Local Game tutorial:

Welcome in the first ud-viz game tutorial. Lines will be add step by step if you want to see complete files go to LocalGame et le projet complet est stoqué dans le dossier examples. At the end of this tutorial you will fly with your zeppelin in the sky of Lyon, and collect some sphere !

Create your game project 😄

Working environment

Steps :

- Create an empty folder that you can call `My_UD-Viz_Game` .
- Create an html script in your folder that you call `index.html` .

> Open the folder in visual studio code or your favorite IDE 💻

ud-viz

For this tutorial you will need to import `ud-viz` in your project, it is the package that contains the **game engine** and **urban data visualization tools** (hence the name).

To begin with, here is the **basis** of an html script, **copy it** :

```
<!--index.html-->
<!DOCTYPE html>
<html>
  <head>
    <title>My awesome game</title>

    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  </head>
  <body></body>
</html>
```

Pour la suite nous aurrons besoins d'heberger la page html, vous pouvez utilisez votre propre localserver, sinon voilà les étapes à suivre :

- Clonez le repo SimpleServer, à part :

```
git clone https://github.com/VCityTeam/UD-SimpleServer.git
```

- Ouvrez le repo SimpleServer dans un terminal
- Installer les packages nodes :

```
npm install
```

- Et enfin hebergez le dossier UD-Viz :

```
node index.js PATH_TO_My_UD-Viz_Game 8000
```

Vous pouvez visitez votre page à http://localhost:8000/ mais rien ne s'affiche.

Before you can use ud-viz, you need somewhere to display it. Save the following HTML to a file on your computer, along with a copy of udv.js in a assets/js/ directory, and open it in your browser.

```
<!DOCTYPE html>
<html>
  <head>
    ...
  </head>
  <body>
    <script src="./assets/js/udv.js"></script>
    <!--the path point your bundle library-->
  </body>
</html>
```

Toujours rien d'afficher mais la librairie est maintenant globalement accessible.

> Pour garder le tutoriel simple ud-viz est importé de cette mainere mais il existe un package npm, il est recommendé de prendre le paquet pour bénéficier des mises à jour notamment.

Création d'un nouveau monde `MyWorld` , ajoutons une balise de script dans notre index.js :

```
<script type="text/javascript">
  const myWorld = new udv.Game.Shared.World({
    name: 'My World',
    origin: { lat: 45.7530993, lng: 4.8452654, alt: 300 },
    gameObject: {
      name: 'GameManager',
    },
  });

  const app = new udv.Templates.LocalGame();
  app.start(myWorld, './assets/config/local_game_config.json');
</script>
```

First a new World called `My World` is created, you have to specified at which 3D coordonate you want to create it. Here we take a random location in Lyon. We also scpecified our root gameobject which is here called `GameManager`

