NEON BLASTER X

How to Build a Commercial-Grade Browser Game

A Full Developer Guide | HTML5 Canvas + Vanilla JavaScript

TLD Productions LLC

NEON BLASTER Made By TLD Productions LLC

Introduction

This guide walks through exactly how Neon Blaster X was built, from a blank HTML file to a fully playable commercial-style arcade game. We cover every system: the game loop, rendering, player movement, weapons, enemies, bosses, audio, UI, and polish. Everything runs in the browser using only HTML5 Canvas and vanilla JavaScript. No frameworks, no game engine, no dependencies.

If you can open a text editor and a browser, you can follow along. We explain not just what the code does but why it was written that way.

> **NOTE:** All code in this guide is taken directly from Neon Blaster X. You can open the game file alongside this guide and see exactly where each piece fits.

What We Are Building

Neon Blaster X is a top-down arcade space shooter with:

- A mouse-controlled ship with trail effects and engine glow
- 4 weapon types: Laser, Spread, Plasma, Homing Missiles
- 3 enemy types: Asteroids that split, Drones that chase and shoot, Seekers that home in
- 3 unique bosses with multiple attack phases
- A shield system and a rechargeable Nova Bomb special move
- Score combos, power-ups, wave progression, and a local leaderboard
- Procedural synthesizer music and sound effects using the Web Audio API
- A full menu system with settings, pause, game over, and how-to-play screens

The whole thing is a single HTML file. Open it in any modern browser on Windows, Mac, or Linux.

Chapter 1: Project Structure and the Game Loop

The Single File Approach

Big game engines want you to split your code across many files and folders. For a project this size, a single HTML file is cleaner and easier to share. CSS goes in a style tag, JavaScript goes in a script tag, and HTML handles the UI overlays. The canvas handles all the actual gameplay rendering.

Here is the basic skeleton of the file:

```
<!DOCTYPE html>
<html>
<head>
  <style>
    /* All CSS here: screens, HUD, buttons, fonts */
  </style>
</head>
<body>
  <div id="W">                  <!-- main wrapper, 900x650 -->
    <canvas id="c"></canvas>    <!-- gameplay rendering -->
    <!-- HTML overlay screens: menu, pause, game over, etc -->
  </div>
  <script>
    // All game logic here
  </script>
</body>
</html>
```

The Game Loop

Every game needs a loop that runs every frame. The browser gives us requestAnimationFrame for this. It calls your function roughly 60 times per second, passing a timestamp. From two consecutive timestamps you get delta time (dt), which is how many milliseconds have passed since the last frame.

Delta time is important. Without it, your game runs faster on a 144Hz monitor than a 60Hz one. By multiplying all movement by dt, you make the game frame-rate independent.

```
let lastT = 0;

function loop(ts) {
  const dt = Math.min(ts - lastT, 50);  // cap at 50ms to avoid spiral of death
  lastT = ts;

  ctx.clearRect(0, 0, W, H);            // wipe canvas
  ctx.save();
  applyShake();                          // optional screen shake offset

  drawGrid();
  drawStars();
```

```
  if (GS === "playing") {
    updateShip(dt);
    updateBullets(dt);
    updateAsteroids(dt);
    updateEnemies(dt);
    updateParticles(dt);
    updatePowerUps(dt);
    if (boss) updateBoss(dt);
    checkCollisions();
    checkWave(dt);
  }

  drawParticles();
  drawBullets();
  drawShip();
  drawAsteroids();
  drawEnemies();
  if (boss) drawBoss();

  ctx.restore();
  requestAnimationFrame(loop);
}

requestAnimationFrame(ts => { lastT = ts; loop(ts); });
```

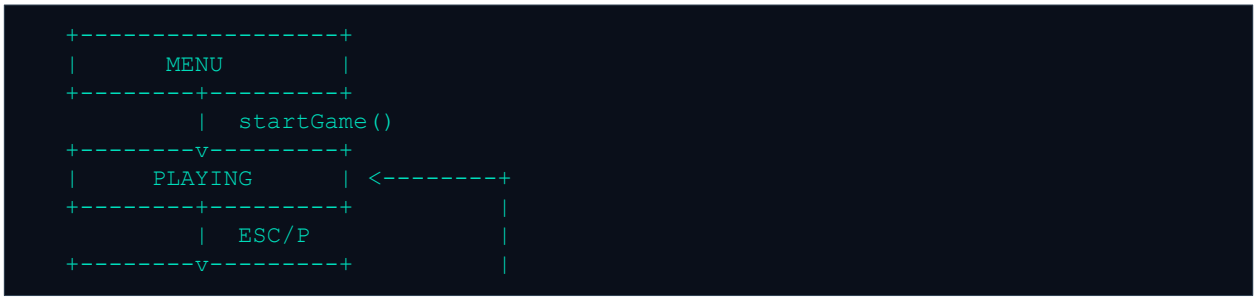
NOTE: We cap dt at 50ms. If the browser tab is hidden or the device stalls, dt could be huge. Without the cap, objects would teleport across the screen when the game resumes.

Game State

We track what mode the game is in using a single string variable GS. All the update calls inside the loop check this. When GS is not "playing", entities freeze in place.

```
let GS = "menu";    // "menu" | "playing" | "paused" | "dead"
```

Switching screens is then just changing GS and showing or hiding HTML overlay divs. The canvas keeps rendering in the background, which means the asteroids and stars still drift behind the pause screen. That is a nice free visual effect.



