0] + (1 - t) * graphs[0][0][0];
    let y = t * graphs[1][0][1] + (1 - t) * graphs[0][0][1];
    points.push([x, y]);
    for (let j = 1; j <= interVecs[0].length; j++) {
      let x =
        points[j - 1][0] +
        interVecs[i - 1][j - 1][0] * Math.cos(interVecs[i - 1][j - 1][1]);
      let y =
        points[j - 1][1] +
        interVecs[i - 1][j - 1][0] * Math.sin(interVecs[i - 1][j - 1][1]);
      points.push([x, y]);
    }
    if (!gifMode) GraphGenerate();
    interPoints.push(points);
  }
  if (gifMode) gifMake();
}

```

4.用户体验

为了用户更好的使用体验，我们还需要完善输入输出。

输入：

- 用户可以在通过点击屏幕自定义生成图形：

```

//
function graphGenerate() {
  graphs.push(points);
  GraphGenerate();
  points = [];
}
//生成几何图形
function GraphGenerate() {
  ctx.beginPath();
  ctx.moveTo(points[0][0], points[0][1]);
  for (let i = 1; i < points.length; i++) {
    ctx.lineTo(points[i][0], points[i][1]);
  }
  ctx.lineTo(points[0][0], points[0][1]);
  ctx.stroke();
}
//监听画布点击事件
canvas.addEventListener("click", function (event) {
  getMousePositon(canvas, event);
});
//获取点击坐标并画点
function getMousePositon(canvas, event) {
  //1
  var rect = canvas.getBoundingClientRect();
  //2
  var x = event.clientX - rect.left * (canvas.width / rect.width);
  var y = event.clientY - rect.top * (canvas.height / rect.height);
  points.push([x, y]);
  DrawPoint(x, y);
  console.log("x:" + x + ",y:" + y);
}
//画单个点
function DrawPoint(x, y) {
  ctx.fillStyle = "rgb(0,0,0)";
  ctx.fillRect(x - 2, y - 2, 4, 4);
}

```

监听网页画布，每当用户点击空白处，记录下当前点的坐标并绘制单个点。当用户设计好初始图形后，点击生成图形按钮，我们取出当前的坐标数组，按照用户点击顺序生成几何图形，并准备好生成最终图形。

- 用户也可以选择我们预制的图形来观察插值效果：

 模型

```

//绘制基本图形铅笔
function drawPencil() {
  //first
  points = [
    [160, 220],
    [160, 240],
    [230, 240],
    [230, 250],
    [260, 230],
    [230, 210],
    [230, 220],
  ];
  if (!graphs.length) graphs.push(points);
  GraphGenerate();
  points = [
    [530, 250],
    [550, 250],
    [550, 180],
    [560, 180],
    [540, 150],
    [520, 180],
    [530, 180],
  ];
  if (graphs.length == 1) graphs.push(points);
  GraphGenerate();
  points = [];
}

```

模型

```

//绘制基本图形★
function drawStar() {
  //大★
  points = [
    [0, 85],
    [75, 75],
    [100, 10],
    [125, 75],
    [200, 85],
    [150, 125],
    [160, 190],
    [100, 150],
    [40, 190],
    [50, 125],
    [0, 85],
  ];
  graphs.push(points);
  GraphGenerate();
  //小★
  let tmp = [];
  for (let point of points) {
    tmp.push([point[0] * 0.5 + 500, point[1] * 0.5 + 100]);
  }
  graphs.push(tmp);
  points = tmp;
  GraphGenerate();
  points = [];
}

```

输出:

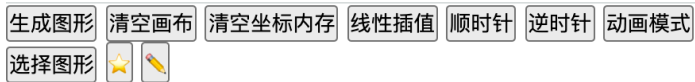
- 将插值中间结果全部显示在一张图片上。具体方法是每完成一幅中间结果插值就调用 `GraphGenerate()` 绘制当前图形
- 以动画模式绘制gif:

