3D打飞机小游戏

> Demo地址: https://omf2333.github.io/aircraft_game/begin.html （使用Chrome浏览）
>
> 由于模型文件和地形资源较大，所以请在良好的网络环境下体验游戏

- 3D打飞机小游戏
  - 1. 游戏概述
    - 1.1 操作介绍
    - 1.2 框架以及主要资源概览
  - 2. 下载与使用
  - 3. 游戏场景实现
    - 地形
    - 海洋和天空
    - 灯光
    - 声音
    - 粒子特效
    - 飞行器
    - 子弹
      - 导弹
    - 后期渲染
  - 4. 游戏逻辑实现
    - 飞行器移动
      - 敌机移动
      - 本机移动
        - 默认状态下
        - 鼠标控制
        - 按键控制
    - 攻击
      - 机枪攻击
      - 导弹攻击
    - 平视显示仪
      - 机枪温度
      - 瞄准镜
      - 敌机显示
        - 位置显示
        - 移动方向显示
      - 敌机锁定
      - 敌机显示移除

- 死亡
  - 本机
  - 敌机
- 游戏胜利
- 游戏失败

1. 游戏概述

在该游戏中，你可以使用键盘操作飞机做出各种诸如加速和左右翻滚等高难度的动作。敌人会随机出现在你的视野范围内，试着用鼠标瞄准并射击敌机。游戏系统通过提供瞄准镜和真实的爆炸效果来提高用户的游戏体验（无论你死还是对方死）。

1.1 操作介绍

[W] : 前进 [S] : 后退 [A] : 左偏航 [D] : 右偏航 [R] : 垂直向上 [F] ： 垂直向下 [MOUSE MOVING UP] : 向上飞行 [MOUSE MOVING DOWN] : 向下飞行 [MOUSE CLICKING LEFT] : 子弹攻击 [MOUSE CLICKING RIGHT] : 导弹攻击

同时屏幕右上角有面板可以实时调整战场环境渲染。

1.2 框架以及主要资源概览

- 前端框架：react.js
- 3D部分：Three.js
- 游戏逻辑部分：PubSub.js (组件传递消息库)
- 地形资源API：Free Terrain Tiles on AWS
- 粒子引擎：Shader Particle Engine
- 图形用户界面库：dat.gui.js
- 键盘库：keyboard.js
- 代码检测：eslint

2. 下载与使用

- 下载后在项目文件夹下

```
npm install
```

- 安装完所有依赖模块后

```
npm start
```

  - 即可访问`http://localhost:3000/begin.html`进行游戏

3. 游戏场景实现

地形

> 地形资源：Free Terrain Tiles on AWS，每块包含 256x256 vertices个点的海拔高度信息

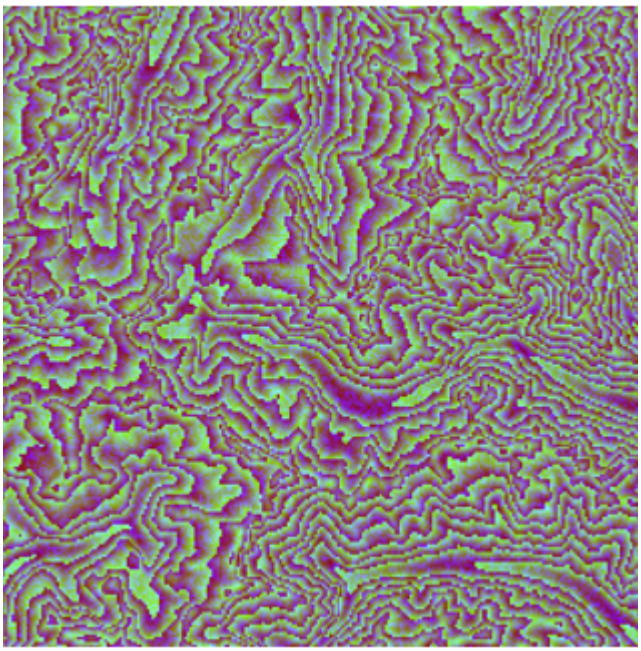
> 贴图资源：百度图片和poliigon 中的free texture
>
> 景深效果和地形着色：主要重用先有的开源资源

