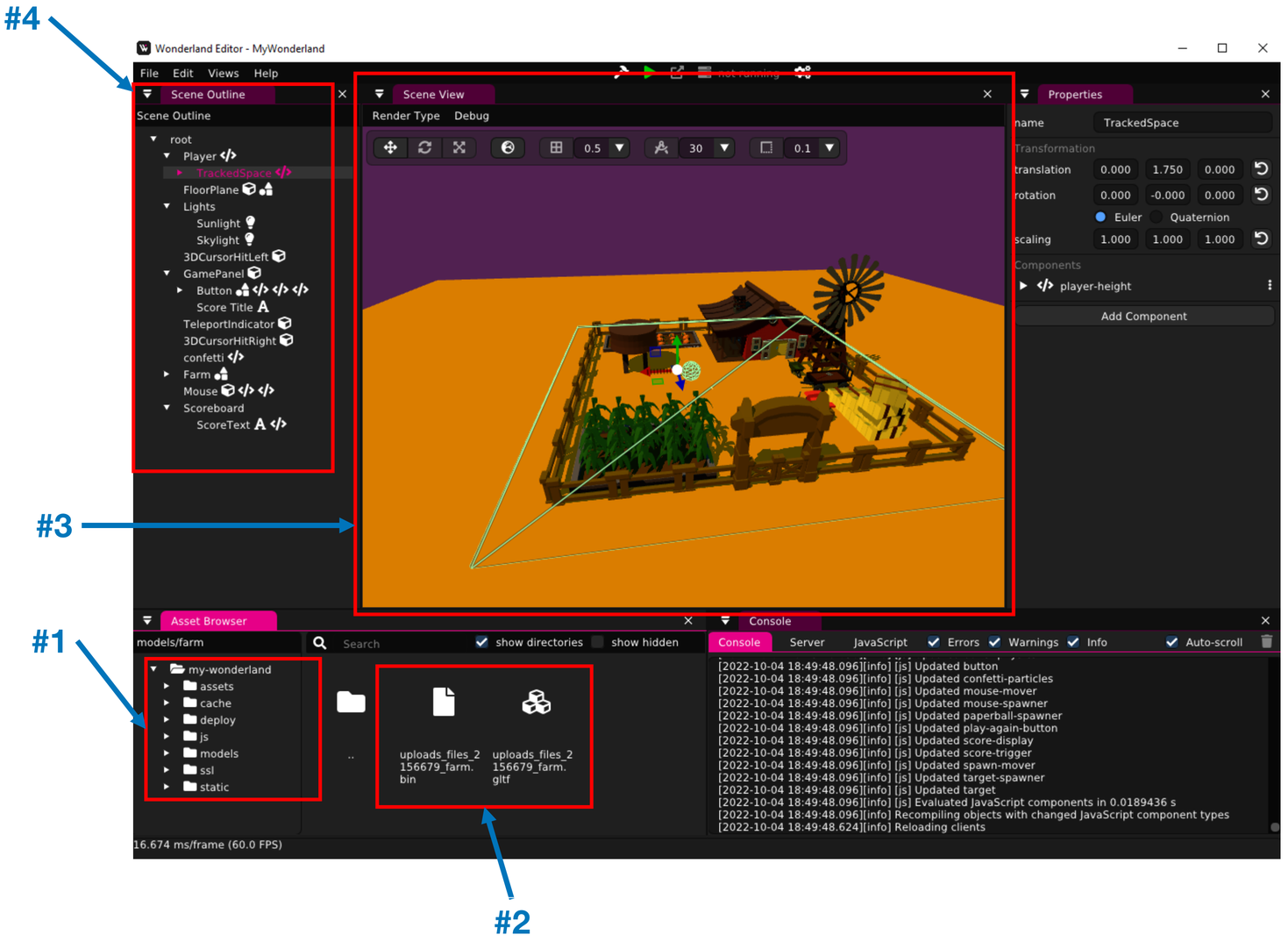
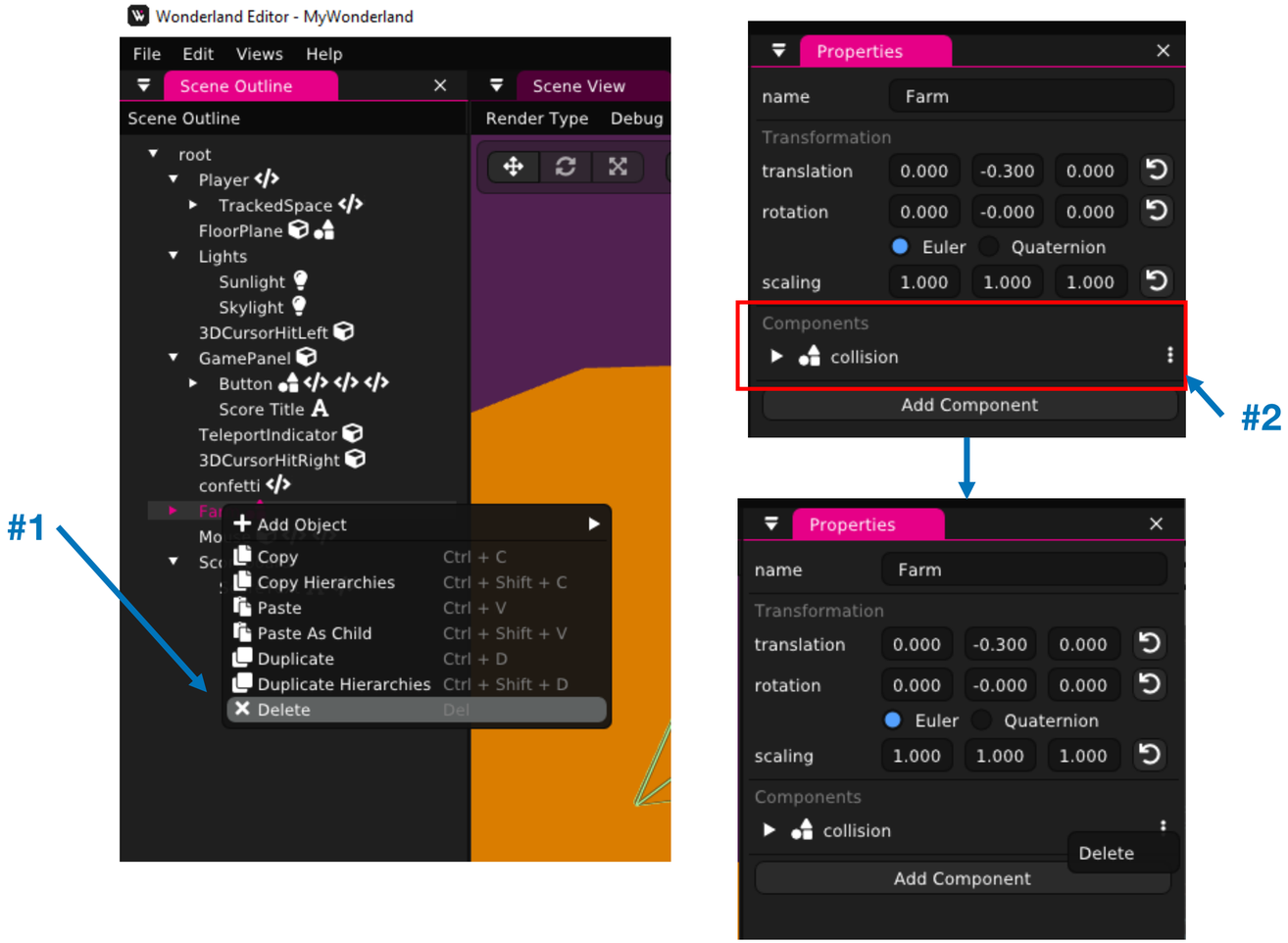
Import 3D objects into the scene



