Magical Cave Project

*What I implemented :*

* Cave Dome
* Volumetric Floor with Depression and Path
* Rock Ring
* Crystals
* Mushrooms
* Water Surface
* Water Ripples
* Falling Water Drops
* Water reflection and distortion
* Fairies
* Sword
* Pedestal
* Torches
* Audio system

How it works overall :   
1. CPU side : geometry generation (JS) – create vertex data, assign transforms, calculate UVs, normals, colors.   
2. Send data to GPU : buffers   
3. GPU side : shaders   
 - vertex shader : receives one vertex at a time, transforms to screen space  
 - fragment shader : computes pixel color for each triangle fragment   
4. Set uniforms before drawing   
5. Drawing : once buffers, shaders and uniforms are set   
- **Each vertex from buffer goes to vert shader**, which applies transforms, computes final screen-space position (gl\_Position) and outputs a list of transformed matrices  
**- GPU uses index buffer** to assemble indices into triangles which become our geometric primitives  
**- Rasterization** : we convert those triangles into a grid of pixels : fragments.  
(a fragment is a proposal for a pixel, not all fragments become pixels : all pixels are fragments but not all fragments become pixels)  
**- Run fragment shader** : GPU runs it once for each fragment.   
**- Write to Framebuffer** : after depth testing and blending, write output of frag into screen’s framebuffer

**General structure for procedural 3D mesh generation :**  
1. Decide shape and structure (surface, volume, radial object // grid, rings, polar sampling)  
2. Define geometry resolution (hm samples or steps)  
3. Generate vertex positions (vec3 positions), which can be stored in 2D grid, or pushed  
4. Generate UV coordinates (decide how mesh maps to texture space – planar projection, spherical/cylindrical mapping, triplanar projection)  
5. Triangulate the surface – push to indices : gives actual renderable faces to WebGL  
6. Add extra layers if needed

Cave Dome, Floor and Rock Ring

To build the **dome mesh** : generateCaveDomeMesh  
I built a hemisphere using spherical coordinates (phi, theta), scaled it non uniformely, and added procedural deformation in the vertex shader :  
- small radial bumps using radialNoise   
- 4 large bumps using uBumpCenters, uBumpHeights, uBumpRadii (generated randomly)  
Bottom ring vertices are stored in domeBaseRing to use when generating the floor.  
We call generateCaveDomeMesh in setupCaveDomeBuffers, and send pos, uv, indices, vertexcount.   
Finalize in drawCave.

For the **floor** : generateVolumetricFloorWithDepression.   
We use as inputs the height (thickness of floor), the radialSteps (number of concentric rings), the depressionRadius, the depressionDepth, and the uvScale   
It’s build from concentric rings of vertices.   
The noise is implemented in the JS.   
Finalize in drawFloor.

For the **rock ring** : base mesh generation for the rocks in generateRockMesh, stored in a GPU buffer and reused for every rock in the rock ring. Access outer ring from generateVolumetricFloorWithDepression. In setupFloorBuffers we loop over the surface ring and place rocks. Store that in rockInstances. Also there, we generate clusters of crystals to position between rocks. Same thing for mushrooms.   
Then we finalize in drawRocks : loop over rockInstances, build rock matrices, send to GPU + crystal light logic.

Crystals and Mushrooms

To **build the mushrooms** : generateMushroomGeometry. We build a procedural mushroom mesh where the stem is a vertical cylinder and the cap is a hemisphere. This function returns the pos, uv, index, vertex count and cap start index, used for rendering.   
We use setupMushroomBuffers, which calls generateMushroomGeometry, uploads pos, uv and index data to GPU via createMeshBuffers, and stores results in mushroomBuffers.  
We then use setupMushroomShaders, where we compile and link the vert and frag , and set attributes and uniforms.  
We place them in the scene in setupFloorBuffers. Then, we render in drawMushrooms (loops over mushrooms defined in setupFloorBuffers, applies model matrix, binds mushroomBuffers and draws each element).

For the **crystals** : we define the geometry : either pointy or rock. Pointy ones use generatePointyCrystalsGeometry , rock ones call generateRockMesh.   
We have setupCrystalBuffers, that stores the geometry in pointyCrystalBuffers and in rockCrystalBuffers.  
We have setupCrystalShaders where we set attributes and uniforms.   
To place them in the scene, we use :  
- generatePedestalCrystals : 8 rock crystals around pedestal.  
- generateDomeCrystals(count) : samples random triangles from dome, uses generateCluster to place groups of pointy crystals, and add their tips positions to dropSources for water drop system. In the lower dome, we add rock crystals directly.   
- setupFloorBuffers : we place the water floor crystals, and the rock ring crystals.  
We draw crystals in drawCrystals.

Water surface, water ripples, falling drops

The **water mesh** is created by generateWaterPlane(res, radius, y) – generates a flat grid mesh representing the water surface (2D grid of mass points). It stores position (vertex coords), uv, triangle indices, gridSize, count of total vertices, radius. The geometry is passed to setupWater() which creates GPU buffers and binds attributes for rendering.  
  