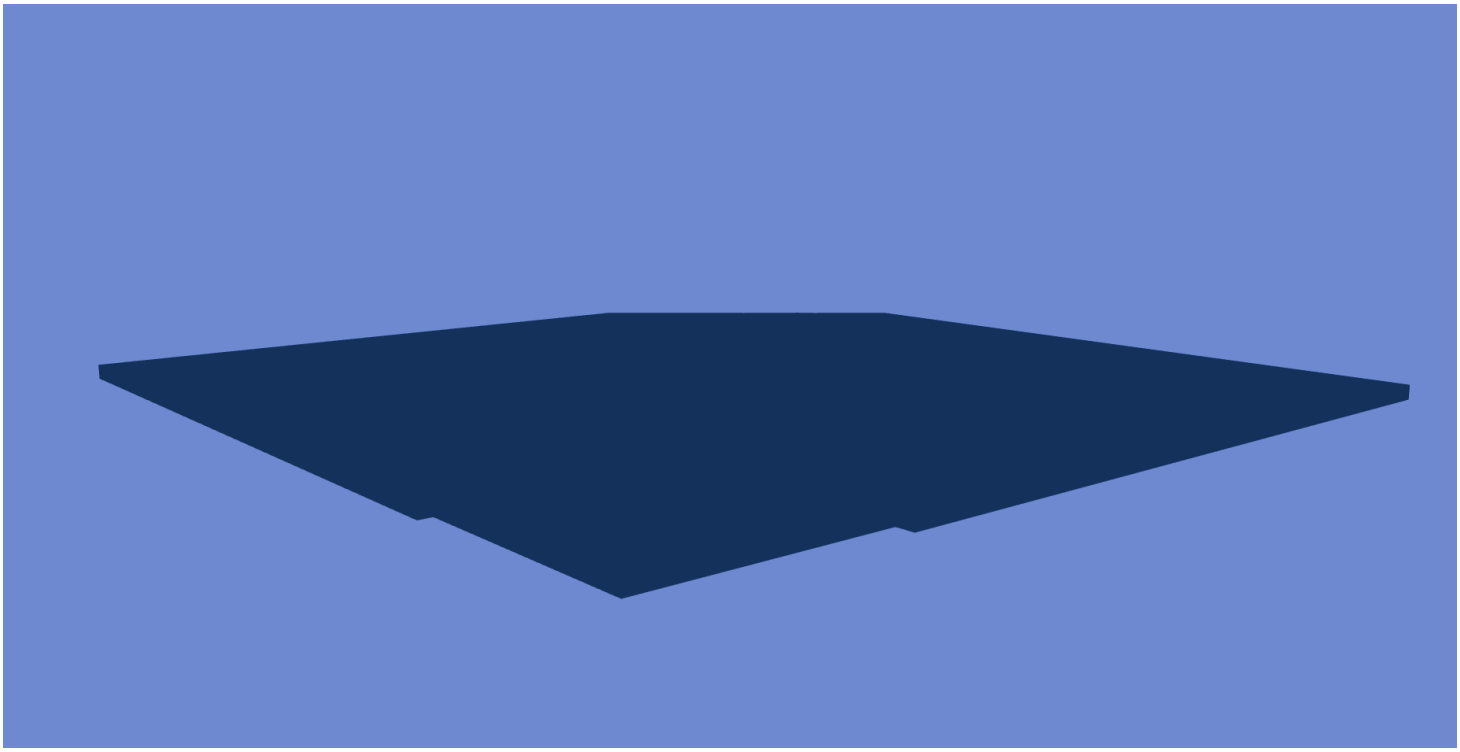
Then a `LocalGame` is instanciate, to start it you need to pass a world and a path to a config file.

Créer un ficher `local_game_config.json` dans le dossier assets/config (see also) et copiez-y :

```
{
  "game": {
    "fps": 30,
    "shadowMapSize": 2046,
    "skyColor": {
      "r": 0.4,
      "g": 0.6,
      "b": 0.8
    }
  },
  "itowns": {
    "radiusExtent": 1000
  }
}
```

Parameters in game section are relative to your GameView (the framerate, the size of the shadow map and the sky color). The itowns parameters is used to delimiter the area around the location of your world.

Ok at this point let's look what should appear on your browser.



That's great, you don't know it yet but this is Lyon, ok let's make the city appear !

