# THE ENGINE QUEST

A Field Manual for Building a Game Engine
in C89 on Windows XP — Thirty Days, No Internet

TARGET: ASUS Eee PC X101CH · ATOM N2600 · 1 GB RAM
GPU: PowerVR SGX545 (GMA 3600) · SHADER MODEL 3.0
TOOLCHAIN: VISUAL C++ 6.0 · OPENGL ES 2.0 VIA ANGLE
AUDIO: BASS 2.4 · ART: MILKSHAPE 3D + PAINT SHOP PRO 7

CABIN PRESS · OREGON · MMXXVI

# Contents

# How to Use This Book

This book is a field manual for thirty days of work. Each day chapter is sized for one sitting: read it start to finish, then build the thing. Do not read three days ahead on a rainy evening and then try to remember them at the keyboard. One day, one sitting, one working result.

Keep `journal.txt` in the project directory and write at least one line per day: what you built, the fps you measured, the bug that ate an hour. Several later chapters will ask you to look numbers up in it. The journal is not sentiment; it is your test log, and on this machine the numbers you record are the only benchmark database you have.

Every day chapter ends with a **DONE WHEN** block. That block is the contract: the day is finished when the sentence in it is true on your screen, not when the code compiles, and not when you feel done. Below it, an **IF YOU ARE STUCK** block lists the hints, ordered from gentle nudge to outright answer. Read them one at a time and stop as soon as you are unstuck.

The appendices at the back hold the reference matter you would otherwise search for: the OpenGL ES 2.0 entry points and enums you are allowed to use, the GLSL ES 1.00 language in full, byte-level layouts of every file format you will parse, and tables of errors with their real causes. When a chapter says "see the appendix", it means it.

Which brings us to the contract this book makes with you: there is no internet in these pages, and none is required. Every constant, every struct layout, every function signature you need for thirty days is printed here. If you find yourself wanting to look something up online, you have either skipped an appendix or found an error in the book. Check the appendix first.

# Chapter 0: The Machine

Before you write a line of engine code, you should know exactly what you are standing on. This chapter is a guided tour of the machine: the software stack from your C89 source down to the GPU silicon, the peculiar personality of that GPU, the arithmetic of vsync, a complete fullscreen implementation you will reuse for the rest of the book, and the debugging tools that replace the internet. Read it once now and again around day 11, when the terrain starts asking hard questions about fill rate.

## The Stack, Top to Bottom

When your code calls `glDrawElements`, the call passes through five layers before any pixel changes. Here is the whole tower:

```
+-----------------------------------------------------+
|  your C89 code   (main.c, engine.c, mathlib.c)    |
|  calls glDrawElements(), eglSwapBuffers(), ...    |
+------------------------+----------------------------+
                         |  function pointers from gl_loader.h
                         v
+-----------------------------------------------------+
|  libGLESv2.dll + libEGL.dll          (ANGLE)      |
|  OpenGL ES 2.0  -->  Direct3D 9 translator        |
+------------------------+----------------------------+
                         |  D3D9 API calls
                         v
+-----------------------------------------------------+
|  d3d9.dll   (Microsoft Direct3D 9 runtime)        |
+------------------------+----------------------------+
                         |  XPDM driver interface
                         v
+-----------------------------------------------------+
|  EMGD 1.15.1   (Intel "GMA 3600" display driver)  |
+------------------------+----------------------------+
                         |  command buffers, shared system RAM
                         v
+-----------------------------------------------------+
|  PowerVR SGX545   (tile-based deferred renderer)  |
+-----------------------------------------------------+
```

Layer by layer. Your code never links against a GL library directly. Instead, `gl_loader.h` declares one function pointer per GL and EGL entry point, named exactly like the real function, and `gl_loader_init()` fills them in with `GetProcAddress` from the two DLLs. This is why you must call `gl_loader_init()` once, after the window exists and before any GL call; a GL call before that is a jump through a NULL pointer, and the debugger will show you an access violation at address zero.

The two DLLs are ANGLE: the Almost Native Graphics Layer Engine. ANGLE was built so that web browsers could run OpenGL ES 2.0 content on Windows machines whose OpenGL drivers were missing or broken - which describes most Windows machines of the era, and this one emphatically. Our copies ship with Firefox 52 ESR. ANGLE takes your GLES2 calls and your GLSL ES 1.00 shaders, translates the shaders to Direct3D 9 HLSL, compiles them with the D3D shader compiler, and issues everything as D3D9 draw calls. It is a full, conformant GLES2 implementation; it is not a toy wrapper. It also quietly papers over several D3D9 eccentricities (half-pixel offsets, depth range, BGRA surfaces) so that you can treat the whole thing as standard GL. Believe the abstraction; it holds.

Below ANGLE sits the Direct3D 9 runtime, and below that the EMGD 1.15.1 driver - Intel's "Embedded Media and Graphics Driver", which is Intel packaging around a licensed PowerVR core. And here is the punch line of the whole

diagram: **EMGD provides no usable desktop OpenGL on XP.** There is no OpenGL ICD for this chip. Without ANGLE, an OpenGL context on this machine falls back to Microsoft's software renderer, which speaks OpenGL 1.1 at single-digit frame rates. Direct3D 9 is the only hardware path the driver offers, so anything that wants hardware acceleration must speak D3D9 - directly, or through a translator.

What this history means for you, practically:

**You have OpenGL ES 2.0 and nothing older.** Every tutorial-book incantation from the 1990s - `glBegin`/`glEnd`, `glVertex3f`, `glMatrixMode`, `glPushMatrix`, `glLightfv`, `glEnable(GL_FOG)` - does not exist here. Not deprecated: absent. There is no fixed-function pipeline at all. Every triangle you draw goes through a vertex shader and a fragment shader that you wrote, fed by vertex buffers and uniforms that you filled. There is no matrix stack; on day 1 you build your own `mat4` library, and on day 8 you build your own stack. This sounds like punishment. It is actually the reason this curriculum works: the fixed-function pipeline was always just somebody else's shaders, and by day 13 you will have reimplemented it and understood it, which is more than most people who used it ever did.

**Extensions are a fixed menu.** ANGLE exposes a specific set of GLES2 extensions on this machine (vertex array objects, instancing, float textures, DXT compression, occlusion queries, and a handful more - the full list with usage notes is in the appendix). If an extension is not on that list, it does not exist for you, whatever any other GL documentation may claim.

**EGL is version 1.4.** The headers you have declare some EGL 1.5 entry points; at runtime those pointers are NULL. Never call them. The 1.4 surface is everything you need.

## The GPU: A Tile-Based Deferred Renderer

The SGX545 does not draw the way a desktop GPU of its day drew. A desktop card was an "immediate mode renderer": each triangle you submitted was rasterized into the framebuffer in video memory, right then, read-modify-write over the bus. The SGX545 is a **tile-based deferred renderer** (TBDR). It divides the screen into small tiles, and when you submit triangles it does not draw them; it bins them, recording which tiles each triangle touches. Only when the frame ends - at `eglSwapBuffers`, effectively - does it shade anything, processing one tile at a time in a small on-chip buffer, and writing each finished tile out to memory exactly once. Before shading a tile it performs hidden-surface removal across everything binned into it, so opaque pixels that lose the depth test are never shaded at all.

This architecture has consequences you will feel every single day, so learn them now:

**Clears are free - so always clear.** A full `glClear` of color and depth at the top of the frame does not write memory; it tells the chip the previous contents are dead, so tiles start from a register state instead of being loaded from RAM. Skipping the clear because "the skybox covers everything anyway" is a desktop habit, and here it makes the frame *slower*, because the hardware must read every tile back in before drawing. Clear color, depth, and stencil, every frame, no exceptions.

**Opaque overdraw is forgiven.** Because of hidden-surface removal, drawing the terrain behind a wall costs binning work but almost no shading work. You still cull (day 9, day 12) - binning and vertex shading are not free, and the Atom pays for every draw call - but you do not need the desktop-era paranoia about front-to-back sorting of opaque geometry. Blended geometry is another matter: transparency defeats hidden-surface removal, so particles (day 18) pay full price per layer.

**Mid-frame readbacks are poison.** Anything that forces the binned scene to be rendered before the frame is over - a `glReadPixels`, mapping a buffer that the GPU still owns, or ping-ponging into and out of the same framebuffer - makes the hardware flush the whole pipeline, render, and start binning again from nothing. One careless readback can halve your frame rate. Render-to- texture (day 25) is fine when the pass is a clean unit: render the whole texture, then use it. Read-modify-write within a frame is not.

**Present overhead dominates simple scenes.** Between the tile writes, ANGLE's D3D9 present, and a blit through GDI-managed surfaces on XP, there is a fixed per-frame cost that has nothing to do with your triangle count. This is why your

empty-window skeleton runs at 60 fps and not 3000, and why a spinning cube benchmarks about the same as an empty window. Do not judge any optimization by an almost-empty scene; the fixed cost drowns the signal. Judge it on day 11's terrain.

## The Vsync Ladder

The skeleton calls `eglSwapInterval(dpy, 1)`, which asks that each `eglSwapBuffers` wait for the display's vertical blank. The panel refreshes 60 times a second, so a swap can only happen at 16.7 ms boundaries. That single fact produces the most important performance behavior on this machine: **frame rate quantizes.** If your frame takes 16.0 ms, you ship every blank: 60 fps. If it takes 17.0 ms, you miss the blank and wait for the next one: every frame now occupies two intervals, and you get exactly 30 fps. Miss that and you get 20, then 15 - the ladder is 60, 30, 20, 15, which is 60/1, 60/2, 60/3, 60/4. There is no 45 fps on this machine. When your counter drops from 60 to 30, you did not lose half your performance; you lost perhaps half a millisecond, and the ladder did the rest. This is why the fps counter stays in the title bar (and later the overlay): a drop to 30 is an early-warning light, not a catastrophe, but it means you are now standing on the edge of a cliff.

Now for the XP-specific part. Windows XP drives the GPU through the XPDM driver model, which has **no GPU scheduler**. The GPU is a shared appliance with no referee: Direct3D work and ordinary GDI window drawing contend for it, first come, first served. Hover the mouse over the taskbar and XP repaints tooltips and buttons with GDI; those paints interleave with your D3D9 presents, your swap misses its blank, and your engine drops to 30 fps because a tooltip faded in. You will see this during development - a windowed engine stuttering the moment anything else on the desktop moves - and it is not your bug.

This is also why, in 1999, every game ran fullscreen. A fullscreen application owned the display: nothing else painted, the desktop was not composited (there was no compositor to do it), and the swap could be a page flip - exchange two pointers during the blank - instead of a copy. Windowed mode was for level editors. Our stack presents windowed-style through D3D9 even when the window covers the screen, so we do not get true exclusive flips, but a borderless fullscreen window still wins: XP hides the taskbar for it, nothing overlaps it, and GDI goes quiet.

## Fullscreen, Properly

So the engine runs fullscreen from day 2 onward. You have two options, and the listing below implements both:

**Native 1024x600.** A borderless `WS_POPUP` window sized to the screen. No mode switch, instant Alt-Tab, and every pixel of your framebuffer maps to one pixel of panel: crisp. This is the default.

**800x600 with a mode switch.** `ChangeDisplaySettings` drops the desktop to 800x600 first. That is 480,000 pixels against 614,400 - about 30% fewer pixels to shade and write per frame, which on a fill-rate-bound day (water, particles, postprocessing) can be the difference between 30 and 60 fps. The cost: the panel scales 800x600 up to 1024x600 non-uniformly, so everything is slightly soft and slightly stretched. Keep this in your pocket for days 25-29 and let the journal decide.

```
/* fullscreen.c - borderless fullscreen with optional mode switch.
   Rule one: whatever happens, the desktop mode gets restored.     */

#include <windows.h>
#include <string.h>

static int g_mode_changed = 0;   /* did we switch the desktop mode? */
static int g_want_lowres  = 0;   /* run at 800x600?                 */

/* Switch the desktop to w x h x 32. Returns 1 on success. */
int Fullscreen_SetMode(int w, int h)
{
    DEVMODE dm;
```

```
    LONG r;

    memset(&dm, 0, sizeof(dm));
    dm.dmSize       = sizeof(dm);
    dm.dmPelsWidth  = w;
    dm.dmPelsHeight = h;
    dm.dmBitsPerPel = 32;
    dm.dmFields     = DM_PELSWIDTH | DM_PELSHEIGHT | DM_BITSPERPEL;

    r = ChangeDisplaySettings(&dm, CDS_FULLSCREEN);
    if (r != DISP_CHANGE_SUCCESSFUL) {
        MessageBoxA(NULL, "Mode switch failed; staying native.",
                    "fullscreen", MB_OK);
        return 0;
    }
    g_mode_changed = 1;
    return 1;
}

/* Put the desktop back. Safe to call any number of times. */
void Fullscreen_Restore(void)
{
    if (g_mode_changed) {
        ChangeDisplaySettings(NULL, 0);
        g_mode_changed = 0;
    }
}

/* Create the borderless window covering the whole screen.
   Register your WNDPROC in wc.lpfnWndProc as usual.       */
HWND Fullscreen_Create(HINSTANCE inst, WNDPROC proc, int lowres)
{
    WNDCLASSA wc;
    int w;
    int h;

    g_want_lowres = lowres;
    if (lowres) {
        Fullscreen_SetMode(800, 600);
    }
    /* query AFTER the switch: metrics reflect the current mode */
    w = GetSystemMetrics(SM_CXSCREEN);
    h = GetSystemMetrics(SM_CYSCREEN);

    memset(&wc, 0, sizeof(wc));
    wc.style         = CS_OWNDC;
    wc.lpfnWndProc   = proc;
    wc.hInstance     = inst;
    wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
    wc.lpszClassName = "EngineWindow";
    if (!RegisterClassA(&wc)) {
        return NULL;
    }
    return CreateWindowA("EngineWindow", "The Engine Quest",
                         WS_POPUP | WS_VISIBLE, 0, 0, w, h,
                         NULL, NULL, inst, NULL);
}

/* In your WNDPROC: be a good citizen on Alt-Tab. */
/*
    case WM_ACTIVATE:
        if (LOWORD(wParam) == WA_INACTIVE) {
            Fullscreen_Restore();
            ShowWindow(hwnd, SW_MINIMIZE);
        } else if (g_want_lowres) {
            Fullscreen_SetMode(800, 600);
```

```
        }
        return 0;
*/
```

Three points of order. First, `Fullscreen_Restore` must run on *every* exit path - normal quit, error `MessageBoxA` paths, all of them - or you strand the user's desktop at 800x600, the classic sin of 1999 shareware. Second, the `WM_ACTIVATE` handler is not optional: when the user Alt-Tabs away you restore the mode and minimize, and when they come back you re-switch. While you are in that handler, also pause your fixed-timestep accumulator (clamp the next dt), or the physics will sprint through every second the window spent minimized. Third, if `eglSwapBuffers` ever returns `EGL_FALSE` and `eglGetError()` reports `EGL_CONTEXT_LOST` after a mode change, tear down and rebuild the EGL context; on this stack it is rare, because ANGLE presents windowed-style and absorbs D3D9 device resets internally, but check for it once and you are covered forever.

## Debugging Without the Internet

Your debugger is Visual C++ 6.0's, and it is better than its age suggests. The keys: **F5** runs under the debugger, **F9** toggles a breakpoint on the current line, **F10** steps over, **F11** steps into. When you are stopped, the **Watch** window evaluates expressions - type `g_camera.pos` or `verts[i*8+3]` and watch it change as you step. The **Memory** window (View, Debug Windows, Memory) shows raw bytes at any address, which is exactly what you want when a file loader misbehaves: point it at your vertex buffer and read the floats yourself. Learn to love it before day 4; the TGA loader will give you a reason.

`OutputDebugString` is your printf before day 19 gives you a console. It sends a string to the debugger's **Debug** output pane (and vanishes harmlessly when no debugger is attached), so it works even in a fullscreen app with no console window. Format with `_snprintf` first - remember, no `snprintf` on this compiler:

```
{
    char buf[128];
    _snprintf(buf, sizeof(buf), "loaded %d verts, %d tris\n",
              nverts, ntris);
    OutputDebugStringA(buf);
}
```

When the program will not even start - "The application failed to initialize" or an instant silent exit - the problem is usually a DLL, and the tool is **Dependency Walker**. Open your .exe in it and it shows the tree of DLLs the loader will look for; anything missing is flagged in red. The usual suspects here: `libGLESv2.dll`, `libEGL.dll`, and `bass.dll` must sit next to your .exe. ANGLE's DLLs additionally pull in the D3DX/D3DCompiler DLL they were built against; Dependency Walker will name it exactly, and it ships in this kit. Check with it on day 30 before you zip anything.

And the errors you will actually hit, with their actual causes in this project:

| Error | What it says | What it usually means here |
|---|---|---|
| `LNK2001` | unresolved external symbol | A .c file exists but was never added to the project (Project, Add To Project, Files), or a .lib is missing from Project, Settings, Link (`bass.lib` is the usual one). If the mangled name in the message looks like `?Foo@@YA...`, a file is compiling as C++ - rename it to .c. |
| `LNK1104` | cannot open file | If it names your .exe: the program is still running - close it (check the taskbar for a hidden fullscreen instance). If it names a .lib: the library directory is not listed under Tools, Options, Directories, Library files, or the name is misspelled. |
| `C2065` | undeclared identifier | A missing `#include`, a typo, or a constant from a newer SDK than VC6's headers know. For GL enums, take the value from the appendix tables and `#define` it yourself. |

| | | |
|---|---|---|
| `C2143` | syntax error : missing ';' before 'type' | The C89 classic: a declaration after a statement. Move every declaration to the top of its block. Also appears when the previous header is missing its final semicolon - check the last struct in the last thing you included. |

## The Budget Card

Finally, the numbers to design against. Pin these where you can see them; every "will this be fast enough?" question for the next thirty days is answered by arithmetic against this table, not by guessing.

| Resource | Budget | Notes |
|---|---|---|
| Frame time | 16.7 ms (60 fps) | Miss it and the vsync ladder drops you to 33.3 ms. Design for ~14 ms so hovering GDI cannot push you over. |
| Memory bandwidth | low hundreds of MB/s (memcpy) | At 60 fps that is roughly 2-4 MB moved per frame, total, for everything: dynamic vertex buffers, texture uploads, your own copies. Day 18's particle buffer lives inside this number. |
| Draw calls | low hundreds per frame | Each call crosses ANGLE and D3D9 on a 1.6 GHz Atom. 200 is comfortable; 500 is asking for trouble. Batch by material. |
| Triangles | ~50-100k per frame | Vertex-shader and binning bound. Day 11 will find your real number; write it in the journal. |
| Textures | max 1024x1024, DXT when possible | DXT1 is 8x smaller than RGBA8 and the GPU samples it natively. Mipmap everything you minify. |
| RAM | engine under ~256 MB | 1 GB total, minus XP, minus the GPU's shared-memory carve-out, minus room to run PSP7 and Milkshape alongside. "Video memory" is this same RAM - textures count twice, CPU copy and GPU copy. |

These are planning numbers, not physical constants - each is the round figure to hold in your head while sketching a system. When a real decision hangs on one, measure it on this machine and journal the result. By day 30 your journal.txt will be a better reference for this hardware than any spec sheet ever printed.

**DONE WHEN:** the skeleton runs fullscreen via the listing above, Alt-Tab minimizes and restores it cleanly with the desktop mode intact afterward, and `journal.txt` exists with line zero: today's date and the fps the empty scene reports.

**IF YOU ARE STUCK:**

- Window appears then vanishes instantly? Run under F5 - an access violation at address zero means a GL call before `gl_loader_init()`.
- Taskbar still visible over your window? The window must be exactly screen-sized at (0,0) with `WS_POPUP` and no border styles; any border makes XP treat it as an ordinary window.
- Desktop stuck at 800x600 after a crash? Run a two-line .exe that calls `ChangeDisplaySettings(NULL, 0)` - and now go make `Fullscreen_Restore` unmissable on every exit path.
- Engine at 30 fps doing nothing? Check that only one `eglSwapBuffers` runs per loop, and that nothing animated (clock, blinking caret, another app) is painting over you.

# The Quest Log

The original thirty-day challenge list, exactly as it shipped on the USB stick. Treat it as your checklist; the chapters that follow are the field notes for each day.

```
THE ENGINE QUEST
30 days from a spinning triangle to a turn-of-the-millennium engine.
Hardware: Eee PC X101CH. OS: Windows XP. Compiler: VC6. Language: C89.
GPU: PowerVR SGX545 pretending to be a GMA 3600, driven over ANGLE.

Rules of the game:
- Everything from scratch except stb_image, BASS, and the GL loader.
- The fps counter stays in the title bar. Know your budget every day.
- Keep a journal.txt. One line per day minimum. Future you will grin.
- A day is done when its DONE line is true. Stretch goals are optional.
- When stuck, ask: how did they do it in 1999 with 8 MB of VRAM?

PHASE 1 - FOUNDATIONS (days 1-6)

Day 1: mathlib. vec3 as float[3], the classic macros (VectorAdd,
  DotProduct, CrossProduct, VectorNormalize), mat4 as float[16],
  mat4_identity/multiply/perspective/lookat/rotate/translate.
  DONE: unit tests in a console exe pass. Yes, write the tests.
Day 2: colored cube. Interleaved vertex buffer, index buffer,
  depth buffer on (EGL_DEPTH_SIZE), your mat4 as uniforms.
  DONE: cube spins without clipping through itself.
Day 3: fly camera. WASD + mouse look via Win32 raw input or
  GetCursorPos/SetCursorPos recentering. Your lookat matrix, not
  anyone else's.
  DONE: you can orbit and strafe around the cube smoothly.
Day 4: TGA loader, by hand. Uncompressed + RLE, 24/32-bit.
  glTexImage2D + mipmaps (glGenerateMipmap). Texture the cube.
  DONE: a hand-drawn PSP7 texture on every face, no color swap bugs
  (BGR vs RGB - the rite of passage).
Day 5: stb_image joins. PNG/JPG loading behind your own
  Texture_Load() so formats are one call. Texture cache: loading the
  same path twice returns the same GL texture.
  DONE: cube wears a JPG photo; cache proven by a debug counter.
Day 6: blending + 2D overlay. Alpha blending, an orthographic
  projection pass, and a bitmap font (grid font texture, one quad per
  glyph). Draw the fps yourself - the title bar retires.
  DONE: text overlay on 3D scene, correct alpha edges.

PHASE 2 - MESHES AND WORLDS (days 7-12)

Day 7: OBJ loader. v/vt/vn/f, 1-based indices, fan-triangulate
  n-gons, and THE dedup of v/vt/vn triples into one vertex buffer.
  DONE: a Milkshape-exported model renders with correct normals.
Day 8: scene graph, 1999 edition. Node = transform + mesh + children.
  Parent-child matrices multiply down the tree.
  DONE: a moon orbits a planet orbiting a sun, one matrix stack, no
  gimbal surprises.
Day 9: frustum culling. Extract 6 planes from view*proj, sphere test
  per node. Draw count in the overlay.
  DONE: looking away from 100 cubes drops draws to near zero.
Day 10: skybox. Cubemap from 6 images, drawn first with depth writes
  off, camera translation stripped from the view matrix.
  DONE: infinite sky, no seams, world floats in it.
Day 11: heightmap terrain. Grayscale image -> grid mesh, per-vertex
  normals from central differences, one big texture or tiling detail.
  DONE: you can fly over rolling hills at interactive fps; journal
  the triangle count where the Atom starts to sweat.
```

```
Day 12: terrain chunks. Split into 33x33 chunks, cull per-chunk with
  day 9. This is how big worlds fit in 1999.
  DONE: draw count falls as you look along the valley.

PHASE 3 - LIGHT AND ATMOSPHERE (days 13-18)

Day 13: per-vertex lighting. Directional sun, ambient, N.L in the
  vertex shader - literally the fixed-function pipeline you were
  denied, reimplemented by you.
  DONE: terrain shading changes as you rotate the sun in real time.
Day 14: per-pixel + point lights. Move N.L to the fragment shader,
  add 2 point lights with distance attenuation, uniforms fed from C.
  DONE: a light orbits the OBJ model, falloff looks right.
Day 15: DOT3 normal mapping. Tangent space on the OBJ loader
  (per-face tangents averaged), a normal map painted or converted in
  PSP7 - bumpy surfaces, year-2001 flagship feature.
  DONE: flat quad looks embossed from every light angle.
Day 16: lightmaps. Second UV set, offline bake in a tool exe you
  write (even a simple sky-visibility/AO bake counts), multitexture
  modulate in shader. The Quake look.
  DONE: a small room scene with baked soft corners, dynamic lights
  layered on top.
Day 17: fog + time of day. Exp2 fog in the shader (recreate GL_FOG
  faithfully), sun color lerps across a day cycle, sky tint follows.
  DONE: dawn over the terrain. Screenshot it. That is the desktop
  wallpaper now.
Day 18: particles. Point-sprite-or-billboard quads, one dynamic
  vertex buffer rebuilt per frame, additive blending. Emitters:
  fountain, smoke, spark burst.
  DONE: 1000 particles at playable fps; journal the number where it
  stops being playable.

PHASE 4 - ENGINE SYSTEMS (days 19-24)

Day 19: the console. Tilde drops a Quake-style console over the
  scene: text input, command table, cvars (get/set), "map", "fog",
  "r_wireframe". Every engine of the era had one. Yours does now.
  DONE: change fog density from the console at runtime.
Day 20: resources + config. Refcounted resource manager for
  textures/meshes/sounds, engine.cfg parsed at boot (resolution,
  fov, binds), console command "exec".
  DONE: no leaks after loading and unloading a scene 100 times
  (write the leak counter proof).
Day 21: BASS day. Music: an .xm or .it you tracked in ModPlug 1.12,
  streamed. SFX: wav samples with distance volume + stereo pan from
  listener position.
  DONE: your own module loops over the terrain; footsteps pan as
  things pass by.
Day 22: collision. AABB world geometry, swept sphere vs AABB player,
  slide along walls, gravity + jump. The player stops being a ghost.
  DONE: you can walk into the day-16 room, jump on a crate, and not
  fall through the floor after 10 minutes of trying.
Day 23: MD2 loader. Quake II's model format, 1997's finest: keyframe
  vertex animation, lerp between frames in the vertex shader (two
  position attributes, mix by uniform).
  DONE: a period MD2 (or Milkshape export) runs its walk cycle.
Day 24: MS3D skeletal animation. Parse Milkshape's native format:
  joints, keyframes, vertex weights (1 bone per vertex is period-
  correct). Skin on the CPU in C89 or in-shader with a bone palette.
  DONE: a skeleton-animated character walks the terrain; compare fps
  vs the MD2 - journal which the Atom prefers.

PHASE 5 - THE BOSS FIGHTS (days 25-30)

Day 25: render to texture. FBO: mirror or security-monitor in the
  room scene, then a full-screen postprocess pass (vignette or
```

```
  sepia).
  DONE: mirror reflects the animated character in real time.
Day 26: water. A plane with the day-25 reflection, dudv/normal-map
  distortion scrolling, fresnel-ish blend by view angle. The 2002
  screenshot feature.
  DONE: terrain pond you would have posted on a forum in 2002.
Day 27: visibility II. Portals between rooms or occlusion queries
  (GL_EXT_occlusion_query_boolean is in your extension list) for the
  crates. Overlay shows objects skipped.
  DONE: standing in room A, room B costs (nearly) nothing.
Day 28: game feel. Weapon viewmodel bobbing in the corner, crosshair,
  HUD from day 6 font, muzzle-flash light + particle burst + BASS
  bang on click. Hitscan against the AABB world.
  DONE: shooting a crate knocks a decal or scorch quad onto it.
Day 29: the level. Glue everything: sky, terrain, room, water,
  characters, music, console. Author a real level layout. Hunt one
  performance problem with the overlay counters and fix it.
  DONE: two minutes of continuous play without a crash, out-of-
  memory, or fps collapse.
Day 30: SHIP IT. Name the engine. Splash screen (PSP7), readme.txt,
  version string in the console, zip with all DLLs, run it on a
  clean XP login like a 1999 shareware demo disc.
  DONE: someone else launches it from the zip with zero help and
  walks around your level.

EXTRA CREDIT (the new game+):
- Stencil shadows (ask EGL for stencil bits first) - Doom 3 called.
- Geomipmapped terrain LOD with skirts.
- Quake3-style material scripts parsed from text.
- A second player: LAN sync over UDP, the 2001 way.
- Port day 30's demo to the Arch box with the same C89 codebase and
  the cross-compiler - one engine, two centuries.

Good luck. Leave no day half-done; half-done days are how engines
become folders named engine_old_final_2.
```

Phase 1 is six days of foundations: math you can prove, a cube you can see, a camera you can steer, and pixels you can load, cache, and draw as text. Nothing here is glamorous. Everything here is load-bearing. Every one of the remaining 24 days sits on what you build this week, so take the DONE lines seriously and write the journal line even when it is just "cube spins, life is good."

# Day 1: The Mathlib

## What you are building, and why 1999 cared

Today you write `mathlib.c`: a vec3 type with the classic Quake-style macros, and a 4x4 matrix type with identity, multiply, perspective, lookat, translate, rotate, and scale. Then you prove it works with a console test program before a single pixel depends on it.

In 1999 every engine shipped its own math library, and there was no choice about it: no templates you would trust, no SIMD intrinsics worth the trouble, no vendor library. Quake's `mathlib.c` is the pattern this whole book follows. You have an extra reason they did not: OpenGL ES 2.0 has no matrix stack. There is no `glTranslatef`, no `glFrustum`, no `glPushMatrix`. The driver will accept a 16-float uniform and nothing else. From today onward, *you* are the matrix stack.

Why tests, on day one, when there is nothing to look at? Because a matrix bug does not crash. It renders a scene that is subtly, maddeningly wrong — a cube that shears instead of spins, a camera that orbits the wrong axis — and you will lose day 3 to it unless you pin the math down while it is still cheap to test.

## Theory: one convention, held with a death grip

Every matrix bug you will ever have is a convention bug. Ours, fixed for the rest of the book:

A mat4 is `float m[16]` in **column-major** order, the memory order OpenGL has always used. Column `c`, row `r` lives at `m[c*4+r]`. The first four floats are the first *column*, not the first row:

```
memory:  m[0] m[1] m[2] m[3] | m[4] m[5] m[6] m[7] | m[8] ...
         '---- column 0 ----'  '---- column 1 ----'

written as math, row r / column c sits at m[c*4+r]:

     | m[0]  m[4]  m[8]   m[12] |     m[12] = translation x
 M = | m[1]  m[5]  m[9]   m[13] |     m[13] = translation y
     | m[2]  m[6]  m[10]  m[14] |     m[14] = translation z
     | m[3]  m[7]  m[11]  m[15] |
```

Vectors are columns and transform as `out = M * v`. Composition therefore reads right to left: `mvp = P * V * M` means the vertex meets the model matrix first, then the view, then the projection. The world is right-handed, +Y is up, and the camera looks down −Z. ANGLE translates our clip-space conventions to D3D9 behind our backs; treat everything as classic OpenGL and never think about it again.

### The perspective matrix, derived

Put the camera at the origin looking down −Z. A point at eye space `(x, y, z)` (with `z` negative in front of us) projects onto the near plane by similar triangles: the projected y is `y*n/(-z)`. The vertical field of view says the visible half-height at distance `n` is `t = n*tan(fovy/2)`, and NDC wants that mapped to [−1, +1]. Divide through:

```
y_ndc = y * n / (-z * t) = f * y / (-z)      where f = 1/tan(fovy/2)
x_ndc = (f/aspect) * x / (-z)                (aspect widens x)
```

The GPU does the divide for us: it divides clip coordinates by `w` after the vertex shader. So we set `w = -z` (that is the entire job of the bottom row, `0 0 -1 0`) and output `y_clip = f*y`. Depth must survive the same divide, so it has the

form `z_clip = A*z + B` with `z_ndc = (A*z+B)/(-z)`. Forcing `z = -n` to map to −1 and `z = -f_far` to +1 and solving the two equations gives:

```
  A = (zfar + znear) / (znear - zfar)
  B = (2 * zfar * znear) / (znear - zfar)

      | f/aspect  0     0     0 |        m[0]  = f/aspect
  P = |    0      f     0     0 |        m[5]  = f
      |    0      0     A     B |        m[10] = A     m[14] = B
      |    0      0    -1     0 |        m[11] = -1
```

Note what fell out of the algebra: `z_ndc` is *not* linear in eye z. It is a hyperbola that spends most of its precision near the near plane. This is why you never set `znear` to 0.001 "to be safe" — you would be spending your entire depth buffer on the first few centimeters. 0.1 near / 100 far is fine for this engine.

### Lookat is the camera's transform, inverted

Build the camera's basis: forward `f` points from eye to target, side `s = normalize(f × up)`, and true up `u = s × f`. The view matrix must move the world so the camera sits at the origin looking down −Z; that is the *inverse* of the camera's own transform. For a rotation the inverse is the transpose, so the basis vectors become the *rows* of the matrix, and the translation column becomes the eye position dotted against each basis vector, negated. You will see this shape in the listing.

## The code

First the header. The macros put the **result last**, Quake style, and `DotProduct` is an expression, not a statement:

```
/* mathlib.h */
#ifndef MATHLIB_H
#define MATHLIB_H

typedef float vec3_t[3];

#define DEG2RAD(d) ((d) * 0.017453293f)

#define DotProduct(a,b) \
    ((a)[0]*(b)[0] + (a)[1]*(b)[1] + (a)[2]*(b)[2])

#define VectorAdd(a,b,out) \
    ((out)[0] = (a)[0] + (b)[0], \
     (out)[1] = (a)[1] + (b)[1], \
     (out)[2] = (a)[2] + (b)[2])

#define VectorSubtract(a,b,out) \
    ((out)[0] = (a)[0] - (b)[0], \
     (out)[1] = (a)[1] - (b)[1], \
     (out)[2] = (a)[2] - (b)[2])

#define VectorScale(v,s,out) \
    ((out)[0] = (v)[0] * (s), \
     (out)[1] = (v)[1] * (s), \
     (out)[2] = (v)[2] * (s))

#define VectorCopy(a,out) \
    ((out)[0] = (a)[0], (out)[1] = (a)[1], (out)[2] = (a)[2])

#define CrossProduct(a,b,out) \
    ((out)[0] = (a)[1]*(b)[2] - (a)[2]*(b)[1], \
     (out)[1] = (a)[2]*(b)[0] - (a)[0]*(b)[2], \
```

```
     (out)[2] = (a)[0]*(b)[1] - (a)[1]*(b)[0])

float VectorNormalize(vec3_t v);   /* in place; returns old length */

void mat4_identity(float *m);
void mat4_multiply(float *out, const float *a, const float *b);
void mat4_perspective(float *out, float fovy_deg, float aspect,
                      float znear, float zfar);
void mat4_lookat(float *out, const vec3_t eye, const vec3_t center,
                 const vec3_t up);
void mat4_translate(float *out, float x, float y, float z);
void mat4_rotate_x(float *out, float deg);
void mat4_rotate_y(float *out, float deg);
void mat4_rotate_z(float *out, float deg);
void mat4_scale(float *out, float x, float y, float z);

#endif /* MATHLIB_H */
```

**WARNING:** Two macro traps. First, `CrossProduct(a, b, a)` is a silent bug: the macro writes `out[0]` before it reads `a[0]` again. The output of a cross product must be a third vector. Second, macros evaluate their arguments more than once, so never pass an expression with side effects — `DotProduct(v++, w)` is a rite you do not want to pass through.

Now the hard 60% of `mathlib.c`. VC6's C mode has no `sqrtf`/`tanf`/`cosf` — those are C99 — so call the double versions and cast:

```
/* mathlib.c */
#include <math.h>
#include "mathlib.h"

float VectorNormalize(vec3_t v)
{
    float len, inv;
    len = (float)sqrt(DotProduct(v, v));
    if (len > 0.0f) {
        inv = 1.0f / len;
        v[0] *= inv; v[1] *= inv; v[2] *= inv;
    }
    return len;
}

void mat4_identity(float *m)
{
    int i;
    for (i = 0; i < 16; ++i) m[i] = 0.0f;
    m[0] = m[5] = m[10] = m[15] = 1.0f;
}

/* out = a * b.  out MUST NOT alias a or b. */
void mat4_multiply(float *out, const float *a, const float *b)
{
    int c, r, k;
    for (c = 0; c < 4; ++c) {
        for (r = 0; r < 4; ++r) {
            float sum = 0.0f;
            for (k = 0; k < 4; ++k)
                sum += a[k*4 + r] * b[c*4 + k];
            out[c*4 + r] = sum;
        }
    }
}

void mat4_perspective(float *out, float fovy_deg, float aspect,
```

```
                        float znear, float zfar)
{
    float f;
    int i;
    f = 1.0f / (float)tan(DEG2RAD(fovy_deg) * 0.5f);
    for (i = 0; i < 16; ++i) out[i] = 0.0f;
    out[0]  = f / aspect;
    out[5]  = f;
    out[10] = (zfar + znear) / (znear - zfar);
    out[11] = -1.0f;
    out[14] = (2.0f * zfar * znear) / (znear - zfar);
}

void mat4_lookat(float *out, const vec3_t eye, const vec3_t center,
                 const vec3_t up)
{
    vec3_t f, s, u;
    VectorSubtract(center, eye, f);
    VectorNormalize(f);
    CrossProduct(f, up, s);
    VectorNormalize(s);
    CrossProduct(s, f, u);

    out[0] =  s[0]; out[4] =  s[1]; out[8]  =  s[2];
    out[1] =  u[0]; out[5] =  u[1]; out[9]  =  u[2];
    out[2] = -f[0]; out[6] = -f[1]; out[10] = -f[2];
    out[3] = 0.0f;  out[7] = 0.0f;  out[11] = 0.0f;
    out[12] = -DotProduct(s, eye);
    out[13] = -DotProduct(u, eye);
    out[14] =  DotProduct(f, eye);
    out[15] = 1.0f;
}

void mat4_rotate_y(float *out, float deg)
{
    float c, s;
    c = (float)cos(DEG2RAD(deg));
    s = (float)sin(DEG2RAD(deg));
    mat4_identity(out);
    out[0] =  c;  out[8]  = s;
    out[2] = -s;  out[10] = c;
}
```

Left for you: `mat4_translate` (identity, then fill `m[12..14]`), `mat4_scale` (fill the diagonal), and `mat4_rotate_x`/`_z`, which are `rotate_y` with the sine/cosine pairs moved to rows/columns (1,2) and (0,1) respectively. Ten minutes, and the tests below will tell you if you got a sign wrong.

## Prove it: the test exe

Create a second VC6 project (Win32 Console Application) in the same workspace, add `mathlib.c` to it, and write `mathtest.c`. A test helper that applies M to a 4-component point is all the harness you need:

```
/* mathtest.c (excerpts) */
static int g_fails;

static int feq(float a, float b)
{
    return fabs(a - b) < 1.0e-4;
}

static void mul_point(const float *m, const float *v, float *out)
{
```

```
    int r;
    for (r = 0; r < 4; ++r)
        out[r] = m[r]*v[0] + m[4+r]*v[1] + m[8+r]*v[2] + m[12+r]*v[3];
}

static void check(const char *name, int cond)
{
    printf("%-44s %s\n", name, cond ? "ok" : "FAIL");
    if (!cond) ++g_fails;
}

int main(void)
{
    float m[16], v[4], o[4];
    vec3_t x = { 1.0f, 0.0f, 0.0f };
    vec3_t y = { 0.0f, 1.0f, 0.0f };
    vec3_t c;

    /* 1: the coordinate system is right-handed */
    CrossProduct(x, y, c);
    check("cross(X,Y) == Z",
          feq(c[0], 0.0f) && feq(c[1], 0.0f) && feq(c[2], 1.0f));

    /* 2: near/far planes land on z_ndc = -1 and +1 */
    mat4_perspective(m, 60.0f, 1024.0f / 600.0f, 0.1f, 100.0f);
    v[0] = 0.0f; v[1] = 0.0f; v[2] = -0.1f; v[3] = 1.0f;
    mul_point(m, v, o);
    check("perspective: near -> z_ndc -1", feq(o[2] / o[3], -1.0f));
    v[2] = -100.0f;
    mul_point(m, v, o);
    check("perspective: far  -> z_ndc +1", feq(o[2] / o[3],  1.0f));

    /* 3: top edge of the near plane -> y_ndc = +1 */
    v[1] = 0.1f * (float)tan(DEG2RAD(30.0f));    /* fovy/2 = 30 deg */
    v[2] = -0.1f;
    mul_point(m, v, o);
    check("perspective: top of near -> y_ndc +1", feq(o[1]/o[3], 1.0f));

    /* 4: lookat maps eye to origin and target onto -Z */
    {
        vec3_t eye = { 0.0f, 0.0f, 5.0f };
        vec3_t ctr = { 0.0f, 0.0f, 0.0f };
        vec3_t up  = { 0.0f, 1.0f, 0.0f };
        mat4_lookat(m, eye, ctr, up);
        v[0] = 0.0f; v[1] = 0.0f; v[2] = 5.0f; v[3] = 1.0f;
        mul_point(m, v, o);
        check("lookat: eye -> origin",
              feq(o[0], 0.0f) && feq(o[1], 0.0f) && feq(o[2], 0.0f));
        v[2] = 0.0f;
        mul_point(m, v, o);
        check("lookat: target -> (0,0,-5)", feq(o[2], -5.0f));
    }

    /* 5: rotate_y(90) turns +X into -Z */
    mat4_rotate_y(m, 90.0f);
    v[0] = 1.0f; v[1] = 0.0f; v[2] = 0.0f; v[3] = 1.0f;
    mul_point(m, v, o);
    check("rotate_y(90): X -> -Z",
          feq(o[0], 0.0f) && feq(o[2], -1.0f));

    printf("%d failure(s)\n", g_fails);
    return g_fails;
}
```

Before you trust a green run, break it on purpose: swap two indices in `mat4_multiply` and confirm tests fail. A test suite that cannot fail is decoration.

## Pitfalls

**The transpose bug.** If you index `m[r*4+c]` anywhere, everything still "works" for symmetric matrices and then falls apart the moment translation enters. Test 4 exists to catch exactly this.

**Aliasing.** `mat4_multiply(m, m, b)` reads `a` while overwriting it. The comment in the listing is a contract; when you need in-place multiply later, use a local temp.

**Degrees vs radians.** Our public API takes degrees (matching the old `glRotatef` habit); `sin`/`cos` take radians. Convert exactly once, at the API boundary.

**Numeric slack.** VC6 does float math on the x87 unit at extended precision, so results differ in the last bits from what you might compute elsewhere. The `1e-4` epsilon absorbs it. Performance does not matter today — a few hundred matrix multiplies per frame will not scratch the Atom — correctness does.

**DONE WHEN:** the console exe prints `ok` for every test and `0 failure(s)`, and deliberately breaking one function makes it fail. Run it from a command prompt and check `echo %ERRORLEVEL%` prints 0.

**IF YOU ARE STUCK:**

- Print a matrix as four lines of four — remember each *column* is contiguous in memory, so print `m[0] m[4] m[8] m[12]` on the first line.
- Work `rotate_y(90)` on paper: +X must become −Z in a right-handed, +Y-up world. If you got +Z, negate the sine terms.
- If test 2 fails at the far plane only, you have `A` and `B` swapped or a sign flipped in `znear - zfar`.
- If lookat sends the eye to `(0,0,-5)` instead of the origin, you forgot to negate the translation dot products.

# Day 2: The Colored Cube

## What you are building, and why 1999 cared

Today: a spinning cube with a different color at each corner, drawn through the full modern pipeline — vertex buffer, index buffer, two shaders, one `mat4` uniform, depth testing. In 1999 you would have typed `glBegin(GL_QUADS)` and been done in ten minutes; you have been denied that on purpose. GLES2 has no immediate mode and no fixed function, so the buffer-and-shader path you build today is not a stepping stone — it is the exact shape of every draw call for the next 28 days. The cube is the "hello, world" that proves the whole chain: your math from day 1, the GPU's rasterizer, and the depth buffer, all agreeing.

## Theory: vertices in one buffer, welded by indices

A vertex today is six floats: position xyz, color rgb, interleaved in one array. Interleaving matters on real hardware: when the GPU fetches vertex 5, everything it needs sits in one contiguous 24-byte slab instead of being gathered from two arrays.

```
byte:    0           12          24          36
         | x  y  z   | r  g  b   | x  y  z   | r  g  b   | ...
         '------ vertex 0 -------' '------ vertex 1 -------'

attribute a_pos   : 3 floats at offset  0 --.
attribute a_color : 3 floats at offset 12 --+-- stride = 24 bytes
```

A cube has 8 corners but 36 triangle-vertices (6 faces × 2 triangles × 3). The index buffer lets you store the 8 corners once and reference them 36 times — the difference between 192 bytes and 864 is nothing here, but on day 7 when a model has 5,000 vertices shared by 30,000 triangle slots it becomes the difference between fitting in the SGX's vertex cache and not.

Winding is a contract: every triangle is listed counter-clockwise as seen from *outside* the cube. GL calls those front faces (the default `glFrontFace(GL_CCW)`), and with `glEnable(GL_CULL_FACE)` the GPU throws away the three faces pointing away from you before rasterizing a single pixel.

### The pointer that is secretly an integer

The last parameter of `glVertexAttribPointer` is typed `const void *` for historical reasons: in 1997 it really was a pointer into your memory. With a VBO bound it is reinterpreted as a *byte offset* into the buffer. C89 gives you no `uintptr_t`, so use the idiom that is safe on VC6 and everywhere else — pointer arithmetic from a null `char *`:

```
#define BUFFER_OFFSET(n) ((const void *)((const char *)0 + (n)))
```

The same idiom feeds the last argument of `glDrawElements`, which is a byte offset into the bound index buffer.

### The depth buffer: ask for it, then turn it on

Two separate steps, and forgetting either produces the same bug. The depth buffer must *exist*: add `EGL_DEPTH_SIZE, 24` to your `eglChooseConfig` attribute list (fall back to 16 if no config comes back). And it must be *used*: `glEnable(GL_DEPTH_TEST)`, and clear it every frame with `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`. Miss any of these and far faces draw over near ones whenever the draw order disagrees with the view — the cube becomes an Escher exhibit as it rotates.

## The code

The full cube. Colors are position + 0.5, which paints the corners of the RGB cube onto the corners of your cube — a free visual checksum that XYZ and RGB agree:

```
/* cube.c - data */
static const float CUBE_VERTS[8 * 6] = {
    /*   x      y      z       r     g     b  */
    -0.5f, -0.5f, -0.5f,   0.0f, 0.0f, 0.0f,
     0.5f, -0.5f, -0.5f,   1.0f, 0.0f, 0.0f,
     0.5f,  0.5f, -0.5f,   1.0f, 1.0f, 0.0f,
    -0.5f,  0.5f, -0.5f,   0.0f, 1.0f, 0.0f,
    -0.5f, -0.5f,  0.5f,   0.0f, 0.0f, 1.0f,
     0.5f, -0.5f,  0.5f,   1.0f, 0.0f, 1.0f,
     0.5f,  0.5f,  0.5f,   1.0f, 1.0f, 1.0f,
    -0.5f,  0.5f,  0.5f,   0.0f, 1.0f, 1.0f
};

/* CCW seen from outside */
static const unsigned short CUBE_IDX[36] = {
    4, 5, 6,   6, 7, 4,     /* front  +Z */
    0, 3, 2,   2, 1, 0,     /* back   -Z */
    0, 4, 7,   7, 3, 0,     /* left   -X */
    1, 2, 6,   6, 5, 1,     /* right  +X */
    0, 1, 5,   5, 4, 0,     /* bottom -Y */
    3, 7, 6,   6, 2, 3      /* top    +Y */
};
```

The smallest useful pair of GLSL ES 1.00 shaders. String arrays, compiled at startup:

```
static const char *VS_SRC =
    "attribute vec3 a_pos;\n"
    "attribute vec3 a_color;\n"
    "uniform mat4 u_mvp;\n"
    "varying vec3 v_color;\n"
    "void main() {\n"
    "    v_color = a_color;\n"
    "    gl_Position = u_mvp * vec4(a_pos, 1.0);\n"
    "}\n";

static const char *FS_SRC =
    "precision mediump float;\n"
    "varying vec3 v_color;\n"
    "void main() {\n"
    "    gl_FragColor = vec4(v_color, 1.0);\n"
    "}\n";

static GLuint Shader_Compile(GLenum type, const char *src)
{
    GLuint sh;
    GLint ok;
    sh = glCreateShader(type);
    glShaderSource(sh, 1, &src, NULL);
    glCompileShader(sh);
    glGetShaderiv(sh, GL_COMPILE_STATUS, &ok);
    if (!ok) {
        char log[1024];
        glGetShaderInfoLog(sh, sizeof(log), NULL, log);
        MessageBoxA(NULL, log, "shader compile failed", MB_OK);
    }
```

```
    return sh;
}
```

Setup and draw. Note that attribute locations are *queried*, never assumed:

```
static GLuint g_prog, g_vbo, g_ibo;
static GLint  g_umvp, g_apos, g_acolor;

void Cube_Init(void)
{
    GLuint vs, fs;
    GLint ok;
    vs = Shader_Compile(GL_VERTEX_SHADER,   VS_SRC);
    fs = Shader_Compile(GL_FRAGMENT_SHADER, FS_SRC);
    g_prog = glCreateProgram();
    glAttachShader(g_prog, vs);
    glAttachShader(g_prog, fs);
    glLinkProgram(g_prog);
    glGetProgramiv(g_prog, GL_LINK_STATUS, &ok);
    if (!ok) MessageBoxA(NULL, "program link failed", "GL", MB_OK);

    g_umvp   = glGetUniformLocation(g_prog, "u_mvp");
    g_apos   = glGetAttribLocation(g_prog, "a_pos");
    g_acolor = glGetAttribLocation(g_prog, "a_color");

    glGenBuffers(1, &g_vbo);
    glBindBuffer(GL_ARRAY_BUFFER, g_vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof(CUBE_VERTS), CUBE_VERTS,
                 GL_STATIC_DRAW);

    glGenBuffers(1, &g_ibo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_ibo);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(CUBE_IDX), CUBE_IDX,
                 GL_STATIC_DRAW);

    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}

void Cube_Draw(float angle_deg)
{
    float model[16], view[16], proj[16], tmp[16], mvp[16];
    static const vec3_t eye    = { 1.6f, 1.2f, 1.6f };
    static const vec3_t center = { 0.0f, 0.0f, 0.0f };
    static const vec3_t up     = { 0.0f, 1.0f, 0.0f };

    mat4_rotate_y(model, angle_deg);
    mat4_lookat(view, eye, center, up);
    mat4_perspective(proj, 60.0f, 1024.0f / 600.0f, 0.1f, 100.0f);
    mat4_multiply(tmp, view, model);
    mat4_multiply(mvp, proj, tmp);        /* P * V * M */

    glUseProgram(g_prog);
    glUniformMatrix4fv(g_umvp, 1, GL_FALSE, mvp);

    glBindBuffer(GL_ARRAY_BUFFER, g_vbo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_ibo);
    glEnableVertexAttribArray((GLuint)g_apos);
    glEnableVertexAttribArray((GLuint)g_acolor);
    glVertexAttribPointer((GLuint)g_apos,   3, GL_FLOAT, GL_FALSE,
                          24, BUFFER_OFFSET(0));
    glVertexAttribPointer((GLuint)g_acolor, 3, GL_FLOAT, GL_FALSE,
                          24, BUFFER_OFFSET(12));
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT,
```

```
                    BUFFER_OFFSET(0));
}
```

Left to you: the angle. Add `g_angle += 45.0f * dt;` in `engine_update` (45 degrees per second), pass it to `Cube_Draw` from `engine_render`. Spin lives in update ticks, not render frames — that discipline pays off tomorrow.

## Pitfalls

**Black screen.** In order of likelihood: a shader failed to compile (the MessageBox tells you; ANGLE's GLSL-to-HLSL translator is pickier than you expect — the `precision` line in the fragment shader is mandatory); an attribute location came back −1; you set the uniform before `glUseProgram`.

**Location −1.** `glGetAttribLocation` returns −1 for any attribute the compiler optimized away. If you comment out the color output while debugging, `a_color` vanishes and enabling array −1 generates GL errors. Check locations once at init and fail loudly.

**Stride and offsets are bytes.** 24 and 12 — not 6 and 3. If the cube renders as a shredded fan of triangles, recount your bytes.

**Cube looks inside-out.** Faces you are looking at are being culled: your winding is clockwise somewhere. Fix the index order, do not switch `glFrontFace` — the book's data all assumes CCW.

**Self-clipping cube.** The DONE-line bug: no `EGL_DEPTH_SIZE` in the config attributes means you got a config with zero depth bits, and `glEnable(GL_DEPTH_TEST)` on a depthless surface is a polite no-op. Also confirm the depth bit is in your `glClear` mask.

Performance: 12 triangles is a rounding error even for this GPU. With `eglSwapInterval(1)` you should sit locked at 60 in the title bar.

**DONE WHEN:** the cube spins with every face a smooth color gradient, near faces always covering far ones, at a steady 60 fps.

**IF YOU ARE STUCK:**

- Call `glGetError` after init and after the first draw; a 1280/1281/1282 narrows things fast (see the GL error table in the appendices).
- Shrink the problem: draw with `glDrawElements(..., 3, ...)` — one triangle. No triangle? The problem is pipeline, not data.
- Replace `mvp` with identity and move a vertex to `(0, 0.5, 0)` in the data. If it appears, your matrix chain is the suspect — back to the day-1 tests.
- Escher cube: print the number of depth bits with `eglQuerySurface` / `eglGetConfigAttrib`. If it says 0, your config list is wrong.

# Day 3: The Fly Camera

## What you are building, and why 1999 cared

WASD plus mouse look: the noclip camera. Every engine of the era grew one before it grew a game, because the camera is the debug tool you use every remaining day — you cannot judge culling, LOD, fog, or a skybox from a fixed viewpoint. Today the `mat4_lookat` you proved on day 1 earns its keep: you feed it an eye position and a target you compute yourself from two angles. Nobody else's lookat, as the challenge sheet says.

## Theory: two angles are enough

A fly camera is a position plus two Euler angles: **yaw** (turn left/right around +Y) and **pitch** (look up/down). No roll — FPS cameras never roll, and dropping it kills gimbal problems before they start. The convention: yaw 0 looks down −Z, positive yaw turns right, positive pitch looks up. From those angles the forward vector is a cosine/sine lattice you should verify on paper once:

```
forward.x =  cos(pitch) * sin(yaw)
forward.y =  sin(pitch)
forward.z = -cos(pitch) * cos(yaw)

right.x   =  cos(yaw)          /* forward x up, already unit */
right.y   =  0
right.z   =  sin(yaw)
```

Check: yaw 0, pitch 0 gives forward (0, 0, −1) and right (1, 0, 0). Yaw 90° gives forward (1, 0, 0) — you turned right. The view matrix is then one call: `mat4_lookat(view, pos, pos + forward, up)`.