The **drops** are spawned every drop interval of 0.5s in updateDrops(time). They are chosen randomly from dropSources. Each drop has a mass, a position, and a velocity. Gravity updates pos/vel over time. When y goes below water height, the drop triggers injectRippleAtPosition(drop.position), and is then removed. Drops fall with **Newton’s second law.**  
**Ripples** are updated every frame by updateRipples(dt) and updateRippleNormals .  
They’re triggered by falling drops or by sword extraction. The normals are used for distortion.   
We use **Eulerian mass-spring** surface model. Basically, the water surface is a **2D grid of mass points, each connected to its four direct neighbors, via virtual springs**. Each point has a **vertical displacement** (rippleHeights[i]) and a **velocity** (rippleVelocities[i]). Each point is affected by its neighbors by a **Laplacian operator**.   
Numerical integration of **wave equation with damping** :   
Immagine che contiene Carattere, calligrafia, tipografia, numero

Il contenuto generato dall'IA potrebbe non essere corretto.

We discretize that and use a system of ODE of 1st order (height and velocity).  
In updateRipples, we have Laplacian = h\_left + h\_right + h\_down + h\_up - 4\* h\_center , which approximates the Laplacian. It models hm the point is pulled towards its neighbors, like springs pulling a mass back to equilibrium. (spring potential energy)  
We then use **Newton’s law** :   
Immagine che contiene Carattere, schermata, nero, Elementi grafici

Il contenuto generato dall'IA potrebbe non essere corretto.

nextVelocities[i] = velocities[i] + (k\* Laplacian – damping \* velocities[i]) \* dt   
where k is the spring stiffness and damping coefficient is resistance.   
This is in updateRipples too.   
We also update the heights, by basic kinematic integration :  
Immagine che contiene Carattere, tipografia, design

Il contenuto generato dall'IA potrebbe non essere corretto.

The drops falling trigger ripple injection , like throwing a stone, it adds energy locally.   
rippleVelocities[i] += 0.8 \* envelope injects velocity spike (kinetic energy).

Water reflection and texture distortion

We render the scene from a mirrored camera into a reflection texture. We simulate ripple normals from wave heights. We encode ripple normals into a distortion map texture. In the frag shader of the water, we sample screen-space reflection using gl\_FragCoord, apply distortion from ripple normals, and blend with water tint, crystal and torch lights.

We first init setupReflectionFramebuffer with 2 textures for the ping-pong effect.   
drawReflectionScene computes the mirrored camera view , flipped vertically across the water plane, and draws everything except the water into the reflection texture.   
The result is stored in one of the reflection textures, used in the main render pass.

updateRippleNormals computes the slope of the water height at each point, encodes normal, normalizes and stores in rippleNormals. These normals are packed into a texture, to use in the fragment shader.

Fairies / navi system

The Navi system adds 7 animated magical fairies to the scene. Each fairy orbits around a center (initially near the cave center), flaps its wings, glows over time, and changes its speed, height, and orbit radius dynamically when the sword is extracted.

We load a pair of navis using loadFairyPair – parses obj and extracts positions for open and close wings, and object tag to determine whether it’s the center or the wings.   
These arrays are passed to createFairyVAO , to build 2 VAOs : fairyBuffers.center and fairyBuffers.wings . Each vertex has aClosed, aOpen, aIsCenter (1 for sphere, 0 for wings).   
We have the setupFairyShader that takes in these attributes. The vert linearly interpolates the wing positions between aClosed and aOpen, using uFlap.   
const flap = 0.5 + 0.5 \* Math.sin(t \* 6.0);  
gl.uniform1f(fairyProgram.uFlap, flap); --- flap smoothy oscillates from 0 to 1

Vert shader outputs vIsCenter to frag shader. The frag shader applies the color and soft pulse.   
fairyFlapTime is 60 fps.

Each fairy orbits around a center point using a circular path.  
y is modulated by a sinusoidal oscillation.   
fairyAttraction (updated in drawScene when sword rises) determines how strongly the fairies are pulled towards the sword. The radius of the trajectory diminishes and the speed augments. Everything is handled in drawAllNavis .

Sword

The sword is loaded via loadSwordObj. It parses the obj and mtl to group materials and store vertices per group. Position and uvs are extracted. For each material group, a vao (vertex array object, stores entire vertex setup once, we don’t need to rebuild buffer and rebind attributes every frame, we bindVertexArray and draw) is created with position and uv buffers. Each group stores textureKey, metallicRoughnessKey and normalMapKey, and fallback color.   
These are stored in swordBuffers.  
We use a PBR shader : physically based rendering , uses roughness and metallic channels (mr.g and mr.b). Normal map adds details.  
Specular highlights computed using Blinn-Phong halfway vector (vettore halfway tra direzione luce e direzione sguardo - BP : coseno angolo tra halfway e normale).   
Fresnel effect increases reflectivity at glancing angle, it generates glimmer on metal. We add a pale blue tint to enhance metallic blade.   
swordState == 0 : starts extraction ; swordState == 2 : starts retraction   
Sword animation in drawScene and sword drawing in drawSword.  
Rotates over time when extracting or extracted with swordRotation.   
PBR adapts lighting with torch and crystal lights + Fresnel, specular, glint and magic tint.

Torches and lighting

The torch tip is the highest vertex of the torch. Each torch light adds diffuse lighting , attenuation based on distance, flicker effect. They are point lights. I manually calculate light at each pixel in the shaders, using the surface normal N and the vector from the surface point to light LThe diffuse lighting formula is Lambertian : float diff = max(dot(N, L), 0.0); --- dot(N, L) is the cos of the angle between the normal and the light direction. Sometimes I set a min brightness, and sometimes I set float diff = 1 to illuminate everything equally. Fallback : ambient light modulated by ao (which darkens crevices) : vec3(0.7) \* ao.