Let's add these lines in your `local_game_config.json` file

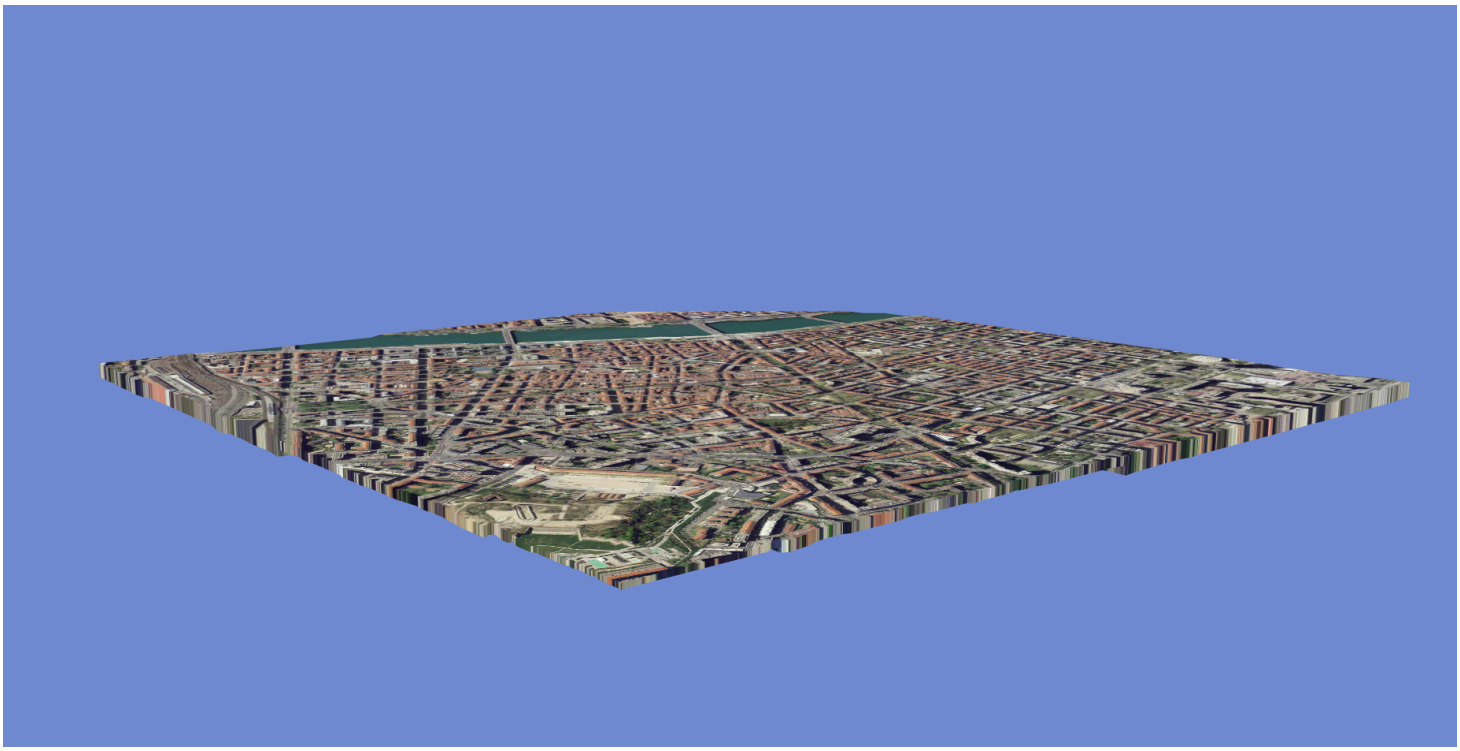
```
"background_image_layer": {
    "url": "https://download.data.grandlyon.com/wms/grandlyon",
    "name": "Ortho2018_Dalle_unique_8cm_CC46",
    "version": "1.3.0",
    "format": "image/jpeg",
    "layer_name": "Base_Map",
    "transparent": true
},
"elevation_layer": {
    "url": "https://download.data.grandlyon.com/wms/grandlyon",
    "name": "MNT2018_Altitude_2m",
    "format": "image/jpeg",
    "layer_name": "wms_elevation_test"
}
```

Here we are parameterized layer of the itowns framework on which `ud-viz` is builded.