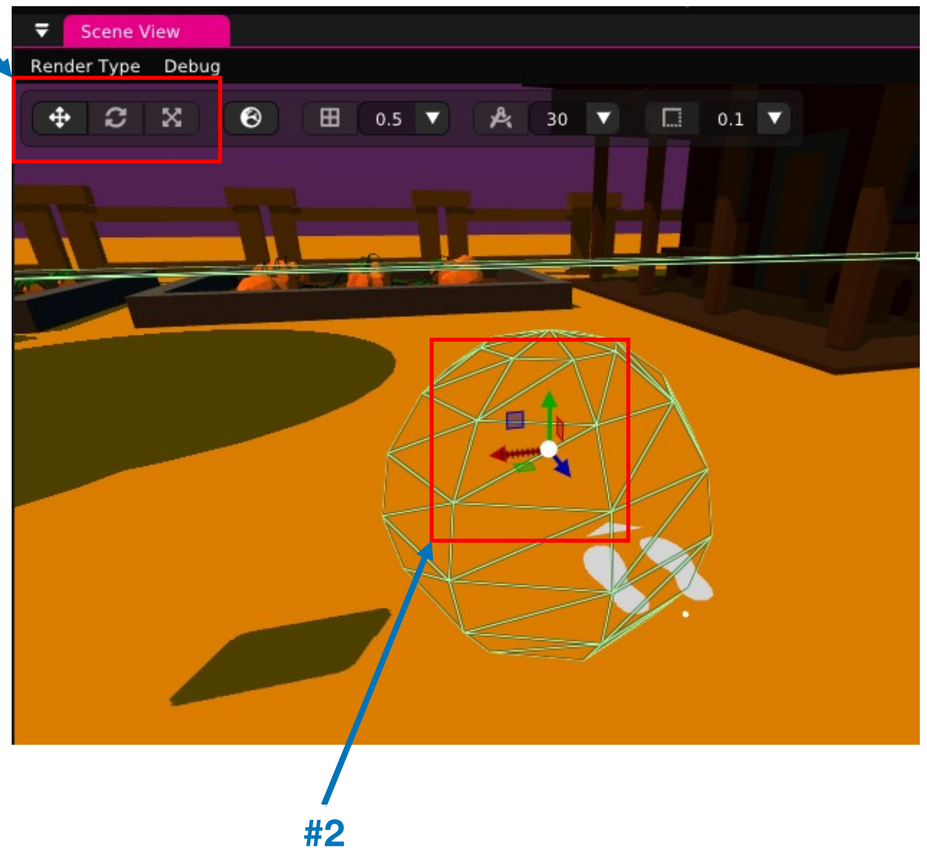
1. From the **Asset Browser**, open the models folder.
2. Drag and drop your 3D objects from your system into the **models folder**.
3. Drag and drop your 3D objects from the **models folder** into the Scene View.
4. Your object should show up in the **Scene View** as well as the **Scene Outline**.

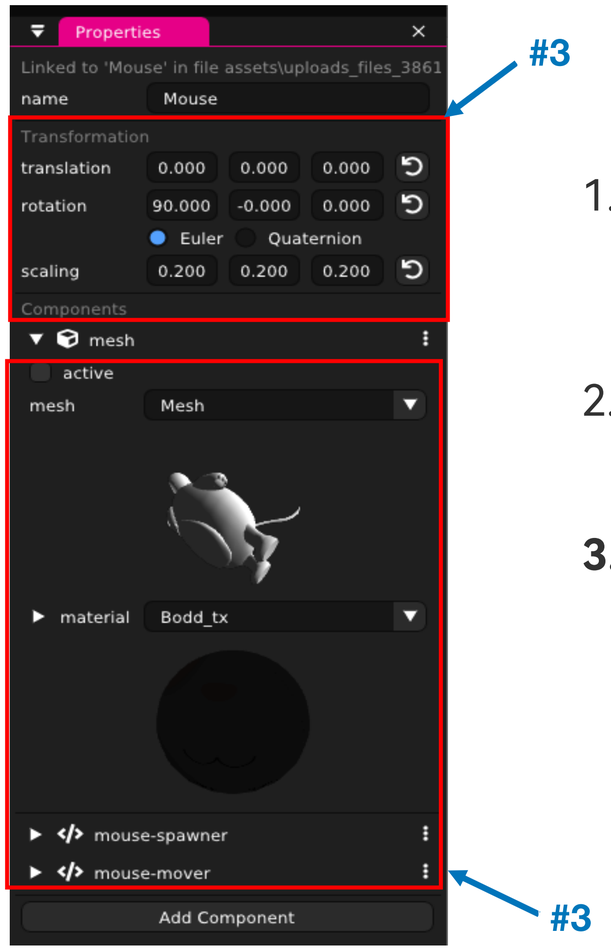
Deleting objects and JS components



1. To remove an object from the scene, right-click the object from the **Scene Outline** and select the **Delete option**.
2. Similarly, for a **JS component**, click the 3-dot menu on the component (from the Properties view) and select the **Delete option.**