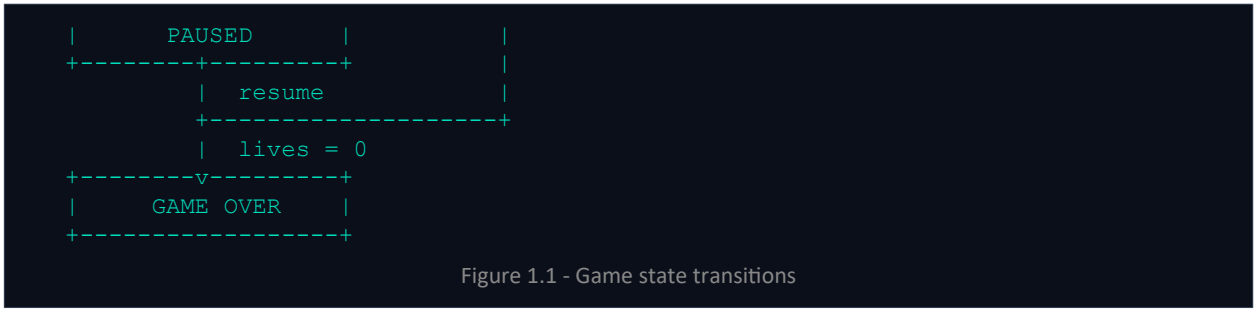


Figure 1.1 - Game state transitions

Chapter 2: Rendering with Canvas

Setting Up the Canvas

The HTML canvas element is just a bitmap you draw to with JavaScript. You get a 2D context from it and then call drawing commands. Nothing is retained between frames so you clear it at the start of each loop.

```
const cv = document.getElementById("c");
const ctx = cv.getContext("2d");
const W = 900, H = 650;  // fixed game resolution
```

We fix the canvas at 900x650 and center it with CSS flexbox. This keeps the game consistent across different screen sizes. On bigger screens there is just a dark border around the game area.

Drawing Order (Painter's Algorithm)

Canvas has no concept of layers. You draw things in order and later draws appear on top of earlier ones. We always draw in this order:

1. Background grid
2. Stars and nebula clouds
3. Nova bomb wave (if active)
4. Particle explosions
5. Power-up pickups
6. Player bullets and missiles
7. Player ship
8. Asteroids
9. Enemies
10. Boss

> **TIP:** Particles are drawn before the ship so explosions appear behind it. The ship always looks like it is flying through the debris rather than behind it.

The Glow Effect

Almost everything in Neon Blaster X has a colored glow. This is just the canvas shadowBlur property. Set shadowColor to your neon color and shadowBlur to a pixel radius, and anything you draw will have a soft bloom around it.

```
ctx.shadowColor = "#00f5ff";   // cyan glow
```

```
ctx.shadowBlur  = 22;

ctx.strokeStyle = "#00f5ff";
ctx.lineWidth   = 2;
ctx.beginPath();
ctx.moveTo(22, 0);
ctx.lineTo(-13, -13);
// ... draw ship shape ...
ctx.stroke();

ctx.shadowBlur = 0;   // always reset or everything after glows too
```

NOTE: shadowBlur is expensive. Keep it enabled only for the current object you are drawing, then set it back to 0 immediately. Never leave it on between objects.

The Background Grid

The subtle grid gives the game that retro-futuristic feel. It is just a series of vertical and horizontal lines drawn at low opacity.

```
function drawGrid() {
  ctx.strokeStyle = "rgba(0,245,255,.035)";
  ctx.lineWidth = 1;
  for (let x = 0; x < W; x += 60) {
    ctx.beginPath(); ctx.moveTo(x,0); ctx.lineTo(x,H); ctx.stroke();
  }
  for (let y = 0; y < H; y += 60) {
    ctx.beginPath(); ctx.moveTo(0,y); ctx.lineTo(W,y); ctx.stroke();
  }
  // corner brackets
  const corners = [[0,0],[W,0],[W,H],[0,H]];
  ctx.strokeStyle = "rgba(0,245,255,.14)";
  ctx.lineWidth = 1.5;
  for (const [cx,cy] of corners) {
    const sx = cx ? -1 : 1, sy = cy ? -1 : 1;
    ctx.beginPath();
    ctx.moveTo(cx + sx*24, cy);
    ctx.lineTo(cx, cy);
    ctx.lineTo(cx, cy + sy*24);
    ctx.stroke();
  }
}
```

Parallax Stars

Stars drift downward at different speeds to give a sense of depth. Each star has a speed value. Slow stars feel far away, fast ones feel close. When a star falls off the bottom it wraps back to the top with a new random x position.

```
function buildStars() {
  stars = [];
  for (let i = 0; i < 220; i++) {
    stars.push({
      x: Math.random() * W,
      y: Math.random() * H,
      r: Math.random() * 1.6 + 0.15,   // radius
      spd: Math.random() * 0.5 + 0.04,  // drift speed
      b: Math.random()                   // brightness for twinkle
    });
  }
}

function updateStars(dt) {
  for (const s of stars) {
    s.y += s.spd * dt * 0.055;
    if (s.y > H) { s.y = 0; s.x = Math.random() * W; }
  }
}

function drawStars() {
  for (const s of stars) {
    // twinkle: alpha oscillates using time + star position as offset
    const a = 0.25 + s.b * 0.55 + Math.sin(Date.now() * 0.0008 + s.x) * 0.12;
    ctx.globalAlpha = Math.max(0, Math.min(1, a));
    ctx.fillStyle = "#fff";
    ctx.beginPath(); ctx.arc(s.x, s.y, s.r, 0, Math.PI*2); ctx.fill();
  }
  ctx.globalAlpha = 1;
}
```

Screen Shake

When something big explodes, the camera shakes. We implement this by translating the canvas context by a random small offset each frame. The shake amount decays over time.

```
let shakeAmt = 0;

function applyShake() {
  if (shakeAmt <= 0) return;
  ctx.translate(
    (Math.random() - 0.5) * shakeAmt * 2,
    (Math.random() - 0.5) * shakeAmt * 2
  );
  shakeAmt *= 0.82;    // decay
  if (shakeAmt < 0.1) shakeAmt = 0;
}

// trigger it when something explodes:
shakeAmt += big ? 10 : 4;
```

TIP: ctx.save() before the shake translate and ctx.restore() after all drawing. Otherwise the translation

accumulates every frame and the game slowly drifts off screen.

