设计开发文档（Odyssey）

前端（用户管理部分）

采用技术

使用了一些不错的开源框架

- Vue.js 前端主要框架
- axios RESTful 请求
- iview 前端视图框架
- socket.io 与游戏服务器进行交互

文件结构

`client\admin`

- `build` webpack 相关配置文件目录
  - `build.js` 生产环境结构代码
  - `check-version.js` 检查 node、npm 等版本
  - `dev-client.js` 热加载相关代码
  - `dev-server.js` 本地服务器
  - `utils.js` 构建工具
  - `webpack.base.conf.js` webpack 基本配置
  - `webpack.dev.conf.js` webpack 开发环境配置
  - `webpack.prod.conf.js` webpack 生产环境配置
- `config` 开发环境配置文件目录
  - `dev.env.js` 开发环境变量
  - `index.js` 项目基本配置
  - `prod.env.js` 生产环境变量
- `src` 项目源文件
  - `api` 与后台交互数据的封装
  - `App.vue` 默认组件，入口
  - `components` 组件目录 `containers` 容器目录 `my-theme` iview 个人主题定制文件
  - `router` 路由目录
  - `store` 个人信息状态管理

    - `utils` 辅助函数目录
    - `views` 细化的视图目录
- `static` 静态文件

Prerequisites

- Node.js v8.0.0 with npm v5.0.0

Commands

```
# install dependencies
npm install

# serve with hot reload at localhost:8080
npm run dev

# build for production with minification
npm run build
```

前端（游戏部分）

接口规范

```
// 新玩家加入房间
socket.on('new-comer', function (player_id) {});
// 新聊天信息
socket.on('new-message', function (name, msg) {});
// 子弹信息
socket.on('shoot-result', function (start, direction, distance) {});
// 同步所有玩家信息
socket.on('syn-pos', function (syn_players) {});
```

- 由于服务器和客户端以socket通信，因此接口设计为通信所需要传输的数据，使开发能同步进行，且不受另一端的干扰

组织及文件说明

- 文件树

```
|  index.html
|
├─audio
|      kill.wav
|      laser.wav
|      step.mp3
|
├─img
|  |  kill.png
|  |  zx2.png
|  |  zx3.png
|  |
|  └─bkg
|          nx.jpg
|          ny.jpg
|          nz.jpg
|          px.jpg
|          py.jpg
|          pz.jpg
|
├─js
|      Chat.js
|      Hero.js
|      Laser.js
|      LaserSound.js
|      Map.js
|      Player.js
|      PointerLock.js
|      Scene.js
|      Socket-single.js
|      Socket.js
|      ThirdPersonControls.js
|
├─lib
|      Detector.js
|      FBXLoader.js
|      inflate.min.js
|      stats.min.js
|
└─model
        gun.fbx
        hero.fbx
        model.fbx
        platform.jpg
        platform.json
        tower.FBX
```

- 说明
  - `index.html` 游戏启动网页
  - `lib` 包含依赖文件
  - `audio` 包含脚步、枪声等音频文件
  - `img` 包含准星、击杀logo等图片文件
  - `img/bkg` 包含星空背景图片
  - `model` 包含模型和贴图等文件
  - `js` 包含js文件

关键功能实现

- 卡顿补偿

```
//获取当前时间计算相邻两帧间隔时间delta，单位为秒
let time = performance.now();
let delta = ( time - prevTime ) / 1000;
prevTime = time;

//客户端不卡，相邻两帧间隔时间较短
if (delta < 0.05) {
  //使用delta处理移动
  hero.move(delta);
}
//客户端卡顿，相邻两帧间隔时间过长
else {
  //将delta分割成0.05s每小片
  for (; delta > 0; delta -= 0.05) {
      //最后一小片取delta分片后剩下的不足0.05s的时间
      let min = Math.min(0.05, delta);
      hero.move(min);
  }
}
```