**Clamp pitch to ±89°.** At exactly ±90° forward is parallel to up, the cross product inside lookat collapses to zero length, and the view matrix fills with NaNs. The classic symptom is a screen that goes black only when you stare at your feet.

### Mouse look by recentering

You need relative mouse motion, but Windows hands you an absolute cursor. The period-correct trick: every frame, read the cursor with `GetCursorPos`, subtract the window center to get a delta, then warp the cursor back to center with `SetCursorPos`. The cursor never reaches the screen edge, so the deltas never run dry. Hide it with `ShowCursor(FALSE)` once at startup. (Win32 raw input via `WM_INPUT` also works on XP, but recentering is fewer moving parts and identical in feel at 60 Hz.)

Two rules make it robust. First, only recenter while your window is in the foreground, or alt-tabbing leaves the user fighting a haunted cursor. Second, always measure the delta *from the center you warped to*, never from the last observed position — otherwise you measure your own `SetCursorPos` as movement and the camera drifts on its own.

### Where input meets the fixed timestep

Your skeleton runs updates at a fixed 60 Hz and renders as fast as vsync allows; one render frame may run zero, one, or several update ticks. Split the camera accordingly:

**Mouse, once per render frame.** A mouse delta is already a displacement — "the hand moved 12 counts since last frame" — so apply it to yaw/pitch directly, scaled only by a sensitivity constant. Never multiply it by dt: that would tie look speed to framerate, and the day your fps dips to 30 your aim would halve.

**Movement, in update ticks.** Velocity is per-second, so it must be integrated with dt in `engine_update`. Sample the keys there with `GetAsyncKeyState` and step the position by `speed * dt`. Movement stays on the XZ plane (drop the pitch term from forward), which gives the classic "look at the sky while strafing a circle" feel; hang vertical motion on Space/Ctrl if you want it.

## The code

Orientation, once per render frame, called from the frame loop before the update ticks:

```
/* camera.c */
typedef struct {
    vec3_t pos;
    float  yaw, pitch;      /* radians */
} camera_t;

camera_t g_cam;

#define MOUSE_SENS 0.0022f  /* radians per mouse count */
#define PITCH_MAX  1.553f   /* 89 degrees */

void Camera_MouseFrame(HWND hwnd)
{
    POINT p, c;
    RECT rc;

    if (GetForegroundWindow() != hwnd) return;

    GetClientRect(hwnd, &rc);
    c.x = rc.right / 2;
    c.y = rc.bottom / 2;
    ClientToScreen(hwnd, &c);
    GetCursorPos(&p);

    g_cam.yaw   += (p.x - c.x) *  MOUSE_SENS;
    g_cam.pitch += (p.y - c.y) * -MOUSE_SENS; /* screen y grows down */
    if (g_cam.pitch >  PITCH_MAX) g_cam.pitch =  PITCH_MAX;
    if (g_cam.pitch < -PITCH_MAX) g_cam.pitch = -PITCH_MAX;

    SetCursorPos(c.x, c.y);
}
```

Movement in update ticks, and the view matrix for the render:

```
void Camera_Update(float dt)
{
    vec3_t fwd, right, step;
    float speed;

    speed = 5.0f;                               /* meters per second */
    if (GetAsyncKeyState(VK_SHIFT) & 0x8000) speed = 15.0f;

    fwd[0]   =  (float)sin(g_cam.yaw);          /* flattened to XZ */
    fwd[1]   =  0.0f;
    fwd[2]   = -(float)cos(g_cam.yaw);
    right[0] =  (float)cos(g_cam.yaw);
    right[1] =  0.0f;
    right[2] =  (float)sin(g_cam.yaw);

    if (GetAsyncKeyState('W') & 0x8000) {
        VectorScale(fwd, speed * dt, step);
        VectorAdd(g_cam.pos, step, g_cam.pos);
```

```
    }
    if (GetAsyncKeyState('S') & 0x8000) {
        VectorScale(fwd, -speed * dt, step);
        VectorAdd(g_cam.pos, step, g_cam.pos);
    }
    if (GetAsyncKeyState('D') & 0x8000) {
        VectorScale(right, speed * dt, step);
        VectorAdd(g_cam.pos, step, g_cam.pos);
    }
    if (GetAsyncKeyState('A') & 0x8000) {
        VectorScale(right, -speed * dt, step);
        VectorAdd(g_cam.pos, step, g_cam.pos);
    }
}

void Camera_ViewMatrix(float *out)
{
    vec3_t fwd, center;
    static const vec3_t up = { 0.0f, 1.0f, 0.0f };

    fwd[0] =  (float)(cos(g_cam.pitch) * sin(g_cam.yaw));
    fwd[1] =  (float)sin(g_cam.pitch);
    fwd[2] = -(float)(cos(g_cam.pitch) * cos(g_cam.yaw));
    VectorAdd(g_cam.pos, fwd, center);
    mat4_lookat(out, g_cam.pos, center, up);
}
```

Wire-up left to you: call `Camera_MouseFrame` once per loop iteration, `Camera_Update` from `engine_update`, and replace the fixed `mat4_lookat` in `Cube_Draw` with `Camera_ViewMatrix`. Start the camera at `(0, 1, 3)` so the cube is in view at boot.

## Pitfalls

**Camera drifts by itself.** You are measuring the warp you caused. Delta must be `cursor - center`, computed after the previous frame's `SetCursorPos` put it at center. If you mix in `WM_MOUSEMOVE` messages you will count every motion twice — poll, and ignore the message queue for looking.

**Look inverted.** Screen Y grows downward, world pitch grows upward. That is the negation in the listing; delete it and you have flight-sim controls, which is a setting, not a bug, but decide on purpose.

**Black screen at the poles.** You skipped the pitch clamp and lookat divided by zero. Clamp at 89°, not 90°.

**Speed depends on framerate.** If movement is in the render path or mouse look is dt-scaled, the camera behaves differently at 60 and 30 fps. On this netbook you *will* see 30 sometimes (vsync quantizes misses to 60/30/20/15), so the bug is not hypothetical.

**Cursor oddities.** `ShowCursor` keeps a counter, not a flag — call it once, not per frame. And skip recentering when another window is foreground, or your build becomes the most hated process on the machine.

> **DONE WHEN:** you can orbit and strafe around the spinning cube smoothly, look straight up and down without incident, and alt-tab away without the cursor being held hostage.

**IF YOU ARE STUCK:**

- Print yaw/pitch (temporarily, in the title bar) and move the mouse slowly right: yaw should increase steadily. If it jitters around zero, your center point is wrong — remember `ClientToScreen`; cursor coordinates are screen-space.
- Diagonal movement is √2 faster than straight — period authentic. Normalize the summed wish direction if it bothers you.
- If motion stutters while looking is smooth, you are moving in render frames and looking in ticks — you wanted the reverse.
- W walks backward? Your forward formula has the wrong sign on z; at yaw 0 forward must be (0, 0, -1).

# Day 4: The TGA Loader

## What you are building, and why 1999 cared

A TGA image loader written by hand — uncompressed and RLE, 24 and 32 bit — feeding `glTexImage2D`, with mipmaps, and a hand-drawn Paint Shop Pro texture on every face of the cube. TGA was *the* texture interchange format of the era: trivial to write, supports alpha, and every tool exported it (Quake 3 shipped its high-quality assets as TGA). It is also the perfect first binary file format: 18 bytes of header, then pixels. Tomorrow a library takes over this job; today you earn the right to use it by understanding exactly what it does.

## Theory: the format

All multi-byte fields are little-endian, which is native on x86: read bytes, never `fread` into a struct (VC6 will pad it unless you `#pragma pack`, and byte assembly is safer than remembering that). The 18-byte header:

| Offset | Size | Field | Meaning |
|---|---|---|---|
| 0 | 1 | id_length | bytes of free-form ID after the header — skip this many |
| 1 | 1 | colormap_type | 0 = no palette; reject anything else |
| 2 | 1 | image_type | 2 = uncompressed truecolor, 10 = RLE truecolor; reject others |
| 3 | 2 | cmap_first | palette fields; zero for type 2/10 |
| 5 | 2 | cmap_length | ditto |
| 7 | 1 | cmap_entry_size | ditto |
| 8 | 2 | x_origin | ignore |
| 10 | 2 | y_origin | ignore |
| 12 | 2 | width | pixels |
| 14 | 2 | height | pixels |
| 16 | 1 | bpp | 24 (BGR) or 32 (BGRA) |
| 17 | 1 | descriptor | bits 0-3: alpha bits; **bit 5: 0 = bottom-up rows, 1 = top-down**; bit 4 (right-to-left) is never set by real tools |

Two traps live in that table. Pixels are **BGR(A)**, not RGB — swapping red and blue is the rite of passage the challenge sheet promised you. And the row order is variable: by default TGA stores the *bottom* row first, which happens to be exactly what GL wants (texture row 0 is v = 0, the bottom of the image), but PSP7 and others set descriptor bit 5 and store top-down, so you must handle both.

**RLE (type 10)** is a stream of packets. Each starts with one header byte: if the high bit is set, it is a run — read *one* pixel and repeat it `(byte & 0x7F) + 1` times; if clear, it is raw — that many literal pixels follow. Writers are not supposed to let packets cross scanlines, but some do, so decode against the total pixel count, not per-row.

### From pixels to texture

Decode everything to RGBA, even 24-bit files: one upload path, and 4-byte pixels sidestep GL's row-alignment rules (GL assumes rows start on 4-byte boundaries; RGB rows of odd widths violate that unless you call `glPixelStorei(GL_UNPACK_ALIGNMENT, 1)`).

Upload with `glTexImage2D`, then call `glGenerateMipmap` and filter with `GL_LINEAR_MIPMAP_LINEAR` (trilinear). Mipmaps are not a luxury on this GPU: minified textures without mips shimmer like a 1996 software renderer, and mips are a bandwidth *win* — distant texels come from tiny mip levels that fit in cache. Wrap mode: `GL_REPEAT` tiles the texture (terrain, walls); `GL_CLAMP_TO_EDGE` stretches the border row outward (UI, skyboxes — day 10 depends on it).

> **WARNING:** The hard NPOT rule of GLES2: a texture whose width or height is not a power of two may only use `GL_CLAMP_TO_EDGE` wrapping and may not have mipmaps (`glGenerateMipmap` fails with `GL_INVALID_OPERATION`). Break the rule and the texture is "incomplete": every sample silently returns black. Author everything at 64/128/256/512 and the rule never bites.

## The code

```
/* tga.c - returns malloc'd RGBA, bottom row first; free() it */
unsigned char *TGA_Load(const char *path, int *out_w, int *out_h)
{
    FILE *fp;
    unsigned char hdr[18];
    unsigned char *pix;
    int w, h, bpp, bytespp, type, topdown;
    int i, count;

    fp = fopen(path, "rb");
    if (fp == NULL) return NULL;
    if (fread(hdr, 1, 18, fp) != 18) { fclose(fp); return NULL; }

    type    = hdr[2];
    w       = hdr[12] | (hdr[13] << 8);
    h       = hdr[14] | (hdr[15] << 8);
    bpp     = hdr[16];
    topdown = (hdr[17] & 0x20) != 0;
    bytespp = bpp / 8;

    if (hdr[1] != 0 || (type != 2 && type != 10) ||
        (bpp != 24 && bpp != 32) || w <= 0 || h <= 0) {
        fclose(fp);
        return NULL;
    }
    fseek(fp, hdr[0], SEEK_CUR);         /* skip the image ID field */

    pix = (unsigned char *)malloc(w * h * 4);
    if (pix == NULL) { fclose(fp); return NULL; }

    count = w * h;
    i = 0;

/* place file-order pixel i into the output, bottom row first,
   swapping BGR(A) to RGBA as it lands */
#define EMIT(src) do { \
        int fx = i % w, fy = i / w; \
        int dy = topdown ? (h - 1 - fy) : fy; \
        unsigned char *d = pix + (dy * w + fx) * 4; \
        d[0] = (src)[2]; d[1] = (src)[1]; d[2] = (src)[0]; \
        d[3] = (bytespp == 4) ? (src)[3] : 255; \
        ++i; \
    } while (0)

    if (type == 2) {                     /* uncompressed */
        unsigned char px[4];
        while (i < count) {
            fread(px, 1, bytespp, fp);
```

```
            EMIT(px);
        }
    } else {                            /* type 10: RLE */
        unsigned char px[4];
        while (i < count) {
            int pk, n;
            pk = fgetc(fp);
            if (pk == EOF) break;
            n = (pk & 0x7F) + 1;
            if (pk & 0x80) {            /* run: one pixel, n copies */
                fread(px, 1, bytespp, fp);
                while (n-- > 0 && i < count) EMIT(px);
            } else {                    /* raw: n literal pixels */
                while (n-- > 0 && i < count) {
                    fread(px, 1, bytespp, fp);
                    EMIT(px);
                }
            }
        }
    }
#undef EMIT

    fclose(fp);
    *out_w = w;
    *out_h = h;
    return pix;
}
```

The uploader — you will reuse this verbatim tomorrow:

```
GLuint Texture_FromPixels(const unsigned char *rgba, int w, int h)
{
    GLuint id;
    glGenTextures(1, &id);
    glBindTexture(GL_TEXTURE_2D, id);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0,
                 GL_RGBA, GL_UNSIGNED_BYTE, rgba);
    glGenerateMipmap(GL_TEXTURE_2D);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR_MIPMAP_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    return id;
}
```

Left to you, and it is the bulk of the day: the cube needs texture coordinates. Eight shared corners cannot carry per-face UVs — the corner where three faces meet needs three different UV pairs — so rebuild the cube as **24 vertices** (4 per face), layout pos3 + uv2, stride 20, offsets 0 and 12; the 36 indices now reference face-local quads. Add `attribute vec2 a_uv` and a varying to the vertex shader; the fragment shader becomes `gl_FragColor = texture2D(u_tex, v_uv);` with a `uniform sampler2D`. Draw something asymmetric with text in PSP7 (256×256, 24-bit TGA, and try an RLE export too) so orientation bugs cannot hide.

## Pitfalls

**Red and blue swapped.** Everyone ships this bug once. The swap in `EMIT` handles it; if your texture still looks like a photo negative of itself, you swapped twice.

**Diagonal shredding.** Skewed, marching garbage means row length is wrong: you forgot to skip `id_length` bytes, or treated a 32-bit file as 24.

**Upside-down texture.** Descriptor bit 5. PSP7 writes top-down files; a loader tested only on bottom-up files flips them. The `dy` computation in `EMIT` is the fix — test with one file of each kind.

**Black cube.** An incomplete texture: NPOT with mips, or you created the texture before the GL context existed, or forgot `glBindTexture` before drawing. GLES2 samples incomplete textures as black, with no error at draw time — check `glGetError` right after `glGenerateMipmap`.

**Performance and memory.** Per-pixel `fread` is lazy — fine at load time for 256², sluggish for a screen-size image; read the whole file into a buffer first if loads ever feel slow. Memory math for the journal: 512×512 RGBA is 1 MB, plus a third for mips. VRAM here is borrowed system RAM and you have 1 GB total; keep the texture budget in the tens of megabytes.

**DONE WHEN:** your own hand-drawn texture is on every face of the cube, right way up, red where you painted red — from both an uncompressed and an RLE TGA.

**IF YOU ARE STUCK:**

- Dump the header: `printf` type, w, h, bpp, descriptor as hex. Most failures are visible right there.
- Feed the loader a 2×2 test image you construct in a hex editor (18-byte header + 4 known pixels). When the four RGBA values land in the right slots, big files follow.
- Cube renders but grey/black: query `glGetAttribLocation` for `a_uv` — if it is −1 the sampler was optimized out; check the fragment shader actually uses `v_uv`.
- RLE file shorter than expected? A run packet's pixel is stored *once*, not n times — if you fread inside the run loop you will desync and the bottom of the image becomes noise.

# Day 5: stb_image and the Texture Cache

## What you are building, and why 1999 cared

Two things: `stb_image.h` wired in behind your own `Texture_Load(path)` so every format is one call, and a texture cache so loading the same path twice returns the same GL texture instead of a duplicate megabyte. The 1999 justification for the cache is direct: with 8 MB of VRAM, an engine that uploaded `wall.tga` once per wall was dead on arrival. Quake called this structure the "registration" pass. Yours is smaller but the same idea, and on day 20 it grows into the full resource manager.

Why a library now, after yesterday's hand-rolled loader? Because you have earned it. A TGA loader is an afternoon; a correct PNG inflate plus a baseline JPEG decoder is a month, and it is not engine work. stb_image is a single header, public domain, pure C89-friendly C — the exact kind of dependency this project allows.

## The rules of stb_image on VC6

stb_image is a "single-header library": the header contains both the declarations and, behind an `#ifdef`, the entire implementation. Exactly **one** translation unit in your project may define `STBI_IMPLEMENTATION` before including it. Zero TUs: every call is an unresolved external (`LNK2001`). Two TUs: duplicate symbols (`LNK2005`). The clean pattern is a dedicated C file that contains nothing else:

```
/* stb_impl.c - the ONLY file that defines STBI_IMPLEMENTATION.
   Keep stb_image out of every other translation unit's compile. */
#define STBI_ONLY_PNG
#define STBI_ONLY_JPEG
#define STBI_ONLY_TGA
#define STBI_ONLY_BMP
#define STBI_IMPLEMENTATION
#include "stb_image.h"
```

On VC6 the dedicated TU is not just tidiness. stb_image is a large lump of C, and this compiler is from 1998: isolating it keeps your rebuilds fast (it compiles once and never again), and if the old optimizer ever miscompiles it — not unheard of with /O2 on code this dense — you can set that one file to /Od in Project Settings without slowing the engine down. The `STBI_ONLY_*` defines strip the decoders you will never ship.

Call `stbi_load(path, &w, &h, &n, 4)`: the final 4 **forces RGBA** regardless of what the file contains, which means one upload path, no alignment traps, and yesterday's `Texture_FromPixels` works unchanged. The returned buffer comes from stb's allocator, so it must go back through `stbi_image_free`, never `free` — keep the pairing honest even though they are the same CRT heap today. One orientation note: stb_image returns the *top* row first, the opposite of your TGA loader. Call `stbi_set_flip_vertically_on_load(1)` once at startup and both loaders agree that row 0 is the bottom.

## The cache: an array and strcmp

Resist the urge to build a hash table. 128 slots, linear scan, `strcmp` per slot: a full miss costs a few thousand cycles, you load textures a handful of times per level, and the whole structure is debuggable at a glance in the VC6 watch window. The refcount field is planted now so day 20's resource manager has roots.

```
/* texture.c */
#define MAX_TEXTURES 128
```

```
typedef struct {
    char   path[64];
    GLuint id;
    int    refs;
} tex_slot_t;

static tex_slot_t g_tex[MAX_TEXTURES];
int g_tex_loads, g_tex_hits;        /* debug counters, overlay-bound */

GLuint Texture_Load(const char *path)
{
    int i, w, h, n;
    unsigned char *pix;
    tex_slot_t *slot = NULL;

    for (i = 0; i < MAX_TEXTURES; ++i) {
        if (g_tex[i].refs > 0 && strcmp(g_tex[i].path, path) == 0) {
            ++g_tex[i].refs;
            ++g_tex_hits;
            return g_tex[i].id;
        }
    }
    for (i = 0; i < MAX_TEXTURES; ++i) {
        if (g_tex[i].refs == 0) { slot = &g_tex[i]; break; }
    }
    if (slot == NULL || strlen(path) >= sizeof(slot->path)) {
        MessageBoxA(NULL, path, "Texture_Load: no slot / path too long",
                    MB_OK);
        return 0;
    }

    pix = stbi_load(path, &w, &h, &n, 4);   /* force RGBA */
    if (pix == NULL) {
        MessageBoxA(NULL, path, "Texture_Load: cannot load", MB_OK);
        return 0;
    }
    slot->id = Texture_FromPixels(pix, w, h);
    stbi_image_free(pix);
    strcpy(slot->path, path);
    slot->refs = 1;
    ++g_tex_loads;
    return slot->id;
}

void Texture_Release(GLuint id)
{
    int i;
    for (i = 0; i < MAX_TEXTURES; ++i) {
        if (g_tex[i].refs > 0 && g_tex[i].id == id) {
            if (--g_tex[i].refs == 0)
                glDeleteTextures(1, &g_tex[i].id);
            return;
        }
    }
}
```

Retire yesterday's direct TGA path by routing it through `Texture_Load` too — keep your own TGA loader in the tree (it becomes the reference when an stb-loaded image looks wrong), but production loads go through one door. Then prove the DONE line: call `Texture_Load("data/photo.jpg")` twice, and show `loads=1 hits=1`. Until tomorrow's overlay exists, `_snprintf` the counters into the title bar — its final day of employment, so let it go out with useful numbers.

## Pitfalls

**Linker errors.** `LNK2001 unresolved _stbi_load`: no TU defines `STBI_IMPLEMENTATION` (or `stb_impl.c` never got added to the project). `LNK2005 already defined`: two TUs define it. There is no third state.

**Cache misses that should hit.** `strcmp` is case-sensitive and the Windows filesystem is not, so `"DATA/Wall.png"` and `"data/wall.png"` are the same file and two cache entries. Same for `/` versus `\`. Normalize in `Texture_Load` before comparing: `_strlwr` the copied path and fold slashes one way.

**Leaks.** Every early-return path after a successful `stbi_load` must free the pixels; the listing frees immediately after upload, which is the easy shape to keep correct. The GL side leaks too if you forget `Texture_Release` — day 20 will count 100 load/unload cycles and shame you.

**JPEG cost on the Atom.** Decoding a big JPEG takes real time on two in-order cores — a 1024×1024 costs tens of milliseconds. Load at startup, never mid-frame. And the NPOT rule from day 4 still stands: a 1000×750 photo will refuse mipmaps. Resample photos to 512×512 in PSP7 before shipping them.

**DONE WHEN:** the cube wears a JPG photo, and the counters prove the cache: loading the same path twice shows `loads=1 hits=1`, and the second call returned the same GLuint.

**IF YOU ARE STUCK:**

- stb_image failure reason: `stbi_failure_reason()` returns a string — put it in the MessageBox.
- Compile of `stb_impl.c` fails on VC6: make sure the file compiles as C, not C++ (no /TP), and that you did not also define `STBI_NO_STDIO` while following someone's advice.
- Photo is sideways or mirrored: that is your UV layout, not the loader — the flip call only swaps top and bottom.
- Second load returns a different GLuint: print both paths inside the compare loop; you will usually see the case or slash mismatch immediately.

# Day 6: Blending, the Overlay, and a Bitmap Font

## What you are building, and why 1999 cared

Three interlocking pieces: alpha blending, a second render pass with an orthographic projection, and a bitmap font drawn from a 16×16 glyph grid — ending with your fps drawn *inside* the frame. Every engine of the era had exactly this stack: Quake's console text, its crosshair, its loading plaque were all textured quads in an ortho pass over the 3D scene. The title bar fps counter retires today. From now on, if a number matters, the engine itself tells you — and days 9, 12, and 27 will pour their draw counts into this overlay.

## Theory: the orthographic matrix

The perspective matrix spent its cleverness on the divide by `-z`. The orthographic matrix has no cleverness: it is an affine map squeezing the box `[l,r] × [b,t] × [n,f]` into the ±1 NDC cube, leaving `w = 1`. For x: `x_ndc = 2(x-l)/(r-l) - 1`, which unpacks into a scale and an offset; y and z are the same story.

```
void mat4_ortho(float *out, float l, float r, float b, float t,
                float zn, float zf)
{
    int i;
    for (i = 0; i < 16; ++i) out[i] = 0.0f;
    out[0]  =  2.0f / (r - l);
    out[5]  =  2.0f / (t - b);
    out[10] = -2.0f / (zf - zn);
    out[12] = -(r + l) / (r - l);
    out[13] = -(t + b) / (t - b);
    out[14] = -(zf + zn) / (zf - zn);
    out[15] = 1.0f;
}
```

The trick that makes UI code pleasant: call it as `mat4_ortho(m, 0, 1024, 600, 0, -1, 1)` — bottom = 600, top = 0. That flips the y axis so the origin is the top-left corner and y grows downward, and you lay out text in pixel coordinates like it is 1989 and you are writing to VGA memory.

## Theory: blending and the order of battle

Blending replaces "write the fragment" with a mix: `glEnable(GL_BLEND)` plus `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)` computes

```
  final = src.a * src.rgb + (1 - src.a) * dst.rgb
```

— standard "over" compositing. Note `dst` in that equation: blending reads what is already in the framebuffer, which means **order matters**. The rule you will follow for the rest of the book:

> **NOTE:** Draw order, forever: (1) opaque geometry first, depth test and depth writes on; (2) transparent world geometry back to front, depth *test* on but depth *writes* off (day 18's particles); (3) UI overlay last, depth test off entirely, blending on. Deviate and you get transparent things punching invisible holes in the scene behind them.

Blending is also not free on this GPU — every blended pixel is a read-modify-write against shared-memory bandwidth — so enable it for the UI pass and disable it when done.

## Theory: a font is a texture atlas with delusions of grammar

One 256×256 texture, 256 glyphs in a 16×16 grid, each cell 16×16 pixels, arranged in ASCII order. Character `c` lives at grid column `c % 16`, row `c / 16` — the glyph's position is computed from its char code, no lookup table. Each cell is 1/16 of UV space, and since your loaders deliver row 0 at the *bottom* while the grid is painted top-down, the v axis inverts:

```
col = c % 16;   u0 = col / 16.0;         u1 = u0 + 1/16.0
row = c / 16;   v_top = 1.0 - row/16.0;  v_bot = v_top - 1/16.0
```

Paint the font in PSP7: white glyphs, transparent background, export as 32-bit TGA (or PNG) so it has an alpha channel. White matters — the shader multiplies the texture by a color, so a white font tints to anything. Keep one pixel of margin inside each cell; at 1:1 scale it is invisible, and the day you scale text, `GL_LINEAR` stops bleeding neighbor glyphs into the edges. Create this texture *without* mipmaps (`GL_LINEAR` min and mag): text is drawn 1:1, and mips would only blur it.

Rendering is one quad per character, all accumulated into a single dynamic vertex buffer and drawn in one call. `Font_Print` appends quads to a CPU array; `Font_Flush` uploads with `GL_DYNAMIC_DRAW` and draws. Re-specifying the whole buffer with `glBufferData` each frame is deliberate: ANGLE turns it into a D3D9 dynamic buffer DISCARD, the fast path for exactly this pattern.

## The code

```
/* font.c - 16x16 grid font, one quad per char, one draw per frame */
#define FONT_MAX   1024        /* chars per frame */
#define GLYPH_SIZE 16.0f       /* on-screen pixels per char */

typedef struct { float x, y, u, v; } fontvert_t;

static fontvert_t g_fverts[FONT_MAX * 6];
static int    g_fcount;
static GLuint g_fvbo, g_ftex, g_fprog;
static GLint  g_fumvp, g_fucolor, g_fapos, g_fauv;

void Font_Print(float x, float y, const char *text)
{
    const char *p;
    for (p = text; *p; ++p) {
        int   c  = (unsigned char)*p;
        float u0 = (c % 16) * (1.0f / 16.0f);
        float u1 = u0 + 1.0f / 16.0f;
        float v1 = 1.0f - (c / 16) * (1.0f / 16.0f);  /* cell top   */
        float v0 = v1 - 1.0f / 16.0f;                 /* cell bottom */
        fontvert_t *q;

        if (g_fcount >= FONT_MAX) return;
        q = &g_fverts[g_fcount * 6];
        /* y grows downward; two CCW triangles per glyph */
        q[0].x = x;              q[0].y = y;              /* top-left  */
        q[0].u = u0;             q[0].v = v1;
        q[1].x = x;              q[1].y = y + GLYPH_SIZE; /* bot-left  */
        q[1].u = u0;             q[1].v = v0;
        q[2].x = x + GLYPH_SIZE; q[2].y = y + GLYPH_SIZE; /* bot-right */
        q[2].u = u1;             q[2].v = v0;
        q[3] = q[0];
        q[4] = q[2];
        q[5].x = x + GLYPH_SIZE; q[5].y = y;              /* top-right */
        q[5].u = u1;             q[5].v = v1;
```

```
        ++g_fcount;
        x += GLYPH_SIZE;
    }
}

void Font_Flush(void)
{
    float ortho[16];
    if (g_fcount == 0) return;

    mat4_ortho(ortho, 0.0f, 1024.0f, 600.0f, 0.0f, -1.0f, 1.0f);

    glDisable(GL_DEPTH_TEST);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    glUseProgram(g_fprog);
    glUniformMatrix4fv(g_fumvp, 1, GL_FALSE, ortho);
    glUniform4f(g_fucolor, 1.0f, 1.0f, 1.0f, 1.0f);  /* tint: white */
    glBindTexture(GL_TEXTURE_2D, g_ftex);

    glBindBuffer(GL_ARRAY_BUFFER, g_fvbo);
    glBufferData(GL_ARRAY_BUFFER,
                 g_fcount * 6 * (int)sizeof(fontvert_t),
                 g_fverts, GL_DYNAMIC_DRAW);
    glEnableVertexAttribArray((GLuint)g_fapos);
    glEnableVertexAttribArray((GLuint)g_fauv);
    glVertexAttribPointer((GLuint)g_fapos, 2, GL_FLOAT, GL_FALSE,
                          16, BUFFER_OFFSET(0));
    glVertexAttribPointer((GLuint)g_fauv,  2, GL_FLOAT, GL_FALSE,
                          16, BUFFER_OFFSET(8));
    glDrawArrays(GL_TRIANGLES, 0, g_fcount * 6);

    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
    g_fcount = 0;
}
```

Left to you: `Font_Init` (load the font texture through `Texture_Load` but with the no-mipmap filter, create the VBO, compile the font shaders and query locations). The shaders are day 2's with two-component position, a `v_uv` varying, and `gl_FragColor = texture2D(u_tex, v_uv) * u_color;`. The `u_color` uniform tints a whole string; when you later want per-character color (day 19's console will), extend `fontvert_t` with r,g,b,a and switch the tint to a vertex attribute — the layout grows from 16 to 32 bytes and nothing else changes shape.

The fps counter itself: count rendered frames, and once per second (compare against your QueryPerformanceCounter clock) format the total into a string. Remember the VC6 spelling and its sharp edge: `_snprintf` does *not* guarantee a terminator when the output fills the buffer, so write `buf[sizeof(buf)-1] = 0;` after it, every time. Then, at the very end of `engine_render`: `Font_Print(8, 8, buf); Font_Flush();`.

## Pitfalls

**No text at all.** Depth test still on (the scene is at those pixels and the ortho quads have z = 0 — whether they survive is luck); or the ortho call has t and b in the wrong order so the text is off-screen; or the font shader's attribute locations were queried from the cube's program.

**White boxes around glyphs.** Blending is off, or the font texture has no alpha channel — a 24-bit export silently fills alpha with 255. Re-export as 32-bit and verify in PSP7 that the background is actually transparent, not white.

**Dark fringes around letters.** The classic: transparent texels have black RGB, and `GL_LINEAR` averages that black into the glyph edge before alpha discards it. Fix in the asset: flood the RGB of transparent regions with white so filtering blends white-into-white.

**Text mirrored or upside down.** The v inversion. If your grid image's top row of glyphs (codes 0-15) renders where codes 240-255 should be, you dropped the `1.0 -` in the v formula.

**The world went translucent.** You left `GL_BLEND` on after the UI pass, and tomorrow's first draw blends the sky into last frame. `Font_Flush` restores state on the way out; keep that habit for every pass you write from now on.

Performance: a full overlay of 1024 characters is 2048 triangles and one buffer upload of 96 KB per frame — the Atom will not notice. The thing to watch is blended *area*, not glyph count; a full-screen 1024×600 blended quad is where this GPU starts coughing (day 18 will find that limit on purpose).

**DONE WHEN:** the fps and the day-5 cache counters render in the corner over the spinning textured cube, glyph edges are clean against any background, and the title bar shows nothing but the window's name.

**IF YOU ARE STUCK:**

- Draw one giant glyph first: `Font_Print(100, 100, "A")` with `GLYPH_SIZE` temporarily 128. Position bugs, UV bugs, and blend bugs are all legible at that size.
- Disable blending deliberately: you should see the glyph in a solid box. If you do not, the problem is geometry/UV, not blending — solve those in that order.
- Ortho sanity: with the flipped call, (0,0) is top-left and (1024,600) is bottom-right. Print a corner marker at each to confirm.
- Tint test: set `u_color` to red. If text turns red, the pipeline is fine and any remaining ugliness lives in the font texture.

# Day 7: The OBJ Loader

Until this morning, every vertex in your engine was typed by hand. That ends today. Wavefront OBJ — a plain-text format from the Advanced Visualizer package of the late 1980s — is the format every modeler of your era exports, including your copy of Milkshape 3D. It is readable in Notepad, writable by a shell script, and it survived because of exactly that. In 1999, writing an OBJ importer was the rite of passage between "renders a cube" and "renders art."

The format itself is trivial. The real lesson of the day is the impedance mismatch between art formats and GPU formats: OBJ indexes positions, texture coordinates, and normals *independently*, and `glDrawElements` allows exactly *one* index stream. Bridging that gap — the dedup of index triples — is an algorithm you will reuse in every importer you ever write.

## The grammar

An OBJ file is a sequence of lines. Each line starts with a keyword tag. A Milkshape export looks like this:

```
# MilkShape 3D Wavefront OBJ Export
mtllib crate.mtl
v -0.500000 -0.500000 0.500000
v 0.500000 -0.500000 0.500000
vt 0.000000 1.000000
vn 0.000000 0.000000 1.000000
g Box01
usemtl Material01
s 1
f 1/1/1 2/2/2 3/3/3
```

| Tag | Meaning | Fields |
|---|---|---|
| `v` | position | 3 floats (x y z) |
| `vt` | texture coordinate | 2 floats (u v), origin bottom-left like GL |
| `vn` | normal | 3 floats, not guaranteed unit length |
| `f` | face | 3 or more corner references |
| `#`, `g`, `o`, `s`, `usemtl`, `mtllib` | comment, groups, smoothing, materials | ignore all of them today |

Each corner of an `f` line references the `v`, `vt`, and `vn` lists by number, and it comes in four spellings:

```
f 1 2 3          position only
f 1/4 2/5 3/6    position/texcoord
f 1//7 2//8 3//9 position//normal   (no texcoord: note the //)
f 1/4/7 2/5/8    position/texcoord/normal (the full triple)
```

**WARNING:** OBJ indices are **1-based**. The first `v` line is index 1, not 0. Forget to subtract 1 and your model renders as an exploded porcupine of triangles reaching for the origin. Everyone does this exactly once. The full spec also allows *negative* indices (relative to the end of the list); Milkshape never writes them, so treat a corner index < 1 after parsing as a file error and move on with your life.

## Fan triangulation

A face may have more than three corners. The GPU wants triangles. For convex polygons (which is all a sane modeler exports) the fix is a fan around corner 0:

```
   c3-------c2         f  c0 c1 c2 c3
    |      / |         =  (c0,c1,c2) + (c0,c2,c3)
    |    /   |
    |  /     |         n corners -> n-2 triangles,
    |/       |         every triangle shares corner c0
   c0-------c1
```

OBJ faces are wound counter-clockwise, the same as GL's default front face, and the fan preserves the winding. No flipping needed.

## The dedup: three index streams into one

Here is the mismatch, concretely. A textured cube has 8 positions. But each corner of the cube belongs to three faces with three different normals and three different UVs, so the GPU needs **24** vertices — each a unique *(position, uv, normal)* combination. `glDrawElements` takes one index per vertex, and that index selects position, uv, and normal *together* from one interleaved buffer.

So the unified vertex is defined by the triple `(v,vt,vn)` from the face line. The algorithm: for each face corner, look up whether this exact triple has been seen before. If yes, reuse its index. If no, append a new interleaved vertex built from the three source arrays and remember the triple. The unified vertex count lands somewhere between the position count and 3×(triangle count).

## Two passes: counting before allocating

You have 1 GB of RAM and Windows XP wants a third of it. Do not guess-and-realloc. Read the file twice: pass one counts `v`, `vt`, `vn` lines and triangles (corners minus 2 per face), then you `malloc` exact-size arrays, rewind, and pass two fills them. The file is in the OS disk cache after pass one, so the second read is nearly free.

One parsing subtlety before the listing: `fscanf` with `%s` reads whitespace-delimited tokens and treats newlines as just more whitespace, which is perfect for the fixed-arity lines (`v`, `vt`, `vn`) but useless for `f`, whose corner count varies and ends at the newline that `fscanf` cannot see. So: dispatch on a tag read with `fscanf`, parse fixed lines with `fscanf`, and for `f` grab the rest of the line with `fgets` and tokenize it.

```
/* ---- listing 7.1: mesh types and pass one ------------------- */

typedef struct {
    float pos[3];
    float uv[2];
    float nrm[3];
} vertex_t;                     /* 32 bytes, interleaved */

typedef struct { int v, t, n; } triple_t;

typedef struct {
    vertex_t       *verts;
    unsigned short *indices;
    int num_verts;
    int num_indices;
    GLuint vbo, ibo;
    vec3_t center;              /* bounds: filled on day 9 */
    float  radius;
```

```
} mesh_t;

static void skip_line(FILE *fp)
{
    int ch;
    while ((ch = fgetc(fp)) != '\n' && ch != EOF)
        ;
}

static int obj_count(FILE *fp, int *nv, int *nvt, int *nvn,
                     int *ntris)
{
    char tag[16];
    char line[1024];
    *nv = *nvt = *nvn = *ntris = 0;
    while (fscanf(fp, "%15s", tag) == 1) {
        if (strcmp(tag, "f") == 0) {
            int corners = 0;
            char *tok;
            if (!fgets(line, sizeof line, fp))
                break;
            tok = strtok(line, " \t\r\n");
            while (tok) {
                corners++;
                tok = strtok(NULL, " \t\r\n");
            }
            if (corners >= 3)
                *ntris += corners - 2;
        } else {
            if (strcmp(tag, "v") == 0)       (*nv)++;
            else if (strcmp(tag, "vt") == 0) (*nvt)++;
            else if (strcmp(tag, "vn") == 0) (*nvn)++;
            skip_line(fp);
        }
    }
    return *nv > 0 && *ntris > 0;
}
```

Pass two is the meat: parse the four corner spellings, dedup, fan. The corner parser leans on an `sscanf` property: the order of the attempts matters, because `"%d/%d/%d"` against `"1//7"` stops after the first number and returns 1, so you try the most specific pattern first.

```
/* ---- listing 7.2: pass two, dedup, triangulate --------------- */

static int parse_corner(const char *tok, int *vi, int *ti, int *ni)
{
    if (sscanf(tok, "%d/%d/%d", vi, ti, ni) == 3) return 1;
    if (sscanf(tok, "%d//%d",   vi, ni) == 2) { *ti = 0; return 1; }
    if (sscanf(tok, "%d/%d",    vi, ti) == 2) { *ni = 0; return 1; }
    if (sscanf(tok, "%d",       vi) == 1) { *ti = *ni = 0; return 1; }
    return 0;
}

/* linear search over the triples seen so far; returns the unified
   index for (vi,ti,ni), appending a new vertex when it is new.
   vi/ti/ni are 0-based here; -1 means "not present in the file". */
static int find_or_add(triple_t *trip, vertex_t *verts, int *count,
                       int vi, int ti, int ni, const float *P,
                       const float *T, const float *N)
{
    int i;
    for (i = 0; i < *count; ++i)
        if (trip[i].v == vi && trip[i].t == ti &&
            trip[i].n == ni)
```

```
            return i;
    i = (*count)++;
    trip[i].v = vi;  trip[i].t = ti;  trip[i].n = ni;
    verts[i].pos[0] = P[vi*3+0];
    verts[i].pos[1] = P[vi*3+1];
    verts[i].pos[2] = P[vi*3+2];
    if (ti >= 0) {
        verts[i].uv[0] = T[ti*2+0];
        verts[i].uv[1] = T[ti*2+1];
    } else {
        verts[i].uv[0] = verts[i].uv[1] = 0.0f;
    }
    if (ni >= 0) {
        verts[i].nrm[0] = N[ni*3+0];
        verts[i].nrm[1] = N[ni*3+1];
        verts[i].nrm[2] = N[ni*3+2];
    } else {
        verts[i].nrm[0] = verts[i].nrm[2] = 0.0f;
        verts[i].nrm[1] = 1.0f;
    }
    return i;
}

int mesh_load_obj(mesh_t *m, const char *path)
{
    FILE *fp;
    int nv, nvt, nvn, ntris, maxverts;
    int iv, it, in, k;
    float *P, *T, *N;
    triple_t *trip;
    char tag[16], line[1024];

    fp = fopen(path, "r");           /* OBJ is text, not "rb" */
    if (!fp) {
        MessageBoxA(0, path, "obj: open failed", 0);
        return 0;
    }
    if (!obj_count(fp, &nv, &nvt, &nvn, &ntris)) {
        fclose(fp);
        return 0;
    }
    rewind(fp);

    P = (float *)malloc(sizeof(float) * 3 * nv);
    T = (float *)malloc(sizeof(float) * 2 * (nvt ? nvt : 1));
    N = (float *)malloc(sizeof(float) * 3 * (nvn ? nvn : 1));
    maxverts   = ntris * 3;          /* dedup worst case */
    m->verts   = (vertex_t *)malloc(sizeof(vertex_t) * maxverts);
    trip       = (triple_t *)malloc(sizeof(triple_t) * maxverts);
    m->indices = (unsigned short *)
                 malloc(sizeof(unsigned short) * ntris * 3);
    m->num_verts = m->num_indices = 0;
    iv = it = in = 0;

    while (fscanf(fp, "%15s", tag) == 1) {
        if (strcmp(tag, "v") == 0) {
            fscanf(fp, "%f %f %f",
                   &P[iv*3+0], &P[iv*3+1], &P[iv*3+2]);
            iv++; skip_line(fp);
        } else if (strcmp(tag, "vt") == 0) {
            fscanf(fp, "%f %f", &T[it*2+0], &T[it*2+1]);
            it++; skip_line(fp);
        } else if (strcmp(tag, "vn") == 0) {
            fscanf(fp, "%f %f %f",
                   &N[in*3+0], &N[in*3+1], &N[in*3+2]);
            in++; skip_line(fp);
```

```
        } else if (strcmp(tag, "f") == 0) {
            int corner[64];
            int nc = 0;
            char *tok;
            if (!fgets(line, sizeof line, fp))
                break;
            tok = strtok(line, " \t\r\n");
            while (tok && nc < 64) {
                int vi, ti, ni;
                if (parse_corner(tok, &vi, &ti, &ni))
                    corner[nc++] = find_or_add(trip, m->verts,
                        &m->num_verts, vi - 1, ti - 1, ni - 1,
                        P, T, N);
                tok = strtok(NULL, " \t\r\n");
            }
            for (k = 2; k < nc; ++k) {
                m->indices[m->num_indices++] =
                    (unsigned short)corner[0];
                m->indices[m->num_indices++] =
                    (unsigned short)corner[k - 1];
                m->indices[m->num_indices++] =
                    (unsigned short)corner[k];
            }
        } else {
            skip_line(fp);
        }
    }
    fclose(fp);
    free(P); free(T); free(N); free(trip);

    if (m->num_verts > 65535) {
        MessageBoxA(0, path, "obj: over 65535 vertices", 0);
        return 0;
    }
    /* upload with your day 2 VBO/IBO code:
       glBufferData(GL_ARRAY_BUFFER, num_verts * 32, verts, ...)
       glBufferData(GL_ELEMENT_ARRAY_BUFFER, ...)               */
    return 1;
}
```

What is left to you: the VBO/IBO upload (identical to day 2), a `mesh_draw()` that binds the buffers, sets the three `glVertexAttribPointer` calls with stride `sizeof(vertex_t)` and offsets 0, 12, 20, and frees. The worst-case `maxverts` allocation over-reserves; for a 3000-triangle model that is at most 280 KB transient, which your 1 GB can tolerate for the duration of a load.

**NOTE:** The linear search in `find_or_add` is O(verts×corners). A 3000-triangle model does roughly 9000 lookups averaging a few thousand integer compares each — well under a second on the Atom. Around 20–30 thousand triangles it becomes seconds, and the upgrade is a hash: bucket heads in a table of, say, 4096 ints, chained through a parallel `next[]` array, keyed by `((vi*31 + ti)*31 + ni) & 4095`. Same interface, same output, load time back to instant. Do it when it hurts, not before.

**WARNING:** Milkshape export quirks, observed in the wild: it writes `-0.000000` for negative zeros (harmless here — but it is one reason we dedup on *indices*, not on vertex bytes; `memcmp` would treat `-0.0` and `0.0` as different vertices and quietly bloat your buffer). Models without texture coordinates export faces as `v//vn` — that is why `parse_corner` handles it. Files use CRLF line endings, which is why every `strtok` set above includes `\r`. And it emits `s`, `g`, `usemtl` lines you must skip without complaint.

**DONE WHEN:** a model you exported from Milkshape renders with correct normals. Proof of normals: a debug fragment shader with `gl_FragColor = vec4(v_nrm * 0.5 + 0.5, 1.0)` shows a smooth rainbow that rotates with the model — top faces greenish (+Y), right faces reddish (+X).

**IF YOU ARE STUCK:**

- Exploded porcupine: you forgot the 1-based-to-0-based subtraction.
- Model renders but lighting-to-be looks wrong: check the export had normals at all (`vn` count nonzero); Milkshape only writes them when the exporter option is ticked.
- Texture upside-down: OBJ `vt` origin is bottom-left, same as GL. If it is flipped, you are double-flipping in your day 4 TGA loader.
- Garbage after the first few faces: your `f` handler is calling `skip_line` after `fgets` already consumed the newline, eating the next line's tag.

# Day 8: Scene Graph, 1999 Edition

One mesh floating at the origin is a demo. A world has structure: the turret sits on the tank, the hand hangs off the arm, the moon circles the planet. In 1999 "scene graph" was the architecture buzzword of the industry — SGI Performer had one, the doomed SGI/Microsoft Fahrenheit project promised one — and what shipping games actually used was the humble version you build today: a tree of transforms, multiplied parent-into-child on the way down.

## Theory: composing transforms

Each node stores its transform *relative to its parent*: position, Euler rotation in degrees, uniform scale. Its local matrix composes those, and its world matrix is:

```
world = parent_world * local
```

With our conventions (`out = M * v`, column vectors), a chain reads right to left: the vertex is scaled, then rotated, then translated into parent space, then the parent's world matrix carries it the rest of the way. We fix the local composition order, once, for the whole engine:

```
local = T * Ry * Rx * Rz * S
```

Read right to left: scale first, then roll (Z), then pitch (X), then yaw (Y), then translate. This is the classic yaw-pitch-roll order: "turn to face the thing, tilt up at it, then bank." Matrix multiplication does not commute, so pick the order today and never change it. About the "gimbal surprises" in the challenge: with this order, pitch of ±90° aligns the yaw and roll axes and you lose a degree of freedom. It will not bite today — orbits only yaw — but know that it is there. Note also the deliberate *uniform* scale (one float, not three): non-uniform scale bends normals, and when lighting arrives on day 13 you will be glad you never allowed it.

## The node

```
/* ---- listing 8.1: node type and matrix propagation ----------- */

#define MAX_CHILDREN 8

typedef struct scene_node_s {
    vec3_t pos;
    vec3_t rot;             /* euler degrees, applied Y, X, Z  */
    float  scale;
    mesh_t *mesh;           /* NULL = pure transform node      */
    struct scene_node_s *parent;
    struct scene_node_s *children[MAX_CHILDREN];
    int num_children;
    float local[16];
    float world[16];
} scene_node_t;

void node_init(scene_node_t *n, scene_node_t *parent, mesh_t *mesh)
{
    memset(n, 0, sizeof *n);
    n->scale = 1.0f;
    n->mesh = mesh;
    n->parent = parent;
    if (parent) {
        if (parent->num_children >= MAX_CHILDREN) {
```

```
            MessageBoxA(0, "node_init", "too many children", 0);
            return;
        }
        parent->children[parent->num_children++] = n;
    }
}

static void node_update_local(scene_node_t *n)
{
    float t[16], ry[16], rx[16], rz[16], s[16];
    float a[16], b[16];
    mat4_translate(t, n->pos[0], n->pos[1], n->pos[2]);
    mat4_rotate_y(ry, n->rot[1]);
    mat4_rotate_x(rx, n->rot[0]);
    mat4_rotate_z(rz, n->rot[2]);
    mat4_scale(s, n->scale, n->scale, n->scale);
    mat4_multiply(a, t, ry);         /* a = T * Ry            */
    mat4_multiply(b, a, rx);         /* b = T * Ry * Rx       */
    mat4_multiply(a, b, rz);         /* a = T * Ry * Rx * Rz  */
    mat4_multiply(n->local, a, s);   /* local = a * S         */
}

void node_update_world(scene_node_t *n)
{
    int i;
    node_update_local(n);
    if (n->parent)
        mat4_multiply(n->world, n->parent->world, n->local);
    else
        memcpy(n->world, n->local, sizeof n->world);
    for (i = 0; i < n->num_children; ++i)
        node_update_world(n->children[i]);
}

void node_render(scene_node_t *n, const float *viewproj)
{
    float mvp[16];
    int i;
    if (n->mesh) {
        mat4_multiply(mvp, viewproj, n->world);
        glUniformMatrix4fv(g_u_mvp, 1, GL_FALSE, mvp);
        mesh_draw(n->mesh);
    }
    for (i = 0; i < n->num_children; ++i)
        node_render(n->children[i], viewproj);
}
```

Call `node_update_world(&g_root)` once per `engine_update` tick, after your animation code has poked the `pos`/`rot` fields, and `node_render(&g_root, viewproj)` in `engine_render`. One recursive pass each; that is the whole "one matrix stack" of the DONE line — the stack is the C call stack.

> **WARNING:** Your day 1 spec says `mat4_multiply(out,a,b)` may **not** alias — `out` must not be `a` or `b`. Notice how listing 8.1 ping-pongs between `a` and `b` for exactly this reason. Write `mat4_multiply(n->world, n->world, x)` and you will get a smeared, shearing model and an afternoon of confusion.

## Worked example: sun, planet, moon

The trick is *pivot nodes* — empty nodes with no mesh. If the planet were a direct child of the sun, the sun's self-spin would drag the planet around at the sun's rotation rate, and the planet's own spin would drag the moon. Splitting "orbit" from "spin" with empties decouples them:

```
root
 +- sun         mesh, scale 2.0        spins  5 deg/s
 +- orbit1      empty                  spins 20 deg/s
     +- pivot   empty, pos x = 8
         +- planet  mesh, scale 0.7    spins 45 deg/s
         +- orbit2  empty              spins 60 deg/s
             +- moon  mesh, pos x = 2, scale 0.25
```

```
/* setup, once */
node_init(&g_root,   NULL,      NULL);
node_init(&g_sun,    &g_root,   &g_ball);
node_init(&g_orbit1, &g_root,   NULL);
node_init(&g_pivot,  &g_orbit1, NULL);
node_init(&g_planet, &g_pivot,  &g_ball);
node_init(&g_orbit2, &g_pivot,  NULL);
node_init(&g_moon,   &g_orbit2, &g_ball);
g_sun.scale    = 2.0f;
g_pivot.pos[0] = 8.0f;
g_planet.scale = 0.7f;
g_moon.pos[0]  = 2.0f;
g_moon.scale   = 0.25f;

/* each update tick, dt = 1.0/60.0 */
g_sun.rot[1]    += 5.0f  * (float)dt;
g_orbit1.rot[1] += 20.0f * (float)dt;   /* year: 18 seconds  */
g_planet.rot[1] += 45.0f * (float)dt;   /* the planet's day  */
g_orbit2.rot[1] += 60.0f * (float)dt;   /* month: 6 seconds  */
node_update_world(&g_root);
```

Follow the moon's matrix chain by hand once, it is worth it: `world = orbit2.world * moon.local`, and `orbit2.world` already contains `orbit1(Ry) * T(8,0,0)`. So the moon sits 2 units from a point that is itself 8 units from the origin and swinging around it. The moon completes an orbit around the planet every 6 seconds while the whole assembly takes 18 seconds around the sun. Note that `g_pivot` carries the translation and `g_orbit2` the rotation — both are cheap, both are invisible, and this pattern (transform-only nodes as joints) is exactly how skeletal animation will work on day 24.

## Pitfalls and performance

Angles accumulate forever; after a few hours `rot[1]` is a large float and precision decays. Wrap: if it exceeds 360, subtract 360. Update order matters: mutate transforms, *then* propagate, *then* render — propagate inside the fixed-timestep update, not in render, or a 30 fps hitch will visibly desync parents from children. Cost on the Atom: each node is 5 matrix multiplies (~64 mul + 48 add each) plus one per parent link; a few hundred nodes per frame is nothing. Recursion depth equals tree depth — single digits — so the XP 1 MB stack is not remotely threatened.

**DONE WHEN:** a moon orbits a planet orbiting a sun, each body also spinning on its own axis at its own rate, and changing the sun's spin rate does not change the planet's orbit rate.

**IF YOU ARE STUCK:**

- Moon orbits the sun instead of the planet: it is parented to `orbit1` or `pivot`'s sibling level, not under the planet's pivot.
- Bodies fly outward in a spiral: you composed `S` or the rotations *before* `T` reading left to right — i.e. you built `Ry * T` where you meant `T * Ry`.
- Model shears or smears: aliased `mat4_multiply`. See the warning above.
- Everything is at the origin: you forgot to propagate (`node_update_world`) before rendering.

# Day 9: Frustum Culling

Scatter 100 cubes around the world, spin the camera, and watch the frame time: you are paying full price for every cube, including the 90 behind your head. In 1999 this was life or death — with software transform, every vertex you failed to cull was transformed by the CPU. It still matters here, differently: on this machine every draw call crosses ANGLE into D3D9 and costs real CPU (tens of microseconds each), and the SGX's vertex shader is no monster either. The cheapest triangle is the one you never submit.

The tool is frustum culling: six planes bound the visible volume, and one cheap test per object — not per triangle — rejects whole meshes. Sphere-vs-frustum is 6 dot products. You will do thousands per frame without noticing.