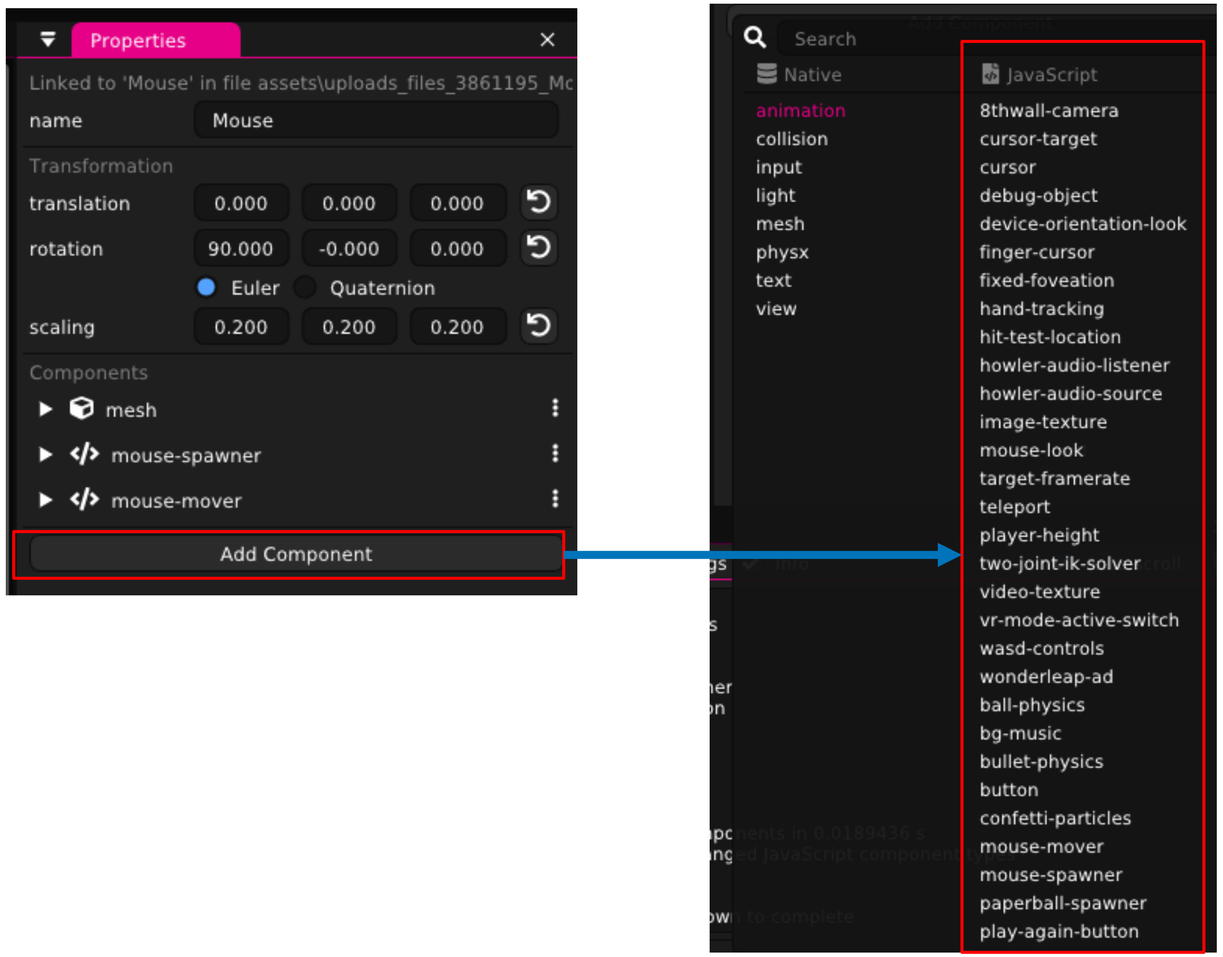
Modify objects





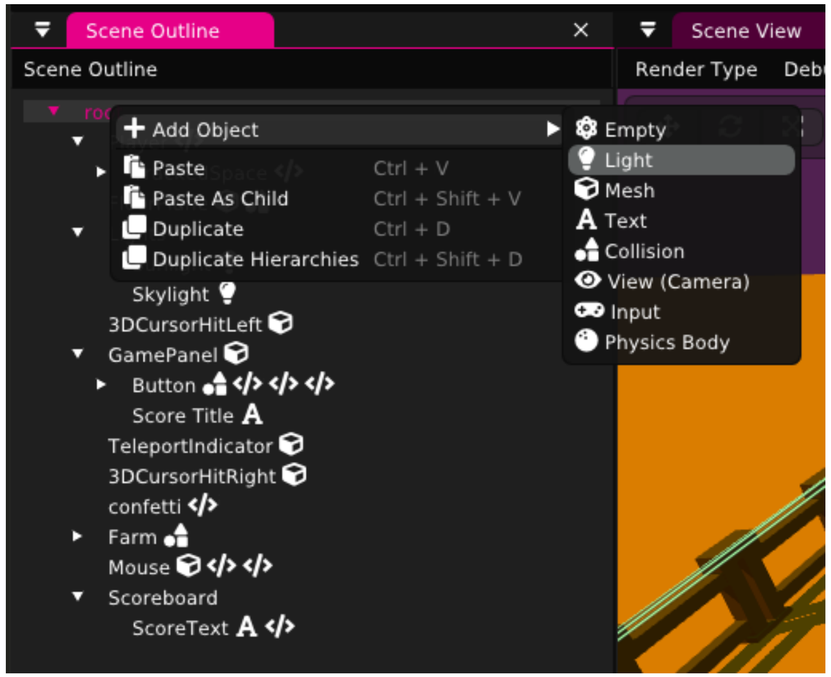
1. Objects can be modified directly in the **Scene View – Translation, rotation,** and **scaling** by **dragging the mouse/trackpad**.
2. Changes can also be made from the **Properties view** by inputting values for greater precision.
3. **Component values** can also be modified in the **Properties view**.

