Voltrum Memory Architecture Design Document

Core Architectural Questions & Decisions

---

1. Where Does Allocation Take Place?

Handmade Hero Approach

```
Platform Layer (win32_handmade.cpp)
    └── Win32AllocateMemory() calls VirtualAlloc
              ↑
              │ function pointer
              │
Game Layer calls Platform.AllocateMemory()
    └── Arena requests new block when exhausted
```

The platform layer owns the actual OS allocation call. The game layer never directly calls VirtualAlloc - it goes through a function pointer table. This keeps the game layer platform-agnostic.

Proposed Voltrum Approach

Two-layer allocation:

```
┌─────────────────────────────────────────────────────────┐
│ Platform Layer (platform.cpp)                           │
│                                                         │
│   platform_allocate_pages(size) → VirtualAlloc/mmap     │
│   platform_free_pages(ptr, size)                        │
│                                                         │
│   These are the ONLY functions that touch OS memory.    │
└─────────────────────────────────────────────────────────┘
                         ↑
                         │ called by
                         │
┌─────────────────────────────────────────────────────────┐
│ Arena Layer (arena.cpp)                                 │
│                                                         │
│   arena_create() → reserves virtual address space       │
│   arena_push() → commits pages as needed, bumps pointer │
│   arena_pop_to() → resets position (optional decommit)  │
│   arena_destroy() → releases all pages                  │
│                                                         │
│   Arenas call platform_allocate_pages() internally.     │
└─────────────────────────────────────────────────────────┘
                         ↑
                         │ used by
                         │
┌─────────────────────────────────────────────────────────┐
│ Application Layer (application.cpp, subsystems)         │
│                                                         │
│   Only sees arenas. Never calls platform allocation     │
│   directly. All memory comes from arena_push().         │
└─────────────────────────────────────────────────────────┘
```

Key principle: The application layer should never call `platform_allocate_pages()` directly. All allocations flow through arenas. This gives you a single chokepoint for tracking and debugging.

---

2. Where Is Subsystem State Stored?

Handmade Hero Approach

`game_state` is the **single source of truth**. It holds: - Arenas directly (embedded): `TotalArena`, `ModeArena`, `AudioArena` - Subsystem state by pointer: `Assets*`, `WorldMode*` - Some state embedded: `AudioState` (small, always needed)

```
struct game_state {
    // Arenas - embedded, not pointers
    memory_arena TotalArena;
    memory_arena ModeArena;
    memory_arena AudioArena;
    memory_arena *FrameArena;  // pointer because it's bootstrapped separa

    // Subsystems - mixed approach
    audio_state AudioState;    // embedded (small, always active)
    game_assets *Assets;       // pointer (large, bootstraps own arena)
    world_mode *WorldMode;     // pointer (only exists in world mode)
};
```

Proposed Voltrum Approach

`Internal_App_State` as the central hub:

```
Internal_App_State
│
├── Arenas (embedded - they ARE the memory)
│   ├── permanent_arena    [Arena]     // App lifetime
│   ├── scene_arena        [Arena]     // Per-project lifetime
│   └── frame_arena        [Arena]     // Per-frame scratch
│
├── Platform State (embedded - always needed, small)
│   └── plat_state         [Platform_State]
│
├── Subsystem State (pointers - allocated in their respective arenas)
│   ├── renderer           [Renderer_State*]   → lives in permanent_arena
│   ├── resources          [Resource_State*]   → lives in permanent_arena
│   ├── events             [Event_State*]      → lives in permanent_arena
│   ├── input              [Input_State*]      → lives in permanent_arena
│   ├── ui                 [UI_State*]         → lives in permanent_arena
│   └── geometry           [Geometry_State*]   → lives in permanent_arena
│
└── Scene State (pointer - allocated in scene_arena)
    └── scene              [Scene_State*]      → lives in scene_arena
```

Rules: 1. **Arenas are embedded** in `Internal_App_State` - they are the foundation 2. **Subsystem state is stored as pointers** in `Internal_App_State` - single source of truth 3. **Subsystems do NOT keep their own global/static state** - no hidden singletons 4. **Scene-specific state lives in scene_arena** - cleared on project change