## Theory: planes from the view-projection matrix

This is the Gribb–Hartmann method, and it looks like magic until you derive it once. Let `M = proj * view`, our world-to-clip matrix, and `v` a world-space point. Then `clip = M * v`, and with our column-vector convention each clip component is a dot product of `v` against a **row** of `M`:

```
clip.x = row0 . v      clip.z = row2 . v
clip.y = row1 . v      clip.w = row3 . v
```

A point is inside the frustum exactly when, in clip space (standard GL conventions — ANGLE's [0,1] depth remap happens after this, invisibly to you):

```
-w <= x <= w     -w <= y <= w     -w <= z <= w
```

Take the left bound: `x >= -w` is `row0.v + row3.v >= 0`, i.e. `(row3 + row0) . v >= 0`. But that sum of rows is just a 4-vector `(a,b,c,d)`, and `ax + by + cz + d >= 0` is a half-space: a world-space plane with normal `(a,b,c)` pointing *into* the frustum. Six inequalities, six planes:

```
left  = row3 + row0      bottom = row3 + row1     near = row3 + row2
right = row3 - row0      top    = row3 - row1     far  = row3 - row2
```

One convention check, because this is where everyone's culling breaks: our matrices are column-major, element (column c, row r) at `m[c*4+r]`. So **row r** of `M` is the strided sequence `m[r], m[4+r], m[8+r], m[12+r]`. If you grab columns instead (contiguous `m[0..3]`), you get the planes of the *transposed* matrix and everything culls wrongly the moment the camera rotates.

The raw plane 4-vectors are not unit length. That is fine for point-inside tests (sign only), but we compare distances against a sphere radius, so normalize: divide all four components by the length of the normal.

## The code

```
/* ---- listing 9.1: plane extraction and sphere test ----------- */

typedef struct {
    vec3_t n;               /* points into the frustum          */
    float  d;               /* dist(p) = DotProduct(n,p) + d    */
} plane_t;

static void plane_set(plane_t *pl, int comp, float value)
{
```

```
    if (comp < 3) pl->n[comp] = value;
    else          pl->d = value;
}

void frustum_from_viewproj(plane_t p[6], const float *m)
{
    int c;
    for (c = 0; c < 4; ++c) {
        float r0 = m[c*4 + 0];       /* row 0, component c */
        float r1 = m[c*4 + 1];
        float r2 = m[c*4 + 2];
        float r3 = m[c*4 + 3];
        plane_set(&p[0], c, r3 + r0);    /* left   */
        plane_set(&p[1], c, r3 - r0);    /* right  */
        plane_set(&p[2], c, r3 + r1);    /* bottom */
        plane_set(&p[3], c, r3 - r1);    /* top    */
        plane_set(&p[4], c, r3 + r2);    /* near   */
        plane_set(&p[5], c, r3 - r2);    /* far    */
    }
    for (c = 0; c < 6; ++c) {
        float len = (float)sqrt(DotProduct(p[c].n, p[c].n));
        if (len > 0.0f) {
            float inv = 1.0f / len;
            VectorScale(p[c].n, inv, p[c].n);
            p[c].d *= inv;
        }
    }
}

/* 1 = visible (inside or straddling), 0 = fully outside */
int frustum_test_sphere(const plane_t p[6], const vec3_t c, float r)
{
    int i;
    for (i = 0; i < 6; ++i)
        if (DotProduct(p[i].n, c) + p[i].d < -r)
            return 0;
    return 1;
}
```

Each mesh needs a bounding sphere, computed once at load from its extents: center of the axis-aligned box, radius to the far corner.

```
/* ---- listing 9.2: mesh bounds + culled render ---------------- */

void mesh_compute_bounds(mesh_t *m)
{
    vec3_t mins, maxs, half;
    int i, k;
    for (k = 0; k < 3; ++k)
        mins[k] = maxs[k] = m->verts[0].pos[k];
    for (i = 1; i < m->num_verts; ++i)
        for (k = 0; k < 3; ++k) {
            float v = m->verts[i].pos[k];
            if (v < mins[k]) mins[k] = v;
            if (v > maxs[k]) maxs[k] = v;
        }
    VectorAdd(mins, maxs, m->center);
    VectorScale(m->center, 0.5f, m->center);
    VectorSubtract(maxs, m->center, half);
    m->radius = (float)sqrt(DotProduct(half, half));
}

int g_draws, g_culled;       /* reset both to 0 each frame */
```

```
void node_render(scene_node_t *n, const float *viewproj,
                 const plane_t planes[6])
{
    float mvp[16];
    vec3_t wc;
    float sc;
    int i;
    if (n->mesh) {
        const float *w = n->world;
        const float *c = n->mesh->center;
        /* sphere center to world space: wc = world * (c,1)   */
        wc[0] = w[0]*c[0] + w[4]*c[1] + w[8]*c[2]  + w[12];
        wc[1] = w[1]*c[0] + w[5]*c[1] + w[9]*c[2]  + w[13];
        wc[2] = w[2]*c[0] + w[6]*c[1] + w[10]*c[2] + w[14];
        /* accumulated uniform scale = length of column 0     */
        sc = (float)sqrt(w[0]*w[0] + w[1]*w[1] + w[2]*w[2]);
        if (frustum_test_sphere(planes, wc,
                                n->mesh->radius * sc)) {
            mat4_multiply(mvp, viewproj, n->world);
            glUniformMatrix4fv(g_u_mvp, 1, GL_FALSE, mvp);
            mesh_draw(n->mesh);
            g_draws++;
        } else {
            g_culled++;
        }
    }
    for (i = 0; i < n->num_children; ++i)
        node_render(n->children[i], viewproj, planes);
}
```

Two subtleties in there. The sphere center must be transformed to world space — a mesh modeled off-center would otherwise cull against the wrong position. And the radius must be multiplied by the *accumulated* scale of the whole parent chain, not just `n->scale`; since our scales are uniform, the length of any world-matrix basis column gives it directly. Yesterday's discipline about uniform scale just paid its first dividend.

The HUD counter, with your day 6 font (remember: VC6 has no `snprintf`, and `_snprintf` does not guarantee termination):

```
char buf[64];
_snprintf(buf, sizeof buf, "draw %d  cull %d", g_draws, g_culled);
buf[sizeof buf - 1] = '\0';
font_draw(4, 20, buf);
```

## Pitfalls and performance

Symptoms map to causes with satisfying precision. Objects vanish at the screen edges while still half-visible: planes not normalized, so the radius comparison is scaled garbage. Everything culls when you face it and appears when you look away: rows/columns transposed. Objects pop out when close to the camera: near-plane sign flipped. Objects vanish only when scaled up: you forgot the radius scale. Test methodically: freeze the frustum (press a key to stop updating the planes), then fly around and *look* at the culling boundary from outside — this one trick turns frustum debugging from guesswork into inspection.

Cost: extraction is ~50 flops once per frame; each sphere test is at most 6 dots (18 mul, 18 add/cmp) and early-outs on the first failing plane. On the Atom, budget roughly 10,000 tests per millisecond — you will run out of scene long before you run out of test budget. The win side: every culled cube saves a full ANGLE→D3D9 draw call plus its vertex work on the SGX.

**DONE WHEN:** with 100 cubes scattered around you, the overlay shows draw ≈ 100 looking at the crowd, and near zero (0–5) looking directly away — and nothing visibly pops at the screen edges while turning.

**IF YOU ARE STUCK:**

- Draw count never drops: you extract planes from `view` or `proj` alone. It must be the product `proj * view`.
- Culls exactly backwards: your planes point outward. Check one by hand: stand at origin looking down -Z, the near plane normal should point toward -Z... i.e. into the volume in front of you.
- Edge popping: normalize the planes; verify with a sphere of radius 0 (a point) which must never pop.
- Counters climb forever: reset `g_draws`/`g_culled` at the top of `engine_render`, not at init.

# Day 10: The Skybox

Look past your cubes and there is the void: clear color, flat and shameless. Every game of your target era wrapped the world in a skybox — Quake II's cloud boxes, Unreal's painted skies — six images on the inside of a cube that never gets closer no matter how far you fly. The illusion is two tricks: strip the camera's translation so the box is always centered on your head, and write no depth so the world draws over it. Combined, the sky reads as infinitely far away for the price of 12 triangles.

## Theory: sampling by direction

A cubemap is six square textures logically folded into a box around the origin. You do not sample it with UVs; you sample it with a *direction*. GLSL's `samplerCube` takes a 3-vector, finds which component has the largest magnitude (that picks the face), and divides the other two by it (that picks the texel within the face). So the vertex shader for a skybox is almost nothing: render a unit cube centered on the camera and pass the vertex *position itself* as the sample direction. No normalization needed — only the direction matters, not the length.

Centered on the camera is the key phrase. The view matrix carries rotation and translation; copy it and zero the translation — in our column-major layout that is elements `m[12]`, `m[13]`, `m[14]`. The box then rotates with your head but never moves, which is precisely what an infinitely distant object does. Parallax is the depth cue; something with zero parallax reads as infinity.

## Building the cubemap

Six images. Paint them in Paint Shop Pro or generate them with any terragen-style tool you have on disk; they must all be square and the same size (512×512 is plenty at 1024×600 — that is 4.5 MB of RGB texture memory for the set).

```
/* ---- listing 10.1: cubemap creation ------------------------- */

static const char *SKY_FACES[6] = {
    "sky_px.tga", "sky_nx.tga",      /* +X -X : right, left  */
    "sky_py.tga", "sky_ny.tga",      /* +Y -Y : top, bottom  */
    "sky_pz.tga", "sky_nz.tga"       /* +Z -Z : back, front  */
};

GLuint skybox_load(void)
{
    GLuint tex;
    int i, w, h, comp;
    glGenTextures(1, &tex);
    glBindTexture(GL_TEXTURE_CUBE_MAP, tex);
    glPixelStorei(GL_UNPACK_ALIGNMENT, 1);
    for (i = 0; i < 6; ++i) {
        unsigned char *px = stbi_load(SKY_FACES[i],
                                      &w, &h, &comp, 3);
        if (!px) {
            MessageBoxA(0, SKY_FACES[i], "sky: load failed", 0);
            return 0;
        }
        glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, 0,
                     GL_RGB, w, h, 0, GL_RGB,
                     GL_UNSIGNED_BYTE, px);
        stbi_image_free(px);
    }
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
```

```
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_S,
                    GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_WRAP_T,
                    GL_CLAMP_TO_EDGE);
    return tex;
}
```

The six face targets are consecutive enums, hence the `+ i` — the order is +X, -X, +Y, -Y, +Z, -Z and you must load in exactly that order. `GL_CLAMP_TO_EDGE` is not optional: with repeat wrapping, bilinear filtering at each face's border grabs texels from the opposite edge of the *same* image and draws a visible seam line along every cube edge. (GLES2 has no seamless cubemap filtering; clamp is as good as it gets, and it is good enough.)

## Drawing it

```
/* ---- listing 10.2: shaders and the render call --------------- */

static const char *SKY_VS =
"attribute vec3 a_pos;\n"
"uniform mat4 u_viewproj;\n"
"varying vec3 v_dir;\n"
"void main() {\n"
"    v_dir = a_pos;\n"
"    gl_Position = u_viewproj * vec4(a_pos, 1.0);\n"
"}\n";

static const char *SKY_FS =
"precision mediump float;\n"
"uniform samplerCube u_sky;\n"
"varying vec3 v_dir;\n"
"void main() {\n"
"    gl_FragColor = textureCube(u_sky, v_dir);\n"
"}\n";

/* first thing in engine_render, right after glClear */
void skybox_render(const float *view, const float *proj)
{
    float v[16], vp[16];
    memcpy(v, view, sizeof v);
    v[12] = v[13] = v[14] = 0.0f;    /* strip translation */
    mat4_multiply(vp, proj, v);

    glDepthMask(GL_FALSE);           /* no depth writes   */
    glDisable(GL_CULL_FACE);         /* we are inside it  */
    glUseProgram(g_sky_prog);
    glUniformMatrix4fv(g_sky_u_vp, 1, GL_FALSE, vp);
    glBindTexture(GL_TEXTURE_CUBE_MAP, g_sky_tex);
    /* bind the unit-cube VBO/IBO from day 2, positions only,
       and set its attribute pointer here */
    glDrawElements(GL_TRIANGLES, 36, GL_UNSIGNED_SHORT, 0);
    glEnable(GL_CULL_FACE);
    glDepthMask(GL_TRUE);
}
```

The geometry is your day 2 cube, positions only, corners at ±1. With translation stripped, the farthest corner is $\sqrt{3} \approx 1.73$ units away — comfortably inside any sane near/far range, so the box itself never clips. We are inside the cube, so either wind the faces inward or, as above, disable culling for one draw. Draw the sky **first**, depth writes off: the depth buffer still reads all-clear, the sky passes the test everywhere, writes nothing, and everything you draw afterward lands on top of

it. You can even drop `GL_COLOR_BUFFER_BIT` from your clear once this works — the sky repaints every pixel anyway (keep clearing depth).

> **NOTE:** The desktop-GL folklore says "draw the sky *last* with z forced to the far plane, so occluded sky pixels never shade." On the SGX545 you can ignore it: a tile-based deferred renderer resolves per-pixel visibility before shading fragments, so opaque overdraw of the sky costs almost nothing regardless of draw order. Sky-first is simpler and correct. File the trick away for desktop ports.

## Face orientation: the inevitable rotated sky

The first run almost always shows a sky that is mirrored, rotated, or has a scrambled top. This is not your bug. GL's cubemap face orientation follows a convention inherited from RenderMan — effectively a *left-handed* layout viewed from inside — and it agrees with nobody's image-editor intuition, while every sky-image set on your disk uses front/back/left/right/up/down names mapped however its author felt that day. Fix it empirically, in this order:

- Horizon faces appear in the wrong compass order as you spin in place: swap the +X and -X images.
- Horizon order correct but each face mirrored left-right: your "front" convention differs from GL's — note that with our camera looking down -Z, the face you see ahead is **-Z** (sample direction (0,0,-1) has largest magnitude in negative z), not +Z. Swap +Z and -Z.
- Side faces fine, but the top (+Y) seam does not line up with any horizon face: rotate the +Y image in 90° steps in Paint Shop Pro until its edges match. Then do the bottom.

This is the authentic 1999 workflow: rotate the bitmap, save, run, squint, repeat. Ten minutes, once, and write the final arrangement in your journal so you never re-derive it.

> **DONE WHEN:** the sky surrounds the world with no visible seams, all four horizon faces meet correctly, the top matches everywhere, and flying in any direction produces zero parallax — the sky never gets closer.

**IF YOU ARE STUCK:**

- Black screen: cull face — you are inside the cube; disable culling or flip winding.
- Sky moves with you (parallax): you zeroed the translation of the wrong matrix, or multiplied `view * proj` instead of `proj * view`.
- World invisible behind sky: you left `glDepthMask` off after the sky draw — re-enable it, and remember it also gates depth *clears*.
- Thin bright seam lines on cube edges: wrap mode is not `GL_CLAMP_TO_EDGE` on both S and T.
- GL_INVALID_OPERATION from glTexImage2D: a face image is not square or the six sizes differ.

# Day 11: Heightmap Terrain

Tribes, Delta Force, Myth, every flight sim on the shelf: the signature big-world look of the late 90s was heightmap terrain. The idea is beautiful economy — terrain is a function y = f(x,z), so store it as a grayscale image: pixel brightness is elevation. An artist paints mountains with an airbrush in Paint Shop Pro; a 513×513 image encodes half a million triangles in a 257 KB file. Today you turn a picture into land and fly over it.

## Theory: image to grid

A W×H grayscale image becomes a grid of W×H vertices in the XZ plane. Grid point (x,z) sits at world position:

```
px = (x - (W-1)/2) * cell        cell   = world units per grid step
py = pixel(x,z) / 255 * yscale   yscale = world height of white
pz = (z - (H-1)/2) * cell
```

(The offsets center the terrain on the origin, which keeps camera numbers small and pleasant.) With `cell = 1.0` and `yscale = 25.0`, a 129×129 map is a 128×128 m world with 25 m peaks. Adjacent grid points connect into two triangles per cell. We index them as plain **indexed triangles**, not triangle strips: strips were the fashionable choice when index bandwidth mattered, but they need degenerate stitching between rows, and on hardware with a post-transform vertex cache indexed triangles reuse vertices nearly as well. Simpler wins.

Normals come from **central differences** — the discrete gradient of the height field. The slope along x at a grid point is `(h[x+1] - h[x-1]) / (2*cell)`, and a surface y = f(x,z) has normal direction `(-df/dx, 1, -df/dz)`. Multiply through by `2*cell` (length does not matter, we normalize) and you get the classic three-subtraction formula:

```
n = normalize( h[x-1][z] - h[x+1][z],
               2 * cell,
               h[x][z-1] - h[x][z+1] )
```

Because it samples both neighbors, it smooths single-pixel noise for free and is exact for constant slopes. Texture coordinates: use world position times a tiling factor (`uv = world_xz * uvscale`) with `GL_REPEAT` wrapping, so a grass texture repeats every `1/uvscale` world units regardless of map size — `uvscale = 0.25` tiles every 4 m, about right for a 256×256 detail texture.

## The code

```
/* ---- listing 11.1: heightmap to mesh ------------------------ */

typedef struct {
    int    w, h;             /* grid size in vertices           */
    float  cell, yscale, uvscale;
    float  xoff, zoff;       /* centering offsets               */
    float *heights;          /* w*h world-space Y values        */
    mesh_t mesh;
} terrain_t;

static float hget(const terrain_t *t, int x, int z)
{
    if (x < 0) x = 0;
    if (x > t->w - 1) x = t->w - 1;
    if (z < 0) z = 0;
    if (z > t->h - 1) z = t->h - 1;
    return t->heights[z * t->w + x];
```

```
}

void terrain_normal(const terrain_t *t, int x, int z, float *n)
{
    n[0] = hget(t, x - 1, z) - hget(t, x + 1, z);
    n[1] = 2.0f * t->cell;
    n[2] = hget(t, x, z - 1) - hget(t, x, z + 1);
    VectorNormalize(n);
}

int terrain_build(terrain_t *t, const char *path,
                  float cell, float yscale, float uvscale)
{
    unsigned char *img;
    unsigned short *idx;
    vertex_t *v;
    int comp, x, z, i, nv, ni;

    img = stbi_load(path, &t->w, &t->h, &comp, 1);
    if (!img) {
        MessageBoxA(0, path, "terrain: load failed", 0);
        return 0;
    }
    t->cell = cell;  t->yscale = yscale;  t->uvscale = uvscale;
    t->xoff = -0.5f * (t->w - 1) * cell;
    t->zoff = -0.5f * (t->h - 1) * cell;

    nv = t->w * t->h;
    ni = (t->w - 1) * (t->h - 1) * 6;
    t->heights = (float *)malloc(sizeof(float) * nv);
    v   = (vertex_t *)malloc(sizeof(vertex_t) * nv);
    idx = (unsigned short *)malloc(sizeof(unsigned short) * ni);
    for (i = 0; i < nv; ++i)
        t->heights[i] = (img[i] / 255.0f) * yscale;
    stbi_image_free(img);

    for (z = 0; z < t->h; ++z)
        for (x = 0; x < t->w; ++x) {
            vertex_t *dst = &v[z * t->w + x];
            dst->pos[0] = t->xoff + x * cell;
            dst->pos[1] = t->heights[z * t->w + x];
            dst->pos[2] = t->zoff + z * cell;
            dst->uv[0]  = dst->pos[0] * uvscale;
            dst->uv[1]  = dst->pos[2] * uvscale;
            terrain_normal(t, x, z, dst->nrm);
        }

    i = 0;
    for (z = 0; z < t->h - 1; ++z)
        for (x = 0; x < t->w - 1; ++x) {
            unsigned short i0 =
                (unsigned short)(z * t->w + x);
            unsigned short i1 = (unsigned short)(i0 + 1);
            unsigned short i2 = (unsigned short)(i0 + t->w);
            unsigned short i3 = (unsigned short)(i2 + 1);
            idx[i++] = i0; idx[i++] = i2; idx[i++] = i1;
            idx[i++] = i1; idx[i++] = i2; idx[i++] = i3;
        }

    /* upload v (nv verts) and idx (ni indices) with the day 2
       buffer code into t->mesh; free v and idx after upload.
       KEEP t->heights: day 12 chunks and day 22 collision
       both read it. */
    return 1;
}
```

The index winding (`i0, i2, i1`) makes triangles counter-clockwise seen from above (+Y), matching GL's default front face — verify by enabling `glCullFace` and confirming the terrain is visible from above, invisible from underneath. Set the detail texture's wrap mode to `GL_REPEAT` on both axes; that requires power-of-two texture dimensions in GLES2, so keep the detail texture 256×256 (the *heightmap* has no such restriction — it never touches GL).

> **TIP:** The vertex buffer already carries the normals, and your normals-as-color debug shader from day 7 works unchanged — hills glow green on top, slopes shade toward red and blue. Enjoy the preview: on day 13 those normals meet a sun direction and one dot product turns this from a colorful quilt into lit land. As a second teaser, sample the grazing-angle blur on distant slopes: `EXT_texture_filter_anisotropic` is in your extension list, and `glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, 4.0f)` sharpens it for a modest fill cost.

## The budget: where the Atom starts to sweat

Count triangles honestly. A W×W map draws $(W-1)^2 \times 2$:

| Heightmap | Vertices | Triangles | Verdict |
|---|---|---|---|
| 129×129 | 16,641 | 32,768 | fine at 60 Hz, one draw call |
| 257×257 | 66,049 | 131,072 | vertex count breaks 16-bit indices; borderline anyway |
| 513×513 | 263,169 | 524,288 | ~31M tris/s at 60 Hz: not per-frame on this GPU |

Two walls, note, not one. The soft wall is throughput: half a million triangles per frame is beyond what the SGX545 will transform at 60 Hz, and vsync quantization means missing 16.6 ms does not cost you a little — it drops you straight to 30. The hard wall is the index type: past 65,535 vertices, `unsigned short` indices cannot address the grid at all. (`OES_element_index_uint` is in your extension list and would paper over the second wall, but not the first.) The 1999 answer to both is the same and it is tomorrow's work: never draw the whole world. For today, ship the 129×129 map, then try 257 with a split into two meshes if you are impatient, and journal the fps at each size — the challenge asks for exactly that number.

> **DONE WHEN:** you can fly over rolling hills with correct smooth normals (rainbow debug shader), a tiled detail texture, at 60 fps on the 129×129 map — and your journal records the triangle count where the frame rate first dropped.

**IF YOU ARE STUCK:**

- Terrain is a flat plane: `stbi_load` returned a non-grayscale image but you indexed it as one — you passed 0 instead of 1 for the requested component count.
- Spiky noise everywhere: your image was saved with dithering (PSP7 loves to dither gradients on save); re-save as true 8-bit grayscale, no dithering, and blur it slightly.
- Terrain visible only from below: winding backwards; swap `i1` and `i2` in both triangles.
- Checkerboard of missing triangles: your index count is (W-1)×(H-1)×6 — check you did not use W×H.
- Texture is one giant stretched image: wrap mode is clamp, or your uv scale multiplies grid coordinates instead of world coordinates.

# Day 12: Terrain Chunks

Yesterday ended at a wall: the 513×513 map is half a million triangles and a quarter million vertices, and neither the GPU throughput nor 16-bit indices will carry it in one piece. The 1999 answer — the answer under every big outdoor game of the era, and the first step toward the geomipmapped LOD in the extra-credit list — is to cut the terrain into chunks and let day 9's frustum culling discard most of them. You already own every tool this needs. Today is pure assembly, and the payoff is the best performance-per-line-of-code in the whole book.

## Theory: cutting the grid

Chunks are **33×33 vertices** — 32×32 cells, 2048 triangles, 6144 indices each. Adjacent chunks *share* their edge row and column of vertices (the last row of one chunk is the first row of the next), so there are no cracks: both chunks evaluate the same world positions along the seam. That means a map divides cleanly when its size is 32n+1 vertices: the 513×513 map yields a 16×16 grid of 256 chunks. (129×129 gives 4×4 = 16 — nice for testing.)

Two design decisions to make explicitly. First, geometry storage: **one VBO per chunk**, rather than one big VBO with per-chunk index ranges. The big-VBO plan sounds cleaner until you meet its problems on this stack: a 513×513 vertex pool needs 32-bit indices (extension, more bandwidth), and GLES2 has no `glDrawElementsBaseVertex`, so you cannot cheaply rebase 16-bit indices per chunk into a shared pool. Per-chunk VBOs each stay comfortably under 16-bit indexing and cost only memory: 1089 vertices × 32 bytes ≈ 34 KB per chunk, 8.5 MB for all 256. Acceptable, and every chunk becomes trivially independent — which day 20's resource manager and any future LOD scheme will thank you for.

Second, a genuinely free win: every chunk has *identical topology* — same 33×33 grid, same triangle order — so all 256 chunks share **one** index buffer with local indices 0..1088. Build it once.

Culling needs a bound per chunk. The XZ extent is known from the chunk's grid rectangle; the Y extent comes from the min and max height inside the chunk, tracked while filling its vertices. That axis-aligned box converts to a bounding sphere exactly as on day 9: center = box middle, radius = distance to a corner. Terrain never moves (its world matrix is identity), so these spheres are already in world space — computed once at load, never touched again.

## The code

```
/* ---- listing 12.1: chunk field ------------------------------ */

#define CHUNK_CELLS 32
#define CHUNK_VERTS (CHUNK_CELLS + 1)
#define CHUNK_NIDX  (CHUNK_CELLS * CHUNK_CELLS * 6)

typedef struct {
    GLuint vbo;
    vec3_t center;
    float  radius;
} chunk_t;

typedef struct {
    chunk_t *chunks;
    int nx, nz;              /* chunk grid dimensions   */
    GLuint shared_ibo;       /* one topology, 6144 idx  */
} chunkfield_t;

static void chunks_build_ibo(chunkfield_t *cf)
{
    unsigned short idx[CHUNK_NIDX];      /* 12 KB, stack is fine */
```

```
    unsigned short *p = idx;
    int x, z;
    for (z = 0; z < CHUNK_CELLS; ++z)
        for (x = 0; x < CHUNK_CELLS; ++x) {
            unsigned short i0 =
                (unsigned short)(z * CHUNK_VERTS + x);
            unsigned short i1 = (unsigned short)(i0 + 1);
            unsigned short i2 = (unsigned short)(i0 + CHUNK_VERTS);
            unsigned short i3 = (unsigned short)(i2 + 1);
            *p++ = i0; *p++ = i2; *p++ = i1;
            *p++ = i1; *p++ = i2; *p++ = i3;
        }
    glGenBuffers(1, &cf->shared_ibo);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, cf->shared_ibo);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof idx, idx,
                 GL_STATIC_DRAW);
}

static void chunk_build(const terrain_t *t, chunk_t *c,
                        int cx, int cz)
{
    vertex_t v[CHUNK_VERTS * CHUNK_VERTS];   /* ~34 KB stack */
    vec3_t mins, maxs, half;
    float ymin = 1e30f, ymax = -1e30f;
    int x, z;

    for (z = 0; z < CHUNK_VERTS; ++z)
        for (x = 0; x < CHUNK_VERTS; ++x) {
            int gx = cx * CHUNK_CELLS + x;
            int gz = cz * CHUNK_CELLS + z;
            vertex_t *dst = &v[z * CHUNK_VERTS + x];
            dst->pos[0] = t->xoff + gx * t->cell;
            dst->pos[1] = t->heights[gz * t->w + gx];
            dst->pos[2] = t->zoff + gz * t->cell;
            dst->uv[0]  = dst->pos[0] * t->uvscale;
            dst->uv[1]  = dst->pos[2] * t->uvscale;
            terrain_normal(t, gx, gz, dst->nrm);
            if (dst->pos[1] < ymin) ymin = dst->pos[1];
            if (dst->pos[1] > ymax) ymax = dst->pos[1];
        }

    mins[0] = v[0].pos[0];  maxs[0] = mins[0] +
              CHUNK_CELLS * t->cell;
    mins[1] = ymin;         maxs[1] = ymax;
    mins[2] = v[0].pos[2];  maxs[2] = mins[2] +
              CHUNK_CELLS * t->cell;
    VectorAdd(mins, maxs, c->center);
    VectorScale(c->center, 0.5f, c->center);
    VectorSubtract(maxs, c->center, half);
    c->radius = (float)sqrt(DotProduct(half, half));

    glGenBuffers(1, &c->vbo);
    glBindBuffer(GL_ARRAY_BUFFER, c->vbo);
    glBufferData(GL_ARRAY_BUFFER, sizeof v, v, GL_STATIC_DRAW);
}
```

Note that `terrain_normal` reads the *full* heightfield through `hget`, not the chunk's local copy. This matters: a vertex on a chunk border gets its neighbors from the adjacent chunk's territory, so both chunks compute the identical normal and the lighting (day 13) will show no seams. Compute normals from per-chunk data and you will earn a grid of faint crease lines across the whole world.

```
/* ---- listing 12.2: culled render ---------------------------- */

void chunks_render(const chunkfield_t *cf, const plane_t pl[6])
```

```
{
    int i, n = cf->nx * cf->nz;
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, cf->shared_ibo);
    for (i = 0; i < n; ++i) {
        const chunk_t *c = &cf->chunks[i];
        if (!frustum_test_sphere(pl, c->center, c->radius)) {
            g_culled++;
            continue;
        }
        glBindBuffer(GL_ARRAY_BUFFER, c->vbo);
        /* re-issue all three glVertexAttribPointer calls HERE:
           attribute pointers latch the ARRAY_BUFFER that was
           bound when they were specified, not at draw time   */
        glDrawElements(GL_TRIANGLES, CHUNK_NIDX,
                       GL_UNSIGNED_SHORT, 0);
        g_draws++;
    }
}
```

The comment in the loop is the classic GLES2 trap of the day: `glVertexAttribPointer` captures the currently bound `GL_ARRAY_BUFFER` at the moment you call it. Bind a different VBO and draw without re-specifying the pointers and you render the *previous* chunk's vertices — usually as one chunk of terrain repeated 256 times, which looks like abstract art. (`OES_vertex_array_object` is in your extension list and bundles the pointer state per VBO; a tidy upgrade once this works naked.)

## The numbers: before and after

Load the 513×513 map and watch the day 9 overlay. Without culling: 256 draw calls, 524,288 triangles, every frame, regardless of where you look. Through ANGLE each draw call costs on the order of 20–40 µs of CPU before the GPU sees anything — 256 draws is 5–10 ms of pure submission overhead, most of a 16.6 ms frame gone before a pixel shades, plus a triangle load the SGX cannot finish at 60 Hz. Expect the vsync quantizer to hand you 20–30 fps.

With culling, standing in a valley looking along it (90° fov, far plane ~300 with cell = 1.0): the frustum covers roughly a quarter of the map, the overlay reads something like `draw 58 cull 198`, call it 120 K triangles and ~2 ms of submission. Back to 60. Turn to face across the valley at a mountain wall and watch the draw count fall further as near chunks occlude nothing but the frustum tightens — the count breathing as you pan is the day's DONE line made visible. Journal both numbers; they are your baseline when day 29's performance hunt comes. And when the far plane at 300 shows chunks popping into view on the horizon — it will — you have found the exact problem day 17's fog was invented to hide.

> **DONE WHEN:** the 513×513 map runs with the overlay showing draw + cull = total chunks, the draw count falls as you look along the valley (and rises as you climb and see more), frame rate holds 60 in typical views, and no cracks or shading seams show at any chunk border.

**IF YOU ARE STUCK:**

- One chunk's terrain repeated everywhere: attribute pointers not re-specified after binding each chunk's VBO. Read the comment in listing 12.2 again, slower.
- Valley chunks vanish while on screen: your bounding sphere ignored the Y extent — a flat chunk is a thin slab, but its sphere must still cover ymin..ymax.
- Cracks along seams: your map is not 32n+1 (a 512×512 image must be resized to 513, or clamp the final chunk), or chunk (cx,cz) starts at grid cell cx*33 instead of cx*32.
- Faint grid of shading creases: normals computed from chunk-local heights; use the full-field `hget`.
- Whole terrain disappears at odd angles: you fed the chunk world matrix (identity) through the day 9 scale-extraction path with an uninitialized matrix — terrain needs no matrix math at all; test the stored world-space sphere directly.

# Day 13: Per-Vertex Lighting

Until today your world has been lit by nothing at all: every texel arrives on screen at exactly the brightness you painted it. Today you build the classic lighting model — the one that shipped in silicon on the GeForce 256 in October 1999 as "hardware T&L" and that every fixed-function driver before it ran on the CPU. You were denied the fixed-function pipeline; tonight you will have reimplemented its lighting stage yourself, in about thirty lines of GLSL, and you will understand it better than anyone who merely called `glEnable(GL_LIGHTING)`.

## The classic model

A matte (diffuse, Lambertian) surface scatters incoming light equally in all directions, so its apparent brightness does not depend on where the camera is — only on how squarely the light hits it. Tilt a page away from a lamp and it dims by the cosine of the tilt angle. If `N` is the unit surface normal and `L` is a unit vector pointing from the surface *toward* the light, that cosine is the dot product `N·L`. When the surface faces away from the light the dot product goes negative; negative light is not a thing, so clamp it at zero.

Real rooms are never pitch black on the shadowed side, because light bounces. The 1999 answer to global illumination is a constant: *ambient*, a flat term added everywhere. The whole model is one line:

```
color = albedo * (ambient + sun_color * max(dot(N, L), 0))
```

where albedo is your texture sample. A *directional light* is a light so far away (the sun) that `L` is the same vector for every vertex in the scene — one uniform, no per-vertex work to find it. That is why it is the cheapest light there is and why we start with it.

## Where to compute it

Per-vertex means the vertex shader computes the lit color and the rasterizer interpolates the *result* across each triangle. On a densely tessellated mesh like your Day 11 terrain this looks smooth. On a big flat quad it looks wrong in ways we will fix tomorrow — today, the terrain is the star, and it already has per-vertex normals from central differences. Reuse them.

## Transforming normals

Positions go through the model matrix; normals must not. Two reasons. First, a normal is a direction — translation must not touch it, so you only apply the upper 3×3 of the model matrix (equivalently, multiply with `w = 0`). Second, and sneakier: under *non-uniform* scale the upper 3×3 bends normals so they are no longer perpendicular to the surface. Squash a sphere into a lens and the surface gets shallower, but the naively transformed normal tilts the *other* way. The mathematically correct transform for normals is the inverse-transpose of the upper 3×3.

We are not going to compute a 3×3 matrix inverse per object on an Atom. We dodge instead, with an engine rule you will thank yourself for: **lit meshes never get non-uniform scale.** For a pure rotation `R` the inverse-transpose is `R` itself (rotations are orthogonal: $R^{-1} = R^{T}$). For uniform scale `s·R`, the inverse-transpose is `(1/s)·R` — the same direction, different length, and the `normalize()` you were going to do anyway erases the difference. So under the rule, the normal matrix is just the upper-left 3×3 of the model matrix, copied out column by column.

## The shaders

```
static const char *LIT_VS =
"attribute vec3 a_position;\n"
"attribute vec3 a_normal;\n"
"attribute vec2 a_uv;\n"
"uniform mat4 u_mvp;\n"
"uniform mat3 u_nmat;\n"
"uniform vec3 u_sun_dir;\n"    /* unit, points TOWARD the sun */
"uniform vec3 u_sun_color;\n"
"uniform vec3 u_ambient;\n"
"varying vec2 v_uv;\n"
"varying vec3 v_light;\n"
"void main(void)\n"
"{\n"
"    vec3 n = normalize(u_nmat * a_normal);\n"
"    float ndotl = max(dot(n, u_sun_dir), 0.0);\n"
"    v_light = u_ambient + u_sun_color * ndotl;\n"
"    v_uv = a_uv;\n"
"    gl_Position = u_mvp * vec4(a_position, 1.0);\n"
"}\n";

static const char *LIT_FS =
"precision mediump float;\n"
"uniform sampler2D u_tex;\n"
"varying vec2 v_uv;\n"
"varying vec3 v_light;\n"
"void main(void)\n"
"{\n"
"    vec4 base = texture2D(u_tex, v_uv);\n"
"    gl_FragColor = vec4(base.rgb * v_light, base.a);\n"
"}\n";
```

Note the comment on `u_sun_dir`. The single most common lighting bug in history is feeding the direction the light *travels* instead of the direction *to* the light. They differ by a sign, and the symptom is a world lit from underground.

On the C side, extract the normal matrix and spin the sun:

```
void mat4_to_nmat(const float m[16], float out[9])
{
    out[0] = m[0]; out[1] = m[1]; out[2] = m[2];
    out[3] = m[4]; out[4] = m[5]; out[5] = m[6];
    out[6] = m[8]; out[7] = m[9]; out[8] = m[10];
}

void sun_from_angle(float deg, vec3_t out_dir)
{
    float r = deg * (3.14159265f / 180.0f);
    out_dir[0] = (float)cos(r);
    out_dir[1] = (float)sin(r);
    out_dir[2] = 0.35f;
    VectorNormalize(out_dir);
}
```

Upload with `glUniformMatrix3fv(loc, 1, GL_FALSE, nmat)` — in GLES2 the transpose argument *must* be `GL_FALSE`, the driver will reject anything else — and `glUniform3fv` for the vectors. Bind two keys to nudge the angle a few degrees per press. The plumbing (attribute binding for `a_normal`, uniform location caching) is yours; it is the same dance as Day 2.

**NOTE:** A word about gamma, so you hear it from me first. Your framebuffer is gamma-encoded: pixel value 128 is nowhere near half the photons of 255. Adding and scaling light in that space is, strictly, wrong — and it is exactly what every game did until roughly 2007. There is no sRGB framebuffer in this stack and there will be no gamma correction in this book. The practical consequence: light saturates to white faster than physics says it should, so when a scene blows out, halve your sun intensity rather than reaching for exotic fixes. This is the authentic 1999 look. Enjoy it guilt-free.

## Pitfalls

All black? Check the sun sign, then check that `a_normal` is actually bound (an unbound attribute reads constant garbage). Lighting subtly wrong only on scaled models? You forgot the `normalize()` after the normal matrix. Terrain showing a star-shaped shading pattern around peaks? Your Day 11 normals are per-vertex from central differences — that is as good as per-vertex gets; the rest is tomorrow's problem. The universal debug trick, worth engraving somewhere: `gl_FragColor = vec4(n * 0.5 + 0.5, 1.0);` renders normals as color. Up-facing ground should be green-ish (0.5, 1.0, 0.5). If it is not, no amount of staring at the dot product will help.

Performance: per-vertex lighting adds a handful of multiply-adds per vertex. On the SGX545 this is noise — you will not be able to measure it. That generosity is precisely why we spend it freely today and per-pixel carefully tomorrow.

**DONE WHEN:** the terrain's shading visibly follows the sun as you rotate it in real time with the keyboard: slopes facing the sun brighten, opposite slopes fall to ambient, and setting ambient to zero makes shadowed slopes go fully black.

**IF YOU ARE STUCK:** Everything black: flip the sign of the sun direction; make sure you normalized it in C, not just in the shader. Everything flat one brightness: `a_normal` not bound, or you passed the position buffer's offset for the normal attribute. Shading rotates with the model when it should not: you multiplied the normal by the view matrix too — light and normal must live in the same space, and here that space is world space. Weird bright triangles on scaled objects: missing `normalize()` after `u_nmat`.

# Day 14: Per-Pixel Lighting and Point Lights

Yesterday's lighting is computed at three corners and interpolated. For a directional sun on dense terrain, fine. But put a small bright light near a big flat wall and the lie collapses: the hotspot lands wherever a vertex happens to be, or vanishes entirely between them. The fix — evaluate the lighting equation at every *fragment* — was the bragging-rights feature of 2001-era hardware. Your SGX545 is a Shader Model 3.0 class part; it does this before breakfast, within reason. Today you move the math, then add the light type that actually needs it: the point light.

## What a varying really does

A `varying` is written per-vertex and arrives per-fragment, interpolated across the triangle (perspective-correctly — the GPU divides by depth so textures and varyings do not swim; you get this for free). Yesterday we interpolated the *answer*. Today we interpolate the *inputs* — the normal and the world position — and compute the answer per-fragment. One subtlety: the interpolation of two unit normals is not a unit normal. Halfway between two unit vectors 60 degrees apart lies a vector of length ~0.87 — the chord cuts inside the arc. Renormalize in the fragment shader or your lighting dims mysteriously in the middle of triangles.

## Point lights and attenuation

A point light has a position, not a direction; `L` must be computed per-fragment as `normalize(light_pos - frag_pos)`, which is exactly why the technique of the day exists. And real lights fade with distance. The classic recipe — the very formula behind fixed-function GL's `GL_CONSTANT/LINEAR/QUADRATIC_ATTENUATION` — is:

```
att = 1.0 / (k0 + k1 * d + k2 * d * d)
```

Keep `k0 = 1.0` always, so the denominator can never reach zero and the light cannot blow up to infinity at its own center. Then pick `k1, k2` for the reach you want. Working values, tuned in world units (1 unit = 1 meter-ish):

| Feel | k0 | k1 | k2 | Useful reach |
|---|---|---|---|---|
| Candle / tight | 1.0 | 0.7 | 1.8 | ~4 units |
| Torch / medium | 1.0 | 0.35 | 0.44 | ~8 units |
| Streetlamp / large | 1.0 | 0.14 | 0.07 | ~20 units |

## Loops in GLSL ES 1.00

You want two lights in a uniform array and a loop. Careful: GLSL ES 1.00 only guarantees loops whose bounds are compile-time constants and whose index is usable as a "constant-index-expression" for array subscripts — the compiler is allowed to reject anything fancier, especially in fragment shaders. `for (int i = 0; i < 2; ++i)` with `i` untouched in the body qualifies, and ANGLE unrolls it into straight-line HLSL anyway. Do not compute the bound from a uniform; if you want a variable light count, upload black for the unused lights.

## The shaders

```
static const char *PPL_VS =
"attribute vec3 a_position;\n"
"attribute vec3 a_normal;\n"
"attribute vec2 a_uv;\n"
```

```
"uniform mat4 u_mvp;\n"
"uniform mat4 u_model;\n"
"uniform mat3 u_nmat;\n"
"varying vec2 v_uv;\n"
"varying vec3 v_normal;\n"
"varying vec3 v_wpos;\n"
"void main(void)\n"
"{\n"
"    v_normal = u_nmat * a_normal;\n"
"    v_wpos = (u_model * vec4(a_position, 1.0)).xyz;\n"
"    v_uv = a_uv;\n"
"    gl_Position = u_mvp * vec4(a_position, 1.0);\n"
"}\n";

static const char *PPL_FS =
"precision mediump float;\n"
"uniform sampler2D u_tex;\n"
"uniform vec3 u_sun_dir;\n"
"uniform vec3 u_sun_color;\n"
"uniform vec3 u_ambient;\n"
"uniform vec3 u_lpos[2];\n"
"uniform vec3 u_lcolor[2];\n"
"uniform vec3 u_latten[2];\n"  /* x=k0, y=k1, z=k2 */
"varying vec2 v_uv;\n"
"varying vec3 v_normal;\n"
"varying vec3 v_wpos;\n"
"void main(void)\n"
"{\n"
"    vec3 n = normalize(v_normal);\n"
"    vec3 light = u_ambient\n"
"        + u_sun_color * max(dot(n, u_sun_dir), 0.0);\n"
"    for (int i = 0; i < 2; ++i)\n"
"    {\n"
"        vec3 dv = u_lpos[i] - v_wpos;\n"
"        float d = length(dv);\n"
"        float att = 1.0 / (u_latten[i].x\n"
"            + u_latten[i].y * d + u_latten[i].z * d * d);\n"
"        light += u_lcolor[i]\n"
"            * max(dot(n, dv / d), 0.0) * att;\n"
"    }\n"
"    vec4 base = texture2D(u_tex, v_uv);\n"
"    gl_FragColor = vec4(base.rgb * light, base.a);\n"
"}\n";
```

Feeding the arrays from C: query the location of the *first element* by name — `glGetUniformLocation(prog, "u_lpos[0]")` — then upload both at once with `glUniform3fv(loc, 2, (const float *)positions)`, six floats tightly packed. Querying the bare name `"u_lpos"` is supposed to work too, but era drivers were sloppy about it; the `[0]` form always works. The orbit itself is Day 8 material: park a small unlit marker cube at each light position so you can see where the light thinks it is — debugging an invisible object is a mug's game.

## Optional: Blinn specular

Shiny highlights come from the Blinn half-vector: compute the view direction `V = normalize(u_eye_pos - v_wpos)`, then `H = normalize(L + V)`, and add `lcolor * pow(max(dot(n, H), 0.0), 32.0) * att`. The exponent (shininess) sharpens the highlight; 8 is plastic, 32 is lacquer, 128 is chrome. Yes, that means a `normalize` and a `pow` per light per fragment.

**WARNING:** `pow` per fragment is where this GPU starts sending you invoices. You have 614,400 pixels at 1024×600 and overdraw multiplies every one of them. Add specular, then check the fps counter *before and after, looking at the same scene*. If it pushed you over a vsync boundary (60 to 30 — remember, missed frames quantize on XP), your options are: specular from the sun only, specular per-vertex, or no specular. Measure. Never assume.

One precision note: the language forces you to declare `precision mediump float;` in every fragment shader, and you should write honest precision qualifiers — but ANGLE compiles to D3D9 HLSL, which runs everything at fp32, so on this machine mediump neither saves nor costs anything. Write it correctly anyway; this code may outlive this netbook.

## Pitfalls

Light center darker than its surroundings: `k0 = 0`, and you divided by a very small `d`. Lighting pops when the model moves: you uploaded light positions in world space but forgot `u_model` on the fragment position (or vice versa) — everything in the equation must live in one space. Faceted shading survived the move to per-pixel: normalizing `v_normal` in the *vertex* shader does nothing; the interpolation happens after. Sun looks different than yesterday: it should not — same equation, finer sampling grid; if it visibly changed, one of the two shaders has a sign or space bug.

**DONE WHEN:** a point light orbits the Day 7 OBJ model and the pool of light it casts on nearby geometry moves with it, brightens near it, and fades smoothly by distance — and the falloff edge is round, not a polygonal blob snapping from vertex to vertex.

**IF YOU ARE STUCK:** Blob is polygonal: you are still computing light in the vertex shader somewhere — check which program is actually bound. Both lights identical or second one dead: array upload count wrong; verify with `glUniform3fv(loc, 2, ...)` and the `"u_lpos[0]"` location form. Highlights crawl when the camera strafes: `u_eye_pos` is stale; upload it every frame from the camera position. Shader fails to compile only on the loop: your driver is stricter than ANGLE should be — unroll the two iterations by hand and move on with your life.

# Day 15: DOT3 Normal Mapping

Per-pixel lighting evaluates the equation at every fragment, but the normal it uses still comes from three vertices. A flat quad has one normal everywhere; no amount of per-pixel math will put a rivet on it. The 2001 flagship trick — "DOT3 bump mapping" on the GeForce 3 and in every tech demo of the era — is to store a normal *per texel* in a texture and light against that. Suddenly a 2-triangle wall has ten thousand tiny slopes. Today you build the machinery: tangent space, the TBN matrix, and the normal map itself, painted on this very netbook in Paint Shop Pro 7.

## Why tangent space

A normal map stores normals for a surface that could be wrapped onto anything — a wall, a barrel, a cave ceiling. So the map's normals cannot be in world space; they are in the surface's own frame: **tangent space**, where +Z sticks out of the surface, +X points along the texture's U axis, and +Y along V. A flat, unbumped texel is (0, 0, 1). Stored in a texture as `n * 0.5 + 0.5`, that is RGB (128, 128, 255) — the famous lavender. To light with these normals, both parties must be in the same space, and the cheap 1999-brained choice is: drag the *light vector* into tangent space in the vertex shader (three dot products, once per vertex) instead of dragging every normal-map sample out to world space per fragment.

## Deriving the tangent

You need, at each vertex, the world-space direction that U increases in (the tangent `T`) and the one V increases in (the bitangent `B`). Take one triangle with corners P0, P1, P2 and their UVs. The two edge vectors, expressed in the surface frame, are made of steps along T and B:

```
E1 = P1 - P0 = du1 * T + dv1 * B
E2 = P2 - P0 = du2 * T + dv2 * B

  where du1 = u1 - u0, dv1 = v1 - v0, etc.
```

Two equations, two vector unknowns — a 2×2 system. Invert it the schoolbook way:

```
det = du1 * dv2 - du2 * dv1
T = (dv2 * E1 - dv1 * E2) / det
B = (du1 * E2 - du2 * E1) / det
```

That is the per-face tangent. A vertex shared by several faces gets the *sum* of its faces' tangents (unnormalized sums weight larger faces more, which is what you want), then we make the frame orthonormal: subtract from `T` its component along the vertex normal (Gram-Schmidt), normalize, and rebuild `B = cross(N, T)` in the shader rather than storing it.

```
typedef struct vertex_s {
    vec3_t pos;
    vec3_t normal;
    float uv[2];
    vec3_t tangent;
} vertex_t;

void mesh_build_tangents(vertex_t *verts, int nverts,
                         const unsigned short *idx, int nidx)
{
    int i;

    for (i = 0; i < nverts; ++i) {
        verts[i].tangent[0] = 0.0f;
```

```
        verts[i].tangent[1] = 0.0f;
        verts[i].tangent[2] = 0.0f;
    }
    for (i = 0; i < nidx; i += 3) {
        vertex_t *v0 = &verts[idx[i + 0]];
        vertex_t *v1 = &verts[idx[i + 1]];
        vertex_t *v2 = &verts[idx[i + 2]];
        vec3_t e1, e2, t;
        float du1, dv1, du2, dv2, det, r;

        VectorSubtract(v1->pos, v0->pos, e1);
        VectorSubtract(v2->pos, v0->pos, e2);
        du1 = v1->uv[0] - v0->uv[0];
        dv1 = v1->uv[1] - v0->uv[1];
        du2 = v2->uv[0] - v0->uv[0];
        dv2 = v2->uv[1] - v0->uv[1];
        det = du1 * dv2 - du2 * dv1;
        if (det == 0.0f)
            continue;               /* degenerate UVs, skip face */
        r = 1.0f / det;
        t[0] = r * (dv2 * e1[0] - dv1 * e2[0]);
        t[1] = r * (dv2 * e1[1] - dv1 * e2[1]);
        t[2] = r * (dv2 * e1[2] - dv1 * e2[2]);
        VectorAdd(v0->tangent, t, v0->tangent);
        VectorAdd(v1->tangent, t, v1->tangent);
        VectorAdd(v2->tangent, t, v2->tangent);
    }
    for (i = 0; i < nverts; ++i) {
        float *n = verts[i].normal;
        float *t = verts[i].tangent;
        vec3_t proj;
        float d;

        d = DotProduct(n, t);
        VectorScale(n, d, proj);
        VectorSubtract(t, proj, t);     /* Gram-Schmidt */
        if (VectorNormalize(t) == 0.0f) {
            vec3_t axis;
            axis[0] = 1.0f; axis[1] = 0.0f; axis[2] = 0.0f;
            if (n[0] > 0.9f || n[0] < -0.9f) {
                axis[0] = 0.0f; axis[1] = 1.0f;
            }
            CrossProduct(n, axis, t);
            VectorNormalize(t);
        }
    }
}
```

Hook this into the Day 7 OBJ loader after the v/vt/vn dedup, add `a_tangent` to the interleaved vertex format, and bind it like any other attribute. One restriction to adopt now: no mirrored UVs on normal-mapped meshes. Mirroring flips the handedness of the (T, B, N) frame, and handling it properly means storing a sign per vertex in a fourth tangent component. Quake-era answer: don't mirror. Ours too.

## The shaders

```
static const char *DOT3_VS =
"attribute vec3 a_position;\n"
"attribute vec3 a_normal;\n"
"attribute vec3 a_tangent;\n"
"attribute vec2 a_uv;\n"
"uniform mat4 u_mvp;\n"
"uniform mat4 u_model;\n"
```

```
"uniform mat3 u_nmat;\n"
"uniform vec3 u_lpos;\n"       /* point light, world space */
"varying vec2 v_uv;\n"
"varying vec3 v_light_ts;\n"
"void main(void)\n"
"{\n"
"    vec3 n = normalize(u_nmat * a_normal);\n"
"    vec3 t = normalize(u_nmat * a_tangent);\n"
"    vec3 b = cross(n, t);\n"
"    vec3 wpos = (u_model * vec4(a_position, 1.0)).xyz;\n"
"    vec3 l = u_lpos - wpos;\n"
"    v_light_ts = vec3(dot(l, t), dot(l, b), dot(l, n));\n"
"    v_uv = a_uv;\n"
"    gl_Position = u_mvp * vec4(a_position, 1.0);\n"
"}\n";

static const char *DOT3_FS =
"precision mediump float;\n"
"uniform sampler2D u_tex;\n"
"uniform sampler2D u_nmap;\n"
"uniform vec3 u_lcolor;\n"
"uniform vec3 u_ambient;\n"
"varying vec2 v_uv;\n"
"varying vec3 v_light_ts;\n"
"void main(void)\n"
"{\n"
"    vec3 n = texture2D(u_nmap, v_uv).rgb * 2.0 - 1.0;\n"
"    vec3 l = normalize(v_light_ts);\n"
"    float ndotl = max(dot(normalize(n), l), 0.0);\n"
"    vec3 base = texture2D(u_tex, v_uv).rgb;\n"
"    gl_FragColor =\n"
"        vec4(base * (u_ambient + u_lcolor * ndotl), 1.0);\n"
"}\n";
```

The three dot products build the light vector in tangent space (multiplying by the transpose of the TBN matrix, which for an orthonormal frame is its inverse). Attenuation is omitted for clarity — splice in yesterday's formula using `length(v_light_ts)`, which interpolation preserves well enough at these light ranges. Note we normalize *after* interpolation, in the fragment shader, for the same chord-inside-the-arc reason as yesterday.

## Authoring the map in Paint Shop Pro 7

There is no normal-map plugin in the cabin, but PSP7 has image arithmetic, and a normal map is just two derivatives wearing a trench coat. Procedure, from a grayscale **height map** (paint one: white = raised, black = recessed; blur it a little — single-pixel steps make one-texel-wide normals):

1. Duplicate the height image twice. In copy A, select all, float the selection, and nudge it one pixel *left* (arrow key). In copy B, nudge one pixel *right*. 2. Image arithmetic: subtract B from A with a bias of 128. The result is the horizontal slope map: mid-gray where flat, bright where the surface rises to the right. This becomes your **red** channel. 3. Repeat with up/down nudges for the vertical slope map: the **green** channel. 4. Combine channels (split/combine RGB): R = horizontal slopes, G = vertical slopes, B = pure white (255). 5. Increase or decrease contrast around 128 on R and G to taste — that is your bump strength knob.

The result is not normalized — the blue channel is a lie — but look at the fragment shader: `normalize(n)` fixes every texel for free at runtime. That is the whole "emboss-then-normalize" method, and it is exactly how artists faked it before dedicated tools. Hand-painting works too: start from flat lavender (128, 128, 255) and push R/G around edges you want raised. An alternative worth an evening: an emboss filter with the light from the right gives you the R channel directly, from above gives you G.

**TIP:** The purple-flat-map sanity test. Before admiring any bumps, feed the shader a solid (128, 128, 255) map. That decodes to n = (0, 0, 1) = the geometric normal, so the render must look identical to Day 14's per-pixel lighting. If it does not, your TBN is broken and every bumpy screenshot you take before fixing it is a coincidence.