  为了在任何客户端上保持移动速度一致，游戏中对移动的处理使用了相邻两帧间隔时间delta。不管客户端卡顿情况如何，只要经过的时间一致，移动距离就必然一致。

  但是，当一个客户端卡到爆炸的时候，两帧之间间隔太长会使得这次处理中移动距离过长，可能导致人物穿过障碍物或卡进障碍物内部。解决办法是将过长的帧间隔分片，保证每小片的移动处理正常。这样一来，如果再出现上面所说的卡到爆炸的情况，人物也会在若干小片的移动处理后撞到障碍物，从而在之后的小片中不会继续朝障碍物移动。

- 控制内存占用

```
//音频队列和音频加载器
var laserSounds = [];
var laserSoundLoader = new THREE.AudioLoader();

//初始化音频队列
function initSounds() {
  initSound(30);
}

//使用迭代实现嵌套闭包初始化音频
function initSound(i) {
  if (i < 0) {
      laserSoundLoadOK = true;
      return;
  }

  laserSoundLoader.load(
      'audio/laser.wav',
      function (audioBuffer) {
          //立体声
          var sound = new THREE.PositionalAudio(audioListener);
          sound.setBuffer(audioBuffer);
          //立体声音源需要添加到场景中
          //通过设置音源位置和听者位置达到立体声效果
          scene.add(sound);
          //添加到音频队列
          laserSounds.push(sound);
          //继续向下迭代
          initSound(--i);
      }
  );
}

//使用队列中的音频资源
function laserAudio(obj) {
  for (var i = 0; i < 30; i++) {
      //检测资源是否空闲
      if (!laserSounds[i].isPlaying) {
          //将音源位置设置为发声物体位置
          laserSounds[i].position.copy(obj.position);
          laserSounds[i].play();
          break;
      }
  }
}
```

Js在闭包中加载的资源不会被自动释放，最初的实现在播放激光声音时反复load音频文件，导致内存一

度飙升到2G。同时，加载器工作是异步的，并且两个加载器不能同时读取同一个资源文件，否则会报错。

解决办法是维护一个音频资源队列，每次获取资源时检测队列中每个资源的状态，获取第一个空闲的资源，将资源的状态改为忙碌，并在使用完后改回空闲。最终解决了反复load消耗内存的问题。

而异步加载的问题是通过迭代函数实现嵌套闭包来使资源按顺序加载，避免两个加载器同时读取同一个资源文件的情况发生。

- 地图碰撞检测

```
//创建射线方向与y轴负方向平行的raycaster
var raycaster = new THREE.Raycaster(new THREE.Vector3(), new THREE.Vector3(0,-
1,0), 0, RAY_LENGTH);

for (let i = 0; i < rayCount; i++) {
  //将raycaster的射线起始位置设为人物身体表面的某位置
  //bias为人物位置到人物身体表面某位置的偏移
  raycaster.ray.origin.x = hero.position.x + biasX[i];
  raycaster.ray.origin.y = hero.position.y + biasY[i];
  raycaster.ray.origin.z = hero.position.z + biasZ[i];

  //使用raycaster与objects中的对象检测碰撞
  var hit = raycaster.intersectObjects(objects);
  if ((hits.length > 0) && (hits[0].face.normal.y > 0)) {
      var actualHeight = hits[0].distance;
      //取碰到的物体中距离最近的
      minDistance = Math.min(minDistance, actualHeight);
  }
}

//在误差允许范围内均认为碰撞
if (minDistanc > -3 && minDistance < 1) {
  //使用碰撞位置修正人物位置
  hero.position.y -= minDistance;
  velocity.y = 0;
}
```