---

3. Pointer Management: Internal Copy vs Parameter Passing

Option A: Subsystems Cache Pointer Internally (Current Voltrum)

```cpp
// Subsystem has internal static/global state
static Renderer_State *g_renderer_state;

void renderer_startup(...) {
    g_renderer_state = allocate(...);
}

void renderer_draw() {
    // Uses g_renderer_state implicitly
    g_renderer_state->device->draw(...);
}
```

Problems: - Hidden dependencies - Hard to test - Unclear ownership - Can't have multiple instances

Option B: Pass State to Every Call (Handmade Hero Style)

```cpp
// No global state. State passed explicitly.
Renderer_State* renderer_startup(Arena *arena, ...) {
    Renderer_State *state = arena_push<Renderer_State>(arena);
    // ... init ...
    return state;
}

void renderer_draw(Renderer_State *renderer, Render_Packet *packet) {
    renderer->device->draw(...);
}
```

Benefits: - Explicit dependencies - Easy to test (pass mock state) - Clear ownership - Multiple instances possible

Proposed Voltrum Approach: Parameter Passing

Every subsystem API takes its state as first parameter:

```cpp
// Renderer
Renderer_State* renderer_startup(Arena *permanent, Arena *frame);
void            renderer_shutdown(Renderer_State *state);
void            renderer_draw_frame(Renderer_State *state, Render_Packet *
b8              renderer_begin_frame(Renderer_State *state, f32 delta_time
void            renderer_end_frame(Renderer_State *state);

// Resources
Resource_State* resource_system_startup(Arena *permanent);
void            resource_system_shutdown(Resource_State *state);
Resource*       resource_system_load(Resource_State *state, const char *na

// Events
Event_State*    event_system_startup(Arena *permanent);
void            event_system_shutdown(Event_State *state);
void            event_fire(Event_State *state, Event_Code code, Event_Data
void            event_register(Event_State *state, Event_Code code, Event_

// Input
Input_State*    input_startup(Arena *permanent);
void            input_shutdown(Input_State *state);
b8              input_is_key_down(Input_State *state, Key key);
void            input_update(Input_State *state);
```

In application.cpp:

```
// Startup - create all subsystems, store pointers in app state
internal_state->renderer = renderer_startup(
    &internal_state->permanent_arena,
    &internal_state->frame_arena
);
internal_state->resources = resource_system_startup(&internal_state->perma
internal_state->events = event_system_startup(&internal_state->permanent_a
// ...

// Main loop - pass state explicitly
while (running) {
    input_update(internal_state->input);
    event_process_pending(internal_state->events);

    client->update(client, delta_time);
    client->render(client, delta_time);

    renderer_draw_frame(internal_state->renderer, &packet);
}

// Shutdown - pass state explicitly
renderer_shutdown(internal_state->renderer);
resource_system_shutdown(internal_state->resources);
// ...
```

For client callbacks, the client receives what it needs:

```
struct Client {
    // Client can access subsystems through helper functions or direct poi
    Internal_App_State *internal;  // Or provide accessor functions
    void *state;                   // Client's own state
};

// Or provide typed accessors:
Renderer_State* client_get_renderer(Client *client);
Resource_State* client_get_resources(Client *client);
```

---

4. Bootstrap Responsibility: Who Creates Subsystem State?

Option A: Application Layer Creates, Passes to Subsystem

```
// Application allocates the struct
Renderer_State *renderer = arena_push<Renderer_State>(arena);

// Subsystem initializes it
renderer_init(renderer, config);
```

Option B: Subsystem Creates Own State (Bootstrap)

```
// Subsystem allocates and initializes
Renderer_State *renderer = renderer_startup(arena, config);
```

Option C: Subsystem Bootstraps Into Own Arena (Handmade Hero Assets)

```
// Subsystem contains its own arena, allocates itself inside it
// Used when subsystem wants memory isolation
Renderer_State *renderer = renderer_startup_isolated(parent_arena);
// renderer->arena now owns the renderer's memory block
```