## Pitfalls

Bumps look correct from one light direction and inverted from another: the green channel is flipped. Whether +V in UV space runs up or down your image depends on your TGA origin (Day 4 flashbacks) — negate G in PSP7 and the pits become bumps. Lighting slides around when the model rotates: you forgot `u_nmat` on the tangent, or built B in a different handedness than the derivation. Faint grid-like shading on smooth meshes: tangents averaged across a UV seam; vertices on seams were (correctly) duplicated by the Day 7 dedup, so this usually means your dedup merged them — check it keys on all three indices. Performance: two texture samples and a normalize per fragment is comfortably within budget on a wall or a model; do not normal-map the terrain today — that fill bill is a different conversation.

**DONE WHEN:** a flat two-triangle quad with a brick or rivet normal map reads as embossed from every angle as the point light orbits it — highlights crawl along the raised edges facing the light, and the purple-flat-map test renders identically to Day 14.

**IF YOU ARE STUCK:** Bumps invert with light angle on one axis only: flip the normal map's green channel. Everything looks flat: you forgot `* 2.0 - 1.0`, so every normal points into the +XYZ octant — the classic. Random sparkle: uninitialized tangents; make sure the accumulate pass zeroed them first. Debug by rendering `v_light_ts` as color: over a flat quad facing +Z with the light straight above the quad center, it should shade smoothly toward blue directly under the light.

# Day 16: Lightmaps

Everything so far is dynamic lighting: computed live, every frame, and therefore restricted to what fits in a frame's budget. Quake's great 1996 insight ran the other way: indoor light barely changes, so compute it *offline*, as long as you like, and store the result in a texture. Soft corners, gentle gradients, light that appears to bounce — all for the runtime cost of one extra texture fetch. That look defined id's engines for a decade. Today you write a small offline baker (a separate console exe, like your Day 1 test runner), give your room geometry a second UV set, and multitexture the result onto the walls.

## The second UV set

Base texture UVs tile and repeat — ten walls can share one brick texture. Lightmap UVs must be unique: every surface owns its own patch of lightmap, because every surface is lit differently. That is the entire reason multitexturing hardware existed in 1998. For our axis-aligned room, uniqueness is easy: each face (floor, each wall, ceiling, crate tops) gets its own little lightmap image, planar-projected — lightmap UV = fractional position along the face's two edges. Resolution: Quake used one lightmap texel ("luxel") per 16 world units; for us, one luxel per 0.5 m reads right. A 6×4 m wall gets a 12×8 lightmap. Yes, that small. Bilinear magnification is the whole aesthetic.

## The baker

Describe the room to the baker as a list of AABBs (walls with thickness, floor, crates) plus a list of faces to bake: origin corner, two edge vectors, normal, luxel dimensions. For each luxel: find its world position at the texel center, push it slightly off the surface, and fire N rays over the hemisphere above the face. The fraction that escape without hitting a box is the **sky visibility** — open floor is bright, corners are dark, under-the-crate is black. (The industry later named this ambient occlusion and pretended it was new.) Sampling with cosine-weighted directions makes the plain hit fraction equal the diffuse response to an evenly bright sky, which is exactly the soft look we want. Ray vs AABB is the classic slab test:

```
typedef struct box_s { vec3_t bmin; vec3_t bmax; } box_t;

typedef struct face_s {
    vec3_t origin;      /* luxel (0,0) corner */
    vec3_t edge_u;      /* full extent along lightmap U */
    vec3_t edge_v;      /* full extent along lightmap V */
    vec3_t normal;
    int w, h;           /* luxels */
} face_t;

static float frand(unsigned *s)
{
    *s = *s * 1664525u + 1013904223u;
    return (float)(*s >> 8) / 16777216.0f;
}

static int ray_hits_box(const vec3_t org, const vec3_t dir,
                        const box_t *b, float tmax)
{
    float tmin = 0.0f;
    int i;

    for (i = 0; i < 3; ++i) {
        if (dir[i] > -1e-6f && dir[i] < 1e-6f) {
            if (org[i] < b->bmin[i] || org[i] > b->bmax[i])
                return 0;
```

```
        } else {
            float inv = 1.0f / dir[i];
            float t0 = (b->bmin[i] - org[i]) * inv;
            float t1 = (b->bmax[i] - org[i]) * inv;
            float tmp;
            if (t0 > t1) { tmp = t0; t0 = t1; t1 = tmp; }
            if (t0 > tmin) tmin = t0;
            if (t1 < tmax) tmax = t1;
            if (tmin > tmax)
                return 0;
        }
    }
    return 1;
}
```

And the luxel loop, an 8×8 jittered grid of rays per luxel (64 rays; raise to 16×16 for the final bake and go make coffee — it is offline, that is the point):

```
#define RDIM 8
#define RAY_LEN 64.0f

void bake_face(const face_t *fc, const box_t *boxes, int nboxes,
               unsigned char *out)
{
    vec3_t tan, bit;
    unsigned seed = 12345u;
    int s, t;

    tan[0] = fc->edge_u[0]; tan[1] = fc->edge_u[1];
    tan[2] = fc->edge_u[2];
    VectorNormalize(tan);
    bit[0] = fc->edge_v[0]; bit[1] = fc->edge_v[1];
    bit[2] = fc->edge_v[2];
    VectorNormalize(bit);

    for (t = 0; t < fc->h; ++t) {
        for (s = 0; s < fc->w; ++s) {
            vec3_t pos;
            int i, j, open = 0;
            float fu = ((float)s + 0.5f) / (float)fc->w;
            float fv = ((float)t + 0.5f) / (float)fc->h;

            pos[0] = fc->origin[0] + fc->edge_u[0] * fu
                + fc->edge_v[0] * fv + fc->normal[0] * 0.05f;
            pos[1] = fc->origin[1] + fc->edge_u[1] * fu
                + fc->edge_v[1] * fv + fc->normal[1] * 0.05f;
            pos[2] = fc->origin[2] + fc->edge_u[2] * fu
                + fc->edge_v[2] * fv + fc->normal[2] * 0.05f;

            for (j = 0; j < RDIM; ++j)
            for (i = 0; i < RDIM; ++i) {
                vec3_t d;
                int k, hit = 0;
                float u = ((float)i + frand(&seed)) / (float)RDIM;
                float v = ((float)j + frand(&seed)) / (float)RDIM;
                float phi = 6.2831853f * u;
                float ct = (float)sqrt(1.0f - v); /* cos-weighted */
                float st = (float)sqrt(v);
                float x = st * (float)cos(phi);
                float y = st * (float)sin(phi);

                d[0] = tan[0]*x + bit[0]*y + fc->normal[0]*ct;
                d[1] = tan[1]*x + bit[1]*y + fc->normal[1]*ct;
                d[2] = tan[2]*x + bit[2]*y + fc->normal[2]*ct;
```

```
                for (k = 0; k < nboxes; ++k)
                    if (ray_hits_box(pos, d, &boxes[k], RAY_LEN)) {
                        hit = 1;
                        break;
                    }
                if (!hit)
                    ++open;
            }
            /* half-scale: 128 = fully open sky (see overbright) */
            out[t * fc->w + s] =
                (unsigned char)(open * 128 / (RDIM * RDIM));
        }
    }
}
```

Write each face's buffer as an 8-bit grayscale TGA (type 3): 18-byte header, `hdr[2] = 3`, width/height little-endian in bytes 12–15, `hdr[16] = 8` bits, `hdr[17] = 0x20` for top-left origin, then the raw pixels. Your Day 4 loader already reads these back (stb_image also handles them, expanding gray to RGB). Leave the face's own wall box in the occluder list, by the way: grazing rays that nick it darken edges and pits, and where two walls meet you get exactly the soft dark corner you came for.

## The shader side: modulate with overbright

Bind the base texture to unit 0 and the lightmap to unit 1 (`glActiveTexture(GL_TEXTURE1)`, bind, and set the sampler uniform to the integer 1), add a second UV attribute, and combine:

```
"    vec3 base = texture2D(u_tex, v_uv).rgb;\n"
"    vec3 lm = texture2D(u_lmap, v_uv2).rgb;\n"
"    gl_FragColor = vec4(base * lm * 2.0, 1.0);\n"
```

The `* 2.0` is Quake's overbright trick, and it is why the baker wrote 128 for "fully lit": lightmaps are stored at half scale so a value of 255 can push a surface to *twice* its texture brightness. Without headroom, a lightmap can only ever darken, and rooms look like someone taxed the electricity. Layer the dynamic point lights from Day 14 into the same shader — add their contribution to `lm * 2.0` before the multiply by `base` — and a torch carried through a baked room reads as one lighting model, which is the era-correct blend: static world, dynamic accents.

## Pitfalls

The one everyone hits: **bilinear seams**. If you later pack many face lightmaps into one atlas texture (and at 12×8 a piece, you will want to), bilinear filtering at a chart's edge reads the neighboring chart's texels and draws a wrong-colored line down your wall. Pad every chart with a 1-texel border that duplicates its edge row/column, and inset the UVs by half a texel so samples never leave the padded region. With one texture per face, set `GL_CLAMP_TO_EDGE` and do the half-texel UV inset, same reason. Also: luxels behind another box (a wall face buried in a crate) bake black and bleed darkness via bilinear — either nudge such faces' luxel centers or accept the grime as atmosphere. Filtering: bilinear, no mipmaps — the texture is already tiny.

**DONE WHEN:** the room scene shows soft darkened corners and contact shading under the crates, from lightmaps baked by your own tool exe, with a Day 14 dynamic light layered on top — and there are no visible seam lines at chart boundaries.

**IF YOU ARE STUCK:** Whole room mid-gray: your rays start inside the geometry — increase the 0.05 push-off, or your wall boxes overlap the face. Whole room white: ray length too short to reach anything; check RAY_LEN against the room size. Lightmap obviously misaligned: your second UV set and the baker disagree on which corner is luxel (0,0) — bake a test map with a black corner texel and see where it lands. Speckle in the bake: that is variance; more rays (16×16) or a final 3×3 blur pass over the TGA. Seams: reread the padding paragraph, slowly.

# Day 17: Fog and Time of Day

Fog in 1999 did two jobs. The honest one: air is not transparent, and distant hills really do fade toward the sky color. The load-bearing one: fog hid the far clip plane, and entire console generations survived on that alibi. Fixed-function GL shipped three fog modes — LINEAR, EXP, EXP2 — and EXP2 was the pretty one: it leaves the foreground almost untouched, then rolls off smoothly. Today you recreate `GL_FOG` faithfully in your shaders, then make the whole atmosphere breathe: a sun that arcs across the sky while sun, ambient, fog, and sky colors slide through a day cycle.

## Exp2 fog

The fog factor for a fragment at eye distance `d` with density `g` is:

```
f = exp(-(g * d) * (g * d))     clamped to [0, 1]
final = mix(fog_color, lit_color, f)
```

Note the convention, straight from the GL spec: `f = 1` means *no* fog (near), `f = 0` means fully fogged — hence lit color takes the third slot of `mix`. Get this backwards and your world is foggy at your feet and clear at the horizon, which is at least memorable.

Compute `f` in the *vertex* shader — fog varies slowly and per-vertex is visually identical at terrain tessellation, for free. Distance comes from view space: transform the position by modelview and take `length()`. That is radial (spherical) distance, which is what "eye distance" meant in the spec; the cheaper `-vpos.z` (planar) works but makes fog visibly swim as you turn the camera, because a hill at the screen edge is farther radially than axially. Pay the `sqrt`; it is per-vertex.

```
/* vertex shader additions */
"uniform mat4 u_modelview;\n"
"uniform float u_fog_density;\n"
"varying float v_fog;\n"
    ...
"    vec3 vpos = (u_modelview * vec4(a_position, 1.0)).xyz;\n"
"    float fd = length(vpos) * u_fog_density;\n"
"    v_fog = clamp(exp(-fd * fd), 0.0, 1.0);\n"

/* fragment shader, last line before writing gl_FragColor */
"uniform vec3 u_fog_color;\n"
"varying float v_fog;\n"
    ...
"    vec3 fogged = mix(u_fog_color, lit.rgb, v_fog);\n"
"    gl_FragColor = vec4(fogged, lit.a);\n"
```

## The day cycle

Let `t` be time of day in [0, 1): 0.0 midnight, 0.25 dawn, 0.5 noon, 0.75 dusk. The sun direction is a circle tilted slightly off the east-west plane so noon light is not perfectly vertical:

```
ang = (t - 0.25f) * 2.0f * 3.14159265f;
dir[0] = (float)cos(ang);       /* rises in +X (east) */
dir[1] = (float)sin(ang);       /* below horizon at night */
dir[2] = 0.30f;
VectorNormalize(dir);
```

At night `dir[1]` goes negative, the terrain's `max(N·L, 0)` kills the direct term on upward faces, and the world runs on ambient — provided your night sun color is black, which it is about to be. Every atmosphere color is a keyframe table, linearly interpolated:

| t | Phase | Sun RGB | Ambient RGB | Fog RGB | Sky tint RGB | Fog density |
|---|---|---|---|---|---|---|
| 0.00 | Night | 0, 0, 0 | 14, 18, 36 | 8, 10, 22 | 40, 46, 82 | 0.020 |
| 0.25 | Dawn | 255, 150, 88 | 64, 56, 72 | 168, 126, 118 | 232, 170, 140 | 0.014 |
| 0.50 | Noon | 255, 244, 214 | 96, 104, 122 | 176, 196, 214 | 255, 255, 255 | 0.005 |
| 0.75 | Dusk | 255, 118, 52 | 72, 56, 66 | 150, 104, 92 | 234, 150, 110 | 0.012 |
| 1.00 | Night | 0, 0, 0 | 14, 18, 36 | 8, 10, 22 | 40, 46, 82 | 0.020 |

(Divide by 255 for the shader.) Notice fog is thickest at night and thinnest at noon, and dawn fog is warm — that single choice sells the sunrise. The evaluator:

```
typedef struct daykey_s {
    float t;
    vec3_t sun, ambient, fog, sky;
    float density;
} daykey_t;

static const daykey_t g_keys[] = {
    { 0.00f, {0.00f,0.00f,0.00f}, {0.05f,0.07f,0.14f},
             {0.03f,0.04f,0.09f}, {0.16f,0.18f,0.32f}, 0.020f },
    { 0.25f, {1.00f,0.59f,0.35f}, {0.25f,0.22f,0.28f},
             {0.66f,0.49f,0.46f}, {0.91f,0.67f,0.55f}, 0.014f },
    { 0.50f, {1.00f,0.96f,0.84f}, {0.38f,0.41f,0.48f},
             {0.69f,0.77f,0.84f}, {1.00f,1.00f,1.00f}, 0.005f },
    { 0.75f, {1.00f,0.46f,0.20f}, {0.28f,0.22f,0.26f},
             {0.59f,0.41f,0.36f}, {0.92f,0.59f,0.43f}, 0.012f },
    { 1.00f, {0.00f,0.00f,0.00f}, {0.05f,0.07f,0.14f},
             {0.03f,0.04f,0.09f}, {0.16f,0.18f,0.32f}, 0.020f }
};

static void lerp3(const vec3_t a, const vec3_t b, float f,
                  vec3_t out)
{
    out[0] = a[0] + (b[0] - a[0]) * f;
    out[1] = a[1] + (b[1] - a[1]) * f;
    out[2] = a[2] + (b[2] - a[2]) * f;
}

void daycycle_eval(float t, daykey_t *out)
{
    int i;
    int n = (int)(sizeof(g_keys) / sizeof(g_keys[0]));
    const daykey_t *a;
    const daykey_t *b;
    float f;

    t -= (float)floor(t);           /* wrap into [0,1) */
    for (i = 0; i < n - 2; ++i)
        if (t <= g_keys[i + 1].t)
            break;
    a = &g_keys[i];
    b = &g_keys[i + 1];
    f = (t - a->t) / (b->t - a->t);
    lerp3(a->sun, b->sun, f, out->sun);
    lerp3(a->ambient, b->ambient, f, out->ambient);
    lerp3(a->fog, b->fog, f, out->fog);
    lerp3(a->sky, b->sky, f, out->sky);
```

```
    out->density = a->density + (b->density - a->density) * f;
}
```

Every frame: advance `t` (a full day in 120 seconds is a good demo speed), call `daycycle_eval`, feed `u_sun_dir`, `u_sun_color`, `u_ambient`, `u_fog_color`, `u_fog_density`. And one more: give the Day 10 skybox shader a `uniform vec3 u_sky_tint` and multiply the cubemap sample by it, fed from the same table. A daylight cubemap tinted deep blue at night and orange at dawn is a cheat, and on this screen it is a cheat that looks like weather. If the transitions feel mushy, add extra keyframes at 0.20 and 0.30 to sharpen the dawn — the table format does not care.

## Pitfalls

The skybox and fog will fight. The box sits at infinity but gets no fog (do not fog it — a fogged skybox at density 0.02 is a gray void), so at the horizon, fogged terrain meets unfogged sky. The fix is in the table above: fog color and horizon sky tint stay close to each other, so terrain fades *into* the sky. If you see a hard silhouette line at the horizon, your fog color has drifted from your sky color. Second classic: uploading density in the wrong units — density pairs with your world scale; if your terrain is 500 units across, density 0.005 at noon means visibility comfortably past 200 units. Third: `exp()` per vertex is cheap, but remember terrain chunks from Day 12 are culled — a fogged-out chunk that is still inside the frustum still costs full vertex work. (Filing that thought away is Day 27's job.)

**DONE WHEN:** you can sit and watch dawn come up over the terrain: the sun rises in the east, warm light rakes the slopes, fog thins as the ambient lifts, the sky tint warms — all from one `t` knob. Screenshot it. That is the desktop wallpaper now, per the challenge, and no jury would convict.

**IF YOU ARE STUCK:** Fog backwards (clear far, foggy near): swap the mix arguments; f=1 is near. Everything fogs to gray even at noon: density units vs world scale; print `g*d` for a far vertex — it should be well under 1.0 at noon. Sun pops at midnight: your keyframe search walked off the table; the final 1.00 row must duplicate the 0.00 row. Sky and ground never match at dusk: tune fog RGB toward the sky-tint RGB in the same row; they must travel together.

# Day 18: Particles

Fire, smoke, sparks, fountains, muzzle flashes: in 1999 every one of them was the same object — a few hundred textured quads, born, flung, faded, and killed by a little CPU simulation, drawn with additive blending so they glow where they overlap. No effect in this book has a better ratio of screen impact to code. Today you build the pool, the emitters, the camera-facing billboards, and the one dynamic vertex buffer that carries them all — and then you find the number where this Atom taps out, because that number is now a line in your journal.

## Why not point sprites

GLES2 has `gl_PointSize` point sprites, and through ANGLE they ride D3D9 point sprites: maximum size capped by the driver (often 64 pixels), no rotation, and — the killer — D3D9 clips a point the moment its *center* leaves the frustum, so big particles pop out of existence at the screen edge. Era engines built quads on the CPU instead, and so will we: four vertices per particle, aimed at the camera every frame.

## The pool and the update

```
#define MAX_PARTICLES 4096

typedef struct particle_s {
    vec3_t pos;
    vec3_t vel;
    float life, max_life;       /* seconds */
    float size;                 /* world units, quad edge */
    unsigned char color[4];
} particle_t;

typedef struct emitter_s {
    vec3_t origin;
    vec3_t gravity;
    float rate;                 /* particles per second */
    float spawn_acc;
    /* per-emitter spawn params: base vel, spread, life, etc. */
} emitter_t;

static particle_t g_parts[MAX_PARTICLES];
static int g_num_parts;

void particles_update(emitter_t *em, float dt)
{
    vec3_t step;
    int i;

    em->spawn_acc += em->rate * dt;
    while (em->spawn_acc >= 1.0f &&
           g_num_parts < MAX_PARTICLES) {
        em->spawn_acc -= 1.0f;
        emitter_spawn_one(em, &g_parts[g_num_parts]);
        ++g_num_parts;
    }
    i = 0;
    while (i < g_num_parts) {
        particle_t *p = &g_parts[i];
        p->life -= dt;
        if (p->life <= 0.0f) {
            --g_num_parts;              /* swap-remove */
            g_parts[i] = g_parts[g_num_parts];
```

```
            continue;                   /* re-test slot i */
        }
        VectorScale(em->gravity, dt, step);
        VectorAdd(p->vel, step, p->vel);
        VectorScale(p->vel, dt, step);
        VectorAdd(p->pos, step, p->pos);
        ++i;
    }
}
```

Run this from `engine_update`, inside the fixed timestep, so particle physics is deterministic and framerate-independent like everything else. Spawning goes through an accumulator so a rate of 300/s emits five particles per 60 Hz tick without drift. Swap-remove keeps the pool dense with no free lists and no allocation, ever. `emitter_spawn_one` is yours: set position at the origin, velocity from the emitter's base velocity plus random spread, life and size from the tables below. Fade by writing alpha from `life / max_life` either here or at build time.

## Billboards from the view matrix

A billboard quad needs the camera's right and up vectors in world space. You already have them. The view matrix rotates world into eye space; the rotation part of that matrix has the camera's basis vectors as its *rows* (a rotation's inverse is its transpose, so the rows of world-to-eye are the columns of eye-to-world). In our column-major storage, `m[c*4+r]`, row 0 and row 1 are:

```
right = ( view[0], view[4], view[8]  )   /* row 0 */
up    = ( view[1], view[5], view[9]  )   /* row 1 */
```

```
typedef struct pvert_s {
    float pos[3];
    float uv[2];
    unsigned char color[4];
} pvert_t;                      /* 24 bytes */

static pvert_t g_pverts[MAX_PARTICLES * 4];

int particles_build(const float view[16])
{
    vec3_t right, up;
    int i;

    right[0] = view[0]; right[1] = view[4]; right[2] = view[8];
    up[0] = view[1]; up[1] = view[5]; up[2] = view[9];

    for (i = 0; i < g_num_parts; ++i) {
        const particle_t *p = &g_parts[i];
        pvert_t *v = &g_pverts[i * 4];
        vec3_t r, u;
        int c;

        VectorScale(right, p->size * 0.5f, r);
        VectorScale(up, p->size * 0.5f, u);
        v[0].pos[0] = p->pos[0] - r[0] - u[0];
        v[0].pos[1] = p->pos[1] - r[1] - u[1];
        v[0].pos[2] = p->pos[2] - r[2] - u[2];
        v[1].pos[0] = p->pos[0] + r[0] - u[0];
        v[1].pos[1] = p->pos[1] + r[1] - u[1];
        v[1].pos[2] = p->pos[2] + r[2] - u[2];
        v[2].pos[0] = p->pos[0] + r[0] + u[0];
        v[2].pos[1] = p->pos[1] + r[1] + u[1];
        v[2].pos[2] = p->pos[2] + r[2] + u[2];
        v[3].pos[0] = p->pos[0] - r[0] + u[0];
```

```
        v[3].pos[1] = p->pos[1] - r[1] + u[1];
        v[3].pos[2] = p->pos[2] - r[2] + u[2];
        v[0].uv[0] = 0.0f; v[0].uv[1] = 0.0f;
        v[1].uv[0] = 1.0f; v[1].uv[1] = 0.0f;
        v[2].uv[0] = 1.0f; v[2].uv[1] = 1.0f;
        v[3].uv[0] = 0.0f; v[3].uv[1] = 1.0f;
        for (c = 0; c < 4; ++c) {
            v[c].color[0] = p->color[0];
            v[c].color[1] = p->color[1];
            v[c].color[2] = p->color[2];
            v[c].color[3] = p->color[3];
        }
    }
    return g_num_parts;
}
```

Indices never change: quad `q` uses vertices `4q+0, 4q+1, 4q+2, 4q+0, 4q+2, 4q+3`. Build that `unsigned short` array once at init into a static index buffer. The vertex buffer is the dynamic one:

```
    glBindBuffer(GL_ARRAY_BUFFER, g_pvbo);
    glBufferData(GL_ARRAY_BUFFER,
                 (long)count * 4 * (long)sizeof(pvert_t),
                 g_pverts, GL_DYNAMIC_DRAW);

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);   /* additive */
    glDepthMask(GL_FALSE);               /* test yes, write no */
    glDrawElements(GL_TRIANGLES, count * 6,
                   GL_UNSIGNED_SHORT, 0);
    glDepthMask(GL_TRUE);
    glDisable(GL_BLEND);
```

The full-size `glBufferData` every frame is deliberate: respecifying the whole buffer lets the driver hand you fresh storage instead of stalling until the GPU finishes reading last frame's data (ANGLE turns this into a D3D9 `DISCARD` lock). Do not "optimize" it into `glBufferSubData`; that is how you invent a pipeline stall. Draw order: after all opaque geometry and the skybox, before the Day 6 overlay. Depth *test* stays on so particles hide behind terrain; depth *write* goes off so particles do not punch invisible holes in each other — and with additive blending, draw order among particles does not matter at all, which is exactly why 1999 loved additive.

## Emitter recipes

| Param | Fountain | Smoke | Spark burst |
|---|---|---|---|
| Rate | 300 /s | 40 /s | one-shot, 150 |
| Life (s) | 1.6 | 3.5 | 0.3 – 0.8 |
| Velocity | up 6.0, cone 15° | up 0.8 ± 0.3 | radial 4 – 12 |
| Gravity | (0, -9.8, 0) | (0, +0.2, 0) rise | (0, -9.8, 0) |
| Size | 0.12 → 0.06 | 0.3 → 1.2 | 0.05 flat |
| Color | (140,180,255) → (40,60,120) | (70,70,75) → fade out | (255,220,120) → (255,60,0) |
| Blend | additive | alpha | additive |

Smoke is the odd one out: additive smoke glows, and smoke should not glow, so it wants ordinary alpha blending (`GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA`) — which reintroduces draw-order sensitivity. The honest fix is sorting by view depth; the 1999 fix is a soft, low-alpha texture and not looking too hard. Take the 1999 fix today, and draw

alpha-blended emitters after the additive ones. The particle texture itself is one PSP7 radial gradient, white core to transparent edge, 32×32.

## The budget

Count the cost per particle per frame: the simulation is trivial, but `particles_build` writes 4 vertices × 24 bytes = 96 bytes and roughly forty float operations, then the driver copies the whole buffer again. At 4096 particles that is 384 KB uploaded per frame and a pile of scalar float work on an in-order Atom core — the CPU, not the SGX545, will be the wall, and you should expect to find that wall somewhere in the low thousands. Find it exactly: put the live particle count in the Day 6 overlay next to the fps, crank the fountain rate until 60 becomes 30 (vsync quantizes, so the wall shows as a hard step, not a slope), and write both numbers in the journal. That number is your effects budget for the rest of the book — muzzle flashes, impacts, and weather all spend from it.

## Pitfalls

Particles invisible: depth mask still off from a previous frame (state leaks are always yours — restore what you touch), or the draw happened before opaque so everything z-failed. Particles visible through walls: you disabled depth *test* instead of depth *write*. Flicker at high counts: you exceeded MAX_PARTICLES and the spawn loop is fighting the reaper; the guard in the while condition is not optional. Quads spin as the camera rolls: they should — they face the screen, not the world; that is what a billboard is.

**DONE WHEN:** the fountain, a smoke column, and a spark burst all run at once, 1000+ particles at a playable framerate, correctly hidden behind terrain, glowing where they overlap — and the journal records the exact count where playable stopped being the right word.

**IF YOU ARE STUCK:** Nothing on screen: draw the buffer once with blending off and depth write on — if white quads appear, the geometry is fine and the problem is state; if not, check the index buffer count (6 per quad, not 4). Quads face the wrong way: you took columns instead of rows — the right vector is view[0], view[4], view[8], not view[0..2]. Fps craters far below the expected wall: you are calling `glBufferData` once per particle instead of once per frame. Additive looks dim: your particle texture has a dark background instead of black — additive adds; black adds nothing, gray adds gray.

# Day 19: The Console

Every engine you admired in 1999 had a console. Quake popularized it: press tilde, a translucent panel slides down over the game, and suddenly you are talking to the engine directly — changing variables, loading maps, binding keys, reading errors that would otherwise vanish into the void. Once you have one, you will wonder how you debugged anything without it. Today you build yours: a slide-down panel, a text input line, a scrollback buffer, a command table, and cvars — console variables, the engine's tunable knobs.

## The pieces

The console is four small systems glued together:

1. **State and animation.** A boolean target (open or closed) and a `slide` value from 0 to 1 that chases it. The panel's screen position is a function of `slide`, so the animation is free.

2. **Input.** Windows already does keyboard-to-character translation for you: handle `WM_CHAR` and you get ASCII with shift state applied, key repeat, everything. Accumulate printable characters into a line buffer; character 8 is backspace, 13 is enter.

3. **Scrollback.** A ring buffer of fixed-size lines. When it fills, the oldest line is overwritten. No allocation, no scrolling logic beyond modular arithmetic.

4. **Dispatch.** Tokenize the entered line in place, look the first token up in a command table, and if that fails, try the cvar list.

## State, ring buffer, input

```
#include <stdarg.h>

#define CON_LINES  128
#define CON_COLS    96
#define CON_HEIGHT 300.0f     /* pixels: top half of 1024x600 */
#define CON_SPEED    4.0f     /* open/close in 1/4 second     */

typedef struct {
    char  lines[CON_LINES][CON_COLS];
    int   head;               /* next slot to write           */
    char  input[CON_COLS];
    int   input_len;
    int   open;               /* target state                 */
    float slide;              /* 0 = closed .. 1 = fully open  */
} console_t;

console_t g_con;

void Con_Printf(const char *fmt, ...)
{
    va_list ap;
    va_start(ap, fmt);
    _vsnprintf(g_con.lines[g_con.head], CON_COLS - 1, fmt, ap);
    va_end(ap);
    g_con.lines[g_con.head][CON_COLS - 1] = '\0';
    g_con.head = (g_con.head + 1) % CON_LINES;
}

void Con_Update(float dt)
{
```

```
    g_con.slide += (g_con.open ? CON_SPEED : -CON_SPEED) * dt;
    if (g_con.slide < 0.0f) g_con.slide = 0.0f;
    if (g_con.slide > 1.0f) g_con.slide = 1.0f;
}

/* In the window procedure: */
case WM_CHAR:
    if (wParam == '`' || wParam == '~') {
        g_con.open = !g_con.open;
        return 0;
    }
    if (!g_con.open) break;
    if (wParam == 8) {                    /* backspace */
        if (g_con.input_len > 0) --g_con.input_len;
    } else if (wParam == 13) {            /* enter     */
        g_con.input[g_con.input_len] = '\0';
        Con_Printf("] %s", g_con.input);
        Con_Exec(g_con.input);
        g_con.input_len = 0;
    } else if (wParam >= 32 && wParam < 127 &&
               g_con.input_len < CON_COLS - 1) {
        g_con.input[g_con.input_len++] = (char)wParam;
    }
    g_con.input[g_con.input_len] = '\0';
    return 0;
```

Two wiring notes. First, `WM_CHAR` only arrives if your message loop calls `TranslateMessage` before `DispatchMessage` — the day 1 skeleton already does. Second, while `g_con.slide > 0`, skip the camera's WASD and mouse handling entirely, or you will strafe around the level while typing `fog 0.05`. On a non-US keyboard layout the tilde character may live somewhere unhelpful; if so, also toggle on `WM_KEYDOWN` with `wParam == VK_OEM_3`.

## Tokenizer, commands, cvars

The tokenizer is deliberately crude: split on spaces, in place, writing NUL terminators into the input buffer. No quoting, no escapes. Quake shipped with barely more.

```
#define MAX_ARGS 8

static int Con_Tokenize(char *text, char **argv)
{
    int argc = 0;
    char *p = text;
    while (argc < MAX_ARGS) {
        while (*p == ' ' || *p == '\t') *p++ = '\0';
        if (*p == '\0') break;
        argv[argc++] = p;
        while (*p && *p != ' ' && *p != '\t') ++p;
    }
    return argc;
}

typedef struct {
    const char *name;
    void (*fn)(int argc, char **argv);
} concmd_t;

typedef struct {
    const char *name;
    float       value;
    char        str[32];
} cvar_t;
```

```
static cvar_t g_cvars[] = {
    { "fog_density", 0.02f, "0.02" },
    { "r_wireframe", 0.0f,  "0"    }
};
#define NUM_CVARS (sizeof(g_cvars) / sizeof(g_cvars[0]))

cvar_t *Cvar_Get(const char *name)
{
    int i;
    for (i = 0; i < NUM_CVARS; ++i)
        if (!_stricmp(name, g_cvars[i].name)) return &g_cvars[i];
    return NULL;
}

void Cvar_Set(cvar_t *cv, const char *str)
{
    if (!cv) return;
    strncpy(cv->str, str, 31);
    cv->str[31] = '\0';
    cv->value = (float)atof(str);
}

static void Cmd_Quit(int argc, char **argv) { PostQuitMessage(0); }

static void Cmd_Fog(int argc, char **argv)
{
    cvar_t *cv = Cvar_Get("fog_density");
    if (argc >= 2) Cvar_Set(cv, argv[1]);
    Con_Printf("fog_density = %s", cv->str);
}

static const concmd_t g_cmds[] = {
    { "quit", Cmd_Quit },
    { "fog",  Cmd_Fog  }
};
#define NUM_CMDS (sizeof(g_cmds) / sizeof(g_cmds[0]))

void Con_Exec(char *text)
{
    char   *argv[MAX_ARGS];
    int     argc, i;
    cvar_t *cv;
    argc = Con_Tokenize(text, argv);
    if (argc == 0) return;
    for (i = 0; i < NUM_CMDS; ++i) {
        if (!_stricmp(argv[0], g_cmds[i].name)) {
            g_cmds[i].fn(argc, argv);
            return;
        }
    }
    cv = Cvar_Get(argv[0]);
    if (cv) {
        if (argc >= 2) Cvar_Set(cv, argv[1]);
        else Con_Printf("%s = %s", cv->name, cv->str);
        return;
    }
    Con_Printf("unknown command: %s", argv[0]);
}
```

Typing a bare cvar name prints its value; a name plus argument sets it. Your day 17 fog shader should now read its density from `Cvar_Get("fog_density")->value` each frame instead of a hardcoded constant. Cache the pointer at init — the table never moves — so the per-frame cost is one float load.

## Wireframe without glPolygonMode

> **WARNING:** GLES2 has no `glPolygonMode`. There is no render state that turns triangles into outlines. If you want wireframe, you draw `GL_LINES` yourself, and the trick is that you can reuse the existing vertex buffer untouched — only the index buffer changes.

Each triangle (a,b,c) becomes three edges: (a,b), (b,c), (c,a). Six line indices per triangle. Build the line index buffer once at mesh load and keep it next to the triangle one:

```
/* tri: 3 indices per triangle. Returns malloc'd 6-per-triangle
   line list; upload to a second GL_ELEMENT_ARRAY_BUFFER. */
unsigned short *Wire_BuildIndices(const unsigned short *tri,
                                  int num_tris)
{
    unsigned short *line;
    int i;
    line = (unsigned short *)malloc(num_tris * 6 * sizeof *line);
    if (!line) return NULL;
    for (i = 0; i < num_tris; ++i) {
        unsigned short a = tri[i * 3 + 0];
        unsigned short b = tri[i * 3 + 1];
        unsigned short c = tri[i * 3 + 2];
        line[i * 6 + 0] = a;  line[i * 6 + 1] = b;
        line[i * 6 + 2] = b;  line[i * 6 + 3] = c;
        line[i * 6 + 4] = c;  line[i * 6 + 5] = a;
    }
    return line;
}

/* At draw time, when r_wireframe is nonzero:      */
/* glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, wire_ibo); */
/* glDrawElements(GL_LINES, num_tris * 6,           */
/*                GL_UNSIGNED_SHORT, 0);            */
```

Shared edges get drawn twice; for a debug view nobody cares, and deduplicating edges buys you nothing you can see. Draw with the normal shader — or better, a constant-color one so the wireframe reads clearly against the sky.

## Rendering the console

This is pure day 6 material, which is why it stays prose. During the orthographic overlay pass, when `slide > 0`: compute the panel's top edge as `y = (slide - 1.0f) * CON_HEIGHT`, draw one translucent quad (black, alpha around 0.7, blending already on from day 6), then draw the input line at the panel bottom with a `"] "` prompt and a cursor that blinks on `(frame / 30) & 1`, then walk the ring backward from `head - 1` drawing scrollback lines upward until you run off the panel. That is the entire renderer: one quad and some glyphs.

## Pitfalls

The classic bug: the tilde keypress that opens the console also lands in the input buffer as a `~` character. The listing above avoids it by handling the toggle before the open check and returning. The second classic: forgetting that `Con_Tokenize` mutates its argument. Never pass a string literal or a buffer you still need — day 20's `exec` and key binds will both trip on this if you let them.

Performance: the console costs one quad plus maybe forty glyph quads through the day 6 batcher. The Atom will not notice. Do not bother scissoring or caching text — this is a debug tool; spend your cleverness elsewhere.

**DONE WHEN:** tilde slides the console over the running scene; `fog 0.08` visibly thickens the day 17 fog without a restart; `r_wireframe 1` shows the terrain as lines; `quit` exits cleanly.

**IF YOU ARE STUCK:**

— No characters arriving: confirm `TranslateMessage` runs, and that your `WM_CHAR` case actually returns 0 instead of falling through to `DefWindowProc`.

— Console text invisible: you are probably drawing it during the 3D pass with depth test on. It belongs in the ortho overlay pass, depth test off, blending on.

— `r_wireframe` draws garbage lines: you bound the line index buffer but left `num_tris * 3` as the count. It is `num_tris * 6` now.

— Slide animation snaps instead of gliding: you are updating `slide` in the render function with a raw frame time instead of in `engine_update` with the fixed `dt`.

# Day 20: Resources and Config

Right now your engine loads a texture wherever it feels like it and frees it never. That was fine for a spinning cube. It stops being fine the moment two meshes share a texture, or you want to unload a scene and load another without restarting — which is exactly what a `map` command needs to do. In 1999 you had maybe 8 MB of VRAM and, on this netbook, 1 GB of system RAM shared with the GPU. Leaks are not a code-quality debate here; they are a countdown timer. Today: refcounted resource tables, a boot-time config file, the `exec` command, and key binds.

## Refcounting, 1999 edition

No smart pointers, no destructors. A resource is a slot in a fixed table: the path it was loaded from, an opaque data pointer, and a reference count. Acquiring by path either finds a live slot with the same path (bump the count, return the same pointer — this replaces and generalizes the day 5 texture cache) or loads into a free slot with count 1. Releasing decrements; at zero, the unload callback runs and the slot is reusable. One generic table type, three instances: textures, meshes, sounds.

```
#define MAX_RES 256

typedef struct {
    char  path[64];
    void *data;
    int   refs;
} res_t;

typedef struct {
    const char *kind;                   /* "texture", "mesh"... */
    res_t  slot[MAX_RES];
    void *(*load)(const char *path);
    void  (*unload)(void *data);
} restab_t;

void *Res_Acquire(restab_t *t, const char *path)
{
    int i, free_i = -1;
    for (i = 0; i < MAX_RES; ++i) {
        if (t->slot[i].refs > 0) {
            if (!_stricmp(t->slot[i].path, path)) {
                t->slot[i].refs++;
                return t->slot[i].data;
            }
        } else if (free_i < 0) {
            free_i = i;
        }
    }
    if (free_i < 0) {
        Con_Printf("%s table full: %s", t->kind, path);
        return NULL;
    }
    t->slot[free_i].data = t->load(path);
    if (!t->slot[free_i].data) {
        Con_Printf("%s load failed: %s", t->kind, path);
        return NULL;
    }
    strncpy(t->slot[free_i].path, path, 63);
    t->slot[free_i].path[63] = '\0';
    t->slot[free_i].refs = 1;
    return t->slot[free_i].data;
}
```

```
void Res_Release(restab_t *t, void *data)
{
    int i;
    if (!data) return;
    for (i = 0; i < MAX_RES; ++i) {
        if (t->slot[i].refs > 0 && t->slot[i].data == data) {
            if (--t->slot[i].refs == 0) {
                t->unload(t->slot[i].data);
                t->slot[i].data = NULL;
                t->slot[i].path[0] = '\0';
            }
            return;
        }
    }
    Con_Printf("%s release of unknown pointer", t->kind);
}

int Res_Leaks(const restab_t *t)      /* call at shutdown */
{
    int i, n = 0;
    for (i = 0; i < MAX_RES; ++i) {
        if (t->slot[i].refs > 0) {
            printf("LEAK %s %s refs=%d\n",
                   t->kind, t->slot[i].path, t->slot[i].refs);
            ++n;
        }
    }
    return n;
}
```

Linear search over 256 slots — resist the urge to hash. Loads happen at scene changes, not per frame, and the Atom walks this table in microseconds. The `unload` callbacks are the day 5 / day 7 / day 21 free paths you already have: `glDeleteTextures`, free the VBO and index buffer, `BASS_SampleFree`. Wrap acquire and release in tiny typed helpers (`Tex_Acquire` returning `texture_t *`, and so on) so the casts live in one file. Route *every* load in the engine through these tables today — a leak counter you can bypass counts nothing.

At shutdown, sum `Res_Leaks` over all three tables and print the total. That number is your proof for the DONE line: bind a test command that loads and unloads the day 16 room scene 100 times, then quit and read the count. It must be zero, and the loop must not grow the process in Task Manager either — watch the working set while it runs.

## engine.cfg

Some settings are needed before the window exists — resolution and field of view cannot wait for the console. Those live in `engine.cfg`, a line-based key/value file parsed at boot with `fgets` and `sscanf`. Nothing fancier is justified:

```
# engine.cfg
width  1024
height 600
fov    75
```

```
typedef struct { int width, height; float fov; } config_t;

void Cfg_Load(const char *path, config_t *cfg)
{
    FILE *f;
    char line[128], key[32], val[64];
    f = fopen(path, "r");
    if (!f) return;                   /* defaults stand */
    while (fgets(line, sizeof line, f)) {
```

```
        if (line[0] == '#') continue;
        if (sscanf(line, "%31s %63s", key, val) != 2) continue;
        if      (!strcmp(key, "width"))  cfg->width  = atoi(val);
        else if (!strcmp(key, "height")) cfg->height = atoi(val);
        else if (!strcmp(key, "fov"))    cfg->fov = (float)atof(val);
    }
    fclose(f);
}
```

Fill `cfg` with defaults first; a missing or half-empty file must never crash the boot. The `%31s` and `%63s` width limits are not decoration — without them a long line in a hand-edited file scribbles over your stack, and on VC6 you will spend an evening in the disassembly window finding out why.

## exec and binds

Everything else — cvar defaults, binds, your fog taste — goes in `autoexec.cfg`, which is not a new format at all: it is a file of console commands, one per line, run by the new `exec` command. This is the Quake trick that makes the console the single configuration surface of the engine.

```
static void Cmd_Exec(int argc, char **argv)
{
    FILE *f;
    char line[128], *nl;
    if (argc < 2) { Con_Printf("usage: exec <file>"); return; }
    f = fopen(argv[1], "r");
    if (!f) { Con_Printf("exec: cannot open %s", argv[1]); return; }
    while (fgets(line, sizeof line, f)) {
        nl = strchr(line, '\n');
        if (nl) *nl = '\0';
        if (line[0] && line[0] != '#') Con_Exec(line);
    }
    fclose(f);
}
```

`Con_Exec` tokenizes in place, and here that is safe: `line` is scratch and refilled by the next `fgets`. Call `Con_Exec("exec autoexec.cfg")` once at startup, after GL and sound are up.

Binds map a key name to a command string. Store them as a flat array indexed by virtual-key code, with a small name table so humans can type `bind space jump`:

```
typedef struct { const char *name; int vk; } keyname_t;

static const keyname_t g_keynames[] = {
    { "space", VK_SPACE }, { "tab",  VK_TAB     },
    { "f1",    VK_F1    }, { "mouse1", VK_LBUTTON },
    /* ...single letters resolve as toupper(name[0])... */
};

char g_binds[256][64];    /* command string per VK code */
```

`Cmd_Bind` resolves `argv[1]` to a VK code (name table first, then the single-character fallback) and copies `argv[2]` into the slot. On `WM_KEYDOWN` — console closed, no autorepeat (check bit 30 of `lParam`) — copy the bound string into a scratch buffer and `Con_Exec` the copy, never the stored one: the tokenizer would chew NULs into your bind table and the bind would work exactly once. That bug is a rite of passage; consider yourself warned. Left out for you: `Cmd_Bind` itself, an `unbind`, and a `bindlist` printer — all under 20 lines each.

## Pitfalls

The subtle one is release order at scene unload: meshes hold texture references (the mesh's unload callback should release the textures it acquired), so free meshes before you audit textures. If your leak count is stubbornly nonzero, print the paths — `Res_Leaks` already does — and the culprit is almost always something acquired in an init path that has no matching shutdown, the skybox being the traditional offender. Also mind `_stricmp` for paths: on Windows, `DATA/Crate.png` and `data/crate.png` are the same file, and loading it twice under two spellings is a silent double allocation.

**DONE WHEN:** a console command loads and unloads a scene 100 times; the shutdown leak report prints 0 across all three tables and the working set stays flat; `exec autoexec.cfg` sets your cvars and binds at boot; changing `width` in engine.cfg changes the window at next launch.

**IF YOU ARE STUCK:**

— Leak count is exactly the number of loop iterations: your scene unload path releases meshes but the mesh unload callback never releases its textures.

— Crash on the second `exec` of the same file: you passed the stored bind string to `Con_Exec` instead of a copy, and the tokenizer NUL-ed it.

— `sscanf` returns 1, not 2, on valid-looking lines: Windows CRLF endings. `fgets` keeps the `\r`; `%s` actually handles it, but if you switched to `%[^\n]` for values with spaces, strip the `\r` yourself.

— Binds fire while typing in the console: gate the `WM_KEYDOWN` bind path on `!g_con.open`.

# Day 21: BASS Day

Silence is the last thing separating your engine from feeling like a game. In 1999 the answer was not a hand-rolled mixer — it was a library: FMOD, Miles, or BASS. You have BASS 2.4 on the shelf (`bass.h`, `bass.lib`, `bass.dll`), and it is a gift: it plays tracker modules natively. The .xm you composed in ModPlug Tracker is not converted, not rendered to a wav — BASS plays the module itself, patterns and samples, in a few hundred KB of RAM. That is how demo scene and shareware games shipped hours of music on a single floppy.

## Init, music, samples

Link `bass.lib`, keep `bass.dll` next to the exe. The whole API is C, handle-based, and returns 0/FALSE on failure with the reason parked in `BASS_ErrorGetCode()`.

```
#include "bass.h"

static HMUSIC  g_music;
static HSAMPLE g_step;

int Snd_Init(HWND hwnd)
{
    if (!BASS_Init(-1, 44100, 0, hwnd, NULL)) {
        Con_Printf("BASS_Init failed: %d", BASS_ErrorGetCode());
        return 0;                /* engine runs on, silently */
    }

    g_music = BASS_MusicLoad(FALSE, "data/track.xm", 0, 0,
                             BASS_MUSIC_LOOP | BASS_MUSIC_RAMP, 0);
    if (!g_music)
        Con_Printf("music: error %d", BASS_ErrorGetCode());
    else
        BASS_ChannelPlay(g_music, FALSE);

    /* max 8 overlapping instances of this sample */
    g_step = BASS_SampleLoad(FALSE, "data/step.wav", 0, 0, 8, 0);
    if (!g_step)
        Con_Printf("step.wav: error %d", BASS_ErrorGetCode());
    return 1;
}

void Snd_Shutdown(void)
{
    BASS_Free();   /* frees every music, sample and channel */
}
```

`BASS_Init(-1, ...)` means the default output device at 44100 Hz; the HWND ties the sound device to your window, which DirectSound on XP wants. `BASS_MUSIC_RAMP` turns on volume ramping so tracker note cuts do not click. A failed init should *not* kill the engine — guard every later call on the handles being nonzero and the game just runs silent, which is exactly what a 1999 shareware title did on a machine with a broken sound card.

Playing a sound effect is a three-step dance: a **sample** is the PCM data loaded once (refcount it in the day 20 sound table); a **channel** is one playing instance of it. The `max` of 8 in `BASS_SampleLoad` caps simultaneous instances — ask for a ninth and BASS recycles the oldest, which for footsteps is precisely right.

```
HCHANNEL Snd_Play(HSAMPLE smp)
{
    HCHANNEL ch;
```

```
    if (!smp) return 0;
    ch = BASS_SampleGetChannel(smp, FALSE);
    if (!ch) return 0;
    BASS_ChannelPlay(ch, FALSE);
    return ch;
}
```

## 3D-ish sound by hand

BASS has a real 3D API, but you do not need it, and doing the math yourself is the point of this book. Two numbers position a sound in the listener's head: volume from distance, pan from direction. Distance attenuation uses the same hyperbolic falloff as your day 14 point lights — `1 / (1 + d * k)` — because it never divides by zero and never quite reaches silence. Pan is the dot product of the camera's right vector with the unit vector toward the source: a sound dead ahead gives 0, hard right gives +1, hard left −1. Conveniently, that is exactly the range `BASS_ATTRIB_PAN` expects.

```
#define SND_ROLLOFF 0.06f     /* tune: bigger = dies faster */

typedef struct {
    vec3_t   pos;
    HCHANNEL chan;            /* 0 when not playing          */
} emitter_t;

void Snd_Spatialize(emitter_t *e, const vec3_t listener,
                    const vec3_t right)
{
    vec3_t dir;
    float  d, vol, pan;
    if (!e->chan) return;
    VectorSubtract(e->pos, listener, dir);
    d = VectorNormalize(dir);
    vol = 1.0f / (1.0f + d * SND_ROLLOFF);
    pan = DotProduct(right, dir);
    if (pan < -1.0f) pan = -1.0f;
    if (pan >  1.0f) pan =  1.0f;
    BASS_ChannelSetAttribute(e->chan, BASS_ATTRIB_VOL, vol);
    BASS_ChannelSetAttribute(e->chan, BASS_ATTRIB_PAN, pan);
}
```

Call this every frame for every live emitter — that is what makes a source *move* through the stereo field as it passes you, and it is a handful of attribute sets per frame, nothing the Atom will feel. Set the attributes once *before* `BASS_ChannelPlay` too, or the first audible instant plays at full center volume. The camera's right vector is one you already own: it is the first column of your day 3 view matrix, or `CrossProduct(forward, up)` normalized. When a channel ends, `BASS_ChannelIsActive(ch)` returns `BASS_ACTIVE_STOPPED`; clear `e->chan` then, and for looping emitters just start it again.

Give music its own volume cvar (`s_musicvol`) applied with `BASS_ChannelSetAttribute(g_music, BASS_ATTRIB_VOL, v)`, and you have a settings menu for free through the console.

## When BASS says no

Every failing BASS call leaves a code in `BASS_ErrorGetCode()`. Print it in the console and look it up here — this table covers what you will actually hit:

| Code | Name | What it really means |
|---|---|---|
| 0 | BASS_OK | No error (you checked too late) |

| | | |
|---|---|---|
| 1 | BASS_ERROR_MEM | Out of memory |
| 2 | BASS_ERROR_FILEOPEN | File not found — check the working directory, not the code |
| 3 | BASS_ERROR_DRIVER | No sound driver available |
| 5 | BASS_ERROR_HANDLE | Invalid handle — you used a freed or zero handle |
| 6 | BASS_ERROR_FORMAT | Sample format not supported (exotic wav encodings) |
| 8 | BASS_ERROR_INIT | BASS_Init has not been called — or it failed and you ignored it |
| 14 | BASS_ERROR_ALREADY | Already initialized / already playing |
| 18 | BASS_ERROR_NOCHAN | No free channel — raise the sample's max count |
| 20 | BASS_ERROR_ILLPARAM | Illegal parameter (pan outside −1..1, negative volume) |
| 23 | BASS_ERROR_DEVICE | Illegal device number |
| 41 | BASS_ERROR_FILEFORM | Unrecognized file format — the classic: you renamed an .it to .xm |
| -1 | BASS_ERROR_UNKNOWN | Some other mystery; usually the driver |

## Pitfalls

The one that bites everyone: `BASS_SampleGetChannel` returns a *new* channel handle per play — do not cache one channel and replay it expecting overlap; that is what the sample/channel split is for. Second: 8-bit or oddly-sampled wavs from ancient sound packs can fail with error 6; resave them as 16-bit 44.1 kHz PCM. Third: XP's kmixer adds latency on some machines — if footsteps feel late, call `BASS_SetConfig(BASS_CONFIG_BUFFER, 100)` before init and see the BASS header for the related update-period setting. And keep music files in the sound resource table from day 20: music is a resource like any other, and `BASS_Free` at shutdown is your backstop, not your bookkeeping.

**DONE WHEN:** your own module loops over the terrain; a footstep sample plays where an object stands, gets quieter as you walk away, and pans smoothly across the stereo field as it passes you.

**IF YOU ARE STUCK:**

— `BASS_Init` fails with error 23: another program grabbed the device exclusively, or you passed a device index other than −1 on a single-device machine.

— Music loads but you hear nothing: you loaded with `BASS_MUSIC_DECODE`-ish flags by accident, or never called `BASS_ChannelPlay`. Check `BASS_ChannelIsActive(g_music)`.

— Pan feels inverted: your camera right vector is actually the left vector — check the cross product order against day 3.

— Sounds pop at start: set VOL/PAN attributes before `BASS_ChannelPlay`, and check the wav has no DC offset (ModPlug's sample editor shows it).

# Day 22: Collision

Your camera is a ghost. It drifts through crates, sinks through floors, and ignores gravity — magnificent for building, useless for a game. Today the player gets a body. The 1999 recipe: the world is a pile of axis-aligned boxes, the player is a sphere, and the physics is "push the sphere out of the boxes and don't let velocity point into walls." No solver, no broadphase trees. Quake-era games shipped on exactly this level of sophistication, plus level design that hid it.

## The world and the player

World geometry for collision is a flat list of AABBs loaded from a text file — one box per line, six floats, `minx miny minz maxx maxy maxz`, parsed with the day 20 `fgets`/`sscanf` pattern (that loader is yours to write; it is ten lines). Author the boxes by hand to match the day 16 room: floor slab, four walls, a crate or two. Rendering geometry and collision geometry are separate things from today onward, and that separation is a feature, not a shortcut — every shipped engine of the era simplified its collision world.

The player is a sphere of radius ~0.4 world units centered at the feet plus radius. A standing human is taller than wide, so real engines use a capsule; the period-honest approximation is two spheres — one at the feet, one at head height — resolved with the same code. Get the single sphere working first; the head sphere is a loop iteration, not a new system.