多次使用raycaster，从人物的身体表面的不同位置向下发射射线进行碰撞检测，使人物拥有底面积的效果。

检测到碰撞时，使用碰撞位置修正人物位置，使得人物爬坡时不会发生坐标抖动，并且始终紧贴地面，保证了操作和视觉上的舒适感。

- 游戏同步

```
// 设置定时器，频率为每50ms一次
var tid = setInterval(function () {
      if (hero.loaded)
          socket.emit('report-pos',
              id,
              hero.playing_run_forward,
              hero.playing_run_backward,
              hero.playing_run_left,
              hero.playing_run_right,
              hero.playing_jump_forward,
              hero.playing_jump_backward,
              hero.playing_fire,
              hero.playing_reload,
              hero.playing_die,
              hero.getModelPosition(),
              hero.getModelRotation());
  }, 50);
```

关键点在于频率的设置，频率过低则玩家能感受到的延迟过大，而频率越高，服务器与客户端承载的压力越大，包括计算与数据传输，经过测试发现每50ms一次既不会有过大的延迟，且服务器的负载也足够承受多个房间的压力。

心得体会

- 从球球大作战、贪吃蛇大作战等IO游戏得到启发，将其与第三人称射击的操作方式相结合，设计出一款“无限复活，无限钢枪” 的Web游戏
- 采用了迭代开发，首先实现了移动、射击等必需的基础功能，接着一边开发一边测试，逐步修复缺陷，同时为游戏设计、开发可以增添乐趣的新功能点
- 在项目开发的过程中分工明确，相互交流谈笑风生，充分发挥了团队开发的优势，有任何有疑问的设计都可以得到实时反馈，进一步提高了开发的效率，也降低了bug出现的频率

后端（用户管理部分）

采用技术

- node js (koa 2框架)

文件结构

- `models` 数据库模型定义目录
- `app.js` Koa 应用启动文件
- `controller.js` 扫描并注册控制器
- `db.js` 定义数据库模型规范

- `model.js` 扫描并导入数据库模型
- `package.json` npm 配置文件
- `rest.js` 支持 REST 的 Koa 中间件
- `socket.js` Socket.IO 事件处理逻辑
- * `controllers` 控制器目录
    - `api.js` RESTful API 业务逻辑
- `util.js` 数据库交互封装

Prerequisites

- MySQL Community Server v5.6.35
- Node.js v8.0.0 with npm v5.0.0

Database

`个人信息表user`

```
create
```

`战绩表record` ``` create

```

```

Commands

```
```shell
# install dependencies
npm install

# serve with hot reload at localhost:8080 (must build first for HTML    changes!)
npm run start

# build for production
npm run build

# clean built files
npm run clean
```

后端（游戏服务器部分）
```

接口规范

```
// 创建房间
socket.on('create', function (room_id, fn) {});
// 玩家加入房间
socket.on('join', function (socketID, playerID, room_id, fn) {});
// 删除房间
socket.on('remove-room', function (room_id) {});
// 用户退出
socket.on('quit-player', function (socketID) {});
// 位置报告
socket.on('report-pos', function (socketID, run_forward,
        run_backward, run_left, run_right,
        jump_forward, jump_backward,
        fire, reload, die, position, rotation) {})
// 射击报告
socket.on('report-shoot', function (socketID, position, direction) {});
// 聊天消息
socket.on('chat-message', function (socketID, msg) {});
```

- 由于服务器和客户端以socket通信，因此接口设计为通信所需要传输的数据，使开发能同步进行，且不受另一端的干扰

组织及文件说明

- 文件树 `shell |---bin | |---www |---package.json |---app.js |---configure.js`
- 说明
    - `bin/www` 服务器启动脚本
    - `package.json` 配置文件
    - `app.js` 处理与客户端通信以及游戏逻辑
    - `configure.js` 游戏里的一些常量，包括血量，死亡时间，分数等等

