System Design of Animation + Visualization (Gerrard Hall) expanded technical version

Goal: research-quality, interactive, cinematic visualization of the Gerrard Hall cluster-merge process, faithful to GTSFM outputs and aligned with Frank's four pillars.

Success Criteria (unchanged)

1. **Slab layout:** clusters arranged on a thin plane parallel to camera (2.5D "sheet of glass").
2. **Per-cluster inspection:** click any cluster → rotate *only that cluster*.
3. **Animated merges:** children move smoothly toward parent, crossfade out, parent fades in (no popping).
4. **Cinematic ending:** after final merge, preset camera flythrough around merged hall (nerfstudio / Rome style).

1. Inputs from GTSFM (explicitly what we consume)

1.1 Reconstructions (point clouds)

For each cluster folder:

- `points3D.txt` (COLMAP format):
  `POINT3D_ID X Y Z R G B ...`
- Optional (later polish):
  - `cameras.txt`
  - `images.txt` (camera poses / frustums)

We already parse `points3D.txt`. That stays the truth source for geometry + RGB.

1.2 Merge tree hierarchy + merge schedule

- Logically: a rooted tree where each node is either:
  - leaf cluster reconstruction (ba_output), or
  - merged reconstruction node.
- We need:
  - **parent → children** links
  - **event ordering** (a topological merge sequence)

Right now you hard-code this as `structure` → `flatPaths` → `mergeEvents`. That is totally fine for the demo; later we can auto-derive from GTSFM logs.

1.3 Derived geometry stats we compute at load time

Per cluster i:

- centroid c_i via bounding sphere center
- radius r_i via bounding sphere radius
- point count n_i

These drive slab layout spacing, animation timing, and camera radius.

2. System architecture = 4 cooperating engines (with concrete algorithms)

2.1 Layout Engine (Slab / Thin-box requirement)

Responsibility

Compute a stable **global arrangement** of clusters in a shallow slab so the merge tree reads visually.

Inputs

- merge tree nodes + edges
- centroids c_i
- radii r_i
- leaf/merged types

Outputs

- slab position $p_i = (x_i, y_i, z_0)$ for every cluster (z fixed)
- optional "layout scale" constant S to fit within your desired on-screen footprint without altering intrinsic point cloud scale.

Algorithm (practical first pass)

Step A: compute intrinsic cluster centers

```
geom.computeBoundingSphere();
c_i = geom.boundingSphere.center;
r_i = geom.boundingSphere.radius;
```

Step B: generate a 2D tree layout
We need a deterministic (non-jittery) tree layout. Use a standard tidy tree algorithm:

- DFS to assign subtree sizes.
- Leaves spaced evenly along X.
- Parents placed at mean X of children.
- Y corresponds to depth (root highest).

Pseudo:

```
function layoutTree(node, depth=0){
  if (node.children.length === 0){
    node.x = nextLeafX();
  } else {
    node.children.forEach(ch => layoutTree(ch, depth+1));
    node.x = avg(node.children.map(ch=>ch.x));
  }
  node.y = -depth * levelSpacing;
}
```

`levelSpacing` should be proportional to typical cluster radius so clusters don't overlap:

```
levelSpacing = k * median(r_i)   // k ~ 3-6
leafSpacing  = k2 * median(r_i)  // k2 similar
```

This matches the "presentation-layer slab" style used in city-scale SfM storytelling where clusters are arranged coherently for readability rather than left in raw 3D scatter. [Cornell Computer Science]

Step C: map tree layout → slab positions
Set:

```
p_i.x = S * node.x
p_i.y = S * node.y
p_i.z = z0  // constant shallow depth
```

Step D: freeze slab positions
Important: **layout is computed once**. During merges, clusters move **toward the parent's slab position**, not recomputed every frame. That's how Progressive/Hierarchical SfM pipelines convey structure without chaos.

Notes tied to Frank's "thin box"

- The slab is not literal slicing of geometry.
- It's a *presentation constraint* so hierarchy reads like a storyboard, similar to Rome-lineage demos. [Cornell Computer Science]

2.2 Interaction Engine (Grab/rotate individual clusters)

Responsibility

Let user select a cluster and rotate it **locally** without disturbing slab layout or other clusters.