## Closest point, pushout, slide

Sphere versus AABB reduces to one lovely primitive: the closest point on a box to a point is the point clamped to the box, per axis. If that closest point is nearer to the sphere center than the radius, you are penetrating, and the vector from closest point to center is the outward normal. Push the center out along it by the penetration depth, then remove the into-wall component of velocity so you slide along the surface instead of sticking to it. That velocity clip is the single line that makes movement feel like Quake.

```
typedef struct { vec3_t mins, maxs; } aabb_t;

/* Push sphere out of box. Returns 1 on contact, fills normal. */
int Sphere_vs_AABB(vec3_t center, float radius,
                   const aabb_t *box, vec3_t normal)
{
    vec3_t closest, delta, tmp;
    float  d2, d, push, best;
    int    i;

    for (i = 0; i < 3; ++i) {
        closest[i] = center[i];
        if (closest[i] < box->mins[i]) closest[i] = box->mins[i];
        if (closest[i] > box->maxs[i]) closest[i] = box->maxs[i];
    }
    VectorSubtract(center, closest, delta);
    d2 = DotProduct(delta, delta);
    if (d2 >= radius * radius) return 0;

    d = (float)sqrt(d2);
    if (d > 0.0001f) {
        VectorScale(delta, 1.0f / d, normal);
        push = radius - d;
    } else {
        /* Center inside the box: exit through nearest face. */
        best = box->maxs[0] - center[0];
        normal[0] = 1.0f; normal[1] = 0.0f; normal[2] = 0.0f;
        for (i = 0; i < 3; ++i) {
```

```
            float dpos = box->maxs[i] - center[i];
            float dneg = center[i] - box->mins[i];
            if (dpos < best) {
                best = dpos;
                VectorClear(normal); normal[i] =  1.0f;
            }
            if (dneg < best) {
                best = dneg;
                VectorClear(normal); normal[i] = -1.0f;
            }
        }
        push = best + radius;
    }
    VectorScale(normal, push, tmp);
    VectorAdd(center, tmp, center);
    return 1;
}

/* Slide: remove velocity pointing into the surface. */
void ClipVelocity(vec3_t vel, const vec3_t normal)
{
    float backoff = DotProduct(vel, normal);
    vec3_t tmp;
    if (backoff < 0.0f) {
        VectorScale(normal, backoff, tmp);
        VectorSubtract(vel, tmp, vel);
    }
}
```

(`VectorClear` is the obvious three-zeros macro; add it to day 1's mathlib if it is not there.) The center-inside branch looks paranoid, but it is what saves you when a substep lands the center inside a crate: without it, `d` is zero, the normal is garbage, and the player launches into orbit.

## Tunneling, and the substep answer

The pushout test is a *static* test — it asks "are we overlapping now," not "did we cross anything on the way." Run at 10 units/second against a 0.1-unit-thin wall and one 60 Hz step moves you 0.166 units — you can be cleanly on one side, then cleanly on the other, and the wall never fires. That is tunneling, and it is why players fell out of maps in 1999.

The rigorous fix is a swept test solving for time of impact. The era-honest fix — robust, and honestly what most shipped code did — is substeps: never move more than a fraction of the radius per integration step. Cheap insurance: your whole world is 20 boxes, and even 8 substeps of 20 tests is 160 clamps per frame. The Atom does that before breakfast.

```
#define GRAVITY     22.0f
#define JUMP_SPEED   7.5f
#define MAX_SUBMOVE  0.25f      /* < player radius */
#define STEP_HEIGHT  0.35f

void Player_Move(player_t *pl, const aabb_t *boxes, int nboxes,
                 float dt)
{
    vec3_t step, normal;
    float  dist;
    int    nsub, s, i;

    pl->vel[1] -= GRAVITY * dt;
    pl->grounded = 0;

    dist = (float)sqrt(DotProduct(pl->vel, pl->vel)) * dt;
    nsub = (int)(dist / MAX_SUBMOVE) + 1;
    VectorScale(pl->vel, dt / (float)nsub, step);
```

```
    for (s = 0; s < nsub; ++s) {
        VectorAdd(pl->center, step, pl->center);
        for (i = 0; i < nboxes; ++i) {
            if (Sphere_vs_AABB(pl->center, pl->radius,
                               &boxes[i], normal)) {
                if (normal[1] > 0.7f) {
                    pl->grounded = 1;
                    if (pl->vel[1] < 0.0f) pl->vel[1] = 0.0f;
                } else {
                    Player_TryStep(pl, &boxes[i], normal);
                }
                ClipVelocity(pl->vel, normal);
            }
        }
    }
    if (pl->grounded && pl->jump_pressed)
        pl->vel[1] = JUMP_SPEED;
}
```

The `normal[1] > 0.7f` test says “this surface is within about 45 degrees of flat, call it floor” — the same walkable-slope threshold Quake used. The grounded flag resets every frame and is re-earned by contact; jumping is one impulse, applied only when grounded, or you have invented the fly cheat.

`Player_TryStep` is the stair tolerance: when you hit a mostly-vertical surface, check whether the box top is within `STEP_HEIGHT` of the player's feet (`center[1] - radius`). If so, lift the player so the feet sit at the box top plus a small epsilon and carry on; otherwise let the clip do its wall job. Ten lines, yours to write — and it is the difference between a world where a 30 cm curb is an impassable cliff and one that feels built for people.

## Pitfalls

Jitter while standing still: gravity pulls you into the floor every frame and the pushout ejects you, and at some float boundaries the two alternate. The grounded zeroing of downward velocity in the listing handles most of it; if you still see shimmer, push out by `penetration - 0.001f` so you rest in kissing contact instead of exactly on the surface. Corner popping — sliding along two walls into a corner and getting shoved through one of them — comes from resolving boxes in a fixed order with big steps; substeps make it vanish in practice. And test the ten-minute soak from the DONE line seriously: walk into every corner, jump against every wall seam. Collision bugs are rare-event bugs, and ten minutes of trying is the test suite.

**DONE WHEN:** you can walk into the day 16 room, climb the doorstep without jumping, jump onto a crate, slide smoothly along walls, and after 10 minutes of deliberate abuse you have not fallen through the floor once.

**IF YOU ARE STUCK:**

— Falling through the floor at high speed: your `MAX_SUBMOVE` is bigger than the thinnest box. Thicken the floor slab to a full unit — era-honest fix — and cap substeps.

— Sticking to walls instead of sliding: you are zeroing velocity on contact instead of removing only the normal component. Only `ClipVelocity` touches velocity.

— Launched violently from inside geometry: the center-inside branch is missing, or your box file has a min/max pair swapped — validate `mins <= maxs` per axis at load and print offenders to the console.

— Can jump up walls: grounded is being set by wall contacts. Check the `normal[1]` threshold is applied before setting the flag.

# Day 23: The MD2 Loader

Quake II shipped in December 1997, and its model format is a small masterpiece of the do-it-with-nothing school: animation as a flipbook of complete vertex snapshots, positions crushed to one byte per axis, and smooth playback recovered by lerping between snapshots. No skeleton, no weights — just keyframes and interpolation. Better still for us: the interpolation moves to the GPU almost for free on GLES2, using a trick the 1997 renderer would have envied. Today you load MD2 and play a walk cycle without the CPU touching a single vertex per frame.

## The binary layout

Everything is little endian (native for you). The file opens with a 68-byte header of seventeen 32-bit ints:

| Offset | Type | Field | Meaning |
|---|---|---|---|
| 0 | s32 | ident | Magic: "IDP2" = 844121161 |
| 4 | s32 | version | Must be 8 |
| 8 | s32 | skinwidth | Skin texture width, pixels |
| 12 | s32 | skinheight | Skin texture height |
| 16 | s32 | framesize | Bytes per frame = 40 + 4*num_xyz |
| 20 | s32 | num_skins | Skin name count |
| 24 | s32 | num_xyz | Vertices per frame |
| 28 | s32 | num_st | Texcoord count (its own pool!) |
| 32 | s32 | num_tris | Triangle count |
| 36 | s32 | num_glcmds | GL command ints (we ignore these) |
| 40 | s32 | num_frames | Keyframe count |
| 44 | s32 | ofs_skins | File offset: skin names (64 bytes each) |
| 48 | s32 | ofs_st | Offset: texcoords |
| 52 | s32 | ofs_tris | Offset: triangles |
| 56 | s32 | ofs_frames | Offset: frames |
| 60 | s32 | ofs_glcmds | Offset: GL commands |
| 64 | s32 | ofs_end | File size |

Texcoords are shorts, divided by skin size to get 0..1:

| Type | Field | Meaning |
|---|---|---|
| s16 | s | u = s / skinwidth |
| s16 | t | v = t / skinheight |

Triangles carry *two* index sets — positions and texcoords are separate pools, exactly the situation you met with OBJ `v/vt` on day 7, and it gets the same dedup cure:

| Type | Field | Meaning |
|---|---|---|
| s16[3] | index_xyz | Into the frame vertex pool |
| s16[3] | index_st | Into the texcoord pool |

Each frame is a scale/translate pair followed by the whole model, one byte per coordinate:

| Type | Field | Meaning |
|---|---|---|
| f32[3] | scale | Per-axis dequantize scale |
| f32[3] | translate | Per-axis offset |
| char[16] | name | "run1", "death3"... |
| u8[4] × num_xyz | verts | x,y,z packed bytes + normal index |

Decompression is `v[i] = packed[i] * scale[i] + translate[i]`. One byte per axis means 256 positions per axis per frame — you can see the quantization wobble in Quake II if you look for it, and now you know what it is. The fourth byte indexes id's table of 162 precomputed normals; we are going to ignore it, for reasons below. Note also MD2 is Z-up like everything id made: swap y and z (and negate one — check handedness against day 7) on load to match your +Y-up world.

## Reading it safely

The header is seventeen ints and tempting to `fread` as a struct. With ints it even works — but the frame struct mixes floats, chars, and byte arrays, and one compiler padding surprise gives you a model that loads as confetti. VC6 honors `#pragma pack(push,1)`, but a header included in the wrong order silently un-packs your struct and costs an afternoon. Read field by field instead; on this little-endian x86, scalars can be `fread` directly:

```
static long read_s32(FILE *f)
{ long v;  fread(&v, 4, 1, f); return v; }
static short read_s16(FILE *f)
{ short v; fread(&v, 2, 1, f); return v; }
static float read_f32(FILE *f)
{ float v; fread(&v, 4, 1, f); return v; }

/* Load one frame's positions, decompressed, axis-swapped. */
static void MD2_ReadFrame(FILE *f, long ofs_frames, long framesize,
                          int frame, int num_xyz, float *out_xyz)
{
    float scale[3], trans[3];
    unsigned char pv[4];
    char  name[16];
    int   i;
    fseek(f, ofs_frames + frame * framesize, SEEK_SET);
    for (i = 0; i < 3; ++i) scale[i] = read_f32(f);
    for (i = 0; i < 3; ++i) trans[i] = read_f32(f);
    fread(name, 1, 16, f);
    for (i = 0; i < num_xyz; ++i) {
        float x, y, z;
        fread(pv, 1, 4, f);
        x = pv[0] * scale[0] + trans[0];
        y = pv[1] * scale[1] + trans[1];
        z = pv[2] * scale[2] + trans[2];
        out_xyz[i * 3 + 0] = x;        /* id Z-up -> our Y-up */
        out_xyz[i * 3 + 1] = z;
        out_xyz[i * 3 + 2] = -y;
    }
}
```

Validate `ident` and `version` before touching anything else, and fail through the console, not a crash.

## Dedup once, build per-frame VBOs

Because triangles pair a position index with a texcoord index, run the day 7 dedup on the pairs `(index_xyz, index_st)`: each unique pair becomes one output vertex, and you get a remap table `unique[k] = which index_xyz` plus a single index buffer. The crucial observation: the pairing is identical in every frame, so you dedup *once* and reuse the remap for all frames.

Then build one static VBO *per frame*, each holding interleaved position + normal (6 floats, 24-byte stride) for the deduped vertex list, one static VBO for texcoords, and one index buffer. For normals, skip id's 162-entry table: it exists to save 1997 CPUs from recomputing lighting normals, and it does not survive our dedup cleanly anyway. Instead, at load, for each frame: zero an accumulator per vertex, compute each triangle's face normal with your day 1 `CrossProduct` from that frame's positions, add it to the triangle's three vertices, then `VectorNormalize` each — the same smooth-normal pass you wrote for the day 11 terrain. Load time grows by milliseconds; quality beats the byte-quantized originals.

Memory check, because you have 1 GB total: 198 frames × ~600 deduped vertices × 24 bytes ≈ 2.8 MB per character. Fine. Ten characters would not be — budget awareness is the toll for the flipbook's simplicity.

## Lerp in the vertex shader

Here is the trick. A GLES2 vertex attribute does not care which VBO it comes from — each attribute slot binds independently. So: bind frame A's VBO and point attributes 0 and 1 at its position and normal; bind frame B's VBO and point attributes 2 and 3 at its; bind the texcoord VBO for attribute 4. One uniform carries the blend factor. The CPU per frame does five pointer binds and one uniform — the actual interpolation runs on the GPU, per vertex:

```
static const char *MD2_VS =
"attribute vec3 aPosA;                                  \n"
"attribute vec3 aNormA;                                 \n"
"attribute vec3 aPosB;                                  \n"
"attribute vec3 aNormB;                                 \n"
"attribute vec2 aST;                                    \n"
"uniform mat4 uMVP;                                     \n"
"uniform float uLerp;    /* 0 = frame A, 1 = frame B */ \n"
"varying vec2 vST;                                      \n"
"varying vec3 vNorm;                                    \n"
"void main() {                                          \n"
"    vec3 p = mix(aPosA, aPosB, uLerp);                 \n"
"    vNorm  = normalize(mix(aNormA, aNormB, uLerp));    \n"
"    vST    = aST;                                      \n"
"    gl_Position = uMVP * vec4(p, 1.0);                 \n"
"}                                                      \n";
```

Driving it is a clock: an animation is a frame range and a rate. `t = time * fps`; frame A is `first + ((int)t % count)`, frame B the next (wrapping to `first`), and `uLerp` is the fractional part of `t`. Standard Quake II models share one frame map; the ranges you will use:

| Animation | Frames | FPS |
|---|---|---|
| stand | 0–39 | 9 |
| run | 40–45 | 10 |
| attack | 46–53 | 10 |
| pain (a/b/c) | 54–65 | 7 |
| jump | 66–71 | 7 |
| crouch stand | 135–153 | 10 |

| | | |
|---|---|---|
| crouch walk | 154–159 | 7 |
| death (fallback) | 178–183 | 7 |

Milkshape exports MD2 too, so a model of your own can carry any ranges you define — the frame `name` fields tell you what the author intended. Left to you: the VAO setup per frame pair (`OES_vertex_array_object` saves the rebinding if you make one VAO per adjacent frame pair, but plain rebinding is five calls and honestly fine), and the skin texture through day 20's `Tex_Acquire`.

## Pitfalls

Lerping across a range boundary — frame 45 back to 40 — is correct and looks right for loops; lerping from frame 45 to 46 (run into attack) blends two unrelated poses for one step. When you switch animations, snap the clock to the new range's start. If the model renders inside-out, your y/z swap flipped the winding: either negate the other axis or flip the index order at load, and re-check against a known-good OBJ. And if the walk cycle looks like it is underwater, you forgot the fractional part and are passing whole `t` as `uLerp` — clamped to 1.0, every frame ends fully at B.

**DONE WHEN:** a period MD2 (or your Milkshape export) runs its walk cycle smoothly over the terrain, lit by day 14 lighting via the interpolated normals, with the console fps unchanged from an equally sized static mesh.

**IF YOU ARE STUCK:**

— Confetti explosion on screen: index buffer built from `index_xyz` but VBOs built in deduped order. Every draw index must reference the deduped list.

— Model is tiny or enormous: Quake II units are roughly 1 unit = 1 inch. Scale by ~0.025 at load to meet your meters-ish world.

— Attributes 2/3 read as zero: you called `glEnableVertexAttribArray` for 0 and 1 only. All five slots need enabling.

— Seams in the skin: MD2 texcoords are texel corners; sample with `GL_CLAMP_TO_EDGE` and regenerate mipmaps (day 4 rules apply).

# Day 24: MS3D Skeletal Animation

MD2's flipbook stores the whole model per keyframe. Skeletal animation stores a handful of joints per keyframe and computes the vertices — the way Half-Life did it in 1998, on CPUs slower than your Atom. Milkshape's native .ms3d format is the friendliest possible introduction: one bone per vertex (exactly the period-correct restriction), Euler keyframes, and a layout you can read with the same field-by-field discipline as yesterday. Today you parse it, build the bind pose, and skin on the CPU into a dynamic VBO.

## The file, section by section

Everything is little endian with 1-byte packing — and this time the structs genuinely do not match VC6's default padding (15-byte vertices, 70-byte triangles), so field-by-field reading is not caution, it is the only correct way short of pragma games. The file is sequential sections, each a count followed by records:

| Order | Section | Contents |
|---|---|---|
| 1 | Header | char id[10] = "MS3D000000", s32 version = 4 |
| 2 | Vertices | u16 count, then vertex records |
| 3 | Triangles | u16 count, then triangle records |
| 4 | Groups | u16 count, then variable-size group records |
| 5 | Materials | u16 count, then 361-byte material records |
| 6 | Animation | f32 fps, f32 currentTime, s32 totalFrames |
| 7 | Joints | u16 count, then variable-size joint records |

Vertex record, 15 bytes:

| Type | Field | Meaning |
|---|---|---|
| u8 | flags | Editor state; ignore |
| f32[3] | vertex | Bind-pose position |
| s8 | boneId | Joint index, -1 = unattached |
| u8 | refCount | Editor state; ignore |

Triangle record, 70 bytes — note normals and texcoords live per corner here, not per vertex:

| Type | Field | Meaning |
|---|---|---|
| u16 | flags | Ignore |
| u16[3] | vertexIndices | Into the vertex section |
| f32[3][3] | vertexNormals | Normal per corner |
| f32[3] | s | U per corner |
| f32[3] | t | V per corner (flip: GL wants 1-t) |
| u8 | smoothingGroup | Ignore |
| u8 | groupIndex | Which group |

Group record, variable size: u8 flags, char name[32], u16 numtriangles, u16 triangleIndices[numtriangles], s8 materialIndex. Material record, 361 bytes: char name[32], f32 ambient[4], diffuse[4], specular[4], emissive[4], f32

shininess, f32 transparency, s8 mode, char texture[128], char alphamap[128] — the texture path goes through day 20's `Tex_Acquire`. Then the three animation globals (order 6 above), and the joints:

| Type | Field | Meaning |
|---|---|---|
| u8 | flags | Ignore |
| char[32] | name | Joint name |
| char[32] | parentName | Empty string = root |
| f32[3] | rotation | Bind rotation, Euler XYZ, **radians** |
| f32[3] | position | Bind translation, parent space |
| u16 | nKeyFramesRot | Rotation key count |
| u16 | nKeyFramesTrans | Translation key count |
| 16 bytes each | rot keys | {f32 time (seconds), f32 rotation[3]} |
| 16 bytes each | trans keys | {f32 time, f32 position[3]} |

**WARNING:** the keyframes are *relative to the joint's bind-pose local transform*, not absolute. The current local matrix is bind_local × key_matrix. Treat the keys as absolute and the model explodes into a pose only a chiropractor could love. This is the number one MS3D loader bug.

Resolve `parentName` to an index by `strcmp` against joint names after reading them all. Milkshape writes parents before children, and the listing below relies on that order; assert it at load (`parent_idx < i`) rather than trusting it.

## Bind pose and its inverse

Each joint's local matrix is its bind rotation with the bind translation in the fourth column — build the rotation as R = Rz·Ry·Rx (X applied first) and then write the position into the translation slots, which is exactly T·R without a second multiply. Absolute matrices chain down the hierarchy, and the inverse bind matrix — the thing that carries a mesh vertex from model space into the joint's own space — is precomputed once. For a rigid transform the inverse is cheap: transpose the rotation, counter-rotate the translation.

```
#define RAD2DEG 57.29577951f

static void Joint_LocalMatrix(float out[16], const float rot[3],
                              const float pos[3])
{
    float rx[16], ry[16], rz[16], t[16];
    mat4_rotate_x(rx, rot[0] * RAD2DEG);
    mat4_rotate_y(ry, rot[1] * RAD2DEG);
    mat4_rotate_z(rz, rot[2] * RAD2DEG);
    mat4_multiply(t, ry, rx);
    mat4_multiply(out, rz, t);          /* R = Rz * Ry * Rx */
    out[12] = pos[0];
    out[13] = pos[1];
    out[14] = pos[2];
}

void mat4_rigid_invert(float out[16], const float m[16])
{
    int c, r;
    for (c = 0; c < 3; ++c)
        for (r = 0; r < 3; ++r)
            out[c * 4 + r] = m[r * 4 + c];
    out[3] = out[7] = out[11] = 0.0f;  out[15] = 1.0f;
    out[12] = -(out[0]*m[12] + out[4]*m[13] + out[8]*m[14]);
```

```
    out[13] = -(out[1]*m[12] + out[5]*m[13] + out[9]*m[14]);
    out[14] = -(out[2]*m[12] + out[6]*m[13] + out[10]*m[14]);
}

/* After loading all joints, parents-first: */
void Skel_BuildBindPose(joint_t *joints, int n)
{
    int i;
    for (i = 0; i < n; ++i) {
        joint_t *j = &joints[i];
        Joint_LocalMatrix(j->local, j->rot, j->pos);
        if (j->parent_idx >= 0)
            mat4_multiply(j->abs_bind,
                          joints[j->parent_idx].abs_bind, j->local);
        else
            memcpy(j->abs_bind, j->local, sizeof j->abs_bind);
        mat4_rigid_invert(j->inv_bind, j->abs_bind);
    }
}
```

## Per frame: interpolate, compose, skin

At animation time `t`, for each joint: find the two keyframes bracketing `t` (clamping at the ends), lerp the translation, and lerp the rotation. Lerping Euler angles is wrong in general — it takes the long way around wrap-around, and combined axes can wobble — but with Milkshape's densely keyed exports it mostly passes, if you wrap each component difference into ±π first. The honest fix is converting to quaternions and slerping; the math lives in Appendix F, and the loader slot for it is this one function. Start with wrapped Euler lerp, upgrade when you see a shoulder twitch.

Build the key matrix from the interpolated values with the same `Joint_LocalMatrix`, then compose:

```
/* current = parent_current * (bind_local * key_local)      */
/* skin    = current * inv_bind  (precompute per joint)     */
void Skel_Pose(joint_t *joints, int n, float t)
{
    float key[16], loc[16], krot[3], kpos[3];
    int i;
    for (i = 0; i < n; ++i) {
        joint_t *j = &joints[i];
        Anim_LerpKeys(j, t, krot, kpos);   /* the search+lerp */
        Joint_LocalMatrix(key, krot, kpos);
        mat4_multiply(loc, j->local, key);
        if (j->parent_idx >= 0)
            mat4_multiply(j->current,
                          joints[j->parent_idx].current, loc);
        else
            memcpy(j->current, loc, sizeof j->current);
        mat4_multiply(j->skin, j->current, j->inv_bind);
    }
}

/* CPU skinning: one bone per vertex, into a dynamic VBO.   */
/* dst layout: pos[3] norm[3] interleaved; ST is static.    */
void MS3D_Skin(const model_t *m, float *dst)
{
    int i;
    for (i = 0; i < m->num_verts; ++i) {
        const skinvert_t *sv = &m->verts[i];
        float *o = dst + i * 6;
        const float *J;
        if (sv->bone < 0) {
            memcpy(o, sv->pos, 3 * sizeof(float));
            memcpy(o + 3, sv->norm, 3 * sizeof(float));
```

```
            continue;
        }
        J = m->joints[sv->bone].skin;
        o[0] = J[0]*sv->pos[0] + J[4]*sv->pos[1]
             + J[8]*sv->pos[2] + J[12];
        o[1] = J[1]*sv->pos[0] + J[5]*sv->pos[1]
             + J[9]*sv->pos[2] + J[13];
        o[2] = J[2]*sv->pos[0] + J[6]*sv->pos[1]
             + J[10]*sv->pos[2] + J[14];
        o[3] = J[0]*sv->norm[0] + J[4]*sv->norm[1]
             + J[8]*sv->norm[2];
        o[4] = J[1]*sv->norm[0] + J[5]*sv->norm[1]
             + J[9]*sv->norm[2];
        o[5] = J[2]*sv->norm[0] + J[6]*sv->norm[1]
             + J[10]*sv->norm[2];
    }
    glBindBuffer(GL_ARRAY_BUFFER, m->vbo);
    glBufferSubData(GL_ARRAY_BUFFER, 0,
                    m->num_verts * 6 * sizeof(float), dst);
}
```

Notes on that listing. The mesh side needs the day 7 dedup yet again: MS3D positions are per vertex but normals and texcoords are per triangle corner, so the tuple is (vertexIndex, normal, s, t), and each deduped output vertex inherits its `boneId` from the source vertex. The vertex positions in the file are in *model* space at bind pose — that is why `inv_bind` exists: skin = current × inv_bind takes the vertex from model space to joint space and back out through the animated joint. Normals use only the rotation part (columns, no `J[12..14]`), legal because rigid transforms need no inverse-transpose. Create the VBO once with `GL_DYNAMIC_DRAW` and update with `glBufferSubData`; ANGLE turns that into a D3D9 dynamic buffer update, which is the fast path on this stack.

## MD2 vs MS3D on the Atom: the scoreboard

Run both characters side by side and journal the fps. The MD2 wins on this machine, and it is worth understanding exactly why. Its per-frame cost is five attribute binds and one uniform; the lerp rides inside vertex shader work the GPU was doing anyway. The MS3D costs, per frame: keyframe searches and ~30 matrix multiplies for the skeleton (nothing), then per deduped vertex about 36 multiply-adds, then a bus upload of the whole vertex buffer through ANGLE to D3D9. On a 1.6 GHz in-order Atom, 1500 vertices of skinning is roughly a tenth of a millisecond of pure FPU work — tolerable — but the upload and driver validation add up, and five skinned characters cost five uploads. Meanwhile MD2's weakness is memory (a VBO per frame) and its rigidity: you cannot blend animations, aim a head, or ragdoll a flipbook. That trade — CPU time versus memory and flexibility — is the whole 1998–2004 character animation story; hardware skinning with a bone palette in the vertex shader (your GLES2 can, with ~20 mat4 uniforms) is the modern answer, and makes a fine extra-credit project after day 30.

## Pitfalls

Model in a heap at the origin: keys treated as absolute (see the warning — it is always this). Limbs detach when animating: `parent_idx` resolution failed, usually a trailing space in `parentName`; compare with `strncmp` against 32 bytes and trim. Model looks right at frame 0 but drifts as it plays: your key times are in seconds, but you indexed them by frame number — Milkshape writes `time = frame / fAnimationFPS`. Upside-down texture: you skipped the `1 - t` flip from the triangle table.

**DONE WHEN:** a skeleton-animated character walks the terrain next to the MD2 model; both are lit correctly; the journal records fps with one, the other, and both, and one sentence on which the Atom prefers and why.

**IF YOU ARE STUCK:**

— Everything reads as garbage after the vertices: your triangle read is off by two bytes — `flags` is u16 here but u8 in vertices. The section sizes in the tables are exact; `ftell` after each section and check.

— Character is rigid in bind pose: `Skel_Pose` runs but the skin matrices are identity — you multiplied `current * inv_bind` with `current` still the bind absolute. Confirm `Anim_LerpKeys` actually varies with `t`.

— Vertices flicker between poses: you are writing the dynamic buffer that the GPU is still drawing from. On this stack, `glBufferSubData` after `eglSwapBuffers` and before the draw is safe; do not skin twice per frame.

— A single arm is wrong: one vertex with boneId -1 pinned in model space — Milkshape lets you forget to assign vertices. Reassign in the editor, or treat -1 as the root joint.

# Day 25: Render to Texture

Welcome to the boss fights. Everything from here to Day 30 is a feature that, in its day, sold graphics cards. Today's is render to texture: pointing the renderer at a texture instead of the screen. In 2000, Deus Ex shipped security monitors that showed live camera feeds and players stopped in hallways just to look at them. Quake III faked its mirrors by re-rendering the scene through a stencil mask, because render-target textures were still exotic on consumer cards. You have it as a core feature of GLES2: the framebuffer object. Today you build a mirror for the Day 16 room and a full-screen postprocess pass, and both reuse the same twenty lines of setup.

## The framebuffer object

Everything you have drawn so far landed in the default framebuffer, the one EGL created with your window. An FBO is an alternative destination you assemble yourself from parts: a *color attachment* (a texture, so you can sample the result later) and a *depth attachment* (a renderbuffer, which is like a texture you can render into but never sample — cheaper, and all we need for depth). Bind the FBO, draw, and fragments land in your texture. Bind framebuffer zero and you are back on the screen.

Render-target textures on GLES2 play by stricter rules than the textures you loaded on Day 4. Follow all three or enjoy a black quad:

- **Power-of-two sizes.** 256, 512, 1024. The screen is 1024x600; your render target is not. We will deal with that below.
- **GL_CLAMP_TO_EDGE** on both wrap axes. Repeat wrapping on a render target is not guaranteed and ANGLE will refuse or misbehave.
- **No mipmaps.** Filter is GL_LINEAR, min and mag. You render into level 0 and level 0 is all there is.

One more quiet trap: core GLES2 only guarantees 16-bit color formats (RGB565 and friends) as renderable. Full 8-bit-per-channel targets come from the `OES_rgb8_rgba8` extension, which is on your extension list, so `GL_RGB` + `GL_UNSIGNED_BYTE` works here. For depth, core guarantees exactly one renderbuffer format: `GL_DEPTH_COMPONENT16`. Sixteen bits of depth is plenty for a room.

```
typedef struct rtarget_s {
    GLuint fbo;
    GLuint color;             /* texture you sample later */
    GLuint depth;             /* renderbuffer, never sampled */
    int    w, h;
} rtarget_t;

int RTarget_Create(rtarget_t *rt, int w, int h)
{
    GLenum status;

    rt->w = w;
    rt->h = h;

    glGenTextures(1, &rt->color);
    glBindTexture(GL_TEXTURE_2D, rt->color);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0,
                 GL_RGB, GL_UNSIGNED_BYTE, NULL);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                    GL_CLAMP_TO_EDGE);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                    GL_CLAMP_TO_EDGE);
```

```
    glGenRenderbuffers(1, &rt->depth);
    glBindRenderbuffer(GL_RENDERBUFFER, rt->depth);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16,
                          w, h);

    glGenFramebuffers(1, &rt->fbo);
    glBindFramebuffer(GL_FRAMEBUFFER, rt->fbo);
    glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                           GL_TEXTURE_2D, rt->color, 0);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                              GL_RENDERBUFFER, rt->depth);

    status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    if (status != GL_FRAMEBUFFER_COMPLETE) {
        Con_Printf("FBO incomplete: 0x%04X\n", status);
        return 0;
    }
    return 1;
}
```

Always check completeness. The status codes tell you exactly what you got wrong:

| Code | Name | What you did |
|---|---|---|
| 0x8CD5 | GL_FRAMEBUFFER_COMPLETE | Nothing. Carry on. |
| 0x8CD6 | INCOMPLETE_ATTACHMENT | An attachment is broken: texture id 0, wrong level, or a non-renderable format. |
| 0x8CD7 | INCOMPLETE_MISSING_ATTACHMENT | You attached nothing at all. Usually a typo'd enum. |
| 0x8CD9 | INCOMPLETE_DIMENSIONS | Color and depth attachments are different sizes. |
| 0x8CDD | UNSUPPORTED | A format combination this driver refuses. Fall back to RGB565 or DEPTH_COMPONENT16. |
| 0 | (zero) | glCheckFramebufferStatus itself failed — bad target argument. |

## The mirror

A mirror shows the world from a camera that does not exist: your eye reflected through the mirror plane. Render the scene from that virtual camera into an FBO, then paste the result onto the mirror quad. The math is one formula. For a plane with unit normal `n` and plane constant `d` (where `d = DotProduct(n, any_point_on_plane)`), the reflection of a point `q` is `q - 2*(DotProduct(n,q) - d)*n`.

```
void ReflectPoint(const vec3_t q, const vec3_t n, float d,
                  vec3_t out)
{
    float k;
    vec3_t tmp;

    k = 2.0f * (DotProduct(n, q) - d);
    VectorScale(n, k, tmp);
    VectorSubtract(q, tmp, out);
}
```

```
     E                  E  = your eye
      \                 M  = a point on the mirror
       \                S  = what the mirror shows at M
        v
```

```
  ======M=======  mirror plane
        ^\
       /  \
      /    v
     E'     S   the ray E'->M continues to exactly the
                point S that the reflected ray from E hits
```

Reflect the eye, reflect the look-at point, and reflect the up vector as a direction (same formula with `d = 0`). One consequence needs calling out: a reflection flips handedness, so every triangle's winding order flips too. Your Day 2 backface culling would cull exactly the wrong faces and the mirror world would render inside out. The fix is one call: `glFrontFace(GL_CW)` for the duration of the mirror pass, back to `GL_CCW` after.

```
void R_RenderMirror(void)
{
    vec3_t eye_r, at_r, up_r;
    float d;

    d = DotProduct(g_mirror_n, g_mirror_pt);
    ReflectPoint(g_cam_eye, g_mirror_n, d, eye_r);
    ReflectPoint(g_cam_at,  g_mirror_n, d, at_r);
    ReflectPoint(g_cam_up,  g_mirror_n, 0.0f, up_r);

    glBindFramebuffer(GL_FRAMEBUFFER, g_mirror_rt.fbo);
    glViewport(0, 0, g_mirror_rt.w, g_mirror_rt.h);
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glFrontFace(GL_CW);
    mat4_lookat(g_view, eye_r, at_r, up_r);
    /* g_proj stays the MAIN camera's projection: same fov,
       same SCREEN aspect (1024/600), even though the FBO
       is square. This is what makes the shortcut below work. */
    R_DrawWorld();           /* everything EXCEPT the mirror quad */
    glFrontFace(GL_CCW);

    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    glViewport(0, 0, g_width, g_height);
}
```

Now, which texel does the mirror quad sample? The rigorous answer is projective texturing: transform each mirror vertex by the reflected camera's view-projection and divide by w in the fragment shader. But there is a shortcut, and it is exact, not an approximation. The reflected camera is mathematically the main camera looking at a reflected world: reflected-view = main-view times the world reflection matrix. Points *on the mirror plane* are their own reflections, so they project to the same screen position in both passes. That means the reflection texture, rendered with the same projection, lines up with the mirror quad in screen space. Sample it at the fragment's own screen position and you are done:

```
static const char *FS_MIRROR =
"precision mediump float;\n"
"uniform sampler2D u_reflection;\n"
"uniform vec2 u_screen;\n"          /* (1024.0, 600.0) */
"void main() {\n"
"    vec2 uv = gl_FragCoord.xy / u_screen;\n"
"    gl_FragColor = vec4(texture2D(u_reflection, uv).rgb, 1.0);\n"
"}\n";
```

`gl_FragCoord` is in window pixels; dividing by the window size gives [0,1], which spans the whole reflection texture because the mirror pass filled its whole (square) viewport with the same field of view. No flip is needed — both passes use GL's bottom-up convention. Draw the mirror quad during the normal scene pass with this shader, and your Day 24

character does its walk cycle in the glass. Skin it once on the CPU and draw the same buffer in both passes; do not run the skinning twice.

## Postprocess: the whole screen is a texture

The second trick of the day: render the *entire scene* into an FBO, then draw one full-screen quad to the real screen through an effect shader. Every screen-space effect of the next decade — bloom, blur, color grading — is this one pattern.

The screen is 1024x600 and render targets must be power-of-two, so allocate 1024x1024 and only use part of it: during the scene pass call `glViewport(0, 0, 1024, 600)` so you render into the bottom 600 rows, then scale your sampling coordinates by `600.0/1024.0` in the postprocess shader. The full-screen quad itself needs no matrices: give it positions in [-1,1] clip space directly and derive UVs as `pos * 0.5 + 0.5` in the vertex shader. The vignette:

```
static const char *FS_VIGNETTE =
"precision mediump float;\n"
"varying vec2 v_uv;\n"
"uniform sampler2D u_scene;\n"
"uniform vec2 u_uvscale;\n"        /* (1.0, 600.0/1024.0) */
"void main() {\n"
"    vec3 c = texture2D(u_scene, v_uv * u_uvscale).rgb;\n"
"    float d = distance(v_uv, vec2(0.5, 0.5));\n"
"    float v = 1.0 - 0.55 * smoothstep(0.35, 0.80, d);\n"
"    gl_FragColor = vec4(c * v, 1.0);\n"
"}\n";
```

Add a console cvar `r_vignette` so you can toggle the whole postprocess path at runtime and see its cost in the fps counter. For sepia, replace the multiply with a dot against luminance weights `vec3(0.30, 0.59, 0.11)` and tint the result.

**WARNING:** Never sample a texture that is attached to the currently bound framebuffer. That is a feedback loop, undefined by the spec, and ANGLE's D3D9 backend will typically give you black or garbage. This is why the mirror pass draws everything *except* the mirror quad.

**NOTE:** Your SGX545 is a tile-based deferred renderer. It does not draw triangles as you submit them; it bins a whole render pass into tiles and shades each tile once at the end. Every `glBindFramebuffer` ends a pass: the chip must flush every tile out to memory, and if you come back to that framebuffer later it must read everything back in. So group your passes — mirror FBO once, scene FBO once, screen once, in that order, never interleaved — and always clear color and depth immediately after binding an FBO. The clear tells the driver the old contents are dead and saves the read-back. On this GPU an unnecessary FBO ping-pong can cost more than the drawing itself.

What is left to you: the fullscreen-quad VBO, the cvar plumbing, and hooking `R_RenderMirror` before the main pass. Budget note for the Atom: the mirror pass re-renders the room, so expect roughly double the room's draw cost. A 512x512 mirror plus the 1024-wide scene FBO holds 60 fps in the Day 16 room with the animated character; if you drop to 30, shrink the mirror to 256x256 and watch it come back.

**DONE WHEN:** the mirror on the room wall reflects the Day 24 animated character in real time, and `r_vignette 1` darkens the screen corners without costing more than 2 ms.

**IF YOU ARE STUCK:**

- Black mirror: check `glCheckFramebufferStatus` first, then check you restored the viewport — a forgotten `glViewport` after the FBO pass squeezes the whole scene into a 512-pixel corner.
- Mirror shows the room inside out: you forgot `glFrontFace(GL_CW)`, or you set it and forgot to set it back (then the *world* renders inside out instead).
- Reflection swims or misaligns: your mirror pass projection must use the screen aspect ratio, not the FBO's 1:1.
- Everything works but slowly: count your FBO binds per frame. The answer should be 2, plus the default framebuffer.

# Day 26: Water

In 2002, water was the screenshot. Morrowind shipped in May with pixel-shader water ripples and magazine covers followed; every hobby engine forum thread grew a pond within the year. The recipe was an open secret: a reflection rendered to texture, a scrolling normal map to wobble it, and a view-angle blend to make it read as liquid instead of chrome. You built the hard part yesterday. Today is assembly, and by tonight you will have the screenshot you would have posted on a forum in 2002. That is not a figure of speech; this day exists so you can take that screenshot.

## The reflection pass, underwater edition

The water surface is one big quad at height `water_h` with normal (0,1,0), so the Day 25 machinery applies directly: reflect the camera across the plane `y = water_h`, render the world into the reflection FBO with `glFrontFace(GL_CW)`, same screen-aspect projection. Do not draw the water quad itself in this pass.

One new problem: the reflected camera sits *under* the terrain, looking up. Geometry below the waterline — the pond bed, the underwater part of the shore — is now in front of it and gets rendered into the reflection, where it does not belong. Real engines solve this with an oblique near clip plane folded into the projection matrix. It is a clever trick and a numerical minefield, and you do not need it. You have a fragment shader; clip there. Pass the world-space height down as a varying and discard anything below the waterline during the reflection pass:

```
/* vertex shader additions (world pass) */
"varying float v_worldy;\n"
    ...
"    v_worldy = worldpos.y;\n"

/* fragment shader additions */
"uniform float u_clip_y;\n"     /* water_h - 0.05 in the       */
                                /* reflection pass, -10000.0   */
                                /* everywhere else             */
    ...
"    if (v_worldy < u_clip_y) discard;\n"
```

Clip a hair *below* the true waterline (`water_h - 0.05`) so the reflection overlaps the shore slightly; clip exactly at the surface and a thin unreflected seam crawls along the water's edge when the distortion kicks in.

> **NOTE:** `discard` has a real cost on your tile-based GPU: a shader that might discard defeats the hardware's hidden-surface removal for everything it might have covered, so those fragments get shaded anyway. On a big scene you would compile a second shader variant without the discard for normal passes (prepend a `"#define CLIP\n"` line to the same source string — cheap to do with your C string arrays). For this scene, setting `u_clip_y` to -10000 outside the reflection pass and eating the cost is fine. Measure it; the fps counter is right there.

## Wobble and Fresnel

A perfectly flat mirror on the ground reads as glass. Two touches make it water. First, **distortion**: sample a normal map with two UV sets scrolling in different directions at different scales, and use the sampled values to push the reflection lookup around. The classic tool for this was a "dudv map" — a texture of precomputed UV offsets — but a normal map's red and green channels are the same information, and you already know how to paint one from Day 15. Reuse it, or bake a wavy one in Paint Shop Pro from render noise.

Second, **the view-angle blend**. Real water reflects strongly at grazing angles and lets you see into it from above (the Fresnel effect). The full Fresnel equations are overkill; the period approximation is one `pow`: reflection weight = `pow(1`

`- dot(viewDir, up), p)`. Looking across the pond, the dot product is near 0 and the weight is near 1: mirror. Looking straight down, weight near 0: you see a deep blue-green instead.

```
static const char *VS_WATER =
"attribute vec3 a_pos;\n"
"uniform mat4 u_viewproj;\n"
"uniform mat4 u_model;\n"
"varying vec2 v_wuv;\n"
"varying vec3 v_worldpos;\n"
"void main() {\n"
"    vec4 wp = u_model * vec4(a_pos, 1.0);\n"
"    v_worldpos = wp.xyz;\n"
"    v_wuv = wp.xz * 0.125;\n"      /* one tile per 8 units */
"    gl_Position = u_viewproj * wp;\n"
"}\n";

static const char *FS_WATER =
"precision mediump float;\n"
"varying vec2 v_wuv;\n"
"varying vec3 v_worldpos;\n"
"uniform sampler2D u_reflection;\n"
"uniform sampler2D u_normalmap;\n"
"uniform vec2  u_screen;\n"
"uniform vec2  u_scroll1;\n"
"uniform vec2  u_scroll2;\n"
"uniform vec3  u_eye;\n"
"uniform float u_strength;\n"
"uniform float u_fresnel_pow;\n"
"uniform vec3  u_deep;\n"
"void main() {\n"
"    vec3 n1 = texture2D(u_normalmap, v_wuv + u_scroll1).rgb;\n"
"    vec3 n2 = texture2D(u_normalmap,\n"
"                        v_wuv * 1.7 + u_scroll2).rgb;\n"
"    vec2 ripple = (n1.xy + n2.xy - 1.0) * u_strength;\n"
"    vec2 uv = gl_FragCoord.xy / u_screen + ripple;\n"
"    vec3 refl = texture2D(u_reflection, uv).rgb;\n"
"    vec3 view = normalize(u_eye - v_worldpos);\n"
"    float f = pow(1.0 - max(view.y, 0.0), u_fresnel_pow);\n"
"    gl_FragColor = vec4(mix(u_deep, refl, f), 1.0);\n"
"}\n";
```

Note the distortion trick: `(n1.xy + n2.xy - 1.0)` is the two samples recentered from [0,1] to [-1,1] and summed — two wave layers for the price of two fetches, and because they scroll in different directions at different tilings, the interference pattern never visibly repeats. The reflection UV is the Day 25 screen-space lookup plus that ripple. `GL_CLAMP_TO_EDGE` on the reflection texture quietly saves you at the screen edges, where the ripple pushes the UV out of range.

Feed the scroll offsets from your accumulated game time on the C side, and wrap them there — `mediump` UV math gets visibly chunky once the offsets grow past a few dozen:

```
sc1[0] = (float)fmod(g_time * 0.026, 1.0);
sc1[1] = (float)fmod(g_time * 0.018, 1.0);
sc2[0] = (float)fmod(g_time * -0.014 + 1.0, 1.0);
sc2[1] = (float)fmod(g_time * 0.031, 1.0);
```

## Starting values

Water tuning is taste, but taste needs a starting point. These values look like water on the first compile:

| Parameter | Start value | Notes |
|---|---|---|
| reflection FBO | 512 x 512 | 256 if the Atom sweats |
| water height | below your terrain's lowest saddle | so the pond has a shore |
| tile scale | 0.125 | one normal-map repeat per 8 world units |
| u_strength | 0.02 | 0.05 reads as boiling soup |
| u_scroll1 speed | (0.026, 0.018) | units: UV per second |
| u_scroll2 speed | (-0.014, 0.031) | must differ in direction from scroll1 |
| u_fresnel_pow | 4.0 | lower = more mirror, higher = more deep color |
| u_deep | (0.04, 0.13, 0.18) | tint toward your sky for coherence |

Frame order for the whole day: reflection pass into its FBO first (with clip and CW winding), then the main pass — sky, terrain, room, characters — then the water quad last among the opaques, sampling the reflection. If Day 25's postprocess is on, all of that lands in the scene FBO and the vignette runs last. Three FBO binds a frame, each used exactly once: the tiler thanks you. What is left to you: the water quad VBO, the uniform plumbing, and an evening of nudging `u_strength`.

Perf notes: the reflection pass costs roughly one extra terrain draw — cull it with your Day 9/12 machinery exactly like the main pass, and consider skipping the room interior in the reflection entirely if the pond cannot see it. The water shader itself is two texture fetches heavier than anything you have written; at 1024x600 with the pond filling half the screen, expect 2-3 ms on this GPU. Do not render the water in the Day 25 mirror; nobody will notice, and recursion is how young engines die.

**DONE WHEN:** there is a pond on your terrain that reflects the sky and hills, ripples convincingly, goes deep blue-green when you look straight down — and you have taken the screenshot.

**IF YOU ARE STUCK:**

- Reflection upside down or sheared: you reflected the eye but not the look-at point (or the up vector). All three, every frame.
- Shoreline shows a crawling seam: clip slightly below the surface, not at it.
- Water looks like mercury: raise `u_fresnel_pow`, darken `u_deep`, and halve `u_strength` — subtle beats spectacular.
- Ripples turn to stairsteps after a few minutes: you forgot the `fmod` wrap and mediump ran out of fraction bits.
- Everything doubles in cost: you are re-rendering the reflection even when no water is on screen. Frustum-test the pond's AABB first.

# Day 27: Visibility II

Day 9 taught the renderer to skip what is outside the view. Today it learns to skip what is *hidden*. This is the older and harder half of the visibility problem, and in 1999 it defined engine lineages: id built Quake on precomputed visibility baked into BSP trees, while Descent and Unreal walked portal graphs at runtime. You are going to take a third road that neither could in 1999, because it needed hardware support that only arrived at the turn of the millennium: asking the GPU itself "would this box have been visible?" — the occlusion query. Your extension list has `GL_EXT_occlusion_query_boolean`, and it is the primary technique today. Portals get a full description afterward, because for a room-corridor-room level they are the period-correct alternative and you may prefer them.

## Occlusion queries

The idea: after drawing the big stuff that hides things (room walls, terrain), draw an invisible proxy box around each expensive object with a query active. The GPU counts whether *any* fragment of the proxy survived the depth test. If none did, the real object is hidden and you skip it next time. The proxy is drawn with color and depth *writes* off — it must test against the depth buffer without touching anything.

The one rule that separates a working implementation from a uselessly slow one: **never wait for a query result in the same frame you issued it.** The GPU runs a frame or more behind your CPU (ANGLE and the XPDM driver see to that). Asking for the result immediately forces the CPU to sit idle until the GPU catches up — a full pipeline drain, every frame, and your 60 fps becomes 20. This is the classic pitfall of the technique; every engine programmer of the era hit it once. The cure is to consume *last* frame's answer and issue *this* frame's query. Objects appear one frame late when they come around a corner. At 60 fps that is 16 ms of popping. Nobody has ever noticed.

```
frame N  : issue query on crate proxy ..... GPU busy
frame N+1: result ready? yes, zero samples -> crate skipped;
           issue a fresh query
frame N+2: result says visible -> crate drawn again
```

Your `gl_loader.h` exposes the EXT entry points, but confirm at boot with `strstr(glGetString(GL_EXTENSIONS), "GL_EXT_occlusion_query_boolean")` and print to the console. The enums, in case your header lacks them:

```
#define GL_ANY_SAMPLES_PASSED_EXT              0x8C2F
#define GL_ANY_SAMPLES_PASSED_CONSERVATIVE_EXT 0x8D6A
#define GL_QUERY_RESULT_EXT                    0x8866
#define GL_QUERY_RESULT_AVAILABLE_EXT          0x8867

typedef struct occludee_s {
    vec3_t mins, maxs;         /* proxy box, slightly inflated */
    GLuint query;              /* from glGenQueriesEXT at load */
    int    in_flight;
    int    visible;            /* last known answer, start 1  */
} occludee_t;

void R_OcclusionPass(occludee_t *list, int count)
{
    int i;
    GLuint avail, passed;

    /* 1. harvest last frame's answers - never block */
    for (i = 0; i < count; ++i) {
        occludee_t *o = &list[i];
```

```
        if (!o->in_flight)
            continue;
        avail = 0;
        glGetQueryObjectuivEXT(o->query,
                GL_QUERY_RESULT_AVAILABLE_EXT, &avail);
        if (avail) {
            glGetQueryObjectuivEXT(o->query,
                    GL_QUERY_RESULT_EXT, &passed);
            o->visible = (int)passed;
            o->in_flight = 0;
        }
        /* not ready: keep the old answer, do not re-issue */
    }

    /* 2. issue new queries against the primed depth buffer */
    glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
    glDepthMask(GL_FALSE);              /* depth TEST stays on */
    R_UseProgram(g_prog_flat);
    for (i = 0; i < count; ++i) {
        occludee_t *o = &list[i];

        if (o->in_flight)
            continue;
        if (Cam_InsideBox(o->mins, o->maxs)) {
            o->visible = 1;   /* proxy faces would be clipped */
            continue;
        }
        glBeginQueryEXT(GL_ANY_SAMPLES_PASSED_EXT, o->query);
        R_DrawBox(o->mins, o->maxs);
        glEndQueryEXT(GL_ANY_SAMPLES_PASSED_EXT);
        o->in_flight = 1;
    }
    glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
    glDepthMask(GL_TRUE);
}
```

Call this after drawing the occluders (walls, terrain) and before drawing the occludees (crates, the character, the water). Draw an occludee only if `visible` is set — and still run the Day 9 frustum test first; a query costs a draw call, and there is no sense querying what is behind you. The camera-inside check matters: if you stand inside the proxy box, its front faces are behind the near plane, zero fragments pass, and the object you are touching vanishes. Inflate each proxy about 0.25 units beyond the real bounds; it makes objects reappear a frame early instead of a frame late at corner edges. `GL_ANY_SAMPLES_PASSED_CONSERVATIVE_EXT` lets the driver err toward "visible" for speed; on ANGLE it is the same thing, but prefer it anyway — you want conservative semantics on principle.

Add a counter to the Day 6 overlay: `occluded: N`. Standing in room A with the doorway out of sight, every crate in room B should report occluded, and the draw count should fall to the walls plus whatever the frustum already kept.

## The portal alternative

Portals deserve their description, because they are how Descent managed full 6-degree freedom in 1995 and how Unreal's zones worked in 1998, and for a handful of rooms they are arguably cleaner. Model your level as convex-ish *cells* (rooms) connected by *portals* (doorway quads). Render recursively: draw the cell the camera is in; for each portal in it, test the portal against the current view; if visible, draw the neighbor cell with the view *narrowed to the portal*, and recurse. Room B costs nothing when A's doorway is off screen — not "one query and a proxy box" nothing, but actually nothing, and with no frame of latency.

The narrowing is the only technical piece: project the portal's four corners to the screen, take their 2D bounding box, intersect it with the current allowed rectangle. If the intersection is empty, the neighbor cell is skipped. Objects in a cell are then tested against that cell's rectangle (project their bounds the same way).

```
/* project 4 portal corners; out rect in [0,1] screen space.
   Returns 0 if the portal is not usefully on screen. */
int Portal_ScreenRect(vec3_t c[4], const float vp[16],
                      float rect[4] /* x0 y0 x1 y1 */)
{
    int i;

    rect[0] = rect[1] = 1.0f;
    rect[2] = rect[3] = 0.0f;
    for (i = 0; i < 4; ++i) {
        float x, y, w;
        const float *p = c[i];

        x = vp[0]*p[0] + vp[4]*p[1] + vp[8]*p[2]  + vp[12];
        y = vp[1]*p[0] + vp[5]*p[1] + vp[9]*p[2]  + vp[13];
        w = vp[3]*p[0] + vp[7]*p[1] + vp[11]*p[2] + vp[15];
        if (w <= 0.001f) {
            /* a corner is behind the eye: the portal crosses
               the near plane. Punt: use the full screen. */
            rect[0] = 0.0f; rect[1] = 0.0f;
            rect[2] = 1.0f; rect[3] = 1.0f;
            return 1;
        }
        x = x / w * 0.5f + 0.5f;
        y = y / w * 0.5f + 0.5f;
        if (x < rect[0]) rect[0] = x;
        if (y < rect[1]) rect[1] = y;
        if (x > rect[2]) rect[2] = x;
        if (y > rect[3]) rect[3] = y;
    }
    if (rect[0] < 0.0f) rect[0] = 0.0f;
    if (rect[1] < 0.0f) rect[1] = 0.0f;
    if (rect[2] > 1.0f) rect[2] = 1.0f;
    if (rect[3] > 1.0f) rect[3] = 1.0f;
    return rect[0] < rect[2] && rect[1] < rect[3];
}
```

The rest is a recursive function carrying the current rectangle, a cell-adjacency table you author by hand alongside the level, and a depth limit of 4 so a hall of mirrors cannot recurse you into the floor. The behind-the-eye punt is the honest simplification; the rigorous fix is clipping the portal polygon against the near plane before projecting, and it is not worth it for five rooms.

## Why not PVS, the Quake way?

Quake's answer was to move all of this offline. Its map compiler cut the world into a BSP tree of convex leaves, found the portals between leaves automatically, then ran a tool called `vis` that flooded sightlines from every leaf to every other leaf — for each leaf, the Potentially Visible Set: a compressed bitvector saying "from anywhere in leaf 217, you might see leaves 3, 9, 214, 218..." At runtime, visibility was a table lookup and a bit test per leaf. Essentially free, utterly robust, and the reason Quake ran on a Pentium 90. The price: `vis` on a full map took minutes to hours (level designers ran it overnight), it required airtight sealed geometry and a whole compiler toolchain, and the world had to be static. Building that toolchain is a 30-day curriculum by itself, which is why it is out of scope — but now you know what those overnight compiles were buying.