- 按照本机当前位置（x,y,z），实时请求aws的api

  > 在游戏开始时请求`AWS`的数据，在移动过程中过超出该地图块范围，则再次请求

```
const tilesElevationURL = 'https://s3.amazonaws.com/elevation-tiles-prod/terrarium'
const tileURL = `${tilesElevationURL}/${z}/${x}/${y}.png`
```

- 解析地图资源时将二维有颜色图片，按照颜色内含的高度信息，进行解析。



```
const pngToHeight = (array) => {
  //高度地图
  const heightmap = new Float32Array(256 * 256)
  for (let i = 0; i < 256; i++) {
    for (let j = 0; j < 256; j++) {
      const ij = i + 256 * j
      const rgba = ij * 4
        //按颜色进行解析
      heightmap[ij] = array[rgba] * 256.0 + array[rgba + 1] + array[rgba + 2] / 256.0 -
32768.0
    }
  }
  return heightmap
}
```

海洋和天空

> （`waterVertexShader`,`waterFragmentShader`,`VertexShader`,`FragmentShader`）为开源资源,使用了OpenGL着色语言，并且包含模型视图矩阵和模型视图投影矩阵

- 。Three.js中· `vertexShader`和`fragmentShader`两个着色器对其进行解析

```
//海洋
const WaterShader = {
//uniform变量是外部application程序传递给（vertex和fragment）shader的变量
// uniform变量一般用来表示：变换矩阵，材质，光照参数和颜色等信息。
  uniforms: {
    'color': {type: 'c', value: null},
    'reflectivity': {type: 'f', value: 0},
    'surface': {type: 'f', value: 0},
    'tReflectionMap': {type: 't', value: null},
    'tRefractionMap': {type: 't', value: null},
    'tNormalMap0': {type: 't', value: null},
    'tNormalMap1': {type: 't', value: null},
    'tDepth': {type: 't', value: null},
    'textureMatrix': {type: 'm4', value: null},
    'clipToWorldMatrix': {type: 'm4', value: null},
    'config': {type: 'v4', value: new Vector4()}
  },
    //表示一些顶点的数据，如：顶点坐标，法线，纹理坐标，顶点颜色等
  vertexShader: waterVertexShader,
  //负责颜色、纹理、光照等等
  fragmentShader: waterFragmentShader
}
```

```
//天空
Sky.SkyShader = {
  uniforms: {
    luminance: { value: 1 },
    turbidity: { value: 2 },
    rayleigh: { value: 1 },
    mieCoefficient: { value: 0.005 },
    mieDirectionalG: { value: 0.8 },
    sunPosition: { value: new Vector3() }
  },

  vertexShader: VertexShader,
  fragmentShader: FragmentShader
}
```

灯光

- `hemishpereLight`半球光，光源直接放置于场景之上，光照颜色从天空光线颜色颜色渐变到地面光线颜色
- 环境光`ambientLight`用于照亮环境中的物体
- 太阳平行光`dirLight`用于产生动态阴影和景深效果

```
  const initLights = (scene, sunPosition) => {
    dirLight.sunPosition = sunPosition
    dirLight.updatePosition = updateDirLightPosition
    dirLight.updatePosition()
    dirLight.up.set(0, 0, 1)
    dirLight.name = 'sunlight'
```

```
    // 平行光会产生动态阴影
    dirLight.castShadow = true
    // 阴影贴图的宽度
    dirLight.shadow.mapSize.width = dirLight.shadow.mapSize.height = 1024
    const d = 1024
    // 生成场景的深度图
    dirLight.shadow.camera.left = -d
    dirLight.shadow.camera.right = d
    dirLight.shadow.camera.top = d
    dirLight.shadow.camera.bottom = -d

    dirLight.shadow.camera.far = 3200
    // 阴影贴图偏差，在确定曲面是否在阴影中时，从标准化深度添加或减去多少。
    // 默认值为0.此处非常小的调整有助于减少阴影中的伪影
    dirLight.shadow.bias = -0.0001
    dirLight.needsUpdate = true

    scene.add(dirLight)
    scene.add(hemishpereLight)
    scene.add(ambientLight)
  }
```

