Comprehensive Project Overview

Real-Time Procedural Resurfacing Using GPU Mesh Shaders

Paper: Raad et al., "Real-time procedural resurfacing using GPU mesh shader," CGF 44(2), Eurographics 2025

`Reference implementation: github.com/andarael/resurfacing`

1. Project Goal

Reimplement from scratch the GPU mesh shader-based procedural resurfacing framework. The system takes a base mesh M and procedurally generates new geometric surfaces on-the-fly at each face/vertex, using the Vulkan mesh shader pipeline (task shader ⟶ mesh shader ⟶ fragment shader). No traditional vertex input is used — all geometry is read from storage buffers and generated on-chip.

2. Technology Stack

Component	Choice
Language	C++17 (host), GLSL with GL_EXT_mesh_shader (GPU)
Graphics API	Vulkan 1.3 with VK_EXT_mesh_shader extension
Shader Compilation	glslangValidator ⟶ SPIR-V (with GL_GOOGLE_include_directive)
Windowing	GLFW
Math	GLM
UI	ImGui (Vulkan backend)
Mesh Loading	OBJ loader (n-gon support), optionally GLTF for skeletal animation
Build	CMake
Target GPU	NVIDIA RTX 3080 (Ampere), driver 535+
Profiling	NVIDIA Nsight Graphics

The reference implementation uses the Vulkan C++ bindings (vulkan.hpp) rather than the C API.

3. High-Level Architecture

3.1 Pipeline Overview (Paper's Framework)

The resurfacing framework has 5 logical stages, mapped to GPU shader stages:

```
Base Mesh M (SSBO)
    |
    v
TASK SHADER (one dispatch per face+vertex of M)
```

```
  F (Mapping):       base mesh -> base elements bi
  P (Payload):       culling, LOD, attributes -> ej
  K (Amplification): compute mesh shader count kj
  Output: taskPayloadSharedEXT + EmitMeshTasksEXT
    | (pipelined memory)
    v
MESH SHADER (kj invocations per element)
  Param: evaluate parametric surface at (u,v)
  S:     sample UV grid -> triangulated surface
  Output: SetMeshOutputsEXT(verts, prims)
    |
    v
FRAGMENT SHADER (standard rasterization)
  Shading, per-primitive coloring, debug viz
```

3.2 Draw Call

Instead of vkCmdDraw, the application calls `vkCmdDrawMeshTasksEXT(cmd, groupCountX, 1, 1)` where `groupCountX = nbFaces + nbVertices` (one task shader workgroup per base element).

3.3 Descriptor Set Layout

Three descriptor sets, derived from the reference implementation:

Set 0 (SceneSet) — per-frame global data:

- Binding 0: ViewUBO (mat4 view, projection, vec4 cameraPosition, float near, far)
- Binding 1: GlobalShadingUBO (light position, ambient, shading params)

Set 1 (HESet) — half-edge mesh data (Structure of Arrays):

- Binding 0: vec4 buffers array[5] (vertex positions, colors, normals, face normals, face centers)
- Binding 1: vec2 buffers array[1] (vertex texcoords)
- Binding 2: int buffers array[10] (vertex edges, face edges, face vert counts, face offsets, HE vertex/face/next/prev/twin, vertex-face indices)
- Binding 3: float buffers array[1] (face areas)

Set 2 (PerObjectSet) — per-object data:

- Binding 0: Config UBO (ResurfacingUBO or PebbleUBO)
- Binding 1: ShadingUBO (per-object shading)
- Binding 2: LUT vertex buffer (control cage points for B-spline)
- Bindings 3–5: Skeletal animation buffers (joint indices, weights, bone matrices)
- Binding 6: Samplers[2] (linear, nearest)
- Binding 7: Textures[2] (AO texture, element type texture/control map)

Push constants: model matrix (mat4), MVP (mat4).

4. Mesh Data Structure

4.1 Input Format (N-gon OBJ)

The system loads OBJ files supporting n-gon faces (not just triangles). Each face stores:

```
struct NGonFace {
    vec4 normal;     // face normal
    vec4 center;     // face centroid
    uint32 offset;   // offset into index array
    uint32 count;    // vertex count (3, 4, 5, ...)
    float faceArea;  // area of the face
};
```