Chapter 3: The Player Ship

Movement with Mouse Tracking

The ship follows the mouse cursor by smoothly interpolating toward it each frame. This is simpler than keyboard movement and feels very responsive. The key is the lerp factor: multiply the distance by a small value each frame. The further away the cursor, the faster the ship moves.

```
function updateShip(dt) {
  const dx = mouse.x - ship.x;
  const dy = mouse.y - ship.y;

  // lerp toward cursor, faster when further away
  const spd = Math.min(1, dt * 0.014);
  ship.x += dx * spd;
  ship.y += dy * spd;

  // clamp to canvas bounds
  ship.x = Math.max(ship.r, Math.min(W - ship.r, ship.x));
  ship.y = Math.max(ship.r, Math.min(H - ship.r, ship.y));

  // rotate to face cursor
  ship.angle = Math.atan2(dy, dx);
}
```

We track the mouse position relative to the canvas element, not the page. This handles the case where the canvas is centered and there is empty space around it.

```
document.getElementById("W").addEventListener("mousemove", e => {
  const r = document.getElementById("W").getBoundingClientRect();
  mouse.x = e.clientX - r.left;
  mouse.y = e.clientY - r.top;
});
```

Drawing the Ship

The ship is drawn using canvas path commands. We translate to the ship position and rotate by the ship angle before drawing, so the ship shape is always defined relative to its center pointing right. This makes the math simple.

```
ctx.save();
ctx.translate(ship.x, ship.y);
ctx.rotate(ship.angle);

// engine flame (drawn first so it appears behind the body)
const ep = 0.5 + 0.5 * Math.sin(ship.thrustPulse);
ctx.shadowColor = "#ff4400";
ctx.shadowBlur = 16;
```

```
ctx.strokeStyle = `rgba(255,${80+ep*80},0,${0.6+ep*0.4})`;
ctx.lineWidth = 3;
ctx.beginPath();
ctx.moveTo(-7, -7); ctx.lineTo(-18 - ep*8, 0); ctx.lineTo(-7, 7);
ctx.stroke();

// ship body
ctx.shadowColor = "#00f5ff";
ctx.shadowBlur = 22;
ctx.strokeStyle = "#00f5ff";
ctx.lineWidth = 2;
ctx.beginPath();
ctx.moveTo(22, 0);
ctx.lineTo(-13, -13);
ctx.lineTo(-7, 0);
ctx.lineTo(-13, 13);
ctx.closePath();
ctx.stroke();

ctx.restore();
```

```
                    (22, 0)  -- nose
                   /|
    engine <-- (-18,0) |
                   \|
  (-13,-13)         (-7, 0)  -- center notch
     \                |
      \              /|
  (-13,13) ----------  |
                   \|

  All coordinates are relative to ship center (0,0)
  Ship points RIGHT by default, then rotated by ship.angle
```

Figure 3.1 - Ship vertex coordinates

The Motion Trail

Each frame we push the ship's current position onto a trail array. We limit the array to 18 entries. When drawing, we loop through the trail from oldest to newest, drawing circles that fade and shrink toward the older end.

```
// update: add current pos each frame
ship.trail.push({ x: ship.x, y: ship.y, a: 1.0 });
if (ship.trail.length > 18) ship.trail.shift();
for (const t of ship.trail) t.a -= 0.055;  // decay alpha

// draw: render from oldest to newest
for (let i = 0; i < ship.trail.length; i++) {
  const t = ship.trail[i];
  const alpha = Math.max(0, t.a * 0.45 * (i / ship.trail.length));
  ctx.globalAlpha = alpha;
  ctx.fillStyle = "#00f5ff";
  ctx.beginPath();
```

```
  ctx.arc(t.x, t.y, 4 * (i / ship.trail.length), 0, Math.PI*2);
  ctx.fill();
}
ctx.globalAlpha = 1;
```

Invincibility Frames

After getting hit the ship becomes temporarily invincible and flashes. We track an invincibility timer in milliseconds. During this time hits are ignored. The flashing is done by skipping the draw call on every other 70ms interval.

```
// in updateShip:
if (ship.inv > 0) ship.inv -= dt;

// in drawShip:
if (ship.inv > 0 && Math.floor(ship.inv / 70) % 2 === 0) return;

// on taking damage:
ship.inv = 2400;  // 2.4 seconds of invincibility
```

Chapter 4: Weapons and Bullets

The Bullet Array

Bullets are simple objects in an array. Each frame we update their position, check if they are off screen, and remove dead ones. We use a filter call to clean up in one line.

```
let bullets = [];

function updateBullets(dt) {
  for (const b of bullets) {
    b.x += b.vx * dt * 0.06;
    b.y += b.vy * dt * 0.06;
    if (b.x < -20 || b.x > W+20 || b.y < -20 || b.y > H+20)
      b.life = 0;
  }
  bullets = bullets.filter(b => b.life > 0);
}
```

NOTE: The 0.06 multiplier on dt converts our velocity units to pixels per millisecond. We tune speed values in comfortable numbers (like 12) and this factor handles the time scaling.