关键功能实现

- 地图载入

```
// 地图载入并加入场景
function makePlatform( jsonUrl, scene ) {
  var loader = new THREE.JSONLoader();
  // 读取地图文件
  var contents = fs.readFileSync(jsonUrl);
  // 将文件内容转换为json格式
  var jsonContent = JSON.parse(contents);
  // 使用JSONLoader通过json数据创建模型
  var model = loader.parse( jsonContent );
  // 计算表面向量
  model.geometry.computeFaceNormals();
  // 创建地图Mesh对象
  var platform = new THREE.Mesh( model.geometry );
  // 放缩
  platform.scale.x = 10;
  platform.scale.y = 10;
  platform.scale.z = 10;
  // 添加名字便于索引
  platform.name = 'platform';
  // 加入场景
  scene.add(platform);
}
```

关键点在于使用 `JSONLoader.parse()` 而非 `JSONLoader.load()` ，后一方法使用的是浏览器内建的XMLHttpRequest对象，通过http网络传输来加载。而我们的地图本就在服务器上，因此使用parse方法解析对应json即可。

- 人物模型加载

```
// 人物模型创建方法，参数为人物当前坐标与朝向
function getObject3d(position, rotation) {
  var object = new THREE.Mesh(geometry);
  object.position.set(position.x, position.y + OFFSET, position.z);
  object.rotation.set(rotation._x, rotation._y, rotation._z);
  return object;
}
var geometry = new THREE.BoxBufferGeometry(6, 20.5, 6);
var OFFSET = 10.25;
```

关键点在于人物模型十分复杂，如果我们直接将人物模型加载进地图，首先加载速度比较慢，其次占用了相当大的内存(如果为每一位玩家都维护一个对应的模型)，最后在碰撞检测的时候也会相应的降低性能。考虑到游戏里玩家操作的频繁，服务器端没有必要维护人物模型中许多无用的细节。因此我们借鉴了计算机图形学的思想，使用了一个BoxGeometry来包裹人物，人物的移动和碰撞检测都可以使用这个盒子而非本身的模型来做。这里用到的一些常量都是通过实验得出的，因此最终效果比较好。

- 子弹碰撞检测

```
function checkShoot(pos, dir) {
  // 方向向量正则化
  dir.normalize();
  // 更新场景里每个对象的世界坐标!!
  scene.children.forEach(function (mesh) {
      mesh.updateMatrixWorld();
  });
  // 初始化rayCaster
  rayCaster.set(pos, dir);
  // 碰撞检测
  var intersects = rayCaster.intersectObjects(scene.children);
}
var rayCaster = new THREE.Raycaster();
```

关键点在于，仅仅修改场景中人物模型的位置和朝向是不起作用的，在服务器端没有使用渲染器渲染，因此修改了人物模型的位置并没有真正修改他的世界坐标，只有在碰撞检测之前手动更新场景中每个对象的世界坐标，才能正确的使用rayCaster的碰撞检测功能。

- 游戏同步

```
// 设置定时器，频率为每50ms一次
setInterval(function () {
  for(var room_id in rooms) {
      var room = rooms[room_id];
      // 更新房间信息
      update(room);
      // 同步玩家状态
      io.to('room-' + room_id).emit('syn-pos', room.players);
  }
}, 50);
```

关键点在于频率的设置，频率过低则玩家能感受到的延迟过大，而频率越高，服务器与客户端承载的压力越大，包括计算与数据传输，经过测试发现每50ms一次既不会有过大的延迟，且服务器的负载也足够承受多个房间的压力。

部署配置

```
# 使用ssh连接上位于阿里云的主机
ssh root@47.100.165.222
# 到阿里云的控制管理台开放80, 3000, 8000等3个端口
# 通过git clone下载github仓库的代码
git clone https://github.com/devilpi/Odyssey.git
# 用户管理服务器部署
# 游戏服务器部署
cd Odyssey/FPSServer
# 安装依赖
npm install
# 运行
nohup node app.js &
```

心得体会

- 因为事先设计好了接口，使得开发非常有效率，出了bug能够第一时间的判断出是服务器还是客户端出现了问题，然后针对性的修复
- 采用了结对编程的方式，有任何有疑问的设计都可以得到实时反馈，进一步提高了开发的效率，也降低了bug出现的频率