Proposed Voltrum Approach: Subsystem Creates Own State (Option B)

Standard pattern - subsystem startup returns pointer:

```
Renderer_State* renderer_startup(Arena *permanent_arena, Arena *frame_aren
    // Subsystem allocates itself from the provided arena
    Renderer_State *state = arena_push<Renderer_State>(permanent_arena);

    // Store arena reference for future allocations
    state->permanent_arena = permanent_arena;
    state->frame_arena = frame_arena;

    // Initialize Vulkan, etc.
    // ...

    return state;
}
```

For subsystems needing isolation (optional, for large subsystems):

```
Resource_State* resource_system_startup_isolated(Arena *parent_arena) {
    // Create a child arena for this subsystem
    Arena *subsystem_arena = arena_create_child(parent_arena, Megabytes(64

    // Allocate state in the child arena
    Resource_State *state = arena_push<Resource_State>(subsystem_arena);
    state->arena = subsystem_arena;  // Owns its arena

    return state;
}
```

Decision tree: - Small subsystem, few allocations → Use parent arena directly - Large subsystem, many allocations, want isolation → Create child arena - Subsystem with different lifetime → Use appropriate arena (scene_arena vs permanent_arena)

---

5. Replacing Tagged Allocation with Arena-Based Tracking

Current Tagged Allocation (malloc-based)

```
void* memory_allocate(u64 size, Memory_Tag tag) {
    void* ptr = malloc(size + header);
    stats.tagged_allocations[tag] += size;
    return ptr;
}
// Gives you: "RENDERER: 45MB, TEXTURE: 128MB, UI: 2MB"
```

Problem

You want arena-based allocation (no malloc) but still want visibility into where memory is used.

Solution: Tagged Arenas + Arena Statistics

Approach: Each logical memory category gets its own arena (or sub-arena)

```
Internal_App_State
│
├── permanent_arena [Arena, tag: PERMANENT]
│   ├── renderer_region      [tagged: RENDERER]
│   ├── resources_region     [tagged: RESOURCES]
│   ├── events_region        [tagged: EVENTS]
│   ├── input_region         [tagged: INPUT]
│   └── ui_region            [tagged: UI]
│
├── scene_arena [Arena, tag: SCENE]
│   ├── geometry_region      [tagged: GEOMETRY]
│   ├── materials_region     [tagged: MATERIALS]
│   └── project_region       [tagged: PROJECT]
│
└── frame_arena [Arena, tag: FRAME]
    └── (single pool, no sub-regions needed)
```

Implementation Options

Option A: Separate Arenas Per Subsystem

```
struct Internal_App_State {
    // Each subsystem gets its own arena
    Arena renderer_arena;      // tag: RENDERER
    Arena resources_arena;     // tag: RESOURCES
    Arena texture_arena;       // tag: TEXTURE
    Arena geometry_arena;      // tag: GEOMETRY
    Arena ui_arena;            // tag: UI
    Arena events_arena;        // tag: EVENTS

    Arena scene_arena;         // tag: SCENE
    Arena frame_arena;         // tag: FRAME
};
```

Tracking:

```
void memory_report() {
    log("RENDERER:  %llu bytes", internal_state->renderer_arena.pos);
    log("RESOURCES: %llu bytes", internal_state->resources_arena.pos);
    log("TEXTURE:   %llu bytes", internal_state->texture_arena.pos);
    // ...
}
```

Pros: Simple, clear isolation, easy tracking **Cons:** Many arenas, each with overhead (page alignment, metadata)