```
//生成动画
function gifMake() {
  let i = 0;
  let fps = 2;
  animate();
  function animate() {
    setTimeout(function () {
      ctx.clearRect(0, 0, width, height);
      points = interPoints[i];
      GraphGenerate();
      i++;
      if (i < interPoints.length) requestAnimationFrame(animate);
    }, 1000 / fps);
  }
}
```

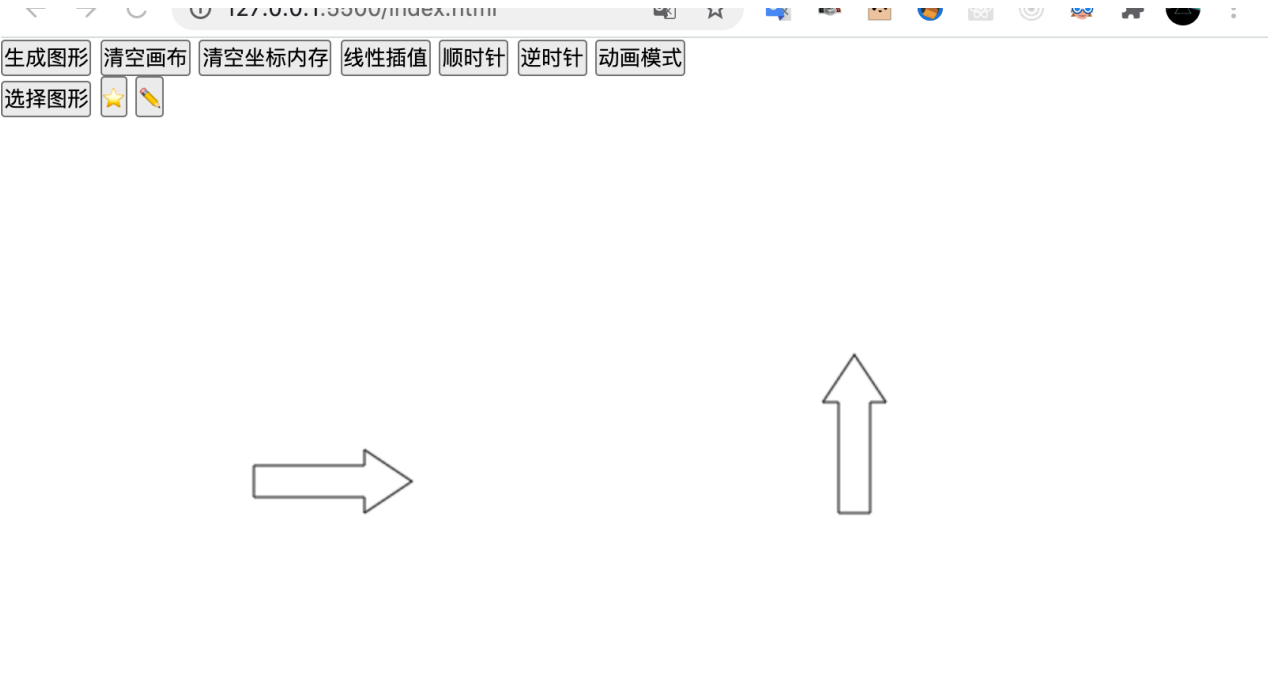
具体原理是设置一个 `Timelag` 或者说fps的倒数，每间隔 `Timelag` 便调用一次绘制函数，每次绘制前清空上一次图形。

五、实验结果分析

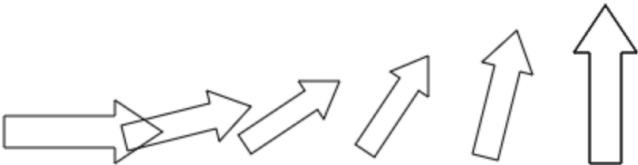
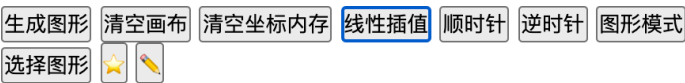
主界面



生成预设图形

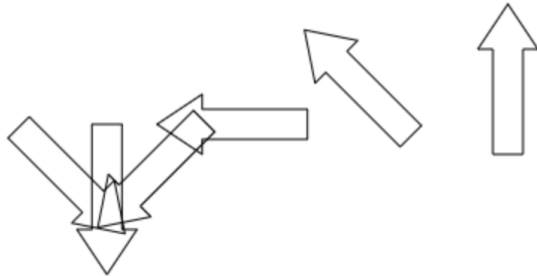


图形模式下线性插值



矢量插值（顺时针）

生成图形 清空画布 清空坐标内存 线性插值 顺时针 逆时针 图形模式
选择图形





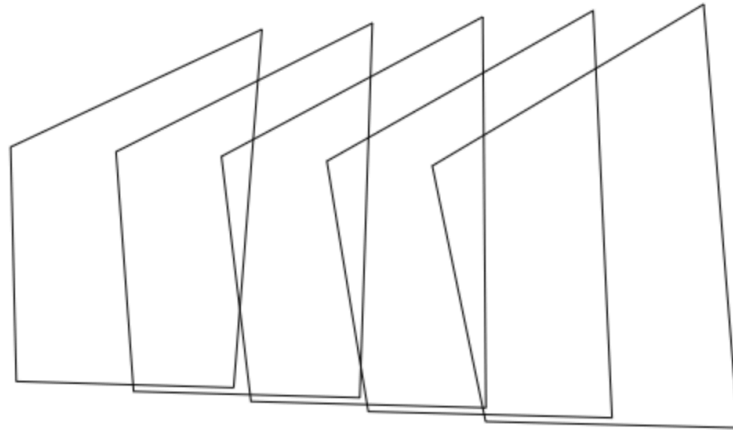
矢量插值（逆时针）

生成图形 清空画布 清空坐标内存 线性插值 顺时针 逆时针 图形模式
选择图形



点击生成图形

生成图形 清空画布 清空坐标内存 线性插值 顺时针 逆时针 图形模式
选择图形  



动画模式及详细演示效果见录屏

总结和反思

- 学习了线性插值算法
- 学习了矢量线性插值算法

在矢量插值时遇到了旋转方向不一致的问题，通过制定顺时针和逆时针解决。

- 进一步掌握了动画的绘制