"background_image_layer" define where (url) to query textures of the ground

"elevation_layer" is the texture used as a heightmap to specified the altitude of the ground

ok let's see how it looks like now:

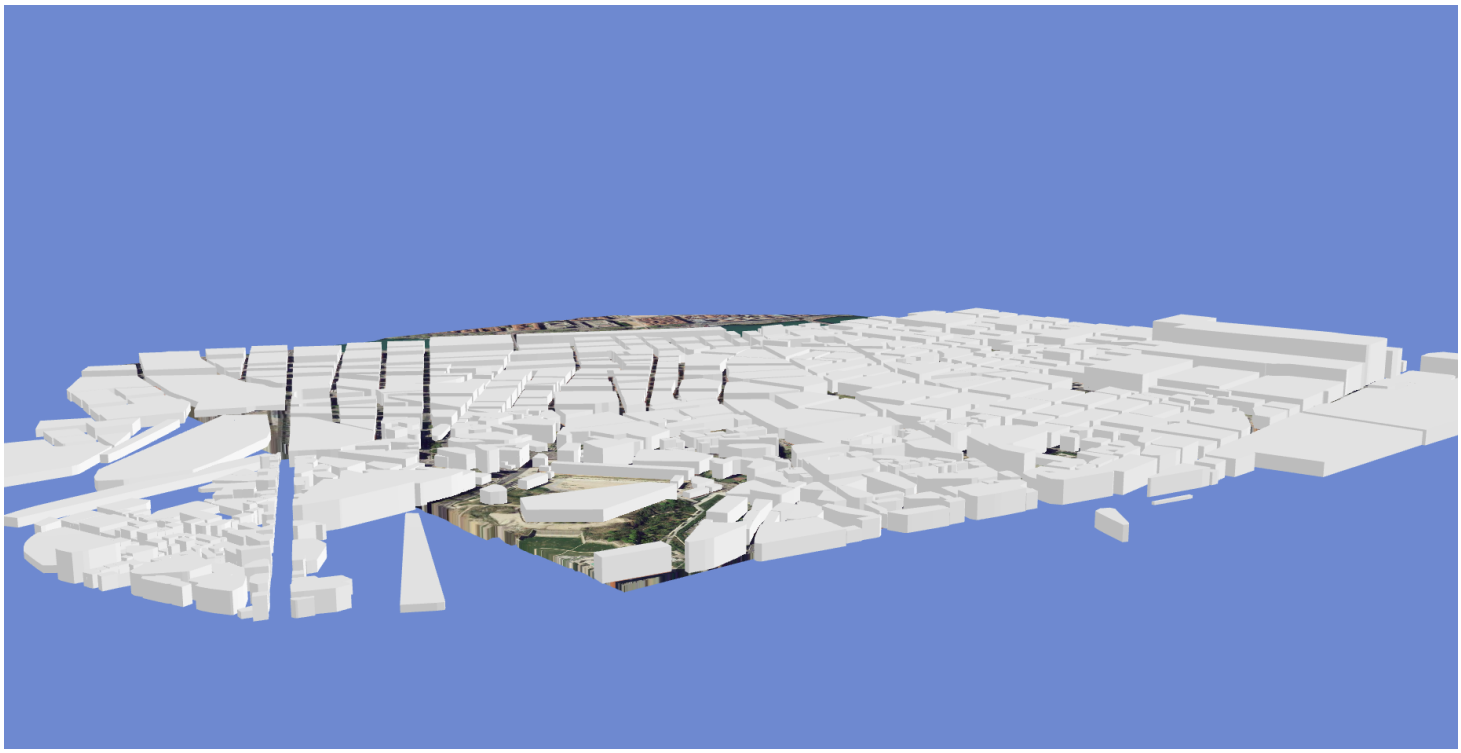


ok ok we are close, add these lines

```
"3DTilesLayer": {
    "id": "3d-tiles-layer-building",
    "url": "./assets/lod_flying_campus/tileset.json",
    "color": "0xFFFFFF",
    "initTilesManager": "true"
}
```

Here data (the geometry of building) are not collected from a distant server but locally, you need to download these folder, cpiez le dans ./assets/

your screen now



That's it lyon is here, ok now we are gonna to add our zeppelin.

First we are gonna to attach a WorldScript to our gameobject GameManager. A worldscript is used to customize the world simulation.