**WARNING:** The one-frame latency rule is not optional. If your fps counter drops the moment you enable queries, you are reading `GL_QUERY_RESULT_EXT` without checking `GL_QUERY_RESULT_AVAILABLE_EXT`. ANGLE will happily stall the whole D3D9 pipeline for you.

Atom notes: each query is a draw call plus bookkeeping through ANGLE, so spend them only on things that cost real money to draw — the skinned character, crate clusters, the water (a skipped water pass skips its whole reflection FBO). Ten to twenty queries a frame is the sweet spot here; two hundred is worse than none.

**DONE WHEN:** standing in room A with the doorway out of view, the overlay shows room B's objects occluded and the draw count near the walls-only floor; walking through the doorway pops them in with no visible hitch and no fps drop.

**IF YOU ARE STUCK:**

- Everything reports occluded: you issued queries before drawing the occluders, so the depth buffer was empty... which would report everything *visible*; if truly all-occluded, your proxy boxes are drawn with depth *test* off... check `glDepthMask(GL_FALSE)` vs `glDisable(GL_DEPTH_TEST)` — you want writes off, test on.
- Objects vanish when you stand next to them: camera-inside-proxy. Add the inside check.
- Flicker every other frame: you re-issue a query while one is in flight and read them out of order. One query object per occludee, one in flight, ever.
- No `glGenQueriesEXT` in your loader: regenerate the loader including the EXT functions; they live in the same libGLESv2.dll.

# Day 28: Game Feel

Load Half-Life and stand still. The weapon sways with your breath; walk, and it bobs in a figure-eight; fire, and the muzzle lights the wall, the sound cracks, and a scorch mark stays behind. None of that is rendering technology — it is a dozen tiny sins of presentation, and together they are why 1998's games suddenly felt *inhabited*. Today you commit all of them: a viewmodel, weapon bob, a crosshair, hitscan shooting with decals, particles, light, and a bang. Individually each is an hour. This is the day your tech demo becomes a game.

## The viewmodel and the depth-clear trick

The weapon in the corner of the screen is a normal mesh (model one in Milkshape, ten minutes, nobody looks closely) rendered with two period tricks. First: draw it **last**, after the entire world, with the depth buffer cleared first — `glClear(GL_DEPTH_BUFFER_BIT)` and nothing else. The gun always wins the depth test, so walking face-first into a wall never pushes the barrel through the bricks. Every engine of the era did exactly this. Second: give it its own projection with a narrower FOV (54 degrees against your 75-degree world) so the gun does not stretch at the screen edge, and its own near plane at 0.05 so it can sit close to the camera.

The viewmodel skips the view matrix entirely — it lives in camera space, welded to your face:

```
void R_DrawViewmodel(void)
{
    float proj[16], model[16], t1[16], t2[16], mvp[16];

    glClear(GL_DEPTH_BUFFER_BIT);        /* the classic trick */
    mat4_perspective(proj, 54.0f,
                     (float)g_width / (float)g_height,
                     0.05f, 20.0f);
    mat4_translate(t1, 0.16f + g_bob_x, -0.14f + g_bob_y, -0.40f);
    mat4_rotate_z(t2, g_bob_r);
    mat4_multiply(model, t1, t2);
    mat4_multiply(mvp, proj, model);
    R_DrawMesh(g_gun_mesh, mvp);
}
```

## Bob: trigonometry of walking

Weapon bob is driven by *distance walked*, not by time — that single choice is why good bob speeds up when you run and stops when you stop. Accumulate horizontal speed into a phase, then derive offsets. The vertical uses `fabs(sin)` so it dips twice per cycle — once per footfall; the horizontal sways once per cycle; a little roll sells the whole thing. A 0-to-1 envelope fades the bob in and out so it does not snap when you halt:

```
#define BOB_RATE  1.6f     /* phase radians per unit walked */
#define BOB_AMP_X 0.012f   /* sideways sway, camera units   */
#define BOB_AMP_Y 0.016f   /* vertical dip                  */
#define BOB_AMP_R 1.5f     /* roll, degrees                 */

/* in engine_update, after the day-22 move */
if (g_player.on_ground) {
    float speed;
    vec3_t hv;

    hv[0] = g_player.vel[0];
    hv[1] = 0.0f;
    hv[2] = g_player.vel[2];
```

```
    speed = (float)sqrt(DotProduct(hv, hv));
    g_bob_t += speed * dt * BOB_RATE;
    g_bob_env += ((speed > 0.5f) ? 4.0f : -6.0f) * dt;
    if (g_bob_env < 0.0f) g_bob_env = 0.0f;
    if (g_bob_env > 1.0f) g_bob_env = 1.0f;
}
g_bob_x =  (float)cos(g_bob_t) * BOB_AMP_X * g_bob_env;
g_bob_y = -(float)fabs(sin(g_bob_t)) * BOB_AMP_Y * g_bob_env;
g_bob_r =  (float)sin(g_bob_t) * BOB_AMP_R * g_bob_env;
```

Tune by feel: double an amplitude, walk a lap, halve it. Most first-timers over-bob by 3x; the numbers above are already on the subtle side of period-correct. If you also add a touch of `g_bob_y` to the camera itself, keep it to a third of the weapon's or your testers will report seasickness. The crosshair is Day 6 material: two 8-pixel lines or a single font glyph, drawn at screen center in the overlay pass. Do not skip it; without a crosshair, hitscan feels broken even when it is perfect.

## Hitscan: the ray and the slabs

Firing is a ray from the camera through the screen center — which is exactly the camera's forward axis, already sitting in your view matrix. Column-major, the world-space forward is the negated third row of the rotation part:

```
dir[0] = -g_view[2];
dir[1] = -g_view[6];
dir[2] = -g_view[10];
```

Test it against every world AABB from Day 22, keep the nearest hit. The standard test is the *slab method*: an AABB is the intersection of three axis-aligned slabs; clip the ray's parameter interval [tmin, tmax] against each slab and if the interval survives, you hit. Track which axis produced the final tmin and you get the hit normal for free — you need it for the decal:

```
#define MAX_TRACE 8192.0f

int Ray_AABB(const vec3_t org, const vec3_t dir,
             const vec3_t mins, const vec3_t maxs,
             float *t_hit, vec3_t normal)
{
    float tmin, tmax, sign;
    int i, axis;

    tmin = 0.0f;
    tmax = MAX_TRACE;
    axis = -1;
    sign = 0.0f;

    for (i = 0; i < 3; ++i) {
        float t1, t2, inv;

        if (dir[i] > -0.000001f && dir[i] < 0.000001f) {
            if (org[i] < mins[i] || org[i] > maxs[i])
                return 0;        /* parallel and outside slab */
            continue;
        }
        inv = 1.0f / dir[i];
        t1 = (mins[i] - org[i]) * inv;
        t2 = (maxs[i] - org[i]) * inv;
        if (t1 > t2) {
            float tmp;

            tmp = t1; t1 = t2; t2 = tmp;
```

```
        }
        if (t1 > tmin) {
            tmin = t1;
            axis = i;
            sign = (dir[i] < 0.0f) ? 1.0f : -1.0f;
        }
        if (t2 < tmax)
            tmax = t2;
        if (tmin > tmax)
            return 0;
    }
    if (axis < 0)
        return 0;               /* ray started inside the box */
    *t_hit = tmin;
    normal[0] = normal[1] = normal[2] = 0.0f;
    normal[axis] = sign;
    return 1;
}
```

## Impact: decal, particles, light, bang

On a hit, four things happen in the same frame, and the simultaneity is the entire effect.

**The scorch decal** is a dark textured quad (paint a soft black splat with alpha in Paint Shop Pro) placed at the hit point, pushed 0.01 units along the hit normal so it never z-fights the wall. Build its orientation from the normal: pick a helper axis that is not parallel to it, cross to get the first edge, cross again for the second:

```
vec3_t up, s, t;

if (fabs(n[1]) < 0.9f) {
    up[0] = 0.0f; up[1] = 1.0f; up[2] = 0.0f;
} else {
    up[0] = 1.0f; up[1] = 0.0f; up[2] = 0.0f;
}
CrossProduct(n, up, s);
VectorNormalize(s);
CrossProduct(n, s, t);
/* corners: hit + n*0.01 +/- s*r +/- t*r, r ~ 0.15 */
```

Store decals in a fixed ring buffer of 64; slot 65 overwrites slot 1, the Quake way — no allocation, no cleanup, no leak. Draw them alpha-blended after the opaque world. **The particle burst** is your Day 18 spark emitter: 12 particles, initial velocity along the normal spread by a random cone, gravity on, 0.4-second life. **The muzzle flash** is a timer: on fire, set `g_flash_frames = 3`; while nonzero, add a warm point light (Day 14, color 1.0/0.8/0.4, radius 6) at the muzzle and draw a small additive-blended sprite at the barrel tip, then decrement in update. Three frames at 60 Hz is 50 ms — the flicker is the point. **The bang** is Day 21: `S_Play("bang.wav")`, no distance attenuation on your own gunshot. If the four do not land on the same frame, the shot feels mushy; that is also your debugging signal.

What is left to you: the fire input edge-detect (fire on press, not per-frame while held), the decal ring buffer, and the crosshair quad. Perf is a non-issue today — everything here is a handful of draws; the whole day costs under a millisecond. Feel is free. That was the 1998 secret.

**DONE WHEN:** shooting a crate leaves a scorch quad stuck to the right face at the right spot, with a spark burst, a one-flash light on the walls, and a bang — and the gun bobs as you walk to the next crate.

**IF YOU ARE STUCK:**

- Decal on the wrong face or floating mid-air: your ray direction is the view matrix's third row, not the negated third row — you are shooting backward. Print `t_hit`; a huge value means a miss being treated as a hit.
- Decal z-fights: 0.01 offset is in world units — if your crates are 100 units wide, scale it accordingly.
- Gun disappears at screen edges: your viewmodel is being frustum culled by the world's culler. Exempt it; it has its own projection.
- Bob stutters: you are accumulating phase in render, not update. Fixed timestep things live in update.
- Gun looks flat: light the viewmodel with the same sun direction uniform as the world; a differently-lit gun reads as pasted-on.

# Day 29: The Level

You have spent 28 days building organs. Today you build the body. Every real engine of the era had two things yours still lacks: a level format — some way for content to exist outside the compiler — and a profiling culture, the `r_speeds` habit of watching the numbers while you play. Quake's designers lived in text files and counters; the .map format was human-readable text right up until the compiler ate it. You will do the same at one-tenth scale: a text format of your own, a real level authored in it, and then a profiling pass where you hunt the frame's biggest number and kill it.

## A level format of your own

Resist the urge to design a binary format with a header and a version field. It is Day 29; you need a text file you can edit in Notepad on the netbook, one entity per line, first word says what it is. Here is the entire specification, by example:

```
# pond.lvl - first real level. '#' starts a comment.
music    canyon.xm
sky      dusk
fog      0.015  180 160 150
terrain  valley.png  0.35 40.0
water    -2.5
spawn    12.0 5.0 34.0   90
wall     10 0 20   4 3 0.5   brick.jpg
wall     14 0 20   0.5 3 6   brick.jpg
crate    15 1 22   1 1 1     crate.jpg
light    14 2.5 21   255 200 120   8
```

`wall x y z sx sy sz texture` places an AABB brush: center, half-extents, texture — it feeds both the renderer and the Day 22 collision list from one line. `light` is position, color, radius. `spawn` is position plus yaw. `water` is just a height; the pond quad sizes itself to the terrain. `crate` is a wall that registers as shootable and occludable. That is the whole grammar. The parser is `fgets` plus `sscanf`, with the `%*s` trick to skip the keyword you already matched:

```
int Level_Load(const char *path)
{
    FILE *f;
    char line[256], key[32];

    f = fopen(path, "r");
    if (!f) {
        Con_Printf("Level_Load: no %s\n", path);
        return 0;
    }
    while (fgets(line, sizeof(line), f)) {
        if (line[0] == '#' || line[0] == '\n')
            continue;
        if (sscanf(line, "%31s", key) != 1)
            continue;
        if (!strcmp(key, "wall") || !strcmp(key, "crate")) {
            float x, y, z, sx, sy, sz;
            char tex[64];

            if (sscanf(line, "%*s %f %f %f %f %f %f %63s",
                       &x, &y, &z, &sx, &sy, &sz, tex) == 7)
                Level_AddBrush(x, y, z, sx, sy, sz, tex,
                               key[0] == 'c');
        } else if (!strcmp(key, "light")) {
            float x, y, z, r, g, b, rad;

            if (sscanf(line, "%*s %f %f %f %f %f %f %f",
```

```
                        &x, &y, &z, &r, &g, &b, &rad) == 7)
                Level_AddLight(x, y, z,
                               r / 255.0f, g / 255.0f,
                               b / 255.0f, rad);
        } else if (!strcmp(key, "water")) {
            sscanf(line, "%*s %f", &g_level.water_h);
        } else if (!strcmp(key, "music")) {
            sscanf(line, "%*s %63s", g_level.music);
        } else {
            Con_Printf("Level_Load: unknown '%s'\n", key);
        }
    }
    fclose(f);
    return 1;
}
```

(The `spawn`, `sky`, `fog`, and `terrain` branches follow the same pattern and are left to you.) Loading order matters and should be boringly explicit: parse `engine.cfg`; parse the level file into plain structs (no GL calls yet); resolve every texture, mesh, and sound through the Day 20 refcounted manager; build the collision and occludee lists; upload GL buffers; start the music; place the player at the spawn. Wire it to the Day 19 console's `map` command, and make `map` tear the old level down first — your Day 20 leak counter proves the teardown, and loading `pond.lvl` fifty times in a row from the console is tonight's soak test. Then spend an hour actually *designing*: a shore worth walking, the room placed so its doorway frames the water, crates arranged so Day 27 has something to occlude. Author with the console open and numbers on screen.

## The profiling pass

Which brings us to the numbers. Extend the Day 6 overlay into a proper counter block, reset at the top of each frame, incremented at the choke points: every draw call site bumps `draws` and adds to `tris`; your texture-bind wrapper bumps `binds` only when the texture actually changes; the particle system reports its live count; Day 27 reports skips. Time the three phases of the frame with QueryPerformanceCounter, which you already trust from the main loop:

```
typedef struct prof_s {
    double update_ms, render_ms, present_ms;
    int draws, tris, binds, particles, occluded;
} prof_t;
extern prof_t g_prof;

double Sys_Ms(void)
{
    LARGE_INTEGER t;

    QueryPerformanceCounter(&t);
    return (double)t.QuadPart * g_ms_per_tick;  /* from boot */
}

/* in the main loop */
t0 = Sys_Ms(); Engine_Update(1.0 / 60.0);
t1 = Sys_Ms(); Engine_Render();
t2 = Sys_Ms(); eglSwapBuffers(g_dpy, g_surf);
t3 = Sys_Ms();
g_prof.update_ms  = t1 - t0;
g_prof.render_ms  = t2 - t1;
g_prof.present_ms = t3 - t2;
```

Overlay it as two dense lines, Quake-style:

```
draws 214  tris 96k  binds 143  parts 380  occl 9
upd 1.8ms  rnd 19.5ms  swp 0.4ms
```

Read it with vsync in mind: with `eglSwapInterval(1)` on XP's driver model, missed frames quantize hard — 60, 30, 20, 15. If update+render crosses 16.6 ms even slightly, the fps counter slams to 30. A big `present_ms` with small update and render is the swap *waiting* for vblank: that is headroom, not cost. Find the biggest honest number and attack that; never optimize without a counter pointing at the target.

## One optimization, worked

Here is how this played out on the author's build of this level, so you know the shape of the hunt. Symptom: fps counter pinned at 30. Counters: `rnd 19.5ms`, so render is the culprit; `draws 214` is modest, but `binds 143` stands out — two of every three draws changes texture. Cause: the scene graph traversal emits draws in tree order, so wall, crate, wall, character, wall ping-pongs between textures. On real GL this would be mildly rude; through ANGLE every bind becomes a D3D9 `SetTexture` with validation, and under the XPDM driver each state change has real CPU cost in the runtime. Death by a hundred and forty paper cuts.

Fix: stop drawing during traversal. Collect draw records (program, texture, VBO, matrix) into an array, qsort by state, then execute in order:

```
static int DrawCmp(const void *a, const void *b)
{
    const drawcmd_t *da = (const drawcmd_t *)a;
    const drawcmd_t *db = (const drawcmd_t *)b;

    if (da->program != db->program)
        return (int)da->program - (int)db->program;
    return (int)da->texture - (int)db->texture;
}
```

Result: `binds 143` fell to 31, `rnd` from 19.5 ms to 13.8, and the quantizer snapped back to 60. Fifteen lines of code, found in ten minutes, because the counter existed. That is the lesson of the day, worth more than the code: the overlay is not decoration, it is the instrument panel. Your bottleneck will be different — maybe particle fill rate, maybe the water pass running when the pond is behind you. Hunt it the same way, and write the numbers in journal.txt, before and after.

**DONE WHEN:** your authored level plays for two continuous minutes — sky, terrain, water, room, character, music, shooting — with no crash, no out-of-memory, and no fps collapse, and your journal records one bottleneck found by counter and fixed with numbers to prove it.

**IF YOU ARE STUCK:**

- Crash after several `map` reloads: something bypasses the resource manager. The Day 20 leak counter knows which pool is growing.
- sscanf silently skips lines: check every sscanf's return value against the field count you demanded, and print the offending line when it falls short — a stray character mid-line fails the whole pattern without a peep.
- Counters read zero: you reset them after render instead of before update.
- fps says 30 but both phases sum under 10 ms: something mid-frame is forcing a GPU sync — a Day 27 query read without the available check, or a glReadPixels you left in.

# Day 30: Ship It

An engine that only runs on its author's machine is a diary. Today it becomes a demo: named, versioned, packaged in a zip, and proven on a clean login — the 1999 shareware-disc standard, where a stranger with no compiler and no help gets to walk around your level. Shipping is a discipline distinct from programming, and most hobby engines die precisely here. Yours will not, because today has a checklist.

## The Release build

You have been living in the Debug configuration for a month. In VC6: Build, Set Active Configuration, "Win32 Release". Then Project Settings (Alt+F7), and verify, do not assume:

- C/C++, Optimizations: **Maximize Speed (/O2)**.
- Confirm **/GZ is gone** from the compile line. It is the Debug-only runtime-check switch (stack probes, 0xCC fill); it hides uninitialized-variable bugs and costs speed. If it ever leaked into Release via copied settings, remove it.
- C/C++, Code Generation, Runtime library: **Multithreaded (/MT)** — statically linked, so the exe carries its own C runtime and you owe the user no compiler DLLs.
- Preprocessor definitions: `NDEBUG`, not `_DEBUG`.
- Link: incremental linking off, debug info off.

Then Rebuild All and **play the whole level again in Release**. The Debug build's stack fill has been quietly zero-ish-initializing your mistakes for a month; /O2 will now let every uninitialized local express itself. If Release misbehaves and Debug does not, suspect your uninitialized memory first and VC6's optimizer second — in that order, but this compiler is old enough that the second is not paranoia. Dropping one file to /O1 is a legitimate period fix.

## The DLL manifest

Everything the exe needs at runtime, next to the exe. This is the complete list — tape it to the monitor:

| File | What it is | From where |
|---|---|---|
| libEGL.dll | ANGLE's EGL front door | the Firefox 52 ESR directory |
| libGLESv2.dll | ANGLE: GLES2 to D3D9 translator | same place |
| d3dcompiler_43.dll | HLSL compiler ANGLE calls to build every shader you compile at runtime | same place — take whichever d3dcompiler_*.dll sits next to libGLESv2.dll |
| bass.dll | audio | the BASS 2.4 package |

d3d9.dll is part of Windows; with /MT there is no C runtime DLL to ship. Verify with Dependency Walker on the packaged exe — but know its blind spot: d3dcompiler_43.dll is loaded by ANGLE at runtime via LoadLibrary, so Depends will *not* list it. Forget it anyway and the symptom is a window that opens and stays black while every shader compile fails. The only trustworthy test is the clean-login protocol below.

## Loading screen, version, readme

Level loading takes several seconds on this SSD-less-era hardware, and a frozen white window reads as a crash. The fix costs one quad: immediately after GL init and before any loading, draw your Paint Shop Pro splash and swap once. Pump the message queue once first so XP actually maps the window:

```
void Loading_Screen(void)
{
    GLuint tex;
    MSG msg;

    while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
        TranslateMessage(&msg);
        DispatchMessage(&msg);
    }
    tex = Texture_Load("data/splash.jpg");
    glViewport(0, 0, g_width, g_height);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    R_DrawQuad2D(tex, 0, 0, g_width, g_height);
    eglSwapBuffers(g_dpy, g_surf);
}
```

Name the engine — you have earned the naming rights — and put the version where users and bug reports can see it: `#define ENG_VERSION "pondwerks 1.0 (" __DATE__ ")"`, printed at console boot and drawn small in a screen corner. When someone reports "it crashes", your first question is answered by the screenshot. The readme is a genre with rules; honor them:

```
============================================================
   PONDWERKS v1.0
   a 3d engine demo, built from scratch in 30 days
============================================================

WHAT IS THIS?
   Terrain, a pond that reflects the sky, a room with baked
   light, a character who walks, and a gun that goes bang.
   Every line of it written by hand. Walk around. Press keys.

REQUIREMENTS
   Windows XP. A Direct3D 9 capable video card. 256 MB RAM.
   Unzip EVERYTHING to one folder. The DLLs must stay next
   to PONDWERKS.EXE or you will meet a rude message box.

CONTROLS
   W A S D ...... move           MOUSE ....... look
   SPACE ........ jump           LMB ......... fire
   ~ ............ console        ESC ......... quit

TROUBLESHOOTING
   "libEGL.dll not found" ... you unzipped only the exe.
   Black screen ............. d3dcompiler dll is missing.
   No sound ................. bass.dll is missing.
   Type "fog 0.03" in the console. Trust me.

CREDITS
   Code, music, textures: the author. Thanks to stb_image,
   BASS, and a 1998 compiler that refused to die.
```

## The zip and the clean-login protocol

```
pondwerks_10.zip
    pondwerks.exe
    libEGL.dll
    libGLESv2.dll
    d3dcompiler_43.dll
    bass.dll
    readme.txt
```

```
    engine.cfg
    data/
        levels/pond.lvl
        textures/  models/  music/  sounds/
        splash.jpg
```

Now prove it, the way a magazine cover-disc tester would have:

1. Control Panel, User Accounts: create a new limited user, "tester". Log off. Log in as tester — a profile that has never seen VC6, your PATH, or your working directory.
2. Copy the zip to tester's desktop. Extract with XP's built-in zip folder, not your archiver.
3. Double-click the exe. No README consulted, because your player will not consult it either.
4. Watch for the classic failures: a missing-DLL message box (you shipped a partial manifest); files not found (your code assumes the working directory is the exe's — it is not guaranteed; derive the base path from `GetModuleFileNameA` and load everything relative to it); a write failure (a limited user cannot write to Program Files — write journal, config, and screenshots next to the exe or not at all).
5. Play the full two minutes. Then find an actual other human being and repeat step 3 over their shoulder, hands off.

## What you built, and what they built

Set your zip next to the giants it imitates. Quake III Arena shipped in December 1999 with BSP levels and precomputed visibility, curved surfaces, a shader script system, and client-server netcode that still holds up — all running on fixed-function multitexture hardware, because programmable pixel shaders did not exist for consumers yet. Unreal Tournament, same month: portal-zoned levels, a software renderer that shamed most hardware, UnrealScript, and UnrealEd shipping in the box. By 2002 the screenshot wars were about water and per-pixel light — Morrowind's ripples, and a leaked Doom 3 build showing unified stencil-shadowed lighting.

Your thirty days produced: a math library and renderer built from GL function pointers up; texture, mesh, and model pipelines across four formats; scene graph, frustum and occlusion culling; per-pixel lighting, normal maps, baked lightmaps, fog; terrain, sky, particles, water with real-time reflections; two animation systems; collision and player physics; a console with cvars, a resource manager, an audio layer, a level format, and a shipped zip. Feature for rendering feature, you are not embarrassed next to 1999's finest — you have programmable shaders they could only fake with register combiners, and your water is honestly better than anything id shipped in the period.

What they had, and you do not, is everything around the renderer: level editors artists could live in, content pipelines measured in artist-years, script VMs, netcode hardened by a million dial-up players, and QA departments. That is the honest lesson of the recreation: the renderer was never the moat. The moat was tools and content — and that is exactly why the extra credit list below starts pushing you toward tools, data, and other machines.

## New game plus: the extra credit list

### Stencil shadows

The Doom 3 look, demonstrated on a Mac stage in 2001 and the reason "Carmack's reverse" entered the vocabulary. First, plumbing: ask EGL for stencil at config-choose time (`EGL_STENCIL_SIZE, 8`) and confirm what you got with `eglGetConfigAttrib` — on this stack the window surface will give you depth24-stencil8 via the packed format, but never assume; for FBO passes you would need `OES_packed_depth_stencil`, which is on your list. The algorithm: for each shadow-casting mesh, find silhouette edges — edges where one adjacent triangle faces the light and the other does not (precompute edge adjacency at load). Extrude those edges away from the light to form a shadow volume, then render it into the stencil buffer only: front faces increment, back faces decrement. Pixels left nonzero are inside some volume — in shadow — and a darkening pass masked by stencil finishes the job. When the camera itself sits inside a volume, depth-pass counting breaks; the fix is Z-fail counting with capped volumes, the reverse in question.

On this GPU, be modest. Shadow volumes are enormous invisible triangles, and a tile-based renderer pays for every covered pixel in binning and stencil traffic; the technique was infamous for fill-rate cost even on 2004's cards. Budget one hero caster — the Day 24 character, about 600 triangles, whose silhouette the Atom can find per-frame without noticing — onto floor-and-walls receivers, sun only. Get the classic sharp-edged wrongness of it, journal the fill cost, and you will understand both why Doom 3 looked the way it did and why everyone moved to shadow maps within five years.

### Geomipmapped terrain LOD

Your Day 12 chunks all render at full resolution whether they are two meters away or two hundred. Geomipmapping (the name is from a 2000 paper, and every flight-sim era terrain did some cousin of it) gives each 33x33 chunk a ladder of detail levels — 33, 17, 9, 5 vertices per side, each level skipping every other vertex of the last — and picks a level per chunk per frame by distance. Implementation is friendlier than it sounds: the vertex buffer never changes; each level is just a different index buffer over the same vertices, so a chunk switch is one index-offset swap. Choose the level from the chunk's max height-error projected to screen pixels (precompute each level's worst-case vertical error at bake time; switch when the projected error would exceed about two pixels), and add hysteresis so a chunk straddling a threshold does not flicker between levels.

The seams are the famous problem: adjacent chunks at different levels crack along shared edges. The industrial-strength fix is stitching strips; the pragmatic 2000-era fix is *skirts* — a curtain of triangles hung a meter down from each chunk's perimeter, textured like the terrain, hiding any crack from any sane camera angle. Skirts cost a few hundred triangles per chunk and zero code complexity at level boundaries, which is why real shipping terrain used them for a decade. Payoff on the Atom: distant chunks drop from 2048 triangles to 32, and you can quadruple the view distance from Day 17's fog wall at the same frame cost. Journal the before and after triangle counts; they are absurd in the best way.

### Quake3-style material scripts

Right now a surface is a texture. In Quake III a surface was a *shader*: a named block in a text script listing render stages — a lightmap stage, a scrolling-texture stage, an additive glow stage — each with its own blend mode and UV animation. Artists invented pulsing lights and flowing lava without a programmer in the room, which is the entire point: it moved surface appearance from code to data. Sketch of yours: a `scripts/*.mtr` parser (your Day 29 tokenizer habits transfer directly) filling a material struct: N stages, each with a texture, blend func, and a `tcmod scroll u v`. Start with exactly five keywords: `map`, `blendfunc` (add, blend, filter), `tcmod scroll`, `rgbgen wave sin`, and `cull none`. Each stage becomes one draw pass; your GLES2 shader already has the uniforms for all of it.

The engine change that matters is not the parser, it is the plumbing: the Day 29 draw-record array now sorts by *material* instead of raw texture, meshes reference materials by name, and the resource manager owns the script table with a console command to reload it live. Live reload is the killer feature — edit the scroll rate in Notepad, type `mat_reload`, and watch the lava speed up without a recompile. The day you do that is the day you stop being a renderer author and start being an engine author; tools and data turnaround were always the real product.

### A second player: UDP LAN sync

The 2001 way: WinSock (`wsock32.lib`, and plain blocking-free UDP via `ioctlsocket` FIONBIO), one machine as server with a fixed 20 Hz tick, discovery by broadcasting a magic packet to port 27960 on 255.255.255.255 and listening for a reply. The client sends a small input packet every tick — buttons, view angles, a sequence number; the server simulates and replies with a snapshot: every entity's position and yaw, positions quantized to 16-bit shorts to keep a packet under 100 bytes. All structs little-endian, packed by hand with explicit byte writes (never `fwrite` a struct across a network boundary), a version byte first so two builds refuse each other politely.

The design insight that makes this feasible in a weekend: with few entities, make every snapshot the *full* state. Then packet loss needs no repair protocol — the next snapshot heals everything, and sequence numbers exist only to discard stale reordered packets. The client renders 100 ms behind the newest snapshot and interpolates between the two straddling ones, exactly Quake 3's client model minus prediction; movement feels one beat laggy on the trigger, and that

is authentically 2001 modem-lord behavior on a LAN with zero loss. Add client-side prediction of your own player later if it bothers you. Test with the netbook and any second XP box on the cabin's switch; watching your own character walk on someone else's screen is a rite the internet era has cheapened, and it still hits.

### The dual-target build

The final flex from CHALLENGES.txt: the same C89 codebase running on the XP netbook and the Arch machine. You are closer than you think, because the curriculum accidentally enforced the discipline: your GL code is GLES2-core through function pointers, your types are C89, your files are little-endian with explicit reads. What remains is quarantining the platform layer: everything that touches Win32 — window creation, raw input, QueryPerformanceCounter, MessageBoxA — moves behind a small `sys_*` interface with two implementations, `sys_win.c` and `sys_linux.c`. On Linux, EGL speaks to X11 natively, so your EGL code barely changes; the loader dlopens the system GLES libraries; the timer is `clock_gettime`. BASS ships a Linux build, or you stub audio behind the Day 21 wrapper you already have. The gotchas are all mundane: `_snprintf` vs `snprintf` behind a #define, backslash paths, and Linux's case-sensitive filenames auditing a month of casual `Data/Textures` spelling.

The cross-compiler makes it stranger and better: mingw-w64 on the Arch box can build the XP executable, so one modern machine builds both centuries — keep a `build.sh` that produces the Linux binary and the XP exe side by side, and the VC6 .dsp becomes a museum piece you still keep honest. This is the extra credit with the longest half-life. The netbook will die; XP already has. A codebase with a two-line platform seam and no API assumptions is the one thing in this whole month that ports to whatever comes next. One engine, two centuries — and the second century is the one your engine gets to grow old in.

**DONE WHEN:** someone who is not you, on a clean XP login, unzips your file, double-clicks the exe with zero coaching, and walks around your level — and the readme, splash, and version string are all wearing the engine's name.

**IF YOU ARE STUCK:**

- Works in your account, black screen for tester: the runtime-loaded d3dcompiler DLL is missing from the zip — Dependency Walker cannot see it; only the clean test can.
- "Ordinal not found" on launch: you shipped a libEGL.dll and libGLESv2.dll from different Firefox versions. They travel as a pair.
- Assets not found from a desktop shortcut: stop trusting the working directory; base every path on GetModuleFileNameA.
- Release crashes where Debug never did: uninitialized local. Read the function twice; it is there.
- The tester keeps walking into the pond and grinning instead of filing bugs: ship it. That is the sign-off.

# Appendix A: OpenGL ES 2.0 Reference

This is your man page. Every core GLES2 entry point reaches your code as a function pointer declared in `gl_loader.h`, named exactly like the real function, and filled in by `gl_loader_init()`. Extension functions are *not* in the loader: fetch them yourself with `eglGetProcAddress` after checking the extension string (see section A.9). Everything below exists on the ANGLE-on-EMGD stack in this book; nothing else does.

> **NOTE:** After any suspect call, ask `glGetError()`. It returns and clears one error: `GL_NO_ERROR`, `GL_INVALID_ENUM` (bad enum value), `GL_INVALID_VALUE` (bad number: negative size, out-of-range index), `GL_INVALID_OPERATION` (call illegal in current state), `GL_INVALID_FRAMEBUFFER_OPERATION` (drawing into an incomplete FBO), `GL_OUT_OF_MEMORY`. Errors are cheap to check on this driver during bring-up; wrap it in a macro you can compile out.

## A.1 State, clears, and whole-pipe controls

| Signature | What it does | Errors / notes |
|---|---|---|
| `void glEnable(GLenum cap)` / `glDisable(GLenum cap)` | Turn a capability on/off (see capability table, A.8). | INVALID_ENUM on desktop-GL caps like GL_TEXTURE_2D: texturing is always "on" in shaders. |
| `GLboolean glIsEnabled(GLenum cap)` | Query a capability. | |
| `void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)` | Maps clip space to window pixels. Origin bottom-left. | Not saved per-FBO: reset it every time you switch render targets. |
| `void glScissor(GLint x, GLint y, GLsizei w, GLsizei h)` | Clips draws AND clears to the box when GL_SCISSOR_TEST is on. | Origin bottom-left: flip y coming from Win32/nuklear (Appendix D). |
| `void glClearColor(GLclampf r, GLclampf g, GLclampf b, GLclampf a)` | Sets the color clear value. | |
| `void glClearDepthf(GLclampf d)` | Sets the depth clear value (default 1.0). | |
| `void glClearStencil(GLint s)` | Sets the stencil clear value. | |
| `void glClear(GLbitfield mask)` | Clears buffers: OR of GL_COLOR_BUFFER_BIT, GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT. | Clear depth+color together in ONE call; two calls costs two passes on this driver. Respects glColorMask/ glDepthMask and scissor. |
| `void glColorMask(GLboolean r, GLboolean g, GLboolean b, GLboolean a)` | Enables/disables writes per channel. | All-false + depth on = depth-only pass. |
| `void glDepthMask(GLboolean flag)` | Enables/disables depth writes. | Off for skybox (day 10) and blended particles (day 18). |
| `void glDepthFunc(GLenum func)` | Depth compare: GL_NEVER, GL_LESS (default), GL_EQUAL, GL_LEQUAL, GL_GREATER, GL_NOTEQUAL, GL_GEQUAL, GL_ALWAYS. | Depth test still needs glEnable(GL_DEPTH_TEST) and EGL_DEPTH_SIZE at config time. |

| | | |
|---|---|---|
| `void glDepthRangef(GLclampf n, GLclampf f)` | Maps NDC z to window depth (default 0..1). | Handy trick: 0..0.1 range for the weapon viewmodel (day 28). |
| `void glCullFace(GLenum mode)` | GL_FRONT, GL_BACK (default), GL_FRONT_AND_BACK. | Needs glEnable(GL_CULL_FACE). |
| `void glFrontFace(GLenum mode)` | GL_CCW (default) or GL_CW winding = front. | Mirrored geometry (day 25 reflections) flips winding: flip this too. |
| `void glBlendFunc(GLenum src, GLenum dst)` | Sets blend factors for both RGB and alpha. | Needs glEnable(GL_BLEND). Factor table in A.8. |
| `void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB, GLenum srcA, GLenum dstA)` | Separate factors for RGB and alpha. | |
| `void glBlendEquation(GLenum mode)` | GL_FUNC_ADD (default), GL_FUNC_SUBTRACT, GL_FUNC_REVERSE_SUBTRACT; GL_MIN_EXT/GL_MAX_EXT via EXT_blend_minmax. | |
| `void glBlendEquationSeparate(GLenum rgb, GLenum a)` | Separate equations for RGB and alpha. | |
| `void glBlendColor(GLclampf r, GLclampf g, GLclampf b, GLclampf a)` | Constant color for GL_CONSTANT_* factors. | |
| `void glStencilFunc(GLenum func, GLint ref, GLuint mask)` / `glStencilFuncSeparate(GLenum face, ...)` | Stencil compare (same funcs as depth). | Needs stencil bits in the EGL config: ask for EGL_STENCIL_SIZE 8. |
| `void glStencilOp(GLenum sfail, GLenum dpfail, GLenum dppass)` / `glStencilOpSeparate(GLenum face, ...)` | What to do on stencil fail / depth fail / pass: GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_INCR_WRAP, GL_DECR, GL_DECR_WRAP, GL_INVERT. | |
| `void glStencilMask(GLuint mask)` / `glStencilMaskSeparate(GLenum face, GLuint mask)` | Write mask for stencil bits. | |
| `void glPolygonOffset(GLfloat factor, GLfloat units)` | Biases depth of filled triangles. | Needs glEnable(GL_POLYGON_OFFSET_FILL). Use for decals (day 28): try factor -1, units -2. |
| `void glLineWidth(GLfloat w)` | Width for GL_LINES* primitives. | Only width 1 is guaranteed; ANGLE/ D3D9 draws 1 regardless. |
| `void glSampleCoverage(GLclampf value, GLboolean invert)` | MSAA coverage tweak. | You have no multisampled window surface; ignore. |
| `void glPixelStorei(GLenum pname, GLint param)` | GL_UNPACK_ALIGNMENT / GL_PACK_ALIGNMENT: 1, 2, 4, 8. | Default is 4. RGB images with odd widths upload garbage until you set |

| | | |
|---|---|---|
| | | UNPACK_ALIGNMENT to 1. This is the day-4 bug. |
| `void glHint(GLenum target, GLenum mode)` | GL_GENERATE_MIPMAP_HINT: GL_FASTEST/GL_NICEST/ GL_DONT_CARE. | |
| `void glReadPixels(GLint x, GLint y, GLsizei w, GLsizei h, GLenum format, GLenum type, void *pixels)` | Reads the current framebuffer into client memory. | Only GL_RGBA + GL_UNSIGNED_BYTE is guaranteed. Stalls the whole pipe; screenshots only. |
| `GLenum glGetError(void)` | Returns and clears one pending error. | |
| `const GLubyte *glGetString(GLenum name)` | GL_VENDOR, GL_RENDERER, GL_VERSION, GL_SHADING_LANGUAGE_VERSION, GL_EXTENSIONS. | Print all five at boot into your log. GL_EXTENSIONS is one long space-separated string. |
| `void glGetIntegerv(GLenum pname, GLint *p)` / `glGetFloatv` / `glGetBooleanv` | Query state and limits (pname table, A.8). | Cache limits at boot; queries can stall. |
| `void glFinish(void)` | Blocks until all GL work completes. | Debugging and timing only. Never in the frame loop. |
| `void glFlush(void)` | Pushes queued work toward the GPU without waiting. | eglSwapBuffers implies it. |

## A.2 Buffer objects and vertex attributes

| Signature | What it does | Errors / notes |
|---|---|---|
| `void glGenBuffers(GLsizei n, GLuint *buffers)` | Allocates n buffer names. | Name 0 is "no buffer". |
| `void glBindBuffer(GLenum target, GLuint buffer)` | Binds to GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER. | The element-array binding is per-VAO once you use OES_vertex_array_object. |
| `void glBufferData(GLenum target, GLsizeiptr size, const void *data, GLenum usage)` | Allocates and optionally fills storage. Usage: GL_STATIC_DRAW, GL_DYNAMIC_DRAW, GL_STREAM_DRAW. | data may be NULL to just allocate. Re-specifying with NULL then SubData ("orphaning") avoids stalls on per-frame buffers (days 18, Appendix D). |
| `void glBufferSubData(GLenum target, GLintptr offset, GLsizeiptr size, const void *data)` | Overwrites a range. | INVALID_VALUE if offset+size exceeds the allocation. There is no glMapBuffer in GLES2: SubData is your only update path. |
| `void glDeleteBuffers(GLsizei n, const GLuint *buffers)` | Frees buffers. | Deleting a bound buffer unbinds it. |
| `GLboolean glIsBuffer(GLuint b)` | TRUE if b is a buffer. | |
| `void glGetBufferParameteriv(GLenum target, GLenum pname, GLint *p)` | GL_BUFFER_SIZE, GL_BUFFER_USAGE of the bound buffer. | |

| | | |
|---|---|---|
| `void glVertexAttribPointer(GLuint index, GLint size, GLenum type, GLboolean normalized, GLsizei stride, const void *ptr)` | Describes attribute `index`: size 1-4 components, type (GL_FLOAT, GL_UNSIGNED_BYTE, GL_SHORT, ...), stride in bytes, ptr = byte offset into the bound GL_ARRAY_BUFFER. | normalized=GL_TRUE maps ubyte 0..255 to 0..1 (colors!). With no buffer bound, ptr is a client-memory pointer: legal but slow through ANGLE; always use VBOs. |
| `void glEnableVertexAttribArray(GLuint index)` / `glDisableVertexAttribArray(GLuint index)` | Turns the attribute array on/off. | Forgetting to disable stale arrays crashes ANGLE with wild reads when the next draw's buffers are smaller. Track enabled arrays. |
| `void glVertexAttrib4f(GLuint index, GLfloat x, y, z, w)` (and 1f/2f/3f, fv forms) | Constant value for a disabled attribute array. | Good for "this mesh has no vertex color: use white". |
| `void glGetVertexAttribiv/fv`, `glGetVertexAttribPointerv` | Query attribute state. | Debug only. |

## A.3 Textures

| Signature | What it does | Errors / notes |
|---|---|---|
| `void glGenTextures(GLsizei n, GLuint *t)` / `glDeleteTextures(GLsizei n, const GLuint *t)` / `GLboolean glIsTexture(GLuint t)` | Create / destroy / query texture names. | |
| `void glActiveTexture(GLenum texture)` | Selects unit GL_TEXTURE0 + i before binding. | Pair with `glUniform1i(sampler_loc, i)`: the sampler uniform holds the unit NUMBER, not the texture name. Mixing those up is the classic black-model bug. |
| `void glBindTexture(GLenum target, GLuint t)` | Binds to GL_TEXTURE_2D or GL_TEXTURE_CUBE_MAP on the active unit. | A texture becomes 2D or cubemap forever at first bind. |
| `void glTexImage2D(GLenum target, GLint level, GLint internalformat, GLsizei w, GLsizei h, GLint border, GLenum format, GLenum type, const void *pixels)` | Uploads level `level`. In GLES2 internalformat MUST equal format. border must be 0. Cubemap faces use target GL_TEXTURE_CUBE_MAP_POSITIVE_X .. NEGATIVE_Z. | Formats table in A.8. pixels=NULL allocates (for FBO color targets). Rows come from bottom-up in GL terms: your TGA/stb data is top-down, so either flip at load or flip V in the tools. |
| `void glTexSubImage2D(GLenum target, GLint level, GLint x, GLint y, GLsizei w, GLsizei h, GLenum format, GLenum type, const void *pixels)` | Updates a subrectangle; format/type must match the original. | Much cheaper than re-TexImage for dynamic textures. |
| `void glCompressedTexImage2D(GLenum target, GLint level, GLenum internalformat, GLsizei w,` | Uploads DXT1/3/5 blocks (formats in A.9). | imageSize must be exact: DXT1 = max(1,w/4)*max(1,h/4)*8 bytes, DXT3/5 = *16. |

| | | |
|---|---|---|
| `GLsizei h, GLint border, GLsizei imageSize, const void *data)` | | |
| `void glCompressedTexSubImage2D(...)` | Updates compressed subrect on 4-pixel boundaries. | |
| `void glCopyTexImage2D(GLenum target, GLint level, GLenum internalformat, GLint x, GLint y, GLsizei w, GLsizei h, GLint border)` / `glCopyTexSubImage2D(...)` | Copies from the current framebuffer into a texture. | Pre-FBO render-to-texture; you have real FBOs, prefer them (day 25). |
| `void glTexParameteri(GLenum target, GLenum pname, GLint p)` (also f/iv/fv forms; `glGetTexParameteriv/fv`) | Sets per-texture state: GL_TEXTURE_MIN_FILTER, GL_TEXTURE_MAG_FILTER, GL_TEXTURE_WRAP_S, GL_TEXTURE_WRAP_T (values in A.8). | Default MIN_FILTER is GL_NEAREST_MIPMAP_LINEAR: a texture without mipmaps samples as BLACK until you set MIN_FILTER or generate mips. Bug number one of day 4. |
| `void glGenerateMipmap(GLenum target)` | Builds the full mip chain from level 0. | INVALID_OPERATION on NPOT textures (see hard rules) and on compressed textures. Call after every TexImage of level 0. |

## A.4 Shaders and programs

| Signature | What it does | Errors / notes |
|---|---|---|
| `GLuint glCreateShader(GLenum type)` | type = GL_VERTEX_SHADER or GL_FRAGMENT_SHADER. | Returns 0 on failure. |
| `void glShaderSource(GLuint shader, GLsizei count, const GLchar *const *string, const GLint *length)` | Attaches source; length=NULL means NUL-terminated strings. | Your `static const char *VS[]` arrays pass count > 1 naturally. |
| `void glCompileShader(GLuint shader)` | Compiles. ANGLE translates GLSL ES to HLSL here. | ALWAYS check GL_COMPILE_STATUS and print the info log; ANGLE's messages are good. |
| `void glGetShaderiv(GLuint s, GLenum pname, GLint *p)` | GL_COMPILE_STATUS, GL_INFO_LOG_LENGTH, GL_SHADER_TYPE, GL_DELETE_STATUS, GL_SHADER_SOURCE_LENGTH. | |
| `void glGetShaderInfoLog(GLuint s, GLsizei bufSize, GLsizei *length, GLchar *infoLog)` | Copies the compile log. | MessageBoxA it on failure; there is no debugger for shaders. |
| `void glDeleteShader(GLuint s)` | Frees the shader (deferred while attached). | Safe to delete right after linking. |
| `GLuint glCreateProgram(void)` | Creates an empty program. | |
| `void glAttachShader(GLuint prog, GLuint shader)` / `glDetachShader` | Attach exactly one VS and one FS before linking. | |

| | | |
|---|---|---|
| `void glBindAttribLocation(GLuint prog, GLuint index, const GLchar *name)` | Pins attribute `name` to location `index`. Must be called BEFORE glLinkProgram. | Do this for every attribute and your VBO setup code never has to ask. Location 0 should always be position. |
| `void glLinkProgram(GLuint prog)` | Links VS+FS; varyings are matched by name here. | Check GL_LINK_STATUS. A varying declared in FS but not written in VS is a LINK error, not a compile error. |
| `void glGetProgramiv(GLuint p, GLenum pname, GLint *v)` | GL_LINK_STATUS, GL_VALIDATE_STATUS, GL_INFO_LOG_LENGTH, GL_ATTACHED_SHADERS, GL_ACTIVE_ATTRIBUTES, GL_ACTIVE_UNIFORMS, GL_ACTIVE_ATTRIBUTE_MAX_LENGTH, GL_ACTIVE_UNIFORM_MAX_LENGTH, GL_DELETE_STATUS. | |
| `void glGetProgramInfoLog(GLuint p, GLsizei bufSize, GLsizei *length, GLchar *log)` | Link/validate log. | |
| `void glUseProgram(GLuint prog)` | Makes prog current for drawing and glUniform* calls. | glUniform* has NO program parameter: it hits the CURRENT program. Set uniforms after glUseProgram, always. |
| `void glValidateProgram(GLuint prog)` | Checks the program against current state. | Debug builds only. |
| `void glDeleteProgram(GLuint prog)` | Frees the program. | |
| `GLint glGetAttribLocation(GLuint prog, const GLchar *name)` | Location of an attribute after link. | -1 = not found (or optimized away!). |
| `GLint glGetUniformLocation(GLuint prog, const GLchar *name)` | Location of a uniform after link. | -1 if unused/optimized out; glUniform* on -1 is a silent no-op, so a misspelled name "works" and does nothing. |
| `void glUniform1f/2f/3f/4f`, `glUniform1i/2i/3i/4i`, and `*fv/*iv` array forms `(GLint loc, GLsizei count, const T *v)` | Sets scalar/vector uniforms on the current program. 1i also sets samplers (unit number). | Type must match the GLSL declaration exactly or INVALID_OPERATION. |
| `void glUniformMatrix2fv/3fv/4fv(GLint loc, GLsizei count, GLboolean transpose, const GLfloat *value)` | Sets matrix uniforms. | transpose MUST be GL_FALSE in GLES2. Your column-major `float m[16]` passes straight through. |
| `void glGetActiveAttrib / glGetActiveUniform (GLuint p, GLuint index, GLsizei bufSize, GLsizei *len, GLint *size, GLenum *type, GLchar *name)` | Enumerates attributes/uniforms by index. | For a console "shaderinfo" command (day 19). |
| | Reads a uniform back. | Debug only. |

| | | |
|---|---|---|
| `void glGetUniformfv/iv(GLuint p, GLint loc, T *params)` | | |
| `void glGetShaderSource(...)`, `glGetShaderPrecisionFormat(...)`, `glReleaseShaderCompiler(void)`, `glShaderBinary(...)` | Introspection and binary-shader plumbing. | ANGLE exposes ZERO shader binary formats: glShaderBinary is dead weight here. Program binaries exist instead via OES_get_program_binary (A.9). |

## A.5 Drawing

| Signature | What it does | Errors / notes |
|---|---|---|
| `void glDrawArrays(GLenum mode, GLint first, GLsizei count)` | Draws count vertices starting at first from the enabled attribute arrays. | Primitive modes in A.8. |
| `void glDrawElements(GLenum mode, GLsizei count, GLenum type, const void *indices)` | Indexed draw. type = GL_UNSIGNED_BYTE or GL_UNSIGNED_SHORT (GL_UNSIGNED_INT only with OES_element_index_uint). indices = byte offset into the bound GL_ELEMENT_ARRAY_BUFFER. | count is the INDEX count, not the triangle count. An out-of-range index is undefined behavior; ANGLE validates and may drop the draw or error, but do not lean on that. |

## A.6 Framebuffers and renderbuffers (core in GLES2)

| Signature | What it does | Errors / notes |
|---|---|---|
| `void glGenFramebuffers(GLsizei n, GLuint *f)` / `glDeleteFramebuffers` / `GLboolean glIsFramebuffer(GLuint f)` | FBO lifetime. | |
| `void glBindFramebuffer(GLenum target, GLuint f)` | target = GL_FRAMEBUFFER. f=0 restores the window surface. | Reset glViewport after every bind: it is global state. |
| `void glFramebufferTexture2D(GLenum target, GLenum attachment, GLenum textarget, GLuint texture, GLint level)` | Attaches a texture level as GL_COLOR_ATTACHMENT0, GL_DEPTH_ATTACHMENT, or GL_STENCIL_ATTACHMENT. textarget = GL_TEXTURE_2D or a cubemap face. | level must be 0 in GLES2. One color attachment only. |
| `void glFramebufferRenderbuffer(GLenum target, GLenum attachment, GLenum rbtarget, GLuint rb)` | Attaches a renderbuffer (rbtarget = GL_RENDERBUFFER). | For depth on RTT: DEPTH_COMPONENT16 renderbuffer, or DEPTH24_STENCIL8_OES attached to depth AND stencil (A.9). |
| `GLenum glCheckFramebufferStatus(GLenum target)` | Returns completeness (codes in A.8). | Check ONCE at creation, MessageBoxA on anything but GL_FRAMEBUFFER_COMPLETE. Drawing into an incomplete FBO gives INVALID_FRAMEBUFFER_OPERATION per draw. |

| | | |
|---|---|---|
| `void glGenRenderbuffers(GLsizei n, GLuint *r)` / `glDeleteRenderbuffers` / `glIsRenderbuffer` / `glBindRenderbuffer(GL_RENDERBUFFER, r)` | Renderbuffer lifetime and binding. | |
| `void glRenderbufferStorage(GLenum target, GLenum internalformat, GLsizei w, GLsizei h)` | Allocates storage. Core formats: GL_RGBA4, GL_RGB565, GL_RGB5_A1, GL_DEPTH_COMPONENT16, GL_STENCIL_INDEX8. Extensions add GL_RGB8_OES, GL_RGBA8_OES, GL_DEPTH24_STENCIL8_OES, GL_DEPTH_COMPONENT32_OES. | |
| `void glGetRenderbufferParameteriv(...)` / `glGetFramebufferAttachmentParameteriv(...)` | Introspection of attachments. | Debug only. |

## A.7 Occlusion queries (EXT_occlusion_query_boolean)

Not core; fetch every one of these through `eglGetProcAddress` (A.9). Boolean only: "did ANY sample pass", not a sample count. That is exactly what day 27 needs.

| Signature | What it does | Errors / notes |
|---|---|---|
| `void glGenQueriesEXT(GLsizei n, GLuint *ids)` / `glDeleteQueriesEXT` / `GLboolean glIsQueryEXT(GLuint)` | Query object lifetime. | |
| `void glBeginQueryEXT(GLenum target, GLuint id)` | target = GL_ANY_SAMPLES_PASSED_EXT or GL_ANY_SAMPLES_PASSED_CONSERVATIVE_EXT. | One active query per target. CONSERVATIVE may report visible when occluded (never the reverse); it is cheaper - use it. |
| `void glEndQueryEXT(GLenum target)` | Ends the active query. | |
| `void glGetQueryivEXT(GLenum target, GLenum pname, GLint *p)` | GL_CURRENT_QUERY_EXT. | |
| `void glGetQueryObjectuivEXT(GLuint id, GLenum pname, GLuint *p)` | pname = GL_QUERY_RESULT_AVAILABLE_EXT (poll!) then GL_QUERY_RESULT_EXT (0 or nonzero). | Asking for RESULT before AVAILABLE blocks the CPU on the GPU: on this Atom that is your whole frame. Use last frame's result and accept one frame of lag. |

## A.8 Key enums by area

### Capabilities (glEnable/glDisable)

| Enum | Meaning |
|---|---|
| GL_DEPTH_TEST | Depth compare and write (off by default!) |
| GL_BLEND | Framebuffer blending |
| GL_CULL_FACE | Back/front face culling |
| GL_SCISSOR_TEST | Scissor rectangle clip |
| GL_STENCIL_TEST | Stencil compare and ops |
| GL_POLYGON_OFFSET_FILL | Depth bias for triangles |
| GL_DITHER | Dithering (on by default; irrelevant at 32bpp) |
| GL_SAMPLE_ALPHA_TO_COVERAGE, GL_SAMPLE_COVERAGE | MSAA controls; unused on this stack |

### Texture formats and types (glTexImage2D)

| format/internalformat | Texel | Legal types |
|---|---|---|
| GL_RGBA | 4 channels | GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_5_5_5_1 (+ GL_FLOAT / GL_HALF_FLOAT_OES via A.9) |
| GL_RGB | 3 channels | GL_UNSIGNED_BYTE, GL_UNSIGNED_SHORT_5_6_5 (+ float types via A.9) |
| GL_LUMINANCE | 1 channel, replicated to RGB | GL_UNSIGNED_BYTE |
| GL_LUMINANCE_ALPHA | 2 channels | GL_UNSIGNED_BYTE |
| GL_ALPHA | alpha only (RGB=0) | GL_UNSIGNED_BYTE - perfect for bitmap fonts (day 6) |

### Filters and wraps (glTexParameteri)

| Enum | Meaning |
|---|---|
| GL_NEAREST / GL_LINEAR | Point / bilinear (MIN and MAG) |
| GL_NEAREST_MIPMAP_NEAREST | Point in level, point between levels |
| GL_LINEAR_MIPMAP_NEAREST | Bilinear, one mip level |
| GL_NEAREST_MIPMAP_LINEAR | Point, blend two levels (the trap default) |
| GL_LINEAR_MIPMAP_LINEAR | Trilinear - the good one, MIN only |
| GL_REPEAT | Wrap (POT textures only) |
| GL_MIRRORED_REPEAT | Wrap mirrored (POT only) |
| GL_CLAMP_TO_EDGE | Clamp; the only wrap legal on NPOT |

### Blend factors (glBlendFunc)

| Factor | Multiplies by |
|---|---|

| | |
|---|---|
| GL_ZERO / GL_ONE | 0 / 1 |
| GL_SRC_COLOR / GL_ONE_MINUS_SRC_COLOR | source RGB / 1-RGB |
| GL_DST_COLOR / GL_ONE_MINUS_DST_COLOR | dest RGB / 1-RGB |
| GL_SRC_ALPHA / GL_ONE_MINUS_SRC_ALPHA | source alpha - the standard transparency pair |
| GL_DST_ALPHA / GL_ONE_MINUS_DST_ALPHA | dest alpha (needs EGL_ALPHA_SIZE > 0) |
| GL_CONSTANT_COLOR / GL_ONE_MINUS_CONSTANT_COLOR / GL_CONSTANT_ALPHA / GL_ONE_MINUS_CONSTANT_ALPHA | glBlendColor value |
| GL_SRC_ALPHA_SATURATE | min(As, 1-Ad); src factor only |

Recipes: transparency = (GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); additive glow/particles = (GL_SRC_ALPHA, GL_ONE); multiply/lightmap- style darken = (GL_DST_COLOR, GL_ZERO).