Adding JavaScript components to objects

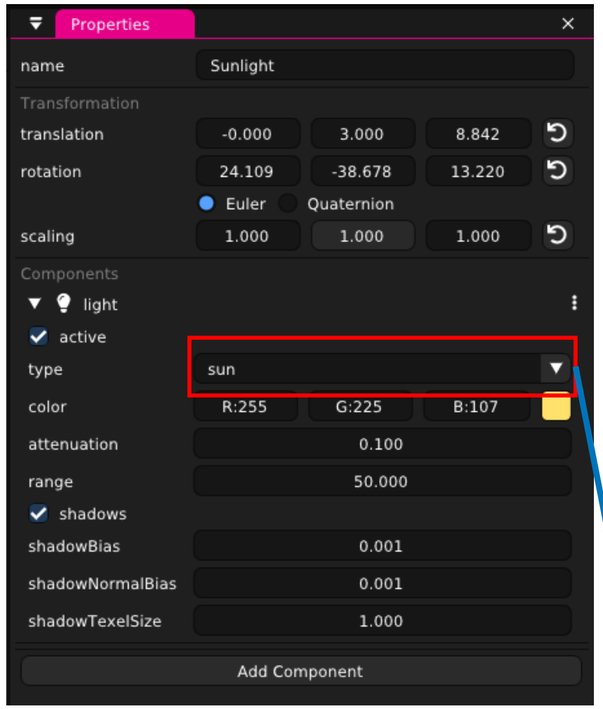


- **JS components** can be added to each object from the **Properties view**.
- Some of these components come with **Wonderland** (ex: 8thwall-camera, howler-audio-source) and the rest are **user defined** (ex: mouse-mover, mouse-spawner).

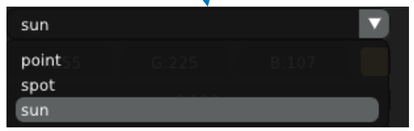
Lighting



#1



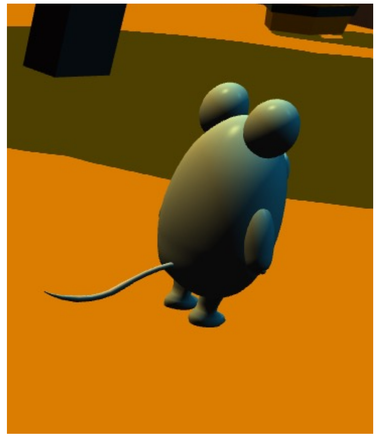
#2



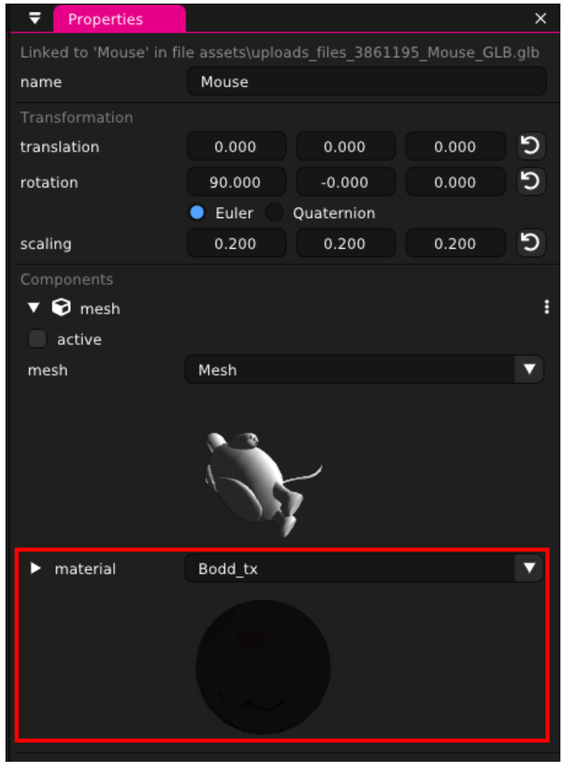
#3

1. From the **Scene Outline**, right-click **root > Add Object > Light** to create a new **Light object**.
2. In the **Properties view**, you can adjust the transformation, light intensity, color, and light type.
3. Expanding the type drop-down menu shows 3 different light types. Single **sun light** can illuminate the entire scene in comparison to **multiple-point lights** while having **better performance**.

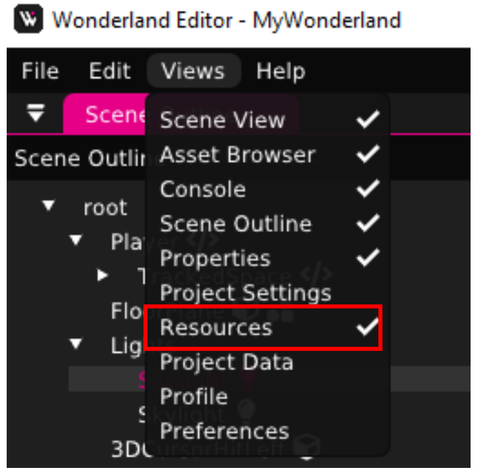
Enabling shadows



#1



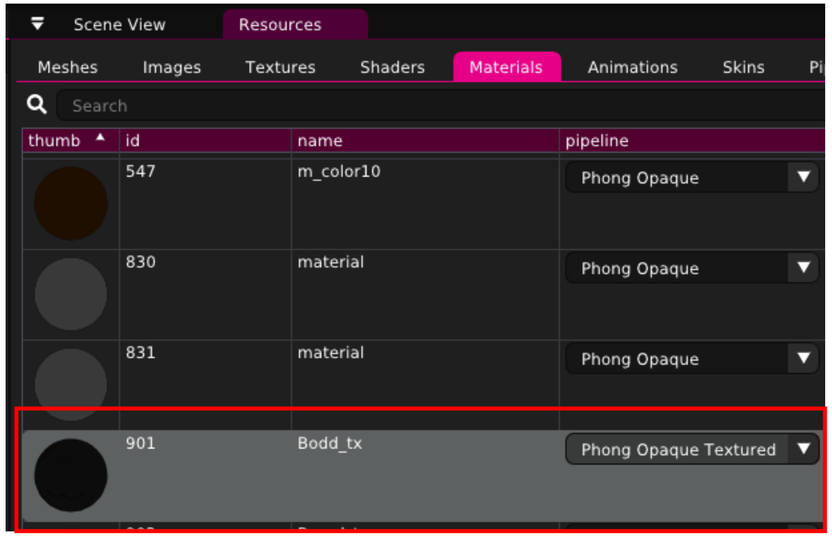
#2



#3

1. Some objects may not **cast shadows**.
2. Navigate to the Properties view of the object and check the material selection (under **Components > mesh > material**)
3. From the top left, open **Views > Resources**.