Ajoutez ça dans `index.html` dans la déclaration de `myWorld`

```
gameObject: {
    name: 'GameManager',
    static: true,
    components: {
        WorldScript: {
            idScripts: ['worldGameManager'],
        }
    },
},
```

static set to true is used for internal optimization and is meaning that this gameobject is not moving into the 3D scene.

now our GameManager is linked to a worldscript named worldGameManager. We need to import that script in our game. To do so add these lines to your `local_game_config.json` file.

```
"assetsManager": {
    "worldScripts": {
        "worldGameManager": {
            "path": "./assets/worldScripts/worldGameManager.js"
        }
    }
}
```

this means that now there is a worldscript named worldgamemanager located at path.

Finally we need to create that script `worldGameManager.js` , in ./assets/worldscripts/ folder.

skeleton of a worldscript is like this

```
let Shared;

module.exports = class WorldGameManager {
  constructor(conf, SharedModule) {
    this.conf = conf;
    Shared = SharedModule;
  }

  init() {}

  tick() {}
};
```

Copiez le dans `worldGameManager.js`

`conf` is metadata pass into the json file here there is none. `SharedModule` is the dynamic import of the ud-viz/Game/Shared lib (librairie qui sert à coder les worldscripts)

`init` is called when the gameobject is added, and `tick` is called every world simulation step.

let's add the zeppelin add these lines into init method.

```
  init() {
    //a context containing all references needed for scripting game
    const worldContext = arguments[1];
    const world = worldContext.getWorld();

    this.zeppelin = new Shared.GameObject({
      name: 'zeppelin',
      components: {
        Render: { idModel: 'zeppelin' },
      },
    });

    world.addGameObject(this.zeppelin, worldContext, world.getGameObject());
  }
```

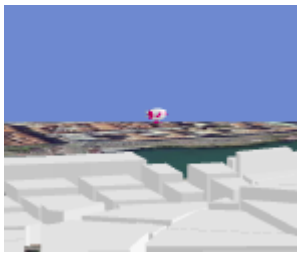
we create a new gameobject called zeppelin and we add a render component with an id of the 3D model.

as always we need to import that 3D model. Here we are gonna to use this one. Like the worldscript add these lines in your `local_game_config.json` file

```
 "assetsManager": {
    "models": {
      "zeppelin": {
        "path": "./assets/models/Zeppelin_Labex_IMU.glb",
        "anchor": "center_min",
        "rotation": { "x": 0, "y": 1.5707, "z": 0 }
      }
    },
    "worldScripts": {
      "worldGameManager": {
        "path": "./assets/worldScripts/worldGameManager.js"
      }
    }
  }
```

`path` point to your .glb
`anchor` mean where the origin of the object is taken
`rotation` tweak a custom rotation in your 3D model

ok let's see what happens on screen



yes a zeppelin appear on the middle of the scene ! trust me...

Ok let's add a localScript now to focus this zeppelin with the camera. These scripts are used to customize client-side game

> Here every script could be a localScript here since the world is simulated on the clientside but it's good pratice to keep them separate since the world simulation could be executed somewhere else

GameManager in `index.html` become

```
gameObject: {
    name: 'GameManager',
    static: true,
    components: {
        WorldScript: {
            idScripts: ['worldGameManager'],
        },
        LocalScript: {
            idScripts: ['focus'],
        },
    },
}
```

Import it the same way that the worldscript with these lines in your `local_game_config.json` file.

```
"localScripts": {
    "focus": {
       "path": "./assets/localScripts/focus.js"
    }
}
```

and here is the focus script, copiez le dans le dossier ./assets/localScripts

focus.js

ok here is what you should see, you should also be able to zoom in/out with the wheel !