Inputs

- mouse events
- list of cluster groups (one per reconstruction)

Outputs

- `selectedCluster`
- local rotation deltas applied only to that cluster's group

Data structure (must-have)

When loading a reconstruction:

```
const clusterGroup = new THREE.Group();
clusterGroup.add(pointCloud);
clusterGroup.position.copy(p_i);  // slab placement
worldGroup.add(clusterGroup);

reconstructions.set(path,{
  group: clusterGroup,
```

```
  pointCloud,
  ...
});
```

A. Raycasting selection

Use Three.js Raycaster:

```
const raycaster = new THREE.Raycaster();
const mouse = new THREE.Vector2();

function onClick(e){
  mouse.x = (e.clientX / canvasW) * 2 - 1;
  mouse.y = -(e.clientY / canvasH) * 2 + 1;
  raycaster.setFromCamera(mouse, camera);

  const hits = raycaster.intersectObjects(
     Array.from(reconstructions.values()).map(r=>r.group),
     true
  );
  if(hits.length){
     selectedCluster = hits[0].object.parentGroup;
  }
}
```

B. Selection mode

Once selected:

- store `selectedCluster`
- on drag, apply rotation only to that group
- do **not** touch `worldGroup`, camera, or other clusters.

C. Rotation handler (simple + controllable)

```
let dragging=false, lastX=0, lastY=0;

function onMouseDown(e){
```

```
  if(selectedCluster) { dragging=true; lastX=e.clientX;
lastY=e.clientY; }
}

function onMouseMove(e){
  if(!dragging) return;
  const dx = e.clientX - lastX;
  const dy = e.clientY - lastY;
  selectedCluster.rotation.y += dx * 0.005;
  selectedCluster.rotation.x += dy * 0.005;
  lastX=e.clientX; lastY=e.clientY;
}

function onMouseUp(){ dragging=false; }
```

D. Visual highlight

When selected:

- temporarily boost point material emissive/glow, or
- swap to a "selected" PointsMaterial with slightly higher size/opacity.

Paper / system lineage

This "inspectable sub-reconstruction" interaction is standard in Photo Tourism → Photosynth-style viewers: click a subscene, inspect locally, while global scene stays fixed. [Wikipedia+1]

2.3 Merge Animation Engine (smooth converging merges, force-feel)

Responsibility

Convert discrete merge events into continuous motion + crossfades.

Inputs

- mergeEvents: `{parent, children[], timeIndex}`
- slab positions `p_i`
- cluster radii/centroids

Outputs per frame

- child group positions
- child opacity
- parent opacity
- eventual visibility commit

Timeline representation

Precompute a scheduled animation queue:

```
mergeAnimations = mergeEvents.map((ev, k)=>({
  parent: ev.path,
  children: ev.hide,
  startTime: k * T_event,
  duration: T_merge,
}));
```

Where:

- `T_event` = spacing between events (e.g., 1.0s)
- `T_merge` = actual animation time per merge (e.g., 0.8s)

A. Motion interpolation

For each child `c` merging into parent `p`:

```
start = child.group.position.clone();
target = parent.group.position.clone();

u = clamp((t - startTime)/duration, 0, 1);
e = easeInOutCubic(u);

child.group.position.lerpVectors(start, target, e);
```

B. “Force-directed feel” without physics

Instead of a physics sim, use a **curved Bezier path**:

```
mid = start.clone().lerp(target, 0.5);
mid.y += 0.3 * medianRadius;  // small lift arc

pos = quadraticBezier(start, mid, target, e);
child.group.position.copy(pos);
```

This gives a “pulled together” visual vibe like Rome-lineage storytelling, but is stable and deterministic. [Cornell Computer Science]

C. Crossfade children → parent

We need per-cluster opacity. Because Three.js PointsMaterial is per-object, easiest is:

- give each cluster its own PointsMaterial
- animate `material.opacity`

```
child.pointCloud.material.transparent = true;
child.pointCloud.material.opacity = 1 - e;

parent.pointCloud.material.transparent = true;
parent.pointCloud.material.opacity = e;
```

At end (`u==1`):

```
children.forEach(ch=>{
  ch.group.visible = false;
  ch.visible=false;
});
parent.group.visible=true;
parent.visible=true;
parent.pointCloud.material.opacity=1;
```

D. Avoid layout drift

Children always move to `p_parent` (slab target), not to raw COLMAP center.
This preserves Frank’s thin-slab mental model.