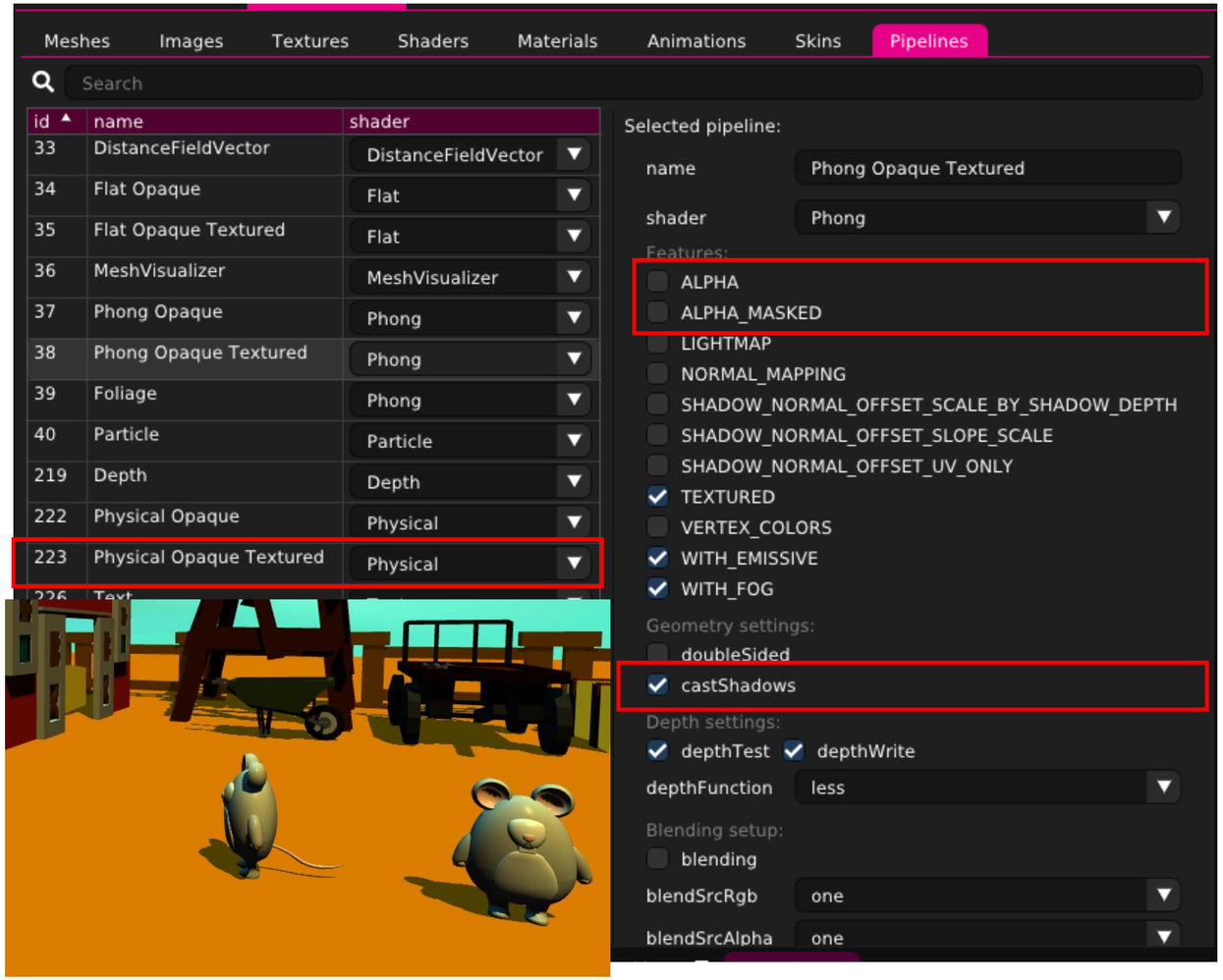
Enabling shadows (cont.)



#1

1. Check which **pipeline** correlates to the **material** of your 3D object.
2. Switch to the **Pipelines tab** (Views > Resources > Pipelines) and select your pipeline.
3. Enable Geometry settings > **castShadows**
4. Disable Features > **ALPHA** and **ALPHA_MASKED**