- 对本机的光线更新是每0.2秒更新一次

声音

- 通过`PubSub`监听外部事件

```
PubSub.subscribe('x.drones.destroy', playExplosion)
PubSub.subscribe('x.drones.explosion', playExplosion)
```

- 启动音效

```
//爆炸音效
const playExplosion = (msg, drone) => {
  const n = Math.floor(Math.random() * explosions.length)
  if (!explosions[n].buffer) return
  const sound = new PositionalAudio(camera.listener)
  sound.setBuffer(explosions[n].buffer)
  sound.setRefDistance(500)
  drone.add(sound)
  sound.play()
}
```

粒子特效

> 使用适用于three.js的`Shader Particle Engine`

- 素材:

```
//爆炸效果
```

```
const fireGroupOptions = {
  //加载材质素材
  texture: {
    value: textureLoader.load(require('../textures/Explosion_002_Tile_8x8_256x256.png')),
      //图片是8*8
    frames: new Vector2(8, 8),
    // value: textureLoader.load(require('../textures/sprite-explosion2.png')),
    // frames: new Vector2(5, 5),
      //循坏一次
    loop: 1
  },
  depthTest: true,
  depthWrite: false,
    //和背景混合
  blending: AdditiveBlending,
    //效果大小
  scale: 600,
    //最大粒子数
  maxParticleCount: 25000
}

//烟雾效果
const pointsGroupOptions = {
  texture: {
    value: textureLoader.load(require('../textures/smokeparticle.png'))
  },
  depthTest: true,
  depthWrite: false,
  blending: NormalBlending,
  maxParticleCount: 25000
}
```

- 渲染:

```
//渲染爆炸
const flashGroup = new SPE.Group(fireGroupOptions)
flashGroup.addPool(poolSize, flashOptions, createNew)
triggerSingleEmitter(flashGroup, target)

//渲染烟雾
const smokeGroup = new SPE.Group(pointsGroupOptions)
smokeGroup.addPool(poolSize, smokeOptions, createNew)
triggerSingleEmitter(smokeGroup, target, true)
```

飞行器

> 玩家飞行器模型和敌机飞行器模型均来自 Sketchfab网站
>
> 模型文件目录 ./public/assets/drone/

- 加载模型

  > ./src/drone/loader.js

```
const loadDroneAssets = () => {
    // 加载本机模型资源
  loader1.load(
    './assets/drone/fighter/scene.gltf',
    function (gltf) {
      droneMesh1 = gltf.scene.children[0]
      PubSub.publish('x.assets.drone.fighter.loaded', {mesh: droneMesh1})
    }
  )
    // 加载敌机模型资源
  loader2.load(
        './assets/drone/enemy/scene.gltf',
      function (gltf) {
        droneMesh2 = gltf.scene.children[0]
        PubSub.publish('x.assets.drone.enemy.loaded', {mesh: droneMesh2})
      }
    )
}
```

- 模型三维包裹

> ./scr/index.js

  - 用一个三维球形网状结构将玩家飞机包裹，以便碰撞和碰撞效果实现

```
const drone = new Mesh(
  new SphereBufferGeometry(5, 5, 5),
  new MeshBasicMaterial({
    color: 0xffffff
  })
)
```

子弹

导弹

- Mesh类
  - 形状：SphereBufferGeometry （球形）
    - 半径为5
  - 材质：MeshPhongMaterial（镜面高光材质）
  - 颜色：白黄
  - 伤害：25