4.2 Half-Edge Conversion

The loaded ngon mesh is converted to a half-edge data structure on the CPU using Structure of Arrays (SoA) layout:

```
struct HalfEdgeMesh {
    uint32 nbVertices, nbFaces;
    // Vertex SoA:
    vector<vec4> positions, colors, normals;
    vector<vec2> texCoords;
    vector<int> edges;           // one outgoing half-edge per vertex
    // Face SoA:
    vector<int> faceEdges, vertCounts, offsets;
    vector<vec4> faceNormals, faceCenters;
    vector<float> faceAreas;
    // Half-edge SoA:
    vector<int> heVertex, heFace, heNext, hePrev, heTwin;
    vector<int> vertexFaceIndices;  // flattened per-face vertex lists
};
```

4.3 GPU Upload and Shader Access

Each array is uploaded as a separate SSBO. The shader accesses them via typed buffer arrays:

```
layout(std430, set=1, binding=0) readonly buffer { vec4 data[]; } heVec4Buffer[5];
layout(std430, set=1, binding=2) readonly buffer { int data[]; }  heIntBuffer[10];

vec3 getVertexPosition(uint vertId) { return heVec4Buffer[0].data[vertId].xyz; }
vec3 getFaceNormal(uint faceId)     { return heVec4Buffer[3].data[faceId].xyz; }
uint getHalfEdgeNext(uint edgeId)   { return heIntBuffer[6].data[edgeId]; }
```

5. Task Shader (F + P + K)

5.1 Mapping Function F

One task shader workgroup is dispatched per base element. The workgroup ID determines whether it’s a face or vertex element:

```
uint faceId = gl_WorkGroupID.x;
bool isVertex = (faceId >= nbFaces);
if (isVertex) {
    vertId = faceId - nbFaces;
    faceId = getFaceId(vertId);  // get face via half-edge
}
```

5.2 Payload Function P

Culling: Back-face culling via dot(viewDir, instanceNormal) > threshold. Frustum culling by projecting center to clip space and checking bounds.

Skinning (optional): If skeletal animation data is bound, apply bone transforms to position/normal before culling.

Element type selection: If a control map texture is bound, sample it to determine element type per face. Otherwise use global elementType from UBO.

5.3 LOD Computation

```
uvec2 getLodMN(LodInfos info, uint elementType) {
    if (!doLod) return MN;
    parametricBoundingBox(info, elementType);
    float screenSpaceSize = boundingBoxScreenSpaceSize(info);
    uvec2 targetMN = uvec2(MN * sqrt(screenSpaceSize * lodFactor));
    return clamp(targetMN, minResolution, MN);
}
```

5.4 Amplification Function K

Due to hardware limits (MAX_VERTICES=81, MAX_PRIMITIVES=128 for 8×8 grids), the UV domain is split across multiple mesh shader invocations:

```
uvec2 getDeltaUV(uvec2 MN) {
    // Find largest deltaU x deltaV fitting hardware limits
    // (deltaU+1)*(deltaV+1) <= MAX_VERTICES
    // deltaU*deltaV*2 <= MAX_PRIMITIVES
    for (deltaU = max..1) for (deltaV = max..1)
        if (fits) return uvec2(deltaU, deltaV);
}
uvec2 numMeshTasks = (MN + deltaUV - 1) / deltaUV;
```

5.5 Task Payload Structure

```
struct TaskPayload {
    vec3 position;    // world-space center
    vec3 normal;      // orientation normal
    float area;       // face area (for scaling)
    uint taskId;      // original workgroup ID
    bool isVertex;    // face vs vertex element
    uvec2 MN;         // target resolution
    uvec2 deltaUV;    // per-invocation UV range
    uint elementType; // which parametric surface
};
```

```
taskPayloadSharedEXT TaskPayload taskPayload;
```

5.6 Emit

```
EmitMeshTasksEXT(numMeshTasks.x * doRender, numMeshTasks.y * doRender, 1);
```

Setting count to 0 skips the element (culled).

6. Mesh Shader (Param + S)

6.1 Configuration

```
layout(local_size_x = 32) in;  // MESH_GROUP_SIZE = 32
layout(max_vertices = 81, max_primitives = 128, triangles) out;
```

6.2 UV Domain Subsection

```
uvec2 startUV = uvec2(gl_WorkGroupID.xy * deltaUV.xy);
uvec2 localDeltaUV = min(deltaUV, MN - startUV);
uint numVertices = (localDeltaUV.x + 1) * (localDeltaUV.y + 1);
uint numPrimitives = localDeltaUV.x * localDeltaUV.y * 2;
SetMeshOutputsEXT(numVertices, numPrimitives);
```

6.3 Surface Sampling (S)

```
for (uint u = gl_LocalInvocationID.x; u <= localDeltaUV.x; u += MESH_GROUP_SIZE) {
    for (uint v = 0; v <= localDeltaUV.y; ++v) {
        vec2 uvCoords = vec2(startUV + uvec2(u, v)) / vec2(MN);
        vec3 pos, normal;
        parametricPosition(uvCoords, pos, normal, elementType);
        offsetVertex(pos, normal);  // scale, rotate, translate
        emitVertex(pos, normal, uvCoords, vertexIndex);
    }
}
```