ok in the next step we are gonna to move the zeppelin above the city

let's add a `commands.js` localscript. add it in the gamemanager gameObject

```
gameObject: {
    name: 'GameManager',
    static: true,
    components: {
        WorldScript: {
            idScripts: ['worldGameManager'],
        },
        LocalScript: {
            idScripts: ['focus', 'commands'],
        },
    },
}
```

Import it

```
"localScripts": {
      "focus": {
        "path": "./assets/localScripts/focus.js"
      },
      "commands": {
        "path": "./assets/localScripts/commands.js"
      }
    }
```

Here is what this localscript looks like, copiez le dans le dossier ./assets/localScripts

commands.js

ok now commands are send to world simulation but the world don't know what to do with them.

in the `worldGameManager.js` worldscript let's add these lines in the `tick` function

```
tick() {
    const worldContext = arguments[1];
    const dt = worldContext.getDt();
    const commands = worldContext.getCommands();
    const speedTranslate = 0.05;
    const speedRotate = 0.0003;
    const zeppelin = this.zeppelin;

    commands.forEach(function (cmd) {
      switch (cmd.getType()) {
        case Shared.Command.TYPE.MOVE_FORWARD:
          zeppelin.move(
            zeppelin.computeForwardVector().setLength(dt * speedTranslate)
          );
          break;
        case Shared.Command.TYPE.MOVE_BACKWARD:
          zeppelin.move(
            zeppelin.computeBackwardVector().setLength(dt * speedTranslate)
          );
          break;
        case Shared.Command.TYPE.MOVE_LEFT:
          zeppelin.rotate(new Shared.THREE.Vector3(0, 0, speedRotate * dt));
          break;
        case Shared.Command.TYPE.MOVE_RIGHT:
          zeppelin.rotate(new Shared.THREE.Vector3(0, 0, -speedRotate * dt));
          break;
        default:
          throw new Error('command not handle ', cmd.getType());
      }
    });
  }
```

Ok now your travel in zeppelin is possible ! Try it with Z,Q,S,D or Arrows.

Final Step

Now we are going to add some collectable sphere.

In `worldGameManager.js` add this method

```
createCollectableSphere(x, y) {
  const size = 10;

  const result = new Shared.GameObject({
    name: 'collectable_sphere',
    static: true,
    components: {
      Render: {
        idModel: 'sphere',
        color: [Math.random(), Math.random(), Math.random()],
      },
    },
    transform: {
      position: [x, y, size],
      scale: [size, size, size],
    },
  });

  return result;
}
```