后期渲染

> BokehShader 和 CopesShader 来自 http://alteredqualia.com/

- 像CopyShader 这样生产阴影的基础常亮，参数预先设置

```
const CopyShader = {

  uniforms: {

    'tDiffuse': { value: null },
    'opacity': { value: 1.0 }

  },
```

- 诸如vertexShader这样会随着飞机移动动态变化的阴影，采用模板字符串，动态改变位置

```
vertexShader: [

    'varying vec2 vUv;',

    'void main() {',

    'vUv = uv;',
    'gl_Position = projectionMatrix * modelViewMatrix * vec4( position, 1.0 );',

    '}'

  ].join('\n'),
```

- EffectComposer 负责将渲染效果和渲染目标结合起来

```
const EffectComposer = function (renderer, renderTarget) {
  this.renderer = renderer

  if (renderTarget === undefined) {
    var parameters = {
      minFilter: LinearFilter,
      magFilter: LinearFilter,
      format: RGBAFormat,
      stencilBuffer: false
    }

    var size = renderer.getDrawingBufferSize()
    renderTarget = new WebGLRenderTarget(size.width, size.height, parameters)
    renderTarget.texture.name = 'EffectComposer.rt1'
  }

  this.renderTarget1 = renderTarget
  this.renderTarget2 = renderTarget.clone()
  this.renderTarget2.texture.name = 'EffectComposer.rt2'

  this.writeBuffer = this.renderTarget1
  this.readBuffer = this.renderTarget2

  this.passes = []

    // dependencies
```

```
    if (CopyShader === undefined) {
      console.error('EffectComposer relies on CopyShader')
    }

    if (ShaderPass === undefined) {
      console.error('EffectComposer relies on ShaderPass')
    }

    this.copyPass = new ShaderPass(CopyShader)
  }
```

- 其余文件大多为从外部引入的用于增强飞机飞行特效的文件，主要是着色器。

4. 游戏逻辑实现

> 该部分代码较多，就不在文档中放置，请在项目代码中查看，重要地方都有注释

飞行器移动

敌机移动

- 敌机的位置是按游戏时间戳更新的，所有的敌机移动轨迹都是固定一致的，在水平面上是一个半径为300的圆，在垂直方向上的高度是一个余弦曲线，具体的轨迹方程如下：

```
x = 300*cos(t/3000)
y = 300*sin(t/3000)
z = 300+ 50*cos(t/1000)
t为游戏时间戳（单位ms）
```

```
drone.position.set(
                radius * Math.cos(timestamp / 1000 / 3 + phase),
                radius * Math.sin(timestamp / 1000 / 3 + phase),
                300 + 50 * Math.cos(timestamp / 1000 + phase)
            )
```

  - 每一架敌机在不被打中的情况下完成飞行按轨迹飞行一圈的时间为18秒左右
  - 敌机的速度

本机移动

> 本机移动是放在一个loop中
>
> 对应文件：pilotDroneInit.js 和 FlyControlls.js

默认状态下

本机会以视野朝向（也就是透视摄像机的朝向）为移动方向矢量e（e为单位向量）

光标在屏幕中央时，e(0,0,0)

