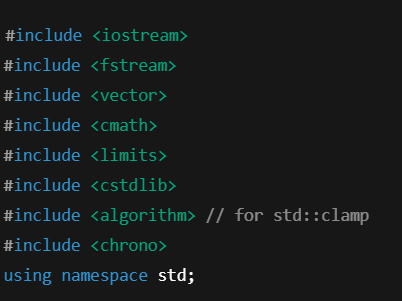
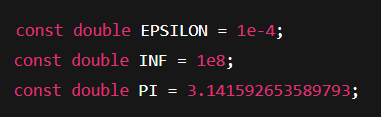
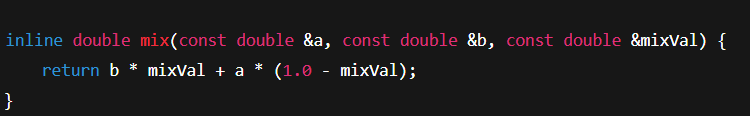
1. **Headers / constants**



* **What:** include standard libraries for IO, containers, math, numeric limits, random, clamp, and timing.
* **Why:** each header provides types/functions used later (e.g., std::vector, sqrt, std::clamp, chrono::high\_resolution\_clock).
* **If you change/remove:** removing one will cause compile errors where those functions/types are used. Replacing using namespace std; with explicit std:: is safer style.

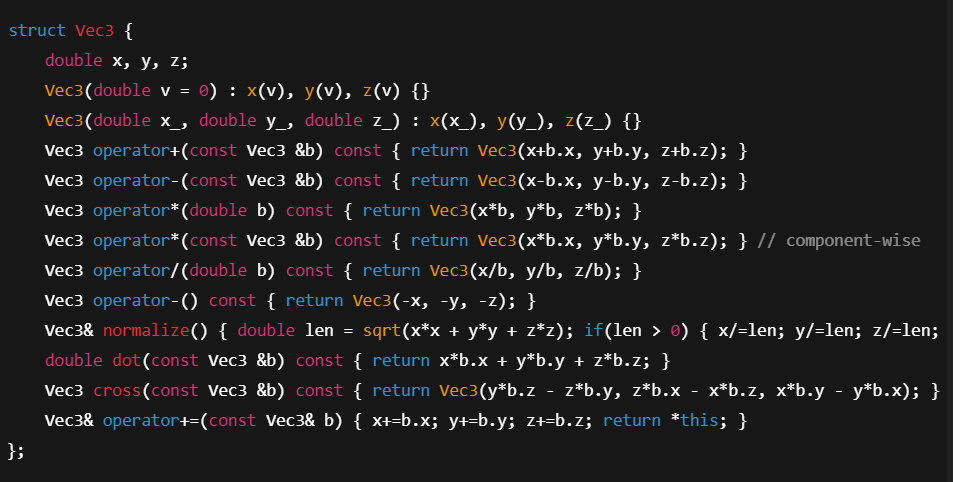


* **What:** small constants used throughout.
* **Why:** EPSILON offsets ray origins to avoid self-intersection ("shadow acne"), INF a large distance value, PI for field-of-view math.
* **Change effects:**
  + Increase EPSILON too much → shadows/grazing hits will appear detached (objects look separated from themselves). Decrease to 0 → self-intersection artifacts.
  + Change INF too small may incorrectly treat far objects as closer; usually keep large.
  + Changing PI is meaningless unless you want a different circle constant.



* **What:** linear interpolation between a and b.
* **Why:** used to blend values (e.g. Fresnel facing ratio hack).
* **Change:** altering implementation changes blending. If you swap order, results flip.