and then in the `init` method

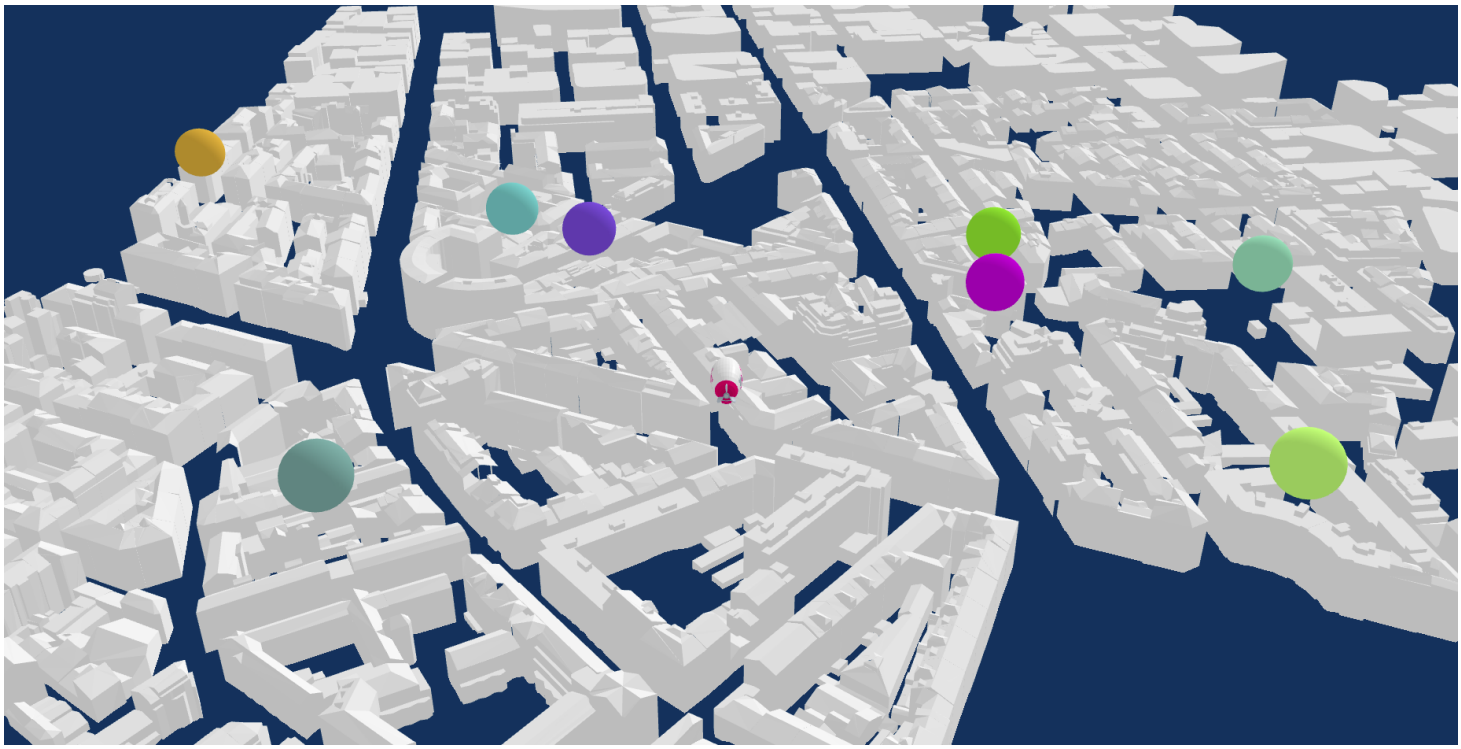
```
//add collectable sphere at random position
const range = 400;
const minRange = 50;
for (let i = 0; i < 10; i++) {
  let x = (Math.random() - 0.5) * range;
  let y = (Math.random() - 0.5) * range;

  if (x > 0) {
    x += minRange;
  } else {
    x -= minRange;
  }

  if (y > 0) {
    y += minRange;
  } else {
    y -= minRange;
  }

  const s = this.createCollectableSphere(x, y);
  world.addGameObject(s, worldContext, world.getGameObject());
}
```

now you should see sphere around your zeppelin (zoom out 😄)



ok that's nice, now let handle the collision with these objects.

first add a collider component to these spheres in `worldGameManager.js`

```
  createCollectableSphere(x, y) {
    const size = 10;

    const result = new Shared.GameObject({
      name: 'collectable_sphere',
      static: true,
      components: {
        Render: {
          idModel: 'sphere',
          color: [Math.random(), Math.random(), Math.random()],
        },
        Collider: {
          shapes: [
            {
              type: 'Circle',
              center: { x: 0, y: 0 },
              radius: size / 2,
            },
          ],
        },
      },
      transform: {
        position: [x, y, size],
        scale: [size, size, size],
      },
    });

    return result;
  }
```

then add a collider component to the zeppelin dans la fonction `init` dans la definiton de `this.zeppelin`

```
this.zeppelin = new Shared.GameObject({
  name: 'zeppelin',
  components: {
    Render: { idModel: 'zeppelin' },
    Collider: {
      shapes: [
        {
          type: 'Circle',
          center: { x: 0, y: 0 },
          radius: 10,
        },
      ],
    },
  },
});
```

ok now let's add a worldscript to the zeppelin to handle collision

create a new worldscript import it with the config files and create it in the assets

zeppelin.js

now when you touch sphere with the zeppelin they are disapearing !!