- 每次移动是在原来空间位置的基础上加上矢量（20 * ex,20 * ey,12 * ez）
- 为了让飞行器下降的不那么快，ez的系数就只乘以12

- 同时在移动过程中会改变飞行器模型的朝向

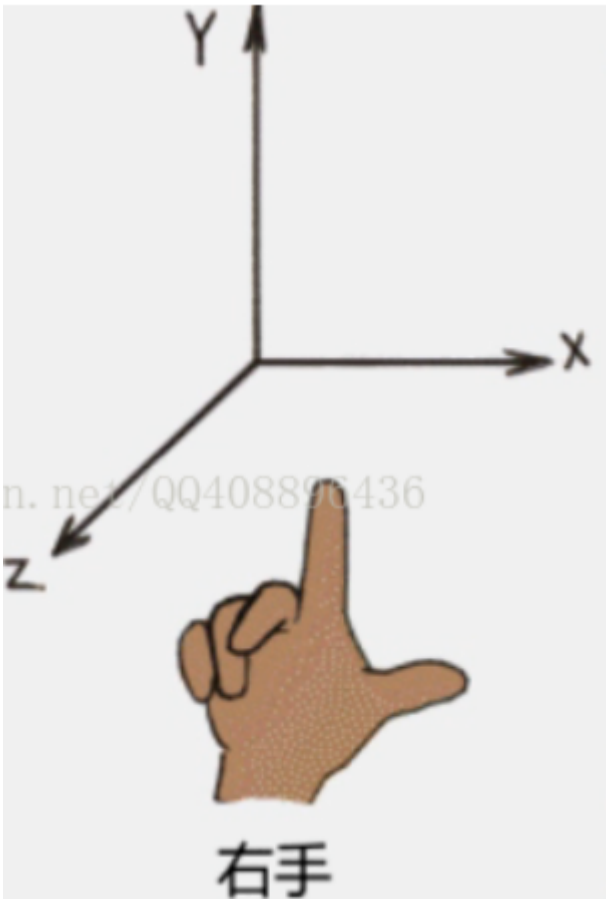
鼠标控制

> index.js

- 根据光标与屏幕中心点距离进行摄像机的移动（即本机）

按键控制

> 下列按键对应的坐标为three.js 中的右手坐标



- w: z轴数值加大，即前进
- s: z轴数值减小，即后退
- a:沿着y（`rotationVector.y`）轴正方向顺时针旋转
- d:沿着y（`rotationVector.y`）轴正方向逆时针旋转
- r:y轴数值加大，即上升
- f:y轴数值减小，即下降
- q:沿着z（`rotationVector.z`）轴正方向顺时针旋转
- e:沿着z（`rotationVector.z`）轴正方向顺时针旋转

攻击

> FireController.js

如果敌机在显示仪的小圈内，则会辅助性的显示其应该攻击的位置，当敌机的生命小于50时，敌机会一直冒烟

机枪攻击

- 点击鼠标左键时，会在`targetsInSight`(即平视显示仪小圈内与中心距离90以内的对象数组)寻找最近的敌机对象
    - 若找到了，以本机位置以及子弹发射方向做出一条直线，如果敌机对象距离该直线距离小于10，则判定击中。在渲染时，按照子弹发射方向，子弹速度取为500进行渲染。
    - 若没找到敌机对象则沿着原发射方向（即相机朝向）前进，直到停止发射子弹后（即松开鼠标左键），子弹消失。