Option B: Single Arena with Tagged Regions (Recommended)

```
struct Arena_Region {
    u64 start_offset;
    u64 current_offset;
    Memory_Tag tag;
};

struct Arena {
    u8 *base;
    u64 pos;
    u64 committed;
    u64 reserved;

    // Tracking: array of regions with tags
    Arena_Region regions[Memory_Tag::MAX_ENTRIES];
    u32 region_count;
};

// Create a tagged region within an arena
Arena_Region* arena_begin_region(Arena *arena, Memory_Tag tag) {
    Arena_Region *region = &arena->regions[arena->region_count++];
    region->start_offset = arena->pos;
    region->current_offset = arena->pos;
    region->tag = tag;
    return region;
}

// Allocate within a region (updates region tracking)
void* arena_push_tagged(Arena *arena, u64 size, Memory_Tag tag) {
    void *ptr = arena_push(arena, size);

    // Update the region for this tag
    for (u32 i = 0; i < arena->region_count; i++) {
        if (arena->regions[i].tag == tag) {
            arena->regions[i].current_offset = arena->pos;
            break;
        }
    }
    return ptr;
}
```

Tracking:

```
void memory_report(Arena *arena) {
    for (u32 i = 0; i < arena->region_count; i++) {
        Arena_Region *r = &arena->regions[i];
        u64 size = r->current_offset - r->start_offset;
        log("%s: %llu bytes", tag_names[r->tag], size);
    }
}
```

Pros: Single arena, fine-grained tracking, low overhead **Cons:** Regions must be allocated in order (no interleaving)

Option C: Arena with Allocation Log (Most Flexible)

```
struct Allocation_Entry {
    void *ptr;
    u64 size;
    Memory_Tag tag;
    const char *file;
    u32 line;
};

struct Arena_With_Tracking {
    Arena arena;

    // Debug tracking (only in debug builds)
    #if DEBUG
    Allocation_Entry *log;
    u32 log_count;
    u32 log_capacity;
    u64 per_tag_total[Memory_Tag::MAX_ENTRIES];
    #endif
};

void* arena_push_tracked(Arena_With_Tracking *a, u64 size, Memory_Tag tag,
                         const char *file, u32 line) {
    void *ptr = arena_push(&a->arena, size);

    #if DEBUG
    // Log the allocation
    a->log[a->log_count++] = { ptr, size, tag, file, line };
    a->per_tag_total[tag] += size;
    #endif

    return ptr;
}

#define ARENA_PUSH(arena, type, tag) \
    (type*)arena_push_tracked(arena, sizeof(type), tag, __FILE__, __LINE__
```

Tracking:

```
void memory_report(Arena_With_Tracking *a) {
    log("=== Memory Report ===");
    for (u32 i = 0; i < Memory_Tag::MAX_ENTRIES; i++) {
        if (a->per_tag_total[i] > 0) {
            log("%s: %llu bytes", tag_names[i], a->per_tag_total[i]);
        }
    }

    // Can also dump individual allocations
    for (u32 i = 0; i < a->log_count; i++) {
        log("  %p: %llu bytes [%s] at %s:%d",
            a->log[i].ptr, a->log[i].size,
            tag_names[a->log[i].tag],
            a->log[i].file, a->log[i].line);
    }
}
```

Pros: Full flexibility, detailed tracking, file/line info **Cons:** Memory overhead for log, slight performance cost

Recommended Approach: Hybrid

```
┌──────────────────────────────────────────────────────────┐
│ Lifetime-Based Arenas (3 arenas)                         │
│                                                          │
│   permanent_arena  ─── App lifetime subsystem state      │
│   scene_arena      ─── Project/scene data                │
│   frame_arena      ─── Per-frame scratch                 │
└──────────────────────────────────────────────────────────┘
                              +
┌──────────────────────────────────────────────────────────┐
│ Per-Tag Tracking (debug only)                            │
│                                                          │
│   Each arena_push() call includes a Memory_Tag           │
│   Accumulated per-tag totals stored in arena             │
│   Zero runtime cost in release builds                    │
└──────────────────────────────────────────────────────────┘
```

Usage:

```
// Macros for tagged allocation
#if DEBUG
#define arena_push_struct(arena, type, tag) \
    (type*)arena_push_tracked(arena, sizeof(type), tag, __FILE__, __LINE__
#else
#define arena_push_struct(arena, type, tag) \
    (type*)arena_push(arena, sizeof(type))
#endif

// Usage in code
Renderer_State *renderer = arena_push_struct(
    &state->permanent_arena,
    Renderer_State,
    Memory_Tag::RENDERER
);
```