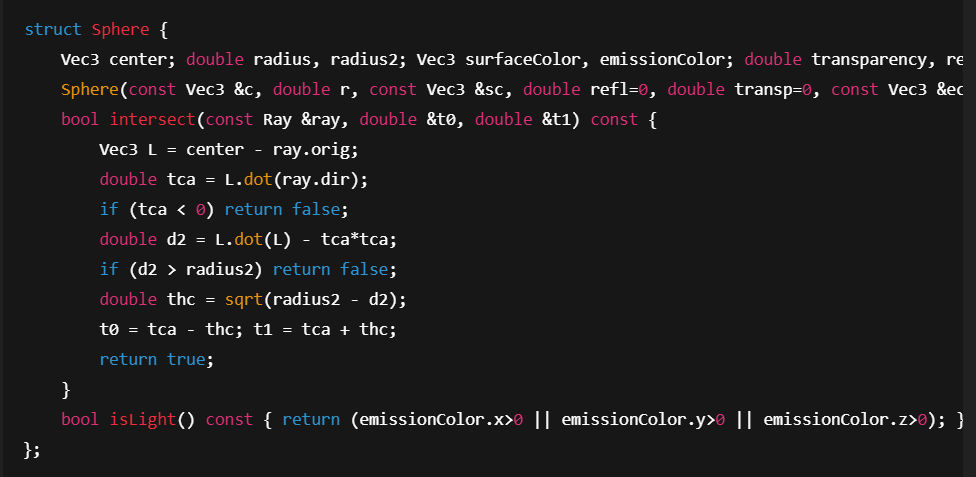
**2) Vec3 (3D vector / color) struct**

* **What:** vector arithmetic + color operations.
* **Why:** all geometry, rays, normals, colors are Vec3.
* **Key lines & changes:**
  + operator\*(double) scales vector. Changing it affects all scalar multiplies (lighting intensity, ray lengths).
  + operator\*(Vec3) is component-wise multiply; used for color modulation (color \* lightColor). Removing it forces code changes.
  + normalize() makes a vector unit length — necessary for correct dot products and reflection math. If you forget to normalize directions, lighting and intersections behave incorrectly.
  + operator+= used to accumulate colors; missing it caused compile errors earlier.
* **If you add methods:** e.g. clamp, length, toInt it's helpful.

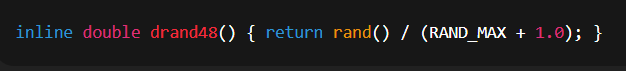
**3) Ray struct**

* **What:** a ray = origin + direction.
* **Why:** primary, reflection, shadow rays are represented by this.
* **Change:** if dir is not normalized when used, dot products/angles aren't correct (but many routines normalize the direction before use).

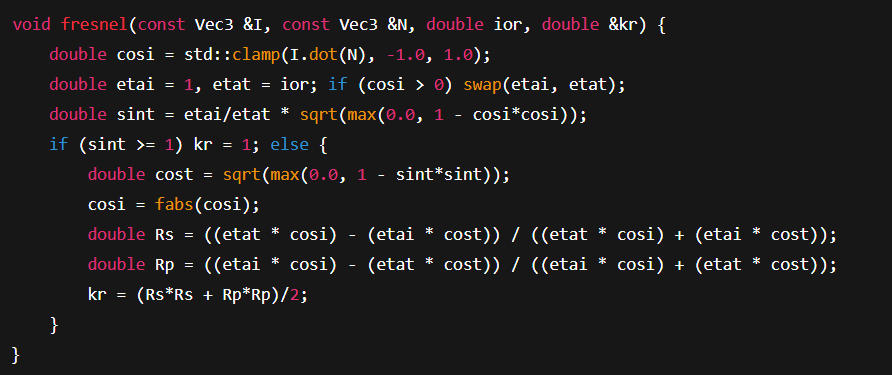
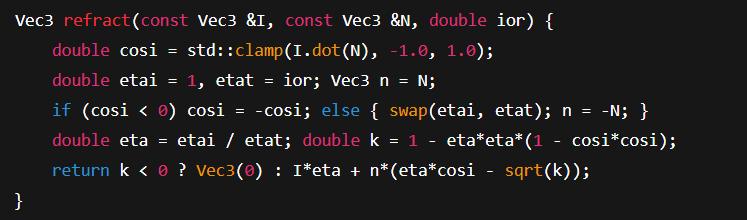
**4) Sphere struct & intersect**

* **What:** sphere data and a geometric ray-sphere intersection.
* **Why:** intersection returns two t values along ray where it enters/exits sphere.
* **Key behavior:**
  + tca < 0 check: if projection of sphere center onto ray is behind origin, it returns false — this is a conservative early-out and assumes origin is outside; it's okay for camera rays but if you cast rays from inside a sphere you must handle that (the code sometimes fixes negative t0 later).
  + d2 > radius2 -> ray misses.
  + t0, t1 are distances along the ray dir.
* **If you change equation:** You could implement algebraic quadratic solution — more robust when origin inside sphere.
* **Changing radius** alters sphere size; modify surfaceColor/emissionColor to change appearance and light role.