6.4 Triangle Emission

```
gl_PrimitiveTriangleIndicesEXT[2*q+0] = uvec3(v00, v10, v11);
gl_PrimitiveTriangleIndicesEXT[2*q+1] = uvec3(v00, v11, v01);
```

6.5 Vertex Transform (offsetVertex)

1. **Scale** by sqrt(faceArea) * userScaling
2. **Rotate** to align with face/vertex normal using align_rotation_to_vector()
3. **Translate** to element position (face center or vertex position)

7. Parametric Surfaces

7.1 Analytical Surfaces

Implemented via switch on elementType (0–8):

- Type 0: Torus — parametricTorus(uv, pos, nrm, majorR, minorR)

- Type 1: Sphere — parametricSphere(uv, pos, nrm, radius)
- Type 2: Mobius Strip
- Type 3: Klein Bottle
- Type 4: Hyperbolic Paraboloid
- Type 5: Helicoid
- Type 6: Cone
- Type 7: Cylinder
- Type 8: Egg

Each function also provides a bounding box function for LOD screen-space size estimation.

7.2 B-Spline Control Cages (types 9–10)

Control cage vertices loaded from separate OBJ (quad grid, sorted by UV), uploaded as SSBO. The mesh shader evaluates bicubic B-spline or Bezier surfaces with configurable degree and cyclic boundary conditions.

```
struct LutData {
    vector<vec4> positions;  // control points flattened row-major
    uint Nx, Ny;             // grid dimensions
    vec3 min, max;           // bounding box of control cage
};
```

7.3 Pebbles (Procedural Cages)

Pebbles use a completely separate pipeline with different payload and generation logic:

Task shader: Maps one element per face only. Fetches face vertex positions into shared memory. Emits vertCount × 2 × 3 patches per face (edge regions × extrusion layers), plus fan/ring patches.

Mesh shader: Constructs control cage on-the-fly: extrude vertices along face normal, project toward center for roundness, subdivide once to get 16 B-spline control points in shared memory. Evaluates bicubic B-spline per patch. Applies procedural noise (Gabor/Perlin).

Pebble UBO parameters: subdivisionLevel (0–8), extrusionAmount, extrusionVariation, roundness (0–2), doNoise, noiseAmplitude, noiseFrequency.

8. Control Maps (Hybrid Surfaces)

Per-face element type selection via a color-coded texture sampled in the task shader using base mesh UV coordinates. Hard-coded color mapping (blue → spike, violet → ball, green → empty). Elements with type=-1 are skipped (base mesh shows through).

9. Shading

9.1 Fragment Shader

Standard Blinn-Phong shading with per-primitive ID coloring, AO texture support, configurable ambient/diffuse/specular/shininess, and HSV-based random coloring from element ID.

9.2 Mesh-to-Fragment Data

- Per-vertex: worldPosU (vec4: xyz=world pos, w=u), normalV (vec4: xyz=normal, w=v)
- Per-primitive: data (uvec4: task ID, face area, debug)

10. Vulkan Renderer Infrastructure

10.1 Pipeline Creation

- Auto-detect shader stage from file extension (.task, .mesh, .frag)
- pVertexInputState = nullptr, pInputAssemblyState = nullptr
- Dynamic rendering (VK_KHR_dynamic_rendering) instead of render passes
- Shader compilation: glslangValidator --target-env vulkan1.3

10.2 Required Extensions and Features

```
// Device extensions
VK_KHR_SWAPCHAIN_EXTENSION_NAME
VK_EXT_MESH_SHADER_EXTENSION_NAME

// Device features
VkPhysicalDeviceMeshShaderFeaturesEXT { .taskShader=TRUE, .meshShader=TRUE }
```

10.3 Frame Loop

```
beginFrame() -> beginRendering() ->
  bind parametric pipeline -> for each object: bind + dispatch
  bind HE pipeline -> for each base mesh: bind + dispatch
  bind pebble pipeline -> for each pebble: bind + dispatch
-> endRendering() -> renderUI() -> endFrame()
```

Each object's bindAndDispatch: bind HE descriptors (set 1), bind per-object descriptors (set 2), push constants (model matrix), call vkCmdDrawMeshTasksEXT(cmd, nbFaces + nbVertices, 1, 1).