Complete Architecture Diagram

```
┌──────────────────────────────────────────────────────────────────────────
┐
│                          PLATFORM LAYER
│
│
│
│   platform_allocate_pages(size) ──→ VirtualAlloc / mmap
│
│   platform_free_pages(ptr, size) ──→ VirtualFree / munmap
│
│
│
│   (Only place that touches OS memory APIs)
│
└──────────────────────────────────────────────────────────────────────────
┘
                                   ↑
                                   │ called by arena layer
                                   │
┌──────────────────────────────────────────────────────────────────────────
┐
│                           ARENA LAYER
│
│
│
│   Arena {
│
│       u8 *base;               // Virtual address base
│
│       u64 pos;                // Current allocation position
│
│       u64 committed;          // Bytes committed (backed by RAM)
│
```

```
│       u64 reserved;          // Bytes reserved (address space only)
│
│       Memory_Tag tag;        // Arena's primary tag
│
│       u64 per_tag[MAX];      // Per-tag byte counts (debug)
│
│   }
│
│
│
│   arena_push(arena, size, tag) ──→ bump pointer, track tag
│
│   arena_pop_to(arena, pos)     ──→ reset to saved position
│
│   temp_begin(arena)            ──→ save current position
│
│   temp_end(temp)               ──→ restore saved position
│
│
│
└──────────────────────────────────────────────────────────────────────────
┘
                                    ↑
                                    │ used by application layer
                                    │
┌──────────────────────────────────────────────────────────────────────────
┐
│                         APPLICATION LAYER
│
│
│
│   Internal_App_State {
│
│       // === ARENAS (embedded, they ARE the memory) ===
│
│       Arena permanent_arena;     // App lifetime
│
│       Arena scene_arena;         // Project lifetime
│
│       Arena frame_arena;         // Frame lifetime
│
│       Temp_Arena frame_temp;     // Reset marker for frame
│
│
│
│       // === PLATFORM (embedded, small, always needed) ===
│
│       Platform_State plat_state;
│
│       Absolute_Clock clock;
│
│
│
│       // === SUBSYSTEMS (pointers into permanent_arena) ===
│
│       Renderer_State  *renderer;     [tag: RENDERER]
│
│       Resource_State  *resources;    [tag: RESOURCES]
│
│       Event_State     *events;       [tag: EVENTS]
│
│       Input_State     *input;        [tag: INPUT]
│
│       UI_State        *ui;           [tag: UI]
│
│       Texture_State   *textures;     [tag: TEXTURE]
│
│       Material_State  *materials;    [tag: MATERIAL]
│
│       Geometry_State  *geometry;     [tag: GEOMETRY]
│
│
│
│       // === SCENE (pointer into scene_arena) ===
```

```
│
│       Scene_State     *scene;        [tag: SCENE]
│
│
│
│       // === CLIENT ===
│
│       Client *client;
│
│   }
│
│
│
└─────────────────────────────────────────────────────────────────────────
┘
                                    │
            ┌───────────────────────┼───────────────────────┐
            │                       │                       │
            ▼                       ▼                       ▼
┌──────────────────────┐ ┌──────────────────────┐ ┌──────────────────────┐
│  RENDERER SUBSYSTEM  │ │  RESOURCE SUBSYSTEM  │ │   EVENT SUBSYSTEM    │
│                      │ │                      │ │                      │
│  Renderer_State {    │ │  Resource_State {    │ │  Event_State {       │
│    Arena *permanent;│ │    Arena *permanent;│ │    Arena *permanent;│
│    Arena *frame;     │ │    // ...            │ │    Event *free_list;│
│    VkDevice device;  │ │    Loader *loaders;  │ │    Event *pending;   │
│    // ...            │ │    Resource *cache;  │ │    Callback *cbs;    │
│  }                   │ │  }                   │ │  }                   │
│                      │ │                      │ │                      │
│  // API: state       │ │  // API: state       │ │  // API: state       │
│  // passed as first  │ │  // passed as first  │ │  // passed as first  │
│  // parameter        │ │  // parameter        │ │  // parameter        │
└──────────────────────┘ └──────────────────────┘ └──────────────────────┘
```