The Four Weapon Types

The game has four weapons selectable with the Q key. Each creates bullets with different properties when the fire function runs.

Laser

Standard shot. At higher power levels, fires 3 bullets in a spread pattern. Fast and precise.

```
const spread = powerLvl >= 4
  ? [-0.2, 0, 0.2]
  : powerLvl >= 2 ? [-0.12, 0.12] : [0];

for (const off of spread) {
  const ang = shipAngle + off;
  bullets.push({
    x: wx, y: wy,
    vx: Math.cos(ang) * 12,
    vy: Math.sin(ang) * 12,
    col: "#00f5ff", r: 3.5, dmg: 1
  });
}
```

Spread Shot

Fires a wide fan of bullets. Great for crowds of small asteroids. Lower damage per bullet.

```
const cnt = 3 + Math.min(powerLvl, 2);
for (let i = 0; i < cnt; i++) {
  const off = (i - (cnt-1)/2) * 0.2;
  bullets.push({
    vx: Math.cos(ang+off)*10,
    vy: Math.sin(ang+off)*10,
    col: "#ffee00",
    dmg: 0.8
  });
}
```

Plasma Ball

Slow, large, pulsing orb. Deals 3x damage. The pulse effect is done with a sine wave on the radius.

```
bullets.push({
  vx: Math.cos(a) * 8,
  vy: Math.sin(a) * 8,
  col: "#bf00ff",
  r: 8, dmg: 3,
  type: "plasma",
  pulse: 0   // increments over time
});

// draw: pulsing radius
const pr = b.r + Math.sin(b.pulse) * 2;
```

Homing Missiles

Steers toward the nearest target. Uses atan2 to find the target angle and rotates toward it each frame.

```
const targetAng = Math.atan2(
  target.y - m.y,
  target.x - m.x
);
const da = angleDiff(targetAng, m.angle);
m.angle += Math.sign(da)
  * Math.min(Math.abs(da), turnRate);

m.vx = Math.cos(m.angle) * m.spd;
m.vy = Math.sin(m.angle) * m.spd;
```

Shoot Timing

We do not fire one bullet per click. Instead we track a shoot timer and fire continuously while the mouse button is held. The timer reset value controls fire rate. Power level shortens the timer.

```
shootT -= dt;
if (shooting && shootT <= 0) {
  fire();
  const baseRate = [180, 150, 120, 220][currentWeapon];  // ms per shot
  const pw = powerLvl >= 4 ? baseRate * 0.55
           : powerLvl >= 2 ? baseRate * 0.75
           : baseRate;
  shootT = pw;
}
```

The Nova Bomb

The Nova is a special ability that requires charging up via kills. When fired, it expands an invisible ring from the ship outward. Anything within the ring at any point during the expansion takes damage or dies. We just check the radius against distances each frame.

```
let novaActive = false, novaR = 0;

function updateNova(dt) {
  if (!novaActive) return;
  novaR += dt * 0.5;   // expand radius over time

  asteroids = asteroids.filter(a => {
    if (dist(ship, a) < novaR + a.r) {
      boom(a.x, a.y, a.col, 16);
      addScore(a, true);
```

```
      return false;        // remove asteroid
    }
    return true;
  });

  if (novaR > Math.max(W, H)) novaActive = false;  // done expanding
}
```

Chapter 5: Enemies and AI

Asteroids

Asteroids are procedurally generated jagged polygons. We pick random radii for each vertex to get that irregular rock shape. They spawn from the screen edges and drift toward the center with some variance.

```
const numPoints = 7 + Math.floor(Math.random() * 5);
const pts = [];
for (let i = 0; i < numPoints; i++) {
  const angle = (i / numPoints) * Math.PI * 2;
  const radius = r * (0.65 + Math.random() * 0.55);
  pts.push({
    x: Math.cos(angle) * radius,
    y: Math.sin(angle) * radius
  });
}
```

Large asteroids split into two mediums when destroyed. Mediums split into two smalls. This creates a satisfying chain reaction effect when you shoot one large rock.

```
// on asteroid death:
if (a.size === "large") {
  spawnAsteroid("medium", a.x + 20, a.y);
  spawnAsteroid("medium", a.x - 20, a.y);
} else if (a.size === "medium") {
  spawnAsteroid("small", a.x + 12, a.y);
  spawnAsteroid("small", a.x - 12, a.y);
}
```

Drone Enemies

Drones home in on the player using smooth velocity steering (not teleporting). They also periodically fire aimed bullets. The steering uses a simple approach: add the difference between desired velocity and current velocity, scaled by a small amount each frame.

```
function updateDrone(e, dt) {
  // steering: nudge velocity toward ship
  const ang = Math.atan2(ship.y - e.y, ship.x - e.x);
  const targetVx = Math.cos(ang) * speed;
  const targetVy = Math.sin(ang) * speed;
  e.vx += (targetVx - e.vx) * 0.03;
  e.vy += (targetVy - e.vy) * 0.03;

  e.x += e.vx * dt * 0.06;
  e.y += e.vy * dt * 0.06;

  // fire at player periodically
```

```
  e.fireT -= dt;
  if (e.fireT <= 0) {
    const aimAng = Math.atan2(ship.y - e.y, ship.x - e.x)
                 + (Math.random() - 0.5) * 0.3;
    enemyBullets.push({
      x: e.x, y: e.y,
      vx: Math.cos(aimAng) * 4,
      vy: Math.sin(aimAng) * 4,
      col: "#ff006e", r: 4
    });
    e.fireT = 1800 + Math.random() * 1200;
  }
}
```

Seeker Enemies

Seekers are simpler than drones. They point directly at the player and move at constant speed. They do not shoot but move faster. The difference in behavior creates nice variety in how you prioritize threats.