**5) drand48() helper**

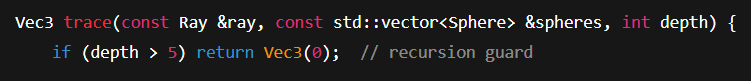
* **What:** random float in [0,1).
* **Why:** used if you add stochastic effects (anti-aliasing, Monte Carlo). You replaced drand48 (POSIX) with portable rand().
* **Change:** rand() has lower quality; consider <random> and std::mt19937 for reproducible better RNG.

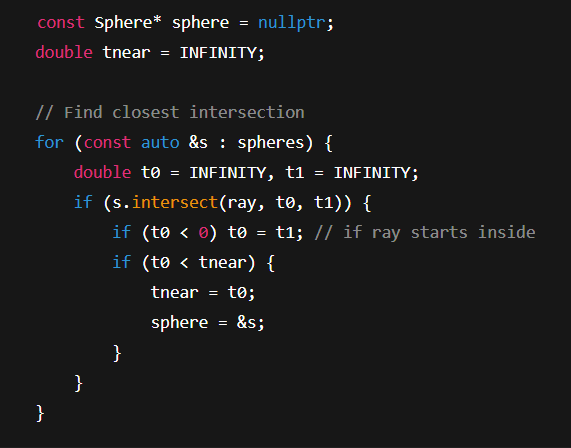
**6) Fresnel & Refraction helpers**

* **What:** computes reflection coefficient kr using Fresnel equations (exact, not Schlick).
* **Why:** determines fraction of light reflected vs. refracted for dielectric (glass) materials.
* **If you change ior:** Higher ior (index of refraction) makes reflectivity increase at grazing angles and change refraction bend (glass vs water).
* **If you replace formula with Schlick approximation:** cheaper, visually plausible.
* **What:** computes the refracted direction using Snell’s law; returns a zero vector on total internal reflection (TIR).
* **Why:** to spawn transmission rays through transparent objects.
* **Change effects:**
  + If k < 0 indicates TIR; returning zero signals no refraction.
  + Using zero vector if used unguarded will produce invalid rays — ensure caller checks length before tracing further.

**7) trace() — the core recursive ray tracer**

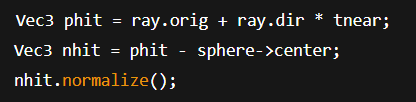
This is the most important function. I’ll break it into blocks:

* **What:** stops recursion (reflection/refraction depth).
* **Change:** increasing 5 → more bounces, more realism and more time; decreasing → faster but less accurate reflections/transparency.

**What:** brute-force nearest-hit (O(N) per ray).

* **Why:** you need the closest surface visible along the ray.
* **Changes:**
  + If you remove if (t0 < 0) t0 = t1, rays starting inside an object might select incorrect intersection (or none).
  + Replacing with BVH/KD-tree → huge speed up for big scenes.
* **What:** background sky color if no hit.
* **Change:** different color or gradient for sky; returning black makes emptiness black.

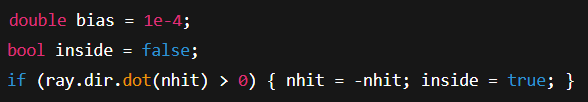
Compute hit point & normal:



**What:** phit is 3D intersection; nhit is normal.

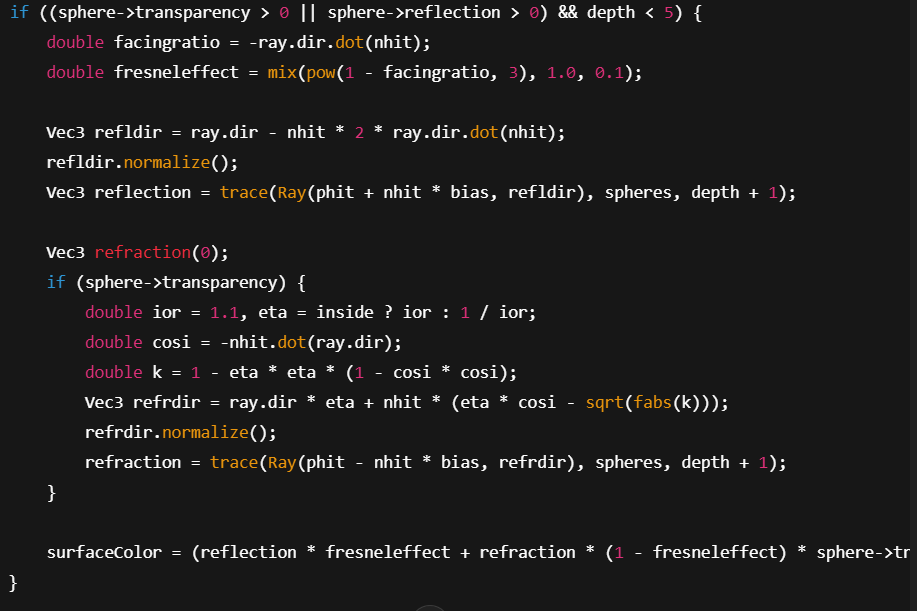
* **Issue:** for non-spheres you'd compute normal differently.
* **Change:** if you forget normalize() subsequent dot products give wrong results.