Startup Sequence

```
b8 application_init(Client *client) {
    // 1. Create arenas (these call platform_allocate_pages internally)
    Internal_App_State *state = /* bootstrap or static allocation */;

    arena_init(&state->permanent_arena, Gigabytes(1), "permanent");
    arena_init(&state->scene_arena, Megabytes(256), "scene");
    arena_init(&state->frame_arena, Megabytes(64), "frame");

    // 2. Platform init (uses permanent_arena for internal state)
    platform_startup(&state->plat_state, &state->permanent_arena);

    // 3. Subsystem startup - each returns pointer, stored in app state
    //    Subsystems receive arena(s) they need
    state->events = event_system_startup(&state->permanent_arena);
    state->input = input_startup(&state->permanent_arena);
    state->resources = resource_system_startup(&state->permanent_arena);

    state->renderer = renderer_startup(
        &state->permanent_arena,  // for persistent state
        &state->frame_arena       // for per-frame allocations
    );

    state->textures = texture_system_startup(&state->permanent_arena, stat
    state->materials = material_system_startup(&state->permanent_arena);
    state->geometry = geometry_system_startup(&state->permanent_arena);
    state->ui = ui_startup(&state->permanent_arena, &state->frame_arena);

    // 4. Client init - receives what it needs
    state->client = client;
    client->internal = state;
    client->initialize(client);

    return true;
}
```

Main Loop

```c
void application_run() {
    Internal_App_State *state = /* ... */;

    while (state->is_running) {
        // === FRAME START ===
        // Save frame arena position for reset at end
        state->frame_temp = temp_begin(&state->frame_arena);

        // Platform message pump
        platform_pump_messages(&state->plat_state);

        // Input update
        input_update(state->input);

        // Event processing
        event_process_pending(state->events);

        // Client update & render
        f32 dt = clock_get_delta(&state->clock);
        state->client->update(state->client, dt);
        state->client->render(state->client, dt);

        // UI
        ui_begin_frame(state->ui);
        ui_render_layers(state->ui, &state->client->layers);
        Render_Packet packet = ui_end_frame(state->ui);

        // Submit to renderer
        renderer_draw_frame(state->renderer, &packet);

        // === FRAME END ===
        // Reset frame arena - all frame allocations freed
        temp_end(state->frame_temp);
    }
}
```

Memory Report Output (Debug)

```
=== VOLTRUM MEMORY REPORT ===

PERMANENT ARENA: 45.2 MB committed / 1.0 GB reserved
  RENDERER:   12.4 MB  (Vulkan state, pipelines, descriptors)
  RESOURCES:   2.1 MB  (loader registry, resource cache metadata)
  TEXTURE:    18.7 MB  (texture metadata, staging buffers)
  MATERIAL:    1.2 MB  (material definitions, shader refs)
  GEOMETRY:    8.3 MB  (mesh metadata, vertex layouts)
  EVENTS:      0.1 MB  (callback registry, free list)
  INPUT:       0.0 MB  (key states, mouse state)
  UI:          2.4 MB  (ImGui context, font atlas)

SCENE ARENA: 128.5 MB committed / 256.0 MB reserved
  SCENE:     128.5 MB  (project data, component storage, undo history)

FRAME ARENA: 2.1 MB peak / 64.0 MB reserved
  (Reset each frame - current: 0 bytes)

TOTAL: 175.8 MB committed / 1.3 GB reserved
```

Summary of Decisions

Question	Decision
Where does allocation happen?	Platform layer (VirtualAlloc), wrapped by Arena layer
Where is subsystem state stored?	Pointers in `Internal_App_State`, memory in arenas
Pointer management?	Pass state as first parameter to all subsystem APIs
Bootstrap responsibility?	Subsystems create own state, receive arena from app layer
Tracking without malloc?	Tagged arena allocations, per-tag counters in debug builds
How many arenas?	3 lifetime-based: permanent, scene, frame
Subsystem isolation?	Optional - subsystems can request child arena if needed