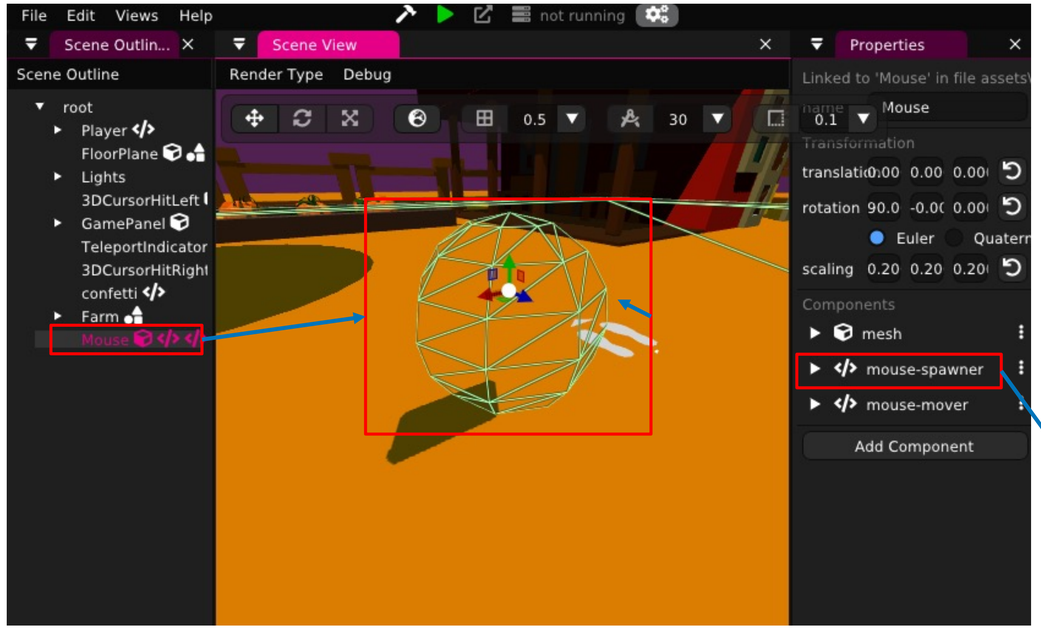
#2

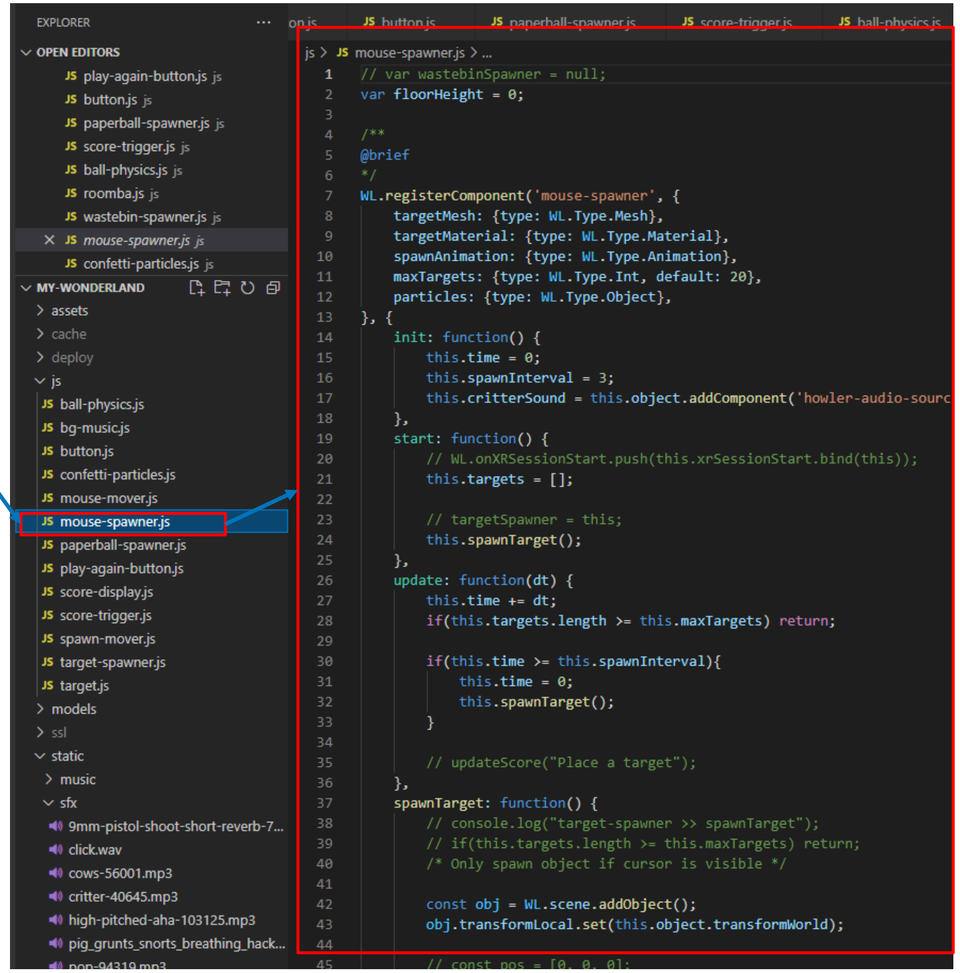


#4

#3

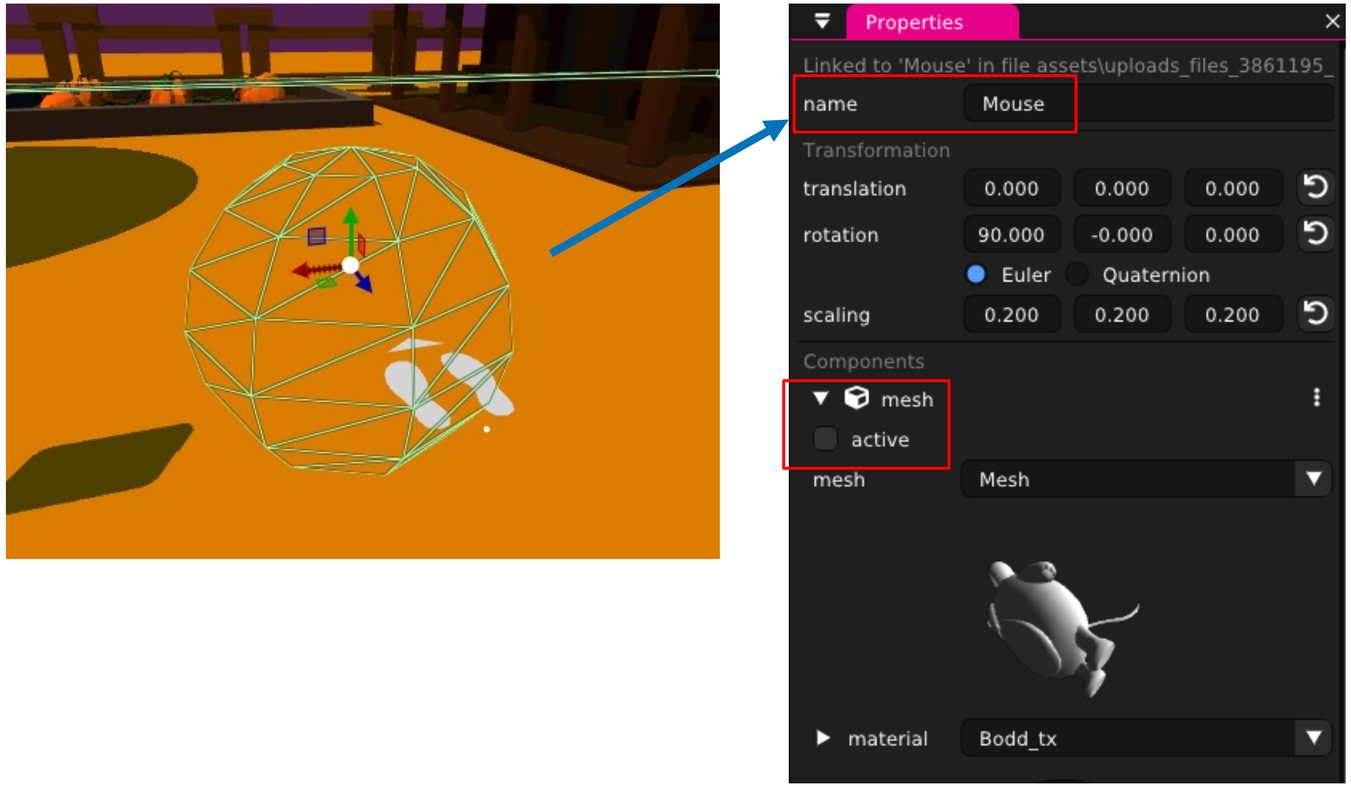
Apply Javascript logic to the scene





Use code editor like Visual Studio Code to add Javascript components and define the behavior of individual objects, such as movement, object spawning, adding/removing components, collision effects, etc.