### Primitive types (glDrawArrays/glDrawElements)

| Mode | Meaning |
|---|---|
| GL_POINTS | 1 vertex each; size from gl_PointSize |
| GL_LINES | Vertex pairs |
| GL_LINE_STRIP / GL_LINE_LOOP | Connected lines / closed |
| GL_TRIANGLES | Vertex triples - use this for everything |
| GL_TRIANGLE_STRIP | N+2 vertices for N triangles |
| GL_TRIANGLE_FAN | Fan around vertex 0 |

There is no GL_QUADS. There never will be. Two triangles.

### Shader/program query pnames

| pname | Use with | Meaning |
|---|---|---|
| GL_COMPILE_STATUS | glGetShaderiv | GL_TRUE on successful compile |
| GL_LINK_STATUS / GL_VALIDATE_STATUS | glGetProgramiv | Link / validate result |
| GL_INFO_LOG_LENGTH | both | Log size incl. NUL |
| GL_SHADER_TYPE / GL_DELETE_STATUS | glGetShaderiv | VS or FS / marked deleted |
| GL_ATTACHED_SHADERS | glGetProgramiv | count |
| GL_ACTIVE_ATTRIBUTES / GL_ACTIVE_UNIFORMS (+ _MAX_LENGTH) | glGetProgramiv | counts for glGetActive* |

### FBO attachment points and status codes

| Enum | Meaning |
|---|---|
| GL_COLOR_ATTACHMENT0 | The one and only color target |
| GL_DEPTH_ATTACHMENT | Depth texture or renderbuffer |
| GL_STENCIL_ATTACHMENT | Stencil renderbuffer |
| GL_FRAMEBUFFER_COMPLETE | Good to draw |

| | |
|---|---|
| GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT | An attachment is broken (wrong format, zero size, NPOT trouble) |
| GL_FRAMEBUFFER_INCOMPLETE_MISSING_ATTACHMENT | Nothing attached at all |
| GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS | Attachments differ in size |
| GL_FRAMEBUFFER_UNSUPPORTED | Legal combination the driver just does not do; try RGBA + DEPTH_COMPONENT16, POT sizes |

### Limits worth caching at boot (glGetIntegerv)

| pname | GLES2 minimum | Note |
|---|---|---|
| GL_MAX_TEXTURE_SIZE | 64 | Expect 2048-4096 here; still, keep textures at 512 or less for 1 GB RAM |
| GL_MAX_CUBE_MAP_TEXTURE_SIZE | 16 | Skybox faces 512 is plenty at 1024x600 |
| GL_MAX_VERTEX_ATTRIBS | 8 | ANGLE gives 16 |
| GL_MAX_VERTEX_UNIFORM_VECTORS | 128 | Bone palettes (day 24) live here: 1 mat4 = 4 vectors |
| GL_MAX_FRAGMENT_UNIFORM_VECTORS | 16 | Budget your light arrays |
| GL_MAX_VARYING_VECTORS | 8 | Every vec4 between VS and FS counts |
| GL_MAX_TEXTURE_IMAGE_UNITS | 8 | FS samplers |
| GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS | 0! | May be ZERO: do not design around vertex texture fetch |
| GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS | 8 | |
| GL_MAX_RENDERBUFFER_SIZE | 1 | Check before big FBOs |
| GL_MAX_VIEWPORT_DIMS | screen size | 2 values |

## A.9 The extension shelf

Check availability once at boot, then fetch pointers. Both steps in C89:

```
/* ext.c - extension check + fetch (pattern) */
#include <string.h>
#include "gl_loader.h"

typedef void (GL_APIENTRY *PFNGLBINDVERTEXARRAYOESPROC)(GLuint array);
static PFNGLBINDVERTEXARRAYOESPROC glBindVertexArrayOES;

static int ext_has(const char *name)
{
    const char *all = (const char *)glGetString(GL_EXTENSIONS);
    return all != NULL && strstr(all, name) != NULL;
}

int ext_init(void)
{
    if (ext_has("GL_OES_vertex_array_object")) {
        glBindVertexArrayOES = (PFNGLBINDVERTEXARRAYOESPROC)
            eglGetProcAddress("glBindVertexArrayOES");
    }
    return 0;
}
```

> **TIP:** `strstr` can false-positive on prefixes ("GL_OES_texture_float" matches inside "GL_OES_texture_float_linear"). Search for the name followed by a space, or check the longer names first. Two minutes now, one confusing evening saved.

**GL_OES_vertex_array_object.** Adds vertex array objects: one object records all glVertexAttribPointer/Enable state plus the element buffer binding, so a mesh switch becomes one call instead of eight. Functions: `glGenVertexArraysOES`, `glBindVertexArrayOES`, `glDeleteVertexArraysOES`, `glIsVertexArrayOES`. This is the ONLY way to get VAOs in GLES2; do not expect the desktop names.

**GL_ANGLE_instanced_arrays.** Draws N copies of a mesh in one call, with selected attributes stepping once per instance instead of per vertex. Functions: `glDrawArraysInstancedANGLE`, `glDrawElementsInstancedANGLE`, `glVertexAttribDivisorANGLE` (divisor 1 = per-instance). Good for the day-18 particles and day-12 chunk props if draw-call count ever becomes the wall.

**GL_OES_texture_float / GL_OES_texture_half_float (+ _linear).** Allows GL_FLOAT / GL_HALF_FLOAT_OES as the type argument of glTexImage2D, giving float texel storage. The _linear variants allow GL_LINEAR filtering on them (without those you get NEAREST only). No new functions. Heightfields and baked data, mostly; rendering TO float is not promised.

**GL_OES_standard_derivatives.** Adds `dFdx()`, `dFdy()`, `fwidth()` to fragment shaders (enable with `#extension GL_OES_standard_derivatives : enable`). No C-side functions. Useful for screen-space normal hacks and anti-aliased procedural lines.

**GL_EXT_frag_depth.** Lets the fragment shader write depth via `gl_FragDepthEXT` (enable `GL_EXT_frag_depth` in the shader). No C-side functions. Writing depth disables early-Z for that draw: use sparingly.

**GL_OES_element_index_uint.** Allows GL_UNSIGNED_INT in glDrawElements. No new functions. Without it 65535 is your ceiling per index buffer - see the hard rules box.

**GL_EXT_texture_filter_anisotropic.** Adds `glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISOTROPY_EXT, n)`; query the max with GL_MAX_TEXTURE_MAX_ANISOTROPY_EXT. Sharpens grazing-angle terrain textures (day 11). Costs bandwidth; try 2.0, not max.

**DXT1/3/5 compressed textures** (EXT_texture_compression_dxt1 + ANGLE_texture_compression_dxt3/5). Adds formats GL_COMPRESSED_RGB_S3TC_DXT1_EXT, GL_COMPRESSED_RGBA_S3TC_DXT1_EXT, GL_COMPRESSED_RGBA_S3TC_DXT3_EXT, GL_COMPRESSED_RGBA_S3TC_DXT5_EXT for glCompressedTexImage2D. 4x-8x memory savings - on 1 GB shared with the GPU, that is not optional for big scenes. You must supply pre-compressed blocks (write a tiny DDS reader; the format is a header plus raw blocks).

**GL_EXT_occlusion_query_boolean.** Boolean occlusion queries; full function table in A.7. Day 27's second option.

**GL_OES_packed_depth_stencil.** Adds GL_DEPTH24_STENCIL8_OES: one renderbuffer that you attach to BOTH GL_DEPTH_ATTACHMENT and GL_STENCIL_ATTACHMENT of an FBO. The only sane way to get stencil on a render target here. No new functions.

**GL_ANGLE_framebuffer_blit / GL_ANGLE_framebuffer_multisample.** Adds `glBlitFramebufferANGLE` plus separate GL_READ_FRAMEBUFFER_ANGLE / GL_DRAW_FRAMEBUFFER_ANGLE binding targets, and `glRenderbufferStorageMultisampleANGLE` for MSAA renderbuffers that resolve via the blit. Nearest-only, whole-surface style blits; fine for copying a resolved scene into a postprocess texture (day 25).

**GL_OES_depth32.** GL_DEPTH_COMPONENT32_OES renderbuffer format. No new functions. 16-bit depth plus a 1000-unit far plane equals z-fighting; this is the fix if you see sparkle in the distance.

**GL_OES_rgb8_rgba8.** GL_RGB8_OES / GL_RGBA8_OES renderbuffer formats - true 8-bit-per-channel color render targets instead of core's 4/5-bit ones. No new functions. Use for every color FBO you make.

**GL_EXT_texture_storage.** Adds `glTexStorage2DEXT(target, levels, sizedformat, w, h)`: allocates a complete, immutable mip chain up front. The driver stops second-guessing whether your texture is complete, which removes a class of black-texture bugs and some first-use hitches.

**GL_OES_get_program_binary.** Adds `glGetProgramBinaryOES` and `glProgramBinaryOES` (query size via GL_PROGRAM_BINARY_LENGTH_OES). ANGLE compiles your GLSL to HLSL and then D3D9 bytecode at link time - slow on the Atom. Cache program binaries to disk at first run and boot in a fraction of the time. Binaries break across driver versions: store a version stamp and fall back to source.

**GL_EXT_blend_minmax.** GL_MIN_EXT / GL_MAX_EXT for glBlendEquation. No new functions.

**GL_EXT_shader_texture_lod.** Adds explicit-LOD texture lookups to FRAGMENT shaders: `texture2DLodEXT`, `texture2DProjLodEXT`, `textureCubeLodEXT`, plus gradient forms (`texture2DGradEXT`...). Enable `GL_EXT_shader_texture_lod` in the shader. Without this, explicit LOD exists only in vertex shaders (Appendix B).

**GL_NV_fence.** GPU sync points: `glGenFencesNV`, `glDeleteFencesNV`, `glSetFenceNV`, `glTestFenceNV` (poll), `glFinishFenceNV` (block), `glIsFenceNV`, `glGetFenceivNV`. Lets you ask "has the GPU reached this point" without glFinish. Useful to detect when it is safe to rewrite a dynamic buffer.

**GL_KHR_debug.** Driver messages delivered to a callback: `glDebugMessageCallbackKHR`, `glDebugMessageControlKHR`, `glDebugMessageInsertKHR`, `glGetDebugMessageLogKHR`, plus `glPushDebugGroupKHR`/`glPopDebugGroupKHR` and `glObjectLabelKHR`. Wire the callback to your day-19 console and ANGLE will tell you what you did wrong in complete sentences. The best free debugging on this machine.

## A.10 The hard rules

**WARNING:** GLES2 law, enforced by this driver.
1. **NPOT textures are second-class.** A non-power-of-two texture gets NO mipmaps (glGenerateMipmap fails with INVALID_OPERATION), NO GL_REPEAT or GL_MIRRORED_REPEAT (sampling returns black), only GL_CLAMP_TO_EDGE with GL_NEAREST/GL_LINEAR. Pad or scale art to power-of-two in PSP7 and life is simple.
2. **No wireframe.** glPolygonMode does not exist. The day-19 "r_wireframe" cvar must draw GL_LINES from your own edge index buffer.
3. **Indices are 16-bit** (GL_UNSIGNED_SHORT/BYTE) unless you verified GL_OES_element_index_uint. Over 65535 vertices? Split the mesh or use the extension - a 32-bit index buffer without it is an INVALID_ENUM at draw time.
4. **VAOs only via OES_vertex_array_object** - the *OES names through eglGetProcAddress. Core GLES2 has none; set attribute pointers per draw if you skip the extension.
5. **No quads, no glBegin, no matrix stack, no fixed function.** Your mathlib (day 1) IS the matrix stack now.
6. glUniform* only touches the program bound by glUseProgram, and glUniformMatrix*fv demands transpose = GL_FALSE.

# Appendix B: GLSL ES 1.00 Reference

GLSL ES 1.00 is the only shading language this stack speaks. It looks like C, it is not C: no implicit type conversion, no pointers, no recursion, and the compiler (ANGLE, translating to HLSL for D3D9 SM3) enforces the small print. Start every shader with nothing - there is no `#version` needed; 100 is the default and the only choice.

## B.1 Types

| Type | Meaning | Notes |
|---|---|---|
| `void` | functions only | |
| `bool` | true/false | |
| `int` | signed integer | NOT guaranteed 32-bit; mediump int only promises 10 bits of magnitude on minimal hardware. No bitwise ops, no % operator in ESSL 1.00. Loop counters and array indices - that is its life. |
| `float` | scalar float | Precision set by qualifiers (B.3) |
| `vec2 / vec3 / vec4` | float vectors | Constructors convert: `vec4(pos, 1.0)`. Swizzle freely: `v.xyz`, `v.rgb`, `v.xxyy`. |
| `ivec2/3/4, bvec2/3/4` | int / bool vectors | Mostly from comparisons (B.5 relational) |
| `mat2 / mat3 / mat4` | square float matrices, column-major | `m[2]` is column 2 (a vec); `m[2][1]` is column 2 row 1. Matches your C mat4 layout exactly. |
| `sampler2D` | 2D texture handle | Uniform only; set from C with glUniform1i(loc, unit) |
| `samplerCube` | cubemap handle | Same rules |

> **NOTE:** No implicit conversions. `float f = 1;` is a COMPILE ERROR - write `1.0`. `vec3 v = vec3(0);` is fine (constructors convert explicitly). Half the compile errors of your first week are missing `.0`.

Also: structs work; arrays are 1-D, constant-sized, cannot be initialized at declaration and cannot be assigned whole; functions pass parameters by value-copy (`in`, `out`, `inout` qualifiers); recursion is forbidden.

## B.2 Storage qualifiers

| Qualifier | Where | Meaning | Limit (min) |
|---|---|---|---|
| `attribute` | VS only, global | Per-vertex input from glVertexAttribPointer | GL_MAX_VERTEX_ATTRIBS (8; ANGLE 16) |
| `uniform` | VS + FS, global | Constant across a draw; set with glUniform* | 128 vec4 in VS, 16 vec4 in FS (query, do not guess) |
| `varying` | VS out, FS in | Written per-vertex, interpolated per-fragment. Declared IDENTICALLY (name and type) in both shaders; matched at link. | GL_MAX_VARYING_VECTORS (8 vec4) |
| `const` | anywhere | Compile-time constant; required for array sizes and (in practice) loop bounds | |

## B.3 Precision qualifiers

| Qualifier | Float range (min) | Float precision | Use for |
|---|---|---|---|
| `highp` | about ±2^62 | 2^-16 relative | Positions, matrices, depth math |
| `mediump` | ±16384 | 2^-10 relative | Texcoords, normals, most lighting |
| `lowp` | ±2 | 2^-8 absolute | Colors, unit vectors you can afford to chew |

Vertex shaders default to `highp` for float and int. Fragment shaders have NO default float precision - every float declaration is an error until you state one. Put this first line in every fragment shader:

```
precision mediump float;
```

> **WARNING:** The missing `precision mediump float;` line is the number one black screen in this book. The fragment shader fails to compile, the link fails, glUseProgram(0 or stale program) draws nothing, and if you did not check GL_COMPILE_STATUS you will stare at a cleared framebuffer wondering about your matrices. It is never the matrices. Check the shader log first.

The SGX545 through ANGLE/D3D9 computes everything at fp24-fp32 anyway, so `mediump` costs nothing here - but declare it correctly and the same shaders will run on a phone someday. You can also qualify single declarations: `highp float dist;`.

## B.4 Built-in variables

| Variable | Type | Stage | Meaning |
|---|---|---|---|
| `gl_Position` | highp vec4 | VS out | Clip-space vertex position. Writing it is MANDATORY. |
| `gl_PointSize` | mediump float | VS out | Point sprite diameter in pixels (GL_POINTS only). Clamped to GL_ALIASED_POINT_SIZE_RANGE - query it before betting the particle system on big sprites (day 18). |
| `gl_FragColor` | mediump vec4 | FS out | Output color. The only output that matters in GLES2. |
| `gl_FragData[0]` | mediump vec4 | FS out | Alias of gl_FragColor; index 0 only (no MRT). |
| `gl_FragCoord` | mediump vec4 | FS in | Window coords of the fragment; origin BOTTOM-LEFT, .z is depth in 0..1, .w = 1/clip.w. |
| `gl_FrontFacing` | bool | FS in | true for front faces; flip normals for two-sided lighting. |
| `gl_PointCoord` | mediump vec2 | FS in | 0..1 across a point sprite - your particle UV. |

Built-in constants mirror the glGetIntegerv limits (`gl_MaxVertexAttribs`, `gl_MaxVaryingVectors`, `gl_MaxVertexUniformVectors`, `gl_MaxFragmentUniformVectors`, `gl_MaxTextureImageUnits`, `gl_MaxVertexTextureImageUnits`, ...).

## B.5 Built-in functions

All work component-wise on vectors unless noted. `genType` means float, vec2, vec3, or vec4.

### Angle and trigonometry

| Function | Meaning |
|---|---|
| `radians(deg), degrees(rad)` | unit conversion |
| `sin(x), cos(x), tan(x)` | radians in |
| `asin(x), acos(x)` | inverse; x outside [-1,1] is undefined |
| `atan(y, x)` / `atan(y_over_x)` | two-arg form gives full-circle angle |

### Exponential

| Function | Meaning |
|---|---|
| `pow(x, y)` | x^y; undefined for x < 0 - the specular-highlight NaN generator, clamp your N.H first |
| `exp(x), log(x), exp2(x), log2(x)` | e- and 2-based |
| `sqrt(x), inversesqrt(x)` | inversesqrt is cheap - normalize() uses it |

### Common

| Function | Meaning |
|---|---|
| `abs(x), sign(x)` | |
| `floor(x), ceil(x), fract(x)` | fract(x) = x - floor(x) |
| `mod(x, y)` | x - y*floor(x/y); this is your % |
| `min(x, y), max(x, y)` | |
| `clamp(x, lo, hi)` | min(max(x,lo),hi) |
| `mix(a, b, t)` | a*(1-t) + b*t - THE lerp; also your day-17 fog and day-23 keyframe blend |
| `step(edge, x)` | 0 if x < edge else 1; branch-free if |
| `smoothstep(e0, e1, x)` | hermite 0..1 between edges |

### Geometric

| Function | Meaning |
|---|---|
| `length(v), distance(a, b)` | |
| `dot(a, b)` | scalar product - the whole of lighting |
| `cross(a, b)` | vec3 only |
| `normalize(v)` | v/length(v); undefined at zero length |
| `faceforward(N, I, Nref)` | N flipped to oppose I |
| `reflect(I, N)` | I - 2*dot(N,I)*N; N must be normalized (cubemap reflections, specular) |
| `refract(I, N, eta)` | refraction vector (day-26 water, if you get fancy) |

### Matrix and relational

| Function | Meaning |
|---|---|
| `matrixCompMult(a, b)` | component-wise product (NOT matrix multiply; use `*` for that) |
| `lessThan, lessThanEqual, greaterThan, greaterThanEqual, equal, notEqual` | component compare, return bvec |
| `any(bv), all(bv), not(bv)` | bvec reductions |

### Texture lookups

| Function | Stage | Meaning |
|---|---|---|
| `texture2D(sampler2D, vec2 uv)` | VS+FS | Standard lookup. In the VS there are no derivatives, so it samples the base level - and remember GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS may be 0 on this stack. |
| `texture2D(s, uv, bias)` | FS only | Adds bias to the computed LOD |
| `texture2DProj(s, vec3\|vec4)` | VS+FS | Divides by last component first - projective texturing |
| `texture2DLod(s, uv, lod)` | VS ONLY | Explicit mip level. Calling it in a fragment shader is a compile error in ESSL 1.00. |
| `texture2DProjLod(s, coord, lod)` | VS ONLY | |
| `textureCube(samplerCube, vec3 dir)` | VS+FS | Direction, not UV; no normalize needed (skybox, day 10) |
| `textureCubeLod(s, dir, lod)` | VS ONLY | |

Need explicit LOD in a fragment shader? That is exactly what GL_EXT_shader_texture_lod adds (`texture2DLodEXT`, see Appendix A.9), behind an `#extension` line. And GL_OES_standard_derivatives adds `dFdx/dFdy/fwidth`, FS only, same deal.

## B.6 Control flow: the fine print

ESSL 1.00 Appendix A only guarantees loops the compiler can fully understand, and ANGLE (compiling to SM3, which has real but rigid loop instructions) holds you to it:

| Rule | Practical form |
|---|---|
| One loop index, initialized with a constant expression | `for (int i = 0; ...)` |
| Compared against a constant expression with < <= > >= == != | `i < 4` - NOT `i < u_count`. Loop to a compile-time max and `if (i >= u_count) break;`? Legal, but see the branch note below. |
| Stepped by a constant: i++, i--, i+=k, i-=k | No modifying the index in the body |
| while / do-while | Not required to work AT ALL. On this stack: treat as absent. |
| Array indexing in the FS | |

| | |
|---|---|
| | Constant expressions and loop indices only. Samplers may NEVER be indexed dynamically - no `u_tex[i]` with variable i; write a chain of ifs or split the draw. |

**NOTE:** On SGX545-class hardware through D3D9, an `if` is frequently executed as BOTH sides + select, and dynamic branches that do work cost extra. For small either/or math, prefer `mix()`, `step()`, and arithmetic over branching. Branch to SKIP genuinely heavy work (a whole light's worth of math) only when whole warps of nearby pixels agree - e.g. "this pixel is beyond fog saturation". Measure with the day-6 fps counter; this Atom tells the truth.

## B.7 Invariance and precision pitfalls

| Symptom | Cause | Fix |
|---|---|---|
| Z-fighting between passes that draw the same geometry (day 16 lightmap pass, day 28 decals) | Two shaders computed gl_Position with different instruction orderings; values differ in the last bit | Declare `invariant gl_Position;` in BOTH vertex shaders, and compute position from the SAME uniforms in the same order (one premultiplied MVP, not V*P in one shader and (M*V)*P in another) |
| Texture shimmer / stairstepping on big terrain UVs | mediump varying loses bits when UV values are large (tiling 100x) | Keep UVs near origin (fract on the CPU per chunk) or qualify that varying highp |
| Lighting sparkles / black pixels | pow(x, e) with x slightly negative; normalize of near-zero vector | `pow(max(x, 0.0), e)`; guard interpolated normals: renormalize in FS |
| Fog banding at 16-bit color FBOs | lowp arithmetic + GL_RGB565 target | Use GL_RGBA8_OES render targets (A.9) |
| Works on ANGLE, breaks on a real phone later | You silently relied on highp in the FS | Keep declared precisions honest even though this GPU forgives |

## B.8 A minimal pair, line by line

The day-13 workhorse: one directional light, one texture, computed per vertex. As C string arrays per the house style:

```
static const char *VS =
"uniform mat4 u_mvp;\n"
"uniform mat3 u_normal_mat;\n"
"uniform vec3 u_light_dir;\n"
"attribute vec3 a_pos;\n"
"attribute vec3 a_normal;\n"
"attribute vec2 a_uv;\n"
"varying vec2 v_uv;\n"
"varying float v_diffuse;\n"
"void main() {\n"
"    vec3 n = normalize(u_normal_mat * a_normal);\n"
"    v_diffuse = max(dot(n, -u_light_dir), 0.0);\n"
"    v_uv = a_uv;\n"
"    gl_Position = u_mvp * vec4(a_pos, 1.0);\n"
"}\n";

static const char *FS =
"precision mediump float;\n"
"uniform sampler2D u_tex;\n"
"uniform vec3 u_ambient;\n"
"varying vec2 v_uv;\n"
"varying float v_diffuse;\n"
"void main() {\n"
"    vec4 texel = texture2D(u_tex, v_uv);\n"
```

```
"    vec3 lit = u_ambient + vec3(v_diffuse);\n"
"    gl_FragColor = vec4(texel.rgb * lit, texel.a);\n"
"}\n";
```

| Line | Why it is there |
|---|---|
| `uniform mat4 u_mvp;` | Projection * view * model, premultiplied in C with mat4_multiply. One matrix upload per draw beats three. |
| `uniform mat3 u_normal_mat;` | Rotation part of the model matrix for normals. With uniform scale, the upper 3x3 works; non-uniform scale needs inverse-transpose (computed in C - GLSL ES has no inverse()). |
| `uniform vec3 u_light_dir;` | World-space direction the light TRAVELS, normalized in C once, not per vertex. |
| `attribute vec3 a_pos; ... a_normal; ... a_uv;` | Matches the interleaved VBO; pinned to locations 0,1,2 with glBindAttribLocation before linking. |
| `varying vec2 v_uv; varying float v_diffuse;` | Interpolated outputs. 1.25 of your 8 varying vectors used. |
| `vec3 n = normalize(...);` | Rotate then renormalize - interpolation and scale both shrink normals. |
| `max(dot(n, -u_light_dir), 0.0)` | Lambert N.L, negated because u_light_dir points FROM the sun; max clamps the dark side to zero instead of negative light. |
| `gl_Position = u_mvp * vec4(a_pos, 1.0);` | The mandatory line. w=1.0 for positions (0.0 would make it a direction). |
| `precision mediump float;` | See the warn box in B.3. First line, every fragment shader, forever. |
| `vec4 texel = texture2D(u_tex, v_uv);` | u_tex was set from C: glUniform1i(loc, 0) with the texture bound on unit GL_TEXTURE0. |
| `u_ambient + vec3(v_diffuse)` | Classic ambient + diffuse. Constructor splats the scalar to all three channels. |
| `gl_FragColor = vec4(texel.rgb * lit, texel.a);` | Modulate color, pass alpha through untouched so day-6 blending still works. |

# Appendix C: BASS 2.4 Mini-Reference

BASS is the one black box you are allowed (rule 1 of the quest). Link `bass.lib`, ship `bass.dll` next to the exe, include `bass.h`. Everything below is the subset an engine needs; signatures are verbatim from the header. All handles are plain `DWORD`s; every function that can fail returns 0/FALSE and leaves a code for `BASS_ErrorGetCode()` (table C.6). `QWORD` is `unsigned __int64` - VC6 accepts it.

## C.1 Boot and shutdown

| Signature | What it does | Notes |
|---|---|---|
| `BOOL BASS_Init(int device, DWORD freq, DWORD flags, HWND win, const void *dsguid)` | Opens the output device and starts the update thread. | Use `BASS_Init(-1, 44100, 0, hwnd, NULL)`: -1 = default device, your window handle, no DirectSound GUID. Call ONCE at boot, after the window exists. Fails with BASS_ERROR_ALREADY if repeated. |
| `BOOL BASS_Free(void)` | Stops everything, frees all streams/musics/samples, closes the device. | One call at shutdown; no need to free handles individually first. |
| `DWORD BASS_GetVersion(void)` | Returns the DLL version. | Check `HIWORD(BASS_GetVersion()) == BASSVERSION` at boot to catch a stray old bass.dll on the path. |
| `int BASS_ErrorGetCode(void)` | Error code of the LAST failed call (per thread). | Read it immediately; the next call overwrites it. |
| `BOOL BASS_SetVolume(float volume)` / `float BASS_GetVolume(void)` | Master device volume 0..1. | This is the Windows-level output volume; prefer per-channel BASS_ATTRIB_VOL for the console's "volume" cvar. |
| `BOOL BASS_Start(void)` / `BASS_Stop(void)` / `BASS_Pause(void)` | Device-wide output control. | Pause on WM_ACTIVATE losing focus, Start on regaining - a 1999 courtesy players still appreciate. |

## C.2 Streams (wav/mp3/ogg from disk)

| Signature | What it does | Notes |
|---|---|---|
| `HSTREAM BASS_StreamCreateFile(DWORD filetype, const void *file, QWORD offset, QWORD length, DWORD flags)` | Opens a file as a stream, decoded on the fly. | filetype = `BASS_FILE_NAME` and `file` = the path for normal use (`BASS_FILE_MEM` plays from a memory block of `length` bytes). offset/length 0,0 = whole file. Returns 0 on failure. |
| `BOOL BASS_StreamFree(HSTREAM handle)` | Stops and frees the stream. | |

Stream flags worth knowing:

| Flag | Meaning |
|---|---|

| | |
|---|---|
| `BASS_SAMPLE_LOOP` | Loop at the end (can be toggled later with BASS_ChannelFlags) |
| `BASS_STREAM_PRESCAN` | Scan whole file up front: exact length and clean seeking for VBR mp3 |
| `BASS_STREAM_AUTOFREE` | Frees itself when playback ends - fire-and-forget one-shots |
| `BASS_STREAM_DECODE` | No playback; you pull data with BASS_ChannelGetData (not needed in this book) |
| `BASS_SAMPLE_MONO / BASS_SAMPLE_8BITS / BASS_SAMPLE_FLOAT` | Decode format overrides |

## C.3 Module music (.xm / .it from ModPlug)

| Signature | What it does | Notes |
|---|---|---|
| `HMUSIC BASS_MusicLoad(DWORD filetype, const void *file, QWORD offset, DWORD length, DWORD flags, DWORD freq)` | Loads a MOD/ XM/IT/S3M/ MTM module for playback. | Same filetype/file convention as streams; freq = mixing rate, 0 = device rate (use 0). Returns 0 on failure - a corrupt or misnamed module gives BASS_ERROR_FILEFORM. |
| `BOOL BASS_MusicFree(HMUSIC handle)` | Stops and frees the module. | |

Music flags (OR them into `flags`):

| Flag | Meaning |
|---|---|
| `BASS_MUSIC_LOOP` | Loop the module - your day-21 soundtrack wants this |
| `BASS_MUSIC_RAMP / BASS_MUSIC_RAMPS` | Normal / sensitive volume ramping; kills the clicks on note cuts. RAMPS is what ModPlug playback sounds like - use it |
| `BASS_MUSIC_NONINTER` | No interpolation - crunchy 1994 authenticity, saves a little CPU |
| `BASS_MUSIC_SINCINTER` | Sinc interpolation - the opposite tradeoff; the Atom can afford it |
| `BASS_MUSIC_PRESCAN` | Compute playback length up front (needed before seeking by time) |
| `BASS_MUSIC_STOPBACK` | Stop at backward jumps (modules that "end" by jumping back) |
| `BASS_MUSIC_POSRESET / BASS_MUSIC_POSRESETEX` | Cut notes (and reset bpm etc. with EX) when you seek |
| `BASS_MUSIC_FT2MOD / BASS_MUSIC_PT1MOD` | .MOD quirk emulation (FastTracker2 / ProTracker1 rules) |
| `BASS_MUSIC_NOSAMPLE` | Parse only, no sample data - for a tracklist tool, not playback |

## C.4 Samples (short SFX, mixed instances)

| Signature | What it does | Notes |
|---|---|---|
| `HSAMPLE BASS_SampleLoad(DWORD filetype, const void *file, QWORD offset, DWORD length, DWORD max, DWORD flags)` | Loads an entire wav (or mp3/ogg) into memory, decoded. | `max` = simultaneous playbacks of THIS sample (1-65535). Footsteps: 4-8. Beyond max, the override flags pick a victim. |

| | | |
|---|---|---|
| `DWORD BASS_SampleGetChannel(HSAMPLE handle, DWORD flags)` | Gets a channel to play the sample on. | flags = 0 reuses/steals per the override policy - the normal case. Returns an HCHANNEL (as DWORD); 0 + BASS_ERROR_NOCHAN when max reached without an override flag. The channel starts PAUSED: set attributes, then BASS_ChannelPlay. |
| `BOOL BASS_SampleFree(HSAMPLE handle)` | Frees the sample and its channels. | |
| `BOOL BASS_SampleStop(HSAMPLE handle)` | Stops all channels of this sample. | |

Useful sample flags: `BASS_SAMPLE_LOOP` (looping - engine hum), `BASS_SAMPLE_OVER_VOL` (steal the quietest channel when full - the right default for SFX), `BASS_SAMPLE_OVER_POS` (steal the longest-playing), `BASS_SAMPLE_MONO` (halve the memory of stereo-sourced effects you will pan yourself anyway).

## C.5 Channel control (works on HSTREAM, HMUSIC, HCHANNEL)

| Signature | What it does | Notes |
|---|---|---|
| `BOOL BASS_ChannelPlay(DWORD handle, BOOL restart)` | Starts or resumes. restart=TRUE rewinds first. | The universal "go" button. |
| `BOOL BASS_ChannelStop(DWORD handle)` | Stops. A stopped sample channel is gone; a stream/ music can Play again. | |
| `BOOL BASS_ChannelPause(DWORD handle)` | Pauses; resume with Play(handle, FALSE). | BASS_ERROR_ALREADY if already paused. |
| `BOOL BASS_ChannelSetAttribute(DWORD handle, DWORD attrib, float value)` / `BASS_ChannelGetAttribute(..., float *value)` | Sets/reads a live attribute (below). | Takes effect immediately; no need to stop. |
| `BOOL BASS_ChannelSlideAttribute(DWORD handle, DWORD attrib, float value, DWORD time)` | Ramps an attribute over `time` ms. | Free crossfades: slide VOL to 0, then stop. |
| `DWORD BASS_ChannelIsActive(DWORD handle)` | Playback state. | Returns BASS_ACTIVE_STOPPED (0), BASS_ACTIVE_PLAYING (1), BASS_ACTIVE_STALLED (2, starved stream), BASS_ACTIVE_PAUSED (3), BASS_ACTIVE_PAUSED_DEVICE (4). |
| `BOOL BASS_ChannelSetPosition(DWORD handle, QWORD pos, DWORD mode)` / `QWORD BASS_ChannelGetPosition(DWORD handle, DWORD mode)` | Seek / tell. | mode = BASS_POS_BYTE for streams; BASS_POS_MUSIC_ORDER for modules, where pos = `MAKELONG(order, row)` - jump straight to pattern positions for interactive music. GetPosition with BASS_POS_MUSIC_ORDER returns order in LOWORD, row in HIWORD. |

| | | |
|---|---|---|
| `QWORD BASS_ChannelGetLength(DWORD handle, DWORD mode)` | Length in bytes (or orders for music with BASS_POS_MUSIC_ORDER). | |
| `double BASS_ChannelBytes2Seconds(DWORD handle, QWORD pos)` / `QWORD BASS_ChannelSeconds2Bytes(DWORD handle, double pos)` | Convert byte positions to seconds and back. | Seek to 30s: SetPosition(h, Seconds2Bytes(h, 30.0), BASS_POS_BYTE). |
| `DWORD BASS_ChannelGetLevel(DWORD handle)` | Instant level: left in LOWORD, right in HIWORD, 0-32768. | Free VU meter for the console. |

Attributes for SetAttribute:

| Attribute | Range | Meaning |
|---|---|---|
| `BASS_ATTRIB_VOL` | 0..1 (above 1 amplifies) | Channel volume - your distance attenuation lives here |
| `BASS_ATTRIB_PAN` | -1 (left) .. +1 (right) | Stereo position - your listener panning lives here |
| `BASS_ATTRIB_FREQ` | 100..100000 (Hz) | Playback rate: pitch and speed together. 44100 * 0.8 = big monster footstep from the same wav |
| `BASS_ATTRIB_MUSIC_AMPLIFY` | 0..100 | Module master amplification |
| `BASS_ATTRIB_MUSIC_SPEED` | 0..255 | Module ticks-per-row speed - slow the soundtrack when the player is dying, 1999 style |
| `BASS_ATTRIB_MUSIC_VOL_CHAN + n` | 0..1 | Per-tracker-channel volume: fade instrument layers in and out for "dynamic music" on a budget |

## C.6 Error codes (BASS_ErrorGetCode)

| Code | Value | Plain English |
|---|---|---|
| BASS_OK | 0 | No error |
| BASS_ERROR_MEM | 1 | Out of memory |
| BASS_ERROR_FILEOPEN | 2 | File not found or unreadable - check your working directory first |
| BASS_ERROR_DRIVER | 3 | No usable output driver (broken sound setup) |
| BASS_ERROR_BUFLOST | 4 | DirectSound buffer lost (alt-tab-era problem) |
| BASS_ERROR_HANDLE | 5 | Bad handle - you freed it or never checked the 0 return at load |
| BASS_ERROR_FORMAT | 6 | Sample format not supported (odd wav encodings) |
| BASS_ERROR_POSITION | 7 | Seek position out of range or wrong mode |
| BASS_ERROR_INIT | 8 | BASS_Init was not called (or failed and you ignored it) |
| BASS_ERROR_START | 9 | Output is stopped: call BASS_Start |
| BASS_ERROR_REINIT | 11 | Device needs reinitialization |
| BASS_ERROR_ALREADY | 14 | Already done: double Init, double pause, etc. |
| BASS_ERROR_NOTAUDIO | 17 | File has no audio track |
| BASS_ERROR_NOCHAN | 18 | No free channel: sample hit its `max` without an OVER_* flag |

| | | |
|---|---|---|
| BASS_ERROR_ILLTYPE | 19 | Illegal type argument (wrong attrib constant, usually) |
| BASS_ERROR_ILLPARAM | 20 | Illegal parameter - range error, wrong flag combination |
| BASS_ERROR_NO3D | 21 | No 3D support (you did not pass BASS_SAMPLE_3D; this book pans by hand anyway) |
| BASS_ERROR_DEVICE | 23 | Bad device index |
| BASS_ERROR_NOPLAY | 24 | Channel is not playing |
| BASS_ERROR_FREQ | 25 | Frequency out of range (ATTRIB_FREQ too wild) |
| BASS_ERROR_NOTFILE | 27 | Not a file stream |
| BASS_ERROR_NOHW | 29 | No hardware voices free (ancient DirectSound; harmless with software mixing) |
| BASS_ERROR_EMPTY | 31 | File has no sample data |
| BASS_ERROR_NONET | 32 | No internet - irrelevant in the cabin |
| BASS_ERROR_CREATE | 33 | Could not create file (writer functions) |
| BASS_ERROR_NOFX | 34 | DX8 effects unavailable |
| BASS_ERROR_NOTAVAIL | 37 | Requested data/action not available (e.g. level of a stopped channel) |
| BASS_ERROR_DECODE | 38 | Channel is (or is not) a decoding channel - you mixed up DECODE streams |
| BASS_ERROR_DX | 39 | DirectX version too old - should not happen on XP SP3 |
| BASS_ERROR_TIMEOUT | 40 | Connection timeout (network) |
| BASS_ERROR_FILEFORM | 41 | Unrecognized file format - the wrong extension trap: BASS_StreamCreateFile does not read .xm, BASS_MusicLoad does not read .wav |
| BASS_ERROR_SPEAKER | 42 | Speaker assignment unavailable |
| BASS_ERROR_VERSION | 43 | Add-on/DLL version mismatch |
| BASS_ERROR_CODEC | 44 | Codec missing - mp3 needs the OS ACM on XP; wav/ogg always work |
| BASS_ERROR_ENDED | 45 | Channel/file already ended |
| BASS_ERROR_BUSY | 46 | Device busy (exclusive-mode squatter) |
| BASS_ERROR_UNKNOWN | -1 | Mystery. Check the obvious thing you are sure is fine; it is not |

## C.7 Worked example: engine_audio.c

```
/* engine_audio.c - day 21: music + panned SFX over BASS 2.4 */
#include <windows.h>
#include <string.h>
#include <math.h>
#include "bass.h"

#define AUDIO_MAX_SFX   32
#define AUDIO_MIN_DIST  4.0f    /* full volume inside this radius  */
#define AUDIO_MAX_DIST  60.0f   /* silent beyond this radius       */

typedef struct {
    char    path[128];
    HSAMPLE sample;
} audio_sfx_t;

static audio_sfx_t g_sfx[AUDIO_MAX_SFX];
static int         g_sfx_count = 0;
static HMUSIC      g_music = 0;
```

```
int Audio_Init(HWND hwnd)
{
    if (HIWORD(BASS_GetVersion()) != BASSVERSION) {
        MessageBoxA(0, "wrong bass.dll version", "audio", MB_OK);
        return -1;
    }
    if (!BASS_Init(-1, 44100, 0, hwnd, NULL)) {
        /* code 23 = bad device, 3 = no driver: see table C.6 */
        MessageBoxA(0, "BASS_Init failed", "audio", MB_OK);
        return -1;
    }
    return 0;
}

void Audio_Shutdown(void)
{
    BASS_Free();                /* frees music + samples too */
}

int Audio_PlayMusic(const char *path)
{
    if (g_music) BASS_MusicFree(g_music);
    g_music = BASS_MusicLoad(BASS_FILE_NAME, path, 0, 0,
                             BASS_MUSIC_LOOP | BASS_MUSIC_RAMPS, 0);
    if (!g_music) return BASS_ErrorGetCode();
    BASS_ChannelSetAttribute(g_music, BASS_ATTRIB_VOL, 0.7f);
    BASS_ChannelPlay(g_music, FALSE);
    return 0;
}

/* refcount-free cache: same path returns same sample (day 20 wraps
   this in the real resource manager) */
int Audio_LoadSfx(const char *path)
{
    int i;
    for (i = 0; i < g_sfx_count; ++i)
        if (strcmp(g_sfx[i].path, path) == 0) return i;
    if (g_sfx_count == AUDIO_MAX_SFX) return -1;
    g_sfx[i].sample = BASS_SampleLoad(BASS_FILE_NAME, path, 0, 0,
                                      8, BASS_SAMPLE_OVER_VOL);
    if (!g_sfx[i].sample) return -1;
    strcpy(g_sfx[i].path, path);
    return g_sfx_count++;
}

/* emitter position vs listener position + right vector (both from
   the day-3 camera). Linear falloff, dot-product pan. */
void Audio_PlaySfxAt(int id, const float pos[3],
                     const float listener[3], const float right[3])
{
    DWORD ch;
    float d[3], dist, vol, pan;

    if (id < 0) return;
    d[0] = pos[0] - listener[0];
    d[1] = pos[1] - listener[1];
    d[2] = pos[2] - listener[2];
    dist = (float)sqrt(d[0]*d[0] + d[1]*d[1] + d[2]*d[2]);

    vol = 1.0f - (dist - AUDIO_MIN_DIST) /
                 (AUDIO_MAX_DIST - AUDIO_MIN_DIST);
    if (vol <= 0.0f) return;          /* out of earshot: skip */
    if (vol > 1.0f) vol = 1.0f;

    pan = 0.0f;
    if (dist > 0.001f)                /* right-vector projection */
```

```
        pan = (d[0]*right[0] + d[1]*right[1] + d[2]*right[2]) / dist;

    ch = BASS_SampleGetChannel(g_sfx[id].sample, 0);
    if (!ch) return;                  /* NOCHAN: voices all busy */
    BASS_ChannelSetAttribute(ch, BASS_ATTRIB_VOL, vol);
    BASS_ChannelSetAttribute(ch, BASS_ATTRIB_PAN, pan);
    BASS_ChannelPlay(ch, FALSE);
}
```

**TIP:** BASS mixes on its own thread; there is nothing to pump in your frame loop. For looping emitters (a waterfall), keep the HCHANNEL and re-run the volume/pan math each frame with BASS_ChannelSetAttribute - it is just two floats a frame.

# Appendix D: Nuklear In One Evening

Nuklear is a single-header immediate-mode UI in C89 - your compiler's native tongue. It draws nothing by itself: each frame it consumes input, you declare windows and widgets, and it hands you vertices to render with the day-2 machinery. One evening gets you sliders that tune fog while you fly around. This appendix is self-contained: setup, input, widgets, and the complete GLES2 backend.

## D.1 Compile-time setup for VC6

Nuklear is configured by macros that must be IDENTICAL in every translation unit that includes it. Enforce that with a wrapper header nobody is allowed to bypass:

```
/* nk_config.h - the only way into nuklear.h in this codebase */
#ifndef NK_CONFIG_H
#define NK_CONFIG_H

#define NK_INCLUDE_DEFAULT_ALLOCATOR      /* malloc-backed contexts  */
#define NK_INCLUDE_FONT_BAKING            /* stb_truetype baker      */
#define NK_INCLUDE_DEFAULT_FONT           /* built-in ProggyClean    */
#define NK_INCLUDE_VERTEX_BUFFER_OUTPUT   /* nk_convert + draw lists */
/* NK_INCLUDE_FIXED_TYPES must NEVER appear: it includes
   <stdint.h>, which VC6 does not have. Without it nuklear
   derives nk_byte/nk_ushort/etc. itself, which works fine. */

#include "nuklear.h"
#endif
```

```
/* nk_impl.c - the implementation lives here and ONLY here */
#define NK_IMPLEMENTATION
#include "nk_config.h"
```

**WARNING:** Define `NK_IMPLEMENTATION` in exactly one .c file, and never define `NK_INCLUDE_FIXED_TYPES` anywhere on VC6 - the compiler predates `stdint.h` by five years and the include fails instantly. If two files disagree on the NK_INCLUDE_* set you get struct-layout mismatches that crash at runtime, not compile time. The wrapper header exists to make that impossible.

Without `NK_INCLUDE_STANDARD_VARARGS` you lose `nk_labelf()`; call `_snprintf` into a local buffer and use `nk_label()` - more C89 anyway.

## D.2 The frame loop and Win32 wiring

Nuklear wants all of a frame's input between `nk_input_begin` and `nk_input_end`, BEFORE you declare any UI. Your skeleton's message pump slots in perfectly:

```
/* in the main loop, once per frame, before engine_update() */
nk_input_begin(&g_ctx);
while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
    TranslateMessage(&msg);
    DispatchMessage(&msg);     /* WndProc feeds nuklear below */
}
nk_input_end(&g_ctx);

ui_frame();          /* declare windows/widgets (D.3)          */
```

```
engine_update(dt);   /* game may read ui state set in ui_frame */
engine_render();     /* 3D first...                            */
ui_render(w, h);     /* ...UI on top (D.4)                     */
eglSwapBuffers(g_dpy, g_surf);
```

And in WndProc, forward what nuklear cares about (coordinates are client-area pixels, y down - exactly what nuklear expects):

```
case WM_MOUSEMOVE:
    nk_input_motion(&g_ctx, (short)LOWORD(lParam),
                    (short)HIWORD(lParam));
    return 0;
case WM_LBUTTONDOWN: case WM_LBUTTONUP:
    nk_input_button(&g_ctx, NK_BUTTON_LEFT,
                    (short)LOWORD(lParam), (short)HIWORD(lParam),
                    msg == WM_LBUTTONDOWN);
    return 0;
case WM_RBUTTONDOWN: case WM_RBUTTONUP:
    nk_input_button(&g_ctx, NK_BUTTON_RIGHT,
                    (short)LOWORD(lParam), (short)HIWORD(lParam),
                    msg == WM_RBUTTONDOWN);
    return 0;
case WM_MOUSEWHEEL:
    nk_input_scroll(&g_ctx,
        nk_vec2(0.0f, (float)((short)HIWORD(wParam)) / 120.0f));
    return 0;
case WM_CHAR:
    if (wParam >= 32 && wParam < 127)
        nk_input_char(&g_ctx, (char)wParam);
    return 0;
case WM_KEYDOWN: case WM_KEYUP:
    {
        int dn = (msg == WM_KEYDOWN);
        switch (wParam) {
        case VK_BACK:   nk_input_key(&g_ctx, NK_KEY_BACKSPACE, dn);
                        break;
        case VK_RETURN: nk_input_key(&g_ctx, NK_KEY_ENTER, dn);
                        break;
        case VK_DELETE: nk_input_key(&g_ctx, NK_KEY_DEL, dn);
                        break;
        case VK_LEFT:   nk_input_key(&g_ctx, NK_KEY_LEFT, dn);
                        break;
        case VK_RIGHT:  nk_input_key(&g_ctx, NK_KEY_RIGHT, dn);
                        break;
        }
    }
    break;  /* fall to DefWindowProc so Esc still quits */
```

**NOTE:** Call nk_input_begin/nk_input_end every frame even when no messages arrived - nuklear derives "still held" and "released" from the bracketing. And when the day-3 camera has captured the mouse (recentering), skip feeding nuklear motion or your UI will fight your mouselook. One boolean, two masters.

## D.3 A window with widgets

```
/* ui_frame: declare the UI. Immediate mode: this runs every
   frame and IS the UI. */
static float g_fog_density = 0.02f;
static nk_bool g_show_wire = nk_false;

void ui_frame(void)
```

```
{
    if (nk_begin(&g_ctx, "engine tune",
                 nk_rect(10.0f, 10.0f, 240.0f, 180.0f),
                 NK_WINDOW_BORDER | NK_WINDOW_MOVABLE |
                 NK_WINDOW_TITLE | NK_WINDOW_MINIMIZABLE)) {

        /* one column, 30 px rows, full window width */
        nk_layout_row_dynamic(&g_ctx, 30.0f, 1);

        nk_label(&g_ctx, "fog density", NK_TEXT_LEFT);
        nk_slider_float(&g_ctx, 0.0f, &g_fog_density,
                        0.10f, 0.001f);

        nk_checkbox_label(&g_ctx, "wireframe", &g_show_wire);

        if (nk_button_label(&g_ctx, "respawn")) {
            /* fires on the frame the button is released */
            player_respawn();
        }
    }
    nk_end(&g_ctx);   /* ALWAYS call nk_end, even if nk_begin
                         returned false (window minimized) */
}
```

That is the whole model: `nk_begin`/`nk_end` bracket a window; `nk_layout_row_dynamic(ctx, height, cols)` starts a row layout; each widget call both draws and reports interaction. State lives in YOUR variables - nuklear holds almost nothing. Wire `g_fog_density` straight into the day-17 fog uniform and you have a live-tuning panel.