Inside/outside & bias:

**What:** if ray direction and normal point the same way, ray came from inside; invert normal and mark inside.

* **Bias:** used to offset secondary rays away from surface to prevent self-intersection.
* **Change:** making bias bigger prevents acne but produces "floating" shadow edges.

Reflection/refraction branch:

* **What:** computes reflection & optional refraction recursively and mixes them (simple Fresnel mix implementation).
* **Key lines:**
  + refldir reflection vector: formula R = I - 2\*(I·N)\*N.
  + trace(... depth + 1): recursively evaluate reflected/refracted rays.
* **Change effects:**
  + Increasing ior changes refraction bending and Fresnel mix.
  + Removing normalize() corrupts vector lengths and lighting.
  + Not offsetting with bias will cause self-intersections (acne).
  + If sphere->transparency is 0, no refraction computed.
* **Performance:** recursion increases runtime; limit depth accordingly.

Diffuse/shadow branch:

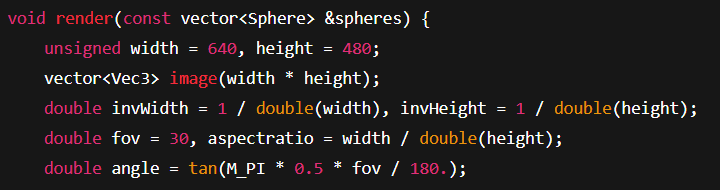
* **What:** single point light, shadow test (casts shadow ray toward light), then Lambert diffuse shading.
* **Change effects:**
  + Moving lightPos changes where shadows/highlights are.
  + Changing 0.8 scales diffuse contribution.
  + Not skipping the light in the shadow loop causes incorrect self-shadowing (we previously had a bug when the light is also in spheres vector).
  + Using multiple lights or area lights introduces more realistic lighting.

Ambient term:



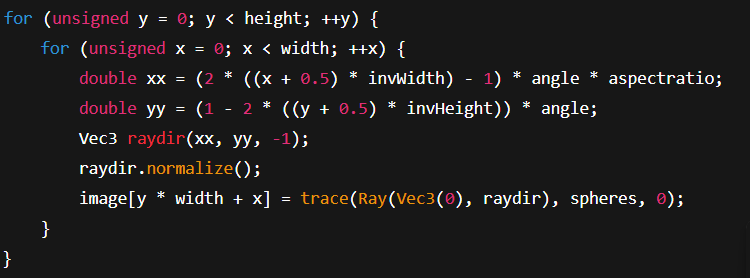
* **What:** small ambient contribution to avoid total black in shadow.
* **Change:** altering constant changes base brightness of shadows.

**8) render() — camera, pixel loop & image write**



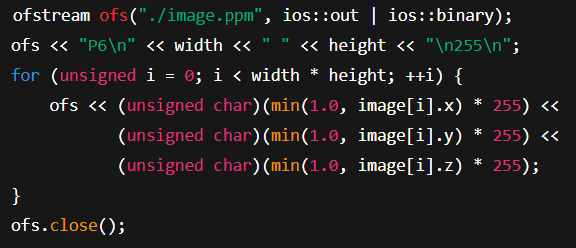
* **What:** sets image resolution, computes perspective projection parameters.
* **Change effects:**
  + Increasing width/height increases image resolution and exponentially increases render time (proportional to number of pixels × rays per pixel).
  + Change fov (field of view) — larger FOV makes scene wider (more distortion), smaller FOV zooms in.