11. Implementation Tasks (Ordered)

Phase 1: Vulkan Infrastructure (Weeks 1–2)

Task 1.1: Project Scaffold

- CMakeLists.txt with Vulkan, GLFW, GLM, Dear ImGui dependencies
- Shader compilation rules (glslangValidator with include directive support)
- Window creation, Vulkan instance (API 1.3), surface

Task 1.2: Device Setup

- Physical device selection with VK_EXT_mesh_shader support
- Logical device creation with mesh shader features enabled via pNext chain
- Queue families, command pool
- Print mesh shader hardware limits (maxMeshOutputVertices, maxMeshOutputPrimitives, maxTaskPayloadSize)

Task 1.3: Swap Chain and Rendering

- Swap chain creation, depth buffer (D32_SFLOAT)
- Dynamic rendering or render pass setup, frame synchronization
- Basic clear-screen render loop

Task 1.4: Descriptor Infrastructure

- Create 3 descriptor set layouts matching Section 3.3
- Descriptor pool, allocate descriptor sets
- Create and map uniform buffers for ViewUBO and GlobalShadingUBO

Task 1.5: Mesh Shader Pipeline Object

- Graphics pipeline with task + mesh + fragment stages, no vertex input
- Load vkCmdDrawMeshTasksEXT via vkGetDeviceProcAddr
- Test with hardcoded triangle output from mesh shader

Task 1.6: ImGui Integration

- ImGui Vulkan backend setup, basic UI with camera controls and FPS

Phase 2: Mesh Loading and GPU Upload (Weeks 2–3)

Task 2.1: OBJ Loader with N-gon Support

- Parse OBJ with arbitrary face vertex counts, compute per-face normals/centers/areas

Task 2.2: Half-Edge Construction

- Convert NgonData → HalfEdgeMesh (SoA layout), build twin map, compute vertexFaceIndices

Task 2.3: GPU Buffer Upload

- Upload each SoA array as separate SSBO, update HE descriptor set bindings

Task 2.4: Shared Shader Interface

- Create shaderInterface.h shared between C++ and GLSL with #ifdef __cplusplus guards

Phase 3: Core Resurfacing Pipeline (Weeks 3–4)

Task 3.1: Task Shader — Mapping Function F

- Dispatch per face+vertex, read position/normal from SSBO, populate payload, emit 1 mesh invocation

Task 3.2: Mesh Shader — Hardcoded Geometry

- Output single quad at payload position to validate task→mesh→fragment data path

Task 3.3: Parametric Surface Evaluation

- Implement parametricTorus in GLSL, evaluate 8×8 UV grid, apply scale/rotate/translate

Task 3.4: Multiple Parametric Types

- Add sphere, cone, cylinder, egg; ImGui control for type and radii

Task 3.5: Proper UV Grid Emission

- Grid vertices with quad→triangle pairs, per-vertex normals/UVs, per-primitive IDs

Phase 4: Amplification and LOD (Weeks 4–5)

Task 4.1: Amplification Function K

- getDeltaUV() for hardware-constrained UV tiles, 2D EmitMeshTasksEXT

Task 4.2: Variable Resolution

- Per-axis MN via UBO, handle edge tiles in mesh shader

Task 4.3: Pipeline Permutations (Optional)

- Multiple compiled variants (4×4 small, 8×8 large) for optimal GPU utilization

Task 4.4: Frustum Culling

- Project center to clip space, check bounds, set count=0 for culled

Task 4.5: Back-Face Culling

- Normal–view direction dot product with configurable threshold

Task 4.6: Screen-Space LOD

- Parametric bounding box → screen space projection → adaptive MN resolution

Phase 5: B-Spline Control Cages (Weeks 5–7)

Task 5.1: LUT Loader

- Load control cage OBJ (quad grid), sort by UV, flatten, upload as SSBO

Task 5.2: B-Spline Evaluation

- Cubic B-spline basis functions, 4×4 control point neighborhoods, cyclic boundaries

Task 5.3: Bezier Evaluation

- Bernstein basis with configurable degree (1–3)

Task 5.4: Control Cage LOD

- Bounding box from LUT extents, same screen-space LOD framework

Phase 6: Pebble Generation (Weeks 7–9)

Task 6.1: Pebble Pipeline

- Separate pipeline + PebbleUBO, one task per face only

Task 6.2: Pebble Task Shader

- Fetch face vertices to shared memory, compute patch count and subdivision level

Task 6.3: Pebble Control Cage Construction

- Extrude, project for roundness, subdivide once → 16 control points in shared memory

Task 6.4: Pebble Surface Evaluation

- Bicubic B-spline from procedural cage, procedural noise, random variation

Task 6.5: Pebble LOD

- Bounding box from extruded vertices, power-of-two subdivision levels

Phase 7: Control Maps and Polish (Weeks 9–10)