导弹攻击

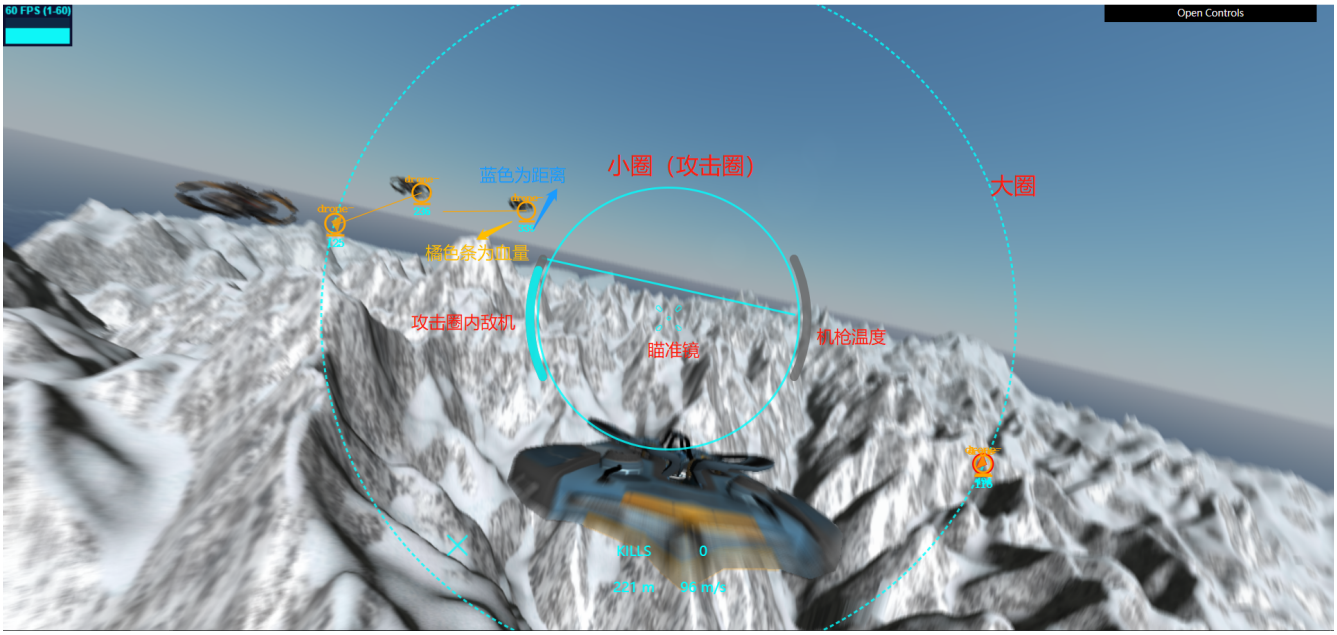
- 点击鼠标右键，会在`targetsInSight`(即平视显示仪小圈内与中心距离90以内的对象数组)寻找最近的敌机对象
    - 若找到了，则以该对象为的实时位置为导弹目的地，以10 * delta / 16.66单位前进，当子弹与目标距离小于10时，对目标敌机造成25点伤害，并触发击中效果
    - 若没找到敌机对象则不发射导弹

平视显示仪

> ./scr/hud

平视显示仪：是指将主要驾驶仪表姿态指引指示器和主要飞行参数投影到驾驶员的头盔前或风挡玻璃上的一种显示设备。使得驾驶员能看到重要的飞行参数。我在游戏中模拟了一个简陋的显示仪其中有：

- 海拔高度
- 击杀数
- 机枪温度
- 水平线
- 瞄准镜
- 敌机
    - 敌机编号
    - 敌机距离
    - 敌机血量
    - 敌机移动方向
    - 锁定敌机



机枪温度

- 显示仪小圈右侧的状态栏即表示机枪温度
- 当玩家开始按鼠标左键时，机枪时钟`pilotDrone.gunClock`会开始计时,机枪温度会随时间上升
- 当上升到最高到温度时，机枪无法继续使用，必须等到机枪温度降到最低，才能再次开火
- 当玩家开始放开鼠标左键时，机枪时钟`pilotDrone.gunClock`会停止计时,机枪温度会随时间下降

瞄准镜

> crosshair.js

- 代表子弹的发射方向，用于瞄准敌人

敌机显示

> index.js targetLoop 函数

位置显示

- 将敌机的三维坐标转为屏幕坐标
- 再将 敌机屏幕坐标 - 屏幕中心点坐标 获得显示仪坐标向量
- 将 敌机显示仪坐标向量 根据下列条件，化为终点在最大圈内的同向向量
  - 若原机显示仪坐标向量长度大于400，则终点在大圈上，同时根据敌机显示仪坐标向量的角度显示 敌机的身上的箭头朝向
  - 若小于400大于150，则终点在大圈内
  - 若小150，则在小圈内
- 同时维护一个数组，将原机显示仪坐标向量长度小于400的加入到其中

移动方向显示

- 若敌机的原机显示仪坐标向量长度 400 则显示它的移动方向
  - 该移动方向向量的长度按照 敌机距离本机的空间距离缩放
  - 同时若敌机的敌机的原机显示仪坐标向量 小于150，则显示攻击敌机的位置点