JavaScript – Mouse Spawner



- The Mouse object in the scene is an **invisible object** that **spawns the actual in-game mice** that the player shoots at. You can see that the mesh object is not active, rendering it invisible.
- Other than the inactive mesh component, there are **two JS components** that control how the mice are spawned and how the spawner itself moves around the map: **mouse-spawner** and **mouse-mover**.

JavaScript – Mouse Spawner (cont.)

```
var floorHeight = 0;
var maxTargets = 0;
var mouseSound = null;

/**
@brief
*/
WL.registerComponent('mouse-spawner', {
    targetMesh: {type: WL.Type.Mesh},
    targetMaterial: {type: WL.Type.Material},
    spawnAnimation: {type: WL.Type.Animation},
    maxTargets: {type: WL.Type.Int, default: 20},
    particles: {type: WL.Type.Object},
}, {
    init: function() {
        maxTargets = this.maxTargets;
        this.time = 0;
        this.spawnInterval = 3;
        mouseSound = this.object.addComponent('howler-audio-source', {src: 'sfx/critter-40645.mp3', loop: true, volume: 1.0 });
    },
    start: function() {
        this.targets = [];
        this.spawnTarget();
    },
    update: function(dt) {
        this.time += dt;
        if(this.targets.length >= this.maxTargets) return;

        if(this.time >= this.spawnInterval){
            this.time = 0;
            this.spawnTarget();
        }
    },
```

#1 #2 #3 #4 #5

1. **Global variables** that can be accessed by **other JS files**.
2. Variables declared here can be accessed and changed from the **Properties View** in **Wonderland Editor**
3. **Init()** is the first function that is called. We set some vars here and initialize the mouse sound effect.
4. **Start()** is always called **after Init()** has completed. Here we call spawnTarget().
5. The **update()** function is called at **every frame update**. The mice are spawned at set time intervals with a maximum number of mice that can be spawned.

JavaScript – Mouse Spawner (cont.)

```
spawnTarget: function() {

#1  const obj = WL.scene.addObject();
    obj.transformLocal.set(this.object.transformWorld);

    obj.scale([0.1, 0.1, 0.1]);
    const mesh = obj.addComponent('mesh');
#2  mesh.mesh = this.targetMesh;
    mesh.material = this.targetMaterial;
    mesh.active = true;
    obj.addComponent("mouse-mover");

    if(this.spawnAnimation) {
        const anim = obj.addComponent('animation');
        anim.playCount = 1;
        anim.animation = this.spawnAnimation;
        anim.active = true;
        anim.play();
    }

    /* Add scoring trigger */
    const trigger = WL.scene.addObject(obj);
    const col = trigger.addComponent('collision');
    col.collider = WL.Collider.Sphere;
#3  col.extents[0] = 0.6;
    col.group = (1 << 0);
    col.active = true;
    trigger.translate([0, 0.4, 0]);
    trigger.addComponent('score-trigger', {
        particles: this.particles
    });

    obj.setDirty();

    this.targets.push(obj);
#4  mouseSound.play();
},
```

1. First we add a new object using **WL.scene.addObject(),** set it's position relative to the mouse-spawner object, and set the scale.
2. Next we add the **mesh component** and add the **mouse-mover JS component** (the same one that the mouse-spawner uses) to define it's movement behavior.
3. Here we create another object on top of the mouse and add the **collision** and **score-trigger**. The collision component allow us *to know if the mouse gets shot by the player*. Score-trigger *updates the score* and *triggers the confetti* when the mouse is shot.
4. Lastly, we **play the mouse sound effect** when it spawns in.

JavaScript – Projectile Spawner

#1

```
onTouchDown: function(e) {
    let curTime = Date.now();
    ballTime = Math.abs(curTime-this.lastTime);
    if(ballTime>50){
        const end = e.inputSource.gamepad.axes;

        const dir = [0, 0, 0];

        this.object.getComponent('cursor').cursorRayObject.getForward(dir);

        this.pulse(e.inputSource.gamepad);
        this.throw(dir);
    }
    this.lastTime=curTime;
    this.soundClick.play();
},
```

#2

```
onActivate: function() {
    if(WL.xrSession) {
        WL.xrSession.addEventListener('selectstart', this.onTouchDown.bind(this));
        WL.xrSession.addEventListener('selectend', this.onTouchUp.bind(this));
    }
},
xrSessionStart: function(session) {
    if(this.active) {
        session.addEventListener('selectstart', this.onTouchDown.bind(this));
        session.addEventListener('selectend', this.onTouchUp.bind(this));
    }
},
```

#3

```
pulse: function (gamepad) {
    let actuator;
    if (!gamepad || !gamepad.hapticActuators) { return; }
    actuator = gamepad.hapticActuators[0];
    if(!actuator) return;
    actuator.pulse(1, 100);
},
```

1. **onTouchDown()** is called when the *controller trigger is pressed down*. First we check if enough time has elapsed since the last time a projectile was spawned to limit the rounds per minute the player can shoot. Then we get the **forward vector of the cursorRayObject** that the paperball-spawner component is attached to get the **direction we shoot** in and plug that into the **throw()** function.
2. **onActivate()** and **xrSessionStart()** are used to bind the **selectstart** and **selectend** events to **onTouchDown** and **onTouchUp** (unused)
3. **Pulse()** vibrates the controller for better player haptics.

JavaScript – Projectile Spawner (cont.)