The Wave System

Enemies spawn continuously from timers. The wave system is about when those timers fire and how fast enemies move, not about spawning preset groups. When the screen is clear, a cooldown starts. After the cooldown a new wave begins, the wave counter increments, and spawn timers get faster.

```
function checkWave(dt) {
  if (waveCooldown) {
    waveT -= dt;
    if (waveT <= 0 && asteroids.length === 0 && enemies.length === 0 && !boss) {
      waveCooldown = false;
      wave++;
      showBanner("WAVE " + wave, "#ffee00");
      if (wave % 5 === 0) setTimeout(spawnBoss, 2000);
    }
  }
  if (!waveCooldown && asteroids.length === 0 && enemies.length === 0 && !boss) {
    waveCooldown = true;
    waveT = 2200;
  }
}
```

Chapter 6: Boss Design

Neon Blaster X has three bosses that rotate on waves divisible by 5. Each one has a completely different shape, movement style, and attack pattern. Having three phases per boss (triggered at 66% and 33% HP) makes the fights escalate dramatically.

Boss 1: HEXON

Hexon is the classic boss. A hexagonal shape with six orbiting purple nodes. It drifts around the top third of the screen, periodically stopping at a new position. The three attack phases are radial burst, radial plus aimed fan, and dense spiral.

```
// orbiting nodes
for (let i = 0; i < 6; i++)
  boss.orbs.push({ angle: (i/6)*Math.PI*2, dist: 80, r: 12 });

// each frame: rotate orbs
for (const o of boss.orbs) o.angle += 0.02 * dt * 0.06;

// draw each orb at its position around the boss center
const ox = boss.x + Math.cos(o.angle) * o.dist;
const oy = boss.y + Math.sin(o.angle) * o.dist;
ctx.arc(ox, oy, o.r, 0, Math.PI*2);
```

Boss 2: SERPENTIS

Serpentis is an 8-segment snake that weaves toward the player. The head follows the ship using atan2 direction plus a sine-wave side offset for the weaving motion. Each segment then follows the one ahead of it by pulling toward it when the gap grows too large.

```
// head steering with side-to-side wave
boss.waveOff += dt * 0.002;
const toShipAng = Math.atan2(ship.y - head.y, ship.x - head.x);
const sideWave = Math.sin(boss.waveOff) * 1.8;
const perpAng = toShipAng + Math.PI / 2;

head.vx += (Math.cos(toShipAng)*speed + Math.cos(perpAng)*sideWave - head.vx) *
0.06;
head.vy += (Math.sin(toShipAng)*speed + Math.sin(perpAng)*sideWave - head.vy) *
0.06;

// each following segment pulls toward the one ahead
for (let i = 1; i < segments.length; i++) {
  const prev = segments[i-1], cur = segments[i];
  const dx = prev.x - cur.x, dy = prev.y - cur.y;
  const d = Math.sqrt(dx*dx + dy*dy) || 1;
  if (d > 34) {
    cur.x += (dx/d) * (d-34) * 0.3;
    cur.y += (dy/d) * (d-34) * 0.3;
  }
```

```
}
```

At higher phases more segments open fire at once. Phase 3 fires from 4 different segments simultaneously, turning the arena into a web of bullets.

Boss 3: THE TWINS

The Twins are two separate bosses that share one HP bar. They each move independently and fire in coordinated patterns. At 50% HP, the first twin dies. The second enters rage mode with much faster crossfire.

The shared HP bar is the key design trick. Players have to damage both twins to advance, but visually it looks like one combined health pool. When the first twin goes down at halftime it feels like a major moment.

```
// twins share boss.hp, but each has its own position and fire timer
boss = {
  type: 3,
  hp: totalHp,
  maxHp: totalHp,
  twins: [
    { x: W/3,   y: -80, rot: 0, fireT: 500,  col: "#ffee00", alive: true },
    { x: W*2/3, y: -80, rot: 0, fireT: 1200, col: "#00f5ff", alive: true }
  ]
};

// when hp drops to 50%, kill first twin
if (boss.hp <= boss.maxHp * 0.5 && boss.twins[0].alive) {
  boss.twins[0].alive = false;
  boom(boss.twins[0].x, boss.twins[0].y, "#ffee00", 50, true);
}
```

```
  HP: 100% ==================================
  Phase 0  [ TWIN A alive ]  [ TWIN B alive ]
           Both fire radial bursts independently

  HP:  50% ==================================
           [ TWIN A DIES  ]  [ TWIN B alive ]
           Explosion + banner: "TWIN DOWN"

  Phase 2  [             ]  [ TWIN B RAGE  ]
           Dense crossfire, fast fire rate

  HP:   0% ==================================
           [ TWIN B DIES  ] -- boss cleared
```

Figure 6.1 - The Twins HP flow

Chapter 7: Collision Detection

Circle vs Circle

Almost every collision in this game is circle vs circle. It is fast, simple, and good enough for an arcade game. Two circles collide when the distance between their centers is less than the sum of their radii.

```
function dist(a, b) {
  const dx = a.x - b.x;
  const dy = a.y - b.y;
  return Math.sqrt(dx*dx + dy*dy);
}

// collision check:
if (dist(bullet, asteroid) < asteroid.r) {
  // hit!
}
```

TIP: For pixel-perfect collision on asteroids you would need polygon intersection tests. We skip that and use the bounding circle radius. Players rarely notice and the game runs much faster.