敌机锁定

- 当显示仪小圈（`targetInsight`数组不为空）内的有敌机，则小圈左侧的蓝色状态栏会上升，当上升到最高时，小圈周围会发出红光，提示玩家要攻击了

敌机显示移除

- 若敌机被摧毁，则会在屏幕上移除敌机

死亡

> ./src/controls/DeathController.js、FireController.js ./src/particles/particles.js

本机

- 本机在移动过程中会每0.2秒检测一下飞行器与地形的相对高度
  - 其中相对高度 = 地表和光线投影相交点与光线起点的z轴距离 - 光线距离与本机的z轴距离
  - 若相对高度小于5，则视为碰撞到地表，本机触发爆炸效果，游戏结束

敌机

- 当敌机收到攻击后的生命值小于等于0时，会触发爆炸效果，并且玩家击杀数+1

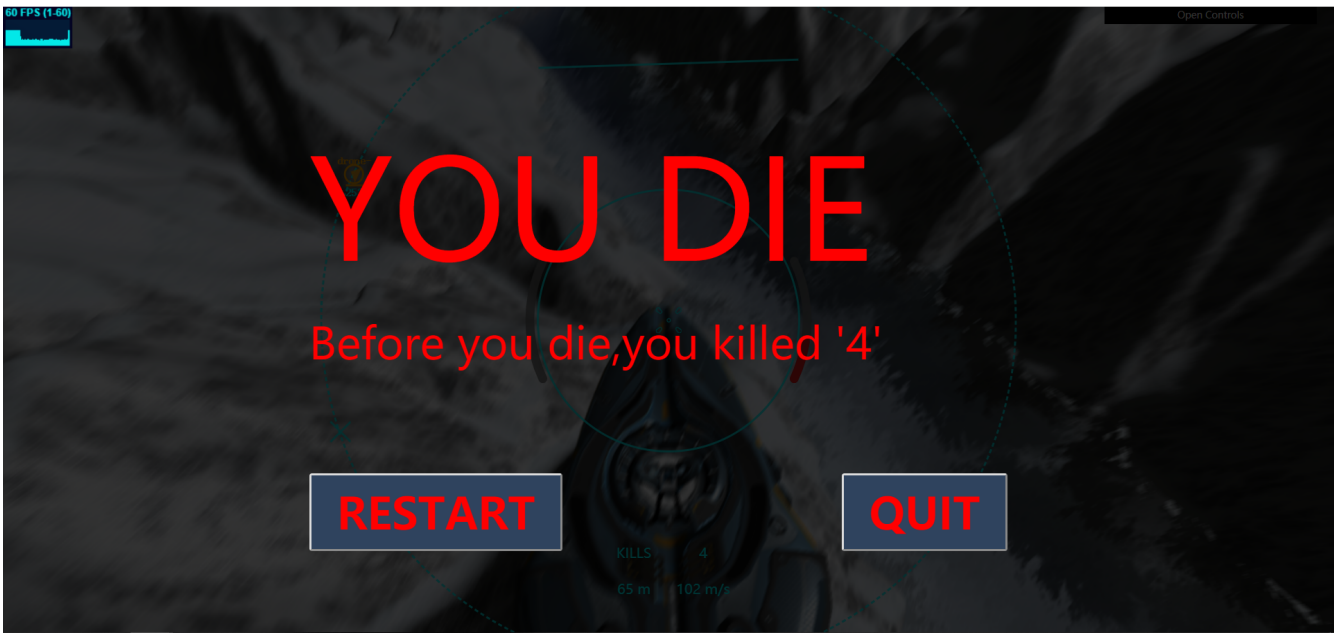
游戏胜利

> ./src/index.js

- 玩家击落全部5架敌机，通过PubSub触发win()函数。

游戏失败

> ./src/controls/DeathController.js



- 本机碰撞地形，导致爆炸死亡,触发`death()`函数。