#1

```
throw: function(dir) {
    let paper =
        this.paperBalls.length == this.maxPapers ?
        this.paperBalls[this.nextIndex] : this.spawnBullet();
    this.paperBalls[this.nextIndex] = paper;

    this.nextIndex = (this.nextIndex + 1) % this.maxPapers;

    paper.object.transformLocal.set(this.object.transformWorld);
    paper.object.setDirty();
    paper.physics.dir.set(dir);

    paper.physics.scored = false;
    paper.physics.active = true;

    this.canThrow = false;
    setTimeout(function() {
        this.canThrow = true;
    }.bind(this), 1000);
},
```

#2

```
spawnBullet:function(){
    const obj = WL.scene.addObject();

    const mesh = obj.addComponent('mesh');
    mesh.mesh = this.paperballMesh;
    mesh.material = this.paperballMaterial;

    obj.scale([0.05,0.05,0.05]);

    mesh.active = true;

    const col = obj.addComponent('collision');
    col.shape = WL.Collider.Sphere;
    col.extents[0] = 0.05;
    col.group = (1 << 0);
    col.active = true;

    const physics = obj.addComponent('bullet-physics', {
        speed: this.ballSpeed,
    });
    physics.active = true;

    return {
        object: obj,
        physics: physics
    };
},
```

1. **Throw()** does a few checks before calling **spawnBullet()**
2. **spawnBullet()** adds a new object and applies the scaling as well as mesh, collision, and bullet-physics components.

JavaScript – Projectile trajectory control

#1 #2 #3 #4

```
update: function(dt) {
    //error checking?
    if(isNaN(dt)){
        console.log("dt is NaN");
        return;
    }

    //update position
    this.object.getTranslationWorld(this.position);
    //deactivate bullet if through the floor
    if(this.position[1] <= floorHeight + this.collision.extents[0]) {
        console.log("bullet penetrated floor >> "+this.position[1]+" <= "+floorHeight + this.collision.extents[0]
        + " ( " + floorHeight, ", ", this.collision.extents[0]," )");
        this.active = false;
        return;
    }
    //deactivate bullet if travel distance too far
    if(glMatrix.vec3.length(this.position)>175){
        this.active = false;
        return;
    }

    let newDir = [0,0,0];
    glMatrix.vec3.add(newDir, newDir, this.dir);
    glMatrix.vec3.scale(newDir, newDir, this.correctedSpeed);

    glMatrix.vec3.add(this.position, this.position, newDir);

    this.object.resetTranslation();
    this.object.translate(this.position);
},
```

1. Most of the work in **bullet-physics** is done in the **update()** function since we expect the ball to be constant motion.
2. If the **bullet** *passes through the floor*, we **deactivate** *the object to preserve performance*.
3. If the **bullet** *travels too far from the player* (beyond what they can see) we also **deactivate** it.
4. Otherwise the **bullet** continues traveling on a **linear path** from its point of origin at a **constant speed**.

JavaScript – Collision detection and scoring

#1 #2 #3 #4

```
WL.registerComponent('score-trigger', {
    particles: {type: WL.Type.Object}
}, {
    init: function() {
        this.collision = this.object.getComponent('collision');
        this.soundHit = this.object.addComponent('howler-audio-source', {src: 'sfx/high-pitched-aha-103125.mp3', volume: 1.9 });
        this.soundPop = this.object.addComponent('howler-audio-source', {src: 'sfx/pop-94319.mp3', volume: 1.9 });
        this.victoryMusic = this.object.addComponent('howler-audio-source', {src: 'music/level-win-6416.mp3', volume: 1.9 });
    },
    update: function(dt) {
        let overlaps = this.collision.queryOverlaps();

        for(let i = 0; i < overlaps.length; ++i) {
            let p = overlaps[i].object.getComponent('bullet-physics');

            if(p && !p.scored) {
                p.scored = true;
                this.particles.transformWorld.set(this.object.transformWorld);
                this.particles.getComponent('confetti-particles').burst();
                this.object.parent.destroy();

                ++score;

                let scoreString = "";
                if(maxTargets!=score){
                    scoreString = score+" rats down, "+(maxTargets-score)+" left";
                }else{
                    scoreString = "Congrats, you got all the rats!";
                    this.victoryMusic.play();
                    bgMusic.stop();
                    mouseSound.stop();
                }

                updateScore(scoreString);

                this.soundHit.play();
                this.soundPop.play();
            }
        }
    },
```

1. The **score-trigger** component is attached to every single **mouse** and will trigger when the bullet hits its collision component.
2. We initialize the **SFX audio** here, but don't play it yet.
3. We want the score-trigger to **check for collisions** as much as possible, so it's done here in the **update()** function.
4. If a **collision overlap** is detected and the object has **a bullet-physics** component, it means that the mouse was hit by the player. We trigger the confetti particles, destroy the mouse object, update the score message, and play a hit SFX. If all the **mice have been eliminated**, the score message changes accordingly, the background music and mouse SFX stops, and the win music plays.

JavaScript – SFX, music

```
/* Global function used to update the score display */
var updateScore = null;
var bgMusic = null;
/**
@brief Marks an object with text component as "score display"

The center top text object that shows various helpful tutorial
texts and the score.
*/
WL.registerComponent('bg-music', {
}, {
    init: function() {
        bgMusic = this.object.addComponent('howler-audio-source', {src: 'music/happy-funny-kids-111912.mp3', loop: true, volume:
        bgMusic.play();
        this.bgDucks = this.object.addComponent('howler-audio-source', {src: 'sfx/recording-ducks-binaural-18742.mp3', loop: tru
        this.bgDucks.play();
        this.bgCow = this.object.addComponent('howler-audio-source', {src: 'sfx/cows-56001.mp3', loop: true, volume: 1.0 });
        this.bgCow.play();
        this.bgSheep = this.object.addComponent('howler-audio-source', {src: 'sfx/sheep-23761.mp3', loop: true, volume: 1.0 });
        this.bgSheep.play();
        this.bgPig = this.object.addComponent('howler-audio-source', {src: 'sfx/pig_grunts_snorts_breathing_hackney_city_farm-73
        this.bgPig.play();
    },
});
```

- All **the background SFX and music** is done in **bg-music** which is attached to the **player object**.
- Initializing and playing sounds is **quite simple**.
- If other **JS components** *need to access this audio*, it just needs to be declared as **a global var**.

View and debug in Huawei VR Glass

1. Enable USB debugging on Huawei VR compatible Huawei phone.
2. Enable ADB over Wi-Fi and connect to Huawei phone.
3. Generate SSL key, certificate, and DH parameter files.
4. Wonderland Engine - navigate to Views > Preferences > Server, enable SSL server and enter the SSL files.
5. Repackage Wonderland project and launch the server.
6. Go to desktop Chrome browser, enter URL chrome://inspect#devices, click Port Forwarding button, add Port: 8081, IP address and port: localhost:8081. Check Enable port forwarding and click Done.
7. Put on Huawei VR Glass, launch either the Wolvic VR Browser and visit https://localhost:8081/index.html