Iterating Efficiently

We check bullets against all asteroids, then bullets against all enemies. The trick is to break out of the inner loop as soon as a bullet hits something. One bullet can only destroy one thing.

```
for (const b of bullets) {
  for (const a of asteroids) {
    if (dist(b, a) < a.r) {
      b.life = 0;
      a.hp -= b.dmg;
      // handle death, particles, etc
      break;  // bullet is gone, stop checking
    }
  }
}

// clean up dead objects in one pass
asteroids = asteroids.filter(a => a.hp > 0);
bullets   = bullets.filter(b => b.life > 0);
```

The Shield System

Before doing direct damage to the player, we check if the shield has HP. If it does, we drain shield HP instead. The shield bar on screen is just a div whose width is set by the shieldHP value as a percentage.

```
if (bulletHitsShip) {
  if (shieldHP > 0) {
    shieldHP = Math.max(0, shieldHP - 30);
    sfx("shield");
  } else {
    takeDamage();
  }
}

function updateShieldBar() {
  document.getElementById("shieldFill").style.width = shieldHP + "%";
}
```

Chapter 8: Particle Systems

Explosion Particles

Particle systems are surprisingly simple. An explosion is just a burst of small objects that fly outward and fade. Each particle has a position, velocity, color, and a life value that counts down from 1 to 0.

```
function boom(x, y, col, count, big) {
  for (let i = 0; i < count; i++) {
    const angle = Math.random() * Math.PI * 2;
    const speed = Math.random() * (big ? 6 : 4) + 0.5;
    particles.push({
      x, y,
      vx: Math.cos(angle) * speed,
      vy: Math.sin(angle) * speed,
      life: 1.0,
      decay: Math.random() * 0.025 + 0.01,
      r: Math.random() * (big ? 5 : 3) + 1,
      col,
      sw: false    // false = dot, true = shockwave ring
    });
  }
  // add a shockwave ring
  particles.push({ x, y, life: 1, decay: 0.028,
    maxR: big ? 90 : 55, r: 2, col, sw: true });
}
```

```
function updateParticles(dt) {
  for (const p of particles) {
    p.x += p.vx * dt * 0.06;
    p.y += p.vy * dt * 0.06;
    p.life -= p.decay * dt * 0.06 * 60;
    if (!p.sw) {
      p.vx *= 0.97;   // drag
      p.vy *= 0.97;
    } else {
      p.r = p.maxR * (1 - p.life);  // expand ring
    }
  }
  particles = particles.filter(p => p.life > 0);
}
```

The Shockwave Ring

The expanding ring is one particle with a sw flag. Instead of moving outward like a normal particle, its radius grows over time and its opacity fades. We draw it as a circle stroke rather than a filled dot.

```
if (p.sw) {
  ctx.strokeStyle = p.col;
  ctx.lineWidth = 2;
  ctx.shadowColor = p.col;
```

```
  ctx.shadowBlur = 8;
  ctx.beginPath();
  ctx.arc(p.x, p.y, p.r, 0, Math.PI*2);
  ctx.stroke();
} else {
  ctx.fillStyle = p.col;
  ctx.beginPath();
  ctx.arc(p.x, p.y, p.r * p.life, 0, Math.PI*2);
  ctx.fill();
}
```

Floating Score Text

Score popup text is not a canvas draw. It is a real HTML div with a CSS animation. We create it, position it over the canvas, let CSS animate it upward, then remove it after the animation ends.

```
function floatText(x, y, text, col) {
  const el = document.createElement("div");
  el.className = "ftxt";
  el.style.left  = x + "px";
  el.style.top   = y + "px";
  el.style.color = col;
  el.textContent = text;
  document.getElementById("W").appendChild(el);
  setTimeout(() => el.remove(), 1250);
}

/* CSS */
.ftxt {
  position: absolute;
  pointer-events: none;
  z-index: 25;
  animation: floatUp 1.2s forwards;
}
@keyframes floatUp {
  0%   { opacity: 1; transform: translateY(0) scale(1); }
  100% { opacity: 0; transform: translateY(-60px) scale(1.3); }
}
```

Chapter 9: Sound and Music with Web Audio API

The Web Audio API lets you synthesize sounds in JavaScript without any audio files. Every sound in Neon Blaster X is generated at runtime: oscillators shaped by frequency envelopes. This keeps the file small and makes the game fully self-contained.

Setting Up the Audio Context

You need one AudioContext for the whole game. It can only be created after a user interaction (browser security rule). We initialize it on the first click or keypress.

```
let ac, masterGain, musicGain, sfxGain;

function initAudio() {
  if (ac) return;  // already initialized
  ac = new (window.AudioContext || window.webkitAudioContext)();

  masterGain = ac.createGain();
  masterGain.connect(ac.destination);

  musicGain = ac.createGain();
  musicGain.gain.value = 0.4;
  musicGain.connect(masterGain);

  sfxGain = ac.createGain();
  sfxGain.gain.value = 0.65;
  sfxGain.connect(masterGain);
}
```

Making Sound Effects

Each sound effect is an oscillator with a frequency envelope. We set the starting frequency and use exponentialRampToValueAtTime to slide it down or up over a short duration. Different oscillator types (square, sawtooth, sine) give different timbres.

```
function sfx(type) {
  const o = ac.createOscillator();
  const g = ac.createGain();
  o.connect(g); g.connect(sfxGain);
  const t = ac.currentTime;

  if (type === "shoot") {
    o.type = "square";
    o.frequency.setValueAtTime(900, t);
    o.frequency.exponentialRampToValueAtTime(150, t + 0.08);
    g.gain.setValueAtTime(0.1, t);
    g.gain.exponentialRampToValueAtTime(0.001, t + 0.08);
    o.start(); o.stop(t + 0.08);
  }
```

```
  if (type === "explode") {
    o.type = "sawtooth";
    o.frequency.setValueAtTime(180, t);
    o.frequency.exponentialRampToValueAtTime(30, t + 0.35);
    g.gain.setValueAtTime(0.25, t);
    g.gain.exponentialRampToValueAtTime(0.001, t + 0.35);
    o.start(); o.stop(t + 0.35);
  }
}
```