Pixel loops:

**What:** for each pixel, compute primary ray direction through pixel center (pinhole camera), normalize it, trace and store color.

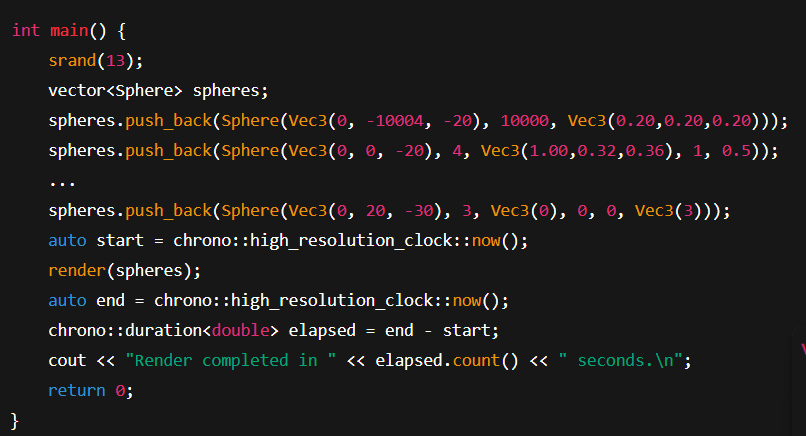
* **Change effects:**
  + Using (x + 0.5) centers the ray through the pixel; changing to (x) or (x + rand()) affects aliasing.
  + Replacing -1 with another z coordinate moves virtual image plane — consistent change to focal length.
  + Adding jitter (supersampling) reduces aliasing but multiplies trace calls.

Write PPM:

**What:** writes the image as binary PPM (P6).

* **Change effects:**
  + Use gamma-correction before mapping to byte for correct brightness (you asked earlier).
  + Writing PNG instead requires linking a library (stb\_image\_write.h is easiest single-file option).

**9) main() — scene setup and timing**

* **What:** seeds RNG, builds scene (ground, spheres, light as emissive sphere), times render.
* **Change effects:**
  + srand(13) — deterministic seed for reproducible randomness. Use different seed for varied results.
  + Changing sphere params (center, radius, colors, transparency/reflection) alters scene composition and visual output.
  + The light sphere uses emissionColor=Vec3(3) — intensity >1 lets lighting be stronger. Changing this affects scene brightness and visibility of reflections.
  + The ground sphere is a large sphere to approximate a plane — changing radius/position moves the floor.

**Quick checklist: common edits and their effects**

* **Increase width/height**: better detail, longer render time (linear in pixels).
* **Increase MAX\_DEPTH or depth limit**: more bounces → better reflections/refractions, longer time.
* **Change EPSILON**: bigger reduces acne but introduces visible gaps; smaller may reintroduce self-shadowing.
* **Change fov**: wide-angle = more scene seen, more perspective distortion.
* **Change lightPos or emissionColor**: shifts highlights/shadows and overall illumination.
* **Set sphere reflection/transparency values**: 0 → diffuse; increasing reflection makes mirror-like; transparency controls refraction.
* **Replace tca < 0 early-out** in intersect() with algebraic quadratic solver: more robust when origin is inside or near-surface.
* **Add BVH**: replace brute-force loop in trace() with acceleration structure to massively speed up scenes with many primitives.
* **Add anti-aliasing (supersampling)**: average several jittered samples per pixel — smoother edges, more time.

**Safety tips & gotchas**

* **Segfaults** usually come from:
  + Using a direction vector of length zero (e.g., refract returned zero and you don't check).
  + Dereferencing sphere when nullptr — your code checks this; keep that guard.
  + Unbounded recursion — ensure the depth guard is present.
* **NaNs**: sqrt(negative) leads to NaN — always max(0.0, value) before sqrt.
* **Performance**: the code is O(pixels × avg\_rays × objects). Profiling will show intersection tests dominate.

**Final suggestions for experiments (with expected effects)**

1. **Add gamma-correction** before writing: image looks more natural (brighter midtones).
2. **Try 2×2 supersampling** (4 jittered rays per pixel): better edge quality (4× slower).
3. **Swap Fresnel exact for Schlick**: kr = R0 + (1-R0)\*pow(1 - cosθ, 5) — much cheaper, similar look.
4. **Turn off shadows**: remove shadow loop → brighter, faster, but unrealistic.
5. **Add a BVH**: large performance gain if you expand scene from 5 spheres to thousands.