## D.4 The GLES2 render backend, complete

Nuklear outputs draw commands; this converts them to triangles and pushes them through GL. The vertex layout is described to `nk_convert` so it packs vertices exactly like our C struct:

```
/* ui_render.c - nuklear GLES2 backend, complete */
#include <string.h>
#include "gl_loader.h"
#include "nk_config.h"

struct ui_vertex {
    float   pos[2];
    float   uv[2];
    nk_byte col[4];
};

#define UI_MAX_VERTS (512 * 1024)   /* bytes */
#define UI_MAX_ELEMS (128 * 1024)   /* bytes */

static struct nk_context           g_ctx;
static struct nk_font_atlas        g_atlas;
static struct nk_buffer            g_cmds;
static struct nk_draw_null_texture g_null;
static GLuint g_font_tex, g_prog, g_vbo, g_ebo;
static GLint  g_uproj, g_utex;

static const struct nk_draw_vertex_layout_element UI_LAYOUT[] = {
    {NK_VERTEX_POSITION, NK_FORMAT_FLOAT,
        NK_OFFSETOF(struct ui_vertex, pos)},
    {NK_VERTEX_TEXCOORD, NK_FORMAT_FLOAT,
        NK_OFFSETOF(struct ui_vertex, uv)},
    {NK_VERTEX_COLOR, NK_FORMAT_R8G8B8A8,
        NK_OFFSETOF(struct ui_vertex, col)},
    {NK_VERTEX_LAYOUT_END}
```

```
};

static const char *UI_VS =
"uniform mat4 u_proj;\n"
"attribute vec2 a_pos;\n"
"attribute vec2 a_uv;\n"
"attribute vec4 a_col;\n"
"varying vec2 v_uv;\n"
"varying vec4 v_col;\n"
"void main() {\n"
"    v_uv = a_uv;\n"
"    v_col = a_col;\n"
"    gl_Position = u_proj * vec4(a_pos, 0.0, 1.0);\n"
"}\n";

static const char *UI_FS =
"precision mediump float;\n"
"uniform sampler2D u_tex;\n"
"varying vec2 v_uv;\n"
"varying vec4 v_col;\n"
"void main() {\n"
"    gl_FragColor = v_col * texture2D(u_tex, v_uv);\n"
"}\n";

int ui_init(void)
{
    const void *image;
    int w, h;
    struct nk_font *font;

    /* shader_build: your day-2 compile/bind/link helper. Bind
       a_pos=0, a_uv=1, a_col=2 BEFORE glLinkProgram. */
    g_prog  = shader_build(UI_VS, UI_FS);
    g_uproj = glGetUniformLocation(g_prog, "u_proj");
    g_utex  = glGetUniformLocation(g_prog, "u_tex");
    glGenBuffers(1, &g_vbo);
    glGenBuffers(1, &g_ebo);

    /* --- font atlas: bake ProggyClean into one RGBA texture --- */
    nk_font_atlas_init_default(&g_atlas);
    nk_font_atlas_begin(&g_atlas);
    font  = nk_font_atlas_add_default(&g_atlas, 13.0f, NULL);
    image = nk_font_atlas_bake(&g_atlas, &w, &h,
                               NK_FONT_ATLAS_RGBA32);
    glGenTextures(1, &g_font_tex);
    glBindTexture(GL_TEXTURE_2D, g_font_tex);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, w, h, 0, GL_RGBA,
                 GL_UNSIGNED_BYTE, image);
    /* hands the GL texture to the atlas, frees CPU-side pixels,
       and fills g_null (the white pixel used for flat shapes) */
    nk_font_atlas_end(&g_atlas, nk_handle_id((int)g_font_tex),
                      &g_null);

    nk_init_default(&g_ctx, &font->handle);
    nk_buffer_init_default(&g_cmds);
    return 0;
}

void ui_render(int width, int height)
{
    struct nk_convert_config cfg;
    struct nk_buffer vbuf, ebuf;
    const struct nk_draw_command *cmd;
    const nk_byte *offset = 0;
```

```
    GLsizei stride = (GLsizei)sizeof(struct ui_vertex);
    GLfloat proj[16];
    /* static: 640 KB does not belong on a VC6 stack */
    static char verts[UI_MAX_VERTS];
    static char elems[UI_MAX_ELEMS];

    /* column-major ortho, origin TOP-left, y down - UI space */
    memset(proj, 0, sizeof(proj));
    proj[0]  =  2.0f / (GLfloat)width;
    proj[5]  = -2.0f / (GLfloat)height;
    proj[10] = -1.0f;
    proj[12] = -1.0f;
    proj[13] =  1.0f;
    proj[15] =  1.0f;

    /* GL state for UI: blend on, everything 3D off */
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glDisable(GL_DEPTH_TEST);
    glDisable(GL_CULL_FACE);
    glEnable(GL_SCISSOR_TEST);
    glUseProgram(g_prog);
    glUniformMatrix4fv(g_uproj, 1, GL_FALSE, proj);
    glUniform1i(g_utex, 0);
    glActiveTexture(GL_TEXTURE0);

    /* convert nuklear's command list into our vertex format */
    memset(&cfg, 0, sizeof(cfg));
    cfg.global_alpha         = 1.0f;
    cfg.shape_AA             = NK_ANTI_ALIASING_ON;
    cfg.line_AA              = NK_ANTI_ALIASING_ON;
    cfg.circle_segment_count = 22;
    cfg.arc_segment_count    = 22;
    cfg.curve_segment_count  = 22;
    cfg.tex_null             = g_null;
    cfg.vertex_layout        = UI_LAYOUT;
    cfg.vertex_size          = sizeof(struct ui_vertex);
    cfg.vertex_alignment     = NK_ALIGNOF(struct ui_vertex);

    nk_buffer_init_fixed(&vbuf, verts, UI_MAX_VERTS);
    nk_buffer_init_fixed(&ebuf, elems, UI_MAX_ELEMS);
    nk_convert(&g_ctx, &g_cmds, &vbuf, &ebuf, &cfg);

    /* upload: orphan, then copy only what this frame used */
    glBindBuffer(GL_ARRAY_BUFFER, g_vbo);
    glBufferData(GL_ARRAY_BUFFER, UI_MAX_VERTS, NULL,
                 GL_STREAM_DRAW);
    glBufferSubData(GL_ARRAY_BUFFER, 0,
                    (GLsizeiptr)vbuf.needed, verts);
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, g_ebo);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER, UI_MAX_ELEMS, NULL,
                 GL_STREAM_DRAW);
    glBufferSubData(GL_ELEMENT_ARRAY_BUFFER, 0,
                    (GLsizeiptr)ebuf.needed, elems);

    glEnableVertexAttribArray(0);
    glEnableVertexAttribArray(1);
    glEnableVertexAttribArray(2);
    glVertexAttribPointer(0, 2, GL_FLOAT, GL_FALSE, stride,
        (const void *)NK_OFFSETOF(struct ui_vertex, pos));
    glVertexAttribPointer(1, 2, GL_FLOAT, GL_FALSE, stride,
        (const void *)NK_OFFSETOF(struct ui_vertex, uv));
    glVertexAttribPointer(2, 4, GL_UNSIGNED_BYTE, GL_TRUE, stride,
        (const void *)NK_OFFSETOF(struct ui_vertex, col));

    /* one glDrawElements per nuklear command; clip_rect is in
```

```
       top-left coords, glScissor wants bottom-left: flip y */
    nk_draw_foreach(cmd, &g_ctx, &g_cmds) {
        GLint sy;
        if (!cmd->elem_count) continue;
        sy = height - (GLint)(cmd->clip_rect.y +
                              cmd->clip_rect.h);
        if (sy < 0) sy = 0;
        glScissor((GLint)cmd->clip_rect.x, sy,
                  (GLsizei)cmd->clip_rect.w,
                  (GLsizei)cmd->clip_rect.h);
        glBindTexture(GL_TEXTURE_2D, (GLuint)cmd->texture.id);
        glDrawElements(GL_TRIANGLES, (GLsizei)cmd->elem_count,
                       GL_UNSIGNED_SHORT, (const void *)offset);
        offset += cmd->elem_count * sizeof(nk_draw_index);
    }

    nk_clear(&g_ctx);            /* MANDATORY once per frame  */
    nk_buffer_clear(&g_cmds);

    /* hand state back to the 3D renderer */
    glDisable(GL_SCISSOR_TEST);
    glDisable(GL_BLEND);
    glEnable(GL_DEPTH_TEST);
    glDisableVertexAttribArray(2);
}

void ui_shutdown(void)
{
    nk_font_atlas_clear(&g_atlas);
    nk_free(&g_ctx);
    nk_buffer_free(&g_cmds);
    glDeleteTextures(1, &g_font_tex);
    glDeleteBuffers(1, &g_vbo);
    glDeleteBuffers(1, &g_ebo);
    glDeleteProgram(g_prog);
}
```

Why each piece is the way it is:

| Piece | Reason |
|---|---|
| UI_LAYOUT + NK_OFFSETOF | Tells nk_convert to emit vertices with OUR struct's exact offsets; the glVertexAttribPointer calls quote the same offsets, so they can never drift apart. |
| NK_FORMAT_R8G8B8A8 + normalized GL_UNSIGNED_BYTE | 4-byte colors instead of 16: a 20-byte vertex, kind to the Atom's memory bus. |
| tex_null | Flat rectangles sample a known white pixel so one shader draws both text and shapes - no program switching inside the UI pass. |
| Orphan + SubData with `.needed` | glBufferData with NULL tells the driver "old contents dead", avoiding a stall on buffers rewritten every frame; then upload only the bytes nuklear actually produced. |
| GL_UNSIGNED_SHORT | `nk_draw_index` is 16-bit (you did not define NK_UINT_DRAW_INDEX). Matches the GLES2 hard rule anyway. |
| Scissor y-flip | nuklear clips in top-left window coords; glScissor is bottom-left. Skip the flip and your UI draws exactly one window title and nothing else - the single most common nuklear bug. |
| nk_clear | Resets the per-frame command list. Forget it and the second frame asserts or the UI freezes solid. |

**TIP:** If you see no pixels: (1) check the FS compiled - `precision mediump float;` present? (2) disable the scissor test entirely for one run; if the UI appears, your y-flip is wrong; (3) draw with the font texture forced to verify geometry; (4) confirm nk_input_begin/end bracket the pump, or windows render but ignore the mouse. In that order. It is always one of these four.

**NOTE:** Budget: this backend costs one program, two buffer uploads, and a handful of draw calls per window - noise next to the 3D scene. Anti-aliasing is the expensive part of nk_convert on the CPU; if the console (day 19) plus a tuning panel ever shows up in the profile, set shape_AA/line_AA to NK_ANTI_ALIASING_OFF and enjoy the 1999 crispness.

# Appendix E: File Formats

Everything the engine reads from disk is specified here, byte by byte, so you never need another document. Every multi-byte value in every format in this appendix is **little-endian**, which is also the native order of your Atom, so a plain read works. All offsets are in bytes and count from the start of the structure being described unless the table says otherwise.

## E.1 Reading binary files without getting burned

One rule above all: **never `fread` a whole struct.** VC6 pads structures to 8-byte alignment by default, so a struct you write to mirror a file layout will usually be bigger than the data on disk, and every field after the first misaligned one reads garbage. `#pragma pack(1)` exists, but it is a loaded gun that follows you around the codebase. Read field by field through four tiny helpers instead; every loader in this book uses them.

```
static unsigned char read_u8(FILE *f)
{
    return (unsigned char)fgetc(f);
}

static unsigned short read_u16(FILE *f)
{
    unsigned int lo, hi;
    lo = read_u8(f);
    hi = read_u8(f);
    return (unsigned short)(lo | (hi << 8));
}

static unsigned int read_u32(FILE *f)
{
    unsigned int b0, b1, b2, b3;
    b0 = read_u8(f); b1 = read_u8(f);
    b2 = read_u8(f); b3 = read_u8(f);
    return b0 | (b1 << 8) | (b2 << 16) | (b3 << 24);
}

static float read_f32(FILE *f)
{
    union { unsigned int u; float fl; } cv;
    cv.u = read_u32(f);
    return cv.fl;
}
```

`(int)read_u32(f)` and `(short)read_u16(f)` cover the signed cases. Error handling is trimmed: check `feof` once per section, not per byte, and bail with a `MessageBoxA` naming the file. Always open with `"rb"` — text mode eats byte 0x1A and translates 0x0D 0x0A pairs, and the resulting bug reports look like haunted files.

## E.2 TGA (Targa)

The reference decoder is on Day 4. This section is the byte-exact spec it implements, plus the edge cases. TGA survives because it is the simplest truecolor format that supports alpha and a lossless RLE, and because Paint Shop Pro 7 writes it natively.

### The 18-byte header

| offset | size | type | name | meaning |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 0 | 1 | u8 | idLength | bytes of free-form ID text after the header; skip them |
| 1 | 1 | u8 | colorMapType | 0 = no palette, 1 = palette present |
| 2 | 1 | u8 | imageType | see the type table below; accept 2 and 10 only |
| 3 | 2 | u16 | cmFirstIndex | palette: first index (only needed to skip it) |
| 5 | 2 | u16 | cmLength | palette: number of entries |
| 7 | 1 | u8 | cmEntrySize | palette: bits per entry (15/16/24/32) |
| 8 | 2 | u16 | xOrigin | intended screen position; ignore |
| 10 | 2 | u16 | yOrigin | intended screen position; ignore |
| 12 | 2 | u16 | width | pixels |
| 14 | 2 | u16 | height | pixels |
| 16 | 1 | u8 | bitsPerPixel | 24 or 32 accepted; 16 rejected (see E.2 edge cases) |
| 17 | 1 | u8 | descriptor | bit field, next table |

### Image types

| value | meaning | your loader |
|---|---|---|
| 0 | no image data | reject |
| 1 | color-mapped (paletted) | reject politely |
| 2 | uncompressed truecolor | **accept** |
| 3 | uncompressed grayscale | reject |
| 9 | RLE color-mapped | reject politely |
| 10 | RLE truecolor | **accept** |
| 11 | RLE grayscale | reject |

### The descriptor byte (offset 17)

| bits | meaning |
|---|---|
| 0–3 | alpha bits per pixel: 8 for 32-bit images, 0 for 24-bit. Some tools write 0 even for 32-bit files — trust `bitsPerPixel`, not this field. |
| 4 | 1 = pixels run right-to-left. Essentially never set; treat as corrupt if it is. |
| 5 | 1 = rows run top-to-bottom. 0 = **bottom-up**, the common case. GL wants the bottom row first, so bottom-up files upload as-is; top-down files need a row flip. |
| 6–7 | ancient interleave flags; must be 0 |

### Finding and reading the pixels

Pixel data starts at:

```
offset = 18 + idLength
       + (colorMapType ? cmLength * ((cmEntrySize + 7) / 8) : 0)
```

Pixels are stored **BGR** (24-bit) or **BGRA** (32-bit). Swapping B and R on load is the Day 4 rite of passage; a face-textured cube with blue skin means you skipped it.

**Type 2** is `width*height` raw pixels, row by row. **Type 10** is a stream of RLE packets, each starting with one header byte:

| header bit 7 | packet | count | payload |
|---|---|---|---|
| 1 | run | `(hdr & 0x7F) + 1` (1..128) | ONE pixel value, repeated count times |
| 0 | raw | `hdr + 1` (1..128) | count literal pixels |

Decode until you have produced exactly `width*height` pixels. The written spec says packets must not cross scanline boundaries; several era tools cross them anyway. Decode into one flat buffer and the question never arises.

### The footer: ignore it

Tools that follow the TGA 2.0 spec append an extension area and a 26-byte footer ending in the string `TRUEVISION-XFILE.` plus a NUL. All of it sits *after* the pixel data. A loader that reads header, then pixels, then stops, never encounters it. Do not seek to the end looking for it; there is nothing there you need.

### Edge cases

- **16-bit files** (A1R5G5B5): reject with a clear message. PSP7 can re-save any image as 24-bit in two clicks; the decode code you would write for 15-bit fields would be used exactly once.
- **Color-mapped files** (types 1 and 9): reject politely — the MessageBox should say "re-save as truecolor", not "bad file". Note a file can have `colorMapType == 1` with `imageType == 2`; the palette is then dead weight, and the offset formula above skips it correctly.
- **Sanity limits:** width or height of 0, or larger than 4096, means corrupt. Fail early; a garbage u16 read as 53000 will otherwise turn into a 10 GB malloc on a 1 GB machine.
- **Upside-down textures:** descriptor bit 5. Check it before blaming your UVs.

## E.3 OBJ and MTL

Wavefront OBJ is a plain-text format: one statement per line, first token is the keyword. The Day 7 loader implements the subset below, which is everything Milkshape 3D 1.8.4 exports and everything you will meet in the wild.

### Whitespace and comment rules

- Tokens are separated by any run of spaces or tabs.
- `#` starts a comment that runs to end of line.
- Blank lines are legal anywhere.
- Lines end in `\n` or `\r\n`; strip the `\r` (Milkshape writes Windows line endings, naturally).
- The official grammar allows a trailing backslash to continue a line. No tool you own writes one. Ignore the rule.
- Unknown keywords: skip the line. OBJ has dozens of exotic statements (curves, surfaces); a robust loader is one that ignores well.

### Line grammar

| keyword | arguments | meaning |
|---|---|---|
| `v` | x y z [w] | vertex position; w (rare) defaults to 1, ignore it |
| `vt` | u [v] [w] | texture coordinate; v defaults to 0, w ignored |
| `vn` | x y z | normal; not guaranteed unit length — normalize on load |
| `f` | 3+ index groups | polygon face; triangulate as a fan (see below) |

| | | |
|---|---|---|
| `o` | name | object name; organizational, safe to ignore |
| `g` | name | group name; same |
| `s` | 1..32 or `off` | smoothing group; ignore — your normals come from `vn` |
| `usemtl` | name | faces that follow use this material |
| `mtllib` | file.mtl | load the material library; path is relative to the OBJ |

### The four face syntaxes

| form | example | provides |
|---|---|---|
| v | `f 1 2 3` | position only |
| v/vt | `f 1/1 2/2 3/3` | position + texcoord |
| v//vn | `f 1//1 2//2 3//3` | position + normal |
| v/vt/vn | `f 1/1/1 2/2/2 3/3/3` | all three |

A well-behaved file uses one form throughout. Do not rely on that; parse each index group independently. A face may have more than three groups — triangulate an n-gon as a fan: corners `(0, i, i+1)` for `i` from 1 to n-2. This is correct for convex polygons, which is all Milkshape emits.

### Indices: 1-based and negative

OBJ indices start at **1**, not 0, and `v`, `vt`, `vn` are indexed independently — that independence is why Day 7 dedups triples into one vertex buffer. A **negative** index counts back from the most recently defined element: -1 is the last `v` (or `vt`, `vn`) so far. Convert at parse time:

```
/* i as parsed, count = elements of that type seen so far */
idx = (i > 0) ? (i - 1) : (count + i);   /* 0-based result */
```

### MTL subset

| keyword | arguments | meaning |
|---|---|---|
| `newmtl` | name | begin a material; name is what `usemtl` refers to |
| `Kd` | r g b | diffuse color, each 0..1 |
| `map_Kd` | file | diffuse texture, path relative to the MTL |

Everything else you will see — `Ka`, `Ks`, `Ns`, `d`, `Tr`, `illum`, `map_bump` — skip the line. Your lighting comes from your shaders (Days 13–15), not from a 1988 material model.

## E.4 MD2 (Quake II models)

MD2 is keyframe vertex animation: every frame stores every vertex position, compressed to 3 bytes each. It is 1997's answer to "how do we animate a character in 8 MB of VRAM", and Day 23 lerps between its frames in the vertex shader. Magic: the first four bytes are `"IDP2"`, which read as a little-endian int is **844121161** (0x32504449). Version must be **8**.

### Header: 17 ints, 68 bytes

| offset | size | type | name | meaning |
|---|---|---|---|---|

| 0 | 4 | s32 | ident | 844121161 ("IDP2") |
|---|---|---|---|---|
| 4 | 4 | s32 | version | must be 8 |
| 8 | 4 | s32 | skinwidth | skin texture width in pixels |
| 12 | 4 | s32 | skinheight | skin texture height in pixels |
| 16 | 4 | s32 | framesize | bytes per frame = 40 + 4*num_xyz |
| 20 | 4 | s32 | num_skins | skin name count |
| 24 | 4 | s32 | num_xyz | vertices per frame |
| 28 | 4 | s32 | num_st | texture coordinate count (independent of num_xyz) |
| 32 | 4 | s32 | num_tris | triangle count |
| 36 | 4 | s32 | num_glcmds | GL command list size in ints — ignored |
| 40 | 4 | s32 | num_frames | animation frame count |
| 44 | 4 | s32 | ofs_skins | file offset: skin names, num_skins * char[64] |
| 48 | 4 | s32 | ofs_st | file offset: texture coordinates |
| 52 | 4 | s32 | ofs_tris | file offset: triangles |
| 56 | 4 | s32 | ofs_frames | file offset: first frame |
| 60 | 4 | s32 | ofs_glcmds | file offset: GL commands — ignored |
| 64 | 4 | s32 | ofs_end | file size; use as a sanity check |

Sections may appear in any order; always `fseek` to the `ofs_` values instead of assuming they follow the header.

### Texture coordinates (at ofs_st)

`num_st` records of `{ s16 s; s16 t; }` — 4 bytes each, in **texels**. Convert to UV by dividing: `u = s / (float)skinwidth`, `v = t / (float)skinheight`. MD2 counts t from the top of the image; if your texture loader delivers rows bottom-up (the Day 4 TGA path does), flip with `v = 1.0f - v`. If the skin looks like the model fell face-first into it, this is why.

### Triangles (at ofs_tris)

`num_tris` records, 12 bytes each:

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 6 | s16[3] | index_xyz | three indices into the frame vertex array |
| 6 | 6 | s16[3] | index_st | three indices into the st array |

Positions and texcoords are indexed separately — the same seam problem as OBJ, solved the same way: dedup (xyz, st) pairs into one vertex buffer (the Day 7 dedup, reused on Day 23).

### Frames (at ofs_frames)

`num_frames` frames, each exactly `framesize` bytes:

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 12 | f32[3] | scale | per-axis decompression scale |
| 12 | 12 | f32[3] | translate | per-axis decompression offset |
| 24 | 16 | char[16] | name | frame name, e.g. "run3"; NUL-padded |

| | | | | |
|---|---|---|---|---|
| 40 | 4*num_xyz | | verts | packed vertices, 4 bytes each |

Each packed vertex is `{ u8 v[3]; u8 normalIndex; }`. Decompression, per axis:

```
pos[i] = scale[i] * (float)v[i] + translate[i];   /* i = 0,1,2 */
```

> **NOTE:** `normalIndex` points into a 162-entry table of precomputed unit normals that shipped inside Quake II itself, not inside the file. You do not have that table and do not need it: ignore the byte and compute smooth normals at load time by averaging face normals per vertex, per frame — the Day 11 technique. Likewise ignore the whole `glcmds` block; it is an optimization for a 1997 rendering path (strips and fans for immediate mode) that your indexed vertex buffers replace outright.

Coordinate system: Quake II uses +X forward, +Y left, +Z up. Map into our world (+Y up, -Z forward) with `(x, y, z)_gl = (x_q, z_q, -y_q)`, which keeps the handedness. The model then faces +X; rotate the node -90° about Y if you want it marching down -Z.

### The classic animation table

Player and monster models exported with the standard Quake II frame layout use these ranges (199 frames, 0–198):

| animation | first | last | frames | fps |
|---|---|---|---|---|
| stand | 0 | 39 | 40 | 9 |
| run | 40 | 45 | 6 | 10 |
| attack | 46 | 53 | 8 | 10 |
| pain1 | 54 | 57 | 4 | 7 |
| pain2 | 58 | 61 | 4 | 7 |
| pain3 | 62 | 65 | 4 | 7 |
| jump | 66 | 71 | 6 | 7 |
| flip | 72 | 83 | 12 | 7 |
| salute | 84 | 94 | 11 | 7 |
| taunt | 95 | 111 | 17 | 10 |
| wave | 112 | 122 | 11 | 7 |
| point | 123 | 134 | 12 | 6 |
| crstnd (crouch stand) | 135 | 153 | 19 | 10 |
| crwalk (crouch walk) | 154 | 158 | 5 | 7 |
| crattak (crouch attack) | 159 | 167 | 9 | 10 |
| crpain (crouch pain) | 168 | 172 | 5 | 7 |
| crdeath (crouch death) | 173 | 177 | 5 | 5 |
| death1 | 178 | 183 | 6 | 7 |
| death2 | 184 | 189 | 6 | 7 |
| death3 | 190 | 197 | 8 | 7 |
| boom | 198 | 198 | 1 | 5 |

Check `num_frames` before trusting this table — a Milkshape export may have any frame count. The robust approach: group frames by name prefix ("run1", "run2"... share the prefix "run") and build the ranges at load time. Animation loops lerp from `last` back to `first`.

## E.5 MS3D (Milkshape 3D)

MS3D is Milkshape's native format and your Day 24 source of skeletal animation: joints, keyframes, and one bone per vertex, which is exactly period-correct. The file is a sequence of sections, each a u16 count followed by that many records. There are no offset fields; you must read the sections in order, and because the groups and joints are variable-length you cannot skip ahead.

> **WARNING:** MS3D records are laid out with **1-byte packing** and mixed field sizes — a 15-byte vertex, a 70-byte triangle, a 361-byte material. A VC6 struct with the same fields will be padded to different sizes, and `fread` of that struct desynchronizes the stream silently: the first symptom is a model that looks like a hedgehog made of triangles. Read every field individually with the E.1 helpers. No exceptions, no `#pragma pack` heroics.

### File order

| # | section | count | record size |
|---|---|---|---|
| 1 | header | — | 14 bytes |
| 2 | vertices | u16 nNumVertices | 15 bytes |
| 3 | triangles | u16 nNumTriangles | 70 bytes |
| 4 | groups | u16 nNumGroups | variable |
| 5 | materials | u16 nNumMaterials | 361 bytes |
| 6 | animation globals | — | 12 bytes |
| 7 | joints | u16 nNumJoints | variable |
| 8 | later-version extras | stop reading here | |

### Header (14 bytes)

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 10 | char[10] | id | "MS3D000000", not NUL-terminated |
| 10 | 4 | s32 | version | must be 4 (Milkshape 1.4 through 1.8.4) |

### Vertex record (15 bytes)

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 1 | u8 | flags | editor selection state; ignore |
| 1 | 12 | f32[3] | vertex | position in model space (bind pose) |
| 13 | 1 | s8 | boneId | joint index, -1 = not attached to any bone |
| 14 | 1 | u8 | refCount | editor bookkeeping; ignore |

```
for (i = 0; i < nverts; ++i) {
    (void)read_u8(f);                    /* flags    */
    verts[i].pos[0] = read_f32(f);
    verts[i].pos[1] = read_f32(f);
    verts[i].pos[2] = read_f32(f);
```

```
    verts[i].bone   = (signed char)read_u8(f);
    (void)read_u8(f);                    /* refCount */
}
```

### Triangle record (70 bytes)

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 2 | u16 | flags | ignore |
| 2 | 6 | u16[3] | vertexIndices | indices into the vertex section |
| 8 | 36 | f32[3][3] | vertexNormals | one normal per corner |
| 44 | 12 | f32[3] | s | u texcoord per corner |
| 56 | 12 | f32[3] | t | v texcoord per corner; Milkshape counts from the top — use `1.0f - t` if textures load bottom-up |
| 68 | 1 | u8 | smoothingGroup | ignore; normals are stored explicitly |
| 69 | 1 | u8 | groupIndex | redundant with the groups section |

Normals and texcoords live on triangle corners while positions and bone IDs live on vertices — so once again you dedup (position, normal, uv) combinations into one vertex buffer, Day 7 style.

### Group record (variable)

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 1 | u8 | flags | ignore |
| 1 | 32 | char[32] | name | copy out and force a NUL at [31] |
| 33 | 2 | u16 | numtriangles | n |
| 35 | 2n | u16[n] | triangleIndices | indices into the triangle section |
| 35+2n | 1 | s8 | materialIndex | -1 = no material |

### Material record (361 bytes)

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 32 | char[32] | name | |
| 32 | 16 | f32[4] | ambient | RGBA, 0..1 |
| 48 | 16 | f32[4] | diffuse | RGBA, 0..1 |
| 64 | 16 | f32[4] | specular | RGBA, 0..1 |
| 80 | 16 | f32[4] | emissive | RGBA, 0..1 |
| 96 | 4 | f32 | shininess | 0..128 |
| 100 | 4 | f32 | transparency | 0..1 |
| 104 | 1 | s8 | mode | combine/sphere-map flags; treat 0 or anything else as a plain texture |
| 105 | 128 | char[128] | texture | diffuse map path |
| 233 | 128 | char[128] | alphamap | ignore |

> **TIP:** The texture path is whatever was on the artist's machine, absolute Windows path and all (`C:\My Documents\skin.tga`). Strip everything up to the last `\` or `/` and load the basename from your own textures directory.

### Animation globals (12 bytes)

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 4 | f32 | fAnimationFPS | playback rate, typically 24 |
| 4 | 4 | f32 | fCurrentTime | editor state; ignore |
| 8 | 4 | s32 | iTotalFrames | animation length in frames |

### Joint record (93 bytes + keyframes)

| offset | size | type | name | meaning |
|---|---|---|---|---|
| 0 | 1 | u8 | flags | ignore |
| 1 | 32 | char[32] | name | |
| 33 | 32 | char[32] | parentName | empty string = root joint |
| 65 | 12 | f32[3] | rotation | local Euler XYZ, **radians**, relative to parent |
| 77 | 12 | f32[3] | position | local, relative to parent |
| 89 | 2 | u16 | nKeyFramesRot | |
| 91 | 2 | u16 | nKeyFramesTrans | |

Immediately after: `nKeyFramesRot` rotation keys, then `nKeyFramesTrans` translation keys, 16 bytes each:

| key type | fields | meaning |
|---|---|---|
| rotation | f32 time; f32 rotation[3] | time in seconds; Euler XYZ radians, **relative to the joint's base rotation** |
| translation | f32 time; f32 position[3] | time in seconds; offset relative to the joint's base position |

Resolve parents by `strcmp` on `parentName`. Milkshape writes parents before children in practice, but a name-based second pass costs nothing and never breaks. Keyframe values being relative to the base pose is the fact that everyone misses on Day 24: the animated local transform is base * key, not key alone. Anything in the file after the last joint belongs to later format revisions (comments, vertex weight extensions); stop reading and close the file.

## E.6 Designing your own text formats

Level files, configs, material lists: when you need a format of your own, make it a line-based text file. This is the accumulated wisdom of the era — every id Software game shipped its `.cfg` and shader scripts as plain text, and the reason is practical: text files are diffable, hand-editable in Notepad on a playtester's machine, and debuggable by looking at them.

- **One record per line.** The first token is the keyword; the rest are its arguments. Parse with `fgets` + `sscanf` and you never write a tokenizer.
- **`#` starts a comment.** Blank lines are free.
- **The first line is a magic word and a version number.** Refuse files newer than you understand; accept older ones.
- **Skip unknown keywords with a printed warning**, never an abort. Old engine builds can then load new levels, minus the new features.

- **No quoted strings, no escape sequences.** Paths use forward slashes and contain no spaces. The parser stays ten lines long.

```
# valley.lvl
LEVEL 1
sky      textures/sky_dawn
terrain  maps/valley_hm.png  2.0 32.0 2.0
fog      0.0025  204 178 153
entity   crate  12.0 1.0 -4.5  30.0
music    music/valley.xm
```

```
char line[256], key[32];
int lineno = 0;
while (fgets(line, sizeof line, fp)) {
    ++lineno;
    if (line[0] == '#' || line[0] == '\n' || line[0] == '\r')
        continue;
    if (sscanf(line, "%31s", key) != 1)
        continue;
    if (strcmp(key, "fog") == 0) {
        float dens; int r, g, b;
        if (sscanf(line, "%*s %f %d %d %d",
                   &dens, &r, &g, &b) == 4)
            fog_set(dens, r, g, b);
        else
            printf("%s(%d): bad fog line\n", path, lineno);
    }
    /* ...else if (strcmp(key, "entity") == 0) ... */
}
```

Print the filename and line number in every error. Future you, at 11 pm, editing the level by hand, will be grateful. Binary formats earn their keep only when the data is bulky (meshes, images) — and then you document them with a table like the ones above, because in six months you are the archaeologist.

# Appendix F: The Math Formulary

Every formula in the book, restated in one place with our exact conventions, so you can re-derive nothing at the cabin. The conventions, fixed on Day 1: **column-major** `float m[16]` (column c, row r at `m[c*4+r]`), **right-handed** world, +Y up, camera looking down **-Z**, transforms as `out = M * v`, vector macros with the **result last**, angles in the `mat4_` API in degrees.

## F.1 mat4 memory layout

```
printed (math) form                    memory (column-major)

| m[0]  m[4]  m[8]   m[12] |           float m[16];
| m[1]  m[5]  m[9]   m[13] |           column c, row r -> m[c*4+r]
| m[2]  m[6]  m[10]  m[14] |
| m[3]  m[7]  m[11]  m[15] |           columns are contiguous:
                                       m[0..3]   X axis (right)
translation sits in m[12],m[13],m[14]  m[4..7]   Y axis (up)
— the last COLUMN, contiguous in      m[8..11]  Z axis (back, +Z)
memory. If your translation lands in   m[12..15] position
m[3],m[7],m[11], you built the
transpose.
```

This is exactly the order `glUniformMatrix4fv` expects; always pass `transpose = GL_FALSE` (GLES2 requires it). The upper-left 3x3 columns are the transformed basis vectors — which makes debugging concrete: print the columns of a suspect matrix and you are looking at where the axes went.

## F.2 Projection matrices

### Perspective: the derivation

View space: camera at the origin, looking down -Z, so a visible point p = (x, y, z) has z < 0. Put the projection window on the near plane, z = -n. Similar triangles give the projected point:

```
x' = n * x / (-z)        y' = n * y / (-z)
```

The window's half-height comes from the vertical field of view: t = n * tan(fovy/2); half-width r = t * aspect. Normalized device coordinates divide by those:

```
y_ndc = y' / t = ( y * cot(fovy/2) ) / (-z)
```

Define f = cot(fovy/2) = 1/tan(fovy/2). The GPU divides every clip coordinate by clip.w after the vertex shader — so if we output clip.w = -z and clip.y = f * y, the hardware divide produces exactly y_ndc. Same for x with f/aspect. That is the whole trick of the perspective matrix: it does not project; it *arranges* for the w-divide to project.

Depth must also survive that divide, so it needs the form z_ndc = (A*z + B)/(-z), mapping z = -n to -1 and z = -f to +1. Two conditions, two unknowns:

```
A = (zf + zn) / (zn - zf)        B = (2 * zf * zn) / (zn - zf)
```

Check: z = -n gives (-An + B)/n = -1; z = -f gives (-Af + B)/f = +1. The resulting matrix and its storage:

```
| f/aspect  0        0                  0            |   m[0]  = f/aspect
|    0      f        0                  0            |   m[5]  = f
|    0      0   (zf+zn)/(zn-zf)  (2*zf*zn)/(zn-zf)   |   m[10] = A
|    0      0       -1                 0            |   m[14] = B
                                                        m[11] = -1
                                                        all others 0
```

```
void mat4_perspective(float out[16], float fovy_deg, float aspect,
                      float znear, float zfar)
{
    float f;
    int i;
    f = 1.0f / (float)tan(fovy_deg * DEG2RAD * 0.5f);
    for (i = 0; i < 16; ++i) out[i] = 0.0f;
    out[0]  = f / aspect;
    out[5]  = f;
    out[10] = (zfar + znear) / (znear - zfar);
    out[11] = -1.0f;
    out[14] = (2.0f * zfar * znear) / (znear - zfar);
}
```

Because of the divide, z_ndc is **nonlinear**: nearly all depth precision sits near the near plane. Raising znear from 0.1 to 0.5 buys more z-fighting relief than any far-plane change ever will. ANGLE maps the -1..1 NDC range to D3D9's [0,1] for you; treat depth as standard GL and never think about it again.

### Why the w-divide happens after the vertex shader

Two hard reasons, both invisible until they bite. First, **clipping** must run between the shader and the divide: a triangle crossing the z = 0 camera plane would divide by w = 0. In homogeneous space the frustum is the well-behaved region -w ≤ x,y,z ≤ w, and clipping there needs no divisions at all. Second, the rasterizer keeps 1/w per vertex to interpolate texture coordinates **perspective-correctly** across each triangle; divide too early and that information is gone — which is precisely why PlayStation 1 textures swim.

### Orthographic

No divide needed: w stays 1 and each axis is a linear remap of [l,r], [b,t], [zn,zf] onto [-1,1]:

```
void mat4_ortho(float out[16], float l, float r, float b, float t,
                float zn, float zf)
{
    int i;
    for (i = 0; i < 16; ++i) out[i] = 0.0f;
    out[0]  =  2.0f / (r - l);
    out[5]  =  2.0f / (t - b);
    out[10] = -2.0f / (zf - zn);
    out[12] = -(r + l) / (r - l);
    out[13] = -(t + b) / (t - b);
    out[14] = -(zf + zn) / (zf - zn);
    out[15] = 1.0f;
}
```

For the Day 6 overlay: `mat4_ortho(m, 0, 1024, 600, 0, -1, 1)` puts pixel (0,0) at the top-left, the way UI code thinks.

## F.3 LookAt

The view matrix is the *inverse* of the camera's own transform. Build the camera basis: forward `f = normalize(center - eye)`, right `s = normalize(f × up)`, true up `u = s × f`. The camera-to-world matrix has columns s, u, -f (recall: camera looks down its -Z) and position eye. Its inverse is the transposed rotation times a translation by -eye — the transpose puts s, u, -f into *rows*, and the translation column becomes the dot products below.

```
void mat4_lookat(float out[16], const vec3_t eye,
                 const vec3_t center, const vec3_t up)
{
    vec3_t f, s, u;
    VectorSubtract(center, eye, f);
    VectorNormalize(f);
    CrossProduct(f, up, s);
    VectorNormalize(s);
    CrossProduct(s, f, u);
    out[0] =  s[0]; out[4] =  s[1]; out[8]  =  s[2];
    out[1] =  u[0]; out[5] =  u[1]; out[9]  =  u[2];
    out[2] = -f[0]; out[6] = -f[1]; out[10] = -f[2];
    out[3] = 0.0f;  out[7] = 0.0f;  out[11] = 0.0f;
    out[12] = -DotProduct(s, eye);
    out[13] = -DotProduct(u, eye);
    out[14] =  DotProduct(f, eye);
    out[15] = 1.0f;
}
```

Degenerate case: looking straight along `up` makes the cross product zero. Clamp your camera pitch (F.4) and it cannot happen. For the Day 10 skybox, zero out m[12..14] of the view matrix to strip translation.

## F.4 Rotation matrices and Euler angles

```
Rx(a)                   Ry(a)                   Rz(a)
| 1   0    0   0 |      |  c   0   s   0 |      |  c  -s   0   0 |
| 0   c   -s   0 |      |  0   1   0   0 |      |  s   c   0   0 |
| 0   s    c   0 |      | -s   0   c   0 |      |  0   0   1   0 |
| 0   0    0   1 |      |  0   0   0   1 |      |  0   0   0   1 |

c = cos(a), s = sin(a); positive angle = counterclockwise when the
axis points at you (right-hand rule).
```

Storage (cells that differ from identity):

| rotation | cells |
|---|---|
| X | m[5]=c m[6]=s m[9]=-s m[10]=c |
| Y | m[0]=c m[2]=-s m[8]=s m[10]=c |
| Z | m[0]=c m[1]=s m[4]=-s m[5]=c |

```
void mat4_rotate_y(float out[16], float deg)
{
    float c, s;
    c = (float)cos(deg * DEG2RAD);
    s = (float)sin(deg * DEG2RAD);
    mat4_identity(out);
    out[0] = c;  out[2] = -s;
```

```
    out[8] = s;  out[10] = c;
}
```

### Composition pitfalls

- **Order is everything.** `mat4_multiply(out, a, b)` computes out = a*b, and since vectors multiply on the right, **b happens first**. A model turned then moved is `mat4_multiply(m, T, R)`.
- **The FPS camera convention** (Day 3): view rotation = Rx(-pitch) * Ry(-yaw) — yaw first, in world space, then pitch. Swap them and the horizon tilts when you look around.
- **Gimbal lock:** at pitch = ±90° the yaw and roll axes coincide and one degree of freedom vanishes. For a camera, clamp pitch to ±89° and the problem is fenced off. For animated joints, use quaternions (F.7).
- **Units:** the `mat4_` API takes degrees; `sin`/`cos` and the MS3D file take radians. Passing degrees to `sin()` produces a rotation that is almost, but not quite, entirely wrong — and it compiles clean.

```
DEG2RAD = 0.01745329252   (pi / 180)
RAD2DEG = 57.2957795131   (180 / pi)
```

## F.5 Frustum plane extraction (Gribb-Hartmann)

After the vertex shader a point is visible when -w ≤ x_clip ≤ w (and likewise y, z). Take the left bound: x_clip ≥ -w means (row 3 + row 0) of the clip matrix, dotted with the input point, is ≥ 0. That sum of rows *is* a plane equation in whatever space the input point lived in. So: build `clip = proj * view` (`mat4_multiply(clip, proj, view)`) and extract six world-space planes as row sums and differences. Extract from `proj * view * model` instead and you get model-space planes for free — handy for testing many objects against one local box.

In our storage, row r column c is `m[c*4+r]`: row 0 is m[0], m[4], m[8], m[12] and row 3 is m[3], m[7], m[11], m[15].

```
typedef struct { float a, b, c, d; } plane_t;
/* point p is inside when a*p[0] + b*p[1] + c*p[2] + d >= 0 */

void frustum_extract(plane_t pl[6], const float m[16])
{
    int i;
    pl[0].a = m[3]+m[0];   pl[0].b = m[7]+m[4];    /* left   */
    pl[0].c = m[11]+m[8];  pl[0].d = m[15]+m[12];
    pl[1].a = m[3]-m[0];   pl[1].b = m[7]-m[4];    /* right  */
    pl[1].c = m[11]-m[8];  pl[1].d = m[15]-m[12];
    pl[2].a = m[3]+m[1];   pl[2].b = m[7]+m[5];    /* bottom */
    pl[2].c = m[11]+m[9];  pl[2].d = m[15]+m[13];
    pl[3].a = m[3]-m[1];   pl[3].b = m[7]-m[5];    /* top    */
    pl[3].c = m[11]-m[9];  pl[3].d = m[15]-m[13];
    pl[4].a = m[3]+m[2];   pl[4].b = m[7]+m[6];    /* near   */
    pl[4].c = m[11]+m[10]; pl[4].d = m[15]+m[14];
    pl[5].a = m[3]-m[2];   pl[5].b = m[7]-m[6];    /* far    */
    pl[5].c = m[11]-m[10]; pl[5].d = m[15]-m[14];
    for (i = 0; i < 6; ++i) {
        float len;
        len = (float)sqrt(pl[i].a*pl[i].a + pl[i].b*pl[i].b +
                          pl[i].c*pl[i].c);
        pl[i].a /= len; pl[i].b /= len;
        pl[i].c /= len; pl[i].d /= len;
    }
}
```

All six normals point **inward**. The normalization matters: without it, comparing a plane distance against a sphere radius (F.6) compares apples to scaled apples.

## F.6 The intersection zoo

Compact, correct, C89. Conventions: `plane_t` normalized as in F.5; AABBs as `mins`/`maxs` corners; ray directions unit length where noted; distances returned through optional out-pointers (pass NULL if unwanted).

### Point in AABB

```
int point_in_aabb(const vec3_t p, const vec3_t mins,
                  const vec3_t maxs)
{
    return p[0] >= mins[0] && p[0] <= maxs[0] &&
           p[1] >= mins[1] && p[1] <= maxs[1] &&
           p[2] >= mins[2] && p[2] <= maxs[2];
}
```

### Sphere vs plane

```
float plane_dist(const plane_t *pl, const vec3_t p)
{
    return pl->a*p[0] + pl->b*p[1] + pl->c*p[2] + pl->d;
}

/* +1 fully in front, -1 fully behind, 0 straddling */
int sphere_plane_side(const plane_t *pl, const vec3_t c, float r)
{
    float d;
    d = plane_dist(pl, c);
    if (d >  r) return  1;
    if (d < -r) return -1;
    return 0;
}
```

### Sphere vs frustum (Day 9)

```
/* 1 = inside or touching, 0 = fully outside */
int frustum_test_sphere(const plane_t pl[6], const vec3_t c, float r)
{
    int i;
    for (i = 0; i < 6; ++i) {
        if (plane_dist(&pl[i], c) < -r)
            return 0;
    }
    return 1;
}
```

Conservative: an object outside the frustum but not fully outside any single plane is kept. That costs a few wasted draws near corners and is the correct trade — culling must never drop a visible object.

### AABB vs frustum (Day 12 chunks)

Per plane, test only the "p-vertex": the box corner farthest along the plane normal. If even that corner is behind the plane, the whole box is.

```
int frustum_test_aabb(const plane_t pl[6], const vec3_t mins,
                      const vec3_t maxs)
{
    int i;
    for (i = 0; i < 6; ++i) {
```

```
        vec3_t v;
        v[0] = (pl[i].a >= 0.0f) ? maxs[0] : mins[0];
        v[1] = (pl[i].b >= 0.0f) ? maxs[1] : mins[1];
        v[2] = (pl[i].c >= 0.0f) ? maxs[2] : mins[2];
        if (plane_dist(&pl[i], v) < 0.0f)
            return 0;
    }
    return 1;
}
```

### Ray vs AABB (slab method, Day 28 hitscan)

Clip the ray's t-interval against each axis slab; if the interval survives all three, it hits. `dir` need not be normalized — t is then in units of `dir`'s length.

```
int ray_aabb(const vec3_t org, const vec3_t dir,
             const vec3_t mins, const vec3_t maxs, float *t_out)
{
    float tmin, tmax;
    int i;
    tmin = 0.0f;
    tmax = 1e30f;
    for (i = 0; i < 3; ++i) {
        if ((float)fabs(dir[i]) < 1e-8f) {
            if (org[i] < mins[i] || org[i] > maxs[i])
                return 0;             /* parallel and outside */
        } else {
            float inv, t0, t1, tmp;
            inv = 1.0f / dir[i];
            t0 = (mins[i] - org[i]) * inv;
            t1 = (maxs[i] - org[i]) * inv;
            if (t0 > t1) { tmp = t0; t0 = t1; t1 = tmp; }
            if (t0 > tmin) tmin = t0;
            if (t1 < tmax) tmax = t1;
            if (tmin > tmax) return 0;
        }
    }
    if (t_out) *t_out = tmin;
    return 1;
}
```

### Ray vs sphere

```
/* dir must be unit length */
int ray_sphere(const vec3_t org, const vec3_t dir,
               const vec3_t center, float radius, float *t_out)
{
    vec3_t m;
    float b, c, disc, t;
    VectorSubtract(org, center, m);
    b = DotProduct(m, dir);
    c = DotProduct(m, m) - radius * radius;
    if (c > 0.0f && b > 0.0f) return 0; /* outside, going away */
    disc = b * b - c;
    if (disc < 0.0f) return 0;              /* misses */
    t = -b - (float)sqrt(disc);
    if (t < 0.0f) t = 0.0f;                 /* started inside */
    if (t_out) *t_out = t;
    return 1;
}
```

### Ray vs triangle (Moller-Trumbore)

Solves for the barycentric coordinates (u, v) and ray parameter t in one shot; no precomputed plane needed.

```
int ray_triangle(const vec3_t org, const vec3_t dir,
                 const vec3_t v0, const vec3_t v1, const vec3_t v2,
                 float *t_out)
{
    vec3_t e1, e2, p, q, s;
    float det, inv, u, v, t;
    VectorSubtract(v1, v0, e1);
    VectorSubtract(v2, v0, e2);
    CrossProduct(dir, e2, p);
    det = DotProduct(e1, p);
    if (det > -1e-8f && det < 1e-8f)
        return 0;                    /* parallel to the plane */
    inv = 1.0f / det;
    VectorSubtract(org, v0, s);
    u = DotProduct(s, p) * inv;
    if (u < 0.0f || u > 1.0f) return 0;
    CrossProduct(s, e1, q);
    v = DotProduct(dir, q) * inv;
    if (v < 0.0f || u + v > 1.0f) return 0;
    t = DotProduct(e2, q) * inv;
    if (t < 0.0f) return 0;          /* behind the origin */
    if (t_out) *t_out = t;
    return 1;
}
```

As written it hits both faces. To hit front faces only (counter- clockwise winding), reject `det < 1e-8f` instead of rejecting only near-zero.

### Sphere vs AABB: closest point and pushout (Day 22)

Clamp the sphere center to the box to get the closest point on the box; the overlap along that direction is the separation you need.

```
/* If touching, write the minimum translation that separates the
   sphere from the box into push and return 1. */
int sphere_aabb_push(const vec3_t c, float r,
                     const vec3_t mins, const vec3_t maxs,
                     vec3_t push)
{
    vec3_t p, d;
    float d2, len;
    int i;
    for (i = 0; i < 3; ++i) {        /* closest point on box */
        p[i] = c[i];
        if (p[i] < mins[i]) p[i] = mins[i];
        if (p[i] > maxs[i]) p[i] = maxs[i];
    }
    VectorSubtract(c, p, d);
    d2 = DotProduct(d, d);
    if (d2 > r * r) return 0;
    len = (float)sqrt(d2);
    if (len > 1e-6f) {
        VectorScale(d, (r - len) / len, push);
    } else {                         /* center inside the box */
        float over, best;
        int a, axis, sign;
        axis = 0; sign = 1;
        best = maxs[0] - c[0];
```

```
        for (a = 0; a < 3; ++a) {
            over = maxs[a] - c[a];
            if (over < best) { best = over; axis = a; sign = 1; }
            over = c[a] - mins[a];
            if (over < best) { best = over; axis = a; sign = -1; }
        }
        push[0] = push[1] = push[2] = 0.0f;
        push[axis] = (float)sign * (best + r);
    }
    return 1;
}
```

The center-inside branch matters: without it a fast player who tunnels a corner gets a zero-length push and stays embedded. The swept version that prevents tunneling in the first place is built on Day 22.

## F.7 Quaternions (Day 24)

A unit quaternion (x, y, z, w) encodes a rotation of angle 2*acos(w) about the axis (x, y, z)/sin(angle/2). For skeletal animation they beat matrices where it counts: 4 floats instead of 9, composition without drift, and — the real prize — interpolation that takes the shortest arc.

```
typedef struct { float x, y, z, w; } quat_t;

void quat_identity(quat_t *q)
{
    q->x = q->y = q->z = 0.0f;
    q->w = 1.0f;
}

/* radians; matches R = Rz * Ry * Rx (the Milkshape joint order) */
void quat_from_euler(quat_t *q, float rx, float ry, float rz)
{
    float sx, cx, sy, cy, sz, cz;
    sx = (float)sin(rx * 0.5f); cx = (float)cos(rx * 0.5f);
    sy = (float)sin(ry * 0.5f); cy = (float)cos(ry * 0.5f);
    sz = (float)sin(rz * 0.5f); cz = (float)cos(rz * 0.5f);
    q->w = cx*cy*cz + sx*sy*sz;
    q->x = sx*cy*cz - cx*sy*sz;
    q->y = cx*sy*cz + sx*cy*sz;
    q->z = cx*cy*sz - sx*sy*cz;
}

/* out = a * b : b's rotation first, then a's. out may not alias. */
void quat_multiply(quat_t *out, const quat_t *a, const quat_t *b)
{
    out->w = a->w*b->w - a->x*b->x - a->y*b->y - a->z*b->z;
    out->x = a->w*b->x + a->x*b->w + a->y*b->z - a->z*b->y;
    out->y = a->w*b->y - a->x*b->z + a->y*b->w + a->z*b->x;
    out->z = a->w*b->z + a->x*b->y - a->y*b->x + a->z*b->w;
}

void quat_normalize(quat_t *q)
{
    float len;
    len = (float)sqrt(q->x*q->x + q->y*q->y +
                      q->z*q->z + q->w*q->w);
    if (len > 1e-6f) {
        float inv;
        inv = 1.0f / len;
        q->x *= inv; q->y *= inv;
        q->z *= inv; q->w *= inv;
```

```
    }
}
```

Normalize after every few dozen multiplies; float error slowly inflates the length, and a non-unit quaternion converted to a matrix scales your model. The joint hierarchy of Day 24 multiplies once per joint per frame — normalizing each result is cheap insurance on the Atom.

### Slerp

Two fixes make textbook slerp production-ready. First, q and -q are the same rotation but interpolate along opposite arcs; when the dot product is negative, negate one input to take the short way around — skip this and your character's forearm whirls 360° between two nearby keyframes. Second, as the angle approaches zero, sin(theta) approaches zero and the weights blow up; below a threshold, plain lerp plus normalize is within float precision of the true answer.

```
void quat_slerp(quat_t *out, const quat_t *a, const quat_t *b,
                float t)
{
    quat_t bb;
    float dot, th, sinth, s0, s1;
    bb = *b;
    dot = a->x*bb.x + a->y*bb.y + a->z*bb.z + a->w*bb.w;
    if (dot < 0.0f) {            /* take the shorter arc */
        bb.x = -bb.x; bb.y = -bb.y;
        bb.z = -bb.z; bb.w = -bb.w;
        dot = -dot;
    }
    if (dot > 0.9995f) {         /* tiny angle: lerp is exact
                                    enough and never divides by 0 */
        out->x = a->x + (bb.x - a->x) * t;
        out->y = a->y + (bb.y - a->y) * t;
        out->z = a->z + (bb.z - a->z) * t;
        out->w = a->w + (bb.w - a->w) * t;
        quat_normalize(out);
        return;
    }
    th    = (float)acos(dot);
    sinth = (float)sin(th);
    s0 = (float)sin((1.0f - t) * th) / sinth;
    s1 = (float)sin(t * th) / sinth;
    out->x = a->x*s0 + bb.x*s1;
    out->y = a->y*s0 + bb.y*s1;
    out->z = a->z*s0 + bb.z*s1;
    out->w = a->w*s0 + bb.w*s1;
}
```

### Quaternion to mat4

```
void quat_to_mat4(float out[16], const quat_t *q)
{
    float x, y, z, w, x2, y2, z2;
    float xx, xy, xz, yy, yz, zz, wx, wy, wz;
    x = q->x; y = q->y; z = q->z; w = q->w;
    x2 = x + x; y2 = y + y; z2 = z + z;
    xx = x*x2; xy = x*y2; xz = x*z2;
    yy = y*y2; yz = y*z2; zz = z*z2;
    wx = w*x2; wy = w*y2; wz = w*z2;
    out[0]  = 1.0f - (yy + zz);
    out[1]  = xy + wz;
    out[2]  = xz - wy;
    out[3]  = 0.0f;
    out[4]  = xy - wz;
```

```
    out[5]  = 1.0f - (xx + zz);
    out[6]  = yz + wx;
    out[7]  = 0.0f;
    out[8]  = xz + wy;
    out[9]  = yz - wx;
    out[10] = 1.0f - (xx + yy);
    out[11] = 0.0f;
    out[12] = 0.0f; out[13] = 0.0f;
    out[14] = 0.0f; out[15] = 1.0f;
}
```

Column 0 of the result is where the X axis goes, and so on — consistent with F.1, ready for `mat4_multiply` against the parent joint and a final translate.

## F.8 Handy numbers

### Constants

| name | value |
|---|---|
| pi | 3.14159265 |
| 2*pi | 6.28318531 |
| pi/2 | 1.57079633 |
| DEG2RAD (pi/180) | 0.01745329 |
| RAD2DEG (180/pi) | 57.2957795 |
| sqrt(2) | 1.41421356 |
| 1/sqrt(2) | 0.70710678 |
| sqrt(3) | 1.73205081 |

### Common fields of view

| fovy (deg) | radians | f = 1/tan(fovy/2) |
|---|---|---|
| 45 | 0.7854 | 2.4142 |
| 60 | 1.0472 | 1.7321 |
| 75 | 1.3090 | 1.3032 |
| 90 | 1.5708 | 1.0000 |

On the 1024x600 screen (aspect 1.7067), a fovy of 60–75 degrees feels like the era's 90-degree 4:3 defaults.

### Sines and cosines

| deg | sin | cos |
|---|---|---|
| 0 | 0.0000 | 1.0000 |
| 15 | 0.2588 | 0.9659 |
| 30 | 0.5000 | 0.8660 |
| 45 | 0.7071 | 0.7071 |
| 60 | 0.8660 | 0.5000 |
| 75 | 0.9659 | 0.2588 |

| | | |
|---|---|---|
| 90 | 1.0000 | 0.0000 |

### Powers of two

| **n** | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 12 | 16 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **2^n** | 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 256 | 512 | 1024 | 4096 | 65536 |

### DXT block sizes

| format | bytes per 4x4 block | bytes per pixel | alpha |
|---|---|---|---|
| DXT1 | 8 | 0.5 | none (or 1-bit) |
| DXT3 | 16 | 1.0 | 4-bit explicit |
| DXT5 | 16 | 1.0 | interpolated |

### Texture memory (base level, no mips)

| size | RGBA8888 | RGB565 | DXT1 | DXT5 |
|---|---|---|---|---|
| 128x128 | 64 KB | 32 KB | 8 KB | 16 KB |
| 256x256 | 256 KB | 128 KB | 32 KB | 64 KB |
| 512x512 | 1024 KB | 512 KB | 128 KB | 256 KB |
| 1024x1024 | 4096 KB | 2048 KB | 512 KB | 1024 KB |

A full mip chain multiplies any row by 4/3 (the geometric series 1 + 1/4 + 1/16 + ...). Remember the GMA 3600 has no dedicated VRAM — every texture lives in your one gigabyte of system RAM, alongside the engine, the level, and Windows XP itself. The DXT column is not an optimization; on this machine it is the budget.