Oscillator Types

square: digital/retro beep

sawtooth: buzzy, good for explosions

sine: smooth, good for pickups

triangle: soft, gentle tones

Frequency Envelopes

pitch sweep down = explosion feel

pitch sweep up = power-up/reward

flat pitch = drone/ambient

fast attack + decay = punchy hit

Procedural Background Music

The background music uses three oscillators running continuously: a low bass drone, a pulsing pad, and an arpeggiated melody. The bass and pad run at fixed frequencies. The arp cycles through a note array on a timer.

```
// bass drone - runs forever
const bass = ac.createOscillator();
bass.type = "sawtooth";
bass.frequency.value = 55;  // low A
bassGain.gain.value = 0.08;
bass.start();

// pad with LFO tremolo
const lfo = ac.createOscillator();
lfo.frequency.value = 0.5;  // 0.5 Hz = one pulse per 2 seconds
const lfoGain = ac.createGain();
lfoGain.gain.value = 0.04;
lfo.connect(lfoGain);
lfoGain.connect(padGain.gain);  // LFO modulates pad volume

// arp: cycles through notes on a setInterval
const notes = [220, 277, 330, 440, 554, 660];
let ni = 0;
setInterval(() => {
  arp.frequency.setValueAtTime(notes[ni % notes.length] * 2, ac.currentTime);
  ni++;
}, 180);
```

Chapter 10: UI System

HTML Overlays on Canvas

The menus, pause screen, and game over screen are regular HTML divs absolutely positioned over the canvas. This is much easier than drawing menus on the canvas itself. You get CSS transitions, hover effects, and the browser handles all the text rendering.

```
/* All screens share this class */
.scr {
  position: absolute;
  inset: 0;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  background: rgba(2, 4, 8, 0.95);
  z-index: 20;
}
.scr.off { display: none; }

/* Switching screens is just toggling the off class */
function showScr(id) {
  document.querySelectorAll(".scr").forEach(s => s.classList.add("off"));
  if (id) document.getElementById(id).classList.remove("off");
}
```

Button Design

The buttons use a clip-path for the angled corners and a sliding fill hover effect. The fill is a pseudo-element that slides in from the left on hover using a CSS transform. This all-CSS approach means no JavaScript needed for the hover effect.

```
.btn {
  clip-path: polygon(10px 0%, 100% 0%, calc(100% - 10px) 100%, 0% 100%);
  border: 1px solid var(--c);
  background: transparent;
  color: var(--c);
  position: relative; overflow: hidden;
  transition: color 0.2s;
}
.btn::before {
  content: "";
  position: absolute; inset: 0;
  background: var(--c);
  transform: translateX(-102%);
  transition: transform 0.22s;
  z-index: -1;
}
.btn:hover { color: #000; }
.btn:hover::before { transform: translateX(0); }
```

The HUD

The HUD is a flex row at the top of the wrapper div. Three blocks: score/best on the left, lives in the middle, wave/kills on the right. The boss HP bar sits in the center block and is hidden until a boss spawns.

All HUD values are updated by simple functions that just set textContent or adjust a div width. Nothing fancy.

```
function updateScoreHUD() {
  document.getElementById("sv").textContent = score;
}

function updateShieldBar() {
  document.getElementById("shieldFill").style.width = shieldHP + "%";
}

function updateLivesHUD() {
  for (let i = 1; i <= 3; i++)
    document.getElementById("l" + i).classList.toggle("gone", i > lives);
}
```

The Combo Multiplier

Kills within a short time window increase the combo count. Every 3 kills the multiplier increments. The multiplier display fades in when active and fades out when the timer expires. The scaling CSS transform gives a satisfying pulse effect as it grows.

```
function addCombo() {
  comboCount++;
  comboTimer = 2800;   // reset timer
  comboMult = Math.min(8, 1 + Math.floor(comboCount / 3));
  updateComboDisplay();
}

// in checkWave, decay the timer each frame:
if (comboTimer > 0) {
  comboTimer -= dt;
  if (comboTimer <= 0) { comboCount = 0; comboMult = 1; }
}
```

The Leaderboard

Scores are saved to localStorage as a JSON array. On game over the player types a name and the score is added, sorted, and sliced to keep only the top 10. The leaderboard screen renders these as HTML rows.

```
function submitScore() {
  const name = document.getElementById("nameInput").value.toUpperCase() ||
"PILOT";
  lb.push({ name, score, wave, kills });
  lb.sort((a, b) => b.score - a.score);
  lb = lb.slice(0, 10);
  localStorage.setItem("nblb", JSON.stringify(lb));
}
```

Chapter 11: Polish and Game Feel

The difference between a rough prototype and a game that feels good to play is almost entirely in polish. None of these features add gameplay but all of them matter a lot for how it feels.