Task 7.1: Control Map Textures

- Sample color-coded texture in task shader, map to element types

Task 7.2: Base Mesh Rendering

- Simple mesh shader pipeline for wireframe/solid base mesh visualization

Task 7.3: Skeletal Animation (Optional)

- GLTF loader, skin transform in task shader before culling

Phase 8: Performance Analysis (Weeks 10–12)

Task 8.1: GPU Timing

- Vulkan timestamp queries, per-pipeline timing, ImGui display

Task 8.2: Benchmarking Suite

- Reproduce Table 1: frame times per surface type, with/without culling/LOD

Task 8.3: Memory Analysis

- VRAM usage comparison: mesh shader vs static vertex buffers

Task 8.4: Comparison Pipelines (Optional)

- Vertex pipeline and tessellation pipeline baselines for Figure 7 reproduction

12. Key Implementation Gotchas

1. pVertexInputState and pInputAssemblyState MUST be nullptr in the mesh shader pipeline. Setting them causes validation errors or crashes.

2. taskPayloadSharedEXT is the ONLY way to pass data from task → mesh. Payload size limited to ~16KB. Keep it lean.

3. Shader include directives require GL_GOOGLE_include_directive. Pass -I flag to glslangValidator.

4. gl_WorkGroupID in mesh shader: when EmitMeshTasksEXT emits (x, y, 1), mesh shader gets gl_WorkGroupID.x and .y for UV tile indices.

5. MAX_VERTICES and MAX_PRIMITIVES are compile-time constants. Cannot change at runtime. Use pipeline permutation strategy.

6. SoA half-edge layout uses arrays of typed buffers (heVec4Buffer[5], heIntBuffer[10]). Requires GL_EXT_scalar_block_layout and matching descriptor array counts.

7. Pebble mesh shader: control cage too large for payload. Must re-evaluate from SSBO in mesh shader, store in shared memory.

8. Push constants for model matrix (changes per draw call), not UBO.

9. Vulkan Y-flip: projection matrix needs proj[1][1] *= -1.

10. Subgroup operations (GL_KHR_shader_subgroup_arithmetic) enabled but used minimally. Useful for cooperative vertex emission.

13. File Structure (Reference Implementation)

```
resurfacing/
+-- CMakeLists.txt
+-- assets/
|   +-- demo/                    # Dragon model, textures, animations
|   +-- icosphere.obj            # Test mesh
|   +-- parametric_luts/         # Control cage OBJ files
+-- libs/                        # glfw, glm, imgui, stb, tinygltf
+-- shaders/
|   +-- shaderInterface.h        # Shared CPU/GPU: bindings, UBOs, getters
|   +-- common.glsl              # lerp, isVisible, rotation alignment
|   +-- lods.glsl                # LOD: bounding box -> screen space
|   +-- noise.glsl               # Procedural noise (Gabor, Perlin)
|   +-- shading.glsl             # Fragment shading
|   +-- stdPerVertexMesh.glsl    # Per-vertex/per-primitive output interface
|   +-- parametric/
|   |   +-- parametric.glsl      # Payload, constants, LOD, amplification
|   |   +-- parametricSurfaces.glsl  # Analytical surface equations
|   |   +-- parametricGrids.glsl # B-spline and Bezier evaluation
|   |   +-- parametric.task/.mesh/.frag
|   +-- pebbles/
|   |   +-- pebble.glsl/.task/.mesh/.frag
|   +-- halfEdges/
|       +-- halfEdge.mesh, halfedge.frag
+-- src/
    +-- main.cpp                 # App, init, render loop, UI
    +-- renderer.cpp/.hpp        # Vulkan wrapper
    +-- vkHelper.hpp             # Buffer, Texture, Pipeline types
    +-- HalfEdge.hpp             # Half-edge structure + conversion
    +-- AppRessources.cpp/.hpp   # Scene objects, buffer management
    +-- loaders/ObjLoader, GLTFLoader
```

14. Success Criteria

4. **Tori/spheres/cones** render correctly on loaded OBJ meshes (chain mail appearance)
5. **B-spline control cages** produce smooth subdivision surfaces (scale appearance)
6. **Pebbles** generate from face geometry with noise perturbation
7. **Culling** eliminates off-screen and back-facing elements
8. **LOD** varies resolution based on screen-space size
9. **Control maps** enable per-face element type selection
10. **Performance** within same order of magnitude as paper's results on RTX 3080
11. **Memory footprint** is constant regardless of subdivision level (geometry never stored)