Reference lineage

- Hierarchical / progressive SfM papers emphasize *progressive merging* across a cluster tree (your animation is literally the visual analog).
- Rome-lineage demo videos use smooth convergence rather than pops. [Cornell Computer Science]

2.4 Camera Engine (stable overview + cinematic flythrough)

Responsibility

Camera has *modes*:

1. overview during merge timeline (stable slab framing),
2. optional assist on cluster inspection,
3. cinematic flythrough after final merge.

Inputs

- merged reconstruction bounding sphere (`center`, `radius`)
- slab bounding rect (for overview)
- flythrough duration

Outputs

- camera pose `(position, lookAt)` each frame.

A. Camera state machine

```
cameraMode = "MERGE_OVERVIEW" | "INSPECT" | "FLYTHROUGH";
```

B. Overview mode

- OrbitControls enabled.
- target fixed to slab center (0,0,0 after centering).
- autoRotate optional.

This is basically your current setup.

C. Flythrough mode (nerfstudio / Rome vibe)

Nerfstudio's viewer popularized smooth preset orbits that are just spline-driven camera poses around a center. [CVF Open Access]

Compute path parameters:

```
flyCenter = mergedSphere.center;
flyRadius = mergedSphere.radius * k; // k ~ 2–3 for wide shot
flyDuration = 6–10 seconds;
```

Define trajectory
Simple circular orbit with easing:

```
function flyPose(u){
  const theta = u * 2*Math.PI * 0.6; // partial orbit
  const phi = 0.25*Math.PI;         // slight elevation

  const x = flyCenter.x + flyRadius * Math.cos(theta)*Math.cos(phi);
  const y = flyCenter.y + flyRadius * Math.sin(phi);
  const z = flyCenter.z + flyRadius * Math.sin(theta)*Math.cos(phi);

  return {pos:new THREE.Vector3(x,y,z), target: flyCenter};
}
```

Drive it in animate():

```
if(cameraMode==="FLYTHROUGH"){
  u = clamp((t-flyStart)/flyDuration,0,1);
  e = easeInOutCubic(u);

  pose = flyPose(e);
  camera.position.copy(pose.pos);
  camera.lookAt(pose.target);

  if(u>=1) cameraMode="MERGE_OVERVIEW";
}
```

D. Trigger

After final merge event completes:

```
cameraMode="FLYTHROUGH";
flyStart = performance.now();
controls.autoRotate=false;
controls.enabled=false;
```

Then re-enable controls at the end.

Reference lineage

- Nerfstudio preset camera orbits: clean “hero shot” ending. CVF Open Access
- Rome-lineage large-scale SfM demos end with similar cinematic orbits. Cornell Computer Science

3. Mapping features → pipeline touchpoints (explicit)

Feature	Pipeline data needed	Source
Slab layout	centroids, radii, merge tree	`points3D.txt` + your merge structure (later GTSFM logs)
Per-cluster rotation	cluster separability	each reconstruction → its own `THREE.Group()`
Animated merges	merge ordering + child/parent links	`mergeEvents` ordering (later GTSFM schedule)
Flythrough	final merged center/radius + poses optional	merged `points3D.txt`, optional `images.txt`

4. Implementation order (kept safe & additive)

1. **Freeze upright + slab layout**
   - keep your worldGroup rotation
   - compute slab positions and assign to each clusterGroup
2. **Per-cluster selection/rotation**
   - raycast → select
   - drag rotation on selected only
   - highlight selection
3. **Replace pop merges with animations**
   - trajectories (lerp → bezier)
   - crossfade

    - commit visibility
4. **Cinematic flythrough**
    - camera state machine
    - spline/orbit around merged reconstruction

Each step should not alter point scaling, density, or camera distance unless *you explicitly change a constant*.

5. Extra papers worth mining (expanded list)

Beyond the two Frank named, these map tightly to your features:

1. **Progressive / hierarchical SfM papers (cluster tree merges)**
   You're not stealing algorithms; you're stealing *merge storytelling metaphors and layout grammar.*
2. **Photo Tourism / Photosynth lineage (interactive inspection)**
   The UI/interaction standard for selecting sub-reconstructions and rotating them in isolation is philosophically the same as what Frank wants. Wikipedia+1
3. **Nerfstudio viewer / NeRF demo papers (camera paths)**
   Their biggest gift is a clean, minimal flythrough implementation: preset, smooth, spline-like orbits that never touch geometry. CVF Open Access
4. **City-scale SfM visualization overviews**
   These explain why you **must** use a shallow presentation layer (your slab constraint) to keep multi-cluster structure readable.