The Custom Cursor

We hide the system cursor on the canvas and overlay an SVG crosshair that we move with JavaScript. The SVG has the right visual style for the game and makes the ship feel properly targeted.

```
/* hide the real cursor over the game */
canvas { cursor: none; }

/* the SVG crosshair follows mouse.x and mouse.y */
cur.style.left = mouse.x + "px";
cur.style.top  = mouse.y + "px";
```

The Wave Banner

Wave banners are temporary DOM elements injected over the game. A CSS keyframe animation handles the scale-in, hold, and fade-out. We remove the element after the animation completes using a timeout.

```
function showBanner(txt, col, dur) {
  const el = document.createElement("div");
  el.style.cssText = `
    position: absolute;
    left: 50%; top: 44%;
    transform: translate(-50%, -50%);
    color: ${col};
    animation: wbIn ${dur}ms forwards;
  `;
  el.textContent = txt;
  document.getElementById("W").appendChild(el);
  setTimeout(() => el.remove(), dur + 100);
}
```

Achievement Toasts

Achievement popups appear in the top right for a few seconds then disappear. They are just a div with display toggled. The animation slides it in from the right.

Hit Flash on Enemies

When an enemy takes damage, it flashes white briefly. We do this with a flash value that starts at 1 and decays. In the draw function, if flash is above a threshold we use white for the stroke color instead of the enemy color.

```
// on hit: set flash
a.flash = 1.0;

// in update: decay it
if (a.flash > 0) a.flash -= dt * 0.015;

// in draw: use white while flashing
ctx.strokeStyle = a.flash > 0.3 ? "#fff" : a.col;
```

Power-up Visual Design

Power-ups need to be noticeable without being distracting. They drift across the screen slowly, rotate, and pulse in scale using a sine wave. The double-diamond shape stands out clearly against asteroids.

Scanlines

The scanline overlay is a CSS repeating-linear-gradient on a div that sits above everything else at high z-index but with pointer-events: none so it does not block interaction. It gives the whole game a slightly retro CRT feel.

```
#sl {
  position: absolute; inset: 0;
  background: repeating-linear-gradient(
    to bottom,
    transparent 0, transparent 2px,
    rgba(0,0,0,0.06) 2px, rgba(0,0,0,0.06) 4px
  );
  pointer-events: none;
  z-index: 30;
}
```

Chapter 12: Performance Tips

Canvas games can get slow when you draw too many things or use expensive operations carelessly. Here are the main things to watch.

Limit shadowBlur Usage

shadowBlur is one of the most expensive canvas operations. Use it only when drawing and reset it immediately. Never leave it set between frames.

```
ctx.shadowColor = "#00f5ff";
ctx.shadowBlur  = 22;
// draw your thing
ctx.shadowBlur  = 0;  // always reset
```

Object Pooling

Creating and destroying thousands of JavaScript objects every frame causes garbage collection pauses. For high-frequency objects like bullets and particles, consider an object pool: a pre-allocated array you reuse rather than allocating new objects.

In Neon Blaster X we use simple array push/filter which works fine at this scale. For a game with hundreds of bullets on screen simultaneously you would want proper pooling.

Avoid Unnecessary Canvas State Changes

Setting fillStyle, strokeStyle, lineWidth etc. are relatively cheap but they add up. Group all objects of the same color/style together in your draw loop so you change state as few times as possible.

Delta Time and Frame Rate

Always multiply movement by delta time. This keeps the game consistent across refresh rates. The 50ms cap on delta time prevents the "spiral of death" where a slow frame causes large jumps which cause more processing which causes more slow frames.

```
const dt = Math.min(ts - lastT, 50);
```

Chapter 13: Quick Reference

All Game Variables at a Glance

Variable	Type	Purpose
GS	string	Current game state: menu | playing | paused | dead
ship	object	Player ship: x, y, angle, inv, trail, thrustPulse
bullets	array	Player bullets and plasma shots
missiles	array	Homing missiles (separate from bullets for targeting logic)
asteroids	array	All active asteroids
enemies	array	Drones and seekers
enemyBullets	array	Bullets fired by enemies and bosses
particles	array	Explosion dots and shockwave rings
powerups	array	Active collectible pickups on screen
boss	object|null	Current boss or null if none active
wave	number	Current wave number (increments when screen clears)
score	number	Current session score
lives	number	Remaining player lives (0-3)
shieldHP	number	Shield health 0-100, absorbs damage before lives
specialCharge	number	Nova bomb charge 0-100, fills on kills
powerLvl	number	Weapon power level 0-5 (affects spread and fire rate)
comboCount	number	Kills in current combo chain
comboMult	number	Score multiplier from combo (1-8x)
currentWeapon	number	Active weapon index 0-3 (laser/spread/plasma/missile)
S	object	Settings: mvol, svol, stars, shake, scan, diff

Key Controls

Input	Action
Mouse Move	Ship follows cursor
Left Click / Space	Fire current weapon (hold to auto-fire)
Q	Cycle weapon type (Laser > Spread > Plasma > Missile)
F	Fire Nova Bomb (requires full special charge)
ESC / P	Pause / resume game

Boss Rotation Schedule

Wave	Boss	Color	Special Mechanic
5, 20, 25...	HEXON	Red + Purple	Orbiting nodes, spiral shots

10, 25...	SERPENTIS	Green	8-segment snake, multi-segment fire
15, 30...	THE TWINS	Yellow + Cyan	Two bodies, shared HP, twin dies at 50%

Closing Notes

Building Neon Blaster X from scratch is a good way to understand how game loops, rendering, and game feel actually work. The concepts here apply to almost any 2D game regardless of platform or language. Once you understand the loop, delta time, and object arrays, the rest is just adding more stuff to update and draw.

From here you could go further by adding:

- A proper object pool for particles and bullets
- Sprite sheets instead of canvas drawing for characters
- WebGL rendering via Three.js or raw WebGL for performance
- Multiplayer via WebSockets
- A level editor for hand-crafted wave layouts
- Touch controls for mobile

The game is a single HTML file. Read through it. Change numbers. Break things. That is how you learn.

NEON BLASTER X

Made By TLD Productions LLC